# Monitoring Application Specifics

## Using "User-Defined Events"

*Often developers need to monitor areas of their application that no tool could possibly anticipate. With CodeTEST "User-Defined Events" these areas are not only accessible; they are easily tracked, timed, and thus understood.*

## Contents



## Introduction

CodeTEST Tools now support a new instrumentation and event mechanism that provides the user with an unprecedented level of code trace. CodeTEST Tools traditionally are used to monitor standard code constructs for measurements of performance, code coverage, memory usage/errors, and trace. These capabilities provide valuable insight into the workings of an application but sometimes fall short when application specifics need monitoring. To solve this problem, the "User-Defined Events" feature has been introduced. With User-Defined Events (UDE), developers now have the tools necessary to track and time specifics that interest them, and consequently are able to quickly solve complex problems.

This paper assumes the reader already has some familiarity with CodeTEST Tools, the trace capabilities, and patented instrumentation technology. The reader should review the CodeTEST *User's Guide* and *Instrumenter Reference Manual* if this is not the case. Also, a good high level description of CodeTEST Tools exists at http://www.codewarrior.com/MW/Develop/AMC/CodeTEST/default.htm.

## Instrumentation and Events

CodeTEST Tools monitor an application by evaluating test point instructions inserted in the code base. These test points provide real-time "where, why, when, and how long" information about the application's behavior and are referred to as instrumentation. An application may be

CodeTEST Development Team

Version 4

freescale semiconductor

instrumented automatically or manually. Automatic instrumentation is done by running the CodeTEST Instrumenter, and can provide analysis capabilities for performance, coverage, memory, and trace. Manual instrumentation is done by making calls, at points of interest in the code base, to the CodeTEST trace instrumentation API. Manual instrumentation can coexist with automated instrumentation. User-Defined Events fall in the manual instrumentation arena.

When an instrumented application is executed, it generates data at each instrumentation point. In CodeTEST trace mode, this generated data is known to the CodeTEST host application as events. There are many different types of events that an instrumented application can produce. Some of the most common events that an automatically instrumented application produces include function entry/exit, task switch, and memory malloc/free. User-Defined Events are just an extension of this same mechanism. The only difference is that they are dependent on the developer for the application-side instrumentation calls.

## Setting up a UDE

### Adding UDE instrumentation to the source code

User-Defined Events require developers to place calls to the `CtUserDef` function in their code. Sixteen unique user-defined events can be set up and called in this manner. Each call specifies the event ID, format of the data, and the actual data itself. The same event ID can be used for multiple calls in the application as long as the format remains the same. Currently, integer and string are the only data types available for these calls. The following code from the CodeTEST ct_demo application illustrates a UDE instrumentation call sequence. The UDE instrumentation calls are in bold text.

```
typedef struct _userStruct {
   short ID;
   char* name;
   int         value;
} UserStruct;
foo1.ID = 0;
foo1.value = 1024;
foo1.name = (char*)malloc(NAME_LENGTH * sizeof(char));
strncpy(foo1.name,"first", NAME_LENGTH);
val1 = 101;
val2 = 202;
val3 = 303;
/* send tags for user event 0 with foo struct values */
CtUserDef(0,"hsl",foo1.ID,foo1.name,foo1.value);
/* send tags for user event 1*/
CtUserDef(1,"lll",val1,val2,val3);
```

### Preparing the CodeTEST Manager to use UDE data

The CodeTEST Manager needs to know how to deal with the application's UDE data. This can be accomplished in two ways. First, one can manually create the host event definitions. This is desirable if one wishes to trigger trace collection on the UDE. The minor drawback here is that

the developer must know and correctly input the event's fields. The second method allows the developer to know nothing of the UDE instrumentation. The events are automatically populated on the first trace collection. This approach assures that the events will be specified correctly but has the drawback of having to run through the UDE instrumented location. For applications that execute the instrumented points frequently this is not a big deal and is probably the desired approach. However, for applications that infrequently execute the UDE instrumentation, manually specifying the definitions only makes sense. Note, this is especially true if triggering or timer measurements are needed. The tables below show an example of a manually-specified event and a fragment of some trace capture from the ct_demo application.

