CodeTEST™

Embedded Software Verification Tools

Applied
Microsystems
Corporation

User's Guide for Unix

Applied
Microsystems
Corporation

CodeTEST™ User's Guide for Unix®

September 1996

P/N 924-08000-02

Information in this document is subject to change without notice. Applied Microsystems Corporation reserves the right to make changes to improve the performance and usability of the products described herein.

CodeTEST tools patent pending.

CodeTEST software uses the EuroBridge Widget Set by VTT Information Technology.

Trademarks

CodeTEST is a trademark of Applied Microsystems Corporation.
Motorola is a registered trademark of Motorola, Inc.
AT is a trademark, and IBM is a registered trademark, of International Business Machines Corporation.
Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.
Sun, Sun Microsystems, Sun Workstation, Sun0S, Sun-4, and NFS are trademarks of Sun Microsystems, Inc.
SPARC is a registered trademark of SPARC International, Inc.
SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.
UNIX is a registered trademark of AT&T.

Preface

This manual provides operating instructions and usage guidelines for Applied Microsystems CodeTEST tools for embedded systems software development and testing.

Note: For additional important information, please refer to the release notes in the CodeTEST Online Help or in the README file in your CodeTEST installation directory.

Assumptions

All materials in the CodeTEST manual set assume that you are familiar with embedded software development practices and have a working knowledge of the Unix operating system and the C or C++ programming language. Also, it is assumed that you have a working knowledge of the target you will be testing and have access to processor-specific documentation from the processor manufacturer.

The CodeTEST manual set

The CodeTEST manual set includes:

- ❑ The *CodeTEST User's Guide* (this book) provides operating instructions and usage guidelines for the host application, the source code instrumenter, and related support files and utilities.
- ❑ The *CodeTEST Installation Guide* covers hardware and software installation procedures and guidelines.
- ❑ The *CodeTEST Online Help* gives quick reference and procedural information for the CodeTEST system.
- ❑ The *probe booklets* provide information specific to each version of the CodeTEST probe.

Notational Conventions

This manual uses the following notational conventions:

Element	Meaning
`monospace font`	Monospace (typewriter-like) font indicates commands that you must enter as shown. This font is also used to represent screen text and file contents.
`monospace italics`	Monospace italic font within a command indicates a variable for which you must substitute a value. (Within body text, italics are used for emphasis and for first occurrence of key terms or concepts.)
`[`*`option`*`]`	Square brackets enclose an optional item or list of optional items. Do not enter the brackets.
|	A vertical bar separates items in a list of mutually exclusive alternatives. Do not enter the vertical bar.
`{`*`opt1`*|*`opt2`*`}`	Curly braces enclose a list from which you must choose one item. Do not enter the braces.
`. . .`	Horizontal ellipsis indicates that you may repeat an argument zero or more times. Do not enter the ellipsis.
`<F1>`	Angle brackets signify a key on the keyboard. Do not enter the brackets.
other punctuation	Within command syntax, punctuation marks other than those defined above must be entered as shown.

Support Services

Applied Microsystems Corporation provides a full range of support services. New software is covered by a 90-day warranty that includes full applications phone support. Additional support agreements are available to extend the initial warranty and to provide additional services.

If you encounter trouble installing or using your CodeTEST system, consult your manuals to verify that you are following appropriate procedures. If a problem persists, call Applied Microsystems Customer Support. Customers outside the United States should contact their sales representative or local Applied Microsystems office.

Frequently asked questions

Before contacting Customer Support, please check the following list of frequently asked questions.

1. Probe connection errors can result from:
    - ❑ Trying to connect to a probe that is already in use.
    - ❑ Trying to connect to a probe for which you have no license.
    - ❑ Trying to connect to a networked device that is not a CodeTEST probe (e.g., a workstation or an emulator).
    - ❑ Trying to connect to a probe that is not correctly configured on your network, is not powered up, or is connected to a target that is not powered up. Refer to the *CodeTEST Installation Guide* and to your probe booklet for details about setting up your probe.
2. Licensing problems

    Licensing error messages may be caused by incorrect configuration of the FLEXlm license manager or problems with your license file. Refer to the installation instructions supplied with your CodeTEST software.

3. GUI appearance problems

   Unusual appearance of the CodeTEST host application user interface (e.g., unusual colors or labels with underbar instead of space between words) indicates a problem with the X11 application defaults files. See "X defaults" on page A-4.

4. IDB problems

   For CodeTEST to make accurate measurements, the host application must be configured to find the correct version of the IDB file (or set of compatible IDB files) for the target program. See "Target program configuration" on page 2-15 and "Compatible IDB Files" on page 6-23.

5. Errors finding source files

   For the host application to access the source files for your target program (for display in the source code viewer and in the Source view of trace) all source directory paths (or an ASCII file containing those paths) must be entered in the configuration dialog. See "Target program configuration" on page 2-15.

6. Loading and viewing data files

   To load a CodeTEST data file, the host application must be able to find the IDB for that data. The IDB for saved data is identified at the top of the data file itself, not in the configuration dialog. See "Saving and Loading Data" on page 2-19.

7. Instrumenter environment errors

   The environment must be correctly set up for your use of the instrumenter. In addition to creating a ctcc/ctc++ configuration file (described on page 5-13) this often depends upon having various other toolchain variables set (e.g., for an MRI compiler, the variable MRI_68K_INC may need to be set to point to -I include paths). Example configuration files are included in $AMC_HOME/bin.

Required Information

When you contact Customer Support, please have the following information available:

- Your support agreement number, if applicable.
- The version number of your CodeTEST software.
- The serial number of your probe (printed on a small sticker affixed to the bottom of the probe chassis).
- The version numbers of the probe's DRP and Controller firmware. This is displayed in the Status Window, which is accessible from the host application Tools menu. It can also be seen by running the ctupdate utility (refer to the software installation procedures before running ctupdate).
- The type of host workstation and operating system you are using.
- The exact sequence of operations or commands that will duplicate the problem.

Contact numbers

Applied Microsystems product support is available at the following numbers:

- 800-ASK-4AMC (800-275-4262)
- 206-882-2000 (from Washington state and Canada)
- World Wide Web: http://www.amc.com
- E-mail: support@amc.com

Contents

Preface

Chapter 1
Introduction

Chapter 2
Running CodeTEST

Chapter 3

The Continuous Mode Toolset

Chapter 4
The Trace Tool

Chapter 5
The CodeTEST Compiler Driver

Chapter 6
The Instrumenter

Chapter 7
CodeTEST Memory Functions

Chapter 8
Using CodeTEST with an RTOS

Chapter 1
Introduction

CodeTEST System Overview

Applied Microsystems CodeTEST tools provide the first software verification solution designed specifically for embedded software engineers—an easy-to-use suite of shared network resources for your entire development and testing team.

CodeTEST has hardware and software components:

- The *instrumenter*—a set of software utilities that prepare your target program for testing.
- The *probe*—a hardware device that connects to your network and to your target hardware, to monitor your program as it runs in-circuit and transmit accumulated measurement data to a host workstation.
- The *host application*—a set of software tools that run on a workstation and can measure performance, coverage, memory allocation, and trace.

The instrumenter

The CodeTEST source code instrumenter prepares your program for in-circuit testing by filtering your source files to insert test point instructions, or *tags,* into your code. These tags enable the probe to identify and track various program activities.

The instrumenter also creates and maintains an Instrumentation Database (IDB) which CodeTEST uses for display of program symbols. (CodeTEST does not use the symbol file created by your compiler.)

The instrumenter supports many implementations of the C and C++ languages, in compliance with the ANSI C, K&R C, and emerging ANSI C++ standards.

The instrumenter comprises several interrelated utilities:

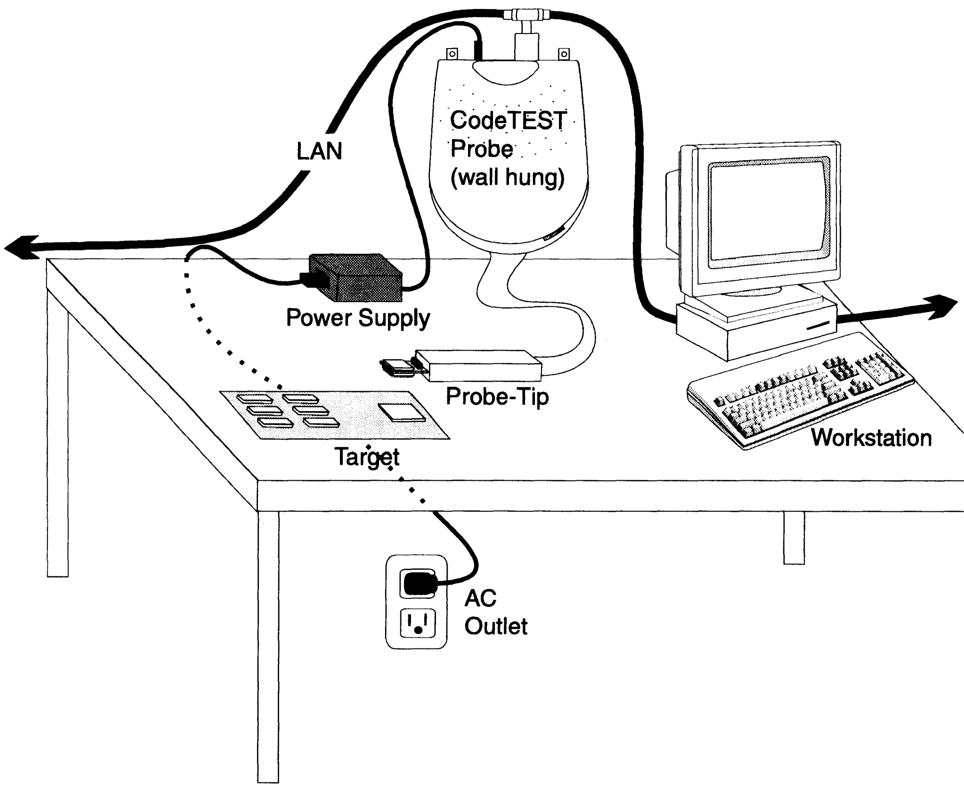
- *amctag* is the program that actually inserts tags into your source code, creates and maintains the IDB, and optionally performs C or C++ preprocessing.
- *ctcc* and *ctc++* are compiler drivers for C and C++ compilers respectively. They behave much like other compiler drivers (gcc for example, the driver for GNU C). But in addition to driving the build procedure, ctcc and ctc++ incorporate instrumentation by calling amctag and linking in various CodeTEST support files.
- The ctcc/ctc++ configuration file defines a set of variables that adapt the instrumenter to your compiler, environment, and particular build methods.

The probe

The probetip connects to the microprocessor socket on your target board, either directly or by means of an adapter. The network interface on the probe's main chassis connects directly to your LAN, so access to CodeTEST connected to your target is sequentially available to multiple users.

The probe passively monitors two specific *tag port* addresses, a control port and a data port, which you assign in the target memory space. As the target program runs, the tags the instrumenter placed in your code write values to the monitored addresses. The probe captures the stream of tag values, performs some initial processing with its internal data reduction processor, and sends measurement data via the network to a host workstation.

Note: Several versions of the CodeTEST probe are available, to support many different processors from a variety of manufacturers. Check with your sales representative for current information.



The host application

The host application's four software tools are sold as separate packages, each of which can operate independently.

The three *Continuous mode* tools (Performance, Memory, and Coverage) can also operate simultaneously and perform several kinds of measurements in a single test run. You need only instrument your code at an appropriate level and select a license for each tool you want to use during your session. Continuous mode measurement setup allows you to qualify your performance and memory measurements to a specific RTOS task.

Trace is an alternative mode that operates independently. The Trace tool provides versatile triggering and context qualification features, as well as display options and search capabilities to give you excellent control over your view into the target program's execution behavior.

When your workstation receives data from the probe, the host application retrieves symbol information from the IDB, then presents measurement results in the various data views. In Continuous mode, you can observe the periodic data update until you decide to stop the measurement. In Trace mode, once the trigger event occurs and the trace buffer fills to the selected depth, the probe stops automatically and the results are presented in the Trace window. Though you cannot run Trace and Continuous mode measurements simultaneously, you can display Trace and Continuous mode views side-by-side and toggle between modes without losing data.

Regardless of whether your program is executed from cache, or dynamically relocated by the operating system, CodeTEST produces accurate and reliable measurements, overcoming the potential confusion surrounding instruction pre-fetch and caching.

RTOS support

For targets that employ a real-time operating system (RTOS) CodeTEST provides task-based triggering and measurement qualification, and a number of features for monitoring the target program on a task-by-task basis.

CodeTEST supports connection to custom RTOS systems and to these commercial RTOS products:

- ❑ pSOS from Integrated Systems, Inc.
- ❑ VxWorks from Wind River Systems, Inc.
- ❑ VRTX from Microtech Research, Inc.

What will it do for me?

Software testing strategies generally focus on three recognized phases of system development: *unit* testing, *integration* testing, and *system* testing. Each of the CodeTEST tools is useful during all development and testing phases, though the emphasis of their usage will vary from one phase to the next.

Development engineers will be most interested in using CodeTEST to debug and unit test relatively small blocks of code. The Trace and Memory tools will be of immediate interest at this stage. Test engineers on the other hand, who are not so concerned with code behavior per se as with determining when sufficient testing has been done and requirements specifications have been met, will likely focus first on the Coverage and Performance tools. But to get maximum benefit, try using each CodeTEST tool at every stage of development and testing.

The Continuous mode toolset

The Coverage tool

With the Coverage tool, you can pinpoint untested code and identify the additional test cases necessary to reduce the chances of passing undetected defects to the next stage.

❑ View a function-by-function display of the target program's branch coverage (i.e., the percentage of basic blocks that have executed).

❑ Pop up the right-mouse-button menu to display:

  - source code for any function in the target program, with executed lines highlighted.

  - a summary of the Continuous mode data collected for any function.

- ❑ Monitor test progress with the Coverage Trend view—a dynamic XY graph that tracks coverage over time.
- ❑ Check the Coverage Summary view—a bar graph that shows the overall level of coverage achieved.
- ❑ Merge coverage data from multiple measurements to see the composite coverage achieved by a test suite.

The Memory tool

Use the Memory tool to examine dynamic memory management behavior proactively and preventively, to root out latent problems before symptoms develop.

- ❑ Monitor the allocation and deallocation performed by each line of your code that calls a memory routine.
- ❑ Pop up the right-mouse-button menu to display:
  - source code with memory calls highlighted.
  - a summary of the collected Continuous mode data for any function that has called a memory routine.
- ❑ Review a log of detailed messages and diagnostics for the memory error conditions CodeTEST detects.
- ❑ Qualify memory measurements to a specific RTOS task.

The Performance tool

Use the Performance tool to run timing benchmark tests and check for call-pair thrashing or other algorithmic problems, in the your entire program or a single code module.

- ❑ View your target program's execution timing and counts on a task-by-task basis.
- ❑ View timing and counts function-by-function.
- ❑ View the call-pair relationships among functions.

- ❑ Pop up the right-mouse-button menu to display:

    - source code for any function that has executed (in the Function Performance table) or any *calling* or *called* function (in the Call Linkage table).

    - a summary of the collected Continuous mode data for any function listed in the performance tables.

- ❑ Qualify your performance measurements to a specific RTOS task.

The Trace tool

Interactively trace target program execution and view the results at several levels of detail.

- ❑ Take quick snapshots of up to 4K or 40K events, or capture a very deep trace of up to 400K events.

- ❑ Trigger on a specific function entry or exit, an RTOS task entry, exit, creation or deletion, a memory allocation, deallocation or error, an AMCPrintf, AMCPuts, or AMCUserTag call, or *any* event (i.e., the first tag the probe receives).

- ❑ Run Trace without a trigger and manually halt the probe, or use the *no trigger* feature to trace events leading up to a target program crash.

- ❑ Position the trigger at the beginning, middle, or end of the trace buffer.

- ❑ Qualify the trigger context to a specific task, function or function calling sequence, or task *and* function or function calling sequence.

- ❑ Qualify the storage context to a specific task, function or function calling sequence, or task *and* function or function calling sequence.

❑ Place AMCPrintf or AMCPuts calls in your code to see printf-style or puts-style strings in trace, or place calls to AMCUserTag to manually flag areas of interest.

❑ Display a High Level view of the trace buffer, showing only RTOS task creation, entry, exit, and deletion events, and function entry and exit points.

❑ Switch to the Control Flow view to add executed branch points and memory management events to the display.

❑ Switch to the Source view to see each executed line of source code.

❑ Use the Trace Find utility to search the buffer for a variety of event types.

❑ Expand loops to show each iteration as a separate event line, or collapse loops to a single event line with a numeric execution count.

❑ Display elapsed time from the start of the trace to each event, or time intervals between events.

❑ Select any two events and display the elapsed time between them.

CodeTEST utilities

❑ Save measurement data for later review, or continuation of Continuous mode measurements.

❑ Record CodeTEST command macros.

❑ Find, filter, and sort Continuous mode data based on the contents of any table column.

❑ Print or export data, with your filters and sorting reflected in the output.

❑ View or save a log of any CodeTEST host application error messages generated during your session.

How do I use it?

This section briefly overviews the ways you can use CodeTEST. Guidelines for actually getting CodeTEST up and running are covered under "Getting Started" on page 2-2.

RTOS instrumentation

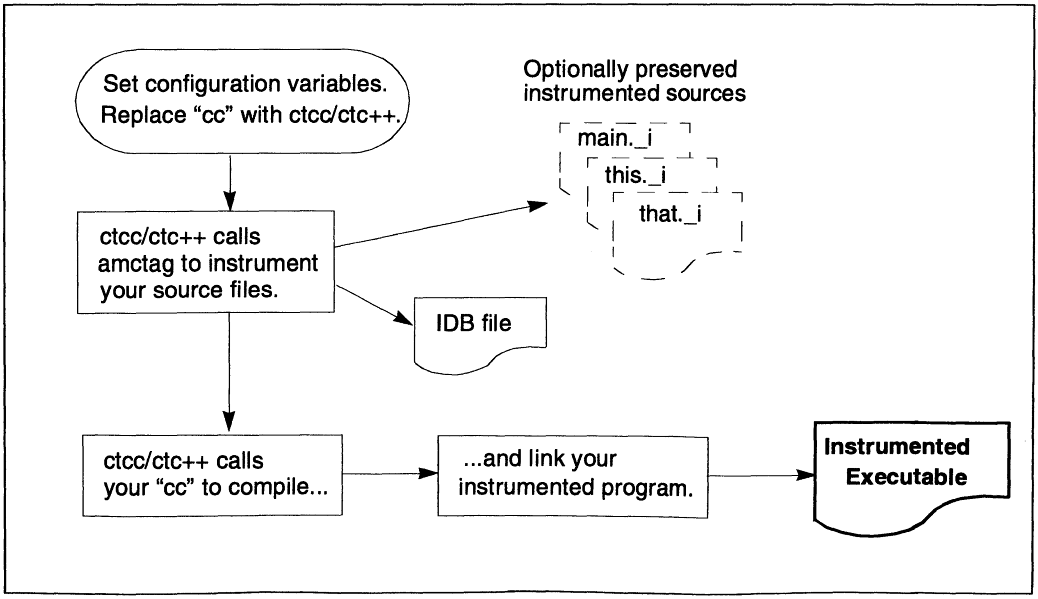
If your target system uses a custom or commercial RTOS, you will need to add a small amount of instrumentation to track each task creation, entry and deletion.

Target program instrumentation

There are a number of ways you can incorporate the instrumentation step into your target program build procedure.

Using a CodeTEST compiler driver

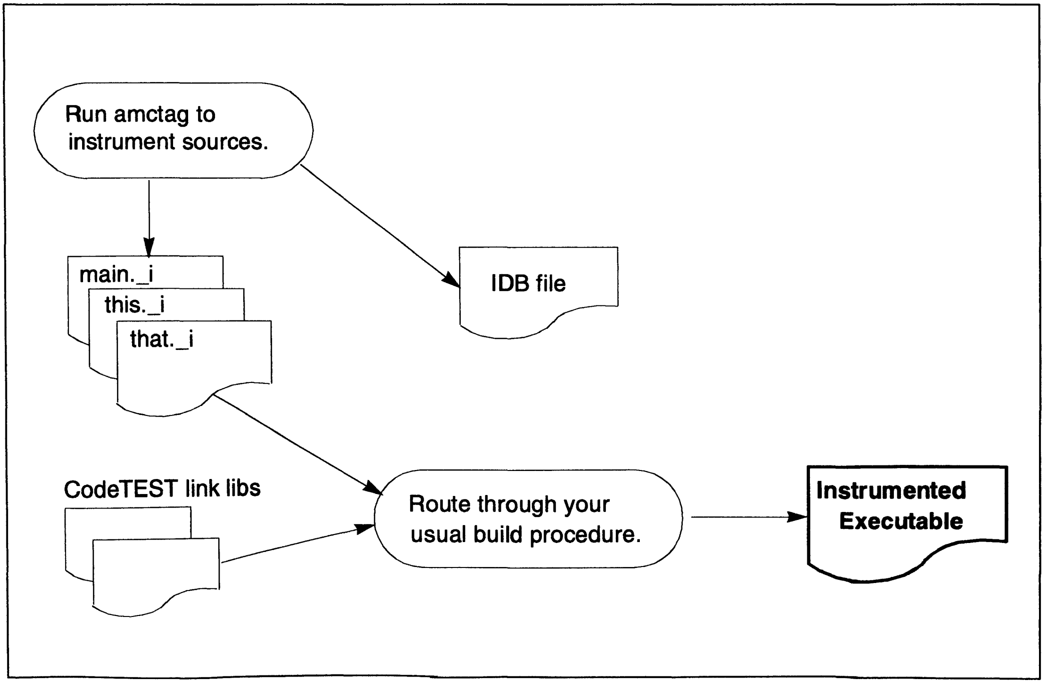
Generally the simplest approach to instrumenting your source code is to use one of the provided compiler drivers (ctcc for C compilers or ctc++ for C++ compilers).


Set configuration variables.
Replace "cc" with ctcc/ctc++.
Optionally preserved instrumented sources
main._i
this._i
that._i
ctcc/ctc++ calls amctag to instrument your source files.
IDB file
ctcc/ctc++ calls your "cc" to compile...
...and link your instrumented program.
Instrumented Executable

By defining a set of variables to adapt the instrumenter to your environment, and substituting ctcc or ctc++ wherever “cc” appears in your makefile or build script, you can automatically instrument your sources with each compilation, link in the necessary CodeTEST libraries, and assign memory locations to the two CodeTEST port addresses. This approach is designed to ease instrumentation into your environment with minimal change to existing procedures.

Invoking amctag directly

An alternative to using the ctcc/ctc++ compiler driver is to invoke amctag explicitly to instrument a body of sources, then follow your usual build procedure to compile and link the instrumented code. If you opt to use this method, you will also need to explicitly build and link the supplied CodeTEST library routines with your target program.


Run amctag to instrument sources.
main._i
this._i
that._i
IDB file
CodeTEST link libs
Route through your usual build procedure.
Instrumented Executable

Interactive operation

To run CodeTEST interactively:

❶ Load your instrumented program into the target system using your usual method. (CodeTEST does not provide this capability.)

❷ Start the CodeTEST host application and set the configuration options for your session.

❸ Set up your Continuous mode or Trace measurement (optional) or use the tools with their default setup.

❹ Start the probe to begin a measurement.

❺ Use your run control device or stimulus to manipulate the target through some test scenario.

❻ In Continuous mode, the probe sends measurement data to your host workstation at an interval you specify. The measurement runs until you manually halt the probe.

In Trace mode, the probe halts automatically, when the trigger event occurs and the buffer fills.

❼ View the results using the various display controls, find, filter, and sort features, etc.

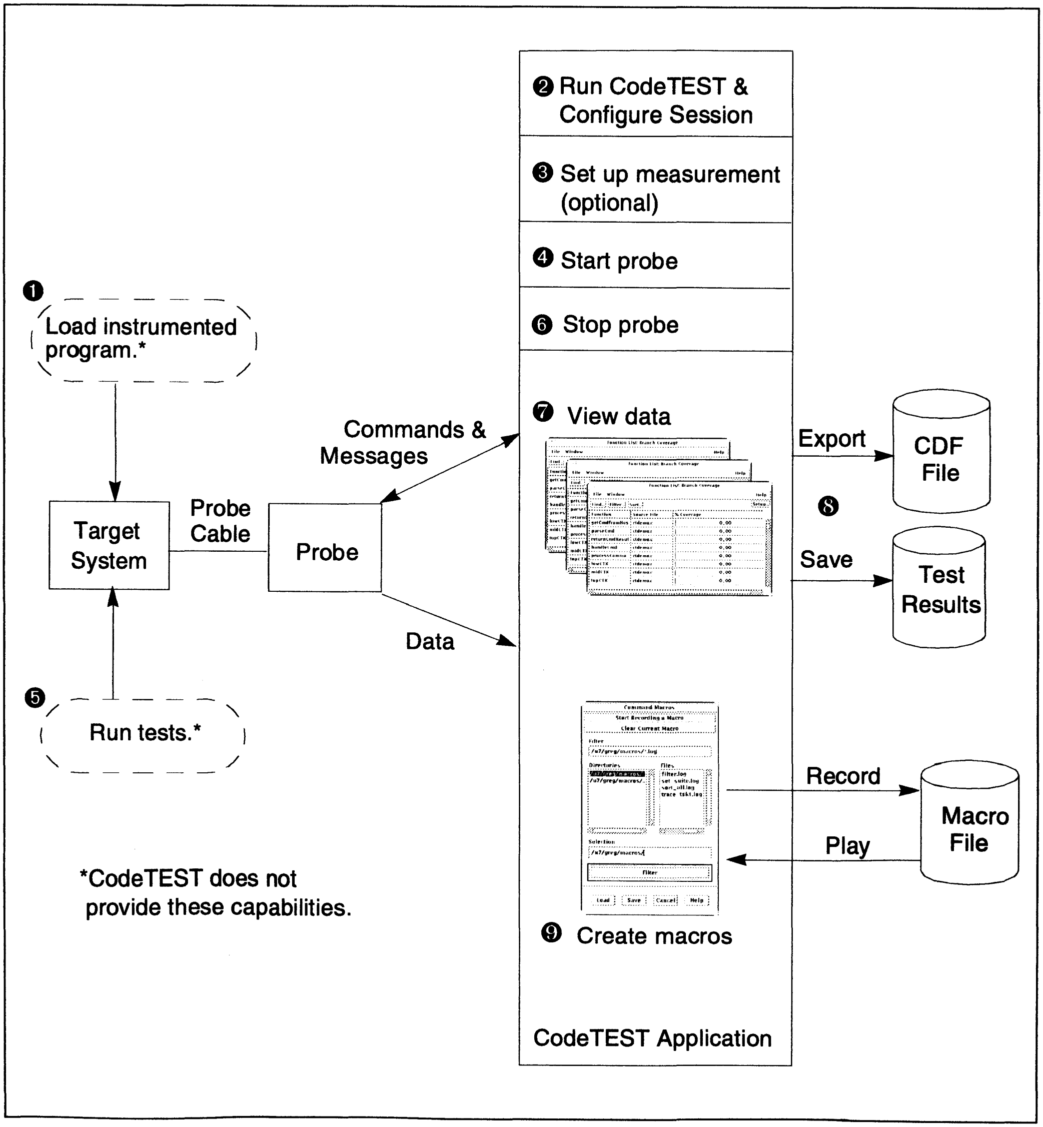
❽ Once you've captured data, you may want to:

- ❑ Save the results for later review.
- ❑ Print all or part of the data.
- ❑ Export data to a spreadsheet for post-processing, or a publishing program for producing reports.

CodeTEST macros

❾ Use the CodeTEST macro utility to streamline testing:

- ❑ Record commands to set up measurements, load data, etc. Most host application procedures can be recorded.
- ❑ Execute your macros from the Macro utility itself, or supply the macro file name as a command line argument when starting the host application.



*CodeTEST does not provide these capabilities.

Chapter 2

Running CodeTEST

Saving and Loading Data 2-19

Getting Started

The approach outlined in this section will help you get CodeTEST up and running, making actual measurements on your own code as expeditiously as possible. Guidelines are given to help you identify the individuals within your organization best suited to handle each task, and pointers are provided to the appropriate documentation for details. Actual startup doesn't usually take long, and most of the effort is a one-time investment that will make day-to-day CodeTEST operation easy for your whole team.

Note: To succeed with CodeTEST, individuals knowledgeable about your target code must be available to assist in the preparation. When the right people participate the process goes quite smoothly; when they don't time and resources may be wasted.

Preliminaries

1. Install CodeTEST software

If your CodeTEST software is not already installed, install it now. Most sites require that this be done by a System Administrator. For details, refer to the *CodeTEST Installation Guide*. Also check the contents of the README file placed in your installation directory by the CodeTEST installation script.

Note: To begin preparing for testing with CodeTEST it is not necessary to have your probe installed. When you are ready to begin using your probe, your System Administrator will need to provide an IP address and netmask and the correct type of ethernet transceiver (see step 5 below).

2. Select some code to test

To get started, select a self-contained subsystem, ideally about 5–10K lines. This code must currently compile, link, and run. It's important that the person assigned to deal with this code through the startup process be able to build, load and run the program in the target hardware.

RTOS connection

If your target uses a real-time operating system (RTOS) follow these steps to begin making CodeTEST measurements of basic task activity. If your target system does not use an RTOS, skip ahead to "Probe installation".

Note: Unless your program is strictly single-threaded and doesn't use an RTOS, you cannot skip these steps and expect CodeTEST to work properly. (In single-threaded operation, interrupts are okay provided they return whence they came.)

3. Prepare your RTOS

To make accurate performance, trace, or memory allocation measurements in a multi-tasking environment, CodeTEST must track the program's RTOS task context. For this to happen, the RTOS must communicate to the CodeTEST probe each time a task is created, deleted, or a swap occurs. See Chapter 8, *Using CodeTEST with an RTOS* for details about preparing a commercial or custom RTOS.

Commercial RTOS

CodeTEST provides simple callout functions for pSOS, VRTX, and VxWorks, which connect to the standard vendor-supplied hooks and emit the necessary information for tracking task context. The person responsible for dealing with this must be able to compile the appropriate (supplied) file for your target processor and link this code with your RTOS/application. This entails modifying your RTOS configuration file so the CodeTEST callouts are executed with each task creation, deletion, or switch.

Custom RTOS or other commercial RTOS

As in the case of a commercial RTOS, the person assigned to this task needs to get the RTOS to "tell" the CodeTEST probe every time a task creation, deletion, or swap occurs. For a custom RTOS this individual must identify in the OS code the location(s) where these events take place, then add code that emits the required "tags" to CodeTEST, and then rebuild the RTOS with these changes. Each callout will require perhaps a half-dozen lines of code.

4. **Link the RTOS and locate CodeTEST ports**

   Once the RTOS callouts are tagged, the RTOS must be linked and loaded into the target system. This process also involves locating two variables ("ports") that the CodeTEST instrumentation will write to as the target program runs. The person who does this will need to select two consecutive non-cached addresses to serve as the CodeTEST ports and locate these anywhere in the processor's address space, beginning on a 0x0 address boundary. This requires knowledge of the target system's memory map, including chip selects, MCONs, or other processor-specific memory configuration information. Actual location of the ports is accomplished with your linker. See "Assigning Addresses for the Tag Ports" on page 5-18.

5. **Run your instrumented RTOS and application in-circuit**

   Run the application in your target system and verify that everything still works as it did before. The person who does this must be able to download and run the build in-circuit, should be experienced in debugging embedded code, and should have tools available for debugging if needed.

Note: It's a good idea to use a software debugger, logic analyzer or emulator to verify that RTOS tags are emitted correctly before proceeding.

Probe installation

If your CodeTEST probe is not already installed on your network and connected to your target hardware, install it now. See the *CodeTEST Installation Guide* for details.

6. Install the probe on the network

Your System Administrator needs to assign the probe an IP address and netmask, update the hosts and ethers files, and provide the correct transceiver type (twisted pair or coaxial cable) for connecting the probe to ethernet.

7. Connect the probe to target hardware

The person assigned to connect the probe to the target system should first make sure the target processor is a footprint CodeTEST supports directly (PGA package) or via an adaptor. If an adapter is required, it should be installed by the time you expect to start using CodeTEST. In some cases it may be necessary to solder an adaptor to the target board. If this is the case, verify that the target system works properly by installing the processor in the adaptor and running your program as usual before connecting the CodeTEST probe.

RTOS measurements

If your target system uses an RTOS, you are now ready to begin making some basic CodeTEST measurements. With your “instrumented” RTOS and your uninstrumented application program, you can make two kinds of measurements—task performance timing and counts, and task execution traces.

If your target does not use an RTOS, skip ahead to “Complete implementation.”

8. Start up and configure CodeTEST

Start the host application and configure CodeTEST for your probe, your target program, etc. See “Running the Host Application” on page 2-8 and “Configuring a Session” on page 2-13 of this manual.

Note: To configure the probe for your target processor, you will need to run a utility to generate a configuration file that will enable the probe to monitor the two port addresses you assigned at link time. The person assigned to this task will need all of the relevant memory map information. Refer to the booklet supplied with your probe.

9. Measure task performance, trace task execution

The Task Performance view will show you a complete measurement of all of the executing tasks in your system, including the task name, number of instances, number of entries, min/max/avg execution *time slice*, and cumulative time spent in the task. See "Making a measurement" on page 3-2 and "Task performance" on page 3-13.

Running CodeTEST Trace and displaying the High Level view of the trace buffer will show the execution sequence of tasks and the amount of time spent in each time slice. See "Making a trace measurement" on page 4-2 and "High Level view" on page 4-10.

Complete implementation

You can now turn your attention to preparing your application code for the other types of CodeTEST measurements.

10. Instrument your selected source code

The person responsible for instrumenting the code that was selected at the beginning of this procedure must be able to build the program, either via a makefile or by manually compiling and linking the code.

If using a makefile, the normal procedure is to replace your "cc" command with CodeTEST's compiler driver (*ctcc* or *ctc++*) which in turn calls your cc. By default, the ctcc/ctc++ driver first invokes your preprocessor, then the CodeTEST instrumenter, and then your compiler.

If you don't want to use a makefile, you don't have to. You can invoke the instrumenter directly, as you would a compiler or any other software tool. For a small test program, this may be easier than modifying an existing makefile. See "Approaches to instrumentation" on page 5-3 for more information.

Note: Switches are available for controlling the level of instrumentation, depending on the kinds of measurements you want to make. By default, your code is instrumented for performance, coverage, and trace only. To instrument for memory allocation measurements, you must supply the *-Xtag-allocator* instrumenter switch, in addition to performing the following step. See "Instrumenter Options" on page 6-16.

11. Prepare for memory allocation measurements

To prepare your program for memory allocation measurements, one additional step is necessary. CodeTEST provides a set of special memory management routines that correspond to the standard C and C++ allocation calls (malloc, free, delete, etc.). Replace your current library with the CodeTEST version of these routines. The person assigned to do this must modify a few lines of source to be consistent with your system, then compile these routines for your target processor and link them in with your application. See Chapter 7, *CodeTEST Memory Functions*.

12. Test your software with CodeTEST

You can now make performance, coverage, memory allocation, and trace measurements. Just download your instrumented application (linked in with your instrumented RTOS, if applicable) and run the program in-circuit. Start the CodeTEST host application and configure your session as explained in the following sections and you're off and running!

Running the Host Application

You can run the CodeTEST host application to load and view saved data without having a probe installed. To acquire new data from your target system, a probe must be correctly installed on your network and connected to the target hardware.

Note: Refer to the *CodeTEST Installation Guide* and the booklet supplied with your probe for installation and configuration procedures. See Appendix A of this manual for a quick-reference guide to the system-wide configuration requirements for using CodeTEST.

Syntax

The host application command syntax:

```
CodeTEST [-configfile name] [-datafile name]
[-macrofile name] [-version]
```

Options

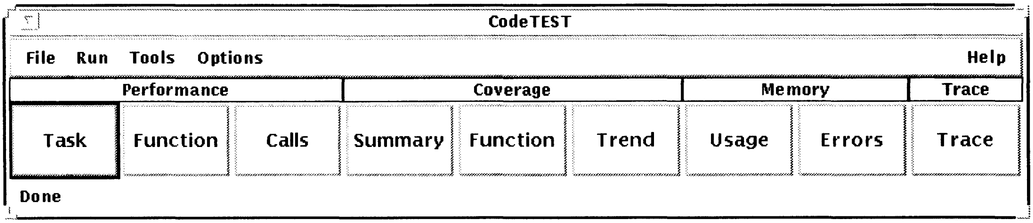
`-configfile` *name*	Starts the host application with a predefined configuration. See "Configuring a Session" on page 2-13 and "Configuration files" on page 2-18.
`-datafile` *name*	Loads a data file upon starting the host application. See "Saving and Loading Data" on page 2-19.
`-macrofile` *name*	Executes a predefined macro upon starting the host application. See "Creating a macro" on page 2-21.
`-version`	Displays the version number of the CodeTEST software.

CodeTEST Windows

CodeTEST uses standard X/Motif-style windows, with pull-down command menus and point-and-click tool buttons.

Toolbar

The CodeTEST toolbar provides top-level program controls as well as access to data views and various utilities. You will generally want to keep the toolbar displayed throughout your operating session.



Done

The status line at the bottom of the toolbar displays self-explanatory operational and error messages. To see a listing of any error messages generated during your session, select Error Log from the Tools menu. Appendix B provides troubleshooting information for the CodeTEST error messages.

Click the tool buttons to display the various data views. Each view is presented in a free-floating window with its own command menus and button controls.

Note: To set preferences such as default window size, placement, colors, etc., you can copy the CodeTEST application defaults file from $AMC_HOME/lib/X11/app-defaults to your home directory. Then edit your copy of that file and set the XFILESEARCHPATH environment variable to point to it. Refer to your X11 online man pages for information about the X resource options.

