SMART ARM-based Microcontrollers

AT03664: Getting Started with FreeRTOS on SAM D20/D21/R21/L21/L22

APPLICATION NOTE

Introduction

Operating systems appear to allow multiple concurrent tasks to be executed simultaneously. Actually, the operating system executes each task for a short time and then rapidly switches between them without the user noticing. This is referred to as multi-tasking. FreeRTOS™ is a light-weight Real Time Operating System (RTOS) which allows multi-tasking on microcontrollers such as SAM MCUs. Readers who are not familiar with the RTOS concept and would like to know more can refer "What is an RTOS?" and "Why use an RTOS?" at the FreeRTOS website: http://www.freertos.org.

Features

This application note describes the steps to get started with FreeRTOS™ on Atmel® | SMART ARM®-based microcontrollers by showing the following steps and describing the demo application.

- To start a FreeRTOS Project in Atmel Studio
- To configure FreeRTOS
- To use the Atmel provided ASF drivers with FreeRTOS

The following devices can use this module:

- Atmel | SMART SAM D20
- Atmel | SMART SAM D21
- Atmel | SMART SAM R21
- Atmel | SMART SAM L21
- Atmel | SMART SAM L22

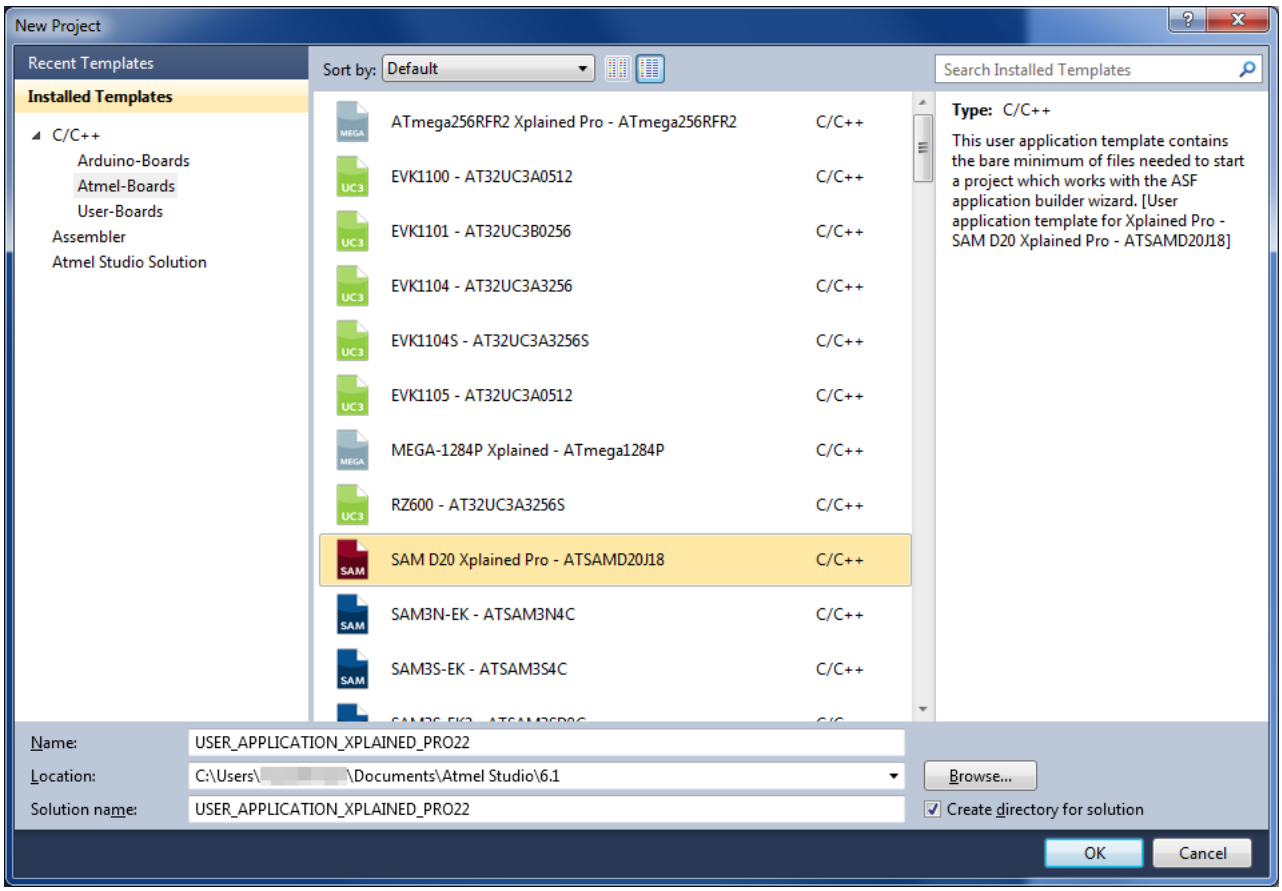
Table of Contents

1. Creating a FreeRTOS Project in Atmel Studio

To create a FreeRTOS project in Atmel Studio, start with either a user board or Atmel board template project for a SAM MCU device, such as the SAM D20 Xplained Pro. The following screen capture shows the **New Project** dialog in this case.

