SYS/BIOS for Stellaris Devices

This page summarizes some features specific to the ARM Cortex-M4-based TM4C12x microcontrollers, and implementation details of SYS/BIOS for TM4C12x devices.

On this page, SYS/BIOS configuration settings are described with configuration script snippets. These configuration steps can also be accomplished via the XGCONF Graphical Configuration Tool, but that is not described here.

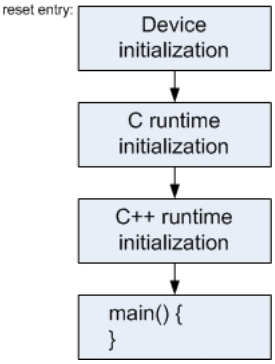
Contents

Overview of the SYS/BIOS boot sequence

This section summarizes the bootstrap sequence for SYS/BIOS. It begins with a review of the boot sequence without SYS/BIOS. This description includes some TM4C12x-specific items, but the general flow of the boot sequence applies to all the device families supported by SYS/BIOS.

Normal boot sequence, without SYS/BIOS

The picture below summarizes the typical device boot sequence to main(), without SYS/BIOS. The next section will describe SYS/BIOS additions to this sequence. The source files mentioned in this section can be found in the rtssrc.zip file that typically resides in this directory: {CCS Install Directory}\tools\compiler\tms470\lib



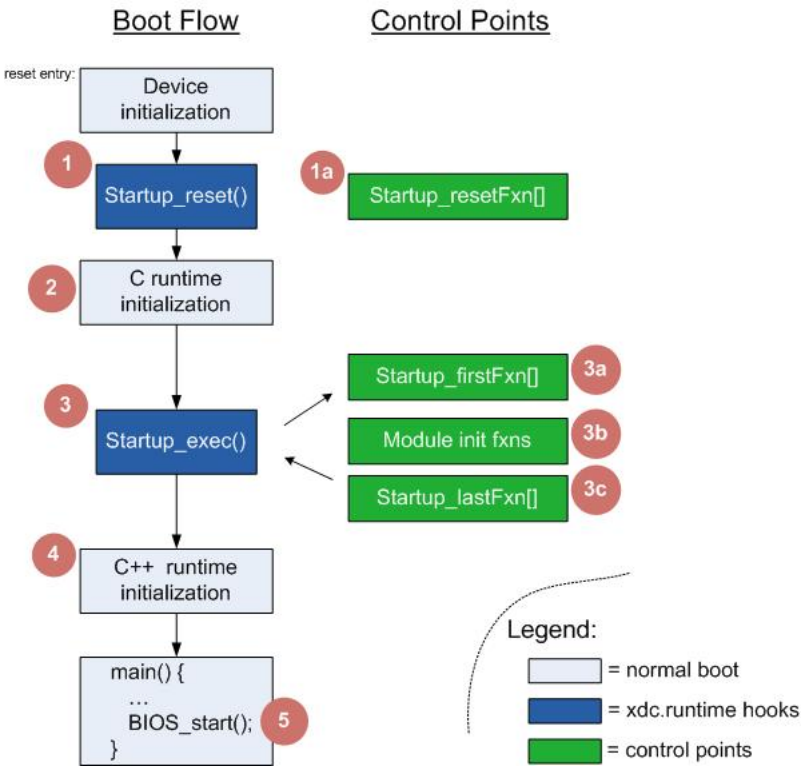
When the device is released from reset, the reset vector will initiate C runtime initialization, at the _c_int00() entry point (in the file "boot.asm"). The first step to initialize the device is to initialize the system stack pointer (to the value defined by the symbol __STACK_END). This same stack is used during initial booting, as well as in and after main(). The size of the system stack defaults to 2048 bytes, but can be overridden via the "--stack_size" or "-stack" linker option (defined via CCS project Build Properties->Tool Settings->TMS470 Linker->Basic Options).