**User-Defined Events**

| Status | ID | UDE Name |
|---|---|---|
| Applied | 0 | aUDE |
| Applied | 1 | bUDE |

**Current Event's Fields**

| Event Data | Data Type | Size |
|---|---|---|
| ID | short | 2 |
| name | string | 6 |
| value | int | 4 |

Sample of some trace output:

| Event Index | Source | Description | Data Description | Delta (us) | Elapsed (us) |
|---|---|---|---|---|---|
| 4 | 2 | Function entry | AppMain(): MainApp.c ln. 34 | -2.06 | -4.11 |
| 5 | 2 | Memory Alloc - malloc | MainApp.c ln. 45 (size, addr, heapID) = (7, 0x322710, 0x0) | -2.06 | -2.06 |
| 6 | 2 | User Defined Event 0: aUDE | ID: 0(0x0000), name: "first", value: 1024(0x00000400) | 0.00 | 0.00 |
| 7 | 2 | User Defined Event 1: bUDE | Val1: 101(0x00000065), val2: 202(0x000000ca), val3: 303(0x0000012f) | 1.03 | 2.06 |

## Advantages of User-Defined Events

### Triggering explored

One big advantage of CodeTEST Tools is the triggering capabilities. Triggering allows the developer to use events as start/end trace collection conditions. User-Defined Events are particularly useful for this since the data specifying the condition can be application-specific. With this mechanism, developers have the ability to start/end trace collection at any point in their code base and specify the condition for when to do it. Imagine a problem occurs in the code base on the 350$^{th}$ iteration of some loop. With a debugger, it would be very time consuming and frustrating to track through all the calls and variables in question leading to the problem. With UDE, however, one could easily solve this difficult situation by simply triggering trace collection a few iterations before the problem occurs and tracking the problem's contributors (variables, function arguments, etc) with other user-defined events over time.

User-Defined Events also can be configured to trigger on dynamic data by using the CodeTEST variable alias mechanism. With this combination, a developer can output the value of a variable that could not possibly be known before running, and have a UDE field trigger on its value. To accomplish this, the developer inserts a `CtSetVarAlias` call into the code base to record the dynamic value and then creates a UDE with a field that corresponds to that value. In the CodeTEST Manager, the UDE field value to set for triggering is specified using the `$VAR#` syntax, where `#` is the same ID as was used in the `CtSetVarAlias` call.

### Variable tracking

Often in debugging an application, it is useful to track the value of variables. Developers regularly use `printf` to do this. The problem with this approach for most applications is the context of the `printf` output is not easily obtained. One would need additional superfluous `printf` statements in various parts of the code base to fully understand how one got there and why the problematic context exists. With a CodeTEST trace-instrumented application, UDE can be added in the same manner as a `printf` and will track not only the suspect data but also the context of each output. This means that for each UDE output, the developer can examine the call stack, what thread was executing, and, of course, the value of the data. The old `printf` certainly cannot do that easily. In fact, if `printf` is used in a multithreaded application, the output is often jumbled together unless precautions are taken. With CodeTEST User-Defined Events, this is not an issue.

### Timing aspects

CodeTEST Tools allow for timing measurements to be gathered between any two events. These are referred to in the CodeTEST Manager as “A/B Timers”. The events specifying the start and stop conditions of an A/B timer can be qualified in the same manner as trigger events and can even have event fields values including `$VAR#` specifications. The combination of UDE and A/B timers allows the developer to time almost any aspect of an application. For instance, one could time how long a specific mutex is held or the delta time of a net packet send/receive cycle. Also, a developer could insert UDE instrumentation before and after third party library calls and determine the timing. The uses of this feature are almost limitless.

## Examples of Use

### Triggering trace when a specified condition exists

This example shows how to trigger trace collection when certain conditions exist. In this case, we will use the long-established DiningPhilosophers problem and trace will be collected up to the point where a certain DiningPhilosopher starts eating.

1. Use a UDE to instrument the code where trace data collection needs to be started or stopped. Output one or more values that can be used as trigger conditions. In this example, the ID of the philosopher that is eating is output. The instrumentation call is bold text.

```
void DiningPhilosopher::run() {
   while(!Thread::interrupted())    {
      Singleton<Display>::instance()->thinking(*this);
      Thread::sleep(random(5)*1000);
```

```
        // Hungry
        Singleton<Display>::instance()->right(*this);
        right.take();

        Singleton<Display>::instance()->left(*this);
        left.take();

        // Eating
        Singleton<Display>::instance()->eating(*this);
        CtUserDef(1, "l", m_philospherID);
        Thread::sleep(random(5)*1000);

        right.drop();
        left.drop();
    }
}
```

2. Set up the CodeTEST Manager to interpret this event, using the **User Defined Event** configuration tab. Note that the event definitions must be applied before they will be available for triggering.

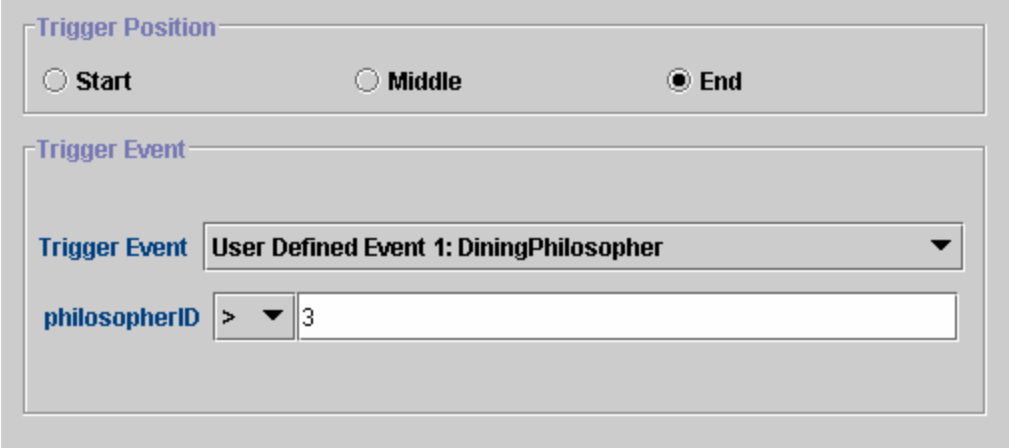
**User-Defined Events**

| Status | ID | UDE Name |
|---|---|---|
| Applied | 1 | DiningPhilosopher |

**Current Event's Fields**

| Event Data | Data Type | Size |
|---|---|---|
| philosopherID | int | 4 |

3. Configure a trigger in the 2nd Level Event tab. Trace is needed up to the trigger condition, so select a trigger position of **End**.
4. Select the UDE event as the trigger event, and choose an operator and value for the trigger comparison. The resulting 2nd Level Event should look something like this.




When the application is run and trace data is collected, the result of this trigger will be the trace leading up to the condition where the philosopher ID was greater than 3. Below is the trace showing where the UDE triggered trace with a philosopher ID of 4. Note that some superfluous events have been removed to help make this trace easier to read.

| Event Index | Source | Description | Data Description | Delta (us) | Elapsed (us) |
|---|---|---|---|---|---|
| 2633 | 7 | Function entry | Display::eating(): DiningPhilosophers.cxx ln. 147 | 0.00 | -72.65 |
| 2,636 | 7 | Function entry | Thread::Guard::Guard(): Guard.h ln. 397 | 0.00 | -68.11 |
| 2,640 | 7 | Function entry | Thread::Mutex::acquire(): Mutex.cxx ln. 45 | -4.54 | -68.11 |
| 2,641 | 7 | User Defined Event 1: Acquire Mutex | hMutex: 0x080a8bb8 | 0.00 | -63.57 |
| 2,649 | 7 | Context - function exit | Thread::Mutex::acquire(): Mutex.cxx ln. 50 | 0.00 | -54.49 |
| 2,651 | 7 | Context - function exit | Thread::Guard::Guard(): Guard.h ln. 401 | 0.00 | -54.49 |
| 2,656 | 7 | Function entry | Thread::Guard::~Guard(): Guard.h ln. 489 | -4.54 | -27.24 |
| 2,663 | 7 | Function entry | Thread::Mutex::release(): Mutex.cxx ln. 62 | 0.00 | -18.16 |
| 2,672 | 7 | User Defined Event 2: Release Mutex | hMutex: 0x080a8bb8 | 0.00 | -4.54 |
| 2,673 | 7 | Context - function exit | Thread::Mutex::release(): Mutex.cxx ln. 66 | 0.00 | -4.54 |
| 2,677 | 7 | Context - function exit | Thread::Guard::~Guard(): Guard.h ln. 498 | 0.00 | 0.00 |

| | | | | | |
|---|---|---|---|---|---|
| 2,678 | 7 | Context - function exit | Display::eating(): DiningPhilosophers.cxx ln. 150 | 0.00 | 0.00 |
| 2,679 | 7 | User Defined Event 1: DiningPhilosopher | philosopherID: 0x00000004 | 0.00 | 0.00 |

### Measuring the lock time for a dynamically created mutex

This example shows how to measure the lock time of dynamically created mutexes. The intent is to show the power UDE provides when combined with other CodeTEST features. The following setup uses A/B Timers, User-Defined Events, and Variable Aliases to track how long each chopstick is held in the DiningPhilosophers problem. Each instance of the class Chopstick contains a mutex to control when it may be taken by a philosopher.