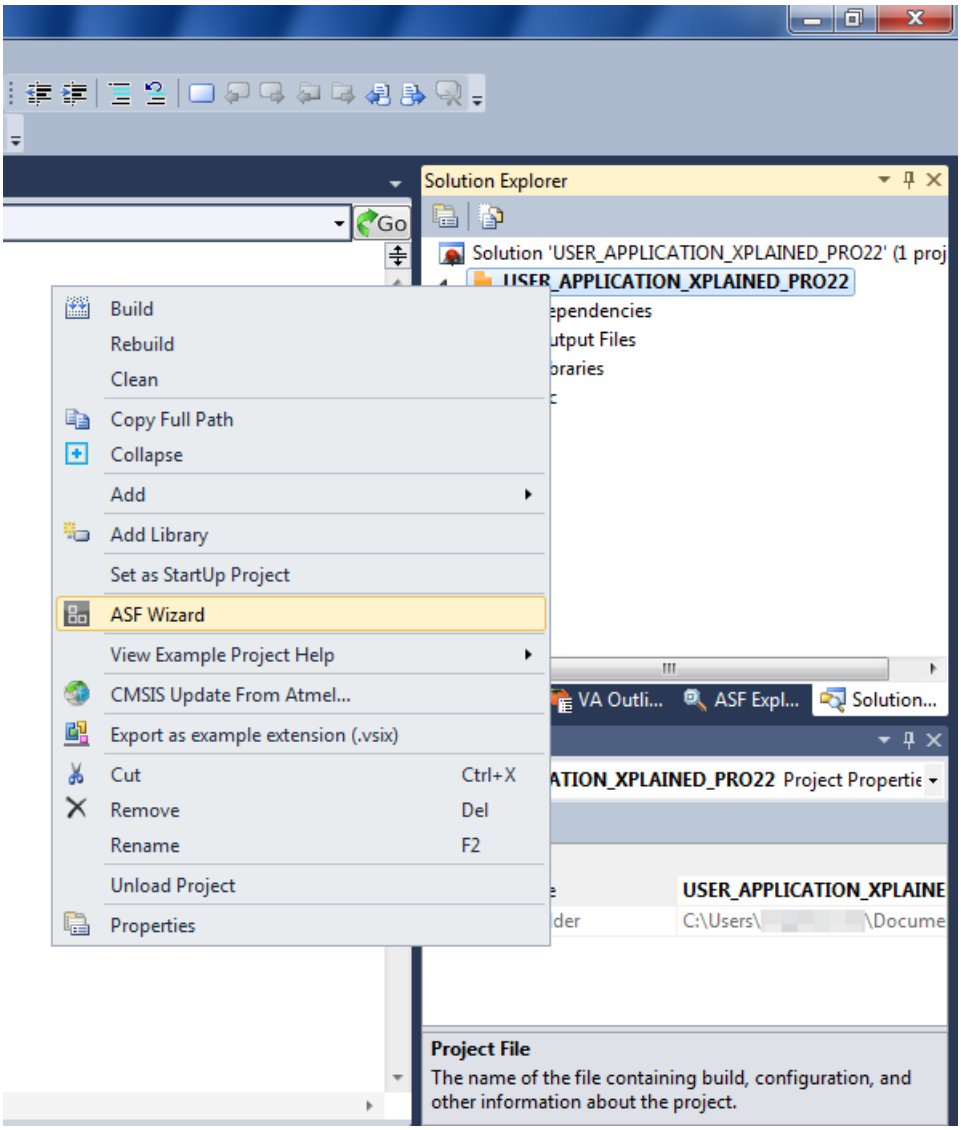
To open this dialog, click **File > New > Project** in the menu bar or press Ctrl + Shift + N.

Figure 1-1. Creating a Project From the SAM D20 Xplained Pro Template



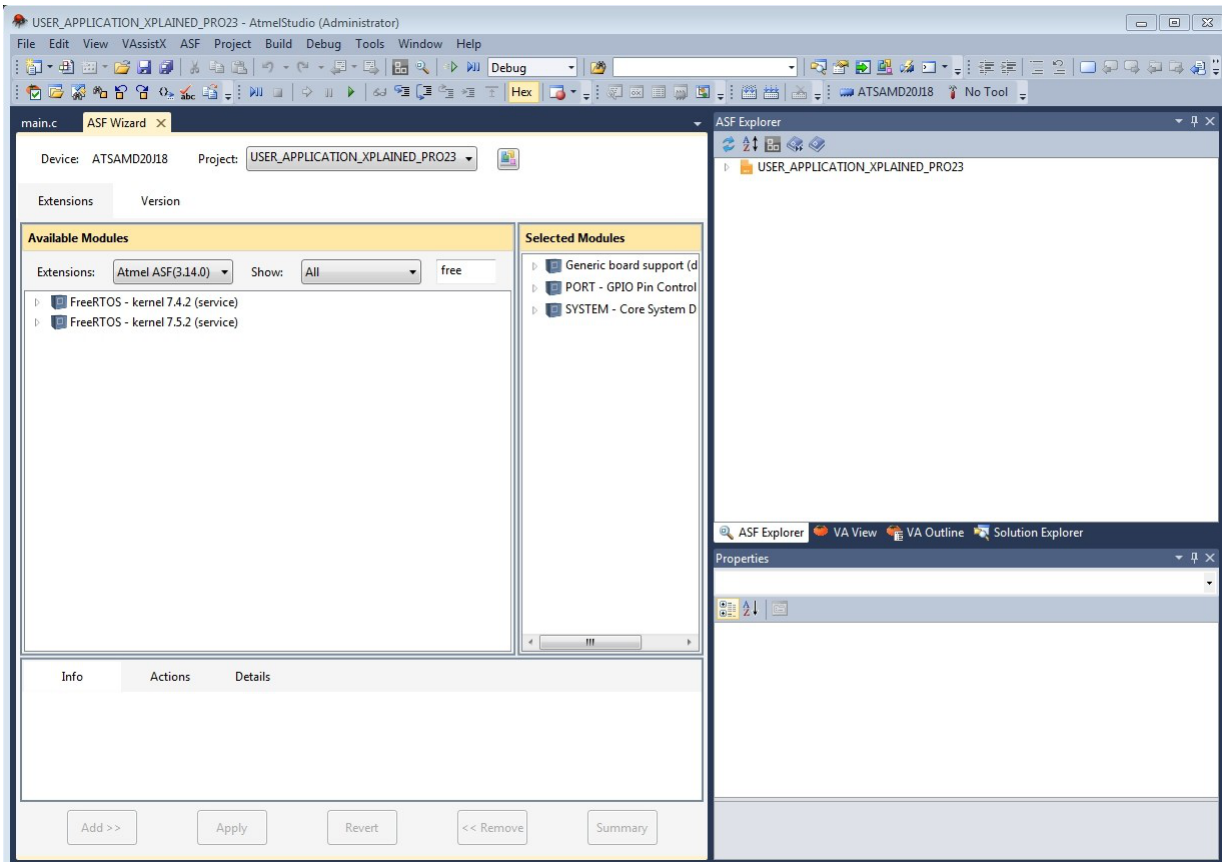
After creating the new project, the FreeRTOS must be added to the project. Start the ASF Wizard by right-clicking the project and select **ASF Wizard** in the context menu, as shown in the following image.

Figure 1-2. Starting the ASF Wizard



FreeRTOS must be selected in the list of available modules, on the left panel in the ASF Wizard, and added to the project. Instead of searching through the list, you can enter "free" as a filtering term to narrow down the list of module names. This is shown in the following figure.

Figure 1-3. Adding FreeRTOS to the Project



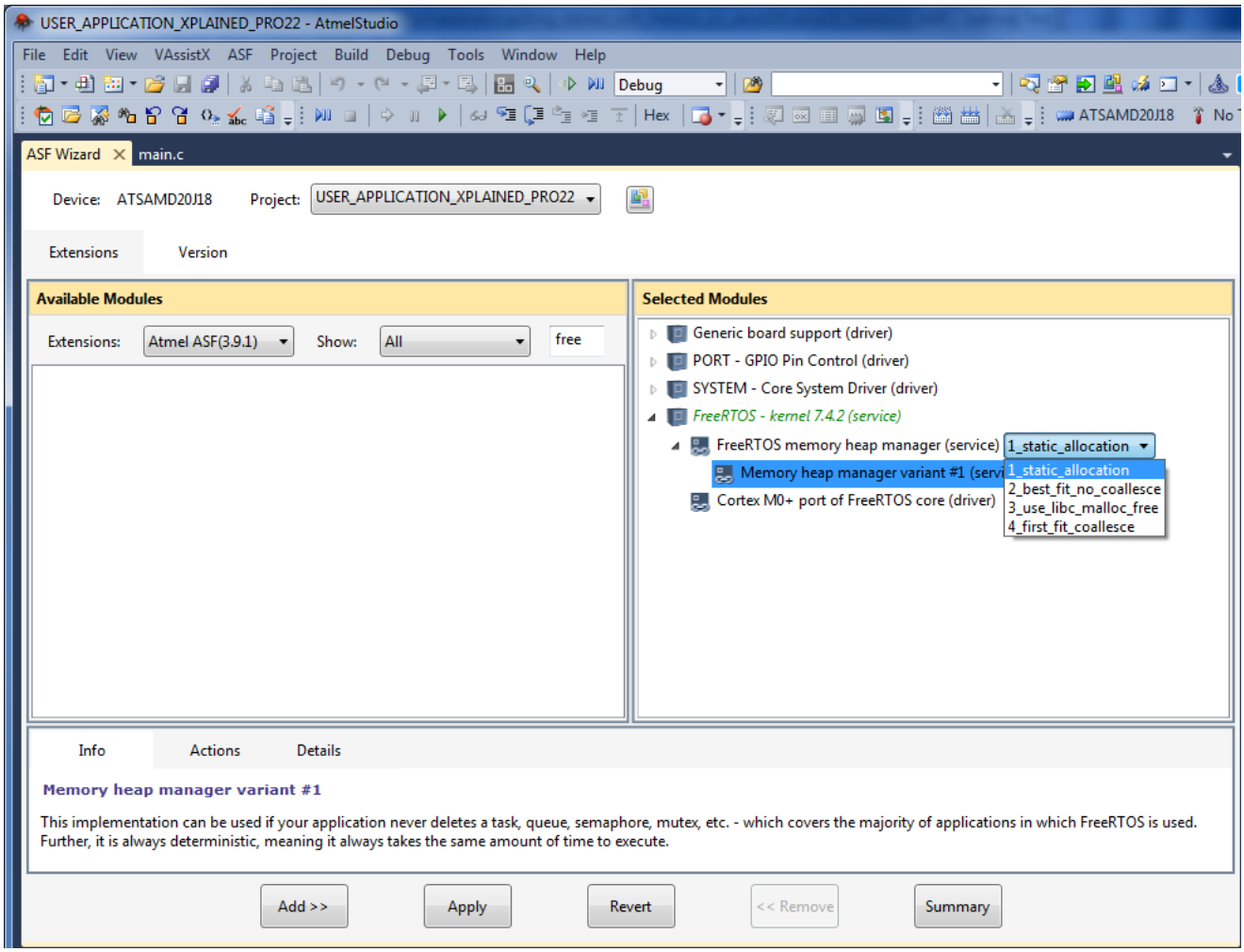
FreeRTOS v7.4.2 is provided for SAM MCUs and an additional version of v7.5.2 is provided from ASF v3.14.0 or newer. A default tickless feature is implemented in v7.5.2, which allows longer sleep periods by shutting down the OS tick when it is not needed. This feature can be toggled on or off to compare power consumption.

After FreeRTOS has been selected, press the **Add** button at the bottom of the wizard to add it to the project. This will move it to the list on the right side of the ASF Wizard, with green text signifying that it is a *staged* change.

You might require to change the selection of which of the four memory manager variants to use in the project. Expand the FreeRTOS dependency tree to reveal the *FreeRTOS memory heap manager* service, as shown in the following figure. Short descriptions of the variants can be read in the info box at the bottom. For more details on the various variants, see information about "Memory Management".

Note: The ASF Wizard can be reopened to change the memory manager variant at a later point of time.

Figure 1-4. Selecting a FreeRTOS Memory Manager Variant



The ASF Wizard has staged all the necessary changes to add FreeRTOS with the selected memory manager to the project. Click the **Apply** button available at the bottom of the screen, to apply the changes.

2. Setting Up Clock and Tick Rate

The FreeRTOS configuration file `FreeRTOSConfig.h` is located in `src/config/` in the Atmel Studio project tree. This file contains defines for system parameters like the CPU frequency and OS tick rate, and for enabling or configuring specific features and functions such as mutexes, co-routines, and memory allocation failure handling.

When starting a FreeRTOS project, the first configuration parameters one should set/verify are the aforementioned CPU frequency and OS tick rate, which are used by FreeRTOS to time the task switches and delays. The corresponding defines are named `configCPU_CLOCK_HZ` and `configTICK_RATE_HZ`. These defines need not be numerical constants, but it is strongly recommended that they are defined as compile-time constants. For example, when using the clock driver and the system clock source is generic clock 0, the clock rate can be specified for FreeRTOS with:

```
#include <gclk.h>

#define configCPU_CLOCK_HZ  (system_gclk_gen_get_hz(GCLK_GENERATOR_0))
```

For more information on the available configuration parameters and defines, refer to the information about Customisation on the FreeRTOS website at http://www.freertos.org.

3. During Development

While developing applications using FreeRTOS, there are several tools and configuration parameters that can help to debug and optimize the project. For example, the following extensions for Atmel Studio are available in the Atmel Studio Gallery:

- Atmel FreeRTOS Viewer
- Percepio FreeRTOS+Trace

The upcoming sections introduce these extensions and describe useful debugging and optimization parameters.

3.1. Atmel FreeRTOS Viewer

Atmel FreeRTOS Viewer allows the user to see the state of FreeRTOS when you break execution of the target in a debug session. The user can see the current state of the tasks in the system, as well as queues, semaphores, and mutexes. To be able to see queues, semaphores, and mutexes the `configQUEUE_REGISTRY_SIZE` need to be configured and the queue, semaphore, or mutex needs to be added to the registry by calling `vQueueAddToRegistry`.

Refer to "FreeRTOS Viewer documentation" for more information about how to use the Atmel FreeRTOS Viewer.

3.2. Percepio FreeRTOS+Trace

Percepio's FreeRTOS+Trace allows the user to record and analyze the runtime behavior of FreeRTOS over time. By using a trace recorder library FreeRTOS+Trace is able to record FreeRTOS behavior without using a debugger. Percepio's FreeRTOS+Trace is also available as a standalone application. Refer to "FreeRTOS+Trace documentation" for more information.

3.3. Debugging and Optimization Parameters

Among the configuration parameters for debugging and optimization, the most important ones are:

- `configUSE_TICK_HOOK`: Enables calling a user-defined function whenever an OS tick occurs, allowing for inspection of the rate and consistency of the ticks. For details, see Hook Functions.
- `configUSE_IDLE_HOOK`: Enables calling a user-defined function whenever the idle task is executed, i.e. there are no other tasks to execute. For details, see Hook Functions and Tasks.
- `configCHECK_FOR_STACK_OVERFLOW`: Enables detection of tasks that use more than their allocated chunk of working memory (stack). Two detection methods are available, both of which require a user-defined handler function. For more details, see Stack Usage and Stack Overflow Checking.
- `configUSE_MALLOC_FAILED_HOOK`: Enables calling of a user-defined handler function if memory allocation fails. Memory is allocated on the heap whenever a semaphore, queue, or task is created, and may fail due to, e.g. insufficient size of or too fragmented heap, or simply using the wrong memory manager. For details, see Hook Functions and Customisation.
- `configGENERATE_RUN_TIME_STATS`: Enables tracking of time spent executing the individual tasks. The statistics can be dumped to a character buffer. For details, see Run Time Statistics.

- Trace hook macros: Allows user-defined functions to be called whenever an OS-related event such as creation or switching of tasks occurs in the application. This is the mechanism which Percepio's FreeRTOS+Trace uses. For details, see Trace Hook Macros.

4. Using Drivers in FreeRTOS

FreeRTOS uses preemptive scheduling, which means that execution of a task can be interrupted at any time for another task to start/continue its execution. The two main considerations for hardware or software resources,used in an environment with task interruptions are:

1. Concurrency issues.
2. Timing criticality.

In the following subsections, these two issues and methods to handle them are explained.

4.1. Concurrency

When a resource is shared between tasks that can interrupt each other, there is a chance that it is accessed or used by multiple tasks, simultaneously. This property is called *concurrency*. If the resource does not support it, an additional layer of access control must be put in place to avoid issues.

Note: If a function supports concurrency, it is said to be *reentrant*. This means that the function can be interrupted in the middle of its execution and safely be called again in the interrupting code.

Let's say we have a large buffer *X* that is shared between tasks *A*, which generates data, and *B*, which consumes data. If the system switches from task *A* to *B* before *A* has finished updating *X*, *B* will get corrupted or will be an incoherent data. In this case, *X* requires mutually exclusive access, meaning only one task can operate on it at the time.

A *mutex* is a type of *semaphore*, a signalling mechanism, which would solve this issue. Its sole purpose is to indicate whether a resource is in use or not , and any tasks which need to use the resource must then wait until they can take the mutex, i.e., the resource becomes available. The operating system avoids running the waiting tasks until the relevant mutex is released, or the tasks' wait times run out.

An alternative method is to define the relevant section of code as being *critical*. This will delay handling of all interrupts for the duration of the section, thus inhibiting task switches. However, this method can affect the timing of other tasks, which may have a higher priority than the currently executing one, and prevents operation of interrupt-based drivers. For these reasons, critical sections should only be used on short blocks of code, such as a read-modify-write operation on a status flag:

```
taskENTER_CRITICAL();

// Read
my_status = SOME_MODULE->status_flags;

if (!(my_status & BUSY_FLAG)) {
    // Modify (set flag)
    my_status |= BUSY_FLAG;
    // Write
    SOME_MODULE->status_flags = my_status;
}

taskEXIT_CRITICAL();
```

In the preceeding example, the critical section prevents other tasks from modifying `status_flags` before the local code has a chance to overwrite it with `my_status`. Without the critical section, if an interrupt occurs after the read and modifies `status_flags`, that modification will be lost once execution returns to the local code because it overwrites it with its `my_status`, based on the outdated `status_flags`.