Toolbar Command Summary

Command	Avail.	Function
File Menu		
Load Data... [Ctrl]-[L]	All modes	Displays the dialog for loading a CodeTEST measurement data file. See "Saving and Loading Data" on page 2-19.
Save Data... [Ctrl]-[S]	All modes	Displays the dialog for saving the current measurement data. All (unfiltered) Continuous mode data and all Trace data is saved in a single file. The host application must be connected to a probe when you save a data file. See "Saving and Loading Data" on page 2-19.
Load IDB... [Ctrl]-[I]	All modes	Displays the dialog for loading an IDB file. This command allows you to load an IDB without affecting anything else in the configuration (e.g., probe communications).
Merge Coverage Data... [Ctrl]-[M]	Coverage license	Displays the dialog for merging coverage data acquired from multiple measurements. See "Merging coverage data" on page 3-12.
Exit [Ctrl]-[X]	All modes	Exits the CodeTEST host application.
Run Menu		
Start Probe	Cont. mode	Clears any Continuous mode data that has been acquired, then begins a new Continuous mode measurement. The kind of data that will be acquired depends upon the level of instrumentation in the target program and the license(s) you have chosen. (To start a Trace measurement, use the Start button on the Trace window.)
Stop Probe	Cont. mode	Halts Continuous mode data acquisition . (To stop a Trace measurement, use the Stop button on the Trace window.)
Continue	Cont. mode	Resumes a Continuous mode measurement without clearing existing data.This command can be used to resume the current measurement, or to add new data to a file that has been saved and then reloaded into the host application.
Set Continuous Mode [Ctrl]-[C]	Trace mode	Switches from Trace mode to Continuous mode. Any data currently in the trace buffer is retained.

Toolbar Command Summary (cont'd)

Command	Avail.	Function
Set Trace Mode [Ctrl]-[T]	Cont. mode	Switches from Continuous mode to Trace mode. Any existing Continuous mode data is retained.
Reset Probe Connection	All modes	Re-establishes the network communication channel with the probe.
Tools Menu		
Macros... [Ctrl]-[M]	All modes	Displays the dialog for recording, saving and executing a CodeTEST macro. See "Macro dialog" on page 2-20.
Error Log...	All modes	Displays a log of any error messages generated.
Status Window...	All modes	Displays information about your probe (including firmware version, total measurement time, etc.). Also shows IDB and RTOS map files in use, and licensing information.
Probe configuration utilities	All modes	All installed probe configuration utilities are listed in this menu. Refer to your probe booklet for information about using the appropriate utility to generate a binary configuration file for your probe.
Options Menu		
Edit Options...	All modes	Displays the dialog for configuring CodeTEST. See "Configuring a Session" on page 2-13.
Load Options File...	All modes	Displays the dialog for loading a saved configuration file. See "File load dialogs" on page 2-27.
Save Options File...	All modes	Displays the dialog for saving the current configuration. See "File save dialogs" on page 2-26.
Help Menu		
Help on Window...	All modes	Displays the help topic for the current window.
Contents...	All modes	Displays the help Contents page.
Search...	All modes	Displays the help Keyword Search dialog.

Toolbar Command Summary (cont'd)

Command	Avail.	Function
How to Use Help...	All modes	Displays help on how to use the help viewer.
About CodeTEST...	All modes	Displays the CodeTEST version number.
Button Controls		
Performance: Task	Perform. license	Displays the RTOS Task Performance view. When you acquire task performance data from the probe or load it from a file, it is displayed here. See "Task performance" on page 3-13.
Performance: Function	Perform. license	Displays the Function Performance view. When you acquire function performance data from the probe or load it from a file, it is displayed here. See "Function performance" on page 3-15.
Performance: Calls	Perform. license	Displays the Call Linkage view. When you acquire call pair data from the probe or load it from a file, it is displayed here. See "Call linkage" on page 3-17.
Coverage: Summary	Coverage license	Displays a bar graph that summarizes the coverage achieved during a measurement. See "Coverage summary" on page 3-11.
Coverage: Function	Coverage license	Displays the Branch Coverage view. When you acquire coverage data from the probe or load it from a file, it is displayed here. See "Branch coverage" on page 3-6.
Coverage: Trend	Coverage license	Displays a trend graph, which plots the progress of a coverage measurement as it runs. See "Coverage trend" on page 3-10.
Memory: Usage	Memory license	Displays the Memory Allocation view. When you acquire memory allocation data from the probe or load it from a file, it is displayed here. See "Memory allocation" on page 3-19.
Memory: Errors	Memory license	Displays the Memory Error log, for viewing or saving a list of memory errors generated. See "Memory error log" on page 3-22.
Trace	Trace license	Displays the Trace window. When you acquire trace data from the probe or load it from a file, it is displayed here. See "The Trace Window" on page 4-10.

Configuring a Session

CodeTEST requires certain basic information, such as the probe's network ID, paths to your source files, etc., which you enter in the Configuration Options dialog. Once you have entered the required information, you may want to save your configuration for use in future sessions.

Configuration dialog

To view or edit the configuration at any time during your session, select Edit Options from the toolbar's Options menu.

Note: Some configuration changes will clear existing measurement data from memory. Before you change the configuration during a session, first save any collected measurement data that you wish to keep.

Configuration categories

The Configuration Options dialog is partitioned into pages for the following information categories:

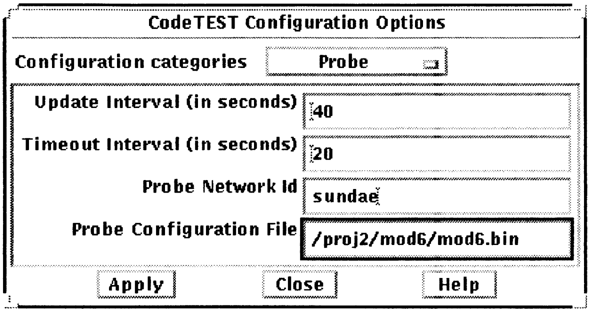
- ❑ Probe configuration
- ❑ Target program configuration
- ❑ License options

To display the page you want, select a category from the categories menu. To put your configuration into effect, click the Apply button.

Note: Before you can configure your session to access a probe, you need to generate a probe configuration file. See your probe booklet for details.

Probe configuration

To define the probe's operating variables, select Probe from the configuration categories menu, then enter the information as described below.



CodeTEST Configuration Options

Configuration categories: Probe

Update Interval (in seconds): 40

Timeout Interval (in seconds): 20

Probe Network Id: sundae

Probe Configuration File: /proj2/mod6/mod6.bin

Update interval

Enter the interval for updating the host application data views with new data from the probe. If the host application receives data faster than it can be processed, this interval will be adjusted automatically.

Timeout interval

Enter the interval after which CodeTEST times out if no response is received from the probe.

Probe network ID

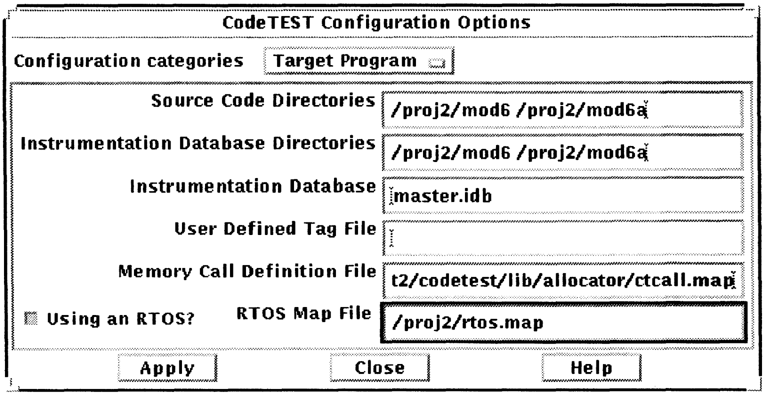
Enter the host name of the probe you will be using. An entry in this field is required. There is no default. (The host name is assigned during installation. See your System Administrator if this has not been done.)

Probe configuration file

Enter the name of a binary (.bin) configuration file generated by the configuration utility for your probe. Refer to your probe booklet for information about using the utility. To download a new configuration file to the probe, stop the probe before clicking Apply.

Target program configuration

To configure CodeTEST to monitor your target program, select Target Program from the Configuration Categories menu, then enter values in the fields described below.



CodeTEST Configuration Options

Configuration categories: Target Program

Source Code Directories: /proj2/mod6 /proj2/mod6a

Instrumentation Database Directories: /proj2/mod6 /proj2/mod6a

Instrumentation Database: master.idb

User Defined Tag File:

Memory Call Definition File: t2/codetest/lib/allocator/ctcall.map

▨ Using an RTOS?

RTOS Map File: /proj2/rtos.map

Source code directories

Enter the full absolute path (beginning with a slash character) to the source files for your target program. This can be a single directory path, multiple directory paths separated by spaces, or the name of an ASCII text file containing a list of directory paths. An entry in this field is required. There is no default.

IDB path and file name

For CodeTEST to produce accurate results, the instrumentation database (IDB) you specify must match the instrumented target program (i.e., must be produced during the same instrumentation).

Instrumentation database directories

Enter the full absolute path or a partial path to the IDB, beginning with a slash (/) character. This can be a single directory path, multiple directory paths separated by spaces, or the name of an ASCII file containing a list of directory paths. CodeTEST will search the path or paths for the file you specify in the *Instrumentation Database* field below.

No entry in this field is required if you choose to specify the full path and file name in the field below.

Instrumentation database

Specify the IDB file for your target program. This can be a single IDB file created by the instrumenter, or an ASCII text file containing a list of compatible IDB file names (see "Compatible IDB Files" on page 6-23).

The entry in this field can be a full absolute path and file name, a relative path and file name, or simply a file name. When searching for the IDB, CodeTEST will append the entry in this field to the entry in the Instrumentation Database Directories field above (if any). Only the first file found will be used.

Note: The entry in this field is written at the top of each saved data file. See "Saving and Loading Data" on page 2-19.

User defined tag file

If you place user defined tags in your target code (i.e., calls to the AMCUserTag function) you can create an optional map file to associate each call's numeric argument with a text string to be displayed in trace. See "User Defined Tags" on page 4-20.

Memory call definition file

Enter the name of the Memory Call Definition file to be used for your target program. The default version of this file is $AMC_HOME/lib/allocator/ctcall.map. See "Memory Call Definition file" on page 7-4.

Using an RTOS?

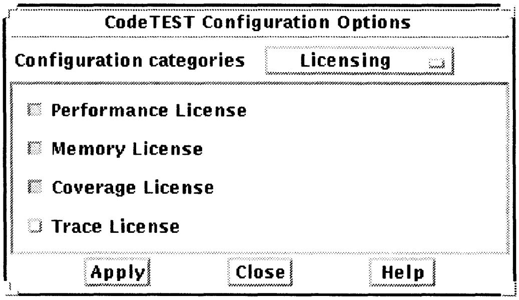
Select this box if your target system uses a custom or commercial real-time operating system (RTOS) . See Chapter 8, *Using CodeTEST with an RTOS* for information about preparing your RTOS for use with CodeTEST.

RTOS map file

If your target system uses an RTOS, you may want to create an RTOS map file to define a text string to represent each task in the CodeTEST user interface. See “Creating an RTOS Map File” on page 8-9.

License options

To select one or more CodeTEST licenses, select Licensing from the Configuration Categories list box, then make your selection from the list of options. To make new measurements, you must select a license for each tool you plan to use.



CodeTEST Configuration Options

Configuration categories: Licensing

■ Performance License

■ Memory License

■ Coverage License

□ Trace License

To load and view saved data that was acquired with any or all of the CodeTEST tools, you need only one license of any type. If you load a data file without first selecting a license, CodeTEST will automatically determine what licenses are available and select one for you.

Note: CodeTEST is a FLEXlm-licensed product. Refer to the *CodeTEST Installation Guide* for information about the license manager and your CodeTEST license file.

Configuration files

Saving a configuration file

To save your current configuration, select Save Options File from the Options menu. The Save Configuration File dialog is displayed for you to enter the path and file name. See “File save dialogs” on page 2-26.

Note: At startup the host application will automatically search for a file named .ctconfig in your home directory. However, you may name your configuration files anything you wish and load them explicitly.

Startup configuration

To start CodeTEST with a predefined configuration, you can use the *-configfile* option to load a configuration file. For example:

```
CodeTEST -configfile /proj/mod3/ctconfig-mod3
```

If you do not specify a configuration file at startup, CodeTEST searches your home directory for a .ctconfig file. If .ctconfig is not present, CodeTEST is started with default values for all of the configuration variables that have defaults (described earlier under “Configuring a Session”).

Loading a configuration file during a session

To load a previously-saved configuration while the host application is running, select Load Configuration File from the Options menu. The Load Configuration File dialog is displayed for you to select a configuration file of any name. See “File load dialogs” on page 2-27.

Note: Most configuration changes will purge existing measurement data from memory. Before you change the configuration during a session, first save any collected measurement data that you wish to keep.

Saving and Loading Data

Saving data

To save collected CodeTEST measurement data, select Save Data from the toolbar Files menu. All of the current Continuous mode data (regardless of filtering) and all of the current trace data are saved in a single file.

Loading data

To load a data file upon starting the host application:

```
CodeTEST -datafile mydata.dat
```

To load a data file during a session, select Load Data from the toolbar File menu.

Licensing

To load and view data acquired with any or all of the CodeTEST tools, you need only one license of any type. If you load a data file without first checking out a license, CodeTEST will determine what licenses are available and automatically check one out for you.

IDB location

The IDB that was used when the data was captured must be available when you load a CodeTEST data file. The host application does NOT look for the IDB in the location specified in the Configuration dialog. The IDB file is specified at the top of each data file. If you have moved the IDB since saving the data, either move it back to its original location or use a text editor to edit the data file to reflect the new location.

Source files

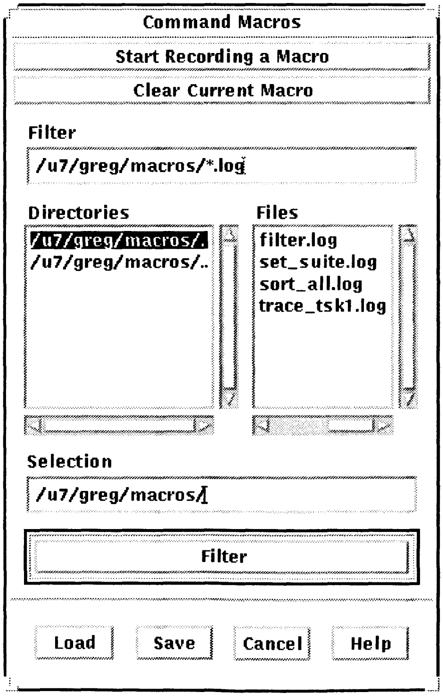
For CodeTEST to find the source code for your saved data (for display in the Source Code Viewer or Source view of a trace) the host application must be configured for the path or paths for all directories containing target source files.

General CodeTEST Utilities

The utilities described in this section are available for use with all of the CodeTEST tools. Utilities for specific modes or tools are covered in Chapter 3 and Chapter 4.

Macro dialog

To display the Macro dialog, select Macros from the toolbar Tools menu. With the Macro dialog you can create, save, and execute command macros using most CodeTEST host application commands.



Filter: /u7/greg/macros/*.log

Directories:
/u7/greg/macros/.
/u7/greg/macros/..

Files:
filter.log
set_suite.log
sort_all.log
trace_tsk1.log

Selection: /u7/greg/macros/

Creating a macro

To create a macro, click Start Recording a Macro, then use the keyboard and mouse controls to execute the series of CodeTEST commands you want to capture. (If you have previously recorded any commands, click the Clear Current Macro button before you begin recording a new macro.)

Note: To save a measurement setup, turn on macro recording to capture your Continuous Mode or Trace Mode setup commands.

Click Stop Recording when you have finished recording commands.

Saving a macro

To save the current macro, enter the path and file name in the Selection box, then click Save.

Executing a macro from within the host application

To run a previously-saved macro, first select the directory where the macro file is stored. If you wish, you can specify a file filter in the Filter box (top text entry box) and click Filter to display a list of files matching your filter criteria (*.log in the example above). Select the desired macro file from the list, then click the Load button to load and execute the macro.

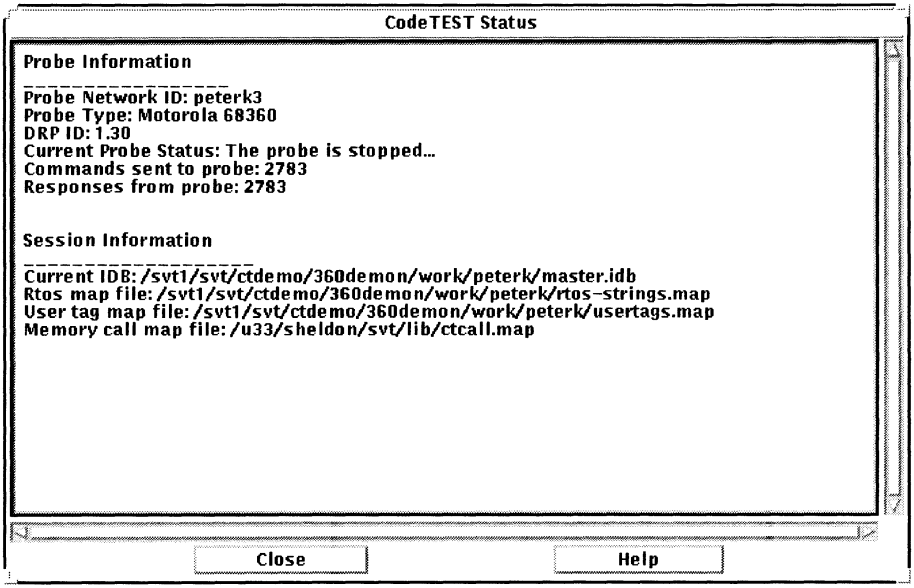
Executing a macro upon starting the host application

To execute a macro automatically upon starting the host application, use the *-macrofile* command line argument to supply a macro file name. For example:

```
CodeTEST -macrofile mymac.log
```

Status window

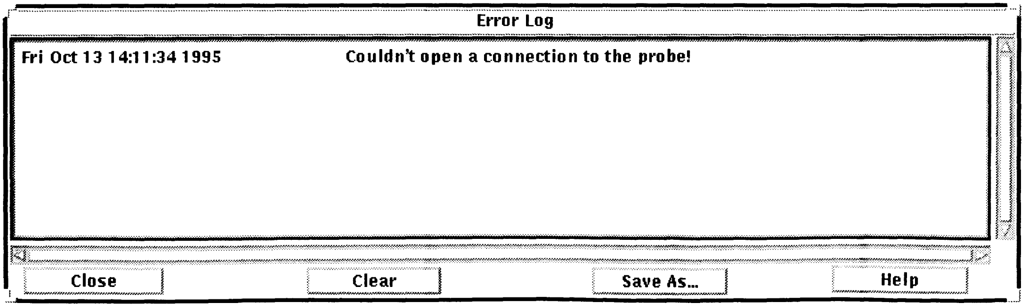
Select Status Window on the Tools menu to see information about your probe (including firmware version, total measurement time, etc.). Also shown are the file names in use during your session.



```
Probe Information
_________________
Probe Network ID: peterk3
Probe Type: Motorola 68360
DRP ID: 1.30
Current Probe Status: The probe is stopped...
Commands sent to probe: 2783
Responses from probe: 2783


Session Information
___________________
Current IDB: /svt1/svt/ctdemo/360demon/work/peterk/master.idb
Rtos map file: /svt1/svt/ctdemo/360demon/work/peterk/rtos-strings.map
User tag map file: /svt1/svt/ctdemo/360demon/work/peterk/usertags.map
Memory call map file: /u33/sheldon/svt/lib/ctcall.map
```

Error log

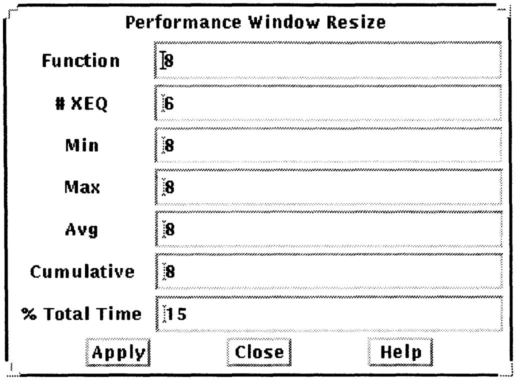
Select Error Log on the Tools menu to see a listing of the messages generated during your session. Appendix B gives troubleshooting tips for CodeTEST error messages.



```
Fri Oct 13 14:11:34 1995          Couldn't open a connection to the probe!
```

Window resize dialog

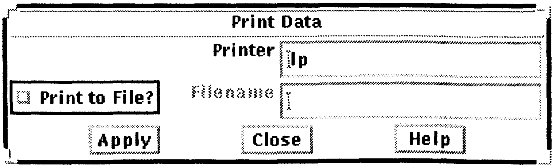
You can set the width of each column in any CodeTEST tabular view. To display the resizing dialog, select Resize on the Window menu for the view of interest. The example below shows the resize dialog for the Function Performance view. Edit the width value for any column you want to resize. Width is specified in M spaces (i.e., the width of an uppercase M in the screen font).



Function	8
# XEQ	6
Min	8
Max	8
Avg	8
Cumulative	8
% Total Time	15

Print dialog

You can print the contents of any CodeTEST view on a PostScript printer, or send the data to a printer-ready file for later printing. To print data, first apply any filters or sorting you want reflected in your printout (Continuous mode only). Only the data that is actually displayed will be printed. Select Print on the File menu for the view of interest, then specify a print queue, or click the Print to File box and enter a file name. Click Apply to execute the print command.



Printer lp

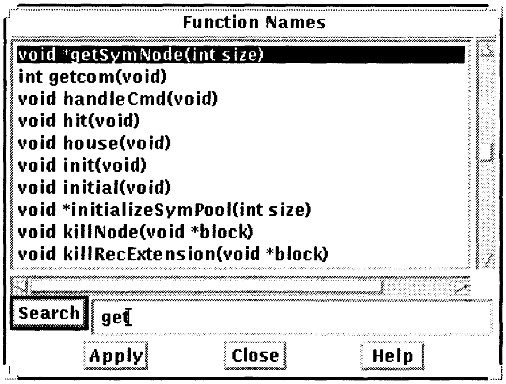
Filename

Selection lists

Selection list dialogs are available for browsing lists of task and function names while setting up your measurements.

Note: The Function Name selection list includes the names of all target program functions listed in the IDB (i.e., all instrumented functions). The Task Name selection list displays the task names from your RTOS map file (see "Creating an RTOS Map File" on page 8-9).

For example, if you are setting up the Trace tool to trigger on entry into a specific function, first select Function Entry as the trigger event, then click ... to display a list of your program's functions.



Function Names

```
void *getSymNode(int size)
int getcom(void)
void handleCmd(void)
void hit(void)
void house(void)
void init(void)
void initial(void)
void *initializeSymPool(int size)
void killNode(void *block)
void killRecExtension(void *block)
```

Search: get

To search for a name, enter a text string in the Search box and click Search to bring to the top of the list the first item that includes the specified string. Continue clicking Search to find other names that include the specified string.

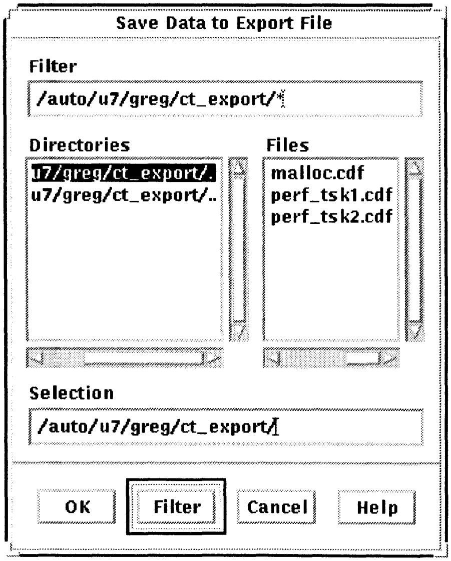
To copy the highlighted function or task name to the parent dialog, click Apply.

Export dialog

You can export the contents of any CodeTEST view to an ASCII comma delimited format (CDF) file.

Note: Before exporting from a Continuous mode view, first apply any filters or sorting you want reflected in the exported data. In Trace mode, first select the view (High Level, Control Flow, or Source) you want reflected in the exported data. Only data that is actually displayed will be exported.

Select Export from the File menu on the view of interest. The Export dialog is displayed. Specify a path and name for the export file, then click OK.



Save Data to Export File

Filter: /auto/u7/greg/ct_export/*

Directories:
- u7/greg/ct_export/.
- u7/greg/ct_export/..

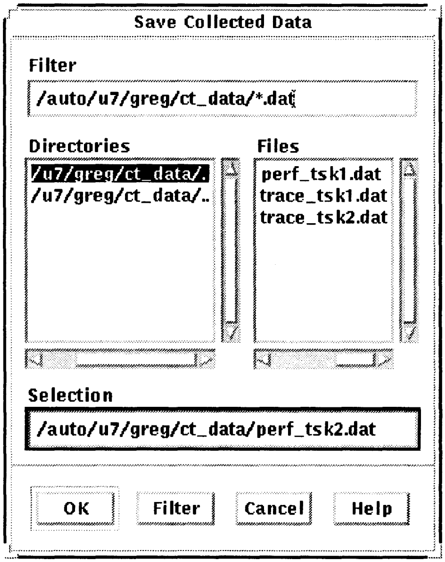
Files:
- malloc.cdf
- perf_tsk1.cdf
- perf_tsk2.cdf

Selection: /auto/u7/greg/ct_export/

You can import CodeTEST CDF files into most popular spreadsheet programs for post-processing measurement data, or into publishing programs for producing printed reports.

File save dialogs

The Save Collected Data dialog is displayed when you select Save Data from the toolbar File menu. A similar dialog is displayed when you select any CodeTEST *save* command (e.g., Save Options on the toolbar Options menu or Save As on the CodeTEST error log). The format and basic functionality of all CodeTEST file save dialogs is the same.



Save Collected Data

Filter

/auto/u7/greg/ct_data/*.dat

Directories

/u7/greg/ct_data/.
/u7/greg/ct_data/..

Files

perf_tsk1.dat
trace_tsk1.dat
trace_tsk2.dat

Selection

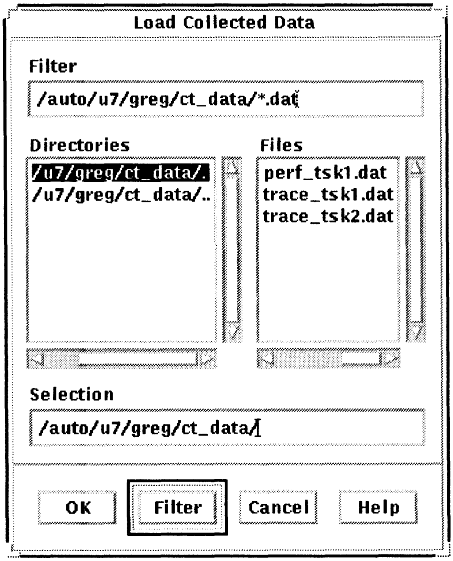
/auto/u7/greg/ct_data/perf_tsk2.dat

To save a file, first select a directory in the Directories scroll box. This places the directory path in the Filter box. If you wish, you can specify a file filter in the Filter box (using the * and ? wildcard characters). Click the Filter button to put your filter into effect. In the example above, the filter /auto/u7/greg/ct_data/*.dat was applied.

A list of files matching the filter criteria appears in the Files list. To overwrite an existing file, select the file name. To create a new file, type the file name into the Selection box. When the Selection box shows the correct path and file name, click OK to execute the save.

File load dialogs

The Load Collected Data dialog is displayed when you select Load Data from the tool bar File menu. A similar dialog is displayed when you select any CodeTEST *load* command (e.g., Load Options on the toolbar Options menu). The format and basic functionality of all CodeTEST file load dialogs is as shown below.



Load Collected Data

Filter

/auto/u7/greg/ct_data/*.dat

Directories

/u7/greg/ct_data/.
/u7/greg/ct_data/..

Files

perf_tsk1.dat
trace_tsk1.dat
trace_tsk2.dat

Selection

/auto/u7/greg/ct_data/

To load a file, first select the directory in the Directories scroll box. This places the directory path the Filter box. If you wish, you can also specify a file filter in the Filter box (using the * and ? wildcard characters). Click the Filter button to put your filter into effect. In the example above, the filter /auto/u7/greg/ct_data/*.dat was applied.

A list of files matching the filter criteria appears in the Files box. Select the file name you want. When the Selection box shows the correct path and file name, click OK to load the file.

Chapter 3

The Continuous Mode Toolset

Overview

The CodeTEST Continuous mode tools work together as a unified toolset, allowing you to make performance, coverage and memory allocation measurements simultaneously, monitoring up to 32,000 target program functions.

Making a measurement

To make a Continuous mode measurement:

1. Prepare your target system according to the guidelines under "Getting Started" in Chapter 2.
2. Download your instrumented program to the target hardware.
3. Start the CodeTEST host application and configure your session for the probe, target program, and licenses you want to use (see Chapter 2).
4. Select Set Continuous Mode on the toolbar Run menu. (Although you cannot make Trace and Continuous mode measurements simultaneously, you can display Trace and Continuous views side-by-side and toggle between modes without losing data.)
5. Click the tool buttons to display the views you want. Use the tools with their default setup, or click Setup on any Continuous mode view to set up the measurement.
6. Select Start Probe on the toolbar Run menu.
7. Use your run control device or stimulus to exercise the target system through some test scenario.
8. As data is received from the probe, views are updated at the rate you specified in step 3. The measurement continues until you select Stop Probe on the Run menu.
9. To add data to an existing measurement (or to a saved data file that has been reloaded into the host application) select Continue on the Run menu.

Continuous mode commands

The commands summarized in the following tables are available on the Continuous mode data views.

Command	Function
File Menu	
Print...	Displays the dialog for printing data (or printing to a file, which can later be sent to a printer). Only the currently displayed data (sorting and filters applied) is printed.
Export...	Displays the dialog for exporting data to an ASCII comma delimited format (CDF) file. Only displayed data in the current view (sorting and filters applied) is exported. Exported data can be loaded into most spreadsheet and publishing programs. **Note:** To save data in a form you can reload into CodeTEST, use the Save command on the toolbar File Menu.
Close	Dismisses the view without losing data.
Window Menu	
Resize...	Displays the dialog for setting column widths.
Help Menu	
Help on Window...	Displays help for the current window.
Contents...	Displays the help Contents page.
Search...	Displays the help Keyword Search dialog.
How to Use Help...	Displays help on how to use the help viewer.
About CodeTEST...	Displays the CodeTEST version number.

Button Controls

Command	Function
Find...	Displays the Find dialog for the active view. See "Search utilities" on page 3-28.
Sort...	Displays the Sort dialog for the active view. See "Sort utilities" on page 3-27.
Filter...	Displays the Filter dialog for the active view. See "Filter utilities" on page 3-29.
Setup...	Displays the Continuous Mode Setup dialog. See "Continuous Mode Setup" on page 3-5.

Right mouse button pop-up menu

Several of the Continuous mode views have pop-up menus you can display by positioning the cursor on any row in the table, then clicking and holding the right mouse button.

View	Menu Options
Function Performance	Display source code of function *xxx* Display statistics for function *xxx*
Call Linkage	Display source code for calling function *xxx* Display source code for called function *xxx* Display statistics for called function *xxx*
Branch Coverage	Display coverage in source file *xxx* Display statistics for function *xxx*
Memory Allocation	Display line *nn* of file *xxx* Display statistics for function *xxx*

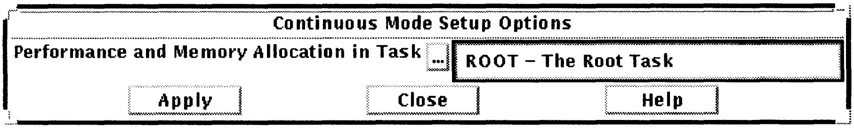
For information on the viewer, see "Source code viewer" on page 3-26. For information on function statistics, see "Function summary" on page 3-25.

Continuous Mode Setup

For RTOS applications, you have the option of qualifying performance and memory allocation measurements (not coverage measurements) to a specific task.

Note: For Continuous mode measurements that are not qualified to a specific task, CodeTEST begins the measurement when the first tag is received. For task-qualified measurements, CodeTEST does not begin the measurement until the first task switch is detected. See Chapter 8 for information about RTOS task tracking.

To display the Continuous Mode Setup dialog, click Setup on any Continuous mode view.



Continuous Mode Setup Options

Performance and Memory Allocation in Task ... ROOT - The Root Task

Qualifying measurements to an RTOS task

To qualify the acquisition of performance and memory allocation data, click the name entry box. Then either type in the task name or click … to make your selection from a list of task names derived from your RTOS map file. See "Creating an RTOS Map File" on page 8-9.

Note: To save a measurement setup, turn on macro recording to capture your setup commands in a CodeTEST macro. See "Creating a macro" on page 2-21.

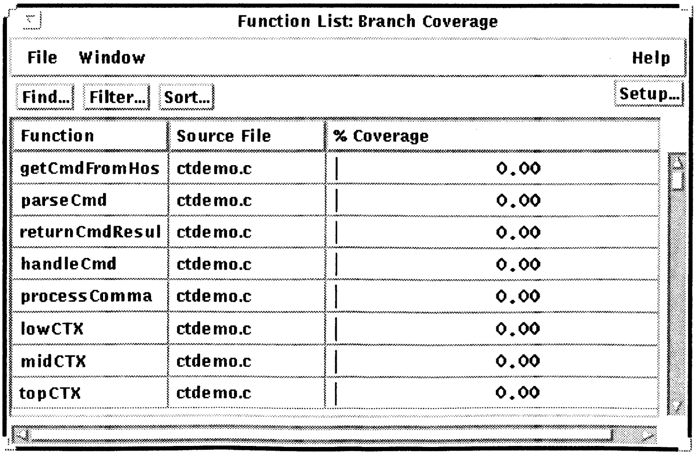
The Coverage Tool

The CodeTEST Coverage tool can provide valuable information about your target program and your test cases by examining your program's flow of execution to reveal exactly which code executes under each set of test conditions. With this knowledge you can develop additional tests more efficiently, and eliminate redundant or ineffective ones.

Note: See "Coverage tagging" on page 6-6 for an explanation of how the instrumenter places coverage tags in your code and how CodeTEST tracks coverage.

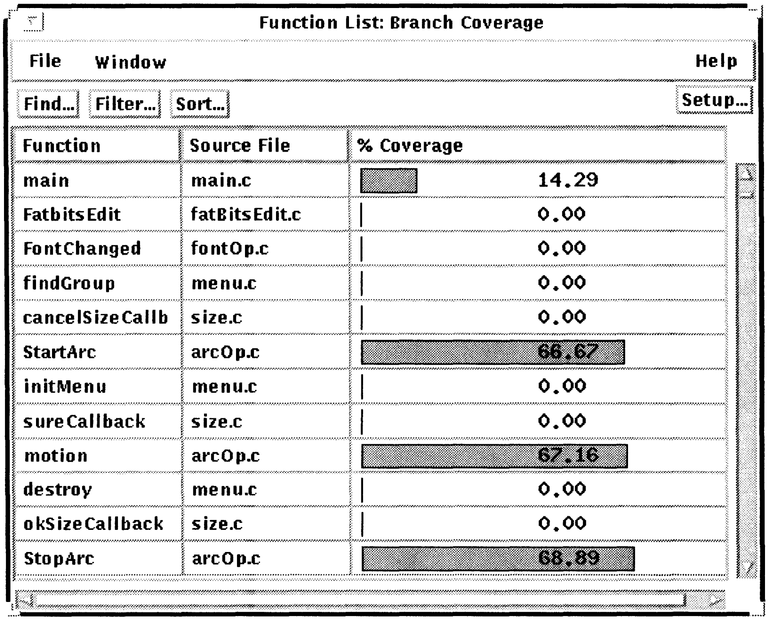
Branch coverage

The Branch Coverage view shows which areas of the target program have or have not executed during a measurement (or if you have merged coverage, during multiple measurements). If you display the Branch Coverage view after configuring your session to monitor the target program, but before starting a measurement, you will see a listing of the program's functions with 0% coverage shown for each.



Function	Source File	% Coverage
getCmdFromHos	ctdemo.c	0.00
parseCmd	ctdemo.c	0.00
returnCmdResul	ctdemo.c	0.00
handleCmd	ctdemo.c	0.00
processComma	ctdemo.c	0.00
lowCTX	ctdemo.c	0.00
midCTX	ctdemo.c	0.00
topCTX	ctdemo.c	0.00

Once you select Start Probe on the toolbar Run menu, and the host application begins to receive data, the view is updated periodically to show the coverage achieved.

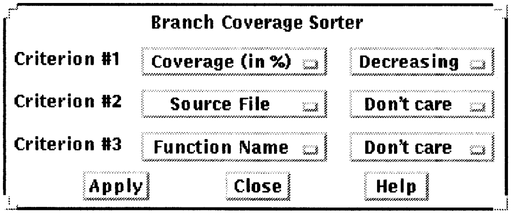


Function	Source File	% Coverage
main	main.c	14.29
FatbitsEdit	fatBitsEdit.c	0.00
FontChanged	fontOp.c	0.00
findGroup	menu.c	0.00
cancelSizeCallb	size.c	0.00
StartArc	arcOp.c	66.67
initMenu	menu.c	0.00
sureCallback	size.c	0.00
motion	arcOp.c	67.16
destroy	menu.c	0.00
okSizeCallback	size.c	0.00
StopArc	arcOp.c	68.89

The Branch Coverage table contains:

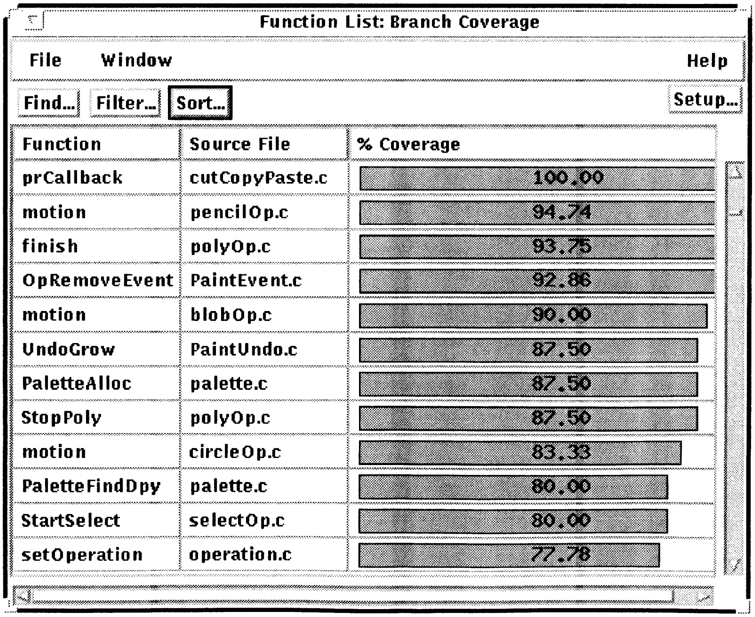
Column	Description
Function	All of the target program's instrumented functions.
File	Source file in which each function is located.
% Coverage	The percentage of basic blocks within each function that have executed during the measurement (or suite of measurements if you have merged coverage data). The function with the highest percentage has a full histogram; histograms for all other functions are sized relative to the high value.

