CodeTEST RTOS Porting Guide

CodeTEST support for a specific RTOS means that CodeTEST Software Analysis Tools (SAT) can be used to analyze an application running on that RTOS, and in many cases, the behavior of the application can be seen in the context of RTOS. To achieve this, CodeTEST SAT needs to have information about RTOS tasks and memory calls. If the RTOS is not multi-tasking, the effort to port CodeTEST SAT is minimal and required only for memory analysis support.

For accurate performance analysis, RTOS task analysis, call pair analysis, and virtual memory analysis, task switching information is required.

For accurate memory analysis on memory allocated/deallocated by an application using RTOS-specific memory calls, you must make memory wrappers for those functions.

RTOS Hooks

Introduction

In order to get information about task switching, CodeTEST SAT uses RTOS hooks, which are functions that need to be called from the RTOS whenever there is a task switch, a task create, or a task delete.

Usually those hook functions are implemented in a library that is linked with the RTOS and application. This model is common for many RTOSes in the embedded world. Also it is possible for the hooks to be included in the RTOS kernel and compiled with it.

The main purpose for those hooks is to send tags to CodeTEST SAT, providing information about any task switching, creation, or deletion event. There are two types of tags: control tags and data tags. A CodeTEST event is formed by a control tag and optional data tags. Control tags define the type of event and specify the number and meaning of the data tags. CodeTEST SAT uses a stack mechanism to interpret the tags: the data tags received are put in a stack and when a control tag is received, the indicated number of data tags are pulled from the stack. So the rule is that the data tags are sent first, in reverse order, and then the control tag is sent.

The hooks send control tags in the following format:

`0x2a`*`xyzzzz`*

where:

x = type of tag

- 1 – TASK_CREATE
- 2 – TASK_ENTER
- 3 – TASK_DELETE
- 4 – TASK_EXIT
- 5 – ISR_ENTER
- 6 – ISR_EXIT

y = type of task identification (name or id)

- 0 – ISRs
- 1 – tasks are identified by their IDs
- 2 – tasks are identified by their names

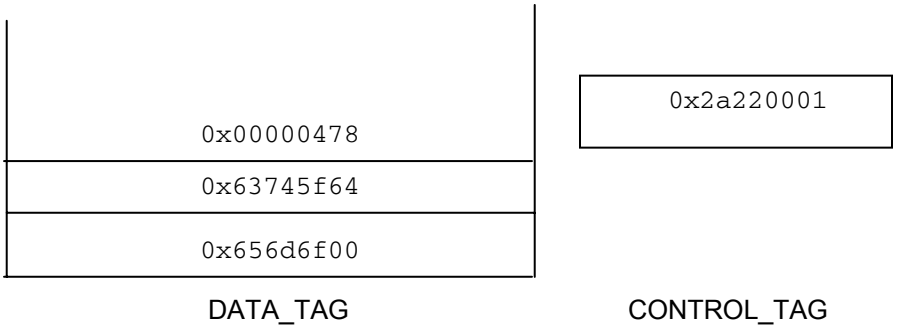
zzzz = ISR number or (number of data tags –1)

CodeTEST Development Team

Version 1

Example

1st tag	2nd tag	3rd tag	4th tag
0x656d6f00	0x63745f64	0x00000478	0x2a220001
DATA_TAG	DATA_TAG	DATA_TAG	CONTROL_TAG



The control tag specifies: a TASK_ENTER event (0x2a**2**20001), the tasks are identified by name (0x2a2**2**0001), and there are two data tags (0x2a22**0001**). In fact, there are three data tags shown, but the data tag, labeled 1st, is a special one providing task ID independent of the *y* field in the control tag. The task ID is used to differentiate between threads from the same process for example. The task ID is shown in the Trace Data and Task Data views, if the task doesn't have a name. If an RTOS map file is used, the task ID is mapped with a name in the map file. The data tags are popped from stack and interpreted by CodeTEST SAT. The first tag pulled is 0x00000478. This specifies the task ID, which is 0x478 (or 1144 in decimal). The next data tag, 0x63745f64, contains the task name (ASCII codes): "c", "t", "_", "d", "e", "m", "o", "\0".

In conclusion, a description of the CodeTEST event looks like "A task enter occurred to task with ID=0x478 and named 'ct_demo'."

Requirement

When creating CodeTEST support for an RTOS, you must identify the points where task switches, creates, and deletes occur. Task switches are most important. Other task events can be handled by CodeTEST SAT, although, it is recommended that you put in create and delete hooks if possible. The first entry to a task is considered to be like a task creation and the last exit from a task like a task deletion.

As shown in the above example, the task ID and task name are required. So the hooks need to be called with this information as parameters; a structure or a pointer to a structure that contains this information will work as well.