Once the stack pointer is initialized, the __TI_auto_init() function (in the file "auto_init.asm") will be called. The auto_init() function will process the global variable initialization records (the "cinit" records) to initialize the global variables in the .bss section. After processing these records, auto_init() will traverse the table of .pinit functions (if any are defined), and call each function. These .pinit functions are typically C++ constructor functions, but can be used for other purposes as well. Once the table of .pinit functions has been processed, __TI_auto_init() returns back to _c_int00(), which will then call main(). If there are arguments to pass to main() (depending upon the application build options), the args_main() function (in the file SHARED\args_main.c) is used to call main(). At this point, the C runtime environment has been fully initialized, and the application can begin its intended purpose.

Boot sequence with SYS/BIOS

The picture below shows additions to the boot sequence when using SYS/BIOS. The dark blue boxes indicate hook mechanisms into the normal boot flow. The green boxes indicate the additional control points that result, and allow calling of SYS/BIOS-defined, as well as user-defined functions, at progressive stages of the boot flow. The red numeric bubbles are used to reference the description below.

Note that for this section there are mentions to source files that are modified versions of those provided with TI's Arm compiler (which were mentioned in the previous section). The modified files referenced here are delivered with XDCtools, and can be found at this location: {XDCtools Install Dir}\packages\ti\targets\arm\rtsarm



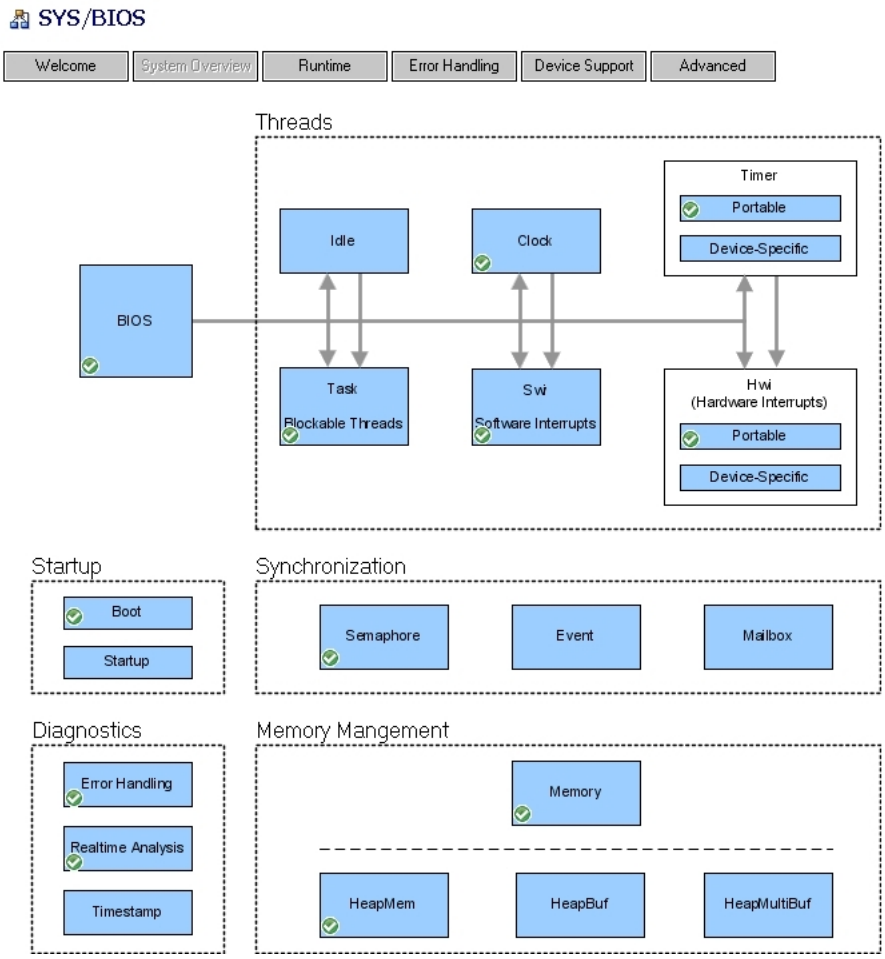
1. Startup_reset() refers to a hook mechanism added to the TI cgtools runtime support library's boot.asm file to allow a table of functions (see '1a' below) to be called immediately after the stack pointer has been initialized. The Startup module resides in xdc.runtime.Startup. Since only the stack pointer is initialized at this point, these functions must not reference any global variables, because global variables have not been initialized yet, and their locations will be overwritten later in the boot sequence.