To sort the data so you can easily see which areas have been covered, perform a sort on the % Coverage column. Click the Sort button on the Branch Coverage view, set up the dialog as follows, then click Apply.


Branch Coverage Sorter

Criterion #1	Coverage (in %)	Decreasing
Criterion #2	Source File	Don't care
Criterion #3	Function Name	Don't care

With the % Coverage column sorted in decreasing order, functions with the highest levels of coverage are arranged at the top of the table.

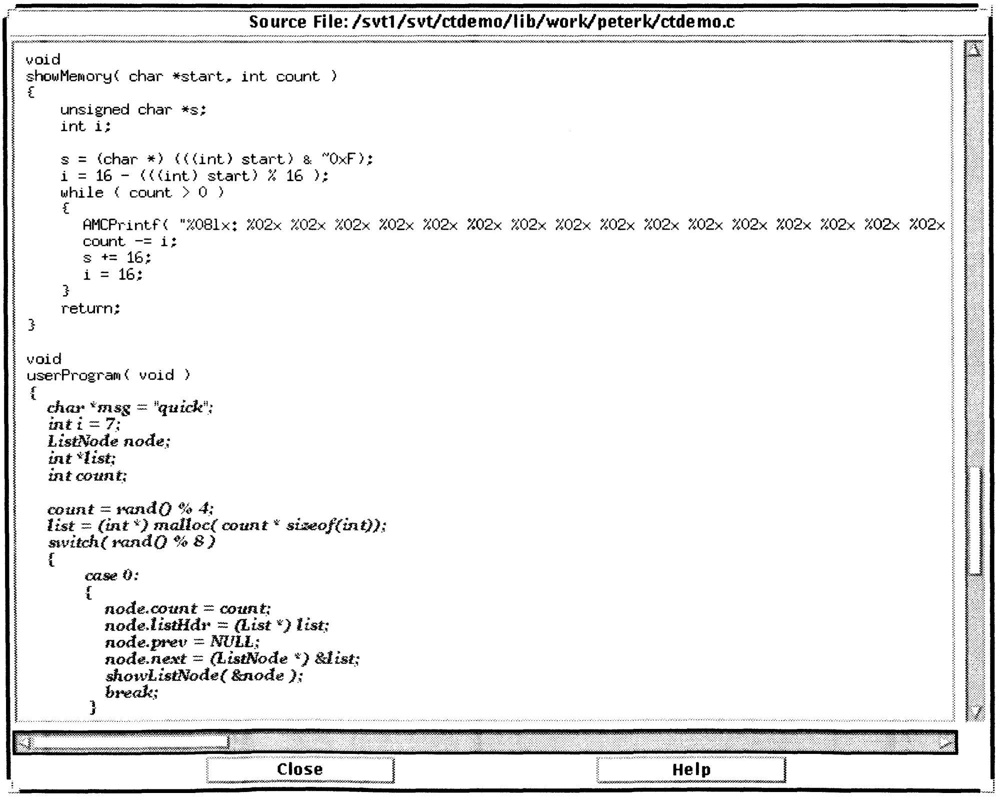

Function List: Branch Coverage

Function	Source File	% Coverage
prCallback	cutCopyPaste.c	100.00
motion	pencilOp.c	94.74
finish	polyOp.c	93.75
OpRemoveEvent	PaintEvent.c	92.86
motion	blobOp.c	90.00
UndoGrow	PaintUndo.c	87.50
PaletteAlloc	palette.c	87.50
StopPoly	polyOp.c	87.50
motion	circleOp.c	83.33
PaletteFindDpy	palette.c	80.00
StartSelect	selectOp.c	80.00
setOperation	operation.c	77.78

Viewing line coverage

When coverage analysis reveals untested areas of your program, you can use the Source Code Viewer to see exactly which code has executed, and examine the control flow leading to uncovered branches.

Position the cursor on the Branch Coverage row for the function of interest, click the right mouse button, and select "Display coverage in source file *xxx*." The source viewer displays the selected file, with executed code highlighted.

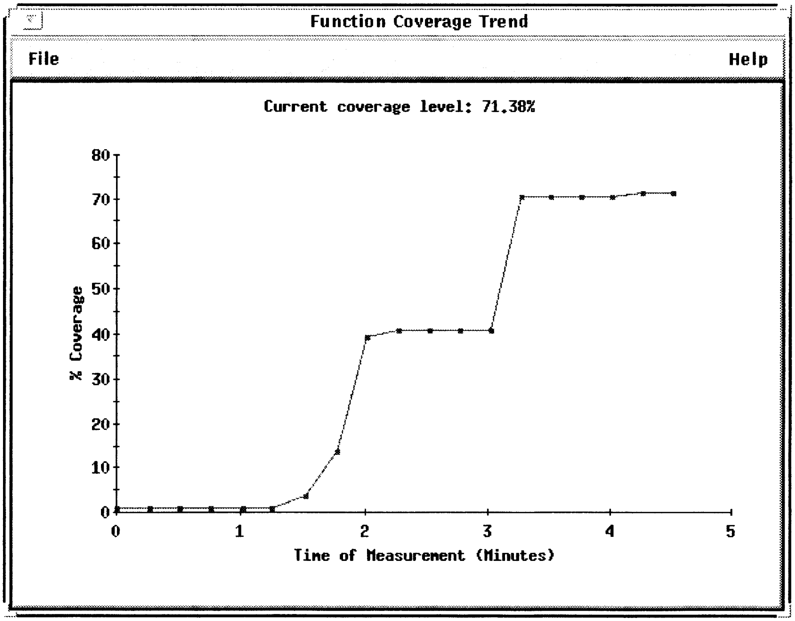


Source File: /svt1/svt/ctdemo/lib/work/peterk/ctdemo.c

```
void
showMemory( char *start, int count )
{
    unsigned char *s;
    int i;

    s = (char *) (((int) start) & ~0xF);
    i = 16 - (((int) start) % 16 );
    while ( count > 0 )
    {
       AMCPrintf( "%081x: %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
       count -= i;
       s += 16;
       i = 16;
    }
    return;
}

void
userProgram( void )
{
   char *msg = "quick";
   int i = 7;
   ListNode node;
   int *list;
   int count;

   count = rand() % 4;
   list = (int *) malloc( count * sizeof(int));
   switch( rand() % 8 )
   {
        case 0:
        {
          node.count = count;
          node.listHdr = (List *) list;
          node.prev = NULL;
          node.next = (ListNode *) &list;
          showListNode( &node );
          break;
        }
```

Coverage trend

The Coverage Trend view is an XY graph showing the level of coverage over time that your testing achieves. As a coverage measurement runs, the graph plots the percentage of the target program's branch points that are executed as of each update from the probe. You specify the update interval on the Probe section of the Configuration dialog (see page 2-14). The graph's horizontal axis automatically scales and adjusts time increments, providing trend information for measurements that run minutes, hours, or days.



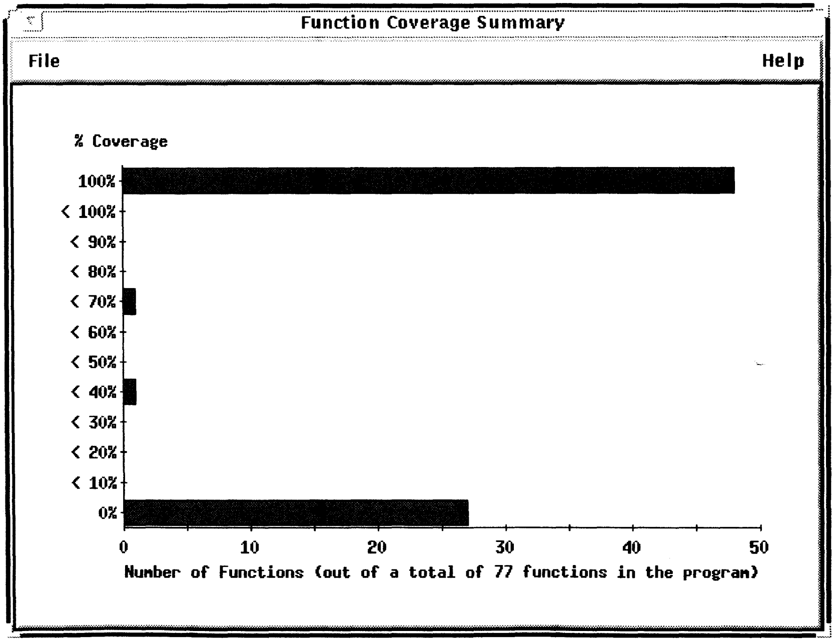
With the Coverage Trend view, you can easily identify *dead spots* in your testing, where no new coverage is being achieved. By modifying your test stimulus accordingly, you may be able to reduce your testing time substantially, while still testing just as thoroughly.

Coverage summary

The Coverage Summary view displays a bar graph showing the overall level of coverage achieved during a measurement (or during multiple measurements, if you have merged coverage data).

The graph categorizes your program's functions into the percentile ranges indicated along the vertical axis. The horizontal axis indicates the total number of target program functions and the number of functions with coverage levels that fall within each percentile range.

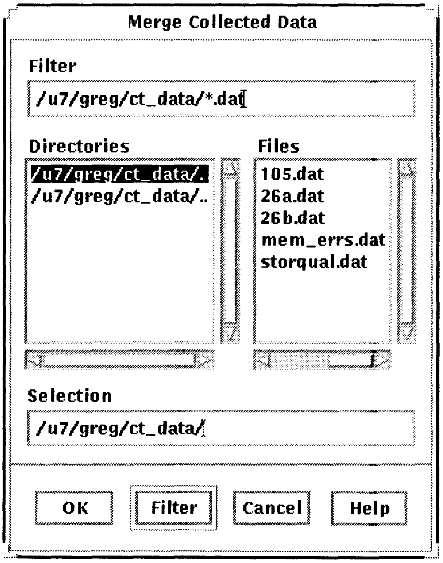
In the example below, the majority of the target module's 77 functions have achieved 100% coverage, but there are still 27 functions that have achieved 0%.


Function Coverage Summary
File
Help
% Coverage
100%
< 100%
< 90%
< 80%
< 70%
< 60%
< 50%
< 40%
< 30%
< 20%
< 10%
0%
0
10
20
30
40
50
Number of Functions (out of a total of 77 functions in the program)

Merging coverage data

You can merge the coverage data from multiple measurements to see the composite level of coverage achieved in a suite of tests.

1. Configure the host application for your probe, target program, and licenses. Then either make a new Continuous mode measurement or load a data file that contains coverage data.
2. Select Merge Coverage Data on the toolbar Files menu and load a file that contains coverage data you want to merge with the data acquired in step 1. Note that the IDB for the file you select must be the same as the IDB that was used when you acquired the data in step 1.



Filter: /u7/greg/ct_data/*.dat

Directories:
/u7/greg/ct_data/.
/u7/greg/ct_data/..

Files:
105.dat
26a.dat
26b.dat
mem_errs.dat
storqual.dat

Selection: /u7/greg/ct_data/

3. Repeat step 2 for each file whose coverage data you want to merge. Composite coverage is reflected in the Branch Coverage view and the Coverage Summary graph.
4. To save your merged coverage data, select Save Data on the toolbar Files menu.

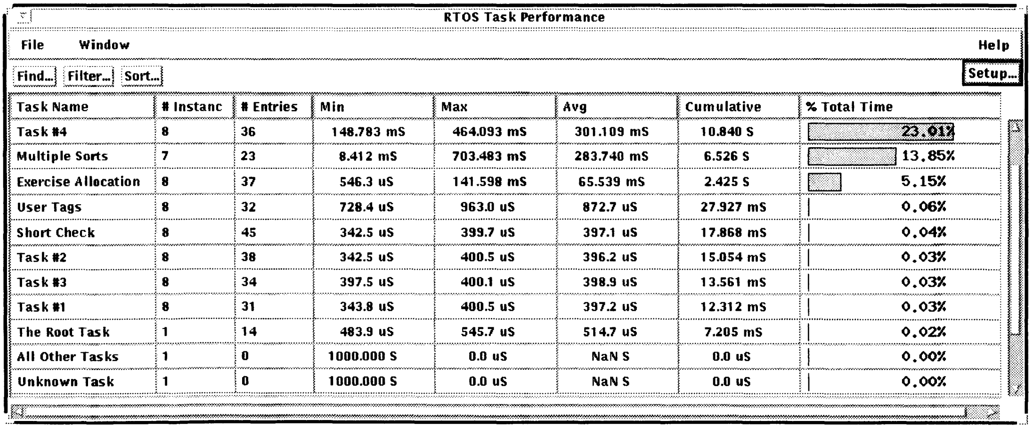
The Performance Tool

The Performance tool can help you determine whether your program is responding quickly enough to meet efficiency requirements for response times and throughput rates. By analyzing performance data, comparing each task against the other tasks, each function against the other functions, you can determine without guesswork which parts of your program need tuning for optimum performance.

Task performance

For RTOS applications, CodeTEST internally tracks the function context for each task running in the target system. See Chapter 8 for information about RTOS task tracking and about preparing your RTOS for use with CodeTEST.

The Task Performance view displays your target program's *timing and counts* information on a task-by-task basis.



Task Name	# Instanc	# Entries	Min	Max	Avg	Cumulative	% Total Time
Task #4	8	36	148.783 mS	464.093 mS	301.109 mS	10.840 S	23.01%
Multiple Sorts	7	23	8.412 mS	703.483 mS	283.740 mS	6.526 S	13.85%
Exercise Allocation	8	37	546.3 uS	141.598 mS	65.539 mS	2.425 S	5.15%
User Tags	8	32	728.4 uS	963.0 uS	872.7 uS	27.927 mS	0.06%
Short Check	8	45	342.5 uS	399.7 uS	397.1 uS	17.868 mS	0.04%
Task #2	8	38	342.5 uS	400.5 uS	396.2 uS	15.054 mS	0.03%
Task #3	8	34	397.5 uS	400.1 uS	398.9 uS	13.561 mS	0.03%
Task #1	8	31	343.8 uS	400.5 uS	397.2 uS	12.312 mS	0.03%
The Root Task	1	14	483.9 uS	545.7 uS	514.7 uS	7.205 mS	0.02%
All Other Tasks	1	0	1000.000 S	0.0 uS	NaN S	0.0 uS	0.00%
Unknown Task	1	0	1000.000 S	0.0 uS	NaN S	0.0 uS	0.00%

If the probe doesn't capture all of the tags written to the tag port addresses, the sampling rate (i.e., the percentage of time within the update period the ports were monitored) is displayed above the column headings.

The Task Performance table contains:

Column	Description
Task Name	The name of each task that has executed. This can be either the task name or a string associated with the name in an RTOS map file (see Chapter 8).
# Instances	The number of instances (creations) of each task during the measurement. CodeTEST can track up to 1,000 active task instances at a time, and tabulate results for thousands of task instances over the duration of the measurement.
# Entries	The number of entries into each task.
Min	The minimum time slice for each task.
Max	The maximum time slice for each task.
Avg	The average time slice for each task.
Cumulative	The cumulative execution time for each task.
% Total Time	Percentage of total measurement time represented by the cumulative execution time for this task, shown as a histogram and a numeric value. The task with the highest percentage has a full histogram bar; histograms for all other tasks are sized relative to the high value.

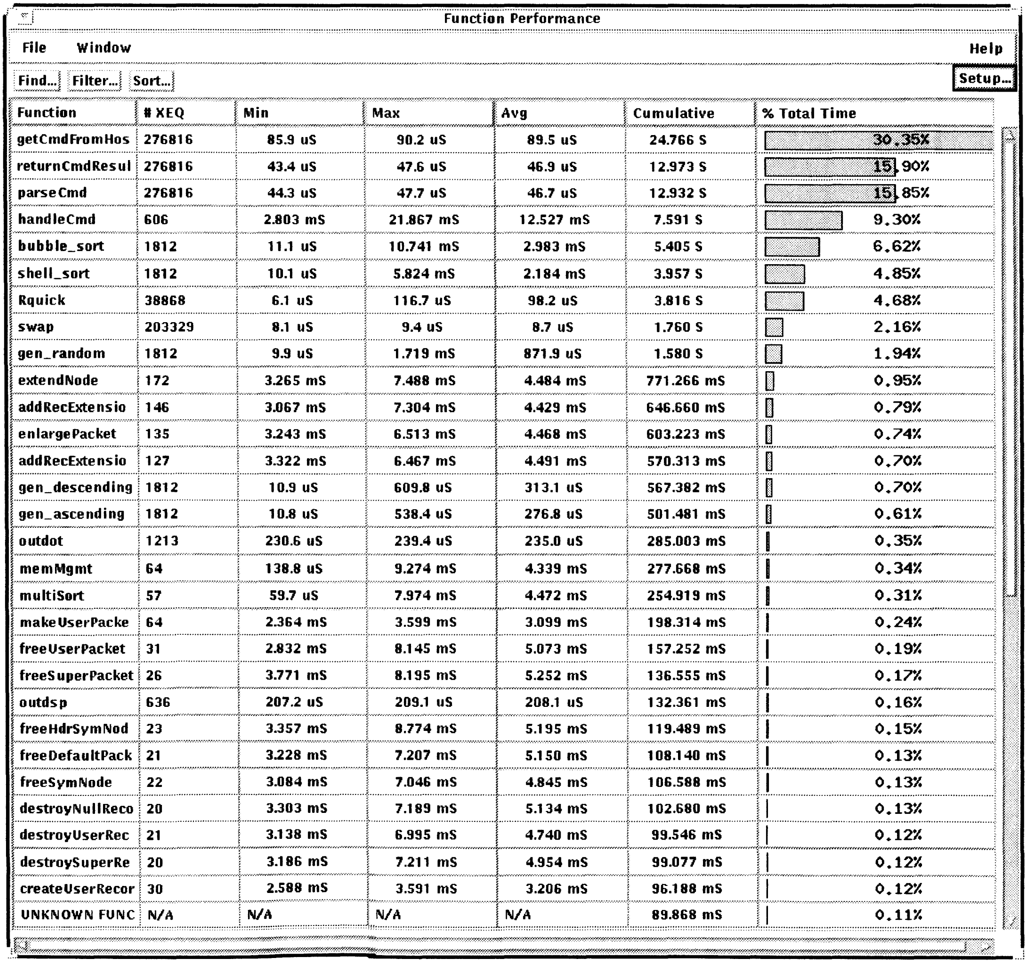
Special entries in the Task Performance view

The *Unknown Task* entry shows execution time for a task that could not be identified. This typically occurs for a task that was already running when you started the measurement—often a task containing an idle loop of some kind.

The *All Other Tasks* entry accumulates time for all tasks that could not be tracked individually because the maximum number of tasks has been exceeded, or because measurement qualification is in effect (see “Continuous Mode Setup” on page 3-5.)

Function performance

The Function Performance view presents timing and counts information for each function that executes during the measurement, allowing you to compare relative efficiencies of various portions of your target program and quickly identify slow-running code. For a discussion of how the instrumenter places function entry and exit tags in your code, see "Performance tagging" on page 6-3.



Function	# XEQ	Min	Max	Avg	Cumulative	% Total Time
getCmdFromHos	276816	85.9 uS	90.2 uS	89.5 uS	24.766 S	30.35%
returnCmdResul	276816	43.4 uS	47.6 uS	46.9 uS	12.973 S	15.90%
parseCmd	276816	44.3 uS	47.7 uS	46.7 uS	12.932 S	15.85%
handleCmd	606	2.803 mS	21.867 mS	12.527 mS	7.591 S	9.30%
bubble_sort	1812	11.1 uS	10.741 mS	2.983 mS	5.405 S	6.62%
shell_sort	1812	10.1 uS	5.824 mS	2.184 mS	3.957 S	4.85%
Rquick	38868	6.1 uS	116.7 uS	98.2 uS	3.816 S	4.68%
swap	203329	8.1 uS	9.4 uS	8.7 uS	1.760 S	2.16%
gen_random	1812	9.9 uS	1.719 mS	871.9 uS	1.580 S	1.94%
extendNode	172	3.265 mS	7.488 mS	4.484 mS	771.266 mS	0.95%
addRecExtensio	146	3.067 mS	7.304 mS	4.429 mS	646.660 mS	0.79%
enlargePacket	135	3.243 mS	6.513 mS	4.468 mS	603.223 mS	0.74%
addRecExtensio	127	3.322 mS	6.467 mS	4.491 mS	570.313 mS	0.70%
gen_descending	1812	10.9 uS	609.8 uS	313.1 uS	567.382 mS	0.70%
gen_ascending	1812	10.8 uS	538.4 uS	276.8 uS	501.481 mS	0.61%
outdot	1213	230.6 uS	239.4 uS	235.0 uS	285.003 mS	0.35%
memMgmt	64	138.8 uS	9.274 mS	4.339 mS	277.668 mS	0.34%
multiSort	57	59.7 uS	7.974 mS	4.472 mS	254.919 mS	0.31%
makeUserPacke	64	2.364 mS	3.599 mS	3.099 mS	198.314 mS	0.24%
freeUserPacket	31	2.832 mS	8.145 mS	5.073 mS	157.252 mS	0.19%
freeSuperPacket	26	3.771 mS	8.195 mS	5.252 mS	136.555 mS	0.17%
outdsp	636	207.2 uS	209.1 uS	208.1 uS	132.361 mS	0.16%
freeHdrSymNod	23	3.357 mS	8.774 mS	5.195 mS	119.489 mS	0.15%
freeDefaultPack	21	3.228 mS	7.207 mS	5.150 mS	108.140 mS	0.13%
freeSymNode	22	3.084 mS	7.046 mS	4.845 mS	106.588 mS	0.13%
destroyNullReco	20	3.303 mS	7.189 mS	5.134 mS	102.680 mS	0.13%
destroyUserRec	21	3.138 mS	6.995 mS	4.740 mS	99.546 mS	0.12%
destroySuperRe	20	3.186 mS	7.211 mS	4.954 mS	99.077 mS	0.12%
createUserRecor	30	2.588 mS	3.591 mS	3.206 mS	96.188 mS	0.12%
UNKNOWN FUNC	N/A	N/A	N/A	N/A	89.868 mS	0.11%

The Function Performance table contains:

Column	Description
Function	The name of each function that has executed.
# XEQ	The number of times each function has executed.
Min	The minimum execution time for each function.
Max	The maximum execution time for each function.
Avg	The average execution time for each function.
Cumulative	The cumulative execution time for each function.
% Total Time	The percentage of the target program's overall CPU time each function has consumed, shown as a histogram and a numeric value. The function with the highest percentage of overall execution time has a full histogram bar; histograms for all other functions are sized relative to the high value.

Note: It is possible for a function to show zero in the #XEQ column and yet have some non-zero execution time attributed to it. This happens when a function entry occurs while a measurement is running, but there is no exit from that function while the measurement is still running. (Execution is counted when CodeTEST detects the function exit, not the entry.) The minimum and maximum execution times are not shown in such a case because it was just a "partial" execution. This situation typically occurs with high-level idle loops that never exit, as in main(). For routines with direct recursion, CodeTEST averages each call's execution time before updating the minimum and maximum execution times.

Special entry in the Function Performance table

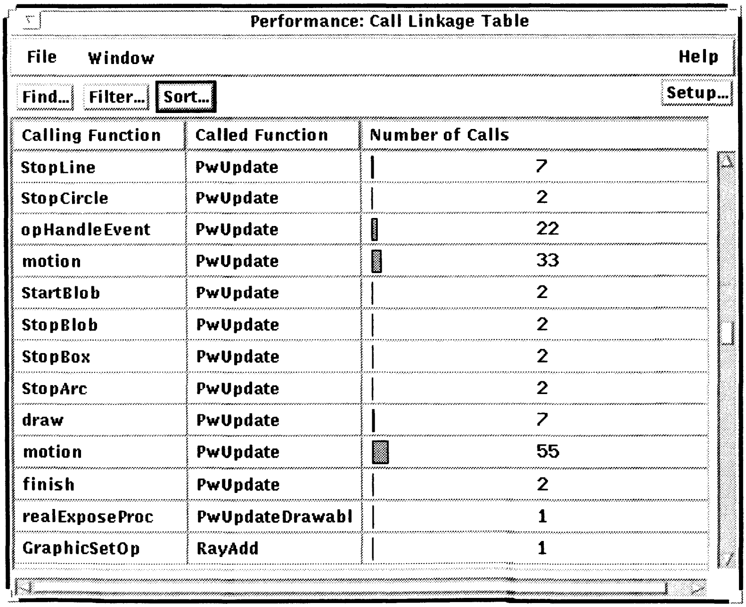
A special entry in the Function Performance table identified as *Unknown Function* shows execution time for a function whose identity is not known because there was no entry or exit during the measurement. This typically occurs for an idle routine the program frequently returns to, or the top-level function of a program that was already running when you started the measurement.

Right mouse-button menu

To display the source code or a summary of the accumulated data for any function position the cursor on the function line, then click the right mouse button.

Call linkage

The Call Linkage view provides information about the call pair relationships among your target program's functions.



Calling Function	Called Function	Number of Calls
StopLine	PwUpdate	7
StopCircle	PwUpdate	2
opHandleEvent	PwUpdate	22
motion	PwUpdate	33
StartBlob	PwUpdate	2
StopBlob	PwUpdate	2
StopBox	PwUpdate	2
StopArc	PwUpdate	2
draw	PwUpdate	7
motion	PwUpdate	55
finish	PwUpdate	2
realExposeProc	PwUpdateDrawabl	1
GraphicSetOp	RayAdd	1

Rather than relying on a static call tree, the Performance tool tracks calls at run time, by means of performance tags in your code, so the Call Linkage table includes function calls made via pointers.

Each row in the Call Linkage table represents a *call pair*. CodeTEST defines a call pair as any two function entry tags captured sequentially by the probe. For each call pair, the Number of Calls column shows how many times the "caller" has called the "callee." The call pair with the highest number of calls has a full histogram bar; histograms for all other call pairs are sized relative to the high value. The default sort for this view is by number of calls.

Investigating a function with an extremely high occurrence count may reveal an algorithmic problem, such as thrashing, or it may be that the frequently called function is a candidate for in-lining.

Right mouse-button menu

To display the source code for any calling or called function, or to display a summary of the current measurement data for the called function, position the cursor on the table entry of interest, then click the right mouse button.

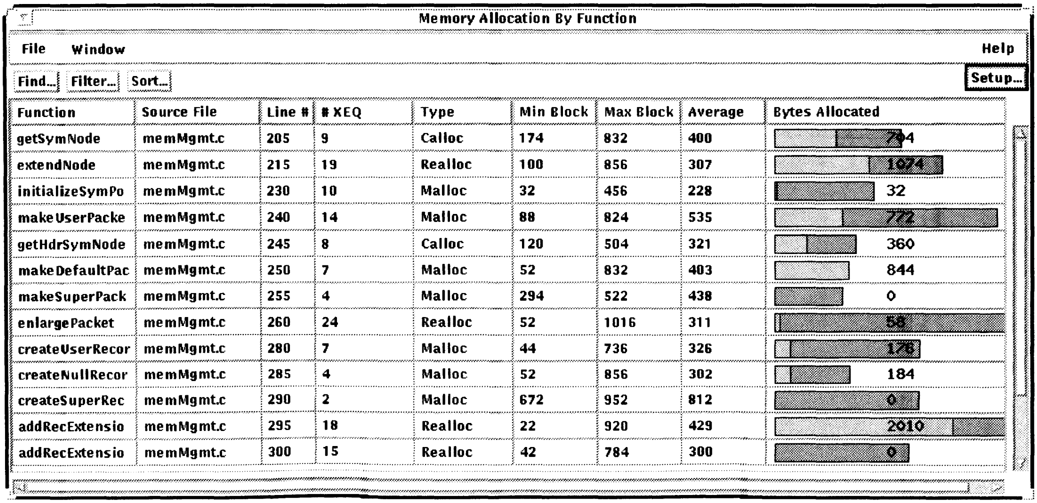
The Memory Tool

Memory allocation errors can be among the most difficult software defects to track down and eliminate. Because the C language and its derivatives are so flexible, errors such as memory *leaks* and multiple *frees* can evade detection, their actual cause often far removed from any visible symptom. Using the Memory tool to preventively monitor your program, you can "filter out" these latent errors, whether a symptom has actually appeared or not.

Note: To use the Memory tool, you must instrument your code for memory monitoring (see "Memory tagging" on page 6-8) and build and link the CodeTEST memory management routines with your target program (see Chapter 7).

Memory allocation

The Memory Allocation by Function view tracks your program's dynamic allocation and deallocation of memory.



Function	Source File	Line #	# XEQ	Type	Min Block	Max Block	Average	Bytes Allocated
getSymNode	memMgmt.c	205	9	Calloc	174	832	400	704
extendNode	memMgmt.c	215	19	Realloc	100	856	307	1074
initializeSymPo	memMgmt.c	230	10	Malloc	32	456	228	32
makeUserPacke	memMgmt.c	240	14	Malloc	88	824	535	772
getHdrSymNode	memMgmt.c	245	8	Calloc	120	504	321	360
makeDefaultPac	memMgmt.c	250	7	Malloc	52	832	403	844
makeSuperPack	memMgmt.c	255	4	Malloc	294	522	438	0
enlargePacket	memMgmt.c	260	24	Realloc	52	1016	311	58
createUserRecor	memMgmt.c	280	7	Malloc	44	736	326	176
createNullRecor	memMgmt.c	285	4	Malloc	52	856	302	184
createSuperRec	memMgmt.c	290	2	Malloc	672	952	812	0
addRecExtensio	memMgmt.c	295	18	Realloc	22	920	429	2010
addRecExtensio	memMgmt.c	300	15	Realloc	42	784	300	0

Each row in the table represents an *allocation caller* (i.e., a specific location in your target code where a memory allocation routine is called). Columns are defined as follows.

Column	Description
Function	The function in which each allocation caller is located.
Source File	The source file that contains the function in which each allocation caller is located.
Line #	The caller's line number location.
# XEQ	The number of times each allocation caller has executed in the current measurement.
Type	The type of memory management routine called.
Min Block	The smallest memory block allocated by this allocation caller during the measurement.
Max Block	The largest memory block allocated by this allocation caller during the measurement.
Average	The average memory block size allocated by this allocation caller during the measurement.
Bytes Allocated	The number of bytes currently allocated by each allocation caller, shown numerically and by the light-colored portion of the histogram. The darker portion of the histogram represents the caller's highwater mark (i.e., the maximum amount of memory allocated at any one time).

Note: A negative value in the Bytes Allocated column indicates that CodeTEST monitored a deallocation but did not monitor the original allocation. To avoid negative allocation values, be sure to start the probe before your target program allocates memory.

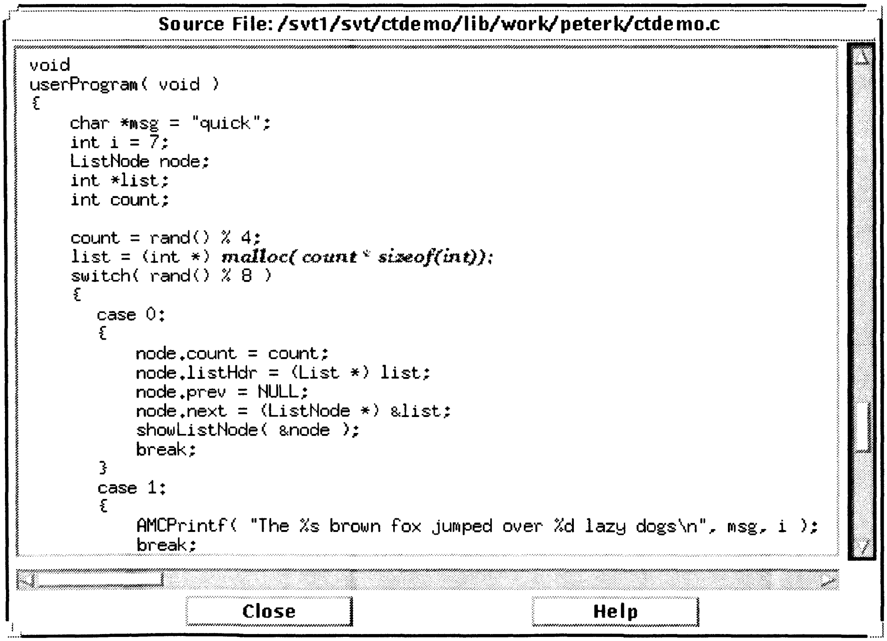
Special entry in the Memory Allocation table

The *Unidentified Malloc Call* entry in the Memory Allocation table represents allocations that cannot be credited to a known (i.e., instrumented) allocation caller. This is the result of non-instrumented code calling the CodeTEST memory routines.

Viewing the source

To view the source code for any executed memory management call, position the cursor on the row of interest in the Memory Allocation table, click the right mouse button, then select “Display line *xxx* of file *xxx*.”

The Source Code Viewer displays the file containing the selected function, with the relevant code highlighted.

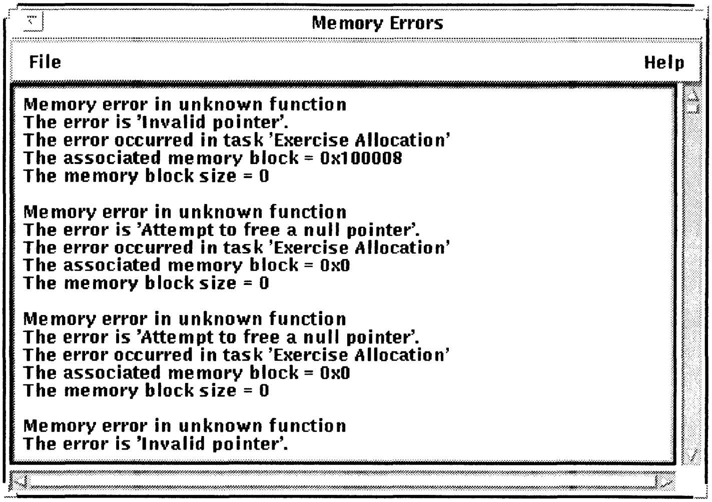


Source File: /svt1/svt/ctdemo/lib/work/peterk/ctdemo.c

```
void
userProgram( void )
{
    char *msg = "quick";
    int i = 7;
    ListNode node;
    int *list;
    int count;

    count = rand() % 4;
    list = (int *) malloc( count * sizeof(int));
    switch( rand() % 8 )
    {
       case 0:
       {
           node.count = count;
           node.listHdr = (List *) list;
           node.prev = NULL;
           node.next = (ListNode *) &list;
           showListNode( &node );
           break;
       }
       case 1:
       {
           AMCPrintf( "The %s brown fox jumped over %d lazy dogs\n", msg, i );
           break;
```

Memory error log

The Memory Error log displays a listing of up to 200 memory error messages. Beyond that, any messages generated are ignored for the remainder of the measurement.



```
Memory error in unknown function
The error is 'Invalid pointer'.
The error occurred in task 'Exercise Allocation'
The associated memory block = 0x100008
The memory block size = 0

Memory error in unknown function
The error is 'Attempt to free a null pointer'.
The error occurred in task 'Exercise Allocation'
The associated memory block = 0x0
The memory block size = 0

Memory error in unknown function
The error is 'Attempt to free a null pointer'.
The error occurred in task 'Exercise Allocation'
The associated memory block = 0x0
The memory block size = 0

Memory error in unknown function
The error is 'Invalid pointer'.
```

Note: Checking the consistency of the heap's internal data structures can sometimes be quite slow. See "CodeTEST memory management switches" on page 7-6 for information about enabling or disabling memory error checks.

Error severity levels

Fatal—continued execution of the target program is not prevented, but future calls to the CodeTEST allocator will return their normal error return values, e.g., malloc() will return NULL. This condition can only be cleared by reinitializing the task that encountered the error.

Nonfatal—the current memory management call failed, but future calls may succeed.

Info—the message does not indicate an error condition.

Error messages

❑ No Errors

This is a normal return from a memory management routine.

❑ Out of heap space (nonfatal)

There is not enough memory available in the heap to satisfy an allocation request. Future requests for smaller heap blocks may succeed. Retrying this request may succeed if the target program first frees one or more heap blocks. Consider reducing the target program's heap usage, or reconfigure the task or operating system to make more memory available to the heap.

❑ Heap has been corrupted (fatal)

Consistency checks have discovered a corrupted field in a heap block header. This is usually caused by a write through an incorrect pointer, possibly to a location in a heap block that was allocated, then freed, then reused as part of a new heap block. Sometimes this is caused by a bad array index.

❑ Free with invalid pointer (nonfatal)

The target program passed an invalid pointer to a deallocation routine. The pointer either contains an address that is not within the heap, or it is incorrectly aligned (possibly due to an error in pointer arithmetic). Other wild pointers may be caused by uninitialized memory, reading from a freed heap block, incorrect use of a union, or adding two pointers together.

❑ Trailing guard bytes overwritten (fatal)

One or more of the guard bytes immediately following your data have been overwritten. Try looking for incorrect pointer arithmetic, a bad array index, a write through a pointer in a freed heap block. There could be an *off-by-one* or other length calculation error causing the target program to write past the end of a string.

- ❑ Leading guard bytes overwritten (fatal)

  One or more of the guard bytes immediately preceding your data have been overwritten. Try looking for incorrect pointer arithmetic, a bad array index, or a write through a pointer in a freed heap block.