Example

Let's imagine the following scenario: a lower priority task (`task_A`) is in a critical region and a higher priority task (`task_B`) waits to enter the same region. The second task is waiting for a variable that signals that the critical region is free. Until the variable is "true", `task_B` waits. The scheduler of the RTOS where those two tasks are running, at the moment when `task_A` leaves the critical region and the variable is set to "true", looks like:

```
Leaving task_A

Put task_A in running_queue after task_B because of its lower
priority
Look in running_queue for the next task to be scheduled
```

```
Task_B is the next task
Enter task_B
```

This scheduler needs to be “hooked” with the CodeTEST calls. It will look like:

```
Leaving task_A
CodeTEST_task_exit_hook(task_A_info)
Put task_A in running_queue after task_B because of its lower
priority
Look in running_queue for the next task to be scheduled
Task_B is the next task
CodeTEST_task_enter_hook(task_B_info)
Enter task_B
```

The same thing needs to be done when a task is created or deleted. It is important to find the right points to put these hooks, because otherwise errors can occur. For example, if a delete tag is sent, but then the same task is scheduled again, CodeTEST SAT will be confused. Also, if a create tag appears after an enter or exit tag to that task is sent, CodeTEST SAT will again be confused. Another common hook misplacement is two consecutive exits from the same task, without an entry to that task between them.

Template

Here is a sample template for writing a CodeTEST hook:

```
#include "ct_rtos.h"

#define CODETEST_TASK_CREATE  (0x2a100000)
#define CODETEST_TASK_ENTER   (0x2a200000)
#define CODETEST_TASK_DELETE  (0x2a300000)
#define CODETEST_TASK_EXIT    (0x2a400000)

#define CODETEST_ISR_ENTER    (0x2a500000)
#define CODETEST_ISR_EXIT     (0x2a600000)

#define CODETEST_TASK_NAMES   (0x00020000)
#define CODETEST_TASK_IDS     (0x00010000)

#define UNIQUE_CHARS    32

void CodeTEST_create_hook(TASK_STRUCT_PTR  td_ptr)
{
   _int_disable();

   if (td_ptr != NULL)
   {
      count = CodeTEST_put_name( td_ptr->TASK_NAME );
      __ct_data_tag__( td_ptr->TASK_ID );
      __ct_ctrl_tag__( CODETEST_TASK_CREATE | CODETEST_TASK_NAMES |
      (count & 0xffff) );
    }

   _int_enable();

   return;
}
```

```
unsigned long
CodeTEST_put_name(const char *p )
{
  size_t len ;
  unsigned long ret ;
  int index;
  const char* p ;  /* walks name backwards in multiples of 4 */
  u32 buffer;

  len = strlen(name);

  if (len > 31) len = 31 ;  /* truncate if name is too long */
  ret = (len+3)/4 ;

  p = name + (ret-1)*4;

  /* The first word is special as it is null padded if
     len is not a multiple of 4 */

  buffer = 0 ;
  switch(len%4) {
  case 0 :
    p += 4 ;
    ret += 1 ;
    break ;
  case 3 :
    buffer += p[2] << 8 ;
  case 2 :
    buffer += p[1] << 16 ;
  case 1 :
    buffer += p[0] << 24 ;
  }
  __ct_data_tag__( buffer);
  p -= 4 ;

  for(index = 1; index < ret ; index++, p -= 4) {
    buffer = 0;
    buffer = (p[0] << 24) + (p[1] << 16)
      + (p[2] << 8) + p[3] ;
    __ct_data_tag__( buffer );
  }

  return ret-1 ;
}
```

In the above template, notice that the `ct_rtos.h` header is included. Typically, this header can be found in the CodeTEST SAT installation in the `cttarget/rtos/common/include` directory. This header defines `__ct_data_tag__` and `__ct_ctrl_tag__`. It is dependent on the type of CodeTEST Probe (HWIC or SWIC), tag format choice, and, environmental constraints on how those defines are set.

If you use CodeTEST for an RTOS that needs run-time configuration of the tag ports, you need to define the TAG_DEST macro as “CT_DEVICE_DRVR”. This means that you need to have ctTag() and ctDataTag() functions implemented (those functions will send the tags to ctserver). For RTOSes that use flat memory space, the TAG_DEST should be “TAG_TO_PTR”. The tags will be written to a pointer dereference,, where the pointers are in fact CodeTEST ports. Something similar happens if the TAG_DEST macro is set to the “PCI_DRVR” value. In this case, the CodeTEST Probe uses a PCI card to transfer the tags from target to ctserver and a

PCI CodeTEST driver is needed. If TAG_DEST has no value, tags are written to ct_port[2] array. ct_port[0] is the control port and ct_port[1] is the data port.

You may also notice that interrupts are disabled for the period of time when CodeTEST tags are processed. It is important to ensure that writing an RTOS task hook event (control tag + data tags) is not interrupted. Otherwise it is possible that tags will be scrambled, adversely affecting the accuracy of the data.

Use of the template provided above is most suitable for CodeTEST Hardware Probes. For CodeTEST Software Probes, you need a deeper understanding of CodeTEST internals, and more steps are required, which will not be described here. For a Software Probe, tags are sent from kernel space to user space through a sharing resource mechanism (like shared memory).

Some RTOSes have a facility that offers already-implemented hook calls, typically: enter task, exit task, create task, delete task. These hooks can call the CodeTEST user functions that handle RTOS events (create, enter, exit, delete).

