SMART ARM-based Microcontrollers

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

APPLICATION NOTE

Features

This application note describes how to get started using FreeRTOS on Atmel® | SMART ARM®-based microcontrollers by showing how to get started in Atmel Studio, how to configure FreeRTOS, how to use the Atmel provided ASF drivers with FreeRTOS, and describing the provided demo application.

The following devices can use this module:

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

Table of Contents

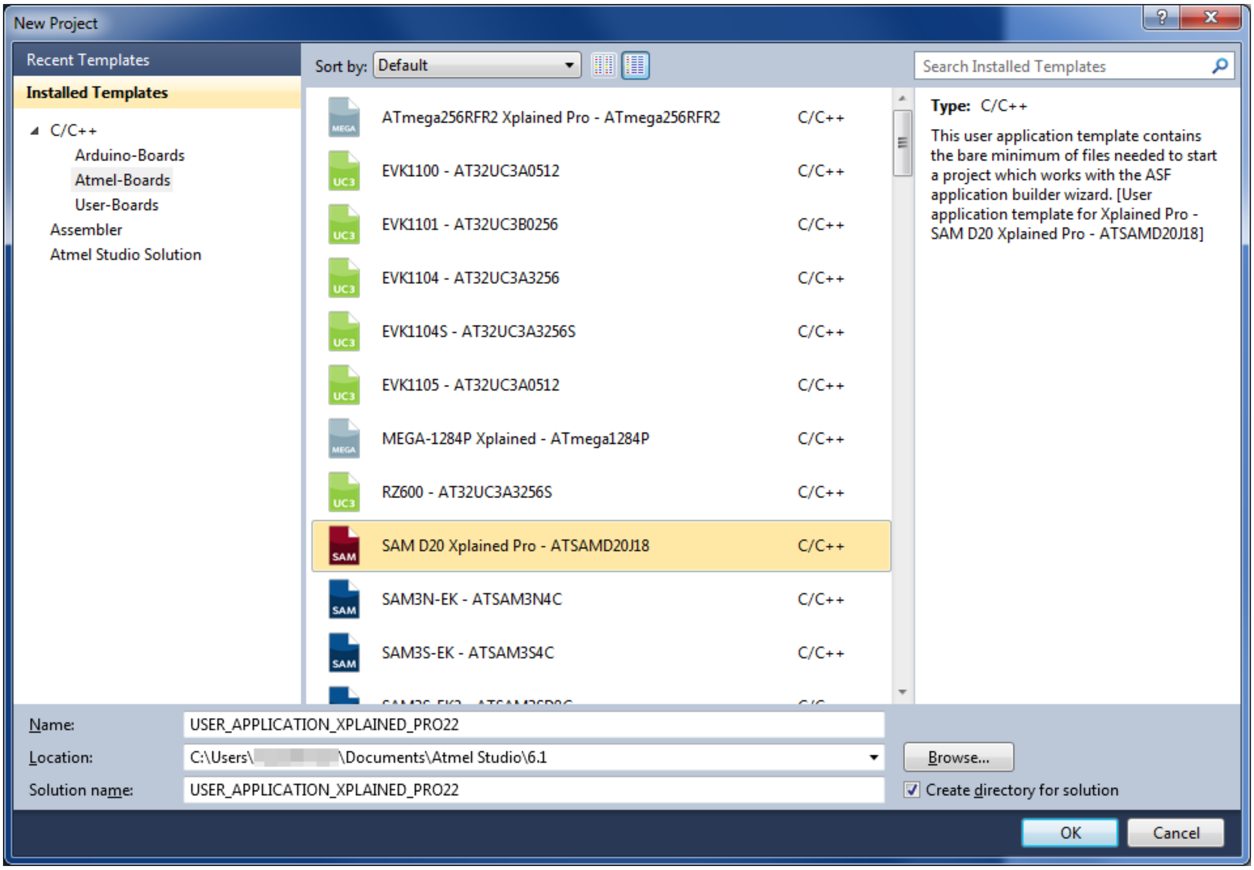
1. Introduction

Operating systems appear to allow multiple concurrent tasks to execute at the same time. In reality the operating system executes each task for a while and then rapidly switches between them without the user noticing. This is referred to as multitasking. FreeRTOS is a light-weight Real Time Operating System (RTOS) which allows multitasking on microcontrollers like SAM MCUs. Readers who are not familiar with the RTOS concept and would like to know more can take a look at "What is an RTOS?" and "Why use an RTOS?" at the FreeRTOS website: http://www.freertos.org