Now consider the case where tasks *A* and *B* send data via USART using a polled driver. Although the USART peripheral and driver may look like they are available for use as soon as a byte transfer has finished, it does not mean that a task has finished sending its data. If task *A*, sending "Hello, world!", is interrupted by task *B*, sending "copter", the system may actually send "Hellocopter, world!".

This issue could be solved by use of either a mutex or a *queue*. For the latter, a queueing layer must be added to the USART driver. Task *A* and *B* could then enqueue their strings (or pointers to them) for sending, and the USART queue layer send them in order.

For more details on FreeRTOS queues, mutexes, and semaphores, see "Inter-task Communication". The macros for defining critical sections to FreeRTOS are documented under "Kernel Control".

4.2. Timing

If an operation on a resource is timing critical, even if a resource is not shared, task switching or other interruptions may cause problems. For example, some system-critical features require that a timed sequence is followed in order to change their configuration. This ensures that the configuration is not inadvertently changed by, e.g., misconfigured DMA or otherwise buggy code. In an application with interrupts and preemptive task scheduling, the only way to ensure that the timing is not disturbed is to define it as a critical section.

As stated earlier, the macros for defining critical sections to FreeRTOS are documented under "Kernel Control".

4.3. OS Compatibility of ASF Drivers

The drivers in ASF have not been designed to support a particular OS, but are reentrant. However, the driver instances do not support concurrent access. It is up to the user to determine the need and to implement an access control or queuing layer if an instance is shared between tasks.

If the application or a driver uses the SysTick peripheral after the FreeRTOS task scheduler has been started, it will interfere with the task scheduling[1]. The timing mechanisms of the application or driver must be replaced with a different implementation. The driver in ASF which uses SysTick is the "Delay routines" one, but other drivers may depend on it.

Note: FreeRTOS uses the SysTick peripheral for timing.

5. Description of Demo Application

5.1. Overview

A graphical FreeRTOS demo application is available for related SAM Xplained Pro and OLED1 Xplained Pro. The application demonstrates basic use of queues and mutexes/semaphores, and creation, suspension, and resumption of tasks. It requires the OLED1 Xplained Pro to be connected to the EXT3 header on the SAM D20/D21/L21/L22 Xplained Pro or the EXT1 header on the SAM R21 Xplained Pro. The demo application is provided with both FreeRTOS versions supported: v7.4.2 and v7.5.2.

The application shows a menu on the bottom of the OLED, giving the user the choice between three different screens:

- **Graph:** Shows pseudo-random graph which is continuously updated, even while it is not shown.
- **Terminal:** Prints text which has been received via the Embedded Debugger (EDBG) Virtual COM Port.
- **About:** Prints a short text about the application, with a simple zooming effect.

To select a screen, press the corresponding button on the OLED1 Xplained Pro.

On the upper part of the OLED, the content of the selected screen is displayed. By default, the graph screen is selected during startup. An example of the display state shortly after startup is displayed in the following image.

