Temporal Isolation CASE Scheduler: TICS

User Guide

January 15, 2022

Background

The Temporal Isolation CASE Scheduler (TICS) is a real-time userland scheduler that executes on the seL4 mixed criticality system (MCS) platform.[1] TICS has been implemented on the CAmkES runtime service layer. TICS supports the specification of multiple operational modes (e.g., initialization followed by runtime), each with its own schedule.

This document describes how to install and configure TICS within an seL4 environment for typical embedded system applications. Example configurations are also presented. The following background concepts are important to understand how to configure and build TICS, but are out-of-scope for this document:

- How to build and use the seL4 microkernel and the CAmkES runtime services,
- How to integrate user-defined applications into seL4 and CAmkES,
- How to use seL4 design and build support tools, such as HAMR or SPICA,
- How to model a real-time embedded system in AADL,
- Real-time scheduling theory and practice.

The goals, assumptions, and requirements for the TICS scheduler component is as follows:

Goals

1. Enforce temporal isolation between application threads, except as required over explicitly declared communications between threads.
2. Rely on existing security and timing capabilities of the seL4 MCS micro-kernel.
3. Support arbitrary number of applications within resource limitations.
4. Use CAmkES runtime framework to specify thread requirements.

Assumptions

1. The target platform is a single-core processor or that processor affinity is fixed to a single core. This assumption may be relaxed in the future.
2. TICS will release and constrain execution times for threads according to a static cyclic schedule provided at system compile time.[2]
3. Validation of a schedule is the responsibility of the system developer. TICS will attempt to execute the schedule as it is configured.

---

[1] This research was funded, in part, by the U.S. Government under prime contract #HR00111890001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

[2] This might also be relaxed in the future in favor of a partitioned design approach, where temporal resources are provided at the partition level, rather than the thread level. A scheduler within a partition would be work-conserving within the temporal confines of the partition.

Requirements

1. The schedule must support one or more modes.
2. All schedules are static cyclic.
3. Transition between scheduling modes is specified as number of major frame cycles.
4. Each entry in the static cyclic schedule contains a thread identifier and thread slice duration.
5. TICS initializes specified threads for time/space partitioned execution on the seL4 mixed criticality systems (MCS) target platform using appropriate timing attributes (e.g., budget and period), assigns the threads to an appropriate scheduling context, and passes the scheduling context to the seL4 MCS kernel for execution.
6. TICS is non-work-conserving. If a thread finishes before its allotted time, TICS will not provide the unused budget to other threads or to future activations of that thread. This preserves temporal isolation. Depending on application design, this might result in runtime performance inefficiency, since cycles are sacrificed.
7. TICS catches temporal exceptions from threads that exceed their assigned temporal allocation. Handling of those exceptions is responsibility of the TICS (NOTE: This feature is not yet implemented).

Installation

This section describes how to install TICS within a CAmkES/seL4 build environment.

The TICS prototype release includes the following content that adheres to the seL4/CAmkES build conventions:

- Source patch file `camkes-tool.patch`.
- CMake source files `CMakeLists.txt` and `settings.cmake`.
- Directory `components`, which contains the CAmkES component for the TICS scheduler, as well as a simple component for the example application.
- Directory `port-types`, which contains the C source files for the various data types enforced on the communication ports of the example application.
- A `README.org` file, which covers some of the information in this document.
- Directory `templates`, which contains the CAmkES templates used to integrate the TICS component and specifications into the full application build.
- The C source file `Tics-Prototype-Schedule.c`, which defines the static cyclic schedule for each scheduling mode in the system. The example application defines two scheduling modes, initialization and operational.
- The CAmkES source file `tics-prototype.camkes`, which specifies the components and configuration for the example application, including the TICS component.