2. How to Start a FreeRTOS Project in Atmel Studio

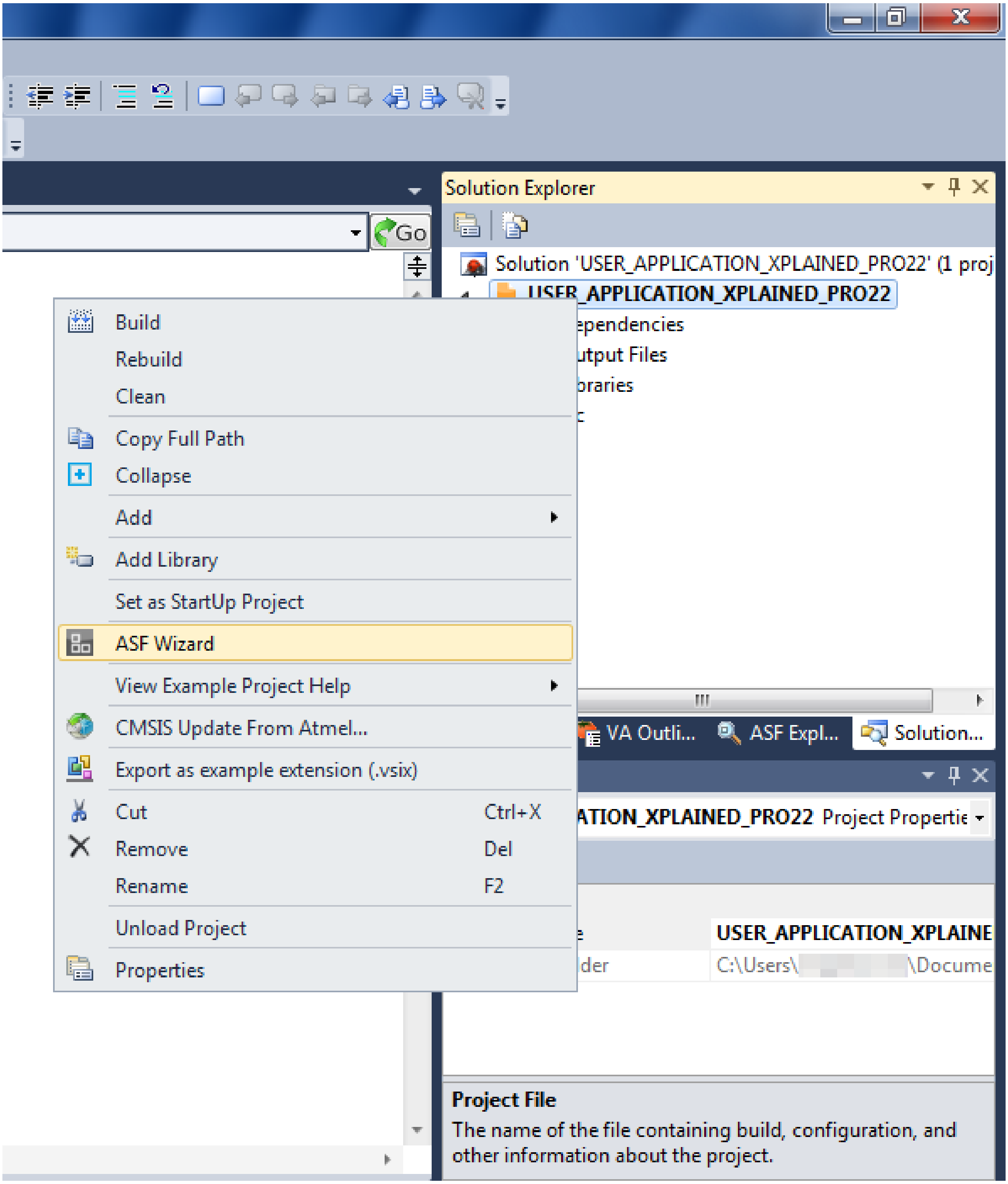
To create a FreeRTOS project in Atmel Studio, start with either a user board or Atmel board template project for a SAM MCU device, such as the SAM D20 Xplained Pro. Figure 2-1 Creating a Project From the SAM D20 Xplained Pro Template on page 4 shows the "New Project" dialog in this case. To open this dialog, click "File" - "New" - "Project" in the menu bar or press Ctrl + Shift + N.