1. Since the mutex to track is dynamically created, inform the CodeTEST Instrumenter that it exists by calling `CtSetVarAlias`, as shown below. Note that `getMutexID` in this example returns the class member (Mutex::m_`hMutex`) for the underlying OS mutex handle.

```
int main() {
   Thread    t[PHILOSOPHERS];
   Chopstick c[PHILOSOPHERS];
   for(int i = 0; i < PHILOSOPHERS; i++) {
      // Output the mutex ID for each chopstick
      CtSetVarAlias(i /* VarAliasID */, c[i].getMutexID());
      int j = (i+1) > (PHILOSOPHERS-1) ? 0 : (i+1);
      t[i].run(new Philosopher(c[i], c[j], i));
   }
   ...
}
```

2. Now instrument the lock and unlock function by adding UDE to the code. User Event 1 signals an acquire (lock) and 2 signals the release (unlock).

```
void Mutex::acquire()
{
   // Create an event signaling the acquire
   CtUserDef(1, "l", m_hMutex);
   // lock the m_hMutex with the appropriate OS call
}

void Mutex::release()
{
   // unlock the m_hMutex with the appropriate OS call

   // Create an event signaling the release
   CtUserDef(2, "l", m_hMutex);
```

```
}
```

The instrumentation is complete; now the Manager needs to be set up to interpret and time the events.

3. Define the Acquire and Release events in the **User Defined Event** tab. Both events will have a field for the `hMutex`. The **User Defined Event** tab should look something like the following when set up.

**User-Defined Events**

| Status | ID | UDE Name |
|---|---|---|
| Applied | 1 | Acquire Mutex |
| Applied | 2 | Release Mutex |

**Current Event's Fields**

| Event Data | Data Type | Size |
|---|---|---|
| hMutex | int | 4 |

Now an A/B Timer can be set to measure the time from the Chopstick's `Mutex::acquire` until the `Mutex::release`.

4. Set up the timing start and stop events in the **A/B Timer** tab. Note that `$VAR#` is used to specify which `VarAliasID` to match. In this case, a `VarAliasID` was output for each Chopstick `mutex(0..4)`. The setup below times the mutex for Chopstick 0, since the alias ID used matches what was set by the `CtSetVarAlias` function. The other chopsticks can also be timed by using their respective VarAlias identifiers. Also, note that these timings could be further restricted by specifying which philosopher task, in essence, giving statistics for how long a given philosopher held a given chopstick.

| On/Off | Description | Start Event | Start Task | Stop Event | Stop Task | Timing Task |
|---|---|---|---|---|---|---|
| √ | Chopstick 0 | UDE 1: Acquire Mutex - $VAR0 | Any | UDE 2: Release Mutex - $VAR0 | Any | Any |

That's it. When you run the application and collect A/B Timers data, the result will be the timing statistics for how long the particular chopstick mutex was held. The following table shows a given runs timing results for each Chopstick mutex.

| Description | Count | Min (us) | Max (us) | Average (us) | Cumulative (us) | % Total Time |
|---|---|---|---|---|---|---|
| Chopstick 0 | 36 | 18.2 | 63.6 | 30.8 | 1107.9 | 0.0% |
| Chopstick 1 | 37 | 18.2 | 72.7 | 30.0 | 1144.3 | 0.0% |
| Chopstick 2 | 36 | 18.2 | 59.0 | 30.4 | 1094.3 | 0.0% |
| Chopstick 3 | 37 | 18.2 | 59.0 | 31.2 | 1153.3 | 0.0% |
| Chopstick 4 | 38 | 18.2 | 63.6 | 30.7 | 1167.0 | 0.0% |

Here is a trace dump that highlights the events contributing to a single locking of the Chopstick 0 mutex. Note that some superfluous events have been removed to help make this trace easier to read.

| Event Index | Source | Description | Data Description | Delta (us) | Elapsed (us) |
|---|---|---|---|---|---|
| 3 | 2 | Function entry | main(): DiningPhilosophers.cxx ln. | 0.0 | 4.54 |
| 9 | 2 | Function entry | Thread::Mutex::Mutex(): Mutex.cxx ln. 30 | 22.7 | 603.91 |