Figure 5-1. The Default Startup Screen of FreeRTOS Demo Application

5.2. Running the Demos

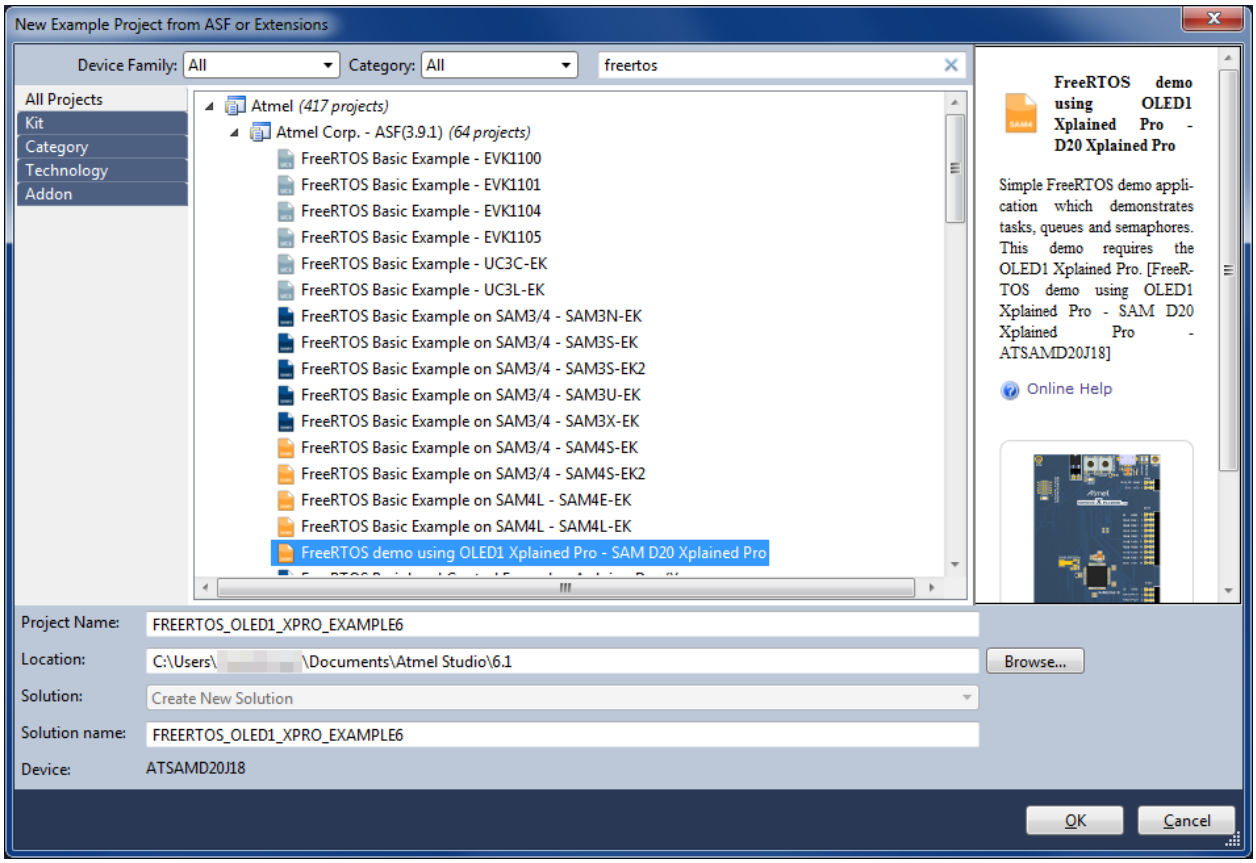
A demo application named "FreeRTOS demo using OLED1 Xplained Pro" is available in the "New Example Project" wizard, for the Atmel Corp. extension "ASF(3.9.1)" or newer.

An additional demo application named "FreeRTOS tickless demo with using OLED1 Xplained Pro", represents tickless feature of FreeRTOS v7.5.2, is provided with "ASF(3.14.0)" or newer.

Both these demos share the same features except the tickless feature of the FreeRTOS v7.5.2 kernel.

Considering the demo of FreeRTOS v7.4.2 for SAM D20 as an example, to open the wizard, click **File > New > Example Project** in the menu bar or press Ctrl + Shift + E. As shown in the following figure, enter "freertos" as a filter term to avoid having to scroll through the entire list of projects.

Figure 5-2. The Demo Project Selected in the Example Project Wizard



After the project has been selected in the wizard, click **OK** to create it. The project creation process will show the applicable licenses for the project, which must all be accepted for the process to complete.

To run the demo on the board, ensure that the Xplained Pro is connected to a USB port on the computer. Then, click **Debug>Start Without Debugging** in the top menu bar in Atmel Studio. This triggers the build process, after which the **Select Tool** dialog will pop up. Select the **XPRO-EDBG** tool and click **OK**. The demo is then programmed into the device and run.

To transmit text to the terminal screen, connect to the EDBG Virtual COM Port with a terminal emulator.

For the communication to be successful, the configuration must be set as:

- 9600 baud
- no handshake
- one stop-bit
- parity disabled

To ensure that the communication works, the application echoes back all characters that have been received without errors. It is not necessary to select the terminal screen for the application to handle the received characters.

5.3. Application Structure, Control, and Data Flow

The demo application consists of five tasks, all of which are defined and configured in `demotasks.c`:

1. **Main task:** Handles button presses, switches/clears the display buffer and draws the menu if needed, and suspends/resumes on-screen tasks.