Figure 2-1 Creating a Project From the SAM D20 Xplained Pro Template



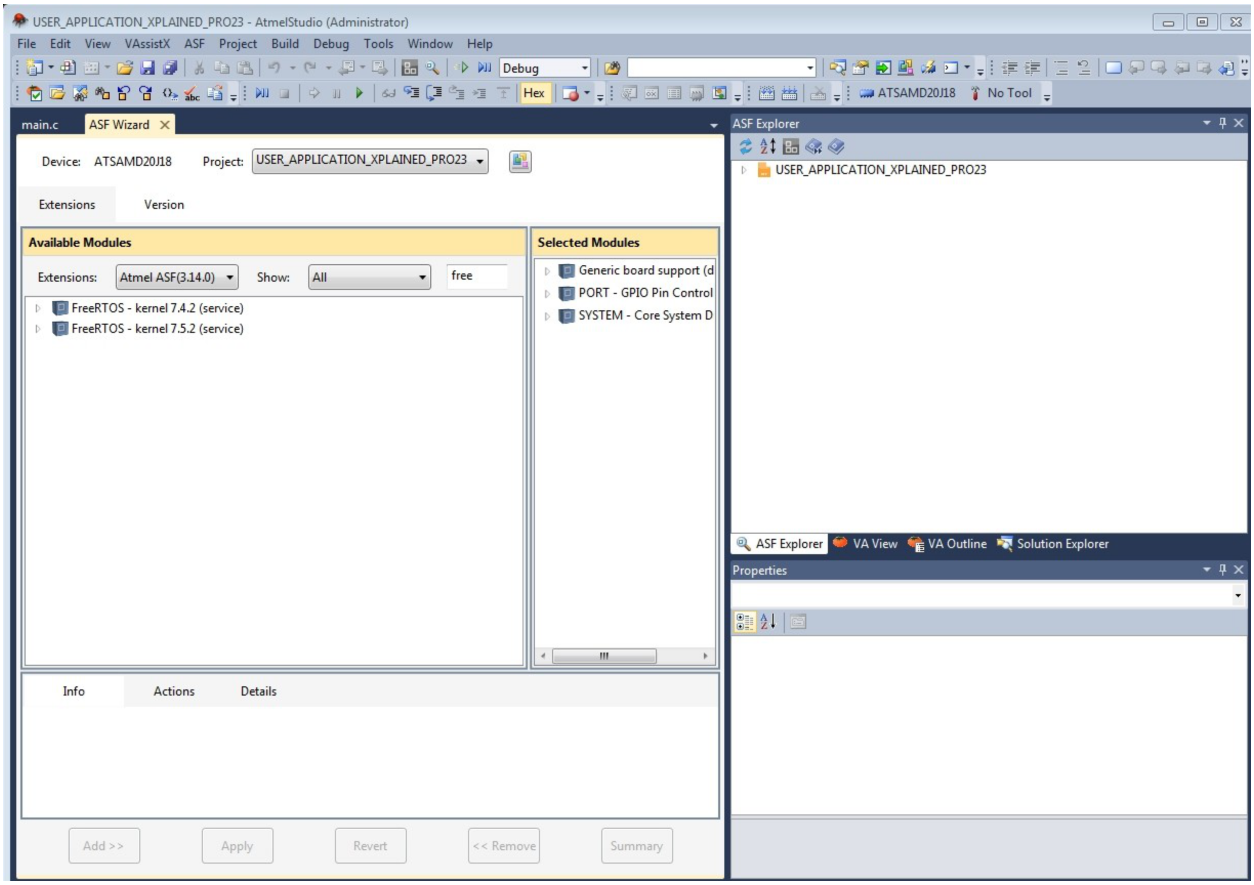
After creating the new project, the FreeRTOS must be added to the project. Start the ASF Wizard by right-clicking the project and selecting "ASF Wizard" in the context menu, as shown in Figure 2-2 Starting the ASF Wizard on page 5.