- ❑ Free with NULL pointer (nonfatal)

  Target program has passed a NULL pointer to a deallocation routine. ANSI C allows NULL pointers to be passed to heap deallocators but since this sometimes indicates a problem in the target program, the CodeTEST memory manager reports it.

- ❑ Free of already free heap block (nonfatal)

  The target program has tried to free a heap block that has already been freed. This error is returned only if the space for the previously-freed block has not been reused as part of a more recently allocated block. This error often indicates a serious defect in the target program. Try looking for duplicate pointers to this heap item in structures or arrays, or for functions that are called more than once as expression side effects. There is a slight chance this error is caused by a wild write that corrupted the heap structure.

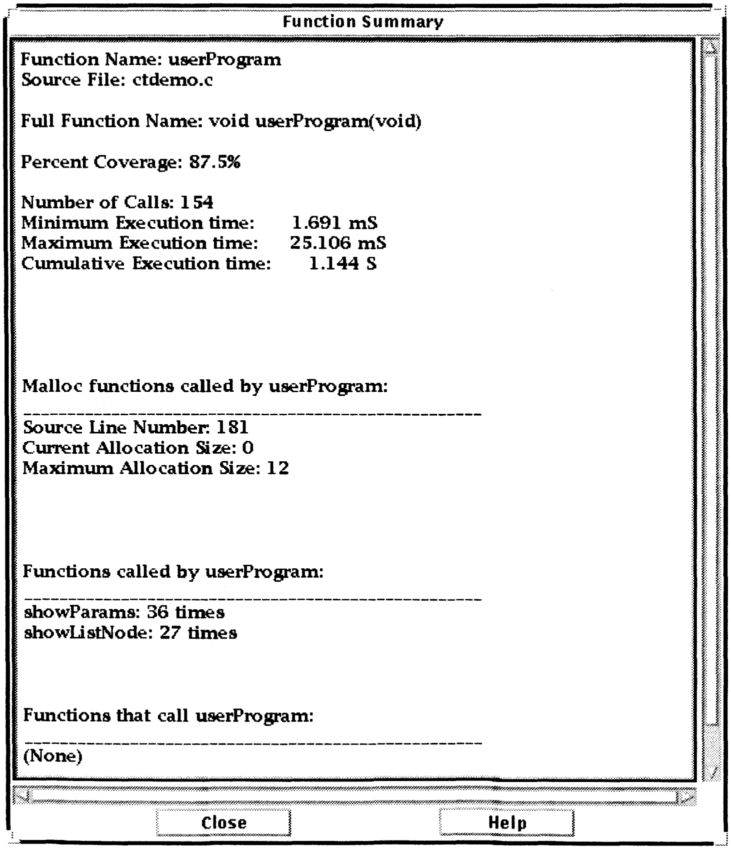
- ❑ New heap highwater mark (info)

  If the CodeTEST memory management routines have been compiled to do so, this message will be generated each time the total size of the heap grows past its previous maximum.

Continuous Mode Utilities

Function summary

The Function Summary summarizes all Continuous mode data collected for a given function during the current measurement. You can access the Function Summary from the Function Performance, Call Linkage, Branch Coverage, and Memory Allocation views by positioning the cursor on the function of interest, clicking the right mouse button, and selecting “Display statistics for function *xxxx*”.

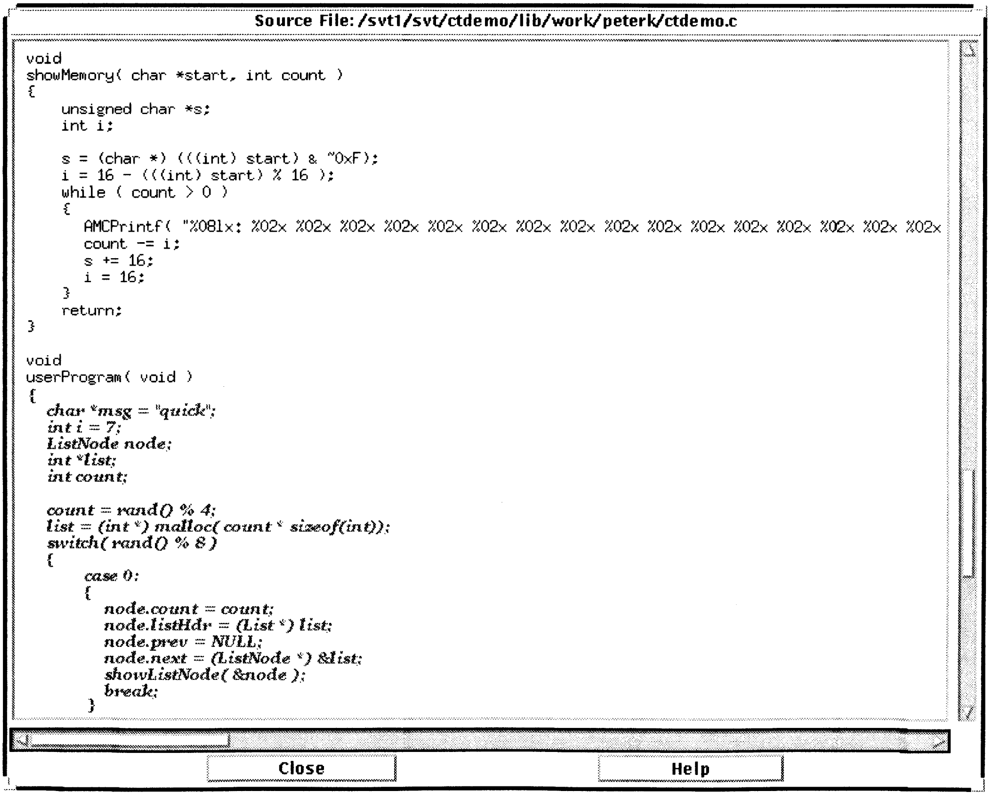


```
Function Name: userProgram
Source File: ctdemo.c

Full Function Name: void userProgram(void)

Percent Coverage: 87.5%

Number of Calls: 154
Minimum Execution time:      1.691 mS
Maximum Execution time:     25.106 mS
Cumulative Execution time:   1.144 S



Malloc functions called by userProgram:
-----------------------------------------------
Source Line Number: 181
Current Allocation Size: 0
Maximum Allocation Size: 12



Functions called by userProgram:
-----------------------------------------------
showParams: 36 times
showListNode: 27 times



Functions that call userProgram:
-----------------------------------------------
(None)
```

Source code viewer

You can use the Source Code Viewer to browse the source for any function listed in the Function Performance, Call Linkage, Branch Coverage, or Memory Allocation views. To invoke the viewer, position the cursor on the table entry for the function of interest, click the right mouse button, then select "Display source code of function *xxx*" from the pop-up menu.

Note: If you instrumented your code for coverage monitoring and selected the Coverage license for your session, code that has executed is highlighted.

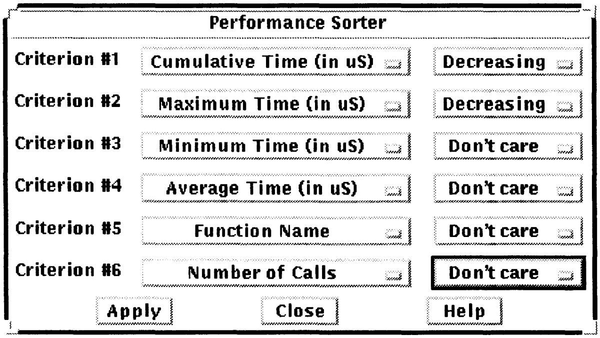


Source File: /svt1/svt/ctdemo/lib/work/peterk/ctdemo.c

```
void
showMemory( char *start, int count )
{
    unsigned char *s;
    int i;

    s = (char *) (((int) start) & ~0xF);
    i = 16 - (((int) start) % 16 );
    while ( count > 0 )
    {
       AMCPrintf( "%08lx: %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
       count -= i;
       s += 16;
       i = 16;
    }
    return;
}

void
userProgram( void )
{
   char *msg = "quick";
   int i = 7;
   ListNode node;
   int *list;
   int count;

   count = rand() % 4;
   list = (int *) malloc( count * sizeof(int));
   switch( rand() % 8 )
   {
        case 0:
        {
           node.count = count;
           node.listHdr = (List *) list;
           node.prev = NULL;
           node.next = (ListNode *) &list;
           showListNode( &node );
           break;
        }
```

Sort utilities

To perform a single-level or multi-level sort on any Continuous mode tabular view, click the Sort button on the view of interest to display a sort dialog.



Performance Sorter

Criterion #1	Cumulative Time (in uS)	Decreasing
Criterion #2	Maximum Time (in uS)	Decreasing
Criterion #3	Minimum Time (in uS)	Don't care
Criterion #4	Average Time (in uS)	Don't care
Criterion #5	Function Name	Don't care
Criterion #6	Number of Calls	Don't care

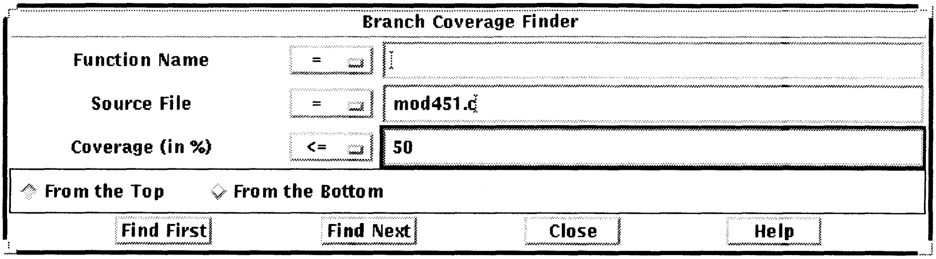
A single-level sort arranges the table's rows in ascending or descending order according to the contents of any one of the columns. A multi-level sort arranges the table rows according to the contents of multiple columns, in hierarchical order.

The list boxes on the left contain the column headings for the active view. The list boxes on the right contain the sort options (Increasing, Decreasing, and Don't Care). In the top pair of list boxes (criterion #1) select the column heading and sort order for your first priority. You are now set for a single-level sort. If you want to perform a multi-level sort, in the second pair of list boxes select the column heading and sort order for your second criterion. You can set sort criteria for as many of the table's columns as you wish. Click Apply to sort the data.

The above example is set to sort the Function Performance view such that the functions with the highest cumulative CPU times will appear at the top of the table. The second level of the sort will order function rows that have the same cumulative CPU time in descending order by maximum execution time.

Search utilities

To perform a search in any Continuous mode tabular view, click the Find button on the view of interest to display a search dialog.



Branch Coverage Finder

Function Name =

Source File = mod451.c

Coverage (in %) <= 50

◆ From the Top ◇ From the Bottom

The left portion of the dialog lists the column headings for the view you are searching. The boxes on the right are for entering your search strings. The list boxes in the middle provide a set of standard relational operators you can apply to each search string:

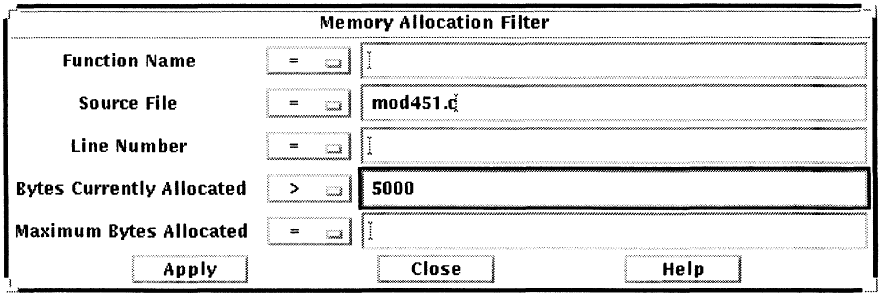
= < > <= >= !=

Note: Selecting the *Like* option enables a set of special characters you can use to build complex search expressions. See "Search and filter expressions" on page 3-30.

You can specify search criteria for as many columns as you wish. In the example above, the dialog is set to search the Branch Coverage view for functions in mod451.c that have achieved 50% coverage or less.

Filter utilities

You can define a filter to display only the data of interest in any Continuous mode tabular view. Click the Filter button on the view of interest to display a filter dialog.



Memory Allocation Filter

Function Name	=	
Source File	=	mod451.c
Line Number	=	
Bytes Currently Allocated	>	5000
Maximum Bytes Allocated	=	

The left side of the dialog lists the column headings of the active view. In the entry fields on the right, you can enter a filter value for each column. The list boxes in the middle provide a set of standard relational operators you can apply to each filter string:

= < > <= >= !=

Note: Selecting the *Like* option enables a set of special characters you can use to build complex filter expressions. See "Search and filter expressions" on page 3-30.

Select any operator to apply to each filter string you enter. You can specify filter criteria for as many of the columns as you wish. The example above is set to filter everything out of the Memory Allocation view except entries for functions in mod451.c that currently have more than 5,000 bytes of memory allocated.

Search and filter expressions

When you select the Like option in the operators list box on any Continuous mode search or filter dialog, a set of special characters are enabled that you can use to form complex search or filter expressions:

+ * ? . [ ] ^ $

Note: A backslash (\) followed by a special character matches the literal character (i.e., this "escapes" the special character).

Expressions that match a single character

To build a *single-character* expression (i.e., an expression that matches a single character in a target string) you can use the literal characters themselves, plus the following special characters:

- ❑ A period (.) matches any single character.
  .umpty matches Humpty or dumpty
- ❑ A set of characters enclosed in brackets [ ] forms an expression that matches any character within the set.
  [akm] matches a, k, or m
  [a-z] matches any lower-case letter
  [^akm] matches any character *except* a, k, or m

Note: The carat (^) looses this negating functionality if it is not the first character in a set. (Also see "Anchoring an expression" on page 3-31" for another use of the carat character.)

Matching a single-character expression to a character in any position within a target string counts as a match. So, the letter "a" by itself matches the words *apple*, *star* and *car*, and [a-c] matches *banana*.

Multi-character expressions

A *multi-character* expression finds a match in a target string only if all of its component single-character expressions find a match. To build multi-character expressions, you can concatenate single-character expressions, and use the following special characters:

- ❑ An expression followed by an asterisk (*) matches zero or more occurrences of that expression.
  a* matches zero or more occurrences of the letter a
  [a-z]* matches zero or more lower-case letters

- ❑ An expression followed by a plus sign (+) matches one or more occurrences of that expression.
  [a-z]+ matches one or more lower-case letters

- ❑ An expression that precedes a question mark (?) can occur zero or one time in the matching string, no more.
  xy?z matches xyz or xz, but not xyyz

Anchoring an expression

You can *anchor* an entire expression, so it will find a match only at the beginning or end of a line:

- ❑ If a caret (^) is at the beginning of the expression, a matching string must be at the beginning of a line.
  ^a matches abc, but not bac
  ^[A-Z] matches any string that begins in a capital letter
  ^[^A-Z] matches any string *not* beginning with a capital

- ❑ If a dollar sign ($) is at the end of the expression, a matching string must be at the end of the line.
  a$ matches Data and Area but not abc
  [A-Z]$ matches any string that ends with a capital letter
  [^A-Z]$ matches any string *not* ending with a capital

Chapter 4

The Trace Tool

Overview

The Trace tool captures and stores in a buffer a trace of events derived from the tag values the probe receives from your target. You can display trace data in several levels of detail, using a variety of display options.

The Trace tool can take snapshots of system activity with no pre-fetch or cache-execution confusion, and can even trace dynamically relocated code.

Making a trace measurement

To make a trace measurement:

1. Prepare your target system according to the guidelines under "Getting Started" in Chapter 2.
2. Download your instrumented program to the target hardware.
3. Start the CodeTEST host application and configure your session for the probe, target program, and licenses you want to use.
4. Select Set Trace Mode on the toolbar Run menu. (Although you cannot make Trace and Continuous mode measurements simultaneously, you can display Trace and Continuous views side-by-side and toggle between modes without losing data.)
5. Click the Trace button to display the Trace window.
6. Click Setup on the Trace window if you want to use the trace setup controls. This is optional. See "Default trace setup" on page 4-6.
7. Click the Start button on the Trace window. CodeTEST now waits for the trigger event. When the trigger event occurs, CodeTEST continues accumulating data until the buffer is filled to the selected depth, then the probe stops automatically and the trace data is displayed.

Trace commands

The following commands are available on the Trace window.

Command	Function
File Menu	
Print...	Displays the dialog for printing data (or printing to a file, which can later be sent to a printer). Only currently displayed data (High Level, Control Flow, or Source view) is printed.
Export...	Displays the dialog for exporting data to an ASCII comma delimited format (CDF) file. Only the currently displayed data (High Level, Control Flow, or Source view) is exported. Exported data can be loaded into most spreadsheet and publishing programs. **Note:** To save data in a form that can be reloaded into CodeTEST, use the Save command on the toolbar File Menu.
Close	Closes the Trace window without losing data.
Window Menu	
Resize...	Displays the dialog for setting column widths.
Help Menu	
Help on Window...	Displays help for the current window.
Contents...	Displays the help Contents page.
Search...	Displays the help Keyword Search dialog.
How to Use Help...	Displays help on how to use the help viewer.
About CodeTEST...	Displays the CodeTEST version number.

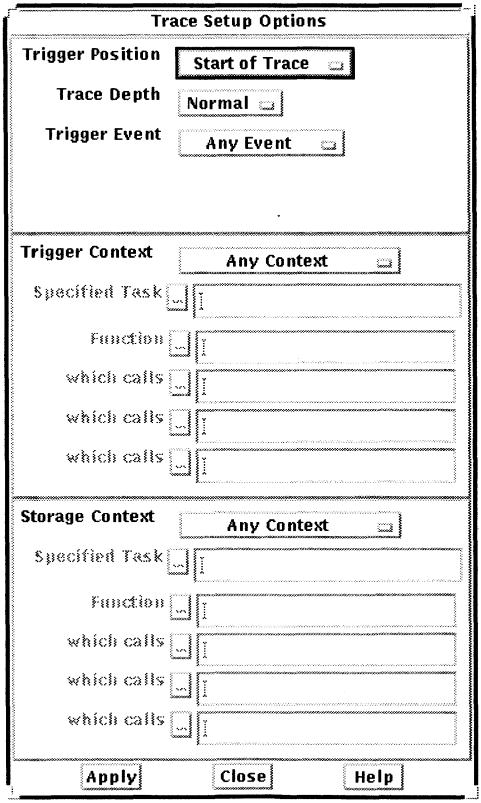
Button Controls

Command	Function
High Level	Displays a view of the trace that shows only function entry and exit points and RTOS events. See "High Level view" on page 4-10.
Control Flow	Displays the Control Flow view of the trace, with all the information in the High Level view plus executed branch points. See "Control Flow view" on page 4-14.
Source	Displays the Source view of the trace, showing every line of source code that has executed. See "Source view" on page 4-16.
Indenting On Indenting Off	Toggles indentation in the source column on and off.
Expand Loops Collapse Loops	Toggles the Control Flow and Source views between expanded (one line per iteration) and collapsed (single line) view of loops.
Elapsed Time Time Intervals	Toggles the information in the Time column between elapsed time (time from the start of the trace to each event) and time intervals (the intervals between events).
Start	Purges any existing trace data, then starts the probe. (To start the probe in Continuous mode, select Start Probe on the Run menu.)
Stop	Stops the probe and displays contents of the trace buffer. (To stop the probe in Continuous mode, select Stop Probe on the Run menu.)
Find...	Displays the Trace Find dialog. See "The Trace Data Finder" on page 4-21.
Setup...	Displays the Trace Setup dialog. See "Trace setup options" on page 4-6.

Trace Setup

The Trace Setup dialog offers a variety of controls that govern when and how a trace is triggered and stored. To display the Trace Setup dialog, click Setup on the Trace window.

Note: To save a setup, turn on macro recording to capture your setup commands. See “Creating a macro” on page 2-21.



Trace Setup Options

Trigger Position: Start of Trace

Trace Depth: Normal

Trigger Event: Any Event

Trigger Context: Any Context

Specified Task

Function

which calls

which calls

which calls

Storage Context: Any Context

Specified Task

Function

which calls

which calls

which calls

Default trace setup

By default, the Trace tool is set up as follows:

- □ Trigger on any event (trace will be initiated by the first tag the probe receives).
- □ Trigger position is at the start of the trace.
- □ Trace depth is normal (up to 4K events).
- □ Trigger will be recognized in any context.
- □ Storage is enabled in any context.

Trace setup options

Trigger position

Option	Definition
Start of Trace	The trigger event is positioned at the beginning of the trace buffer (i.e., trace will show events that follow the trigger). Approximately 20 events are included before the trigger event.
Center of Trace	The trigger event is positioned at the center of the trace buffer (i.e., trace will show a fifty-fifty split of events that precede and follow the trigger).
End of Trace	The trigger event is positioned at the end of the trace buffer (i.e., trace will show events that precede the trigger). Approximately 20 events are included after the trigger event.

Note: The default trigger (Any Event) triggers on the first tag the probe receives. Used in combination with the End of Trace trigger position, this will not produce much data because the buffer will be empty when the trigger event occurs. Likewise, using Any Event and Center of Trace will produce a trace that shows only events following the trigger.

Trace depth

Option	Definition
Normal	Up to 4K events
Deep	Up to 40K events
Very Deep	Up to 400K events

Trigger event

Event	Description
Any Event	Trigger on the first tag the probe receives.
Function Entry	Trigger on entry into any function or a specified function.
	Note: When you select Function Entry or Function Exit, a box is displayed for you to enter a function name. Click ... to make your selection from a list of your target program's instrumented functions (from the IDB).
Function Exit	Trigger on exit from any function or a specified function.
Memory Error	Trigger on the first memory error detected. See Chapter 7 for a list of CodeTEST memory errors.
Memory Allocation	Trigger on the first memory allocation detected.
Memory Deallocation	Trigger on the first memory deallocation detected.
RTOS Task Entry	Trigger on entry into any task or a specified task.
RTOS Task Exit	Trigger on exit from any task or a specified task.

Event	Description
	Note: When you select any of the RTOS options (RTOS users only) a box is displayed for you to enter the task name. Click ... to make your selection from the task list (from your RTOS map file). See Chapter 8 for definitions of these events and a discussion of how CodeTEST tracks tasks.
RTOS Task Creation	Trigger on creation of any task or a specified task.
RTOS Task Deletion	Trigger on deletion of any task or a specified task.
AMCPrintf Call	Trigger on any AMCPrintf (or AMCPuts) call, or on a specified call. When you select this event type, a box is displayed. To trigger on a specific call, enter the first few characters of the call's text output. See "AMCPrintf and AMCPuts" on page 4-17.
AMCUserTag Call	Trigger on any AMCUserTag call or no a specified call. When you select this event type, a box is displayed. To trigger on a specific call, enter the appropriate integer value. See "User Defined Tags" on page 4-20.
No Trigger	Trace is initiated as soon as you start the probe and continues until you stop the probe or until the target program stops executing.

Trigger context

Trigger context is the execution context within which CodeTEST will recognize the specified trigger event.

Option	Definition
Any	Trigger event will be recognized in any context.
Task	For RTOS applications only. Trigger event will be recognized only within the task that you specify.

Option	Definition
Function	Trigger event will be recognized only within the function or function calling sequence that you specify.
Task and function	Trigger event will be recognized only within the task and function or function calling sequence you specify.

Note: If you specify a task-qualified trigger or storage context, CodeTEST does not begin searching the tag stream for a trigger event until after the first task switch is detected. Otherwise, CodeTEST begins searching for the trigger event when the first tag is received.

Storage context

Storage context is the execution context within which the probe will store events in the trace buffer. When one or more qualification criteria are not met, all events are disregarded. The storage may go "on and off" multiple times during a single trace. The elapsed time during a "storage off" period will be reflected in the trace display.

Option	Definition
Any	Trace storage is enabled in any context.
Task	For RTOS applications only. Storage is enabled only within the task that you specify.
Function	Storage is enabled only within the function or function calling sequence that you specify.
Task and function	Storage is enabled only within the task and function or function calling sequence that you specify.

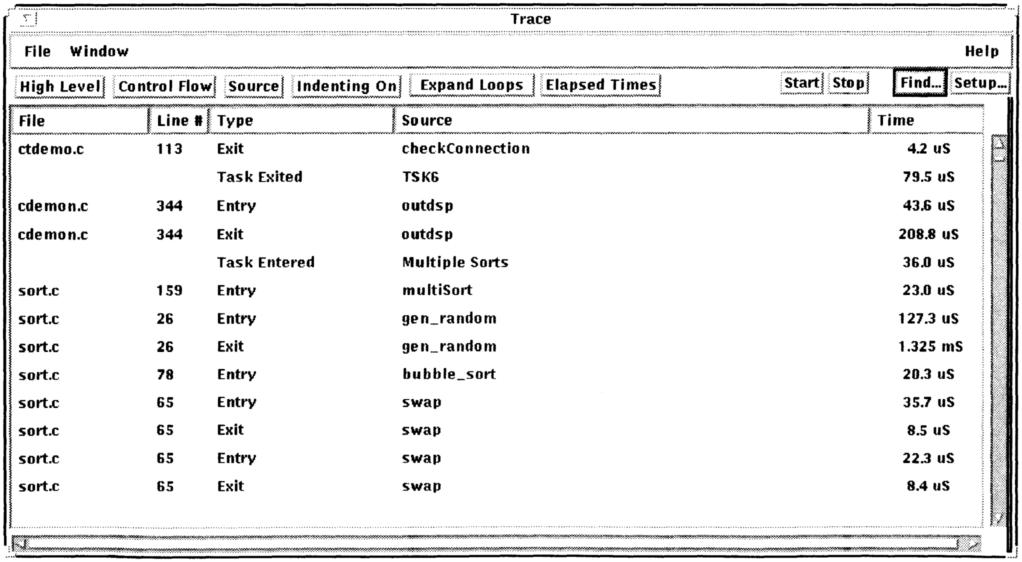
The Trace Window

The Trace window displays the execution history of your target program, with several user-selectable levels of detail and other viewing options available.

Note: The type of data the Trace tool captures is dependent on the level at which you have instrumented your target code (see “Instrumenter Theory Overview” on page 6-2 and “Tagging summary” on page 6-10).

High Level view

The High Level trace view gives you the big picture of execution activity. The example below is a High Level view of a trace captured with the default setup. Two RTOS events and a number of function entries and exits are shown.

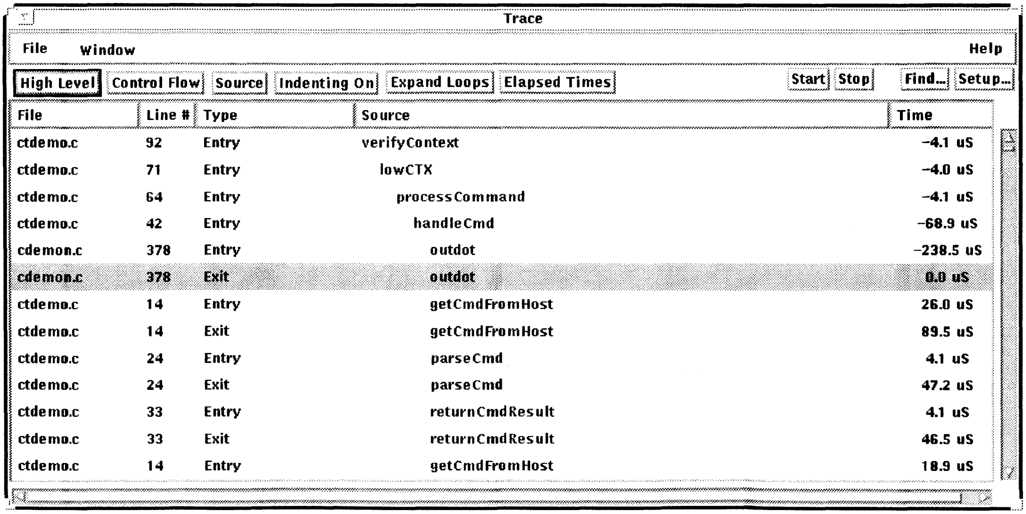


File	Line #	Type	Source	Time
ctdemo.c	113	Exit	checkConnection	4.2 uS
		Task Exited	TSK6	79.5 uS
cdemon.c	344	Entry	outdsp	43.6 uS
cdemon.c	344	Exit	outdsp	208.8 uS
		Task Entered	Multiple Sorts	36.0 uS
sort.c	159	Entry	multiSort	23.0 uS
sort.c	26	Entry	gen_random	127.3 uS
sort.c	26	Exit	gen_random	1.325 mS
sort.c	78	Entry	bubble_sort	20.3 uS
sort.c	65	Entry	swap	35.7 uS
sort.c	65	Exit	swap	8.5 uS
sort.c	65	Entry	swap	22.3 uS
sort.c	65	Exit	swap	8.4 uS

For each displayed event, High Level trace view shows:

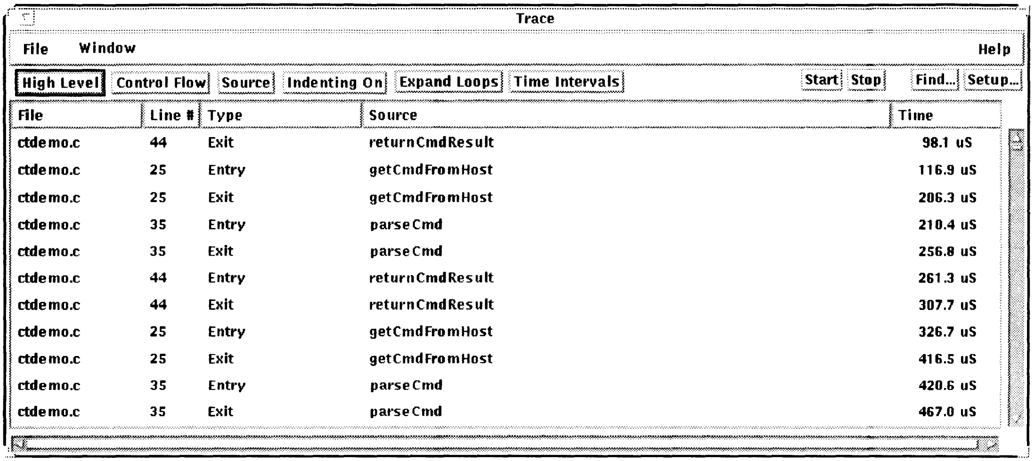
Column	Definition
File	Target program events: source file name RTOS events: blank
Line	Target program events: source file line number RTOS events: blank
Type	Target program events: function entry and exit RTOS events: task creation, entry, exit, and deletion
Source	Target program events: function name RTOS events: task name or string from RTOS map file
Time	Elapsed time from the start of the trace to each event, or the time intervals between events.

The trigger event (exit from function *outdot* in the example below) is highlighted, and shows zero in the Time column. Event times before the trigger are shown as negative numbers; event times after the trigger as positive numbers.



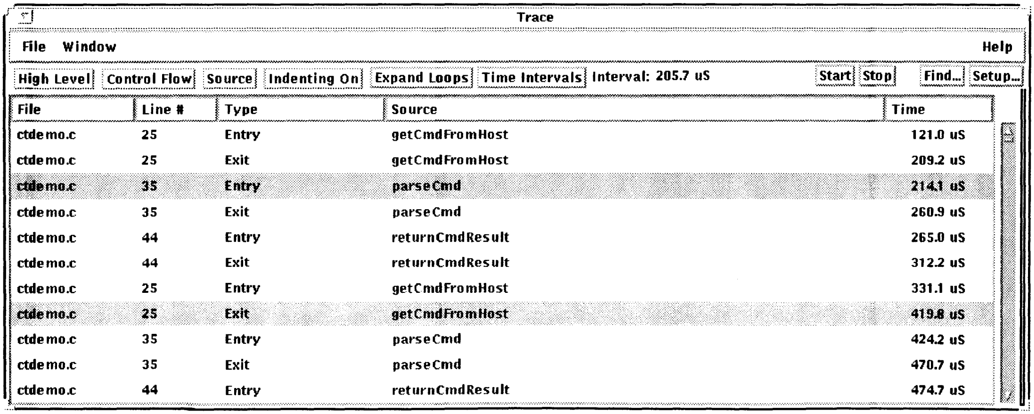
File	Line #	Type	Source	Time
ctdemo.c	92	Entry	verifyContext	-4.1 uS
ctdemo.c	71	Entry	lowCTX	-4.0 uS
ctdemo.c	64	Entry	processCommand	-4.1 uS
ctdemo.c	42	Entry	handleCmd	-68.9 uS
cdemon.c	378	Entry	outdot	-238.5 uS
cdemon.c	378	Exit	outdot	0.0 uS
ctdemo.c	14	Entry	getCmdFromHost	26.0 uS
ctdemo.c	14	Exit	getCmdFromHost	89.5 uS
ctdemo.c	24	Entry	parseCmd	4.1 uS
ctdemo.c	24	Exit	parseCmd	47.2 uS
ctdemo.c	33	Entry	returnCmdResult	4.1 uS
ctdemo.c	33	Exit	returnCmdResult	46.5 uS
ctdemo.c	14	Entry	getCmdFromHost	18.9 uS

By default, the time column shows the intervals between events. Click *Elapsed Time* to see the total elapsed time from the trigger event to each successive event.



File	Line #	Type	Source	Time
ctdemo.c	44	Exit	returnCmdResult	98.1 uS
ctdemo.c	25	Entry	getCmdFromHost	116.9 uS
ctdemo.c	25	Exit	getCmdFromHost	206.3 uS
ctdemo.c	35	Entry	parseCmd	210.4 uS
ctdemo.c	35	Exit	parseCmd	256.8 uS
ctdemo.c	44	Entry	returnCmdResult	261.3 uS
ctdemo.c	44	Exit	returnCmdResult	307.7 uS
ctdemo.c	25	Entry	getCmdFromHost	326.7 uS
ctdemo.c	25	Exit	getCmdFromHost	416.5 uS
ctdemo.c	35	Entry	parseCmd	420.6 uS
ctdemo.c	35	Exit	parseCmd	467.0 uS

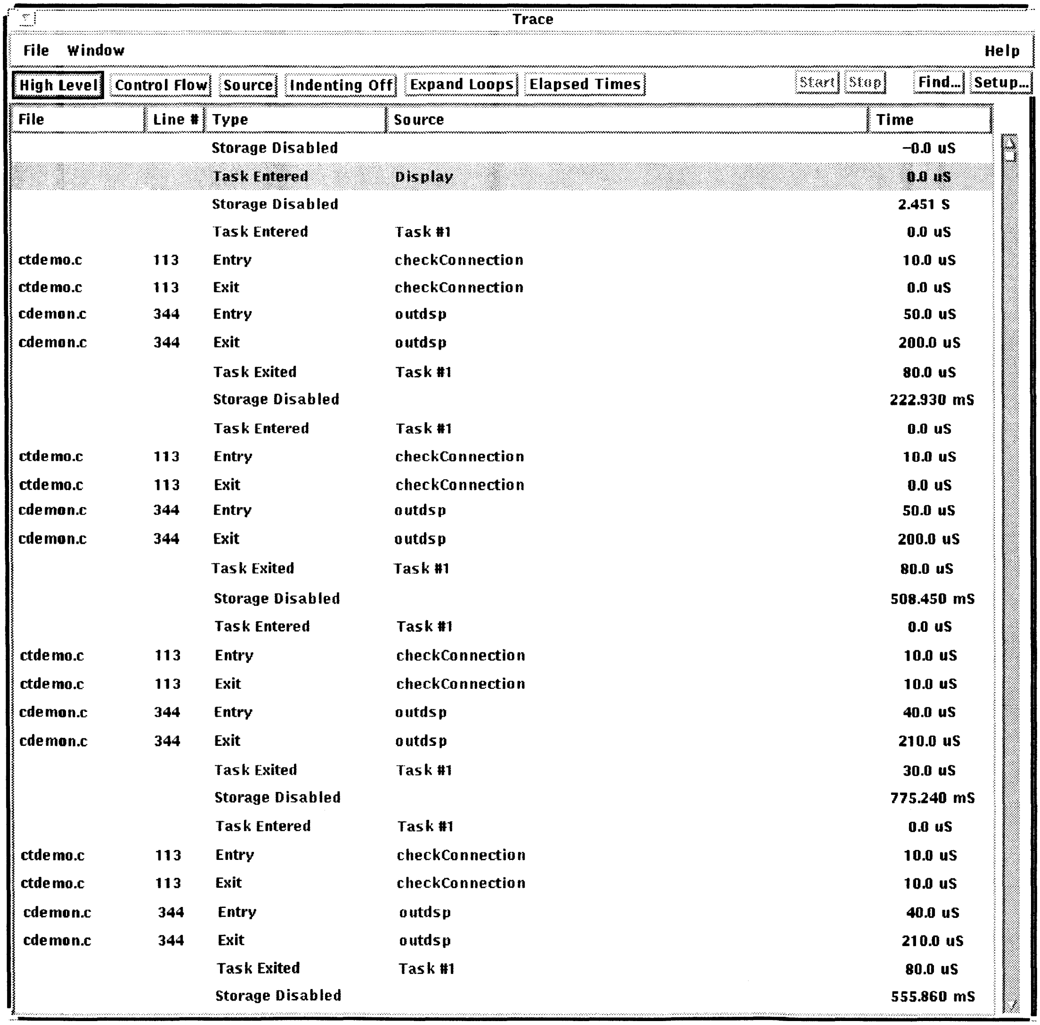
To see the time interval between any two events, left mouse-button click an event to place the first marker, then middle mouse-button click a second event to place the second marker. The interval between marked events is displayed in the Trace window's toolbar.