A small change to the normal `camkes-tool` source is required to give TICS the access to the thread control blocks (TCBs) and scheduling contexts (SCs) of other threads. A patch for this change is in `camkes-tool.patch`. Installation of the patch is described below.[3]

These instructions assume the build environment directory is a direct sibling of the cloned CAmkES build directory. If you do not follow this convention, the symbolic links specified below will have to be adjusted. We will refer to the parent directory as `ROOT`.

First, copy the `tics-prototype` directory to `$ROOT`. Next, to setup the build environment with the patch to CAmkES:

```
cd $ROOT
mkdir tics-prototype-build-env
cd tics-prototype-build-env
repo init -u https://github.com/seL4/camkes-manifest.git
repo sync
cd projects/camkes/apps
ln -sf ../../../../tics-prototype tics-prototype
cd ../../camkes-tool
git apply --reject --whitespace=fix ../../../tics-prototype/camkes-tool.patch
```

Then build as you would a typical CAmkES application:

```
cd tics-prototype-build-env
mkdir build
cd build
../init-build.sh -DPLATFORM=pc99 -DSIMULATION=1 -DCAMKES_APP=tics-prototype
ninja
```

CAmkES Configuration

This section describes how to configure the TICS component within a CAmkES build environment.

The TICS scheduler uses the CAmkES `TimeServer` component. The TICS scheduler also requires specific configurations to operate correctly. All of these configuration requirements are defined in the `*.camkes` file for the application. First, the TICS component itself must be defined. For example:

```
import <std_connector.camkes>;
import <TimeServer/TimeServer.camkes>;
import <global-connectors.camkes>;

component Tics {
    control;
    uses Timer timeout;
}

// Other components go here…
…
```

[3] We are in the process of submitting this patch to the seL4 technical steering committee for consideration to be included in the baseline CAmkES release. As of this writing, this capability still requires the patch.

The TICS component is defined in addition to the application components. Next, the TICS scheduler must be connected to the `TimeServer` component. For example:

```
assembly {
    composition {
        component TimeServer time_server;

        // User application components and connections…
        …

        // Tics timer connection
        connection seL4TimeServer tics_timer(from tics.timeout, to time_server.the_timer);
    }
```

Lastly, the TICS scheduler and `TimeServer` must have their runtime attributes defined, following these guidelines:

- The TICS scheduler should run at a priority higher than all the scheduled application components, but lower than the `TimeServer`.
- The TICS scheduler should be a round-robin task. Its period and budget are set to a large arbitrary value (defined in microseconds). It simply needs to always be runnable.
- The `TimeServer` component requires only one timer for TICS.
- The task priorities need to be lower than tics.priority. Otherwise, the priorities are irrelevant since TICS controls when the tasks are released to run.

Here is an example configuration in the CAmkES file:

```
    configuration {
            // Make tics a round robin thread (period=budget) with
            // highest priority so it has complete control over when tasks
            // run.
            tics._priority = 254;
            tics._period = 1000000; // Round robin (period=budget), 1 second
            tics._budget = 1000000;
            tics.sched_ctrl = 0;    // Default CPU core 0

            // Task priority MUST be lower than tics. We leave other
            // thread attributes unspecified because the tics will
            // handle them (eg period and budget).
            task1._priority = 50;
            task2._priority = 50;
            task3._priority = 50;
            task4._priority = 50;

            // We only need one timer.
            time_server.timers_per_client = 1;
            // Time server must be highest priority
            time_server._priority = 255;
    }
```

TICS Schedule

The TICS schedule itself is defined in a `*.c` source file, using specific data structures. The following is a description of the data structures in an example with multiple scheduling modes. This example is captured in the `Tics-Prototype-Schedule.c` source file.

Threads

The `threads` data structure identifies the threads in the schedule and their implicit order and internal identification number. Each Thread in the schedule corresponds to a CAmkES component, has an entry in this data structure, and each entry has a `name` and `periodNotifyFn` field.

The `name` field is the named handle of the thread. The name MUST match the name used internally by the CAmkES tool (otherwise the application will not compile). For a typical CAmkES component the pattern is:

`COMPONENTNAME_0_control`

The "`_0_`" looks odd, but this format is hardcoded in the CAmkES tool. If you need other thread names, the safest thing to do is to build your CAmkES project and then look at the instantiated template code in your build directory. The path will be:

`BUILDDIR/tics/tcp.template.c`

Look at the generated `get_tcb_by_name`() function, which lists all the thread names.

The period-notify function is a pointer to a function customized by the system developer to return a notification reply to the sending component. This is an advanced feature required by virtualized CAmkES components. Non-virtualized components do not require a return period notification and should specify `NULL` in this field. TICS itself does not require period notifications. Non-virtualized tasks should use `seL4_Yield()` in their behavior code to wait for the next period (i.e., scheduling window. See later section on behavior code).

The unique identifier for each thread is implied by its order listed within the array. For example, the first thread has an identifier of 0, the second thread an identifier of 1, and so on.

An example threads definition is given here:

```
struct Thread threads[] = {
    // name                 periodNotifyFn
    {  "task1_0_control", NULL },
    {  "task2_0_control", NULL },
    {  "task3_0_control", NULL },
    {  "task4_0_control", NULL },
};
```

Windows

The `Window` data structure specifies the duration of time for a single thread to run within a schedule. A list of Window data structures defines the ordered sequence of time allotments to execute within the schedule for a single major frame of a particular scheduling mode.

Fields in each `Window` data structure include the thread number and a duration. The thread number is the identification number implied by the `threads` array data structure (see above). The identifier `-1` can be used to specify no thread (or idle thread) for a duration.

The `duration` field is a time amount defined in nanoseconds (ns) assigned to the thread's window.

Typically, the system developer will define a schedule for each mode in the system. For example, the following defines two schedules, one for an initialization mode and one for normal operational mode:

```
struct Window initWindows[] = {
    // Thread  Duration (ns)
    {  0, 0.5 * NS_IN_S },
    {  1, 1.0 * NS_IN_S },
    {  2, 0.5 * NS_IN_S },
    {  3, 1.5 * NS_IN_S },
};

struct Window normWindows[] = {
    // Thread  Duration (ns)
    {  0, 0.5 * NS_IN_S },
    {  1, 0.5 * NS_IN_S },
    { -1, 1.0 * NS_IN_S },
    {  0, 0.5 * NS_IN_S },
    {  1, 0.5 * NS_IN_S },
    {  2, 0.5 * NS_IN_S },
    {  3, 0.5 * NS_IN_S },
};
```

Modes

The `modes` data structure specifies a sequence of scheduling modes for the system. The first mode is the initial mode invoked by TICS.

The fields in each mode entry include name, number of cycles, scheduling windows to execute, and number of windows.

The `name` of the mode is a character string used for logging purposes only.

The number of cycles field defines the number of cycles (major frames) to execute before switching to the next mode in the sequence. This value must be greater than or equal to zero for all but the last mode. The last mode may specify a value of 0 to indicate that the mode runs indefinitely.

The scheduling `windows` field is a pointer to the array of `Window` data structures representing the schedule to execute in this mode.

The “number of windows” field must be the number of `Window` entries defined in the `windows` field. For example, the following defines the modes for the modal schedules defined earlier:

```
struct Mode modes[] = {
    { .name     = "Init",
      .nCycles  = 2,
      .windows  = initWindows,
      .nWindows = sizeof(initWindows)/sizeof(initWindows[0]),
    },
    { .name     = "Norm",
      .nCycles  = 0, // Run indefinitely
      .windows  = normWindows,
      .nWindows = sizeof(normWindows)/sizeof(normWindows[0]),
    },
};
```

Schedule

The `Schedule` data structure puts together the specification for the overall multi-mode schedule of the system. Fields in this data structure include the following.

- `threads`: Pointer to the list of threads in the system.
- `nThreads`: Number of threads in the system.
- `modes`: Pointer to the list of modes in the system.
- `nModes`: Number of modes in the system.

The global function `getTicsSchedule` is used by the TICS component to access the user-defined schedules. Here is the example, tying together the prior examples.

```
const struct Schedule schedule = {
    .threads  = threads,
    .nThreads = sizeof(threads)/sizeof(threads[0]),
    .modes  = modes,
    .nModes = sizeof(modes)/sizeof(modes[0]),
};

// Fixed function that returns the user-defined mode-based
// schedules.
const struct Schedule* const getTicsSchedule() {
    return &schedule;
}
```

The Schedule Source File

The user-defined schedule is captured in a `*.c` file that must be identified in the CAmkES `CMakeLists.txt` compilation file. The following example shows an entry in `CMakeLists.txt` for a user-defined schedule, named `Tics-Prototype-Scheduler.c`. An entry should include all lines of this example (`SOURCES`, `TEMPLATE_SOURCES`, `LIBS`, and `INCLUDES`).

```
DeclareCAmkESComponent(Tics
    SOURCES components/Tics/src/Tics.c port-types/sb_queue_int8_t_1.c Tics-Prototype-Schedule.c
    TEMPLATE_SOURCES tcb.template.c
    LIBS sel4utils sel4runtime
    INCLUDES templates components/Tics/src port-types
    )
```

Component Behavior Code

Component behavior code should use the `seL4_Yield` function within its main iteration loop to synchronize its timing with the TICS scheduler. Like all periodic tasks, a scheduled component within

the TICS environment should have an initialization phase followed by an iterative loop, where the main behavior resides. Here is a simple example for a ping-pong application. This example can be found in the `components/Task/src/Task.c` prototype source:

```
// Copyright 2021, Adventium Labs
//
// Example task that simple ping/pongs data via its dataIn and dataOut
// ports. It uses sel4_Yeild to wait for next TICS window.

// Override log level for this file. Comment to get application default.
#define ZF_LOG_LEVEL ZF_LOG_VERBOSE

#include <utils/zf_log.h>
#include <sel4utils/sel4_zf_logif.h>
#include <camkes.h>
#include <sel4bench/sel4bench.h>
#include "seqNum.h"
#include "sp_uintmax.h"

int run(void)
{
    const char* name = get_instance_name();

    uintmax_t data;
    seqNum_t dataIn_seqNum = 0;

    uint64_t ts, prev = (uint64_t)sel4bench_get_cycle_count();

    init_sp_uintmax(dataIn, &dataIn_seqNum);
    ZF_LOGD("    [%s] Started", name);

    // Run indefinitely
    for (int i = 0; ; i++) {
        // Yeild is period wait for periodci MCS task
        seL4_Yield();

        // Ping pong data
        read_sp_uintmax(dataIn, &data, &dataIn_seqNum);
        data++;
        write_sp_uintmax(dataIn, &data, &dataIn_seqNum);

        // TODO: convert cycle count to ms
        ts = (uint64_t)sel4bench_get_cycle_count();
        uint64_t diff = ts - prev;
        prev = ts;

        ZF_LOGD("    [%s] Tick: %5d  Period(cycles): %12"PRIu64"  Data: %"PRIu64"",
                name, i, diff, data);
        fflush(stdout);
    }

    ZF_LOGD("    [%d] Finished", name);
    return 0;
}
```

Transcribing from SPICA

As of this writing, the HAMR[4] system build tool does not target the MCS variant of seL4, and consequently does not support the auto-generation of a system schedule from an AADL model. In theory, a modeling and build tool like HAMR could automatically generate the seL4/CAmkES artifacts described in this document. In the meantime, thread execution schedules developed by a system developer must be transcribed manually into the TICS infrastructure.

The process of developing and refining a workable thread execution schedule for complex embedded systems can be a significant engineering challenge, and system developers often utilize timing analysis tools to generate valid schedules. The Adventium Labs SPICA[5] tool analyzes timing properties within an AADL system model and generates a thread execution schedule as an extension to the AADL model if a valid schedule is feasible. The following section describes how to convert a SPICA generated schedule into a TICS implementation.

Consider the following AADL model, which is the typical output generated by the SPICA tool for a simple four task system, like the one described in prior sections. The model specifies a thread execution schedule for each of the four application tasks:

```
package TICSPrototype_TICS_Example_Dsch
public
    with ARINC653;
    with TICSPrototype;

    TICS renames system TICSPrototype::TICS;
    system implementation TICS.Example extends TICSPrototype::TICS.Example
    properties
        ARINC653::Module_Major_Frame => 800 ms applies to TICS.CPU;
        ARINC653::Module_Schedule => (
            [Partition => reference(TICS.Task1_Partition); Duration => 100 ms;
                          Periodic_Processing_Start => true; ], -- 0..100
            [Partition => reference(TICS.Task2_Partition); Duration => 150 ms;
                          Periodic_Processing_Start => true; ], -- 100..250
            [Partition => reference(TICS.Task3_Partition); Duration => 100 ms;
                          Periodic_Processing_Start => true; ], -- 250..350
            [Partition => reference(TICS.Task4_Partition); Duration => 150 ms;
                          Periodic_Processing_Start => true; ]  -- 350..500
        ) applies to TICS.CPU;
        Priority => 50 applies to TICS.Software.Task1;
        Priority => 50 applies to TICS.Software.Task2;
        Priority => 50 applies to TICS.Software.Task3;
        Priority => 50 applies to TICS.Software.Task4;
    end TICS.Example;

end TICSPrototype_TICS_Example_Dsch;
```

[4] The High Assurance Modeling and Rapid Engineering (HAMR) tool is developed and maintained by Kansas State University. For background, see https://www.youtube.com/watch?v=ceCJ_F3sjBc.
[5] Separation Platform for Integrating Complex Avionics (SPICA), for an introduction see https://www.youtube.com/watch?v=d4LIO_JkFSE.

The partition schedule details are added to the `*.c` file representing the TICS schedule. In the ongoing example, the `Tics-Prototype-Schedule.c` file would look like:

```
// Copyright 2021, Adventium Labs
// Example TICS Schedule instance for use with tics-prototyle.camkes.

#include <stddef.h>
#include <stdint.h>
#include <sel4/sel4.h>
#include <Tics-Schedule.h>
#include <platsupport/ltimer.h>
#include <camkes.h>

struct Thread threads[] = {
    // name               periodNotifyFn
    { "task1_0_control",  NULL },
    { "task2_0_control",  NULL },
    { "task3_0_control",  NULL },
    { "task4_0_control",  NULL },
};

struct Window basicWindows[] = {
    // Thread  Dur (ns)
    {  0, 0.1  * NS_IN_S },
    {  1, 0.15 * NS_IN_S },
    {  2, 0.1  * NS_IN_S },
    {  3, 0.15 * NS_IN_S },
};

struct Mode modes[] = {
    { .name     = "Basic",
      .nCycles  = 0,
      .windows  = basicWindows,
      .nWindows = sizeof(basicWindows)/sizeof(basicWindows[0]),
    }
};

const struct Schedule schedule = {
    .threads  = threads,
    .nThreads = sizeof(threads)/sizeof(threads[0]),
    .modes  = modes,
    .nModes = sizeof(modes)/sizeof(modes[0]),
};

const struct Schedule* const getTicsSchedule() {
    return &schedule;
};
```

This example specifies a single scheduling mode. For multi-mode systems, you would add another `Window` definition for each additional mode and then include each new scheduling window within the `Mode` data structure.

If the system does have multiple scheduling modes, then the `nCycles` field of each `Mode` data structure must specify how many major frame cycles are completed before transitioning to the next mode. If the `nCycles` field is zero, then the TICS scheduler runs that mode indefinitely without transitioning to another mode.