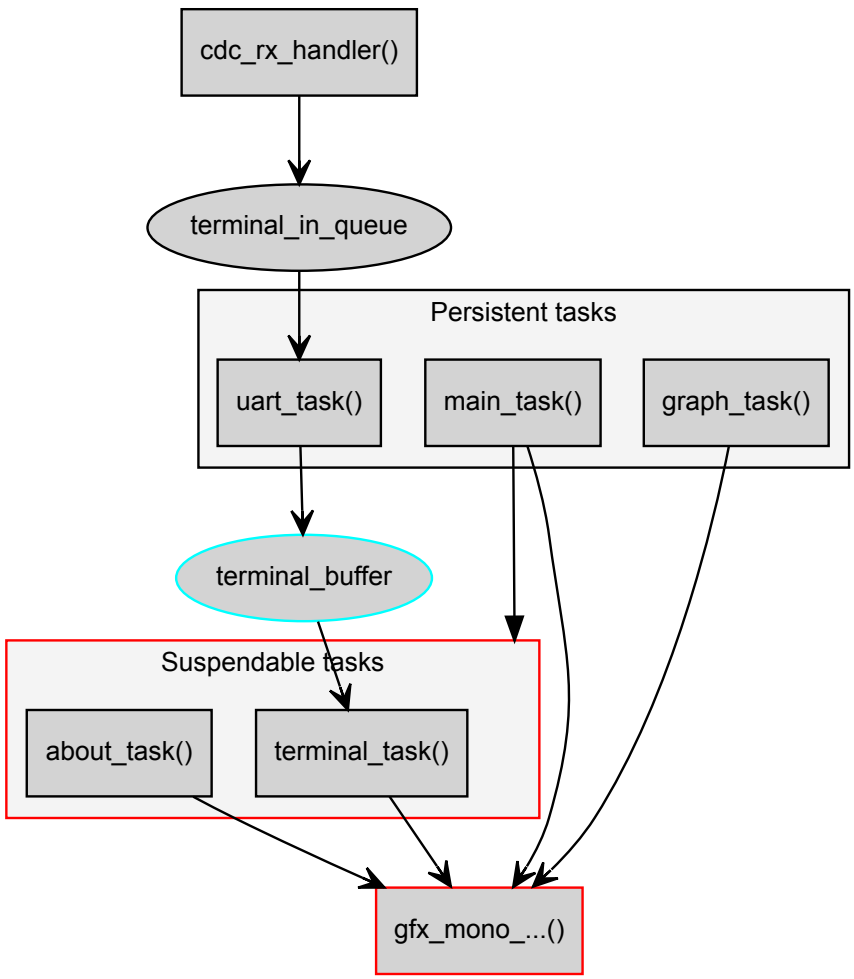
2. **Graph task:** Draws a pseudo-random graph to a dedicated display buffer, one pixel every 50 milliseconds.
3. **Terminal task:** Prints text received from UART (up to three lines with 21 characters each) to the display every second.
4. **About task:** Prints a short text about the demo in several iterations to achieve a zooming effect.
5. **UART task:** Reads from a FreeRTOS queue that contains incoming characters and updates the terminal buffer accordingly every 10 milliseconds.

The main, graph, and UART tasks are persistent, they are never suspended. Depending on the screen selected by the user, the main task will suspend and resume the terminal and about tasks. Since the graph screen has a dedicated display buffer, the main task only needs to switch to its display buffer when it is selected.

In addition, there is a custom interrupt handler for the UART communication via the EDBG Virtual COM Port. The interrupt handler is based on the ASF's SERCOM USART callback driver's handler. It has been modified to handle only the receive interrupt, and to put the received characters into a FreeRTOS queue (for handling by the UART task) before echoing them back.

The following figure is a visualization of the five tasks and their flow of data and control.

Figure 5-3. Application Structure, Control, and Data Flow



Mutexes are used to avoid problems with concurrent access to the display driver or terminal buffer by different tasks. These are indicated with the cyan and red coloured edges in the preceding figure.

The terminal buffer mutex is used to prevent the terminal screen becomes corrupted if the EDBG Virtual COM Port receives characters while the screen is being printed. In this case, the UART task must wait until the printing is done before it can process the queue of incoming characters. Vice versa, the terminal screen cannot be printed while the UART task is processing incoming characters.

The display mutex is used to prevent tasks from interfering with each other's graphical output. Also, it signals when it is safe for the main task to switch display buffers or suspend the currently displayed task because it is released at the end of the task loops, when they have updated the display.

Note: The graphics stack, GFX MONO, relies on the "Delay routines" driver, which uses SysTick for timing. However, this is only used for a hard reset in a low level driver. The hard reset occurs only once during initialization of the stack, which is done before the FreeRTOS task scheduler is started. The graphics driver does not interfere with the task scheduler.

6. Revision History

Doc. Rev.	Date	Comments
42138D	03/2016	Added support for SAM L22
42138C	10/2015	Added support for SAM R21 and L21
42138B	01/2014	Added support for FreeRTOS v7.5.2 and SAM D21
42138A	06/2013	Initial release

Atmel | Enabling Unlimited Possibilities®

Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA **T:** (+1)(408) 441.0311 **F:** (+1)(408) 436.4200 | **www.atmel.com**