Interval: 205.7 uS

File	Line #	Type	Source	Time
ctdemo.c	25	Entry	getCmdFromHost	121.0 uS
ctdemo.c	25	Exit	getCmdFromHost	209.2 uS
ctdemo.c	35	Entry	parseCmd	214.1 uS
ctdemo.c	35	Exit	parseCmd	260.9 uS
ctdemo.c	44	Entry	returnCmdResult	265.0 uS
ctdemo.c	44	Exit	returnCmdResult	312.2 uS
ctdemo.c	25	Entry	getCmdFromHost	331.1 uS
ctdemo.c	25	Exit	getCmdFromHost	419.8 uS
ctdemo.c	35	Entry	parseCmd	424.2 uS
ctdemo.c	35	Exit	parseCmd	470.7 uS
ctdemo.c	44	Entry	returnCmdResult	474.7 uS

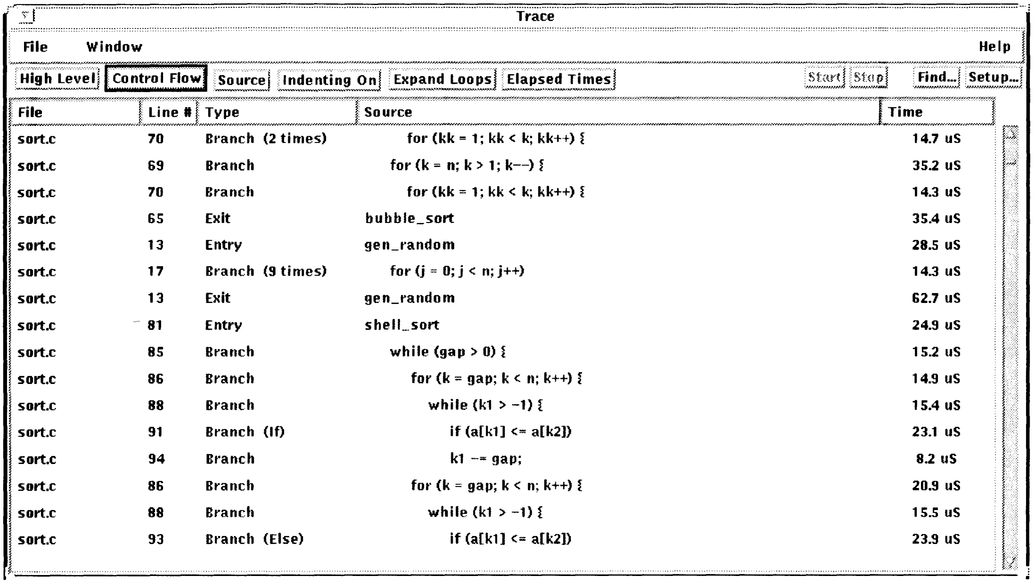
For this example, the storage context was qualified to a task named "Task #1." Note the "Storage Disabled" event lines in the resulting trace, showing target execution time outside Task #1.



File	Line #	Type	Source	Time
		Storage Disabled		-0.0 uS
		Task Entered	Display	0.0 uS
		Storage Disabled		2.451 S
		Task Entered	Task #1	0.0 uS
ctdemo.c	113	Entry	checkConnection	10.0 uS
ctdemo.c	113	Exit	checkConnection	0.0 uS
cdemon.c	344	Entry	outdsp	50.0 uS
cdemon.c	344	Exit	outdsp	200.0 uS
		Task Exited	Task #1	80.0 uS
		Storage Disabled		222.930 mS
		Task Entered	Task #1	0.0 uS
ctdemo.c	113	Entry	checkConnection	10.0 uS
ctdemo.c	113	Exit	checkConnection	0.0 uS
cdemon.c	344	Entry	outdsp	50.0 uS
cdemon.c	344	Exit	outdsp	200.0 uS
		Task Exited	Task #1	80.0 uS
		Storage Disabled		508.450 mS
		Task Entered	Task #1	0.0 uS
ctdemo.c	113	Entry	checkConnection	10.0 uS
ctdemo.c	113	Exit	checkConnection	10.0 uS
cdemon.c	344	Entry	outdsp	40.0 uS
cdemon.c	344	Exit	outdsp	210.0 uS
		Task Exited	Task #1	30.0 uS
		Storage Disabled		775.240 mS
		Task Entered	Task #1	0.0 uS
ctdemo.c	113	Entry	checkConnection	10.0 uS
ctdemo.c	113	Exit	checkConnection	10.0 uS
cdemon.c	344	Entry	outdsp	40.0 uS
cdemon.c	344	Exit	outdsp	210.0 uS
		Task Exited	Task #1	80.0 uS
		Storage Disabled		555.860 mS

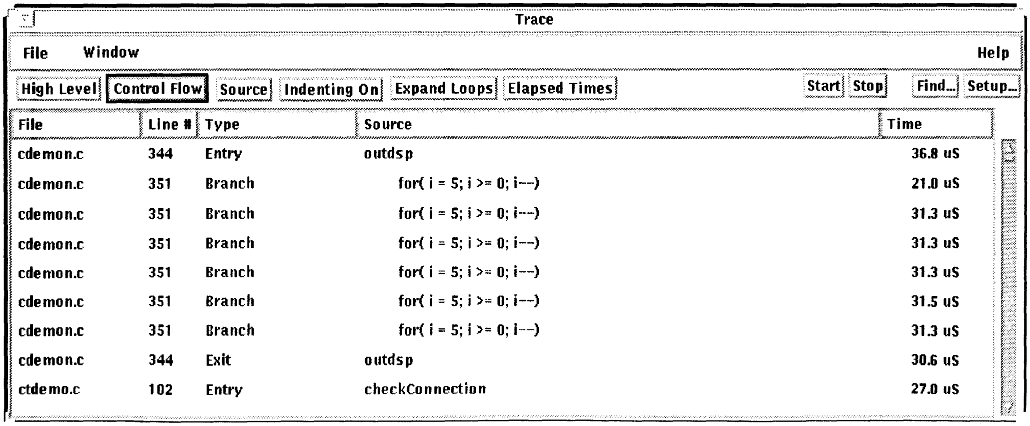
Control Flow view

Click *Control Flow* to add the dimension of executed branch points to your view of the trace. This example shows loops collapsed. Note the indicators showing the number of loop iterations and path taken at decision points.



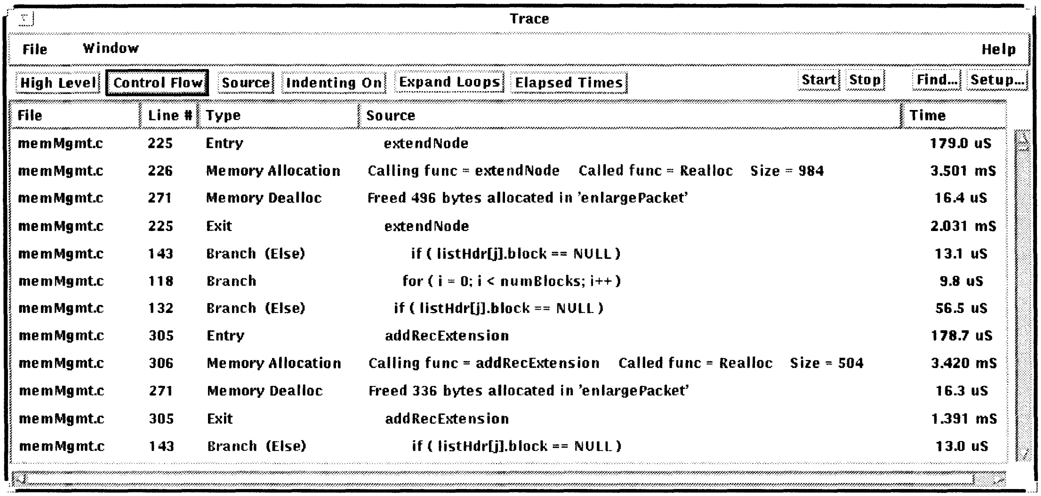
File	Line #	Type	Source	Time
sort.c	70	Branch (2 times)	for (kk = 1; kk < k; kk++) {	14.7 uS
sort.c	69	Branch	for (k = n; k > 1; k--) {	35.2 uS
sort.c	70	Branch	for (kk = 1; kk < k; kk++) {	14.3 uS
sort.c	65	Exit	bubble_sort	35.4 uS
sort.c	13	Entry	gen_random	28.5 uS
sort.c	17	Branch (9 times)	for (j = 0; j < n; j++)	14.3 uS
sort.c	13	Exit	gen_random	62.7 uS
sort.c	81	Entry	shell_sort	24.9 uS
sort.c	85	Branch	while (gap > 0) {	15.2 uS
sort.c	86	Branch	for (k = gap; k < n; k++) {	14.9 uS
sort.c	88	Branch	while (k1 > -1) {	15.4 uS
sort.c	91	Branch (If)	if (a[k1] <= a[k2])	23.1 uS
sort.c	94	Branch	k1 -= gap;	8.2 uS
sort.c	86	Branch	for (k = gap; k < n; k++) {	20.9 uS
sort.c	88	Branch	while (k1 > -1) {	15.5 uS
sort.c	93	Branch (Else)	if (a[k1] <= a[k2])	23.9 uS

Expand loops to show each iteration as an event line.



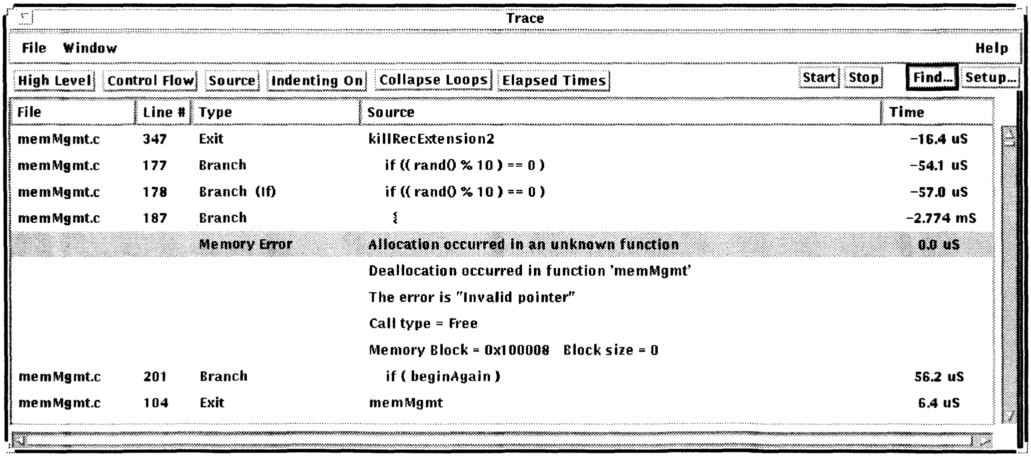
File	Line #	Type	Source	Time
cdemon.c	344	Entry	outdsp	36.8 uS
cdemon.c	351	Branch	for(i = 5; i >= 0; i--)	21.0 uS
cdemon.c	351	Branch	for(i = 5; i >= 0; i--)	31.3 uS
cdemon.c	351	Branch	for(i = 5; i >= 0; i--)	31.3 uS
cdemon.c	351	Branch	for(i = 5; i >= 0; i--)	31.3 uS
cdemon.c	351	Branch	for(i = 5; i >= 0; i--)	31.5 uS
cdemon.c	351	Branch	for(i = 5; i >= 0; i--)	31.3 uS
cdemon.c	344	Exit	outdsp	30.6 uS
ctdemo.c	102	Entry	checkConnection	27.0 uS

If your code is instrumented for memory monitoring, Control Flow view will show any memory events that occurred.



File	Line #	Type	Source	Time
memMgmt.c	225	Entry	extendNode	179.0 uS
memMgmt.c	226	Memory Allocation	Calling func = extendNode Called func = Realloc Size = 984	3.501 mS
memMgmt.c	271	Memory Dealloc	Freed 496 bytes allocated in 'enlargePacket'	16.4 uS
memMgmt.c	225	Exit	extendNode	2.031 mS
memMgmt.c	143	Branch (Else)	if (listHdr[j].block == NULL)	13.1 uS
memMgmt.c	118	Branch	for (i = 0; i < numBlocks; i++)	9.8 uS
memMgmt.c	132	Branch (Else)	if (listHdr[j].block == NULL)	56.5 uS
memMgmt.c	305	Entry	addRecExtension	178.7 uS
memMgmt.c	306	Memory Allocation	Calling func = addRecExtension Called func = Realloc Size = 504	3.420 mS
memMgmt.c	271	Memory Dealloc	Freed 336 bytes allocated in 'enlargePacket'	16.3 uS
memMgmt.c	305	Exit	addRecExtension	1.391 mS
memMgmt.c	143	Branch (Else)	if (listHdr[j].block == NULL)	13.0 uS

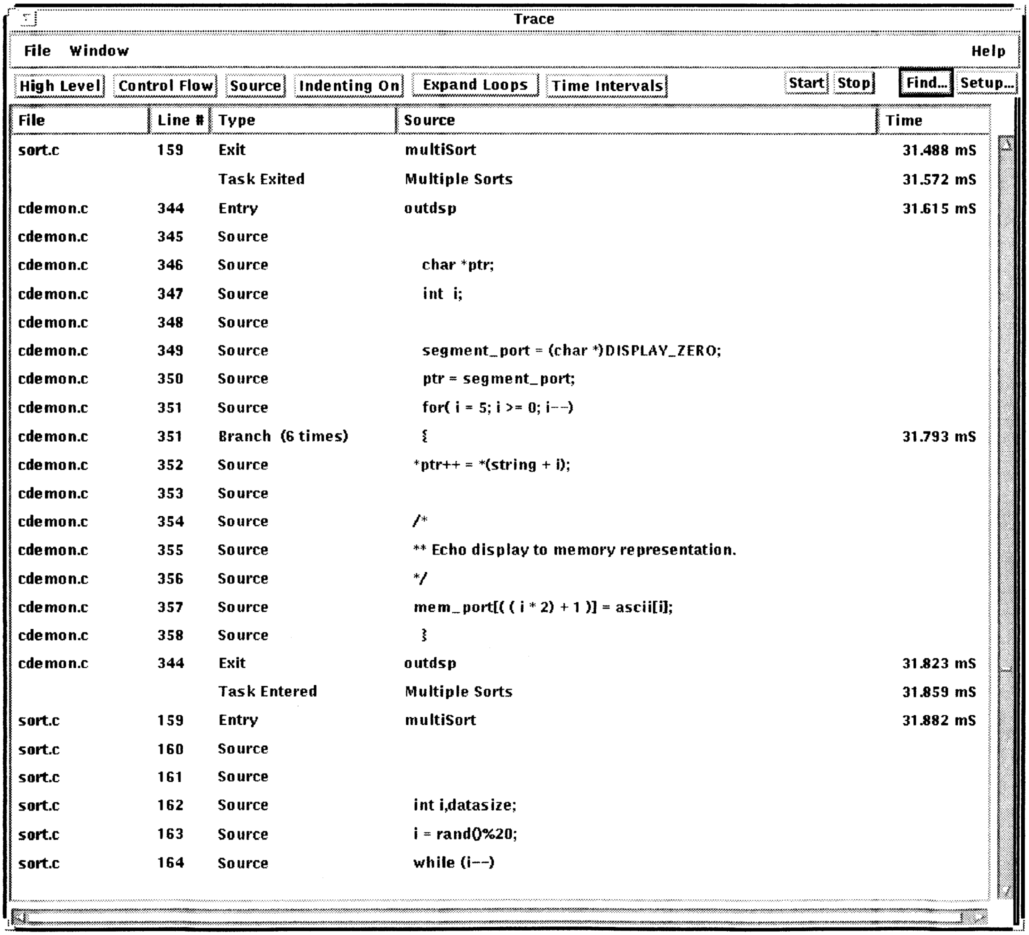
In this example, the trigger event was set to Memory Error. The resulting trace shows the details of an invalid pointer error that occurred.



File	Line #	Type	Source	Time
memMgmt.c	347	Exit	killRecExtension2	-16.4 uS
memMgmt.c	177	Branch	if ((rand() % 10) == 0)	-54.1 uS
memMgmt.c	178	Branch (If)	if ((rand() % 10) == 0)	-57.0 uS
memMgmt.c	187	Branch	{	-2.774 mS
		Memory Error	Allocation occurred in an unknown function	0.0 uS
			Deallocation occurred in function 'memMgmt'	
			The error is "Invalid pointer"	
			Call type = Free	
			Memory Block = 0x100008 Block size = 0	
memMgmt.c	201	Branch	if (beginAgain)	56.2 uS
memMgmt.c	104	Exit	memMgmt	6.4 uS

Source view

Click *Source* to reveal the target program's complete execution history. Note that time information is shown only for the actual *events* in the trace buffer (derived from the tag values received from the target). The displayed source lines are gathered from the files you specified in the Configuration dialog. If CodeTEST cannot find some or all of your source files, an error message is displayed when you switch to this view.



File	Line #	Type	Source	Time
sort.c	159	Exit	multiSort	31.488 mS
		Task Exited	Multiple Sorts	31.572 mS
cdemon.c	344	Entry	outdsp	31.615 mS
cdemon.c	345	Source		
cdemon.c	346	Source	char *ptr;	
cdemon.c	347	Source	int i;	
cdemon.c	348	Source		
cdemon.c	349	Source	segment_port = (char *)DISPLAY_ZERO;	
cdemon.c	350	Source	ptr = segment_port;	
cdemon.c	351	Source	for(i = 5; i >= 0; i--)	
cdemon.c	351	Branch (6 times)	{	31.793 mS
cdemon.c	352	Source	*ptr++ = *(string + i);	
cdemon.c	353	Source		
cdemon.c	354	Source	/*	
cdemon.c	355	Source	** Echo display to memory representation.	
cdemon.c	356	Source	*/	
cdemon.c	357	Source	mem_port[((i * 2) + 1)] = ascii[i];	
cdemon.c	358	Source	}	
cdemon.c	344	Exit	outdsp	31.823 mS
		Task Entered	Multiple Sorts	31.859 mS
sort.c	159	Entry	multiSort	31.882 mS
sort.c	160	Source		
sort.c	161	Source		
sort.c	162	Source	int i,datasize;	
sort.c	163	Source	i = rand()%20;	
sort.c	164	Source	while (i--)	

AMCPrintf and AMCPuts

The *AMCPrintf* and *AMCPuts* functions are respectively analogous to the standard C *printf* and *puts* library functions. You can place AMCPrintf function calls in your target source code to write printf-formatted output to the trace buffer, or use AMCPuts calls as a faster alternative for writing simple character strings. The output of these functions will be displayed in the trace Control Flow and Source views. Calls to AMCPrintf and AMCPuts are disregarded when you run CodeTEST in Continuous mode.

To "disable" calls to these functions without removing them from your code, use the *-Xremove-calls-to=AMCPrintf* or *-Xremove-calls-to=AMCPuts* instrumenter option.

AMCPrintf Syntax

For C code:

```
void AMCPrintf(const char *, ...);
```

For C++ code:

```
extern "C" void AMCPrintf(const char *, ...);
```

AMCPuts Syntax

For C code:

```
void AMCPuts(const char *);
```

For C++ code:

```
extern "C" void AMCPuts(const char *);
```

Description

AMCPrintf accepts the same format specifiers and arguments as your compiler's printf function. AMCPrintf uses either stdarg.h or varargs.h macros and your compiler's vsprintf library function to perform formatting. AMCPuts takes a single argument, which must be a character string.

Support files

To use the AMCPrintf and AMCPuts functions, compile and link these files with your target code:

```
$AMC_HOME/lib/printf/ctprintf.c
$AMC_HOME/lib/printf/ctprintf.h
```

Note: Do not instrument ctprintf.c with the instrumenter.

Limits

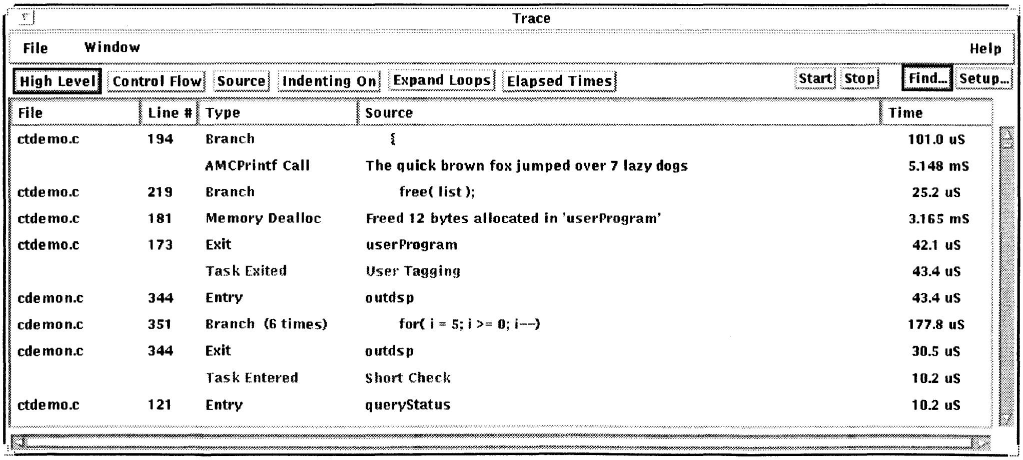
AMCPrintf can handle a single string up to 1024 bytes in length. Formatted strings longer than 1024 bytes may cause target memory corruption. You can modify ctprintf.c to enlarge this limit if you wish. See comments in the file for details.

The Trace window can only display approximately 70 characters per line using the default screen font. Use newlines (\n) to break longer strings into multiple lines.

Examples

The AMCPrintf event in the trace below was produced by:

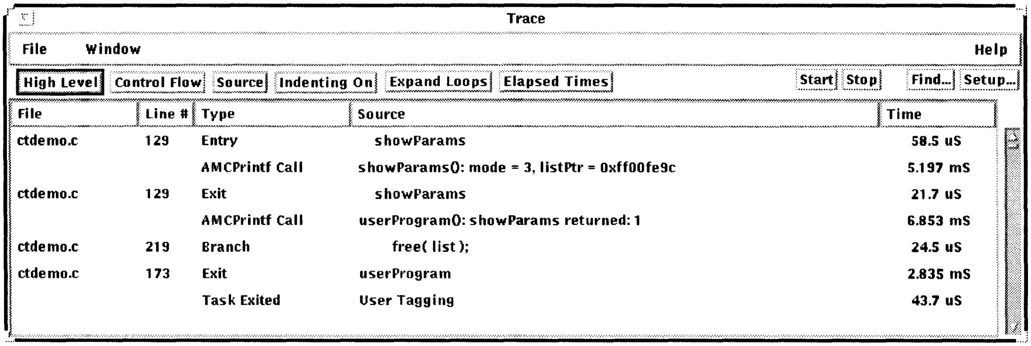
```
AMCPuts( "The quick brown fox jumped over 7
lazy dogs\n" );
```



File	Line #	Type	Source	Time
ctdemo.c	194	Branch	{	101.0 uS
		AMCPrintf Call	The quick brown fox jumped over 7 lazy dogs	5.148 mS
ctdemo.c	219	Branch	free(list);	25.2 uS
ctdemo.c	181	Memory Dealloc	Freed 12 bytes allocated in 'userProgram'	3.165 mS
ctdemo.c	173	Exit	userProgram	42.1 uS
		Task Exited	User Tagging	43.4 uS
cdemon.c	344	Entry	outdsp	43.4 uS
cdemon.c	351	Branch (6 times)	for(i = 5; i >= 0; i--)	177.8 uS
cdemon.c	344	Exit	outdsp	30.5 uS
		Task Entered	Short Check	10.2 uS
ctdemo.c	121	Entry	queryStatus	10.2 uS

Here AMCPrintf displays the values of passed parameters:

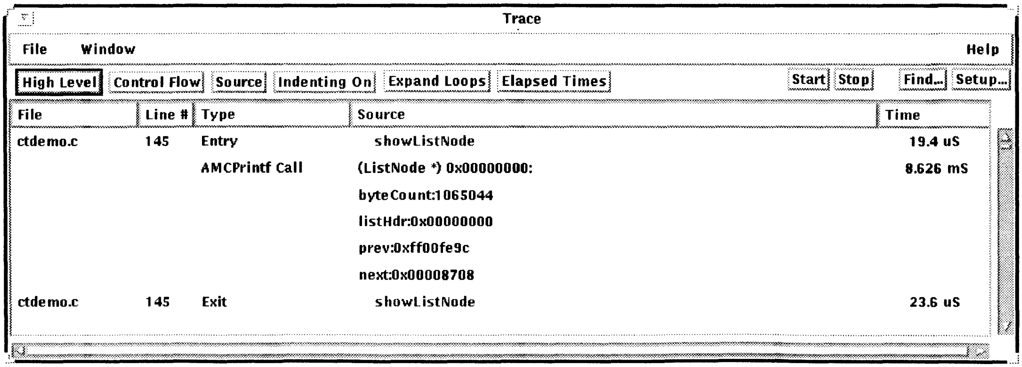
```
showParams( int mode, int *ptr )

AMCPrintf( "showParams(): mode = %d, listPtr =
0x%08lx", mode, ptr );
```



File	Line #	Type	Source	Time
ctdemo.c	129	Entry	showParams	58.5 uS
		AMCPrintf Call	showParams(): mode = 3, listPtr = 0xff00fe9c	5.197 mS
ctdemo.c	129	Exit	showParams	21.7 uS
		AMCPrintf Call	userProgram(): showParams returned: 1	6.853 mS
ctdemo.c	219	Branch	free(list);	24.5 uS
ctdemo.c	173	Exit	userProgram	2.835 mS
		Task Exited	User Tagging	43.7 uS

This AMCPrintf call shows the contents of a structure:

```
AMCPrintf( "(ListNode *) 0x%08lx:\n\
\tbyteCount:\t%ld\n\
\tlistHdr:\t0x%08lx\n\
\tprev:\t0x%08lx\n\
\tnext:\t0x%08lx\n", node->count,
node->listHdr, node->prev, node->next );
```



File	Line #	Type	Source	Time
ctdemo.c	145	Entry	showListNode	19.4 uS
		AMCPrintf Call	(ListNode *) 0x00000000: byteCount:1065044 listHdr:0x00000000 prev:0xff00fe9c next:0x00008708	8.626 mS
ctdemo.c	145	Exit	showListNode	23.6 uS

User Defined Tags

To manually flag events of interest, you can place calls to the AMCUserTag function in your source code. When you compile and run the program, your manually-tagged events will be reflected in the Control Flow and Source trace views. User defined tags are disregarded when you run CodeTEST in Continuous mode.

To "disable" calls to AMCUserTag without removing them from your code, use the *-Xremove-calls-to=AMCUserTag* instrumenter option.

User Defined Tag file

To map each AMCUserTag call to a text string, you can create a User Defined Tag file and enter the file name in the Configuration Options dialog (see page 2-16).

```
    1      "Set up structure"
    2      "Add new nodes"
    3      "Begin read data loop"
    4      "Handle out-of-memory error"
```

Placing user defined tags

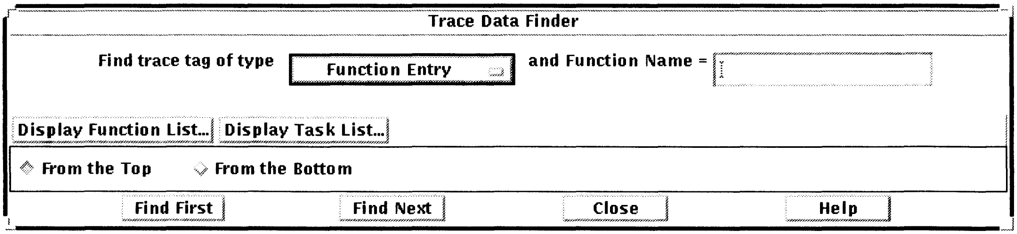
To place a user defined tag in your code, insert a function call to AMCUserTag at the point where you want the text string to be displayed. AMCUserTag takes a single argument: an integer value (which can be mapped to a string in your User Defined Tag file). The minimum value is 1; maximum is 1048575. To use hexadecimal, prefix each value with 0x. Values without this prefix are interpreted as decimal. For example:

```
AMCUserTag(3)
```

will cause the instrumenter to place a proper CodeTEST tag in your code, which will ultimately cause the text line "Begin read data loop" to be displayed in trace.

The Trace Data Finder

Click *Find* on the Trace window to display the Trace Data Finder and search the trace buffer for specific events.



Trace Data Finder

Find trace tag of type: Function Entry and Function Name =

Trace search options

Option	Definition
Function entry	Search for function entry events, for any function or a function you specify. **Note:** When you select Function Entry or Exit, a box is displayed for you to enter a function name. Click ... to make your selection from a list of your target program's instrumented functions (from the IDB).
Function exit	Search for function exit events, for any function or a function you specify.
Memory allocation	Search for events that reflect calls to the CodeTEST allocation routines. See "CodeTEST Memory Management Routines" on page 7-2.
Memory deallocation	Search for events that reflect calls to the CodeTEST deallocation routines. See "CodeTEST Memory Management Routines" on page 7-2.

Option	Definition
Memory error	Search for memory error events. See “Error codes and messages” on page 7-8 for information about memory error conditions CodeTEST can detect.
	Note: When you select any of the following RTOS options, a box is displayed for you to enter the task name. Click ... to make your selection from the task list (from your RTOS map file).
RTOS task entry	Search for RTOS task entry events, for any task or a task you specify.
RTOS task exit	Search for RTOS task exit events, for any task or a task you specify.
RTOS task created	Search for RTOS task creation events, for any task or a task you specify.
RTOS task deleted	Search for RTOS task exit deletion, for any task or a task you specify.
Storage disabled	Search for a gap in the trace during which no tags were stored due to a storage context qualification.
Unknown tag	Search for an event associated with an unrecognized tag type. When you select this option, a tag value entry box is displayed for you to enter the hex value of the unknown tag.
Processing limit	Search for events that indicate a period when the tag rate from the target exceeded the probe’s processing limit (i.e., a FIFO buffer overflow). The time tags are ignored is indicated in the Time column.
Capture limit	Search for events that indicate a period during which the tag rate from the target exceeded the probe’s capture rate. The time shown for these events indicates how long tags are ignored.

Option	Definition
Target reset	Search for an event that indicates a reset of the target hardware.
AMCPrintf Call	Search for any AMCPrintf (or AMCPuts) call, or a specified call. When you select this event type, a box is displayed. To search for a specific call, enter the first few characters of the call's text output. See "AMCPrintf and AMCPuts" on page 4-17.
AMCUserTag Call	Search for any AMCUserTag call or a specified call. When you select this event type, a box is displayed. To search for a specific call, enter the appropriate integer value. See "User Defined Tags" on page 4-20.
Trigger position	Search for the trigger event.

Chapter 5

The CodeTEST Compiler Driver

Overview

The CodeTEST compiler driver (*ctcc* for C compilers, *ctc++* for C++ compilers) is designed to incorporate the instrumentation of your target source code and linking of CodeTEST support libraries into your build procedures. In most cases configuring ctcc/ctc++ and introducing it into your build routine is relatively simple and straightforward, and once you have things set up, it's generally not something you'll need to change very often.

Note: Aside from minor differences for language support, ctcc and ctc++ are functionally identical and will be treated here as a single utility.

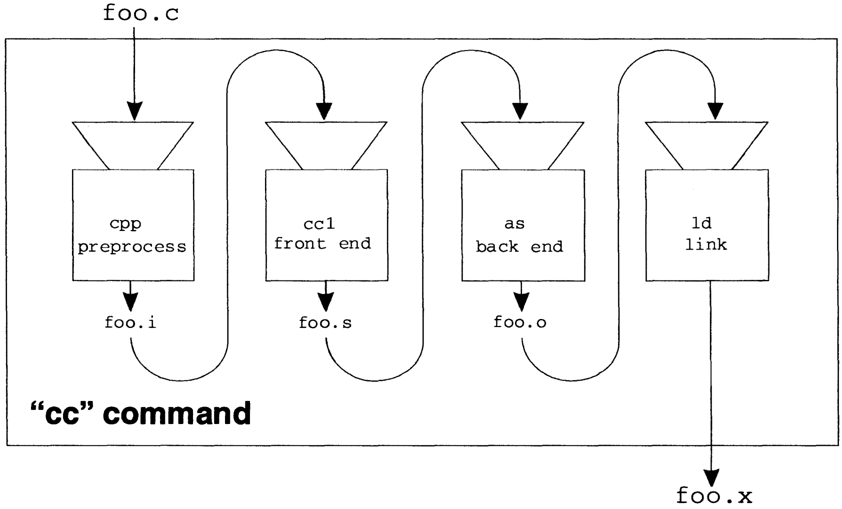
To use ctcc/ctc++, you first need to edit one of the supplied configuration templates to create a target-specific configuration file. The ctcc/ctc++ configuration file defines a set of variables that specify how things are named, where files are located, etc. Once you've created a configuration file, using ctcc/ctc++ can be as succinct as substituting ctcc/ ctc++ for your compiler's own driver or "cc" command. For example, if you currently use gcc, the driver for the GNU C compiler, and your build procedure is based on use of a makefile, you need only substitute CC=ctcc for CC=gcc in your makefile. (If you use g++, the driver for the GNU C++ compiler, you would substitute ctc++.)

The primary advantage of using ctcc/ctc++ is the ease with which you can incorporate instrumentation into your build procedure. A potential drawback is the assumption that you will want to instrument every file your makefile compiles. Excluding individual files from instrumentation would require explicit rules to direct those files to your compiler's driver instead of ctcc/ctc++.

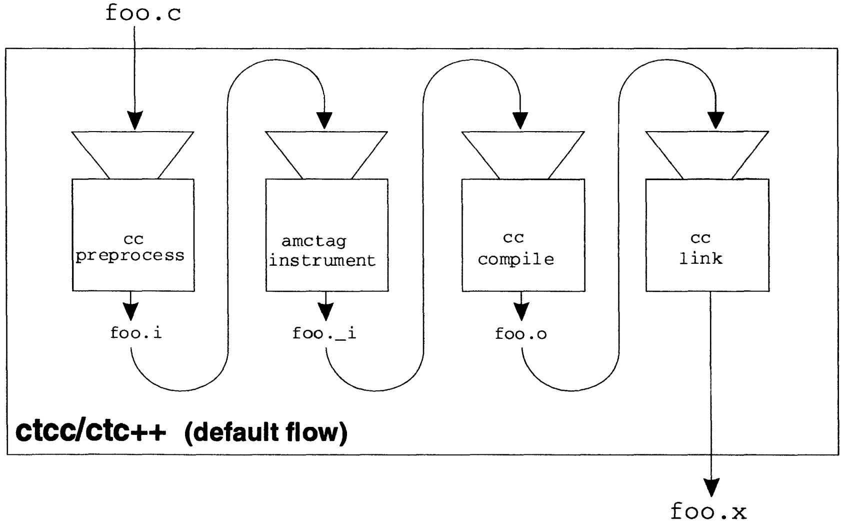
Approaches to instrumentation

The ctcc/ctc++ command flow

The usual C compiler driver command flow is:



Substituting ctcc/ctc++ for your "cc" changes the flow to:

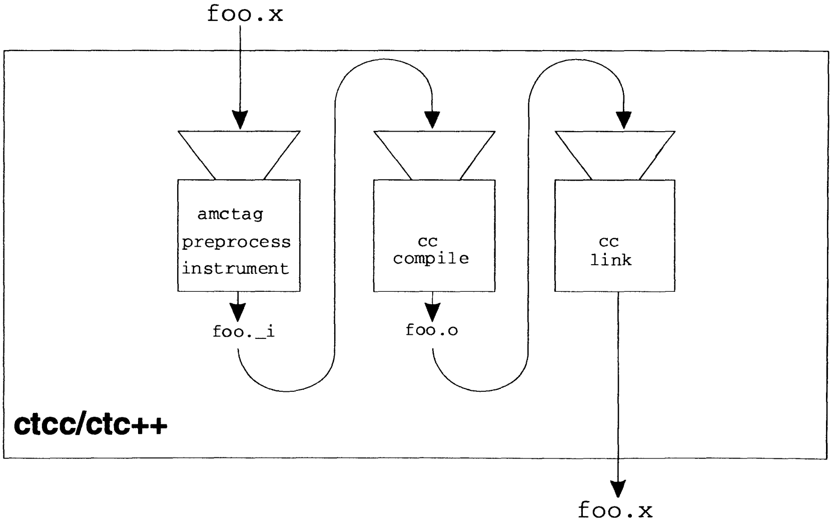


By default, ctcc/ctc++ first invokes your compiler's driver to perform C/C++ preprocessing, then calls the instrumenter (amctag) to insert CodeTEST tags into your preprocessed code, and then sends the instrumented code back to your compiler's driver to complete the build.

However, to support many implementations of C and C++, as well as differences in environment, build methods, etc., ctcc/ctc++ is a highly configurable utility, flexible enough to provide a variety of ways you can go about incorporating instrumentation into your build. The following are just a few examples of the many ways ctcc/ctc++ can be configured.

Preprocessing with amctag

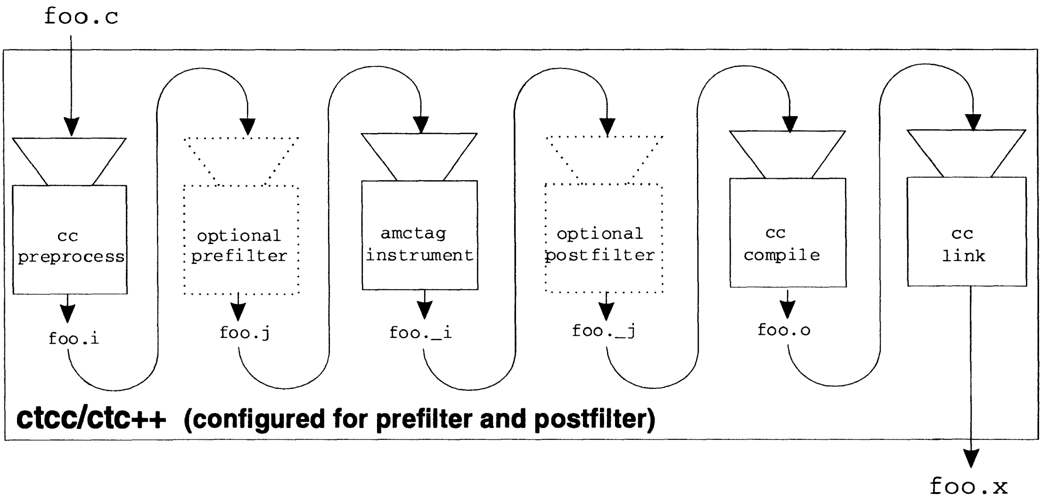
The instrumenter will automatically perform any C/C++ preprocessing that has not already been done. In some cases, you may want amctag to preprocess as well as instrument your code. The *-Xamctag-cpp* instrumenter option changes the ctcc/ctc++ flow to:



Customizing the ctcc/ctc++ command flow

In some cases, it may be necessary to tailor the compiler driver by changing the actual syntax ctcc/ctc++ uses to build the command lines that are executed at each successive stage of the build. (i.e., preprocessing, instrumentation, and compilation). A special set of ctcc/ctc++ command variables are available for this purpose, which you can define in your configuration files or in the environment.