1a. For Stellaris devices SYS/BIOS installs a configurable Startup reset function that:

- Sets up the device's internal clocking mechanism to achieve the desired CPU frequency.
- Sets the Low Dropout Voltage (LDO) voltage.

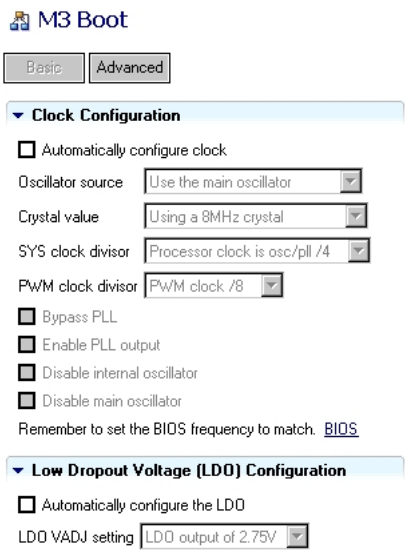
By default, the Startup reset function sets the CPU frequency to the maximum supported by the device. This behavior can be modified by carefully configuring the various parameters provided in the 'ti.catalog.arm.cortexm3.lm3init.Boot' module.

Select the Boot module from the SYS/BIOS System Overview page in CCS:



and configure the various parameters provided.

To suppress the functionality of the 'ti.catalog.arm.cortexm3.lm3init.Boot' module uncheck the "Automatically configure clock" and "Automatically configure the LDO" check boxes in the Boot module GUI:



Or, within your config script:

<syntaxhighlight lang='javascript'> var Boot = xdc.useModule('ti.catalog.arm.cortexm3.lm3init.Boot'); Boot.configureClock = false; Boot.configureLdo = false; </syntaxhighlight>

If you want to add your own functions to the table of early reset functions, you can do so by adding statements like the following to your application configuration file:

<syntaxhighlight lang='javascript'> Reset = xdc.useModule('xdc.runtime.Reset'); Reset.fxns[Reset.fxns.length++] = '&myResetFxn'; </syntaxhighlight>

2. After the table of Startup reset functions has been processed, booting continues in _c_int00(). The next step is to call the auto_init() routine to process .cinit records to initialize global variables.

3. After .cinit records have been processed, additional SYS/BIOS and user-defined initialization functions can be called via addition of the Startup_exec() hook mechanism added to auto_init() (in file autoinit.asm). Since global variables have been initialized, these functions have much more freedom in what they can do, compared to the early reset functions described in 1 and 1a above. One thing that they must not do is to enable global interrupts, because that would allow interrupts to fire before the bootstrap sequence is completed. The startup functions are partitioned into three categories: first functions, module initialization functions, and last functions.

3a. First functions execute before module initialization or last functions. First functions are referenced in a table, and each function in the table is called before proceeding to the module initialization functions. SYS/BIOS typically provides one first function: a function that initializes the system stack with a 'watermark' value; this allows checking of the stack later, for depth of use, and overflow.

If you want to add your own first function to be called at this stage, you can do so by adding statements like the following to your application configuration file (*.cfg):

<syntaxhighlight lang='javascript'> var Startup = xdc.useModule('xdc.runtime.Startup'); Startup.firstFxns[Startup.firstFxns.length++] = '&myFirstFxn'; </syntaxhighlight>

3b. Module initialization functions are used to initialize the SYS/BIOS or other modules that have been configured into the application. A typical use for these functions is to initialize statically-configured instances of module objects. For example, if the application configuration file statically created two semaphores, the Semaphore module will initialize the corresponding data structures at this stage of boot. Note that this stage only applies to *module* initialization; it cannot be hooked into for application-level initialization purposes.