Memory wrappers

Introduction

For memory analysis, CodeTEST SAT needs to know when a memory is allocated and when memory is deallocated. Memory wrappers convey this information. Besides memory calls from C/C++ Runtime, CodeTEST SAT can track specific RTOS memory allocation. The main purpose for a CodeTEST memory wrapper is to execute the original memory call and to send `ctserver` a proper tag.

Unlike RTOS hooks that need to be put in “by hand”, memory wrappers are automatically introduced by the CodeTEST Instrumenter (when instrumentating for memory analysis). A special file, usually called ct*rtosname*wrap.map, maps CodeTEST memory wrappers with RTOS real memory calls. An example of such a file in the CodeTEST installation is in the lib/rtos/*rtosname* directory.

Map File structure

A detailed explanation of the map file structure in the file header follows.

```
# user-name     wrapper-name subtype function-type replacement-function-prototype
#

malloc,           ct_malloc,    3,    2,    extern "C" void *ct_malloc( unsigned long,_ct_tag_t);

calloc,           ct_calloc,    3,    3,    extern "C" void *ct_calloc(unsigned long,unsigned long,_ct_tag_t);

realloc,          ct_realloc,   3,    4,    extern "C" void *ct_realloc( void *, unsigned long, _ct_tag_t );

"operator new",        "operator new",    3,    5

free,                  ct_free,           4,    6, extern "C" void ct_free( void *, _ct_tag_t );

"operator delete",   "operator delete",   4,   7

"operator new[]",    "operator new[]",    3,   8

"operator delete[]", "operator delete[]", 4,   9
```

Each line contains the user function name (e.g., `malloc`), wrapper name (e.g., `ct_malloc`), subtype (e.g., `3`), function-type (e.g., `2`) and wrapper function prototype.

- User function name is the real name for the memory call that is used in the analyzed application.

- Wrapper name is the name that will be used by the Instrumenter in place of the real name. The name is your choice.
- Subtype differentiates between allocation calls and deallocation calls: 3 is for allocation and 4 is for deallocation.
- Function type is an integer between 1 and 15 which uniquely identifies the memory function. Numbers from 1 through 9 are used for CodeTEST internals (as you can see in the example). So numbers from 10 through 15 can be used for RTOS-specific calls (6 calls). This can look like:

```
_mem_alloc,  ct_rtos_mem_alloc,  3,  10, void*   ct_rtos_mem_alloc(unsigned long , _ct_tag_t);


_mem_free,   ct_rtos_mem_free,   4,  11, unsigned long ct_rtos_mem_free(void* , _ct_tag_t);
```

Memory wrappers

The wrappers must have a parameter representing the CodeTEST tag that will be used by the Instrumenter.

The memory wrappers must be created and put in a library that will be linked with the instrumented application. Here is how a memory wrapper looks:

```
#include "ct_rtos.h"

void *
ct_malloc( unsigned long nBytes, _ct_tag_t  tag )
{
    void *addr;

    addr = malloc(nBytes);
    if (addr == 0)
    {
            __ct_data_tag__( (_ct_tag_t ) NOT_ENOUGH_MEMORY ); /* error */
            __ct_data_tag__( (_ct_tag_t ) addr );           /* pointer */
            __ct_data_tag__( (_ct_tag_t ) nBytes );         /* size */
            __ct_ctrl_tag__( 0x29f00000 | ( tag & 0xfffff ));
            return addr;
    }

    __ct_data_tag__( (_ct_tag_t ) 0 );          /* heap ID */
    __ct_data_tag__( (_ct_tag_t ) addr );       /* pointer */
    __ct_data_tag__( (_ct_tag_t ) nBytes );           /* size */
    __ct_ctrl_tag__( tag );


    return(addr);
}

ct_free( void *addr, _ct_tag_t  tag )
{
    __ct_data_tag__( (_ct_tag_t ) 0 );          /* heap ID */
    __ct_data_tag__( (_ct_tag_t ) addr );             /* pointer */
    __ct_ctrl_tag__( tag );

    free(addr);
}
```

Like RTOS hook events, memory events consist of a control tag and several data tags.

- The first data tag that needs to be sent represents the Heap ID. This identifies each allocation/deallocation for a memory call. The same call, for example, malloc, could be used to allocate memory from different heaps (e.g., from different tasks). The CodeTEST “Virtual Memory” feature analyzes such cases. Otherwise, the tag is not used.
- The second data tag represents the pointer from where memory is allocated/deallocated.
- The third data tag appears just for allocation calls, and represents the size of memory allocated (in bytes).
- The control tag is the tag received by the wrapper as a parameter.

In case of an error at memory allocation, a memory error tag needs to be sent to `ctserver`. You can see that in this case the first data tag represents error code. Also the control tag is modified to signal a memory error to `ctserver`. You can use an error map file to map error codes with strings. An example of such a file in the CodeTEST installation is in the lib/rtos/*rtosname* directory.

C++ memory analysis support

C++ support for memory analysis can be found in the CodeTEST installation in the `ct_mem.cxx` file in the `lib/c++-memory` directory. The file header describes the way file could be modified if it is needed. This file needs to be built and linked with the application, if C++ memory analysis is desired. If not, it can be omitted.