Manual Instrumentation

Manually insert test-point instructions, or tags, in source code to gather real time execution analysis using the Freescale CodeTEST® Tools.

Purpose

The purpose of this paper is to provide a detailed “how to” description of instrumenting source code that is not normally supported by the CodeTEST Instrumenters. In this document, we refer to this source code as non-supported or as an unsupported language. Currently, the CodeTEST Instrumenters support the C, C++ and Ada languages.

After reading this document, typical CodeTEST users should be able to manually instrument their code and gather real time execution analysis data using the CodeTEST Manager. The focus and examples in this document will be based on the manual instrumentation of assembly language code. However, these instructions can be applied to most unsupported languages. This document assumes the reader is somewhat familiar with the instrumentation process and the CodeTEST Tools.

Dependencies and Limitations

The current limitations of the standard CodeTEST Tools apply, as well as the following:

- Not all of the CodeTEST coverage levels or features are supported. For example, a user would not be able to gather MC/DC coverage analysis on an application that is written in assembly language. There are no multiple condition decisions in assembly code.

  The difficulty in using some coverage levels is that highly optimized assembly code sometimes does not easily map back to equivalent C code. This is not a problem for performance results, since the CodeTEST Tools need only the function entry and exit tags in order to analyze performance, but coverage results may be more difficult to obtain.

- Manually inserted “Function Entry” tags must have corresponding “Function Exit” tags associated with them. The code base cannot have arbitrary jumps in it and therefore should execute in a somewhat linear fashion. That is, functions that have been called by other functions must be exited before the functions that called them exit. The CodeTEST Manager will report an error if it receives an out of sequence or unexpected “Function Exit” tag.

- The user must aware that the tags being used are unique and are not being used any where else in the code base. If duplicate tags are present in the code, the data shown will be inaccurate. The CodeTEST Manager will not report an error.

- The user must be able to modify the code base by inserting specific text. This inserted text must be present during code execution.

- The user assumes all responsibility for correctness of instrumentation.

CodeTEST Development Team

Version 2

freescale semiconductor

Manual Instrumentation Process

Throughout this document we will refer to the code in an unsupported language as the original code. The manual instrumentation process is as follows:

1. Identify original code you want to instrument.
2. Write source code ("new code") in a supported language to mimic the original code.
3. Instrument the new code.
4. Identify the instrumentation tags in the instrumented new code.
5. Add the instrumentation tags to the original code.
6. Instrument any code in your project that you want to support.
7. Compile and link into an executable.

We will apply the process to the following sample code to show how to manually instrument a non-supported language. The assembly function shown below is a simple subroutine that contains a while loop.

Assembly Code

```
_While_Construct:
      link     a6,#0
      move.l   d7,-(a7)
      move.l   d6,-(a7)
      subq.l   #4,a7
      moveq    #10,d6
      move.l   d6,d7
      bra.s    *+10
      move.l   d7,(a7)
      jsr      _gotosleep
      move.l   d6,d0
      subq.l   #1,d6
      tst.l    d0
      bne.s    *-14
      lea      4(a7),a7
      move.l   (a7)+,d6
      move.l   (a7)+,d7
      unlk     a6
      rts
```

Using a Supported Language

The first thing that you must do is reproduce the source code in a supported language. The new code must mimic the original code of the unsupported language. This does not mean that each line of code must be identically mirrored in the new code of the supported language. In fact, if you're looking for performance or trace data, you may be able to just *outline* your original code base with the new code. This will become more evident as we work through our example. The C code below is a very close approximation of the original code.

C Code

```
void While_Construct (void)
{
      int i = 10;
      while( i-- )
            {
            gotosleep(10);
            }
}
```

We suggest the following:

- When you write new code to approximate the original code, we suggest you create the minimum files necessary.
- The directories or files contained in your source base do not need to be represented.
- Make sure the C code has the same functions and their names are similar or identical to that of the subroutines or functions of the original code. This is important, because later you will want to be able to make a correlation of new code to original code, when you monitor the code execution using the CodeTEST Manager.
- If feasible, create a single file that contains all of the functions you wish to profile or reproduce.
- See the section titled "Simplification Suggestions" for suggestions for minimizing the task of reproducing the original source code.
- Once a reasonable representation of the original code is present in a C source file, you can generate the instrumentation tags needed. The actual generation of the tags is automatic and is explained in the next section.