3c. The last functions will be invoked after all regular SYS/BIOS module initialization functions have executed. At this stage, SYS/BIOS will typically insert a function to start the Timestamp counter (if enabled in the application configuration).

If you want to add your own last functions to be called at this state, you can add statements like the following to the application configuration file:

<syntaxhighlight lang='javascript'> var Startup = xdc.useModule('xdc.runtime.Startup'); Startup.lastFxns[Startup.lastFxns.length++] = '&myLastFxn'; </syntaxhighlight>

4. After all last functions have been executed, control returns back to auto_init(), which will proceed to invoke any .pinit functions that have been defined. After this, control returns back to boot.c, which then calls to main().

Once in main() the C runtime has been fully initialized, and SYS/BIOS modules have been initialized. But SYS/BIOS has not “started up” yet; this will happen at the end of main(), when BIOS_start() is called. Before that point, the application can do many things, like creating new threads, doing application-specific initializations, etc. Two things that the application must not do at this point are: do a global interrupt enable, or to call a SYS/BIOS API that blocks execution waiting for some condition. These types of calls must wait until after BIOS_start() is invoked. See each SYS/BIOS module’s API descriptions to see the valid calling contexts.

5. The final phase of the startup process begins when the user's main() function calls BIOS_start() after all the user initialization code has executed. Note that once SYS/BIOS is "started up", control will never return back to main().

The final steps for SYS/BIOS startup are:

a. Execute any user-defined BIOS startup functions. If you want to define a function that is called at this stage, you can add code like the following to your application configuration script:

<syntaxhighlight lang='javascript'> var BIOS = xdc.useModule('ti.sysbios.BIOS'); BIOS.addUserStartupFunction('&myBiosStartFxn'); </syntaxhighlight>

b. Enable servicing of hardware interrupts. Any interrupts that are fully enabled can now be triggered.

c. Initialize any Timers that have been configured for the application. If individual timers are configured to "auto start", start them now.

d. If enabled within the application configuration, enable the Software Interrupt (Swi) scheduler. If any Swis are ready to run (i.e., they have been "posted"), they will preempt the startup sequence at this point, and control returns here when no Swis are ready to run.

e. If enabled within the application configuration, enable the Task scheduler. If any user-defined Tasks are ready to run they will run at this point. When none are ready, the Idle Task runs.

f. If execution reaches this point (because Tasks are not enabled in the application configuration), any configured Idle functions will be run in a continuous loop.

SYS/BIOS M4 Hardware Interrupt (Hwi) Handling

The ARM Cortex M4 CPU within TM4C12x devices natively supports efficient interrupt handlers written in 'C'. In order to support the 3 tiered threading model provided by SYS/BIOS (ie Hwis, Swis, and Tasks), the native interrupt support provided by the M4 core must be augmented with a few house-keeping functions that enforce the expected thread execution behavior.

To consolidate the overhead of these house keeping functions and provide a standardized set of instrumentation features, SYS/BIOS uses a centralized interrupt dispatcher that invokes the application's custom ISR (Hwi) functions.

The interrupt dispatcher provides the following functionality:

- The interaction of Hwi, Swi, and Task threads is carefully orchestrated:
  - Hwi threads can schedule the execution of Swi threads by means of the various Swi_post() APIs
  - Hwi threads can schedule the execution of Task threads by means of the Semaphore_post() and Event_post() APIs
- Application specific Hwi 'begin' and 'end' hook functions are safely invoked at standardized times.
- Informational 'Log' events (if enabled) are issued at standardized times.

By default, all SYS/BIOS managed interrupts are routed to the interrupt dispatcher which subsequently invokes the user's interrupt handler.

For details of the TM4C12x Hwi module, read the CDOC pages here (http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/bios/sysbios/6_31_04_27/exports/bios_6_31_04_27/docs/cdoc/ti/sysbios/family/arm/m3/Hwi.html)

Hwi MaskingOptions and Priorities

Supported MaskingOptions

The Cortex M4's Nested Vectored Interrupt Controller (NVIC) natively supports configurable interrupt priorities.