Figure 2-2 Starting the ASF Wizard



FreeRTOS must then be selected in the list of available modules, on the left side of 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 Figure 2-3 Adding FreeRTOS to the Project on page 6.

Figure 2-3 Adding FreeRTOS to the Project

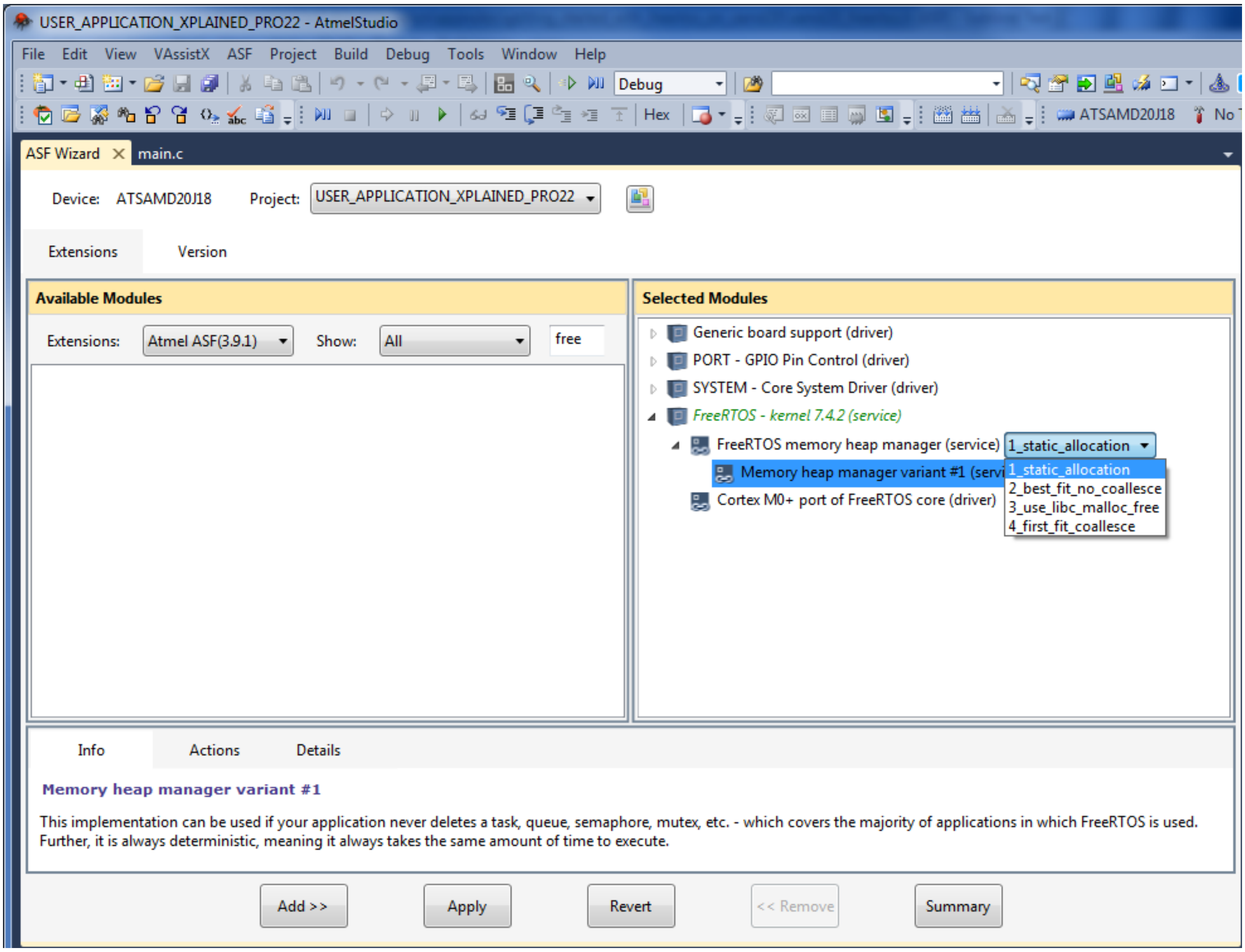


FreeRTOS v7.4.2 is provided for SAM MCUs and an additional version of v7.5.2 is provided from ASF v3.14.0 or newer. A default tickless feature is implemented in v7.5.2, which allows longer sleep periods by shutting down the OS tick when it is not needed, can be toggled on or off to compare power consumption.

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

At this point, you may need 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 Figure 2-4 Selecting a FreeRTOS Memory Manager Variant on page 7. Short descriptions of the variants can be read in the info box at the bottom. For further details on the different variants, see "Memory Management". Note that the ASF Wizard can be reopened if you wish to change the memory manager variant at a later point in time.

Figure 2-4 Selecting a FreeRTOS Memory Manager Variant



The ASF Wizard has now staged all the necessary changes to add FreeRTOS with the selected memory manager to the project. Simply click the "Apply" button at the bottom to make the wizard apply them.

3. How to Set Up Clock and Tick Rate

The configuration file for FreeRTOS is named `FreeRTOSConfig.h` and is located under `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`. They do not need to be defined as numerical constants, but are strongly recommended to be defined as compile-time constants. For example, if 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 extensive documentation for the available configuration parameters and defines, refer to the "Customisation" chapter on the FreeRTOS website at http://www.freertos.org.

4. Nice to Know During Development

When developing applications with FreeRTOS, there are several tools and configuration parameters which can be helpful in debugging and optimization of the project. For example, the following extensions for Atmel Studio are available in the Atmel Studio Gallery:

- Atmel FreeRTOS Viewer
- Percepio FreeRTOS+Trace

The next chapters introduce these extensions as well as describe useful debugging and optimization parameters.

4.1. 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.

4.2. 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.

4.3. Debugging and Optimization Parameters

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

- `configUSE_TICK_HOOK`: Enables calling of 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 of 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 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 stats 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 occurs in the application, such as creation or switching of tasks. This is the mechanism which Percepio's FreeRTOS+Trace uses. For details, see "Trace Hook Macros".

5. 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. With regards to resources, be it in hardware or software, there are mainly two things one must think of in an environment with task interruptions:

1. Concurrency issues.
2. Timing criticality.

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

5.1. Concurrency

When a resource is shared between tasks that can interrupt each other, there is a chance that it is accessed or used by several tasks at the same time. This property is called *concurrency*, and 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 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 or not a resource is in use, 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 will prevent operation of interrupt-based drivers. For these reasons, critical sections should only be used on short blocks of code, for example 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 example above, the critical section will prevent 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 would need to 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".

5.2. Timing

Even if a resource is not shared, task switching or other interruptions may cause problems if an operation on it is timing critical. For example, some system-critical features require that a timed sequence is followed in order to change their configuration. This is to ensure 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".

5.3. 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. This means that it is up to the user to determine the need for and to implement an access control or queueing layer if an instance is shared between tasks.

It is also important to note that FreeRTOS uses the SysTick peripheral for timing. 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. The timing mechanisms of the application or driver must then 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.

6. Description of Demo Application

6.1. 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 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, the corresponding button on the OLED1 Xplained Pro must be pressed.

On the upper part of the OLED, the content of the selected screen is shown. By default, the graph screen is selected upon startup. An example of the display state shortly after startup is shown in Figure 6-1 The Default Startup Screen of FreeRTOS Demo Application on page 13.

Figure 6-1 The Default Startup Screen of FreeRTOS Demo Application

6.2. Running the Demos

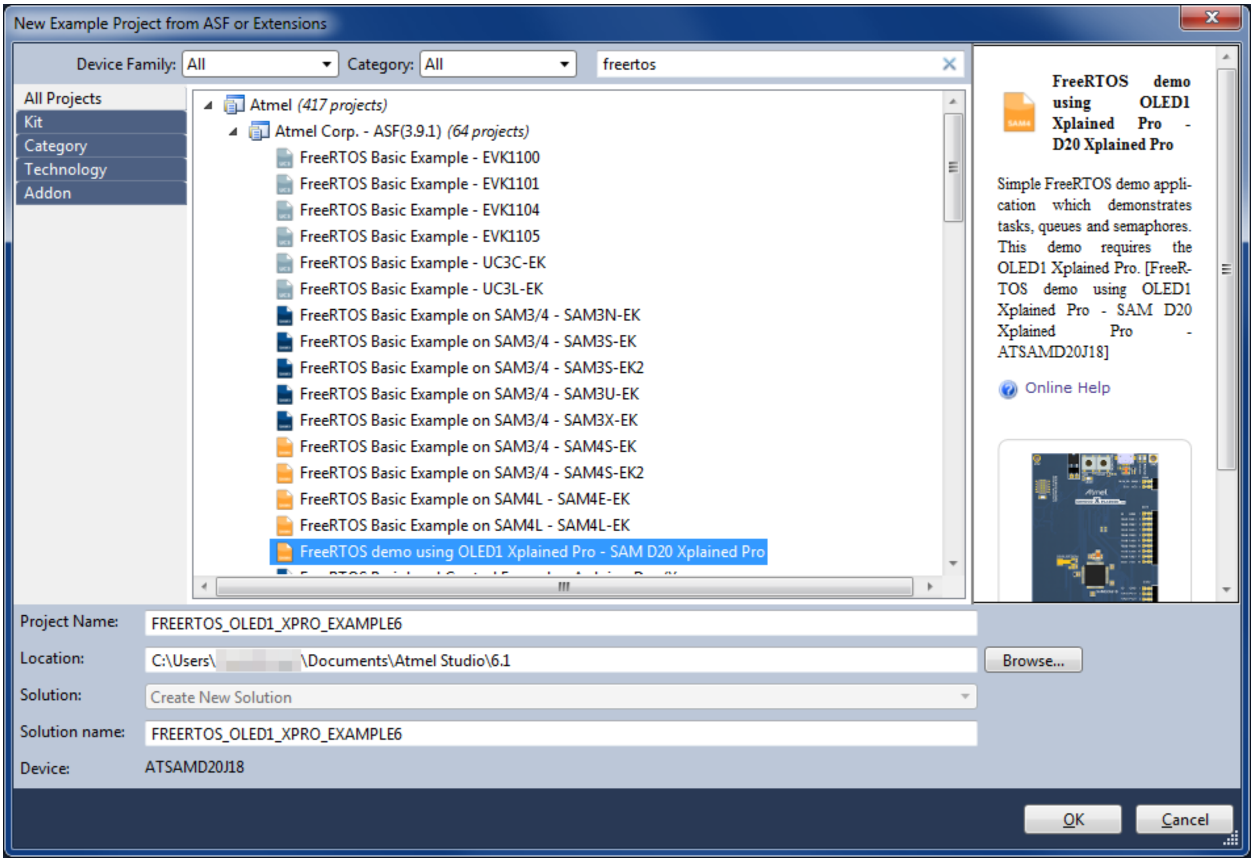
A demo application named "FreeRTOS demo using OLED1 Xplained Pro" can be found in the "New Example Project" wizard, for the Atmel Corp. extension "ASF(3.9.1)" or newer.

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.

These two demos share the same features except the tickless feature of the FreeRTOS v7.5.2 kernel.

Taking 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 Figure 6-2 The Demo Project Selected in the Example Project Wizard on page 14, you can enter "freertos" as a filter term to avoid having to scroll through the entire list of projects.

Figure 6-2 The Demo Project Selected in the Example Project Wizard



Once 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, first 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 will trigger 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 send text to the terminal screen, connect to the EDBG Virtual COM Port with a terminal emulator. For the communication to work, the configuration must be set to:

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

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

6.3. Application Structure, Control, and Data Flow

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

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

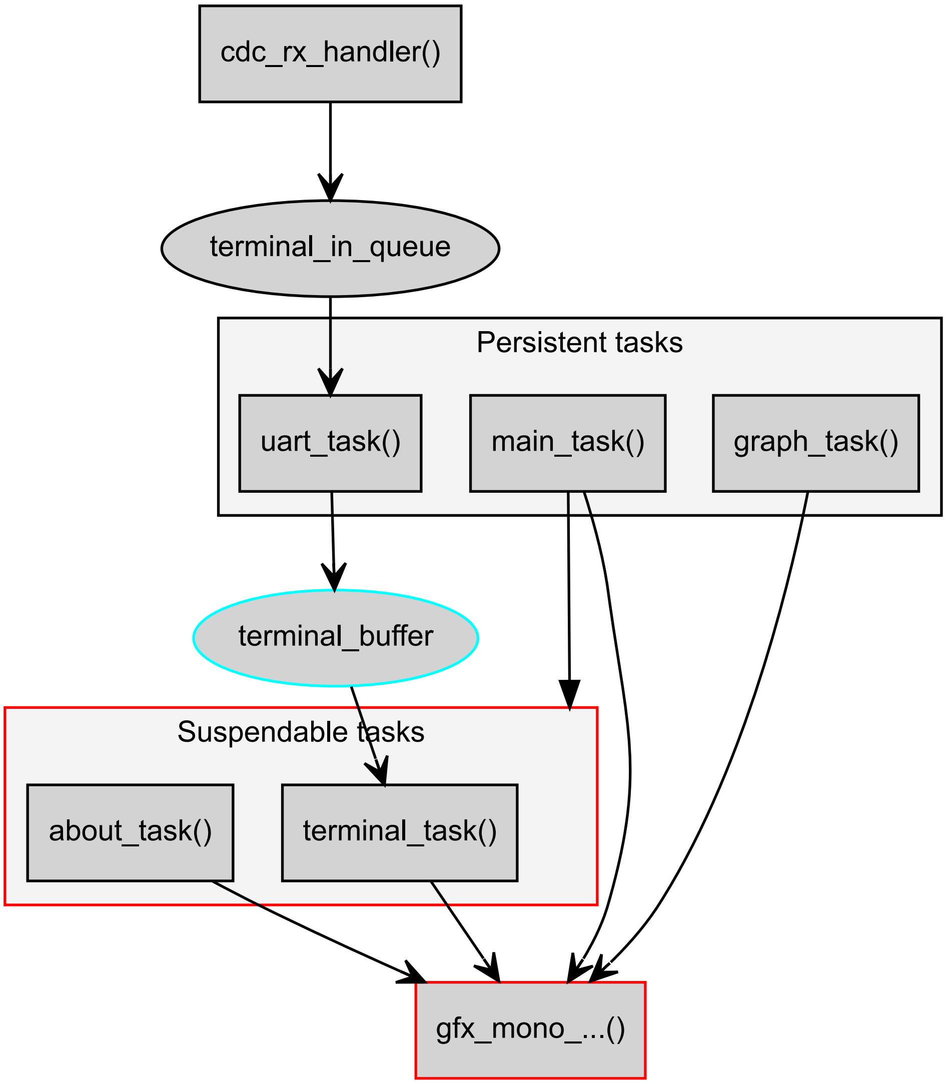
- Graph task: Draws a pseudo-random graph to a dedicated display buffer, one pixel every 50 milliseconds.
- Terminal task: Prints text received from UART (up to 3 lines with 21 characters each) to the display every second.
- About task: Prints a short text about the demo in several iterations to achieve a zooming effect.
- 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, meaning they are never suspended. Depending on which screen is 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.

Figure 6-3 Application Structure, Control, and Data Flow on page 16 is a visualization of the five tasks and their flow of data and control.

Figure 6-3 Application Structure, Control, and Data Flow



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

The terminal buffer mutex is used to prevent that 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 has to 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 are all done updating 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.

7. Revision History

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