Instrumenting the New Code

The new code must be run through the CodeTEST Instrumenter. The instrumented source code will be saved to a file with the same name as the new code file with a `._i` extension. You preserve the instrumented file by passing the `-CTkeep` switch to the Instrumenter.

The command line to use when instrumenting your new code will depend on your build requirements. However, there are some things you should keep in mind:

- You must pass the `-CTkeep` switch to the Instrumenter, as explained above.
- If your application has a combination of supported and non-supported languages, you should work with the non-supported language first.
- When you instrument the new code that represents the original code, specify a name for the IDB file that is created during instrumentation. Use the switch `-CTidb=Manual_Insert_Tags.idb`. This IDB file will keep a record of the tags that were used.
- If you have supported code in your source base, you must inform the Instrumenter that another IDB has been created. The switch to pass to the Instrumenter is `–CTidb-compatible-with=Manual_Insert_Tags.idb`. This switch will ensure the new IDB is compatible with the other IDB that was created to generate tags.

The command line below represents an invocation of the Instrumenter Driver on the new code, to generate the tags needed.

Instrumenter Command Line

```
$> ctcc -CTtag-level=1 -CTkeep -CTidb=Manual_Insert_Tags.idb Replicated_Code.c
```

Manually Copying Instrumentation Tags to Original Code

You can look at the instrumented source file and see the tag statements inserted by the Instrumenter, as shown below.

Instrumented C code

```
void While_Construct(void)
  {amc_ctrl_port=0x7430001C;{
  int i = 10;
  while( i-- )
      {        gotosleep(10);
      }
  amc_ctrl_port=0x2210001C;}
```

The code in the example above was instrumented at the Performance Level. These tags can now be added to the original source, as shown below. Once the tags have been manually inserted into the original code, the code is ready to be built into an executable image. The CodeTEST Manager will be able to monitor this “hand instrumented” executable image.

Hand-Instrumented Assembly Code

```
_While_Construct:
      lea       -16(a7),a7
      movem.l   d5-d7,4(a7)
      lea    _amc_ctrl_port,a0
      move.l    #0x7430001C,d0
      move.l    d0,(a0)
      moveq     #10,d5
      move.l    d5,d7
      bra.s     *+10
      move.l    d7,(a7)
      jsr       _gotosleep
      move.l    d5,d0
      subq.l    #1,d5
      tst.l     d0
      bne.s     *-14
      lea       _amc_ctrl_port,a0
      move.l    #0x2210001C,d0
      move.l    d0,(a0)
      movem.l   4(a7),d5-d7
      unlk      a6
      rts
```

***Note**: After the tags have been added to the original source code, the new code is no longer needed.*

Simplification Suggestions

This section describes some alternatives and shortcuts for the manual instrumentation process.

- We recommend the user instrument the new source code at the Performance level. This will provide the function entry and function exit tags needed and will ensure the complexity of adding the tags to the original source is minimized.
- The process of creating source code that replicates the original unsupported source can sometimes be automated. We have written scripts for customers that created the supported source needed. In those environments, each directory contained one file and each file

contained a procedure which shared the same name as the file. The script we wrote traversed the directories capturing the names of each file in the directories. We then automatically created dummy C functions (in a separate C file), which contained the names of the original files. Because the file names of the original source matched the procedures, we looked only at file names. The entire script that automated this process was less then 15 lines long.

- Instead of manually inserting tags in your code, you could create a wrapper function for each unsupported function or procedure you are interested in. For example, if you have an asm function named `foo`, create a function `c_foo`. The function `c_foo` will call the unsupported asm function `foo`. This wrapper function will be instrumented and a function entry tag will be written before and after the call to the asm function. The CodeTEST host will report the time spent in the asm function `foo`.