The SYS/BIOS interrupt dispatcher used in Stellaris devices is designed specifically to support the native interrupt nesting functionality of the NVIC. Consequently, only the **Hwi.MaskingOption_LOWER** enumeration is honored for an individual Hwi's params.maskingOption setting.

No support is provided for the following Hwi masking options:

- Hwi.MaskingOption_NONE,
- Hwi.MaskingOption_ALL,
- Hwi.MaskingOption_SELF,
- Hwi.MaskingOption_BITMASK,

Supported Priority Values

While the ARM Cortex M4 can be implemented to support up to 256 priority settings, the amount of internal chip real-estate required to support this number of priorities is exceeding large. Since most embedded system applications can be fully supported with much fewer interrupt priorities, the TM4C12x devices are designed to support 8 priorities.

Due to a hardware design subtlety, the 3 bits of priority required to define the 8 priority values occupy bits [7:5] of the 8 bit priority value rather than bits [2:0] as one might expect. Consequently, the values of the 8 supported priorities are not 0x00 thru 0x07 but 0x00, 0x20, 0x40, 0x60, 0x80, 0xa0, 0xc0, and 0xe0, where 0x00 is the HIGHEST priority and 0xe0 the LOWEST priority.

Additionally, the NVIC design defines interrupt PRIORITY GROUPS within a given priority setting. The Hwi.priGroup module configuration parameter is provided so the user can specify which of the 3 significant bits of the priority setting are group bits and which are priority bits.

A thorough discussion of the NVIC's priority and priority-group behavior is beyond the scope of this overview. For a more complete understanding, the user is referred to the ARM-provided online CORTEX M3 documentation of the Interrupt Priority Registers and Application Interrupt and Reset Control registers located here (http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/Cihcbadd.html#Cihgjeed)

No validity checking is performed on a user's Hwi params.priority setting.

It is up to the user to make sure that their configured interrupt priorities will be effective for their application.

Zero Latency Interrupt Support

The M4 Hwi module used by TM4C12x devices supports Zero Latency Interrupts. When configured, a Zero Latency Interrupt is NEVER disabled internally by SYS/BIOS functions, not even during critical thread execution. Thus, a zero latency interrupt handler should never call any BIOS APIs . See the 'How To' below for an example of configuring a zero latency interrupt.

Zero Latency interrupts fall into the commonly used category of **Unmanaged Interrupts**. However they are somewhat distinct from that definition in that in addition to being unmanaged, they are also almost never disabled by SYS/BIOS code, thus gaining the **Zero Latency** title.

SYS/BIOS M4 Operating Modes and Stack Usage

M3 Operating Modes

The Cortex M4 internally supports two operating modes: 'Thread' mode and 'Handler' mode. Within 'Thread' mode, the M4 code can run in either 'Privileged' or 'User' mode. By definition, 'Handler' mode code always runs in 'Privileged' mode.

Within SYS/BIOS:

- **Swi and Task threads run in 'Thread' mode.**
- **Hwi threads run in 'Handler' mode.**
- **All SYS/BIOS internal code as well as application code (ie Hwi, Swi, and Task threads) run in 'Privileged' code mode.**
- **For performance reasons, SYS/BIOS does NOT support 'User' code mode.**

Stacks

The Cortex M4 internally supports two stacks: the 'Main' stack and the 'Process' stack.

After reset and throughout the entire boot up sequence prior to the BIOS_start() invocation in main(), the M4 core is executing in 'Thread' mode and using the 'Main' stack.

The 'Main' stack ALWAYS refers to the stack buffer defined and managed by the codegen tools (ie the buffer which is placed in the ".stack" section and who's size is configured using the "-stack XXX" linker command option).