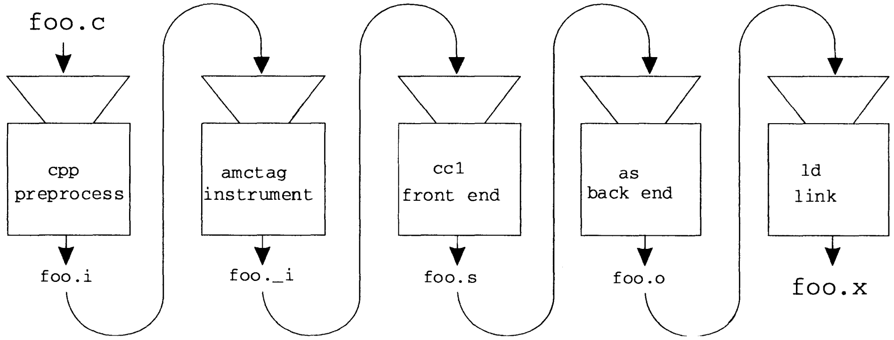
In some cases, it may be necessary to perform minor filtering of your target source files, before or after instrumentation. If so, ctcc/ctc++ can be configured to include a prefilter and/or postfilter step.



Note: These and other ctcc/ctc++ customizing capabilities are provided to ensure compatibility. Most users will never need these features. See Appendix C for details.

Invoking amctag directly

An alternative to using ctcc/ctc++ is to run amctag directly to produce a set of instrumented sources, then route those sources through your usual build procedure. Procedures and examples are covered in Chapter 6.



This approach can also be quite straightforward, though in some cases it requires a bit of effort to set up. Instead of defining configuration variables and letting ctcc/ctc++ handle things, you need to use explicit command line switches to control each instrumentation. A potential drawback to this method is that your existing build procedures may require considerable modification. In most cases, this should be fairly easy; in the worst case you would need to insert a new step to instrument each file before compiling. The more explicit compilation rules your build procedure has, the less concise this approach tends to be.

Note: Running amctag directly can be particularly useful when you want to instrument a small amount of code, such as when you are first getting started with CodeTEST or when you are setting up a new target for testing.

Configuring ctcc/ctc++

The ctcc/ctc++ compiler driver is configured by variables defined in your environment and in a configuration file.

Note: CodeTEST definitions for C and C++ are provided in files amc_c.h and amc_cxx.h installed under $AMC_HOME/lib. Refer to the comments in those files for edits that may be required for your environment.

Environment variables

Before you run ctcc/ctc++, the following variables must be set in the environment.

Variable	Description
AMC_HOME	The home directory of the CodeTEST software installation.
AMC_TARGET	The target-specific portion of an instrumenter configuration file name or include-file path. See "Using ctcc/ctc++ Configuration Files" on page 5-13.

Configuration variables

The following variables, if needed, should be defined in a configuration file. Note that these variables *must* be defined if ctcc/ctc++ will be asked to perform the specific tasks that require them. For information about the naming of configuration files and the paths ctcc/ctc++ searches for them, see "Using ctcc/ctc++ Configuration Files" on page 5-13. For definitions of the instrumenter options referenced here, see "Instrumenter Options" on page 6-16.

Variable	Description
AMC_CPP	The name of your C/C++ preprocessor.
AMC_CC	The name of your C compiler's driver command. This is the default name ctcc will invoke to preprocess and compile your target C source files.
AMC_CXX	The name of your C++ compiler's driver command. This is the default name ctc++ will invoke to preprocess and compile your target C++ sources.
AMC_RCFILE	The base name of your ctcc/ctc++ configuration file(s). Default: .ctccrc See "Using ctcc/ctc++ Configuration Files" on page 5-13.
AMC_INSTRUMENTER	The path and name of the CodeTEST source code instrumenter. Default: $AMC_HOME\bin\amctag.exe
AMC_TAGDEFAULTS	The default instrumenter options to be sent to amctag for each source file to be instrumented. Default: -Xtag-level See "Instrumenter Options" on page 6-16.
AMC_CC_LIBS	One or more paths to the CodeTEST-specific libraries and objects needed at link time for C programs. See "Code-TEST Memory Management Routines" on page 7-2.
AMC_CXX_LIBS	One or more paths to the CodeTEST-specific libraries and objects needed at link time for C++ programs. See "Code-TEST Memory Management Routines" on page 7-2.

Variable	Description
AMC_LIBS	One or more paths to the CodeTEST-specific libraries and objects needed at link time for either C or C++ programs. (Used only if a language-specific version of this variable is not defined.)
AMC_CC_MLIBS	One or more paths to the CodeTEST-specific memory management libraries needed at link time for C programs. See "CodeTEST Memory Management Routines" on page 7-2.
AMC_CXX_MLIBS	One or more paths to the CodeTEST-specific memory management libraries needed at link time for C++ programs. See "CodeTEST Memory Management Routines" on page 7-2.
AMC_MLIBS	One or more paths to the CodeTEST-specific memory management libraries needed at link time for either C or C++ programs. (Used only if a language-specific version of this variable is not defined.)
AMC_CC_IFLAGS	One or more preprocessor options used if preprocessing C source files.
AMC_CXX_IFLAGS	One or more preprocessor options used if preprocessing C++ source files.
AMC_IFLAGS	One or more preprocessor options used if preprocessing C or C++ source files. (Used only if a language-specific version of this variable is not defined.)
AMC_CC_LFLAGS	One or more link-time options required by the CodeTEST-specific libraries and objects for C programs.

Variable	Description
`AMC_CXX_LFLAGS`	One or more link-time options required by the CodeTEST-specific libraries and objects for C++ programs.
`AMC_LFLAGS`	One or more link-time options required by the CodeTEST-specific libraries and objects for C or C++ programs. (Used only if a language-specific version of this variable is not defined.)
`AMC_CC_INCLUDE`	One or more -I preprocessor options used if amctag is to perform C preprocessing (i.e., if the -Xamctag-cpp option is used).
`AMC_CXX_INCLUDE`	One or more -I preprocessor options used if amctag is to perform C++ preprocessing (i.e., if the -Xamctag-cpp option is used).
`AMC_INCLUDE`	One or more -I preprocessor options used if amctag is to perform C or C++ preprocessing. (Used only if a language-specific version of this variable is not defined.)
`AMC_CC_DEFINITIONS`	One or more -i = *include_file* options added when you instrument C sources to provide instrumenter definitions. Default: -i=$AMC_HOME\lib\ $AMC_TARGET\ amc_c.h **Note:** See comments in amc_c.h and amc_cxx.h for edits that may be required for your environment.
`AMC_CXX_DEFINITIONS`	One or more -i = *include_file* options added when you instrument C++ sources to provide instrumenter definitions. Default: -i=$AMC_HOME\lib\ $AMC_TARGET\ amc_cxx.h

Variable	Description
AMC_TMPDIR	The path to the directory for storing temporary files created during instrumentation. Default: .\tmp
AMC_TAGFORMAT	A printf-style format string, where up to 8 integer format specifiers can be supplied, each of which will be replaced with a tag value. The default format is equivalent to *amc_ctrl_port=%d.* See “Changing the CodeTEST tag format” on page 6-15.

Command variables

The following variables can also be defined in your ctcc/ctc++ configuration file if you need to modify the syntax of the commands ctcc/ctc++ executes at various stages of the build.

Note: Appendix C provides complete descriptions of the default syntax and information about editing these commands.

Variable	Description
AMC_CMD_CPP	Syntax of the command line ctcc/ctc++ builds to call your compiler’s driver command in order to perform C/C++ preprocessing.
AMC_CMD_TAG	Syntax of the command line ctcc/ctc++ builds to call amctag when amctag is to perform instrumentation only (not preprocessing).
AMC_CMD_CPPTAG	Syntax of the command line ctcc/ctc++ builds to call amctag when amctag is to perform preprocessing and instrumentation.

Variable	Description
AMC_CMD_COMPILE	Syntax of the command line ctcc/ ctc++ builds to call your compiler's driver command in order to compile the target sources.
AMC_CMD_PREFILTER	Syntax of an optional command ctcc/ctc++ can execute to filter the target source files before they are sent to amctag for instrumentation.
AMC_CMD_POSTFILTER	Syntax of an optional command ctcc/ctc++ can execute to filter the target source files after instrumentation and before compilation.

Response files

To enable ctcc/ctc++ to construct long command lines for the user-configurable commands described above, you can use @{ and }@ in the command syntax to delimit any part of the command you want to be placed in a temporary *response file*. See "Response Files" on page C-9 for examples.

Response file variables

You can define the following variables in your ctcc/ctc++ configuration file to name any response files created by the user-configurable commands.

Variable	Description
AMC_RSP_CPP	Names response files created by the AMC_CMD_CPP command.
AMC_RSP_TAG	Names response files created by the AMC_CMD_TAG command.
AMC_RSP_COMPILE	Names response files created by the AMC_CMD_COMPILE command.

Using ctcc/ctc++ Configuration Files

To configure ctcc/ctc++, create one or more files with names equivalent to .ctccrc-$AMC_TARGET, then set the environment variable AMC_TARGET to point to the file you want ctcc/ctc++ to use. For example, to use .ctccrc-gnu, set AMC_TARGET to *gnu*. The ctcc/ctc++ command searches in the following order for a configuration file:

```
$AMC_HOME/bin/.ctccrc-$AMC_TARGET
$HOME/.ctccrc-$AMC_TARGET
./.ctccrc-$AMC_TARGET
```

If .ctccrc-$AMC_TARGET is present in more than one of the above directories, the individual variable settings in each successive file override previous settings:

- Any variable set in the $HOME version of the file overrides the setting for that variable in the $AMC_HOME/bin version
- Any variable set in the current directory version overrides the setting from the $HOME version.
- Finally, any variable set in your environment (e.g., via the setenv command) overrides the setting made by any version of the .ctccrc-$AMC_TARGET file.

Creating a .ctccrc-$AMC_TARGET file

The example file on the following pages shows how the ctcc/ctc++ variables might be set for an MRI 68K compiler environment. Refer to $AMC_HOME/bin for other examples on which to model your own configuration files.

One or more default configuration files for your site should be stored in your $AMC_HOME/bin directory. If you want to define a personal default configuration file, store it in your $HOME directory. Keep project-specific configuration files in the directory from which you will run ctcc/ctc++.

An example configuration file:

```
#*******************************************************************************
# .ctccrc-$AMC_TARGET
#*******************************************************************************
# ctcc/ctc++ requires that AMC_HOME and AMC_TARGET be defined.
# AMC_HOME is the CodeTEST installation directory.
# AMC_TARGET is a user-chosen name of a build target environment (e.g., "mri360").
#
# Note: ctcc/ctc++ looks for {.,$HOME,$AMC_HOME/bin}/.ctccrc-$AMC_TARGET
#       (e.g., $AMC_HOME/bin/.ctccrc-mri360).
#*******************************************************************************
#===============================================================================
# The following variable definitions are intended for C and C++ environments.
#===============================================================================
#-------------------------------------------------------------------------------
# Path/filename of the CodeTEST Instrumenter.
#-------------------------------------------------------------------------------

AMC_INSTRUMENTER=$AMC_HOME/bin/amctag

#-------------------------------------------------------------------------------
# Default options for the CodeTEST Instrumenter.
#-------------------------------------------------------------------------------
#     Note: These are for amctag, only. Do not put ctcc-specific options here
#           (e.g., -Xuse-cpp or -Xkeep)!

AMC_TAGDEFAULTS=-Xtag-level=2 -Xmri

#===============================================================================
# The following variable definitions are paired into C and C++
# (AMC_CC_* and AMC_CXX_*) language-specific definitions.
#===============================================================================
#-------------------------------------------------------------------------------
# Default "include" file for each C and C++ source.
#-------------------------------------------------------------------------------
#     These provide required declarations, and are always supplied to amctag.

AMC_CC_DEFINITIONS=-i=$AMC_HOME/lib/$AMC_TARGET/amc_c.h
AMC_CXX_DEFINITIONS=-i=$AMC_HOME/lib/$AMC_TARGET/amc_cxx.h

#-------------------------------------------------------------------------------
# Default include directory for C and C++ preprocessing.
#-------------------------------------------------------------------------------
#     These are used only when amctag is asked to perform C or C++ preprocessing
#     (i.e., when the -Xamctag-cpp switch is used with ctcc or ctc++).
#
#     Note: This example expects the environment variable MRI_68K_INC to be
#           defined. This *will* go badly if MRI_68K_INC is not defined!

AMC_CC_INCLUDE=-I$MRI_68K_INC
AMC_CXX_INCLUDE=-I$MRI_68K_INC
```

Example configuration file (cont'd):

```
#-----------------------------------------------------------------------------
# Default preprocessor options, when using amctag as a preprocessor.
#-----------------------------------------------------------------------------
#     These are used only when amctag is asked to perform C or C++ preprocessing
#     (i.e., when the -Xamctag-cpp switch is used with ctcc or ctc++).

AMC_CC_IFLAGS=-D__MRI_68K__
AMC_CXX_IFLAGS=-D__MRI_68K__

#-----------------------------------------------------------------------------
# The name for each of the C and C++ compiler [drivers].
#-----------------------------------------------------------------------------
#     These are used by ctcc/ctc++ to preprocess any C or C++ source files
#     (unless the -Xamctag-cpp switch is used), to compile instrumented
#     source and non-C/C++ source files, and link objects.

AMC_CC=mcc68k -Fee
AMC_CXX=ccc68k -Fee

#-----------------------------------------------------------------------------
# Default link-time options.
#-----------------------------------------------------------------------------
#     If AMC_CC_LIBS or AMC_CXX_LIBS is used to specify source rather than objects
#     or libraries, these variables might be used to specify compiler options at
#     "link" time.

AMC_CC_LFLAGS=-D__MRI_68K__
AMC_CXX_LFLAGS=-D__MRI_68K__

#-----------------------------------------------------------------------------
# Default link-time files.
#-----------------------------------------------------------------------------
#     Generally, these variables will specify the libct.a support libraries,
#     but can be used to reference, e.g. support source files to be compiled
#     and linked in at "link" time.

AMC_CC_LIBS=$AMC_HOME/lib/$AMC_TARGET/libct.a
AMC_CXX_LIBS=$AMC_HOME/lib/$AMC_TARGET/libct.a

#-----------------------------------------------------------------------------
# Default link-time files for -Xmalloc support.
#-----------------------------------------------------------------------------
#     Generally, these variables will specify a libct-*.a support library for
#     inclusion when the -Xmalloc switch is used.

AMC_CC_MLIBS=$AMC_HOME/lib/$AMC_TARGET/libctmalloc.a
AMC_CXX_MLIBS=$AMC_HOME/lib/$AMC_TARGET/libctmalloc.a
```

Using the ctcc/ctc++ Command

The ctcc command accepts C source files as input; the ctc++ command accepts C++ source files as input. Both of these commands produce as output executable load modules or relocatable binary programs for subsequent loading with your linker.

ctcc syntax

```
ctcc [options] sourcefile ...
```

ctc++ syntax

```
ctc++ [options] sourcefile ...
```

See “Instrumenter Options” on page 6-16 for definitions of all ctcc/ctc++ and amctag command options.

Command description

The ctcc command assumes that files with the suffix .c are C source program. The ctc++ command assumes that files with a suffix of .C, .cc, .cpp, or .cxx are C++ source programs. Files with names that do not end in one of these suffixes are not instrumented and are passed unchanged to the compiler.

By default, all C/C++ source files are first routed to your compiler’s cc driver for preprocessing (e.g., gcc with the -E switch). The preprocessed files are then routed to amctag to be instrumented. The instrumented source files are then passed back to your cc command, along with any options or switches that were not intended for ctcc/ctc++ or amctag.

If ctcc/ctc++ determines that it will be producing an executable (i.e., if the -c, -E, and -P switches are not present) your instrumented programs, together with the results of any other specified compilations or assemblies and any required CodeTEST-specific libraries, are loaded in the order given to produce an executable output file.

Example 1: running ctcc from the command line

Suppose you have written an application that uses a.c, b.c, and c.c to create executable *foo*. Ordinarily, to compile and link foo using the GNU C compiler, you might:

```
$ gcc -ofoo a.c b.c c.c
```

To instrument this program before compiling, you could:

1. Create a configuration file (for example, ctccrc-gnu960) to define a set of variables for the ctcc compiler driver.
2. Set the environment variable AMC_TARGET to point to the correct configuration file:

   ```
   setenv AMC_TARGET gnu960
   ```

3. Run ctcc to C preprocess, instrument, compile, and link foo:

   ```
   $ ctcc -ofoo a.c b.c c.c
   ```

Example 2: invoking ctc++ from a makefile

If your build procedure uses a makefile, you can change an existing definition of the CC macro to invoke ctc++ in place of your usual “cc” command. Or, in many environments, simply define the CC macro on the make command line:

```
$ make CC=ctc++
```

Example 3: using the make -n option

In some cases, the best approach may be to invoke your makefile as it currently exists, using the -n (no execution) make option to produce a text file of the commands that would have executed.

This use of make resolves all of the various dependencies, and builds command lines for each tool the makefile invokes. You could then edit the resulting command file to build as much or as little of the code as you want, substituting ctcc or ctc++ for cc.

Assigning Addresses for the Tag Ports

In the normal CodeTEST tagging scheme, two *tag port* variables (amc_ctrl_port and amc_data_port) must be defined and each assigned a specific absolute 32-bit address in the target address space. The address assigned to amc_ctrl_port is the address you will need to enter in the Port Address field of the probe configuration utility to generate a probe configuration file.

Note: *Hybrid tagging*, an alternative CodeTEST instrumentation scheme for supporting 16-bit core processors, has different memory requirements. See Appendix D for details.

The address assigned to amc_ctrl_port must precede the amc_data_port address, and the two ports must be located at adjacent 32-bit words aligned on an 8-byte boundary. In other words, amc_ctrl_port must be located at an address that ends in 0 or 8 hex, and amc_data_port must be located at an address that ends in 4 or C hex. The following example shows one approach for locating the ports.

MRI example

This example shows how tag port addresses can be assigned in an MRI environment, using the PUBLIC command in the linker command file link.cmd.

```
ORDER    vectors,startup,code
ORDER    strings,literals,const
SECT     vars=$00100000
ORDER    vars,zerovars,data,heap
LOAD     vectors.o
LOAD     startup.o
LOAD     sample.o
PUBLIC   _amc_ctrl_port=0xFF00FFF0
PUBLIC   _amc_data_port=0xFF00FFF4
END
```

Chapter 6
The Instrumenter

Theory Overview

The source code instrumenter prepares your program for in-circuit testing by filtering your source files to insert test point instructions, or *tags,* into your code.

Command options are provided to allow you to control the level of tagging the instrumenter performs, based on which CodeTEST tools you plan to use and how critical performance is in your current testing. If you want to see the instrumented version of your code, you can use the *-Xkeep* option to preserve the otherwise intermediate files.

While instrumenting your code, the instrumenter also creates and maintains an Instrumentation Database (IDB) for your program, which contains the symbolic information necessary for CodeTEST to gather measurement data. The host application reads the IDB you specify for your session (i.e., the IDB produced during the build of your target program). If your target includes executables from multiple builds, you can specify multiple compatible IDB files.

Another set of duties the instrumenter can optionally perform are those of a C or C++ preprocessor (i.e., expansion of include files, macro replacements, etc.). Generally, however, you will want to filter your code through your compiler's own preprocessor before instrumenting. By default, ctcc/ctc++ does this automatically.

When you compile and run the instrumented program, the tags placed by the instrumenter write values to a pair of port addresses (amc_ctrl_port and amc_data_port) which you assign in your target memory space at link time. Because the C language variables associated with these port addresses are declared to be volatile, tags will not be affected by any optimization that might be performed during compilation.

Performance tagging

Performance tagging enables CodeTEST to track function entries and exits as your execution path threads through the instrumented program. This level of tagging provides the data necessary to calculate the amount of CPU time consumed by each function, and to dynamically identify the *call pair* relationships among your target program functions.

Performance tagging is generally the least invasive level of instrumentation, and the only kind that is required for you to use the Function Performance, Call Linkage, and High Level Trace views.

Note: The instrumenter does not perform the tagging necessary for you to use the Task Performance view, or any of the other task-based CodeTEST features. See Chapter 8, *Using CodeTEST with an RTOS*.

When you instrument your code at the performance level (using the *-Xtag-level=1* option) the instrumenter reads your sources and inserts a tag at the start of each function, at each function's return statement, and at any other function entry or exit points. The following tag types are used:

- function entry
- function exit

The instrumenter assigns each function a unique number, which serves as a reference to that function's description in the IDB. All entry and exit tags placed in a given function contain that reference number. During execution, the entry and exit tag values are written to amc_ctrl_port. When your workstation receives function entry or exit data from the probe, the CodeTEST host application looks up the corresponding function description in the IDB, then gathers the necessary information for display.

The following example shows a code fragment (AddMove) before instrumentation. Throughout this discussion AddMove will be used to illustrate the tagging levels.

```
int AddMove (uint *touch, uint from, uint to, Piece becomes)
{
    /* We can’t move there (or any further) if it’s invalid, or*/
    /* occupied by our own.*/

    if ((board[to] & PieceMask) == invalid ||
     (board[to] != empty && (((board[to] ^ board[from]) & BLACK) == 0))
       )
    {
     return 0;
    }

    /* Make sure we have room for it on the stack.*/

    if (SzMStk <= MSPtr) {
     MStk = (Move (*)[]) realloc
     (MStk, (SzMStk + IncrSzMStk) * sizeof (Move));
     if (MStk == NULL) {
         fatal (“Couldn’t realloc MStk!”);
     }
     SzMStk += IncrSzMStk;
    }

    /* Add the new move to the list.*/

    (*MStk)[MSPtr].from = from;
    (*MStk)[MSPtr].to = to;
    (*MStk)[MSPtr].becomes.p = becomes;
    MSPtr += 1;

    /* Keep track of the fact that we’ve reached this location, and*/
    /* by whom (for check and castling).*/

    if (becomes & BLACK) {
     touch[to] += 256;
    }
    else {
     touch[to] += 1;
    }

    /* Okay, we did it.  Let ‘em know if they can continue.*/

    return ((board[to] & PieceMask) == empty);
}
```

In this example AddMove has been instrumented with performance tagging. Note that both return statements emit the same exit tag value.

```
int AddMove(uint *touch, uint from, uint to, Piece becomes)
{
  unsigned long amc_entry_dummy = (amc_ctrl_port = 1947205646UL);
  if ((board[to] & (7)) == invalid ||
      (board[to] != empty && (((board[to] ^ board[from]) & 8) == 0))
     )
  {
    return (amc_ctrl_port = 571473934UL,0);
  }
  if (SzMStk <= MSPtr) {
    MStk = (Move (*)[0])realloc(MStk,(SzMStk + 100U) * sizeof(Move));
    if (MStk == 0) {
      fatal("Couldn't realloc MStk!");
    }
    SzMStk += 100;
  }
  (*MStk)[MSPtr].from = from;
  (*MStk)[MSPtr].to = to;
  (*MStk)[MSPtr].becomes.p = becomes;
  MSPtr += 1;
  if ((becomes & 8) != 0) {
    touch[to] += 256;
  }
  else {
    touch[to] += 1;
  }
  return (amc_ctrl_port = 571473934UL,
    ((board[to] & (7)) == empty)
   );
}
```

Coverage tagging

For CodeTEST to track which parts of each function actually execute, your code must be instrumented with *coverage tagging* in addition to the performance tagging previously described. Coverage tagging enables CodeTEST to analyze code coverage, and also enables use of the Trace tool's Control Flow and Source views.

Coverage tags

When you instrument at the coverage level (using the *-Xtag-level=2* option) the instrumenter places coverage tags at branch points in your code. CodeTEST can then track the execution paths taken or not taken. For purposes of coverage analysis, CodeTEST treats code as a series of basic blocks (i.e., sequences in which execution of the beginning of a block implies execution of all code within the block). It is not necessary to know the sequence of execution or the number of iterations; it is only necessary to know whether each block has executed. Control structures that execute a block of code at least once, such as *do* loops, do not require a separate coverage tag. For control structures that may or may not execute a block, such as *if* statements, *for* loops and *while* loops, the instrumenter inserts a tag at the beginning of the block. For *switch* statements, the instrumenter inserts a tag at the beginning of each case, including the default case. Subroutine calls are not tagged, because a subroutine may *longjmp* or *throw* an exception. Subroutine returns are tagged.

Trace tags

The Trace tool requires some additional information to determine the actual sequence of execution. When you instrument at the coverage level, the instrumenter also places trace tags to flag the beginning of certain constructs of iterative execution where coverage tagging alone is insufficient (e.g., *do while* loops). Coverage tagging would flag entry into such a construct, but would not reveal how many times the controlled block executed. Trace tags enable CodeTEST to track the actual number of iterations.

In this example, AddMove has been instrumented with coverage tagging.

```
int AddMove(uint *touch, uint from, uint to, Piece becomes)
{
 unsigned long amc_entry_dummy = (amc_ctrl_port = 1947205646UL);
 if ((board[to] & (7)) == invalid ||
   (board[to] != empty && (((board[to] ^ board[from]) & 8) == 0))
  )
 {
  amc_ctrl_port = 1141932295UL;
  return (amc_ctrl_port = 571473934UL,0);
 }
 amc_ctrl_port = 1141932296UL;
 if (SzMStk <= MSPtr) {
  amc_ctrl_port = 1141932297UL;
  MStk = (Move (*)[0])realloc(MStk,(SzMStk + 100U) * sizeof(Move));
  if (MStk == 0) {
   amc_ctrl_port = 1141932298UL;
   fatal("Couldn't realloc MStk!");
  }
  amc_ctrl_port = 1141932299UL;
  SzMStk += 100;
 }
 amc_ctrl_port = 1141932300UL;
 (*MStk)[MSPtr].from = from;
 (*MStk)[MSPtr].to = to;
 (*MStk)[MSPtr].becomes.p = becomes;
 MSPtr += 1;
 if ((becomes & 8) != 0) {
  amc_ctrl_port = 1141932301UL;
  touch[to] += 256;
 }
 else {
  amc_ctrl_port = 1141932302UL;
  touch[to] += 1;
 }
 amc_ctrl_port = 1141932303UL;
 return (amc_ctrl_port = 571473934UL,
     ((board[to] & (7)) == empty)
    );
}
```

Memory tagging

Memory tagging enables CodeTEST to track your program's dynamic allocation and deallocation of memory, and also to perform error checking and produce diagnostic messages when error conditions are encountered. You can instrument your code with memory tagging alone, or use memory tagging in conjunction with any of the other tagging levels.

When you instrument your code with memory tags (using the *-Xtag-allocator* option) the instrumenter replaces the standard C/C++ memory management calls (malloc, calloc, realloc, free, new, delete) with calls to corresponding CodeTEST routines. For example, each call to malloc is replaced with a call to amc_malloc. The amc_malloc function takes as an argument a tag that identifies the line of code that originally called malloc (i.e., the *caller id*). Beyond that, amc_malloc simply serves as a *wrapper* for a memory allocation routine. Once amc_malloc captures a tag and sends it to the control port address (amc_ctrl_port) it calls the memory allocation routine to perform the actual allocation.

When a successful allocation or deallocation occurs, the CodeTEST library routines write additional tags to the data port (amc_data_port) to keep a running total of the amount of memory allocated, as well as the caller's allocation highwater mark, etc. If an error condition occurs, data tags indicate the nature of the error and point to the diagnostic information the host application will display.

CodeTEST memory management support files

In order for you to use the Memory tool, a supplied set of routines providing allocation, deallocation and consistency checking must be built and linked with your target program. By default, the ctcc/ctc++ compiler driver will include these automatically at link time.

Another instrumenter option *(-Xmalloc)* causes ctcc/ ctc++ to include at link time a supplied set of allocation wrappers.

This effectively replaces *all* memory management calls with calls to the CodeTEST allocation routines, including calls made from uninstrumented code such as standard libraries or a portions of the target program that are not currently being tested. This is necessary for accurate memory error reporting and proper functioning of the Memory tool.

Note: See Chapter 7 for information about using the memory management support files.

In the following example, AddMove has been instrumented for memory monitoring only.

```
int AddMove(uint *touch, uint from, uint to, Piece becomes)
{
  if ((board[to] & (7)) == invalid ||
      (board[to] != empty && (((board[to] ^ board[from]) & 8) == 0))
     )
  {
    return 0;
  }
  if (SzMStk <= MSPtr) {
    MStk = (Move (*)[0])amc_realloc(MStk,(SzMStk + 100U) * sizeof(Move),
                                   672399367UL
                                  );
    if (MStk == 0) {
      fatal(“Couldn’t realloc MStk!”);
    }
    SzMStk += 100;
  }
  (*MStk)[MSPtr].from = from;
  (*MStk)[MSPtr].to = to;
  (*MStk)[MSPtr].becomes.p = becomes;
  MSPtr += 1;
  if ((becomes & 8) != 0) {
    touch[to] += 256;
  }
  else {
    touch[to] += 1;
  }
  return ((board[to] & (7)) == empty);
}
```

Tagging summary

The following table shows which CodeTEST tools (or tool features) are enabled by the various levels of tagging.

	CodeTEST Tools			
Tagging Option	**Performance**	**Coverage**	**Memory**	**Trace**
Performance tagging -Xtag-level=1	■			■ (high level only)
Coverage tagging -Xtag-level=2	■	■		■
Memory tagging -Xtag-allocator			■	■ (allocation calls only)
Full tagging -Xtag-level=2 and -Xtag-allocator	■	■	■	■

Note: If you want the instrumenter to exclude an individual function or functions from tagging, you can use the *no_tagging* pragma to turn off tagging. See “Selectively Turning Off Tagging” on page 6-22.

Tagging Inline functions

An instrumenter option is available for controlling the level of tagging applied to functions defined with the *inline* keyword. You can choose to turn off tagging of all inline functions, tag only the inline functions in your target source files but not in include files (default), or tag all inline functions. See “Instrumentation options” on page 6-19.

Compiler-specific extensions

The instrumenter provides options for enabling or disabling support for the compiler-specific extensions available with several implementations of the C and C++ languages. See “Instrumentation options” on page 6-19.

Additional tag types

In addition to the tags the instrumenter places in your code, two other CodeTEST tag types are of interest here.

RTOS tags

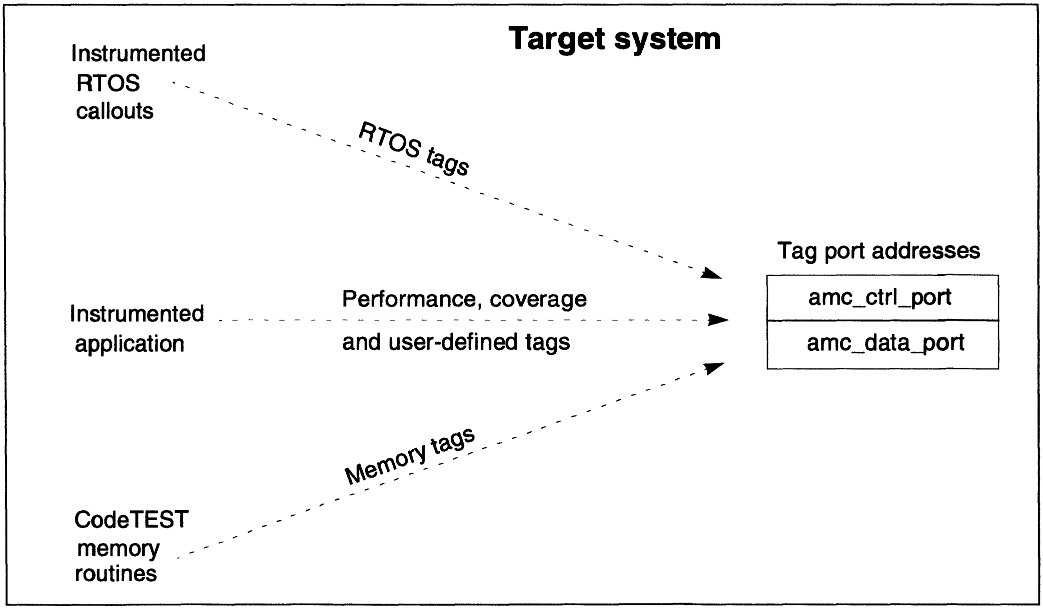
If your target system uses a custom or commercial real-time operating system, you must make some special preparations to monitor basic task activity. See Chapter 8, *Using CodeTEST with an RTOS*.

User-defined tags

To flag events of interest in trace, you can insert function calls to AMCPrintf, AMCPuts, or AMCUserTag into your code. See “AMCPrintf and AMCPuts” on page 4-17 and “User Defined Tags” on page 4-20.

Writing values to the tag ports

All CodeTEST instrumentation writes tags to two monitored ports, which you assign addresses in the target memory space according to guidelines on page 5-18.



Performance overhead

Use of instrumented source code is fundamental to the CodeTEST design. This approach makes accurate measurements possible, even for processors executing out of cache and operating systems that dynamically relocate code. However, running instrumented code is always going to incur some amount of execution overhead. The degree of performance degradation will vary with the specifics of the target program and the level of instrumentation used. Though there is no way of predicting the exact level you will experience, the figures in the following table are representative of what can typically be expected.

Tagging Level	Performance Overhead	Object Expansion
Performance	<2%	2–20% (9% typical)
Coverage	10–20%	3–47% (29% typical)
Memory	*	<2%
All at once	12–22%	3–48% (30% typical)

* If your program rarely calls memory allocation routines, there will be no noticeable effect. Intensive use of allocation routines will incur substantial overhead. In general, each allocation will take approximately twice as long to execute.

Impact on build time

Depending on the number of files and libraries in your build, the number of IDB files involved, etc., the instrumentation process can sometimes have a significant impact on build time. In general, instrumentation will increase build time by about 50%.

Very large builds (>1,000 files) are best built in sections rather than all at once. The *-Xidb-compatible-with* option enables sequential builds to use compatible instrumentation. See "Compatible IDB Files" on page 6-23.

Using the amctag Command

The amctag command takes a C or C++ source file as input, performs C/C++ preprocessing if needed, performs the specified (or default) level of tagging, and creates or updates an IDB file whose name defaults to ./amctag.idb.

If an output file name is not supplied, amctag sends its instrumented source output to stdout.

Syntax

```
amctag [options] sourcefile [outputfile]
```

Options

See "Instrumenter Options" on page 6-16 for definitions of all amctag and ctcc/ctc++ command options.

Example 1: using amctag to preprocess and instrument code

To invoke amctag explicitly to preprocess and instrument your code, you might replace the command:

```
$ gcc -ofoo a.c b.c c.c
```

with the sequence:

```
$ amctag -Xc-mode -Xtag-level=1 a.c a._i
$ amctag -Xc-mode -Xtag-level=1 b.c b._i
$ amctag -Xc-mode -Xtag-level=1 c.c c._i
$ gcc -ofoo a._i b._i c._i
```

The first three commands invoke amctag to preprocess and instrument your original sources. The *-Xc-mode* option tells amctag to expect C rather than C++ code. The tagging option *-Xtag-level=1* instructs amctag to insert performance tags only. For each .c input file, amctag produces a new ._i output file. The last command invokes gcc to compile the three source files to create foo.

Example 2: instrumenting preprocessed files

If you filter your source files through your compiler's own preprocessor before instrumenting with amctag, you must use the *-Xtag-source* option to identify the original sources so amctag can gather the correct information for the IDB. For example:

```
g++ -E a.cxx > a.ii
g++ -E b.cxx > b.ii
g++ -E c.cxx > c.ii
amctag -Xtag-source=a.cxx a.ii a._ii
amctag -Xtag-source=b.cxx b.ii b._ii
amctag -Xtag-source=c.cxx c.ii c._ii
g++ -ofoo a._ii b._ii c._ii
```

In this example, amctag is instrumenting preprocessed c++ code, using the default *(-Xtag-level=2)* tagging option. The *-Xtag-source* option identifies the original source files. The last command in the sequence invokes g++ to compile the preprocessed and instrumented files to create foo.

Example 3: invoking amctag from a makefile

If your build procedure is based on the use of a makefile, and the compilation of each source file can be specified by default rather than requiring an explicit makefile rule, you can insert the instrumentation step by modifying the default .c -> .o rule to run amctag prior to compiling. The default rule for a Sun makefile compiling .c to .o files, for example, might be:

```
.c.o: $(CC) $(CFLAGS) -c $<
```

All that's necessary in this case is to change the rule to:

```
.c.o:
   amctag -Xc-mode -Xtag-level=1 $< $(<:.c=.i)
   $(CC) $(CFLAGS) -c $(<:.c=.i)
```

Changing the CodeTEST tag format

You can use the *-Xtag-format* option to change the format of the tags the instrumenter places in your source code. The default CodeTEST tag format is equivalent to:

```
    "amc_ctrl_port = %d"
```

The *-Xtag-format* option allows you to change the assignment of amc_ctrl_port to a function call to AMCCtrlPort.

Using -Xtag-format with amctag

If you invoke amctag directly (without using the ctcc or ctc++ compiler driver) you can supply the *-Xtag-format* option, as in:

```
amctag -Xtag-format=AMCCtrlPort(%d) foo.c
foo.I
```

Using -Xtag-format with ctcc or ctc++

If you use the ctcc or ctc++ compiler driver, set the variable AMC_TAGFORMAT before using the *-Xtag-format* option.

The AMC_TAGFORMAT variable is interpreted as a printf-style format string, where up to eight integer format specifiers can be supplied, each of which will be replaced with a tag value.

To set AMC_TAGFORMAT in a .ctccrc-$AMC_TARGET file, for example:

```
AMC_TAGFORMAT=(AMCCtrlPort(%d), amc_ctrl_port=%d)
```

The ctcc/ctc++ driver then builds the actual definition of the *-Xtag-format* option to supply to amctag when you specify:

```
"-Xtag-format=$AMC_TAGFORMAT"
```

Instrumenter Options

Note that some of the instrumenter options can only be used with the ctcc/ctc++ compiler driver, while others can be used either with ctcc/ctc++ or on the amctag command line. Any options or switches you supply that are not defined below are passed through to the compiler. Refer to your compiler documentation as necessary.

General operation and preprocessor options

Option	ctcc/ ctc++	amctag	Description
`-c`	■		Suppress linking and produce a .o file for each source file. A single object file can be named using the -o option.
`-D` *name [=def]*	■	■	Define the symbol (*name*) for the C/C++ preprocessor and for amctag, which can optionally do C/C++ preprocessing. Equivalent to a #define directive in the source. If you do not specify a definition (*def*), the value 1 is assigned.
`-E`	■		Perform only preprocessing and instrumentation on input source file(s). Send output (with cpp-style line numbering) to stdout or to a file named with the -o option. See also the -P option.
`-i` *file1=file2*	■	■	Use *file2* instead of *file1* in a #include statement.
`-i` *file1=*	■	■	Ignore any #include *file1* statements.
`-i` *=file2*	■	■	Include file2 before preprocessing any other source file.
`-I` *include_path*	■	■	Add *include_path* to the list of directories to search for #include files with relative names. First, amctag searches the directory containing the source, then any directories named with the -I option, then directories named with the AMC_CC_INCLUDE or AMC_CXX_INCLUDE variables, and finally in /usr/include.

General operation and preprocessor options (cont'd)

Option	ctcc/ ctc++	amctag	Description
`-o` *`filename`*	■		Name the output file *filename*. The name must have the appropriate suffix for the type of file to be produced. The name you assign cannot be the same as the source file name; the compiler does not overwrite the source file.
`-P`	■		Perform only preprocessing and instrumentation on the source file(s). Send the output of each source file to a file of the same name with a .i suffix. No cpp-style line numbering.
`-U` *`name`*	■	■	Remove any existing definition of the cpp symbol *name*. Equivalent to a #undef directive. (Inverse of the -D option.)
`-v`	■	■	Print on standard error output the command executed to run each stage of the compilation.

`-Xpaths-from`
`=`*`env_var`* `...`
	■	■	Add include paths from one or more environment variable names matching the egrep pattern *env_var*.

`-Xuse-cpp`	■		Instruct ctcc to use the C compiler driver (e.g., gcc with -E) to perform C preprocessing on the target source files before calling amctag to instrument the code. Instruct ctc++ to use the C++ compiler driver (e.g., g++ with -E) to perform C++ preprocessing on the target source files before calling amctag to instrument the code.
`-Xamctag-cpp`	■		Use amctag to perform C/C++ preprocessing on the target source files, rather than calling the C/C++ compiler's driver.
`-Xrelaxed`	■	■	Relax the level of error reporting amctag performs. Many amctag syntactic and semantic errors are relaxed to warnings or are silenced. This is the amctag default.
`-Xno-relaxed`	■	■	Allow amctag to perform strict error reporting.

`-Xremove-calls-to`
`=`*`function`*
	■	■	Disable calls to *function* (AMCPrintf, AMCPuts, or AMCUserTag) without removing those calls from your code.

`-@`*`filename`*		■	Specifies a response file (*filename*) which you can create to provide options and file names for amctag.

Compatibility options

Option	ctcc/ ctc++	amctag	Definition
`-Xc-mode`	■	■	Inform amctag that the target source files contain C rather than C++ code. Note: ctcc supplies this switch to amctag automatically.
`-Xk-and-r`	■	■	When instrumenting C source code, assume the code conforms to the Kernighan and Ritchie standard for C preprocessor directives. See "Compatibility modes" on page 6-24.
`-Xansi`	■	■	When instrumenting C source code, assume the code conforms to the ANSI X3.159-1989 standard with some additions. See "Compatibility modes" on page 6-24.
`-Xstrict-ansi`	■	■	When instrumenting C source code, assume the code strictly conforms to the ANSI X3.159-1989 standard. See "Compatibility modes" on page 6-24.
`-Xpcc`	■	■	When instrumenting C source code, assume the code conforms to the System V.3 Unix C compiler standard for preprocessor directives. See "Compatibility modes" on page 6-24.
`-Xbcc`	■	■	Enable Borland C language extensions.
`-Xno-bcc`	■	■	Inverse of -Xbcc. This is the amctag default.
`-Xbso`	■	■	Enable Boston Systems Office C language extensions.
`-Xno-bso`	■	■	Inverse of -Xbso. This is the amctag default.
`-Xgnu`	■	■	Enable amctag support for GNU C language extensions: *alternate* and *inline* keywords, asm labels, cast to union, character escapes, $ in identifiers, explicit reg vars, extended asm syntax, function attributes, incomplete enums, long long, Lvalues, pointer arithmetic, statement expressions, typeof, variable attributes, variable length, zero length. Enable instrumenter support for GNU C++ extensions: destructors and goto.

Compatibility options (cont'd)

Option	ctcc/ ctc++	amctag	Definition
-Xno-gnu	■	■	Inverse of -Xgnu. This is the amctag default.
-Xghs	■	■	Recognize Green Hills Software asm syntax.
-Xno-ghs	■	■	Inverse of -Xghs. This is the amctag default.
-Xicc	■	■	Enable Introl C language extensions.
-Xno-icc	■	■	Inverse of -Xicc. This is the amctag default.
-Xmri	■	■	Enable MRI extensions: *interrupt* and *packed* keywords.
-Xno-mri	■	■	Inverse of -Xmri. This is the amctag default.
-Xmwc	■	■	Enable MetaWare C language extensions.
-Xno-mwc	■	■	Inverse of -Xmwc. This is the amctag default.

Instrumentation options

Option	ctcc/ ctc++	amctag	Definition
-Xtag-level *[=0-2]*	■	■	Cause amctag to insert tags into the target source code. The (optional) value you supply controls the tagging level. See "Theory Overview" on page 6-2 for a discussion of tagging. 0 Do not insert any tags. 1 Tag for performance monitoring only. 2 Tag for performance, coverage, and trace. This is the amctag default.
-Xtag-allocator	■	■	Cause amctag to tag the target source code for memory monitoring, in addition to any tagging done with the -Xtag-level option. See Chapter 7.

Instrumentation options (cont'd)

Option	ctcc/ ctc++	amctag	Definition
`-Xno-tag-allocator`	■	■	Inverse of -Xtag-allocator. This is the amctag default.
`-Xallocator-call-map =`*`mapfile`*	■	■	Cause amctag to use *mapfile* in place of the default Memory Call Definition file: $AMC_HOME/lib/ allocator/ctcall.map. If multiple instances of this switch are used, all *mapfiles* are read in order and duplicate mappings in successive files override previous ones. See "Memory Call Definition file" on page 7-4.
`-Xmalloc`	■		When linking programs into executable load modules, include the CodeTEST-specific definition of the malloc, calloc, realloc, free, new, and delete. With this option, all references to these routines are effectively replaced with references to CodeTEST-specific versions, including references found in uninstrumented code. See "CodeTEST Memory Management Routines" on page 7-2.
`-Xtag-inlines` *`[=0-2]`*	■	■	Specify which functions defined with the *inline* keyword will or will not be tagged at the level you specify with -Xtag-level. `0` Do not tag any inline functions. `1` Tag inline functions defined in the target source files, but do not tag inline functions defined in included files. This is the amctag default. `2` Tag all inline functions.
`-Xno-tag-inlines`	■	■	Do not tag any functions defined with the *inline* keyword. This is the functional equivalent of -Xtag-inlines = 0.
`-Xtag-source=`*`sourcefile`*	■	■	Identify the original source file when instrumenting a source file that has been preprocessed. This enables amctag to incorporate the correct information in the IDB.

Instrumentation options (cont'd)

Option	ctcc/ ctc++	amctag	Definition
`-Xidb=`*`idb_file`*	■	■	Name the Instrumentation Database (IDB) file created or updated during this instrumentation. The amctag default name is amctag.idb.
`-Xidb-compatible-with={`*`idb_file`*`|@`*`file`*`}`	■	■	Make the IDB file created or updated during this instrumentation compatible with *idb_file*, or with multiple compatible IDB files listed in ASCII *file*. See "Compatible IDB Files" on page 6-23.
`-Xkeep`	■		Preserve the preprocessed and instrumented version of each source file. Preprocessed .c files are saved as .i files; instrumented as ._i files. Preprocessed .cpp, .cxx, .cc, or .C files are saved as .ii files; instrumented as ._ii files.
`-Xtag-format=`*`format`*	■	■	Specify *format* (a printf-style string) as the format of the tag statements that amctag inserts in your code. See "Changing the CodeTEST tag format" on page 6-15
`-Xtag-16`	■	■	Enables 16-bit tagging. For use in target systems that employ the CodeTEST hybrid tagging scheme. See Appendix D.
`-Xpassthru`	■	■	If a syntactic or semantic error is encountered in a function, that function is passed through uninstrumented to the compiler. A list of all passed through functions is sent to stderr. This is the amctag default.
`-Xno-passthru`	■	■	If a syntactic or semantic error is encountered in a function, the instrumenter stops.
`-Xabsolute-paths`	■	■	Cause amctag to specify absolute paths to all files listed in the IDB.
`-Xno-absolute-paths`	■	■	Allows amctag to specify relative paths in the IDB. This is the amctag default.

Selectively Turning Off Tagging

You can use the *no_tagging* pragma to exclude individual functions from tagging:

```
#pragma no_tagging name [, name ...]
```

or:

```
#pragma no_tagging qualified name
[, qualified name ...]
```

C++ function names can be an operator, as in:

```
#pragma no_tagging operator delete
static void operator delete (void *) { }
#pragma no_tagging operator delete[]
static void operator delete[] (void *) { }
```

In this example, functions are members of a specific class:

```
#pragma no_tagging amc_guardclass::amc_guardclass
#pragma no_tagging amc_guardclass::~amc_guardclass

class amc_guardclass {
  public:
    amc_guardclass (amctag_t entry, amctag_t exit) : exittag (exit)
    {
        amc_ctrl_port = entry;
    }
    ~amc_guardclass ()
    {
        amc_ctrl_port = exittag;
    }
  private:
    amctag_t exittag;
};
```

Note: No distinction is made among C++ overloaded functions. For example, *foo() foo(int) foo(double)* will all be excluded from tagging if they are defined following:
`#pragma no_tagging foo`

Compatible IDB Files

If your target program is going to include executables produced by multiple builds, use the *-Xidb-compatible-with* option to make the IDB files compatible. To configure a session to use multiple compatible IDB files, see "IDB path and file name" on page 2-15.

Compatible with a single existing IDB file

To produce an IDB compatible with a single existing IDB file, supply the existing IDB file name in the ctcc/ctc++ command using the *-Xidb-compatible-with* option. For example, suppose you instrumented a library called mod2, and during that instrumentation an IDB named mod2.idb was created. Now you want to build an application that uses mod2. Using the *-Xidb-compatible-with* option ensures that tag values in mod2 will not be reused when instrumenting your application. So, the IDB file produced by:

```
ctcc -Xidb-compatible-with=mod2.idb *.c
```

will be compatible with mod2.idb.

Multiple compatible IDB files

To produce an IDB that is compatible with several other compatible IDB files, enter the names of the existing compatible IDB files in an ASCII text file. Then specify the text file name (preceded by an @ sign) using the *-Xidb-compatible-with* option. For example:

```
ctc++ -Xidb-compatible-with=@master.idb *.c
```

will produce an IDB compatible with all of the IDB files listed in the file master.idb.

The warning "Unable to open compatible IDB" indicates that the instrumenter could not find a file specified with the *-Xidb-compatible-with* option. This warning does not prevent instrumentation or compilation.

The amctag C and C++ Preprocessor Features

The amctag command will automatically perform any C/ C++ preprocessing that has not already been done. If you do not want amctag to do the preprocessing, simply filter your files through your compiler's own preprocessor before instrumenting with amctag.

If you use the ctcc/ctc++ compiler driver, by default your source files will be sent to your compiler's preprocessor before being sent to amctag for instrumentation. In that case, refer to your compiler manual for information on preprocessor issues and disregard the information in this section. In some cases, however, you may want to have amctag do the preprocessing. If so, use the *-Xamctag-cpp* option of the ctcc/ctc++ compiler driver.

Compatibility modes

The amctag command offers four language compatibility modes for preprocessor functions.

- K&R mode - conforms to the pre-ANSI de facto standard defined by *The C Programming Language* (1st ed.) by Kernighan and Ritchie. In this mode, most ANSI extensions are activated. (Use the *-Xk-and-r* option.)
- ANSI mode - conforms to ANSI X3.159-1989 with some additions. (Use the *-Xansi* option.)
- Strict ANSI mode - conforms strictly to the ANSI X3.159-1989 standard. (Use the *-Xstrict-ansi* option.)
- PCC compatibility mode - emulates the behavior of System V.3 Unix compilers. (Use the *-Xpcc* option.)

The following table defines the differences among the four compatibility modes.

K&R	ANSI	Strict	PCC	Feature
n	s	s	n	Comments are replaced by nothing (n) or a space (s).
y	n	n	y	Macro arguments are replaced in strings and character constants. For example: #define x(a) if (a) printf("a\n") Yes (y) or no (n).
n	n	y	n	Missing parameter name after a # in a macro declaration generates an error. Yes (y) or no (n).
n	n	y	n	Characters after an #endif directive will generate a warning. Yes (y) or no (n).
e	e	e	w	Preprocessor errors are either errors (e) or warnings (w).
n	0	1	n	__STDC__ macro is predefined to (0), (1), or is not defined (n).
y	y	n	y	__STDC__ macro can be undefined with #undef. Yes (y) or no (n).
n	n	y	n	__STRICT_ANSI__ macro is predefined. Yes (y) or no (n).
n	y	y	n	Spaces are legal before cpp # directives. Yes (y) or no (n).
w	e	e	w	Parameters redeclared in the outer-most level of a function will be given an error (e) or a warning (w).
r	r	r	s	If the function setjmp() is used in a function, variables without the register attribute will be forced to the stack (s) or can be allocated into registers (r).
y	y	n	y	C++ comments "//" are recognized in C files. Yes (y) or no (n).
y	y	n	y	Predefined macros (such as unix, m68k, etc.) are available. Yes (y) or no (n).

Predefined macros

The instrumenter defines the following preprocessor macros. Macros not starting with two underscores (__) will not be defined if the *-Xstrict-ansi* option is given.

Macro	Definition
`__DATE__`	The current data in "Mmm dd yyy" format. It cannot be undefined.
`__DCC__`	The decimal constant 1.
`__DCPLUSPLUS__`	The decimal constant 1 in C++. Only defined when compiling in C++ mode.
`__cplusplus__`	The constant 1 when compiling C++ code otherwise undefined.
`__STDC__`	The constant 0 if *-X*ansi and the constant 1 if -Xstrict-ansi is given in C mode. It cannot be undefined if -Xstrict-ansi is set. It is never defined in C++ mode.
`__STRICT_ANSI__`	The constant 1 if -Xstrict-ansi.
`__FILE__`	The current file name. It cannot be undefined.
`__LINE__`	The current source line. It cannot be undefined.
`__TIME__`	The current time in "hh:mm:ss" format. It cannot be undefined.
`__LDBL__`	The constant 1 if the type long double is different from double
`unix`	The constant 1 when compiling for a Unix target system.
`m68k`	The constant 1.
`mc68k`	The constant 1.
`__m68k`	The constant 1.

Pragmas

The instrumenter supports the following pragma for turning off tagging for a specific function or list of functions:

```
#pragma no_tagging name [, name ...]
```

or:

```
#pragma no_tagging qualified name
[, qualified name ...]
```

Note: See “Selectively Turning Off Tagging” on page 6-22.

All other pragmas are passed through to the compiler.

Include files

The amctag command searches for include files in this order:

1. The directory that contains the source files.
2. Directories named with the -I option to amctag or ctcc.
3. Directories named in an environment variable supplied with the *-Xpaths-from* option to amctag or ctcc.
4. Directories named in the ctcc environment variable AMC_CC_INCLUDE or AMC_CXX_INCLUDE.
5. The default /usr/include directory.

You can use the -i instrumenter command option to substitute, exclude, or include individual include files.

Chapter 7
CodeTEST Memory Functions

CodeTEST Memory Management Routines

CodeTEST-specific definitions of the following memory management routines are provided:

- ❑ malloc
- ❑ calloc
- ❑ realloc
- ❑ free
- ❑ new
- ❑ delete

To use these routines, the appropriate files described in the table below (in your $AMC_HOME/lib directory) must be built and linked with your target programs. By default, the ctcc or ctc++ compiler driver will handle this automatically. See "Configuring ctcc/ctc++" on page 5-7.

If you run amctag directly to instrument your code (without using ctcc or ctc++) you will need to explicitly build and include with your target program the files indicated below.

All of the following sources are written in strict ANSI C, and can be compiled using any ANSI C compiler.

Memory management support files

File	Description	C	C++	Notes
`ctmalloc.c` `ctmalloc.h`	The low-level CodeTEST memory management routines. These routines provide basic allocation, deallocation and consistency checking functions.	■	■	1
`ctmenv.c` `ctmenv.h`	Environment-dependent support routines for ctmalloc.c. Porting the CodeTEST memory management routines to new hardware, new operating system or other environment change generally requires changes in this source file only.	■	■	1

Memory management support files (cont'd)

File	Description	C	C++	Notes
`ctmnew.cxx`	CodeTEST versions of the high-level, standard C++ library routines: operator new and new[]		■	1
`ctmlib.c`	CodeTEST versions of the high-level, standard C library routines: malloc(), calloc(), realloc(), and free(). For calls made from uninstrumented code when the -Xmalloc instrumenter option is used.	■	■	2, 3
`ctndlib.cxx`	CodeTEST versions of the C++ new and delete operators. For calls made from uninstrumented code when the -Xmalloc instrumenter option is used.		■	2, 3
`ctmapi.h`	The API for the low-level CodeTEST memory management routines. If you are committed to using your own allocator, use this API to interface between your allocator and the functions in ctmlib.c	■	■	

Notes:

1. These files must be referenced by the AMC_CC_LIBS variable (for C programs) or AMC_CXX_LIBS variable (for C++ programs) in your .ctccrc-$AMC_TARGET file. The ctcc or ctc++ compiler driver will then include them automatically in each build. To reduce the amount of compilation effort, you may choose to compile these files once and place them in a library (e.g., libct.a). If so, set AMC_CC_LIBS and AMC_CXX_LIBS to point to that library.

2. These files must be referenced by the AMC_CC_MLIBS variable (for C programs) or AMC_CXX_MLIBS variable (for C++ programs) in your .ctccrc-$AMC_TARGET file. The ctcc or ctc++ compiler driver will then include them automatically when you use the -Xmalloc option. To reduce the amount of compilation effort, you may want to compile these files once and place them in a library (e.g., libctmalloc.a). If so, set AMC_CC_MLIBS and AMC_CXX_MLIBS to point to that library.

3. The CodeTEST Memory Allocation view will attribute allocations and deallocations made by uninstrumented code to a special entry labeled UNKNOWN MALLOC CALL. If you do not include these routines with your uninstrumented code, CodeTEST may incorrectly report memory errors. For example, if an allocation is made by an instrumented area of code and then deallocated by an uninstrumented area of code, CodeTEST would erroneously report a memory error.

Compiling the memory management sources

Any ANSI C compiler should be able to compile the CodeTEST memory management files.

An environment symbol must be defined during compilation to specify a combination of a compiler and target operating system. For example, with most Unix-style compilers, adding to the compiler's command line:

```
-DM68K
```

specifies that the sources are being compiled by a Microtec 68K compiler, and will run under an operating system that provides Unix-like brk() and sbrk() routines for allocating process memory. Check your compiler documentation to see how this symbol must be defined.

Note: The symbol AMC_MEMFLAGS may be defined at compile time to set the initial value of the error checking flags. See "The AMC_MEMFLAGS symbol" on page 7-7 for more information about the use of this flag.

Memory Call Definition file

The *Memory Call Definition* file maps allocator function names to wrapper names and CodeTEST tag values. When instrumenting with the *-Xtag-allocator* option, amctag uses the contents of this file to build a table of allocator function names to look for. The default version of the Memory Call Definition file is:

```
    $AMC_HOME/lib/allocator/ctcall.map
```

To override this default and specify a different file, you can use the *-Xallocator-call-map=mapfile* instrumenter option. If more than one instance of this switch is present, all specified files are read, in order, with duplicate mappings in successive files overriding the previous ones.

CodeTEST Memory Error Checking

CodeTEST's memory management routines perform a number of tests, each of which returns a different error code. The memory error codes are grouped into categories, according to the general types of error conditions that generated them (see "Error codes and messages" on page 7-8).

In most cases, the messages provide enough information about the nature of the error. If you need to modify the memory manager for your own environment, you may be interested in the exact conditions each error code represents. (Refer to ctmalloc.c.)

Severity levels

Each error category is assigned one of three severity levels:

Fatal	An unrecoverable error has occurred. Fatal errors do not prevent continued execution of the target program, but all future calls to CodeTEST memory management routines will return their normal error return values, e.g., malloc() will return NULL. This condition can be cleared only by reinitializing the task that encountered the error.
Nonfatal	A recoverable error has occurred. The current memory management call failed, but future calls may succeed.
Info	An informational message that does not indicate an error condition.

Guard bytes

A common heap-related error is for a target program to write to addresses just after the end of a heap block, or, less commonly, just before the beginning of a block. To detect these errors, the CodeTEST memory management routines insert a small number of bytes with known values, called *guard bytes*, immediately before and after the target program's data area in each heap block.

CodeTEST memory management switches

Your target program can control some aspects of the behavior of the CodeTEST memory management routines by setting or clearing the following bit flags in the global variable amc_memFlags.

Name	Value
amc_ChkConsistency	0x00000001
amc_ZeroFreedBlocks	0x00000002
amc_ZeroAllocBlocks	0x00000004
amc_NoFreeReuse	0x00000008

These symbols are defined in ctmalloc.h, which also contains other symbols you might want to access, such as an external declaration of amc_memFlags.

The amc_ChkConsistency flag

The CodeTEST memory management routines can detect a number of error conditions. Because checking the consistency of the heap's internal data structures can be slow when the number of heap blocks is large, the target program can control whether or not these checks are performed by using the amc_ChkConsistency flag. Setting this bit enables the following checks.

Error codes:

0x02, 0x03, 0x12, 0x23, 0x24, 0x25, 0x26, 0x31, 0x32

0x13, 0x21

0x14, 0x22

0x16

See “Error codes and messages” on page 7-8.

The amc_ZeroFreedBlocks flag

The amc_ZeroFreedBlocks flag causes the contents of freed heap blocks to be set to all zeros. If you suspect the target program of reading data from a freed heap block, setting this switch may help detect the problem.

The amc_ZeroAllocBlocks flag

The amc_ZeroAllocBlocks flag causes the contents of all newly allocated blocks to be initialized to zero. Essentially, when this bit is set malloc() becomes functionally equivalent to calloc().

The amc_NoFreeReuse flag

The amc_NoFreeReuse flag prevents the memory management routines from reusing freed heap blocks to satisfy future allocation requests. This is another aid to debugging target program reads of data in freed heap blocks. It is particularly useful for diagnosing cases where the target program is writing through pointers that are contained in freed heap items.

The AMC_MEMFLAGS symbol

To set default values of the CodeTEST memory management switches, compile the CodeTEST memory management sources with the symbol AMC_MEMFLAGS defined as the default switch bits. For example, with most compilers, compiling with

```
-DAMC_MEMFLAGS=0x09
```

would set amc_ChkConsistency and amc_NoFreeReuse on by default. If AMC_MEMFLAGS is not set when these routines are compiled, amc_ChkConsistency and amc_ZeroFreedBlocks will be set by default.

To change the value of a switch at target program run time, set or clear the switch bit in the global unsigned long amc_memFlags. The target program may modify these bits at any time.

Error codes and messages

Code	Severity	Messages
0x00		**No Errors** This is a normal return from a memory management routine.
0x01	Nonfatal	**Out of heap space** There is not enough memory available in the heap to satisfy an allocation request. Future requests for smaller heap blocks may succeed. Retrying this request may succeed if the target program first frees one or more heap blocks. Suggestions: Reduce the target program's heap usage. Reconfigure the task or operating system to make more memory available to the heap. Notes: Generated regardless of the state of amc_ChkConsistency flag.
0x02 0x03 0x04 0x12 0x13 0x14 0x19 0x23 0x24 0x25 0x26 0x27 0x31 0x32 0x33 0x41 0x42	Fatal	**Heap has been corrupted** The memory management routines' internal consistency checks have discovered a corrupted field in a heap block header. Suggestions: Usually caused by a write through an incorrect pointer, possibly to a location in a heap block that was allocated, then freed, then reused as part of a new heap block. Sometimes caused by a bad array index. Notes: These codes are only generated when the amc_ChkConsistency flag is set.

Error codes and messages (cont'd)

Code	Severity	Messages
0x11	Nonfatal	**Free with invalid pointer** The target program has passed an obviously invalid pointer to a deallocation routine. The pointer either contains an address that is not within the heap, or it is not aligned correctly. Null pointers cause the error "Free with NULL pointer" (error code 0x17) rather than this error. Suggestions: An incorrectly aligned pointer might be due to an error in pointer arithmetic. Other wild pointers might be caused by uninitialized memory, reading from a freed heap block, incorrect use of a union, or adding two pointers together. Notes: Error code 0x11 is generated regardless of the state of the amc_ChkConsistency flag.
0x15 0x21	Fatal	**Trailing guard bytes overwritten** One or more of the guard bytes immediately after the user's data have been overwritten. Suggestions: Try looking for incorrect pointer arithmetic, a bad array index, a write through a pointer in a freed heap block. There could be an *off-by-one* or other length calculation error that causes the target program to write past the end of a string. Notes: Error codes 0x13 and 0x21 are generated only when the amc_ChkConsistency flag is set.

Error codes and messages (cont'd)

Code	Severity	Messages
0x16 0x22	Fatal	**Leading guard bytes overwritten** One or more of the guard bytes immediately before the user's data have been overwritten. Suggestions: Try looking for incorrect pointer arithmetic, a bad array index, or a write through a pointer in a freed heap block. Notes: Error codes 0x14 and 0x22 are generated only when the amc_ChkConsistency flag is set.
0x17	Nonfatal	**Free with NULL pointer** Target program has passed a NULL pointer to a deallocation routine. Suggestions: The ANSI C standard allows NULL pointers to be passed to heap deallocators. But since this sometimes indicates a problem in the target program, the CodeTEST memory manager reports it. Notes: Error code 0x15 is generated regardless of the state of the amc_ChkConsistency flag.

Error codes and messages (cont'd)

Code	Severity	Messages
0x18	Nonfatal	**Free of already free heap block** The target program has tried to free a heap block that has already been freed. Suggestions: This error is returned only if the space for the previously-freed block has not been reused as part of a more recently allocated block. If this heap block has been reused as part of a new block, one of the following errors (listed from most to least likely) will be returned instead: 0x12 Block not free 0x13 End of block overwritten 0x14 Beginning of block overwritten This error often indicates a serious bug in the target program. Try looking for duplicate pointers to this heap item in structures or arrays, or for functions that are called more than once as expression side effects. There is a very slight chance that this error is caused by a wild write that corrupted the heap structure. Notes: Generated only when the amc_ChkConsistency flag is set.
0x43	Info	**New heap highwater mark** If the CodeTEST memory management routines have been compiled to do so, this message will be generated each time the total size of the heap grows past its previous maximum.

Memory Management Code Portability

This section is only for users who want to modify the CodeTEST memory management code to port it to their system.

All environment dependencies are isolated in the ctmenv.h and ctmenv.c files, making these routines easy to port to other environments. If you are going to port this code to another environment, you will need to examine ctmenv.h and ctmenv.c in detail.

Assumptions

These routines assume about their environment:

- All pointer types are the same size, and pointers are the same size as long integers.
- There is only one heap.
- The heap consists of a single contiguous block of memory. Multiple extents are not supported.
- The size of the heap may be fixed during initialization, or there may be a way to expand the heap at runtime. If the heap can be enlarged, new space must be added immediately above the existing heap space.
- The standard C library functions memset() and memcpy() must be available.

ctmenv.h

Defined symbol TALIGN

This should be set to the basic alignment type on your CPU, (e.g., on machines that support 4 byte integers and require them to be aligned on a 4 byte address, TALIGN should be defined to be a type that is 4 bytes long). The actual type is not important; only the length of the type is significant.

All other include files used by the CodeTEST memory management routines are #included by this file. Some compilers and operating systems may provide different include files, or there may be differences in the symbols defined in particular include files in other environments. Any changes needed to the list of include files should be made in ctmenv.h.

ctmenv.c

Function amc_enlargeheap()

This function is called whenever the memory management routines need to ask the operating system for more heap memory. The new memory must be directly above the current heap space (i.e., the address of the new space must be equal to the address of the old space plus 1).

Function amc_gettopofheap()

This function returns the highest address allocated by the operating system for use as heap space.

Function amc_lockheap() and Function amc_unlockheap()

The CodeTest memory management routines use a few static data items to describe the state of the heap at any given time. As a result, these routines are not reentrant, and must be protected from nested execution caused by interrupts, including task or thread switching.

The amc_lockheap() function is called upon entry to the CodeTest memory management routines. It must turn off interrupts or implement a semaphore to prevent nested execution of the CodeTest memory management routines.

The amc_unlockheap() function is called on exit from the memory management routines. It should re-enable interrupts or release the semaphore.

Function amc_initheap()

This function is called once to initialize the heap.

Function amc_initheap() is visible only within ctmenv.c.

Chapter 8
Using CodeTEST with an RTOS

Overview

If your target system uses a custom or commercial real-time operating system (RTOS) you need to prepare your RTOS for use with CodeTEST. Once your RTOS is prepared, there are a number of task-oriented CodeTEST features available to you.

Continuous mode features

- ❑ Measurement of target program performance on a task-by-task basis
- ❑ Qualification of performance and memory measurements (not coverage measurements) to a specified task

Trace mode features

- ❑ Trace of task creation, entry, exit, and deletion
- ❑ Triggering on creation, entry, exit, or deletion of a specified task
- ❑ Searching the trace buffer for task creation, entry, exit, or deletion events
- ❑ Qualification of the trigger context to a specified task
- ❑ Qualification of the storage context to a specified task

RTOS preparation

For CodeTEST to make accurate measurements within your RTOS environment, and to enable you to use the task-oriented features, you need to “instrument” your RTOS to enable CodeTEST to track basic task activity.

For the commercial RTOS products that CodeTEST supports (pSOS, VxWorks, and VRTX) interface routines are provided to supply the necessary instrumentation. If you are using a custom RTOS, you will need to manually insert some instrumentation in your RTOS source code. Procedures and examples are provided later in this chapter.

RTOS Task Tracking

CodeTEST internally tracks the function context (call-chain) for each task running in the target system. For this to happen, the target system must notify CodeTEST when to create a new task context, switch from one context to another, or delete a context that is no longer needed.

- ❑ *Task creation* is the point at which the operating system allocates resources for the task.
- ❑ *Task switch* is the point at which the task begins executing, either for the first time or any subsequent time.
- ❑ *Task deletion* is the point at which the OS deallocates the task's resources.

Limits

In Continuous mode, CodeTEST can track up to 1,000 distinct task instances per update period. Exceeding this limit results in new task creations being ignored until an update occurs. (This limit does not apply in Trace mode.)

Error log

If the 1,000 task limit is exceeded, a message is entered in the error log indicating that one or more tasks is being ignored. Execution time of any ignored (i.e., untracked) tasks is attributed to the *All Other Tasks* entry in the Task Performance view.

RTOS tag formats

For CodeTEST to track RTOS events, the RTOS must write to the tag port addresses (amc_ctrl_port and amc_data_port) tag values in the formats defined below.

Task name tags

The task name is the value by which the programer knows the task, and is the key used for specifying tasks in the CodeTEST user interface. Multiple instances of a task with the same task name may be running at the same time.

Two types of task names are supported: *integer* names and *string* names.

An integer name is a single task name tag, which is a unique 32-bit integer value.

A string name is a null-terminated character string comprising 1 to 8 name tags, each of which contains up to 4 ASCII characters. Individual name tags are packed with the first of the 4 characters (C_1) in the high-order byte.

31			0
C_1	C_2	C_3	C_4

For example, the task name:

```
"datalogger"
```

would require three name tags. Note that name tags are emitted in reverse order, and any unused bytes in the last name tag must be set to 0. An entire RTOS event involving the task "datalogger" would be written:

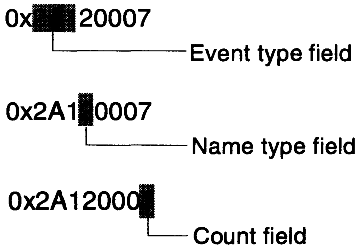
er\0\0
logg
data
Task ID tag
Control tag

Task ID tags

The Task ID tag is a unique 32-bit value containing the ID by which the OS knows the task. Typically, this is an integer the OS assigns at task creation. CodeTEST uses task ID to track individual instances of a task. Task IDs may be reused, provided there is never more than one instance of the same ID value in existence at a time.

Control tags

Each RTOS control tag specifies the type of event that occurred, the type of task names used, and the number of task name tags being sent with the control tag. This information is carried in "fields" within the control tag format. Unused fields are always 0.



The *event type* field specifies the type of event that occurred. RTOS event types are:

2A1	Task create
2A2	Task enter
2A4	Task exit
2A3	Task delete

The *name type* field specifies the type of task names in use:

1	Integer names
2	String names

The number of name tags used is encoded in the *count* field. The value of the count field is the number of name tags -1. For integer names, which require 1 name tag, the count field is always 0. For string names, the number of name tags may vary. For each individual task, it is only necessary to write the number of name tags required to hold that task's name. So a 14-character task name, for example, would require 4 name tags and the value of the count field would be 3.

RTOS tag writing

Each time a task is created or deleted, the target system must write tags in this order:

1	Task Name	1 to 8 name tags written to amc_data_port
2	Task ID	1 task ID tag written to amc_data_port
3	Task Create or Task Delete	1 control tag written to amc_ctrl_port

When a task switch occurs, the target system must write tags in this order:

1	Previous Task Name	1 to 8 name tags written to amc_data_port
2	Previous Task ID	1 task ID tag written to amc_data_port
3	Task Exit	1 control tag written to amc_ctrl_port
4	New Task Name	1 to 8 name tags written to amc_data_port
5	New Task ID	1 task ID tag written to amc_ctrl_port
6	Task Enter	1 control tag written to amc_ctrl_port

Instrumenting Your RTOS

Use the following guidelines to instrument a commercial or custom RTOS for use with CodeTEST.

Instrumenting a commercial RTOS

CodeTEST provides RTOS instrumentation support files for pSOS, VRTX, and VxWorks. To instrument any of these RTOS products:

1. Select the appropriate instrumentation support file from the RTOS-specific directory under $AMC_HOME/lib/rtos (see examples at the end of this chapter). To configure CodeTEST for use with VRTX, for example, select the instrumentation file in $AMC_HOME/lib/rtos/vrtx.

Note: Make sure you select the correct file for your RTOS.

2. Add the instrumentation file to your build procedure so it is compiled (or assembled) and linked with your target program.
3. Add callouts for the instrumentation routines to your RTOS configuration file. The procedure for doing this varies with each RTOS. Typically it involves placing in a configuration file the names of the functions contained in the supplied instrumentation support file so they can be picked up when you link your kernel.

Note: Check the comments in the supplied instrumentation files for additional information specific to each RTOS.

4. Create an RTOS map file as explained on page 8-9. (This is optional.)

Instrumenting a custom RTOS

The following guidelines for instrumenting a custom RTOS assume that the RTOS is a true preemptive operating system (i.e., it can maintain multiple sets of resources such as stacks, register sets, etc., it can suspend execution in one location, execute elsewhere, and then resume execution at the first location.)

Note: DO NOT use the CodeTEST source code instrumenter to instrument your RTOS source code.

To instrument a custom RTOS:

1. Identify the locations in the system where tasks are created, entered, or deleted. There may or may not be a single location for each of these event types. All such locations must be identified. Identifying all of these locations requires a good understanding of the RTOS.
2. Identify the task names (integer or string) and task IDs. If the system does not use task names, use the same value for task name and task ID.
3. Refer to the example instrumentation files at the end of this chapter. Using the appropriate example (for integer names or string names) create code fragments that will write the necessary tag values to the tag port addresses, as explained under "RTOS Task Tracking" earlier in this chapter. The code fragments you create will need to extract the task name and ID. This requires knowledge of the RTOS data structures.
4. Incorporate the code fragments you created in step 3, either by in-lining them at the appropriate locations in your RTOS source code, or by placing them in functions that are called from those locations.
5. Compile/assemble and link the modified RTOS code.
6. Create an RTOS map file as explained on page 8-9. (This is optional.)

Creating an RTOS Map File

Use of an RTOS map file allows you to map each task name to a text string that will represent the task in the CodeTEST user interface. If you do not create an RTOS map file, CodeTEST will use the task name itself to represent each task.

For example, if your RTOS uses ASCII task names, your map file might look like this:

```
"ROOT"    "Root Task"
"TSK1"    "Task #1"
"TSK2"    "Task #2"
  .          .
  .          .
```

If your RTOS uses numeric task IDs (VRTXsa, for example) do not place quotes around the ID values:

```
0         "Root Task"
1         "Task #1"
2         "Task #2"
.            .
.            .
```

Note: To make the RTOS map file known to the CodeTEST host application, enter the file name on the Target Program page of the Configuration Options dialog. See “Target program configuration” on page 2-15.