| Event Index | Source | Description | Data Description | Delta (us) | Elapsed (us) |
|---|---|---|---|---|---|
| 13 | 2 | Context - function exit | Thread::Mutex::Mutex(): Mutex.cxx ln. 34 | 0.00 | 626.62 |
| 15 | 2 | Function entry | Chopstick::Chopstick(): DiningPhilosophers.cxx ln. 62 | 27.24 | 658.40 |
| 16 | 2 | Context - function exit | Chopstick::Chopstick(): DiningPhilosophers.cxx ln. 62 | 0.00 | 658.40 |
| 49 | 2 | Coverage - for | main(): DiningPhilosophers.cxx ln. 172 | 0.00 | 753.76 |
| 50 | 2 | Function entry | Chopstick::getMutexID(): DiningPhilosophers.cxx ln. 64 | 0.00 | 753.76 |
| 51 | 2 | Context - function exit | Chopstick::getMutexID(): DiningPhilosophers.cxx ln. 64 | 0.00 | 753.76 |
| 52 | 2 | Set Var Alias | $VAR0: 0xbffff858 | 4.54 | 758.30 |
| 58 | 2 | Function entry | Philosopher::Philosopher(): DiningPhilosophers.cxx ln. 98 | 4.54 | 1,307.72 |
| 59 | 2 | Context - function exit | Philosopher::Philosopher(): DiningPhilosophers.cxx ln. 99 | 0.00 | 1,307.72 |
| 65 | 2 | Coverage - for | main(): DiningPhilosophers.cxx ln. 172 | 572.13 | 1,988.83 |
| 66 | 2 | Function entry | Chopstick::getMutexID(): DiningPhilosophers.cxx ln. 64 | 4.54 | 1,993.37 |
| 67 | 2 | Context - function exit | Chopstick::getMutexID(): DiningPhilosophers.cxx ln. 64 | 0.00 | 1,993.37 |
| 68 | 2 | Set Var Alias | $VAR1: 0xbffff874 | 0.00 | 1,993.37 |
| 72 | 2 | Function entry | Philosopher::Philosopher(): DiningPhilosophers.cxx ln. 98 | 0.00 | 2,002.45 |
| 73 | 2 | Context - function exit | Philosopher::Philosopher(): DiningPhilosophers.cxx ln. 99 | 0.00 | 2,002.45 |
| 76 | 2 | RTOS - task exit | ID:400 Name:DiningPhils_1024 | 95.35 | 2,102.34 |
| 905 | 7 | RTOS - task switch | ID:1807 Name:DiningPhils_6151 | 0.00 | 3,221,681.14 |
| 955 | 7 | Function entry | Chopstick::take(): DiningPhilosophers.cxx ln. 66 | 4.54 | 3,221,781.03 |
| 958 | 7 | Function entry | Thread::Guard::Guard(): Guard.h ln. 397 | 0.00 | 3,221,781.03 |
| 962 | 7 | Function entry | Thread::Mutex::acquire(): Mutex.cxx ln. 45 | 4.54 | 3,221,785.57 |

| Event Index | Source | Description | Data Description | Delta (us) | Elapsed (us) |
|---|---|---|---|---|---|
| 963 | 7 | User Defined Event 1: Acquire Mutex | hMutex: 0xbffff858 | 13.62 | 3,221,790.12 |
| 971 | 7 | Context - function exit | Thread::Mutex::acquire(): Mutex.cxx ln. 50 | 4.54 | 3,221,799.20 |
| 973 | 7 | Context - function exit | Thread::Guard::Guard(): Guard.h ln. 401 | 0.0 | 3,221,799.20 |
| 975 | 7 | Function entry | Thread::Guard::~Guard(): Guard.h ln. 489 | 18.16 | 3,221,799.20 |
| 982 | 7 | Function entry | Thread::Mutex::release(): Mutex.cxx ln. 62 | 0.0 | 3,221,808.28 |
| 991 | 7 | User Defined Event 2: Release Mutex | hMutex: 0xbffff858 | 4.54 | 3,221,817.36 |
| 992 | 7 | Context - function exit | Thread::Mutex::release(): Mutex.cxx ln. 66 | 4.54 | 3,221,817.36 |
| 996 | 7 | Context - function exit | Thread::Guard::~Guard(): Guard.h ln. 498 | 0.0 | 3,221,821.90 |
| 997 | 7 | Context - function exit | Chopstick::take(): DiningPhilosophers.cxx ln. 75 | 4.54 | 3,221,821.90 |

## Summary

The introduction of User-Defined Events into the CodeTEST Tool Suite provides developers a way to track application specifics that often could not be monitored by conventional methods. Setup and use is very straight forward, and the output generated provides context details that would be otherwise difficult to gather. The ability to trigger on UDE data allows developers to zero in on problematic areas when certain conditions exist and thus eliminate tedious debugging. User-Defined Events can also be used as timer endpoints to allow almost any aspect of an application to be timed for performance. UDE is a powerful addition to the CodeTEST Tool Suite.