If Tasks are supported by the application (ie **BIOS.taskEnabled = true**), then within the Task_startup() function which is called by BIOS_start(), the M4 processor is configured to use the 'Process' stack while in 'Thread' mode and to automatically switch to the 'Main' stack when an interrupt occurs (ie when transitioning to 'Handler' mode). When execution is subsequently passed to a Task thread, the 'Process' stack then becomes dedicated to Task threads and the 'Main' stack becomes dedicated to Hwi and Swi threads.

As opposed to Hwi threads which automatically switch to the 'Main' stack when 'Handler' mode is entered, the Swi scheduler manually switches the processor to and from the 'Main' stack before and after Swi threads are run.

If Tasks are not supported by the application (ie **BIOS.taskEnabled = false**), the split between 'Process' stack and 'Main' stack is not made. All threads run on the 'Main' stack. Consequently, all configured Idle functions (ie the application's background functions) share the same stack as Hwi and Swi threads.

Within SYS/BIOS:

- **SYS/BIOS Hwi and Swi threads always use the 'Main' stack.**
- **SYS/BIOS Task threads always use the 'Process' stack.**

SYS/BIOS TM4C12x Timer Support

By default, the SYS/BIOS Clock module uses a timer provided by the "ti.sysbios.family.arm.lm3.Timer" module to automatically configure its periodic tick interrupt source. This Timer module manages the LM3 device family's General-Purpose 32-bit timers. The number of timers managed by the Timer module varies depending on the TM4C12x device selected.

SYS/BIOS ordinarily **DOES NOT** use the M4 core's internal SysTick timer, which is managed by the "ti.sysbios.family.arm.m3.Timer" module. See the **How To** below that describes how to use the SysTick timer in an application.

HOW TOs

Configuring a Zero Latency Interrupt

To define an interrupt that is NEVER disabled by SYS/BIOS, use the normal Hwi.create() / Hwi_create() APIs and set the params.priority value to be anything below the Hwi.disablePriority setting. By default, priority 0 is reserved for zero latency interrupts.

Remember that zero latency interrupts are not handled by the SYS/BIOS interrupt dispatcher and are NEVER DISABLED. Consequently, in order to guarantee the integrity of the SYS/BIOS thread schedulers, zero latency interrupt handlers are severely restricted in terms of SYS/BIOS APIs they can invoke. Specifically the following APIs can not be called within a zero latency interrupt handler:

- **Semaphore_post();**
- **Event_post();**
- **Swi_post(); /* nor any of its derivatives */**
- **System_printf(); /* Unless the interrupt handler is the only user of this API */**

Below is a static configuration example of mapping the function 'myZeroLatencyHwi' to the GPIO Port A interrupt (id = 16):

<syntaxhighlight lang='javascript'> var Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi');

var hwiParams = new Hwi.Params(); hwiParams.priority = 0; Hwi.create(16, '&myZeroLatencyHwi', hwiParams); </syntaxhighlight>

Below is the corresponding C code for dynamically configuring the same interrupt:

<syntaxhighlight lang='c'>

1. include <xdc/std.h>
2. include <ti/sysbios/BIOS.h>
3. include <ti/sysbios/family/arm/m3/Hwi.h>

Int main(Int argc, Char* argv[]) {

```
  Hwi_Params params;
```

```
  Hwi_Params_init(&params);
  params.priority = 0;
  Hwi_create(16, myZeroLatencyHwi, &params, NULL);
```

```
  BIOS_start();
```

} </syntaxhighlight>

How to build your application to run from RAM rather than FLASH

To reduce FLASH programming cycles during application development, you can place all the application's code and data in IRAM rather than in FLASH.