Example instrumentation support file for VRTX using integer task names:

```
/*-------------------------------------------------------------------------------------
 *   File:     ctvrtx.c
 *   Contents: Provides CodeTEST RTOS instrumentation for VRTX32 and VRTXsa.
 *             Copyright (c) 1995-1996 Applied Microsystems Corporation
 *                          All Rights Reserved
 *
 *   These routines are provided by AMC and must be called from the appropriate VRTX
 *   callout. These routines may be installed by adding the entry:
 *
 *       sys.entry_point2: CodeTEST_hook_init
 *
 *   to the system .def file. If a routine already exists for entry_point2 the routine
 *   CodeTEST_hook_init may be called from anywhere in the existing routine.
 *
 *   These routines write the task name, the task ID, and a CodeTEST control tag to the
 *   CodeTEST tag ports. Integer task names are used. If you are not using variables named
 *   amc_ctrl_port and amc_data_port you must modify these routines to write to your port
 *   location. The write must be made as a single 32-bit operation.
 *-----------------------------------------------------------------------------------*/

#include <compiler.h>
#include <vrtxvisi.h>

extern   volatile unsigned long   amc_ctrl_port;
extern   volatile unsigned long   amc_data_port;

void
CodeTEST_create_hook( TCB *createe, TCB *creator )
{
    register unsigned long id;
    id = (unsigned long) createe->tbid;
    amc_data_port = id;
    amc_data_port = id;
    amc_ctrl_port = 0x2a110000;
    return;
}

void
CodeTEST_delete_hook( TCB *deletee, TCB *deletor )
{
    register unsigned long id;
    id = (unsigned long) deletee->tbid;
    amc_data_port = id;
    amc_data_port = id;
    amc_ctrl_port = 0x2a310000;
    return;
}

void
CodeTEST_switch_hook( TCB *old, TCB *new )
{
    register unsigned long id;
    id = (unsigned long) old->tbid;
    amc_data_port = id;
    amc_data_port = id;
    amc_ctrl_port = 0x2a410000;
    id = (unsigned long) new->tbid;
    amc_data_port = id;
    amc_data_port = id;
    amc_ctrl_port = 0x2a210000;
    return;
}

void
CodeTEST_hook_init( void **arg )
{
    sys_insert_hooks( CodeTEST_create_hook, CodeTEST_delete_hook, CodeTEST_switch_hook );
    return;
}
```

Example instrumentation support file for VxWorks using string task names:

```
/*------------------------------------------------------------------------------
 *  File:   ctvxworks.c
 *  Contents: Provides CodeTEST RTOS instrumentation for VxWorks.
 *            Copyright (c) 1995-1996 Applied Microsystems Corporation
 *                        All Rights Reserved
 *
 *  These routines are provided by AMC and are to be called from the appropriate VxWorks hook.
 *  These routines should installed by calling the function CodeTEST_hook_init() from the
 *  kernel context.
 *
 *  These routines write the task name, the task ID, and a CodeTEST control tag to the
 *  CodeTEST tag ports. If you are not using variables named amc_ctrl_port and
 *  amc_data_port then you must modify this code to write to your port location.
 *  The write must be made as a single 32-bit operation.
 *-----------------------------------------------------------------------------*/

#include "taskLib.h"
#include "taskHookLib.h"

extern   volatile unsigned long   amc_ctrl_port;
extern   volatile unsigned long   amc_data_port;

unsigned long
CodeTEST_put_name( char *p )
{
    int shift;
    int count;
    int i = 0;
    union
    {
       unsigned char c[4];
       unsigned long l;
    } buffer;

    while ( *p && ( i < 31 )) p++, i++;

    count = i;

    do {
       buffer.l = 0;
       do {
           buffer.c[i%4] = *p--;
       } while (( i-- % 4 ) != 0 );
       amc_data_port = buffer.l;
    } while ( i >= 0 );

    return( count / 4 );

}

void
CodeTEST_create_hook( WIND_TCB *new_task )
{
    unsigned long count;

    count = CodeTEST_put_name( new_task->name );
    amc_data_port = (unsigned long) new_task;
    amc_ctrl_port = 0x2a120000 | count;
    return;
}
```

Example instrumentation support file for VxWorks using string task names (cont'd):

```
void
CodeTEST_delete_hook( WIND_TCB *deleted_task )
{
    unsigned long count;

    count = CodeTEST_put_name( deleted_task->name );
    amc_data_port = (unsigned long) deleted_task;
    amc_ctrl_port = 0x2a320000 | count;

    return;
}

void
CodeTEST_switch_hook( WIND_TCB *old_task, WIND_TCB *new_task )
{
    unsigned long count;
    count = CodeTEST_put_name( old_task->name );
    amc_data_port = (unsigned long) old_task;
    amc_ctrl_port = 0x2a420000 | count;

    count = CodeTEST_put_name( new_task->name );
    amc_data_port = (unsigned long) new_task;
    amc_ctrl_port = 0x2a220000 | count;

    return;
}

void
CodeTEST_hook_init( void )
{
    taskCreateHookAdd( (FUNCPTR) CodeTEST_create_hook );
    taskDeleteHookAdd( (FUNCPTR) CodeTEST_delete_hook );
    taskSwitchHookAdd( (FUNCPTR) CodeTEST_switch_hook );

    return;
}
```

Appendix A

System Configuration Reference

This appendix provides a quick reference for the system-wide CodeTEST configuration requirements. References are provided to detailed information about configuring each CodeTEST component.

Probe Configuration

Refer to the *CodeTEST Installation Guide* for details of the probe installation, configuration, and diagnostics.

Network interface

Chapter 5 of the *CodeTEST Installation Guide* covers the probe's network configuration requirements and provides a procedure for performing a confidence test.

The basic requirements are:

- ❑ The probe must be correctly cabled to your 10Base2, 10Base5, or 10BaseT Ethernet medium.
- ❑ The probe must be assigned an IP address and netmask, and a host name. These must be added to your network database files.
- ❑ The mode switch on the probe's back panel must be set for the correct protocol for your network (0=RARP, 1=BootP, 2=stored IP address).
- ❑ To use RARP or BootP protocol, the appropriate server must be available on your network to answer address requests. To use a stored IP address, the IP address must be manually loaded into the probe's flash.

Target connection

Chapter 4 of *CodeTEST Installation Guide* covers the probe's target connection and configuration requirements.

The probetip must be correctly connected to the target hardware. PGA processor applications are supported directly. Adapters are available for other processor packages and for raising or rotating the probetip.

Probe diagnostics

Appendix B of the *CodeTEST Installation Guide* describes normal and abnormal LED behavior, and provides procedures for viewing power-up diagnostic messages.

Tag port addresses

Two CodeTEST tag port variables (amc_ctrl_port and amc_data_port) must be assigned specific, absolute addresses in the target memory space. See "Assigning Addresses for the Tag Ports" on page 5-18 of this manual.

Note: If you are using *hybrid tagging*, see Appendix D for details of tag port memory requirements.

Probe configuration file

Each version of the CodeTEST probe has its own processor-specific configuration utility. To configure CodeTEST to work with your probe, you need to run the appropriate probe utility to generate a probe configuration file. You will need information from your target system's boot code (refer to the booklet supplied with your probe for details).

Once you have generated a probe configuration file, you need to enter that file name on the Probe page of the CodeTEST Configuration Options dialog. See "Probe configuration" on page 2-14 of this manual for details.

Probe firmware version

The probe must have the correct version of the Controller (ctrl.bin) and Data Reduction Processor (drp.bin) firmware for the version of the CodeTEST software you are using. To see the firmware version numbers, select Status Window from the host application Tools menu. Refer to the README file or the release notes in the online help to determine the correct version number.

Software Installation and User Configuration

Refer to Chapter 2 of the *CodeTEST Installation Guide* for information about installing the CodeTEST software.

Permissions

The installed CodeTEST files must allow read and execute permission to all CodeTEST users. Only the owner (preferably someone designated to maintain CodeTEST at your site) should have write permission.

FLEXlm license manager

CodeTEST tools are licensed with the FlexLM license manager. To make the tools available on your network, your license file (supplied by your Applied Microsystems customer representative) must be correctly configured and the license manager daemon (lmgrd) must be running on the machine specified in the license file. (Refer to the *CodeTEST Installation Guide* for details.)

User configuration

The installation script creates shell-specific environment configuration files. Before running the CodeTEST host application or the source code instrumenter, each user should either "source" the appropriate file for the shell they are using or include its contents in a login script. See the *CodeTEST Installation Guide* for details.

X defaults

The installation script places several X11 application defaults files in $AMC_HOME/lib/X11/app_defaults. If your site has a standard location for X11 application defaults (often `/usr/lib/X11/app-defaults`) these files should be moved there during installation. If these files are not moved to your standard X11 application defaults location, the XFILESEARCHPATH environment variable must be to set to point to them **before** the X server is started.

Instrumenter Configuration

For details about instrumenter configuration, refer to Chapter 5 of this manual.

The ctcc/ctc++ environment

The ctcc or ctc++ CodeTEST compiler driver is configured by variables defined in your environment and in a configuration file (.ctccrc-$AMC_TARGET). See "Configuring ctcc/ctc++" on page 5-7 of this manual.

The AMC_TARGET environment variable

The AMC_TARGET environment variable must be set for ctcc/ctc++ to find the appropriate configuration file.

The .ctccrc-$AMC_TARGET file

To define the variables necessary to configure ctcc/ctc++ for your environment, you can create one or more configuration files named .ctccrc-$AMC_TARGET. Examples are installed in $AMC_HOME/bin.

ctcc/ctc++ searches for .ctccrc-$AMC_TARGET in the following order:

1. $AMC_HOME/bin
2. $HOME (your home directory)
3. current directory

If you have versions of .ctccrc-$AMC_TARGET in more than one of these directories, the individual variable settings in each successive version that ctcc/ctc++ finds override the setting made by previously-found versions. Any variable set in your environment (e.g., via setenv) overrides the setting made by any version of .ctccrc-$AMC_TARGET.

RTOS Configuration

For details about RTOS configuration requirements, see Chapter 8 of this manual.

If your target system uses a custom RTOS or one of the off-the-shelf commercial RTOS products CodeTEST supports, you need to instrument your RTOS before using CodeTEST with that target. This allows CodeTEST to make accurate measurements in your RTOS environment and enables use of the task-oriented CodeTEST features.

Note: Do not use the CodeTEST instrumenter to instrument your RTOS source code.

For the commercial RTOS products that CodeTEST supports (pSOS, VxWorks, and VRTX) interface routines are provided to supply the necessary instrumentation.

If you are using a custom RTOS, you need to manually insert tags into your RTOS source code to track basic task activity.

RTOS map file

Use of an RTOS map file is optional. It allows you to map each task name to a text string that will represent the task in the CodeTEST user interface. If you do not create an RTOS map file, CodeTEST will use the task names to represent tasks in the user interface.

To make the RTOS map file known to the CodeTEST application, enter the file name on the Target Program page of the CodeTEST Configuration Options dialog. See “Target program configuration” on page 2-15 of this manual.

Host Application Configuration

To configure the CodeTEST host application to communicate with your probe, find your target source files, etc., select Edit Options from the toolbar Options menu. Then enter your configuration information.

For details, see "Configuring a Session" on page 2-13 of this manual. (Also see "Continuous Mode Setup" on page 3-5 and "Trace setup options" on page 4-6 for information about setting up the tools to make measurements.)

Configuration files

Once you have entered the necessary information, you can save your configuration in a file.

Note: You can give your configuration files any name, however CodeTEST will search at startup for a file named .ctconfig in your home ($HOME) directory. If no such file exists, the host application will be started with default values for all of the configuration variables that have defaults.

Starting CodeTEST with a configuration file

To start CodeTEST with a predefined configuration, use the *-configfile* command line option to supply a configuration file name.

Loading a configuration file during a session

To load a predefined configuration during a session, select the Load Options File command of the Options menu and select the appropriate file.

Appendix B
CodeTEST Error Messages

The messages produced by the host application are generally self-explanatory and require no further information. The following list includes only those messages for which additional troubleshooting tips may be useful.

An unknown tag was identified!

A tag with an unrecognized format was captured by the probe and discarded. Possible causes are bad values written to the tag ports, or amc_ctrl_port and amc_data_port are inverted (see "Assigning Addresses for the Tag Ports" on page 5-18). The collected data is valid.

Bad sequence number returned from the probe!

There is a problem in network communications between the host application and the probe. Execute the Reset Probe Connection command on the Run menu.

Couldn't find information about the unknown function "*xxx*"!

Information for the specified function could not be found in the IDB. Make sure the host application is configured for the correct IDB (i.e., the IDB produced during the instrumentation of the target code you are testing. If you are using multiple compatible IDB files, see "Compatible IDB Files" on page 6-23.

Couldn't find the source file "*xxx*"! Check the 'Source Code Directories' entry in the Configuration window!

All source directory paths for the target program must be entered in the Configuration Options dialog. See "Target program configuration" on page 2-15 for information about specifying one or more source directories.

Couldn't get a license for the feature "*xxx*"! The specific error was: "*xxx*"

Refer to the *CodeTEST Installation Guide* for information about configuring the FLEXlm license manager and your CodeTEST license file.

Couldn't initialize the licensing system!
The specific error was: "*xxx*"

Refer to the *CodeTEST Installation Guide* for information about configuring the FLEXlm license manager and your CodeTEST license file.

Couldn't load some of the source code from the file "*xxx*". Check the 'Source Code Directories' entry in the Configuration window.

All directory paths for the target source files must be entered in the Configuration Options dialog. See "Target program configuration" on page 2-15 for information about specifying one or more source directories.

Couldn't obtain licenses for the unknown probe type "*xxx*"!

The identified probe type is not included in your license file. Check with your CodeTEST system administrator about the probes for which you are licensed.

Couldn't open a connection to the probe!

The host application was unable to open a connection to the probe specified in the configuration dialog. Probe connection errors can result from:

- ❑ Trying to connect to a probe that is already in use.
- ❑ Trying to connect to a probe for which you have no license.
- ❑ Trying to connect to a networked device that is not a CodeTEST probe (e.g., a workstation or an emulator).
- ❑ Trying to connect to a probe that is not correctly configured on your network, is not powered up, or is connected to a target that is not powered up. (Refer to the *CodeTEST Installation Guide* and to your probe booklet for details about setting up your probe.)

Couldn't release the license for the feature "*xxx*". The specific error was: "*xxx*"

Refer to the *CodeTEST Installation Guide* for information about configuring the FLEXlm license manager and your CodeTEST license file.

Event rate has exceeded real-time processing limit!

The rate at which tags were being captured by the probe exceeded it's real-time processing capability. CodeTEST ignores some of the tags while it recovers. In Continuous mode, this initiates *sampled* operation. In Trace mode, a *storage disabled* event is written into the buffer. When the tag rate comes back within limits, CodeTEST continues normal operation. The collected data is valid.

Invalid tag found on internal stack!

This message may be caused by a variety of conditions, including: the target program writing invalid tags to the CodeTEST ports, corrupted tags due to faulty target hardware, or internal CodeTEST error conditions. The collected data may not be valid.

Multiple functions with the same ID (*xxx*) were found while loading IDB file "*xxx*"!
Make sure that all IDBs were built to be compatible with one another.

Compatible IDB files were not specified correctly during the target build process. See "Compatible IDB Files" on page 6-23.

Must specify the probe host name or IP address!

A host name and IP address are assigned to the probe and entered in your network database files during installation. Check with your site system administrator for this information. To configure a CodeTEST session, you must enter either the host name or the IP address in the Configuration Options dialog. See "Probe configuration" on page 2-14.

RTOS tags were encountered while in singletasking mode!

Function context tracking is not working correctly. Try switching to RTOS mode in the Configuration Options dialog. See “Target program configuration” on page 2-15.

Tags were identified as being out of sequence!

A function exit tag was captured whose function ID did not match the function ID of the function currently executing. The collected data is valid.

Task instances exceeded capacity during update interval!

More than 1,000 tasks were active and/or deleted during the update period. Some tasks were not tracked and their execution time was attributed to “All Other Tasks” in the Task Performance display. Try decreasing the update interval in the Configuration Options dialog. See “Probe configuration” on page 2-14.

The CodeTEST application has a protocol version (*xxx*) that is incompatible with the probe “*xxx*”, and is unable to connect.

The host application and the probe firmware versions are incompatible. Check the release notes supplied with your CodeTEST software (in $AMC_HOME/README) for information about updating the probe firmware to the correct version.

The datafile “*xxx*” doesn’t use the current IDB file “*xxx*”. All merged datafiles must use the same IDB file.

You have tried to merge coverage data from measurements that were made using different IDB files. You can only merge coverage data for measurements that were made using the same IDB. See “Merging coverage data” on page 3-12.

The datafile must contain a valid probe ID! Loading cancelled...

Near the top of each saved CodeTEST data file, the probe ID is written. If the file is edited or becomes corrupted such that the probe ID is not present, you cannot load that data file into the host application.

The environment variable 'AMC_HOME' is not set! The application may not run correctly...

AMC_HOME must point to the CodeTEST home directory. The CodeTEST Installation Script creates a shell-specific configuration file for setting up the user environment. Each user should either source that file or copy it into a login script to set AMC_HOME as well as other environment variables. See the *CodeTEST Installation Guide* for details.

The file "*xxx*" has an inappropriate size ("*xxx*" bytes), and is probably not a Probe Configuration file!

Run the configuration utility for your version of the CodeTEST probe to generate a binary configuration file. Then enter the configuration file name in the Configuration Options dialog. See your probe booklet for details.

The IDB file "*xxx*" has an incorrect version number for this version of CodeTEST!

Refer to the release notes ($AMC_HOME/README) for compatibility information. You may need to reinstrument your code for the current release.

The IDB version file has the incorrect format! Consult your CodeTEST administrator.

The file that specifies IDB versions that are compatible with the various host application and the instrumenter versions has become corrupted. Your CodeTEST administrator may need to reinstall this file.

The IDB version file "*xxx*" is missing! Consult your CodeTEST administrator.

The file that specifies IDB versions that are compatible with the various host application and the instrumenter versions is missing. Your CodeTEST system administrator may need to reinstall this file.

The LCA file has an incorrect probe id (*xxx*) which doesn't match the probe's id (*xxx*)!

See your CodeTEST system administrator about installing the correct LCA file for your probe.

The LCA file has an incorrect image size (*xxx*) which doesn't match the actual image size (*xxx*)!

See your CodeTEST system administrator about installing the correct LCA file for your probe.

The LCA file has an incorrect checksum value (*xxx*) which doesn't match the actual checksum value (*xxx*)!

See your CodeTEST system administrator about installing the correct LCA file for your probe.

The network connection to the probe has died unexpectedly!

Try selecting the Reset Probe Connection command on the Run menu. If that doesn't work, check the probe's network connection. See the *CodeTEST Installation Guide* for details about network connection.

The Probe Configuration file "*xxx*" was generated for a different type of probe than the current one (whose id = "*xxx*")!

Run the correct configuration utility for your probe and generate a binary configuration file. Then enter the file name in the Configuration Options dialog. See your probe booklet for details.

B
CodeTEST Error Messages

The target has been reset!

The CodeTEST probe detected that the target processor was reset. CodeTEST continues normal operation. The collected data is valid.

Unable to access file "*xxx*"!
The reason was "*xxx*".
CodeTEST may not be installed correctly.

The host application was unable to access one of its files. See your CodeTEST system administrator about installing the specified file.

Unable to process incoming data rapidly enough!
Increasing the update interval to "*xxx*" seconds.

CodeTEST is automatically adjusting the update interval. The change will be reflected in the configuration options dialog. You may want to save the new configuration for future use with this target.

You must load an IDB before starting the probe!

Before you can make CodeTEST measurements, the host application must be configured to find the IDB file or list of compatible IDB files for your target program. See "IDB path and file name" on page 2-15 and "Compatible IDB Files" on page 6-23.

You must obtain one or more licenses in order to view saved data!

If you try to load a data file without selecting at least one license on the Configuration Options dialog, the host application will automatically check out a license for you if one is available. If no license is available, you cannot load a data file.

You must specify a Probe Configuration file before starting the probe!

Before you can start the probe, you must run the appropriate probe configuration utility to generate a probe configuration file, then enter the file name in the Configuration Options dialog. See “Probe configuration” on page 2-14 and the booklet supplied with your probe.

Appendix C
Customizing ctcc/ctc++

User-configurable Commands

To ensure compatibility of the ctcc/ctc++ compiler driver with the widest possible assortment of C/C++ compilers, environments, user methods and preferences, a set of user-configurable command variables enable you to modify the syntax used to build the actual command lines ctcc/ctc++ executes at each successive stage of the build .

Preprocessor command (AMC_CMD_CPP)

The definition of AMC_CMD_CPP is the syntax ctcc/ctc++ will use to build the preprocessor command line if your compiler's preprocessor is to perform C/C++ preprocessing (i.e., if the *-Xuse-cpp* switch is present).

Default syntax

```
$cc -E $cppflags $< > $@
```

`$cc`	Your C or C++ compiler's driver command, as defined by $AMC_CC or $AMC_CXX.
`-E`	Causes your complier driver to perform only C/C++ preprocessing on the input source file(s) and send the output (with cpp-style line numbering) to stdout.
`$cppflags`	Accumulates flags intended for your compiler's preprocessor, such as -I, -D, -U, -J, etc. Default: $AMC_CC_IFLAGS (for C sources) $AMC_CXX_IFLAGS (for C++ sources) To modify the definition of this variable, see "Configuring Switch Recognition" on page C-7.
`$<`	The input source file.
`>`	Redirects output from stdout to an output file.
`$@`	The preprocessor output file.

Example

To invoke a preprocessor named cpp, for instance, you might first define the configuration variable AMC_CPP as /usr/lib/cpp, then modify the syntax of AMC_CMD_CPP as an environment variable:

```
setenv AMC_CMD_CPP $AMC_CPP $cppflags $< $@
```

or as a line in your configuration file:

```
AMC_CMD_CPP = $AMC_CPP $cppflags $< $@
```

Instrumenter command

(AMC_CMD_TAG)

The definition of AMC_CMD_TAG is the syntax ctcc/ctc++ will use to build the amctag command line if amctag is to perform instrumentation only and not C/C++ preprocessing (i.e., if the *-Xuse-cpp* switch is present).

Default syntax

```
$AMC_INSTRUMENTER $AMC_TAGDEFAULTS $tagmode
$tagformat $tagsource $tagflags $< $@
```

`$AMC_INSTRUMENTER`	The source code instrumenter. Default: $AMC_HOME/bin/amctag
`$AMC_TAGDEFAULTS`	Default options sent to amctag.
`$tagmode`	For C sources: $AMC_CC_DEFINITIONS -Xc-mode For C++ sources: $AMC_CXX_DEFINITIONS
`$tagformat`	Format of the tags amctag places in your code, as defined by : -Xtag-format=$AMC_TAGFORMAT
`$tagsource`	The name of the original source file, as defined by -Xtag-source=*sourcefile*

`$tagflags`

Accumulates these amctag options:
compiler extension switches
(e.g., -Xmri or -Xno-mri)
-Xrelaxed or -Xno-relaxed
-Xpassthru or -Xno-passthru
-Xtag-allocator or -Xno-tag-allocator
-Xtag-inlines or -Xno-tag-inlines
-Xtag-format
-Xtag-level
-Xtag-source
-Xidb
-Xidb-compatible-with
-Xremove-calls-to
To modify the definition of this variable, see “Configuring Switch Recognition” on page C-7.

`$<`

The input source file.

`$@`

The instrumenter output file.

Example

In the following example `@{` and `}@` have been inserted into the default syntax to delimit a part of the command line to be placed in a temporary response file (i.e., the definitions of $AMC_TAGDEFAULTS $tagmode $tagformat $tagsource and $tagflags).

```
AMC_CMD_TAG = $AMC_INSTRUMENTER @{
$AMC_TAGDEFAULTS $tagmode $tagformat
$tagsource $tagflags }@ $< $@
```

See “Response Files” on page C-9 for more information about response files and the variables you can define to name them.

Preprocessor/ instrumenter command (AMC_CMD_CPPTAG)

The definition of AMC_CMD_CPPTAG is the syntax ctcc/ ctc++ will use to build the amctag command line if amctag is to perform both preprocessing and instrumentation (i.e., when the *-Xamctag-cpp* option is present).

Default syntax

```
$AMC_INSTRUMENTER $AMC_IFLAGS $cpptagflags
$include $AMC_TAGDEFAULTS $tagmode $tagformat
$tagsource $tagflags $< $@
```

`$AMC_INSTRUMENTER` The source code instrumenter.
Default: $AMC_HOME/bin/amctag.exe

`$AMC_IFLAGS` One or more preprocessor options (used only if a language-specific version of this variable is not defined.)

`$cpptagflags` Accumulates preprocessor switches intended for amctag, such as -I, -D, -U.
Default: $AMC_CC_IFLAGS or
$ AMC_CXX_IFLAGS
See “Configuring Switch Recognition” on page C-7.

`$include` -I preprocessor options, as defined by $AMC_CC_INCLUDE or $AMC_CXX_INCLUDE

`$AMC_TAGDEFAULTS` Default options sent to amctag. See “Instrumenter Options” on page 6-16.

`$tagmode` For C: $AMC_CC_DEFINITIONS
-Xc-mode
For C++: $AMC_CXX_DEFINITIONS

`$tagformat` Format of the tags amctag places in your code, as defined by :
-Xtagformat=$AMC_TAGFORMAT

`$tagsource` The name of the original source file as defined by -Xtag-source=*sourcefile.*

`$tagflags` Accumulates these amctag options:)
compiler extension switches
(e.g., -Xmri or -Xno-mri)
-Xrelaxed or -Xno-relaxed
-Xpassthru or -Xno-passthru
-Xtag-allocator or -Xno-tag-allocator
-Xtag-inlines or -Xno-tag-inlines
-Xtag-format
-Xtag-level
-Xtag-source
-Xidb
-Xidb-compatible-with
-Xremove-calls-to
To modify the definition of this variable, see “Configuring Switch Recognition” on page C-7.

`$<` The input source file name.

`$@` The instrumenter output file name.

Compile command (AMC_CMD_COMPILE)

The definition of AMC_CMD_COMPILE is the syntax ctcc/ctc++ uses to build the compiler command line.

Default syntax

```
$cc $ccargs { $lflags $libs }
```

`$cc` Your C or C++ compiler’s driver command, as defined by $AMC_CC or $AMC_CXX.

`$ccargs` Accumulates arguments intended for the compiler (e.g., unrecognized switches, object files, etc.)

`$lflags` Link-time options required by CodeTEST-specific libraries, as defined by $AMC_CC_LFLAGS or $AMC_CXX_LFLAGS.

`$libs` Paths to CodeTEST link libraries and options, as defined by $AMC_CC_LIBS and $AMC_CC_MLIBS or $AMC_CXX_LIBS and $AMC_CXX_MLIBS.

Configuring Switch Recognition

ctcc/ctc++ can be configured to "recognize" command switches and decide whether to pass them to the preprocessor, the instrumenter, or the compiler. To configure switch recognition, you append or delete switches from the list of switches that will be accumulated by the internal variables ctcc/ctc++ uses to build the command lines for the successive stages of the build. For example, if you need ctcc/ctc++ to pass -J switches to your compiler's preprocessor, you can add a line to your configuration file to append -J to the definition of the variable *cppflags* (used by AMC_CMD_CPP). Then, when -J is present, ctcc/ctc++ will include it when it builds the preprocessor command line.

Note: Configuring switch recognition can only be done via a configuration file, not in the environment.

The following table lists the variables you can configure for switch recognition. To append or delete a switch from a variable, you can enter in your configuration file regular expressions as described below.

Variable	Description
cppflags	Accumulates preprocessor switches for AMC_CMD_CPP (see page C-2).
tagflags	Accumulates various instrumenter options for AMC_CMD_TAG and AMC_CMD_CPPTAG (see page C-3).
cpptagflags	Accumulates preprocessor options for AMC_CMD_CPPTAG (see page C-5).
ccargs	Accumulates compiler options for AMC_CMD_COMPILE (see page C-6).

Appending a switch

To append a switch to the definition of a variable:

```
variable += switch
```

For example, to configure ctcc/ctc++ to recognize a switch as one that needs to be passed to your compiler's preprocessor, you could enter in your configuration file:

```
cppflags += -Yp,.*
```

ctcc/ctc++ will then append to $cppflags any switch matching the regular expression -Yp,.* for eventual use on the preprocessor command line.

Appending a switch followed by an argument

To append to the definition of a variable a switch followed by a space and then an additional argument:

```
variable +=& switch argument
```

For example:

```
cppflags +=& -include argument
```

will append to $cppflags any instance of `-include` followed by a space then an argument.

Deleting a switch

To delete a switch from the definition of a variable:

```
variable -= switch
```

For example:

```
ccargs -= -J.*
```

deletes -J from the list of switches ctcc/ctc++ will use to build the compilation command line.

Response Files

To allow ctcc/ctc++ to construct command lines longer than 256 characters, you can define the syntax of the user-configurable commands to place part of the command line in a temporary response file. Use @{ and }@ to delimit the part you want placed in the response file. Then, as ctcc/ctc++ builds the command line, everything between @{ and }@ will be placed in the response file, which will be called when the command executes.

For example, if you define AMC_CMD_CPP:

```
$cc @{ -E $cppflags }@ $< > $@
```

ctcc/ctc++ will create a temporary response file containing the definitions of:

```
-E $cppflags
```

Response file naming variables

To accommodate differences in the naming conventions for response files used by various compilers, you can define the following variables:

AMC_RSP_CPP	Names the response file created by AMC_CMD_CPP. Default: @$<
AMC_RSP_TAG	Names the response file created by AMC_CMD_TAG. Default: -@$<
AMC_RSP_COMPILE	Names the response file created by AMC_CMD_COMPILE. Default: @$<

Prefilter and Postfilter Commands

In some cases it may be necessary to perform minor filtering of your target sources, before and/or after sending them to amctag.

Prefilter command (AMC_CMD_PREFILTER)

To filter your source files before instrumentation, you can define AMC_CMD_PREFILTER to specify the prefilter syntax. AMC_CMD_PREFILTER requires input file and output file parameters. Input can be original C or C++ source files, or .i or .ii preprocessed files. Output files are .j or .jj for input to amctag.

For example, the following definition calls bcpp2cpp.exe to prefilter your sources.

```
AMC_CMD_PREFILTER = bcpp2cpp.exe $< $@
```

Postfilter command (AMC_CMD_POSTFILTER)

To filter instrumented sources before ctcc/ctc++ sends them to the compiler, you can define AMC_CMD_POSTFILTER to specify the postfilter syntax. AMC_CMD_POSTFILTER requires input file and output file parameters. Input files are ._j or ._jj instrumented C/C++ files produced by amctag. Output files are ._i or ._ii for input to the compiler.

For example, the following definition adds the *far* keyword to the amc_ctrl_port and amc_data_port declarations in your instrumented sources before compiling with bcc.

```
AMC_CMD_POSTFILTER = sed -e "s/long
\(amc_...._port\)/long far \1/" $< > $@
```

Use the *-Xkeep* instrumenter option if you want to see the intermediate `._j files.`

Appendix D

Hybrid Tagging

Hybrid tagging, an alternative to the normal 32-bit CodeTEST tagging scheme, provides support for 16-bit core processors.

80C186EA/XL Support

The Intel 80C186EA/XL is a 16-bit core microprocessor, and therefore incapable of writing a 32-bit tag value in a single bus cycle using only the data bus. Each 32-bit data write will be broken into 2 or 4 bus cycles with the possibility of an interrupt being serviced between data cycles. If the interrupt service routine is instrumented, the tag stream will become corrupted. Two solutions are available:

- Use the *normal tagging* scheme and prevent interrupts from occurring during a tag write. This approach is the most economical in terms of target memory space.
- Use the *hybrid tagging* scheme, in which the entire tag is broadcast in a single bus cycle. This approach requires a larger amount of target memory space.

Normal Tagging

In the normal tagging scheme, two CodeTEST tag ports (amc_ctrl_port and amc_data_port) must be defined and assigned specific absolute 32-bit memory locations in the target memory space. See "Assigning Addresses for the Tag Ports" on page 5-18 for further information and examples.

Masking off interrupts

When a tag is going to be written, the tagging software must mask off interrupts through the processor's status register. After the tag is written, the software will restore the interrupt to its previous value. This prevents all interrupts except the non-maskable interrupt (NMI) from occurring.

The amc_c.h and amc_cxx.h files installed under $AMC_HOME/lib provide support for masking off interrupts. Refer to comments in those files for edits that may be required for your environment.

Include the following switch in your instrumenter command line:

```
-Xtag-format = #AMCCtrlPort (%d)
```

Disabling NMIs

If any of the following is true, disable NMIs by setting jumper JP1 to position 1=2.

- ❑ If any of the NMI service routine (or function called by the service routine) is instrumented
- ❑ If an RTOS task switch may occur within the NMI service routine
- ❑ If user-defined tags (i.e., calls to AMCUserTag, AMCPrintf, or AMCPuts) have been placed in the NMI service routine

Note: The NMI disabling jumper is only effective for targets using solder-down adapters or socketed targets (i.e., targets in which the CPU is placed in the top socket on the CodeTEST probetip). If a processor is installed in the target board's socket while CodeTEST is in use (i.e., using a clip-on adapter) JP1 must be in position 2=3.

Hybrid Tagging

In the *hybrid tagging* scheme, the entire tag is written in a single instruction. During the tag write, the upper half of the tag value is placed on the lower 16 bits of the address bus, and the lower half of the tag value on the data bus.

Note: In this tagging scheme, interrupts do not need to be masked and the NMI should remain enabled (JP1 in position 2=3).

Instrumenting your code for hybrid tagging

To instrument your target source code for the hybrid tagging scheme, use the *-Xtag-16* instrumenter option.

Reserving tag port memory

Use of hybrid tagging requires two consecutive 64K-byte segments of target memory space, aligned on a 128K-byte boundary. The lower segment is used for control tags; the upper segment for data tags.

The following example shows how the required memory can be reserved in an MRI environment using the PUBLIC command in the linker command file (link.cmd).

```
ORDER     vectors,startup,code
ORDER     strings,literals,const
             .
             .
PUBLIC    _amc_ctrl_array=0xFF000000
PUBLIC    _amc_data_array=0xFF010000
END
```

Index

Applied Microsystems Corporation

Applied Microsystems Corporation maintains a worldwide network of direct offices committed to quality service and support. For information on products, pricing, or delivery, please call the nearest office listed below. In the United States, for the number of the nearest local office, call 1-800-426-3925.

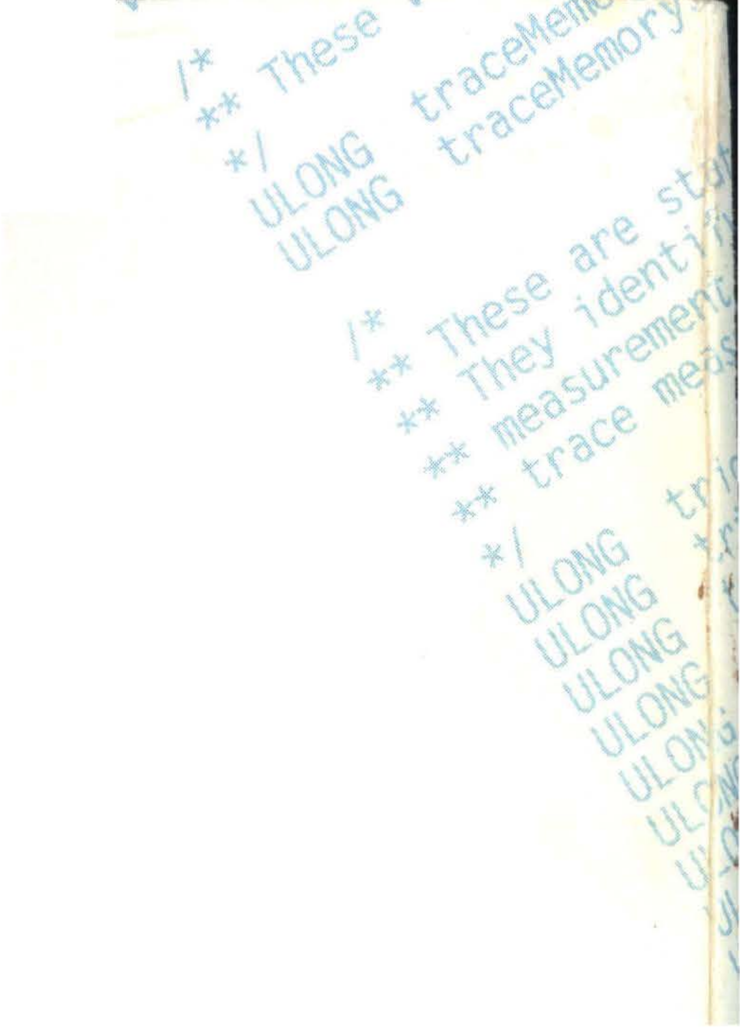
Corporate Office
Applied Microsystems Corporation
5020 148th Avenue Northeast
P.O. Box 97002
Redmond, WA 98073-9702
Tel: 206-882-2000
Toll-free: 1-800-426-3925
CodeTEST Sales:
1-800-895-0831
Customer Support:
1-800-ASK-4AMC (1-800-275-4262)
TRT Telex: 185196
Fax: 206-883-3049

Europe
Applied Microsystems Corporation Ltd.
AMC House, South Street
Wendover, Buckinghamshire, HP22 6EF
United Kingdom
Tel: +44 (0) 1296-625462
Fax: +44 (0) 1296-623460

Germany
Applied Microsystems GmbH
Stahlgruberring 11a, 81829 Muenchen
Germany
Tel: +49 (0)89-427-4030
Fax: +49 (0)89-427-40333

Japan
Applied Microsystems Japan, Ltd.
Arco Tower 13 F
1-8-1 Shimomeguro, Meguro-ku
Tokyo 153
Japan
Tel: +81-3-3493-0770
Fax: +81-3-3493-7270

Part No.	Revision History	Date
924-08000-00	Initial release of CodeTEST for Unix User's Guide. Three-ring bound, 8.5x11-inch page size.	11/95
924-08000-01	Manual update concurrent with CodeTEST v1.1 release for Sun and HP. First perfect-bound version.	2/96
924-08000-02	Manual update concurrent with CodeTEST v1.3 release for Sun-OS, Sun-Solaris, and HP. Also incorporates the ctcc and amctag changes from v1.2 (formerly covered only in README).	9/96



```
/*
** These
*/
ULONG  traceMem
ULONG  traceMemory

/*
** These are st
** They identi
** measuremen
** trace meas
*/
ULONG  tr
ULONG  *
ULONG
ULONG
ULONG
ULONG
ULONG
```

Applied Microsystems Corporation