Introduction

The Atmel® Software Framework (ASF) is a set of source files, libraries, and reference projects that enable rapid development of application code that runs on, and interfaces with, a range of Atmel microcontrollers.

The ASF provides the build configuration and peripheral driver framework necessary to host an application, allowing the application’s writer to focus development time solely on creating core functionality.

The ASF is delivered with FreeRTOS. FreeRTOS is a real time operating system (RTOS) that is designed specifically for use on small and medium sized microcontrollers. FreeRTOS is free to use, even in commercial applications, and without any obligation to expose your proprietary source code. http://www.FreeRTOS.org/license provides license information.

The ASF also includes ASF specific FreeRTOS functionality that integrates ASF driver libraries with FreeRTOS services to provide a set of highly efficient peripheral interface functions.

The FreeRTOS ASF functions use the peripheral DMA controller (PDC) to send and receive data with practically no processing overhead, and use the FreeRTOS multi-tasking services to ensure the microcontroller can continue executing application code at all times, even when one thread of execution has to wait for a send or receive to complete.

The scope of this application note is limited to describing how to use the ASF specific FreeRTOS functionality. As a minimum, it is intended that readers understand the term ‘task’ to mean a thread of execution in a multi-threaded (multi-tasking) application. http://www.freertos.org/about-RTOS.html provides additional information for readers who are not already familiar with the concept of an RTOS, and wish to learn more. http://www.FreeRTOS.org/why-RTOS.html provides additional information for readers who want to know when to use an RTOS, and the benefits of using an RTOS appropriately. http://www.freertos.org/FreeRTOS-quick-start-guide.html provides a quick start guide to FreeRTOS, and includes a link to the FreeRTOS API documentation.

Additional resources include example projects that demonstrate the use of the FreeRTOS ASF peripheral control drivers. The example projects are contained in the ASF distribution, and are a good resource for users who want to make an immediate start.
Table of Contents

1. API overview ................................................................. 3
   1.1 Initializing ................................................................. 3
   1.1.1 Initialization functions ......................................... 3
   1.2 freertos_peripheral_options_t .................................... 3
   1.2.2 Examples .......................................................... 5
   1.3 Transmitting ............................................................ 8
       1.3.1 Transmit functions ............................................. 8
       1.3.2 Standard transmit functions ............................. 9
       1.3.3 Fully asynchronous transmit functions .............. 11
   1.4 Receiving ............................................................... 13
   1.5 Receive functions for SPI and TWI ......................... 14
       1.5.1 Standard receive functions for SPI and TWI ....... 14
       1.5.2 Fully asynchronous receive functions for SPI and TWI peripherals .. 16
   1.6 Receive functions for USART ................................. 19
2. Revision History ........................................................... 21
1. API overview

FreeRTOS ASF functions are provided to manage the efficient transmission and reception of serial data. At the time of writing, the TWI, I2C and USART peripherals are supported.

A uniquely named function is provided for each supported peripheral to initialize the FreeRTOS driver for that peripheral, write bytes to that peripheral, and read bytes from that peripheral.

The source files that implement the FreeRTOS ASF functions for SAM3 and SAM4 parts are contained in the ASF/common/services/freertos/sam directory of the ASF distribution. The tasks that demonstrate how to use the FreeRTOS ASF functions are contained in the ASF/thirdparty/freertos/demo/peripheral_control directory, and sub-directories.

1.1 Initializing

1.1.1 Initialization functions

Initialization functions are shown in Listing 1.

Listing 1. FreeRTOS ASF driver initialization functions.

```c
freertos_spi_if
freertos_spi_master_init( Spi *p_spi,
    const freertos_peripheral_options_t * const freertos_driver_parameters )

freertos_twi_if
freertos_twi_master_init( Twi *p_twi,
    const freertos_peripheral_options_t * const freertos_driver_parameters )

freertos_usart_if
freertos_usart_serial_init( Usart *p_usart,
    const sam_usart_opt_t * const uart_parameters,
    const freertos_peripheral_options_t * const freertos_driver_parameters )
```

Parameters, and return values, are fully described in the ASF API documentation, and in source file comments directly above the respective function prototypes.

In all cases, the first parameter is a pointer to the peripheral, and uses the standard ASF defined type.

sam_uart_options_t is also a standard ASF type. sam_uart_options_t is used to define the USART operation parameters, such as baud rate and parity.

freertos_peripheral_options_t is a FreeRTOS ASF specific type that is described in the next sub-section.

Source code that demonstrates the use of each initialization function is provided at the end of this section.

1.2 freertos_peripheral_options_t

freertos_peripheral_options_t is the structure shown in Listing 2. The structure members are described in Table 1.
Listing 2. The freertos_peripheral_options_t data type.

```c
typedef struct freertos_peripheral_options {
    uint8_t *receive_buffer;
    uint32_t receive_buffer_size;
    uint32_t interrupt_priority;
    enum peripheral_operation_mode operation_mode;
    uint8_t options_flags;
} freertos_peripheral_options_t;
```

Table 1. freertos_peripheral_options_t structure members.

<table>
<thead>
<tr>
<th>Structure Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive_buffer</td>
<td>A pointer to a buffer into which the PDC will write received data. This parameter is (at the time of writing) only required by the USART driver, and is ignored by drivers provided for other peripherals.</td>
</tr>
<tr>
<td>receive_buffer_size</td>
<td>The size, in bytes, of the buffer pointed to by the receive_buffer structure member. This parameter is (at the time of writing) only required by the USART driver, and is ignored by drivers provided for other peripherals.</td>
</tr>
<tr>
<td>interrupt_priority</td>
<td>Sets the priority of the interrupt generated by the PDC. PDC interrupts are configured and processed by the FreeRTOS ASF drivers. The application writer does not need to concern themselves with installing or handling interrupts. It is essential that the interrupt priority is not set above configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY. configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY is defined in FreeRTOSConfig.h. Invalid priority values are trapped by an assert() in the driver code if configASSERT() is defined in FreeRTOSConfig.h. <a href="http://www.freertos.org/RTOS-Cortex%E2%84%A2-M3-M4.html">http://www.freertos.org/RTOS-Cortex™-M3-M4.html</a> provides additional information for readers who are not familiar with interrupt priority assignment on Cortex-M3 and Cortex-M4 microcontrollers, or with FreeRTOS configuration.</td>
</tr>
<tr>
<td>operation_mode</td>
<td>operation_mode is included to allow future versions of the FreeRTOS ASF drivers to support multiple modes of operation on a single peripheral. Valid values are documented in Table 2. At the time of writing, there is only one valid value for each supported peripheral.</td>
</tr>
<tr>
<td>options_flags</td>
<td>Individual bits in the options_flags value configure an aspect of the driver’s behavior. Bit definitions are documented in Table 3. Bit definitions can be ORed together.</td>
</tr>
</tbody>
</table>

Table 2. Valid values for the operation_mode parameter of the freertos_peripheral_options_t structure. At the time of writing, only a single value is supported for each peripheral.

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>operation_mode value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>USART</td>
<td>USART_RS232</td>
<td>Configures the USART peripheral as an RS232 port.</td>
</tr>
<tr>
<td>SPI</td>
<td>SPI_MASTER</td>
<td>Configures the SPI peripheral as an SPI master.</td>
</tr>
<tr>
<td>TWI</td>
<td>TWI_I2C_MASTER</td>
<td>Configures the TWI peripheral as an I2C master.</td>
</tr>
</tbody>
</table>
Table 3. Valid bit value definitions for the options_flag member of the freertos_peripheral_options_t structure.

<table>
<thead>
<tr>
<th>Bit Definition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_TX_ACCESS_MUTEX</td>
<td>This bit makes the transmit functionality of the peripheral driver thread aware. Set this bit if the application code writes to the peripheral from more than one RTOS task.</td>
</tr>
<tr>
<td>USE_RX_ACCESS_MUTEX</td>
<td>This bit makes the receive functionality of the peripheral driver thread aware. Set this bit if the application code reads from the peripheral from more than one RTOS task.</td>
</tr>
<tr>
<td>WAIT_TX_COMPLETE</td>
<td>The FreeRTOS ASF drivers provide both standard (blocking) and fully asynchronous transmit functions. The transmit functions are described later in this application note. The WAIT_TX_COMPLETE bit must be set to use the standard transmit function, and must be clear to use the fully asynchronous transmit function.</td>
</tr>
<tr>
<td></td>
<td>If this bit is set, then an RTOS task that starts a transmission will wait for the transmission to complete before executing any further code. Other RTOS tasks will execute while the transmission is in progress, ensuring no processing time is wasted.</td>
</tr>
<tr>
<td></td>
<td>Refer to the section in this application note that describes the transmit functions for more information.</td>
</tr>
<tr>
<td>WAIT_RX_COMPLETE</td>
<td>The FreeRTOS ASF drivers provide both standard (blocking) and fully asynchronous receive functions. The receive functions are described later in this application note. The WAIT_RX_COMPLETE bit must be set to use the standard receive function, and must be clear to use the fully asynchronous receive function.</td>
</tr>
<tr>
<td></td>
<td>If this bit is set, then an RTOS task that starts a receive operation will wait for the receive operation to complete (or time out) before exiting the receive function. Other RTOS tasks will execute while the reception is in progress, ensuring no processing time is wasted.</td>
</tr>
<tr>
<td></td>
<td>Refer to the application note section that describes the receive functions for more information.</td>
</tr>
</tbody>
</table>

1.2.2 Examples

- Listing 3 is a code snippet that demonstrates how to use freertos_spi_master_init().
- Listing 4 is a code snippet that demonstrates how to use freertos_twi_master_init().
- Listing 5 is a code snippet that demonstrates how to use freertos_usart_serial_init().
Listing 3. Example use of the freertos_spi_master_init() function.

freertos_spi_if prepare_spi_port(Spi *spi_base) {

    /* Handle used to access the initialized port by other FreeRTOS ASF functions. */
    freertos_spi_if freertos_spi;

    /* Configuration structure. */
    const freertos_peripheral_options_t driver_options = {
        /* This peripheral does not need a receive buffer, so the receive_buffer value is just set to NULL. */
        NULL,
        /* There is no receive buffer, so receive_buffer_size is not used and can take any value. */
        0,
        /* The interrupt_priority value. */
        0x0f,
        /* The operation_mode value. */
        SPI_MASTER,
        /* All the available options_flags bits are set. */
        (USE_TX_ACCESS_MUTEX | USE_RX_ACCESS_MUTEX | WAIT_TX_COMPLETE | WAIT_RX_COMPLETE)
    }

    /* Call the SPI specific FreeRTOS ASF driver initialization function. */
    freertos_spi = freertos_spi_master_init(spi_base, &driver_options);

    if (freertos_spi != NULL) {
        /* Calling freertos_spi_master_init() will enable the peripheral clock, and set the SPI into master mode. Other ASF configuration functions, such as spi_set_clock_polarity(), and spi_set_baudrate_div() can then be called here. */
    }

    return freertos_spi;
}

Listing 4. Example use of the freertos_twi_master_init() function.

freertos_twi_if prepare_twi_port( Twi *twi_base ) {

    /* Handle used to access the initialized port by other FreeRTOS ASF functions. */
    freertos_twi_if freertos_twi;

    /* Configuration structure. */
    freertos_peripheral_options_t driver_options = {

        /* This peripheral does not need a receive buffer, so the receive_buffer
         * value is just set to NULL. */
        NULL,

        /* There is no receive buffer, so receive_buffer_size is not used and can
         * take any value. */
        0,

        /* The interrupt_priority value. */
        0x0f,

        /* The operation_mode value. */
        TWI_I2C_MASTER,

        /* All the available options flags are used for demonstration purposes. */
        ( USE_TX_ACCESS_MUTEX | USE_RX_ACCESS_MUTEX | WAIT_TX_COMPLETE | WAIT_RX_COMPLETE )
    };

    /* Call the TWI specific FreeRTOS ASF driver initialization function. */
    freertos_twi = freertos_twi_master_init( twi_base, &driver_options );

    if (freertos_twi != NULL) {
        /* Calling freertos_twi_master_init() will enable the peripheral clock,
         * and set the TWI into I²C master mode. Other ASF configuration functions,
         * such as twi_set_speed(), can then be called here. */
    }

    return freertos_twi;
}
Listing 5. Example use of the freertos_usart_serial_init() function.

```c
freertos_usart_if prepare_usart_port( Usart *usart_base,
    uint8_t *receive_buffer,
    uint32_t receive_buffer_size_in_bytes ) {

    /* Handle used to access the initialized port by other FreeRTOS ASF functions. */
    freertos_usart_if freertos_usart;

    /* Configuration structure. */
    freertos_peripheral_options_t driver_options = {

        /* This peripheral has full duplex asynchronous operation, so the
         * receive_buffer value is set to a valid buffer location. */
        receive_buffer,

        /* receive_buffer_size is set to the size, in bytes, of the buffer pointed
         * to by the receive_buffer structure member (receive_buffer above). */
        receive_buffer_size_in_bytes,

        /* The interrupt_priority value. */
        0x0e,

        /* The operation_mode value. */
        USART_RS232,

        /* All the available options flags are used for demonstration purposes. */
        ( USE_TX_ACCESS_MUTEX | USE_RX_ACCESS_MUTEX | WAIT_TX_COMPLETE | WAIT_RX_COMPLETE )
    };

    /* The RS232 configuration. This structure, and the values used in its setting,
    are from the standard ASF USART driver. */
    const sam_usart_opt_t usart_settings = {
        USART_BAUD_RATE,
        US_MR_CHRL_8_BIT,
        US_MR_PAR_NO,
        US_MR_NBSTOP_1_BIT,
        US_MR_CHMODE_NORMAL,
        0 /* Only used in IrDA mode, so all values are ignored. */
    };

    /* Call the USART specific FreeRTOS ASF driver initialization function. */
    freertos_usart = freertos_usart_serial_init( usart_base, &usart_settings, &driver_options );

    return freertos_usart;
```

1.3 Transmitting

1.3.1 Transmit functions

FreeRTOS ASF functions use the PDC to transfer data from a buffer to a peripheral for transmission. It should be noted, due to hardware restrictions, the buffer must be located in RAM. The microcontroller continues executing application code while the data is being transmitted.

Data can be transmitted using either a standard (blocking) function call, or a fully asynchronous function call. The next two sub-sections describe the options.
Note: In an RTOS application, blocking does not mean polling or spinning, or in fact, using any processing time at all. If the executing task (thread of execution) enters the Blocked state, the RTOS will ensure no processing time is wasted by immediately starting to execute a different task.

1.3.2 **Standard transmit functions**
Standard transmit functions are shown in Listing 6.

**Listing 6. FreeRTOS ASF standard transmit driver functions.**

```c
status_code_t freertos_spi_write_packet( freertos_spi_if *p_spi, const uint8_t *data, size_t len, portTickType block_time_ticks );

status_code_t freertos_twi_write_packet( freertos_twi_if *p_twi, twi_packet_t *p_packet, portTickType block_time_ticks );

status_code_t freertos_usart_write_packet( freertos_usart_if *p_usart, const uint8_t *data, size_t len, portTickType block_time_ticks );
```

Parameters, and return values, are fully described in the ASF API documentation, and in source file comments directly above the respective function prototypes.

In all cases, the first parameter is the handle of a peripheral that was opened using a FreeRTOS ASF initialization function (see Listing 1). Standard (as opposed to fully asynchronous) transmit functions can only be used if the freertos_driver_parameters.options_flags parameter passed into the initialization function had the WAIT_TX_COMPLETE bit set.

The behavior of standard transmit functions is defined in Figure 1-1. The task that calls the transmit function does not exit the transmit function until all the data has been completely sent, or the time specified by the block_time_ticks parameter has expired.

Block times are specified in RTOS tick counts. The portTickType type is defined by FreeRTOS. To specify a block time in milliseconds, divide the millisecond value by portTICK_RATE_HZ, and pass the result in the block_time_ticks parameter. portTICK_RATE_HZ is a constant defined by FreeRTOS.
Figure 1-1. The behavior of standard (blocking) transmit functions.

Listing 7 is a code snippet that demonstrates how to use a standard transmit function.
Listing 7. FreeRTOS ASF standard transmit driver functions.

```c
status_code_t write_two_strings( freertos_usart_if freertos_usart ){
    uint8_t write_buffer[ 5 ];
    status_code_t result;

    /* Send a string to the USART. The string must be in RAM, so copy it into an array. */
    strcpy( write_buffer, "one" );

    /* Using a block time of 10 / portTICK_RATE_MS means “don’t block any longer than 10ms”. */
    result = freertos_usart_write_packet( freertos_usart, write_buffer, strlen( "one" ),
                                        10 / portTICK_RATE_MS );

    if( result == STATUS_OK ) {
        /* freertos_usart_write_packet() does not return until transmission of the string has
           completed, meaning the write_buffer array can be re-used immediately without any risk
           of corrupting the original transmission. */
        strcpy( write_buffer, "two" );
        result = freertos_usart_write_packet( freertos_usart, write_buffer, strlen( "two" ),
                                              10 / portTICK_RATE_MS );
    }

    /* freertos_usart_write_packet() does not return until transmission of the string has
       completed, meaning the function can exit even though the buffer being transmitted is
       declared on the function’s stack. */
    return result;
}
```

1.3.3 Fully asynchronous transmit functions.

Asynchronous transmit functions are intended for advanced users who are writing time critical and processing intensive applications. Care must be taken not to alter the contents of the buffer being transmitted while the transmission is still in progress.

Fully asynchronous transmit functions are shown in Listing 8.

Listing 8. FreeRTOS ASF fully asynchronous transmit driver functions.

```c
status_code_t freertos_spi_write_packet_async( freertos_spi_if p_spi, const uint8_t *data,
                                            size_t len, portTickType block_time_ticks,
                                            xSemaphoreHandle notification_semaphore );

status_code_t freertos_twi_write_packet_async( freertos_twi_if p_twi, twi_packet_t *p_packet,
                                                   portTickType block_time_ticks, xSemaphoreHandle notification_semaphore );

status_code_t freertos_usart_write_packet_async( freertos_usart_if p_usart,
                                           const uint8_t *data, size_t len, portTickType block_time_ticks,
                                           xSemaphoreHandle notification_semaphore );
```

Parameters, and return values, are fully described in the ASF API documentation, and in source file comments directly above the respective function prototypes.
In all cases, the first parameter is the handle of a peripheral that was opened using a FreeRTOS ASF initialization function (see Listing 1). Fully asynchronous transmit functions can only be used if the freertos_driver_parameters.options_flags parameter passed into the initialization function had the WAIT_TX_COMPLETE bit clear.

The behavior of standard transmit functions is defined in Figure 1-2. The task that calls the transmit function exits the transmit function as soon as the transmission starts, and uses the notification semaphore to know when the transmission has ended. The buffer being transmitted must exist, and cannot be modified, until the transmission has ended.

Block times are specified in RTOS tick counts. portTickType is a type defined by FreeRTOS. To specify a block time in milliseconds, divide the millisecond value by portTICK_RATE_HZ, and pass the result in the block_time_ticks parameter. portTICK_RATE_HZ is a constant defined by FreeRTOS.

Figure 1-2. The behavior of fully asynchronous transmit functions.

Listing 9 is a code snippet that demonstrates how to use a fully asynchronous transmit function.
1.4 Receiving

On synchronous peripherals, such as the TWI and SPI, data reception timing is controlled by the bus master. On these peripherals, data is only received by the master when the master calls a receive function, the master controls the amount of data being received, and the received data is placed directly into the buffer specified in a parameter to the receive function.

```c

/* xSemaphoreHandle is a FreeRTOS type used to store a handle to a semaphore. In this example, the semaphore has already been created using a call to the FreeRTOS xSemaphoreCreateBinary() API function, and is being passed in as a function parameter. */
status_code_t write_two_strings( freertos_usart_if freertos_usart, 
xSemaphoreHandle notification_semaphore ){ 

    uint8_t write_buffer[ 5 ];
    status_code_t result;

    /* Send a string to the USART. The string must be in RAM, so copy it into an array. The array must exist for the entire time taken to transmit the string. This can be ensured by making it global, static, or by allocating it on the stack and then ensuring the stack frame does not change until the transmission is complete. */
    strcpy( write_buffer, "one" );

    /* notification_semaphore, passed into the function, is used by the FreeRTOS ASF driver to signal that the transmission has finished. Using a block time of 100 / portTICK_RATE_MS means “don’t block any longer than 100ms". */
    result = freertos_usart_write_packet_async( freertos_usart, write_buffer, strlen( "one" ),
                                                  100 / portTICK_RATE_MS, notification_semaphore );

    if( result == STATUS_OK ) {
        /* Transmission of the string was started successfully. */
    }

    /* .. other processing can be performed here, while the string is being transmitted .. */

    /* Another string is going to be sent, but the write_buffer array must not be altered until the original transmission is complete. If the notification semaphore is available, then the transmission is complete and the following function call will return immediately. If the notification semaphore is not available, then the following function call will place this task into the Blocked state for a maximum of 200ms to wait for it to become available (other tasks will execute during the wait). */
    xSemaphoreTake( notification_semaphore, 200 / portTICK_RATE_MS );
    strcpy( write_buffer, "two" );
    result = freertos_usart_write_packet_async( freertos_usart, write_buffer, strlen( "two" ),
                                                        100 / portTICK_RATE_MS, notification_semaphore );

    /* .. other processing can be performed here, while the string is being transmitted .. */

    /* In this example, the array being transmitted is declared on the stack. If this function exits, the array will no longer exist, and if it was still being transmitted, the transmitted data can be corrupted. Therefore, xSemaphoreTake() is used again to ensure the transmission has completely finished before allowing the function to return. */
    xSemaphoreTake( notification_semaphore, 200 / portTICK_RATE_MS );

    return result;
}
```
On asynchronous full duplex peripherals, such as the USART, data can be received at any time, not just when a receive function is called. On these peripherals, to ensure data is not lost, the FreeRTOS ASF drivers are configured to place all received data into a dma buffer, no matter when the data arrives. The dma buffer used is that specified by the freertos_driver_parameters.receive_buffer parameter to the function used to initialized the peripheral (see Listing 5). When a receive function is called, the data already in the dma buffer is copied into the user buffer specified in a parameter to the receive function. The receive function returns the number of bytes copied into the user buffer. Available but unread data remains in the dma buffer, ready for the next receive function call.

1.5 Receive functions for SPI and TWI
FreeRTOS ASF functions use the PDC to transfer data from a peripheral to a buffer. The microcontroller continues executing application code while the data is being received.

Data can be received using either a standard (blocking) function call, or a fully asynchronous function call. The next two sub-sections describe the two options.

Note: In an RTOS application, Blocking does not mean polling or spinning, or in fact, using any processing time at all. If the executing task (thread of execution) enters the Blocked state, the RTOS will ensure no processing time is wasted by immediately starting to execute a different task.

1.5.1 Standard receive functions for SPI and TWI
Standard receive functions for the SPI and TWI peripherals are shown in Listing 10.

Listing 10. FreeRTOS ASF standard receive functions for SPI and TWI peripherals.

```c
status_code_t freertos_spi_read_packet( freertos_spi_if p_spi, uint8_t *data, uint32_t len,
portTickType block_time_ticks );

status_code_t freertos_twi_read_packet( freertos_twi_if p_twi, twi_packet_t *p_packet,
portTickType block_time_ticks );
```

Parameters, and return values, are fully described in the ASF API documentation, and in source file comments directly above the respective function prototypes.

In all cases, the first parameter is the handle of a peripheral that was opened using a FreeRTOS ASF initialization function (see Listing 1). Standard (as opposed to fully asynchronous) receive functions can only be used if the freertos_driver_parameters.options_flags parameter passed into the initialization function had the WAIT_RX_COMPLETE bit set.

The behavior of standard receive functions is defined in Figure 1-3. The task that calls the receive function does not exit the receive function until all the requested data has been completely received, or the time specified by the block_time_ticks parameter has expired.

Block times are specified in RTOS tick counts. portTickType is a type defined by FreeRTOS. To specify a block time in milliseconds, divide the millisecond value by portTICK_RATE_HZ, and pass the result in the block_time_ticks parameter. portTICK_RATE_HZ is constant defined by FreeRTOS.
Figure 1-3. The behavior of standard receive functions for SPI and TWI peripherals.

Listing 11 is a code snippet that demonstrates how to use a standard receive function.
Listing 11. Using a standard receive function on a synchronous master peripheral.

```c
void a_function( freertos_spi_if freertos_spi )
{
    /* The receive buffer is declared static to ensure it does not overflow the task stack. */
    static uint8_t receive_buffer[ 50 ];
    const max_block_time_50ms = 50 / portTICK_RATE_MS;

    /* Loop around, reading and then processing 20 bytes from freertos_spi on each iteration. */
    for(;; )
    {
        /* Receive 20 bytes from freertos_spi into receive_buffer. */
        if( freertos_spi_read_packet( freertos_spi, receive_buffer, 20,
                                         max_block_time_50ms ) == STATUS_OK )
        {
            /* freertos_spi_read_packet() does not return until all the requested bytes have
               been received, so it is known that the data in receive_buffer is already complete,
               and can be processed immediately. */

            /* ... Process received data here ... */
            do_something( receive_buffer );
        }
    }
}
```

1.5.2 Fully asynchronous receive functions for SPI and TWI peripherals

Asynchronous receive functions are intended for advanced users who are writing time critical and processing intensive applications.

Fully asynchronous receive functions for the SPI and TWI peripherals are shown in Listing 12.

Listing 12. FreeRTOS ASF fully asynchronous receive driver functions for the SPI and TWI peripherals.

```c
status_code_t freertos_spi_read_packet_async( freertos_spi_if p_spi, uint8_t *data,
                                             uint32_t len, portTickType block_time_ticks,
                                             xSemaphoreHandle notification_semaphore );

status_code_t freertos_twi_read_packet_async( freertos_twi_if p_twi, twi_packet_t *p_packet,
                                              portTickType block_time_ticks, xSemaphoreHandle notification_semaphore );
```

Parameters, and return values, are fully described in the ASF API documentation, and in source file comments directly above the respective function prototypes.

In all cases, the first parameter is the handle of a peripheral that was opened using a FreeRTOS ASF initialization function (see Listing 1). Fully asynchronous receive functions can only be used if the freertos_driver_parameters.options_flags parameter passed into the initialization function had the WAIT_RX_COMPLETE bit clear.

The behavior of standard receive functions is defined in Figure 1-4. The task that calls the receive function exits the receive function as soon as the reception starts, and uses the notification_semaphore to know when the reception has ended.

Block times are specified in RTOS tick counts. portTickType is a type defined by FreeRTOS. To specify a block time in milliseconds, divide the millisecond value by portTICK_RATE_HZ, and pass the result in the block_time_ticks parameter. portTICK_RATE_HZ is a constant defined by FreeRTOS.
Figure 1-4. The behavior of fully asynchronous receive functions for the SPI and TWI peripherals.

Listing 13 is a code snippet that demonstrates how to use a fully asynchronous receive function with a synchronous master peripheral.
Listing 13. Using a fully asynchronous receive function.

/* This example demonstrates how a single task can process data while additional data is being received on the SPI bus. Error checking is omitted to simplify the example. */
void a_function( freertos_spi_if freertos_spi )
{
    /* The buffers into which the data is placed are too large to be declared on the task stack, so are instead declared static (making this function non-reentrant - meaning it can only be called by a single task at a time, otherwise multiple tasks would use the same buffers). */
    static uint8_t first_receive_buffer[ BUFFER_SIZE ], second_receive_buffer[ BUFFER_SIZE ];
xSemaphoreHandle first_notification_semaphore = NULL, second_notification_semaphore = NULL;
const max_block_time_500ms = 500 / portTICK_RATE_MS;

    /* Create the notification semaphores, one per buffer. vSemaphoreCreateBinary() is a FreeRTOS API function. */
vSemaphoreCreateBinary( first_notification_semaphore );
vSemaphoreCreateBinary( second_notification_semaphore );

    /* Nothing has been read over the SPI bus yet, so make sure both semaphores are empty. */
xSemaphoreTake( first_notification_semaphore, 0 );
xSemaphoreTake( second_notification_semaphore, 0 );

    /* Start an asynchronous read to fill the first buffer. The function will be able to access the port immediately because nothing else has accessed it yet - allowing the block_time_ticks value to be set to 0. */
freertos_spi_read_packet_async( freertos_spi, first_receive_buffer, BUFFER_SIZE, 0,
                        first_notification_semaphore );

    for( ;; )
    {
        /* Wait until the first buffer is full. Other tasks will run during the wait. */
xSemaphoreTake( first_notification_semaphore, max_block_time_500ms );

        /* Start an asynchronous read to fill the second buffer. Again block_time_ticks is set to zero as it is known that the read operation that was filling the first buffer has completed - leaving the SPI port available. */
freertos_spi_read_packet_async( freertos_spi, second_receive_buffer, BUFFER_SIZE, 0,
                        second_notification_semaphore );

        /* Process the data in the first receive buffer while the second receive buffer is being refreshed. */
process_received_data( first_receive_buffer );

        /* Wait until the second buffer is full. Other tasks will run during the wait. */
xSemaphoreTake( second_receive_buffer, max_block_time_500ms );

        /* Start an asynchronous read to fill the first buffer again. */
freertos_spi_read_packet_async( freertos_spi, second_receive_buffer, BUFFER_SIZE, 0,
                        second_notification_semaphore );

        /* Process the data in the second receive buffer while the first receive buffer is being refreshed. */
process_received_data( second_receive_buffer );
    }
}
1.6 Receive functions for USART

The USART only has one receive function because all USART reception operations are asynchronous. The USART receive function is shown in Listing 14.


```c
uint32_t freertos_usart_serial_read_packet( freertos_usart_if p_usart, uint8_t *data, uint32_t len, portTickType block_time_ticks);
```

Parameters, and return values, are fully described in the ASF API documentation, and in source file comments directly above the respective function prototypes.

The FreeRTOS ASF USART driver uses the PDC to transfer data from a peripheral to a circular buffer (the DMA buffer). Reception happens in the background, while the microcontroller is executing application code.

`freertos_usart_serial_read_packet()` copies bytes from the DMA buffer into the buffer passed as a parameter.

The behavior of `freertos_usart_serial_read_packet()` is defined by Figure 1-5.

Figure 1-5. The behavior of the USART receive function.

[Diagram showing the flow of operations for the USART receive function]

Listing 15 is a code snippet that demonstrates how to use `freertos_usart_serial_read_packet()`.
Listing 15. Using the freertos_usart_serial_read_packet() function.

void a_function( freertos_usart_if freertos_usart )
{
uint8_t receive_buffer[ 20 ];
uint32_t bytes_received;
portTickType max_wait_20ms = 20 / portTICK_RATE_MS;

    /* Attempt to read 20 bytes from freertos_usart. If fewer than 20 bytes are available, then
     wait a maximum of 20ms for the rest to arrive. */
    bytes_received = freertos_usart_serial_read_packet( freertos_usart, receive_buffer, 20,
                                                      max_wait_20ms );

    if( bytes_received == 20 )
    {
        /* All the bytes were received. The RTOS task calling this function *may* have been
        placed into the Blocked state to wait for all the bytes to be available. Other tasks
        will execute while this task is in the Blocked state. */
    }
    else
    {
        /* Fewer than the requested number of bytes have been received, so the RTOS task
        calling this function did enter the blocked state for the full 20 milliseconds. */
    }
}
# Revision History

<table>
<thead>
<tr>
<th>Doc. Rev.</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>42049A</td>
<td>11/2012</td>
<td>Initial document release</td>
</tr>
</tbody>
</table>