Assuming your device has enough internal RAM to contain all of your application, add the following to your config script to force all code/data to be placed into IRAM:

<syntaxhighlight lang='javascript'> var Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi'); Hwi.resetVectorAddress = 0x20000000; Program.sectMap[".text"] = "IRAM"; Program.sectMap[".stack"] = "IRAM"; Program.sectMap[".bss"] = "IRAM"; Program.sectMap[".neardata"] = "IRAM"; Program.sectMap[".rodata"] = "IRAM"; Program.sectMap[".cinit"] = "IRAM"; Program.sectMap[".init_array"] = "IRAM"; Program.sectMap[".const"] = "IRAM"; Program.sectMap[".data"] = "IRAM"; Program.sectMap[".fardata"] = "IRAM"; Program.sectMap[".switch"] = "IRAM"; Program.sectMap[".sysmem"] = "IRAM"; Program.sectMap[".far"] = "IRAM"; Program.sectMap[".args"] = "IRAM"; Program.sectMap[".cio"] = "IRAM"; Program.sectMap["xdc.meta"] = "IRAM"; </syntaxhighlight>

Configure Divide-by-Zero to be an Exception

By default, out of reset, the M3 core does not treat Divide-by-Zero as an exception. To configure the M3 to treat Divide-by-Zero as an exception, add the following to your config script:

<syntaxhighlight lang='javascript'> var Hwi = xdc.useModule('ti.sysbios.family.arm.m3.Hwi'); Hwi.nvicCCR.DIV_0_TRP = 1; </syntaxhighlight>

Using the SysTick Timer

By default, SYS/BIOS uses the 32-bit General-Purpose timers managed by the "ti.sysbios.family.arm.lm3.Timer" module to provide the Clock module's periodic tick interrupt.

The M4 core's internal SysTick timer is managed by the "ti.sysbios.family.arm.m3.Timer" module.

To use the SysTick timer as the periodic Clock tick source, add the following to your config script:

<syntaxhighlight lang='javascript'> var halTimer = xdc.useModule('ti.sysbios.hal.Timer'); halTimer.TimerProxy = xdc.useModule('ti.sysbios.family.arm.m3.Timer'); </syntaxhighlight>

Beware that the SysTick timer module only manages ONE timer and the above configuration change will dedicate that timer for use by the Clock module. Consequently, applications that use the "ti.sysbios.hal.Timer" proxy module for additional timer resources will fail to build/run due to there being no timers available.

To overcome this problem, the "ti.sysbios.family.arm.lm3.Timer" module can be used directly by the application for any additional timer needs. Simply add the following to the config script:

<syntaxhighlight lang='javascript'> var Timer = xdc.useModule('ti.sysbios.family.arm.lm3.Timer'); </syntaxhighlight>

In place of any existing usage of the "ti.sysbios.hal.Timer" module.

Also, change the corresponding C code #include lines to:

<syntaxhighlight lang='c'>

1. include <ti/sysbios/family/arm/lm3/Timer.h>

</syntaxhighlight>

Training

The "Intro to TI-RTOS Kernel Workshop" is now available. Follow the link below to find out more. The TI-RTOS Kernel Workshop covers the SYS/BIOS operating system available for all TI embedded processors - C28x, MSP430, Tiva-C, C6000 and AM335x (Cortex A-8). You can take a LIVE workshop (scheduled at various sites around the U.S.) or download/stream the videos of each chapter online and watch at your own pace. All of the labs, solutions, powerpoint slides, student guides, installation instructions, lab procedures, etc., are all available to you. The workshop labs run on all MCU platforms and the C6000. Check it out...

Intro to TI-RTOS Kernel Workshop (http://processors.wiki.ti.com/index.php/TI-RTOS_Workshop)

{{

1. switchcategory:MultiCore=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **SYS/BIOS for Stellaris Devices** here.

Keystone=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **SYS/BIOS for Stellaris Devices** here.

C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **SYS/BIOS for Stellaris Devices** here.*

DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **SYS/BIOS for Stellaris Devices** here.*

MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **SYS/BIOS for Stellaris Devices** here.*

OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **SYS/BIOS for Stellaris Devices** here.*

OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **SYS/BIOS for Stellaris Devices** here.*

MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **SYS/BIOS for Stellaris Devices** here.*

*For technical su please post you questions at http://e2e.ti.cor Please post on comments abo article **SYS/BIO Stellaris Devic** here.*

}}

Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters
DLP & MEMS
High-Reliability
Interface
Logic
Power Management
Processors

- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity