Release Notes - SAM3X-EK Demo

Release version: 1.0
Release date: 2012-02-14

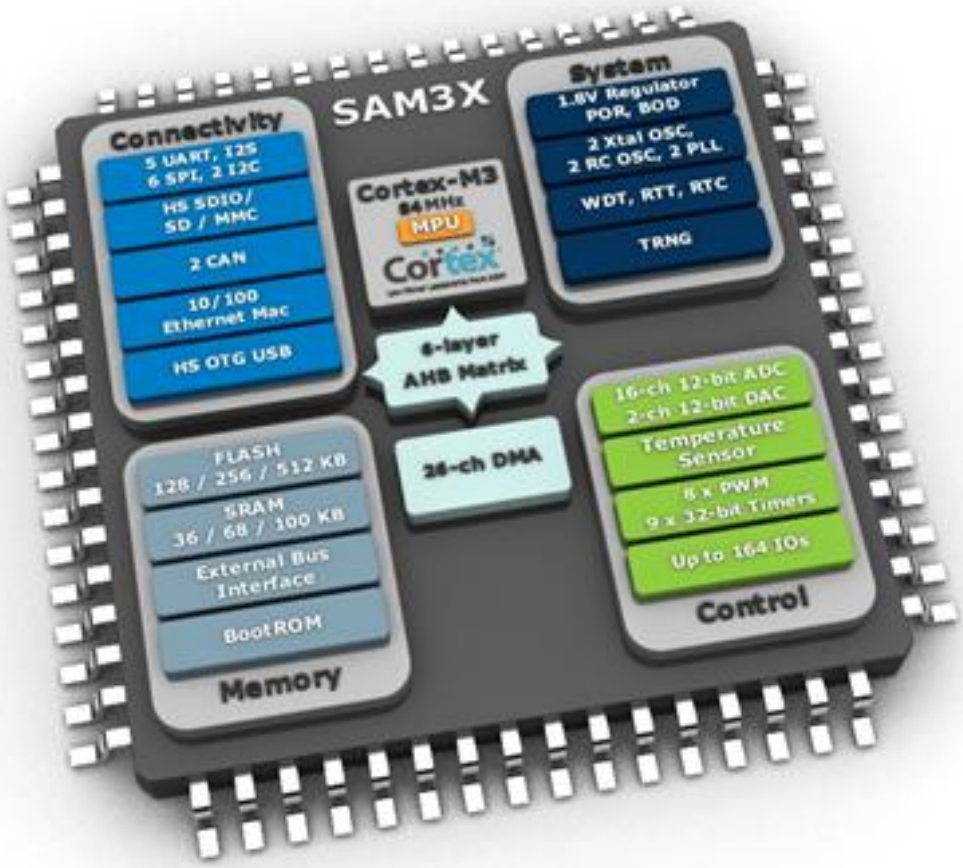


Table of contents

1 Overview

The SAM3X-EK Demo consists in a demonstration applications programmed by default in the SAM3X-EK evaluation kit.

This package includes binary and source code of the full demo application.

This demo package module is provided with full source code, binary, and ready-to-use projects for ARM GCC compilers.

2 Downloading and Installing

2.1 SAM3X-EK BeRTOS HTTP Demo

This demo is based on Open Source RTOS named BeRTOS. This RTOS provides micro kernel and software layers (drivers, services) which make easier the kick off of any new development.

2.1.1 Files

Files present in sam3x_ek_bertos_http_demo_1.0_binaries.7z:

- prog_flash.bat: DOS Batch file using SAM-BA Command Line Interface to upload binaries into device.
- prog_flash_binaries.tcl: SAM-BA script used by prog_flash.bat
- sam3x_ek_bertos_http_demo.bin: Demo binary, to be put into SAM3X8H Flash 0.
- sd_data.img: Demo data, to be put into SAM3X8H Flash 1.

Files present in sam3x_ek_bertos_http_demo_1.0_sources.7z:

- sam3x_ek_bertos_http_demo folder

2.1.2 Instruction for deployment

2.1.2.1 Using prog_flash.bat and JTAG connection

- Install SAM-BA 2.11 or above.
- Connect a SAM-ICE between PC and SAM3X-EK.
- Power ON the board.
- Open a DOS Shell and run prog_flash.bat.
- Power OFF the board.
- Plug an Ethernet cable between board and Router or PC with DHCP server.
- Power ON the board.
- SAM3X-EK will wait to receive the IP address from DHCP server and then display the IP address to be used in a PC web browser.

2.1.2.2 Using SAM-BA and USB connection

- Using the USB connection, upload the file sam3x_ek_bertos_http_demo.bin with SAM-BA at the beginning of SAM3X8H Flash 0. Set the GPNVM "Boot from Flash 0".
- Upload the file sd_data.img at beginning of SAM3X8H Flash 1.
- Power OFF the board.
- Plug an Ethernet cable between board and Router or PC with DHCP server.
- Power ON the board.
- SAM3X-EK will wait to receive the IP address from DHCP server and then display the IP address to be used in a PC web browser.

2.1.3 Instructions for project compilation

2.1.3.1 Using a standalone GCC compiler for ARM

SAM3X-EK demo project requires an installation of a recent GCC compiler for ARM. The path to this toolchain must be present into the system path, either Microsoft Windows, Linux or MacOS.

- Open a shell and go to the demo sources extracted folder.
- Run ‘make’
- The binary shall be present into a newly created ‘images’ sub folder.
- Upload the binaries as described into previous chapter.

2.1.3.2 Using BeRTOS SDK

Open a Web browser and go to URL http://www.bertos.org/download/ to obtain BeRTOS SDK Trial Edition.

Once installed, you can run it and load the SDK workspace present at root of demo source folder: sam3x_ek_bertos_http_demo.workspace.

Compilation is done by pressing F7 key or clicking on the build icon. See BeRTOS web page for help.

2.1.4 Out-of-the-box experience

Once the demo obtains an IP address from DHCP server, it will print on the in-board TFT LCD this address. You have then to use this address in a web browser on PC host or tablet present on the same network.

The demo web page allows you to monitor the SAM3X8H internal temperature sensor, play with in-board LEDs (amber, green, blue) and read the acquired data from in-board potentiometer.

3 Introduction to BeRTOS

BeRTOS is a real time open source operating system supplied with drivers and libraries designed for the rapid development of embedded software. Perfect for building commercial applications with no license costs nor royalties, BeRTOS allows you to cut the economic investment for your products.

BeRTOS supports multiple platforms, including Atmel AVR, SAM7, SAM3 and many other vendors. Visit http://www.bertos.org/discover for a complete list.

3.1 Key features

- Kernel: BeRTOS features both a preemptive and a cooperative kernel with synchronization primitives. It has thread stack monitor, inter-thread messaging, binary semaphores and low level signals.
- Drivers: supports many integrated peripherals, such as ADC, MAC, I2C and serial.
- CPU independent drivers: drivers for DC and stepper motors, keyboard, displays, dataflash and many more. Visit http://www.bertos.org/use#t-drivers for a complete list.
- Other useful libraries included with BeRTOS are TCP/IP stack, AFSK modem, encryption algorithms, GPS protocol, customizable command line interpreter (Telnet style)
- Configuration Wizard: create with a few clicks a project template, select the modules you need and modify the configuration of each module with an intuitive and user friendly interface, or create ready made project for many development boards.

3.2 License

BeRTOS is licensed under a modified GPL that allows you:

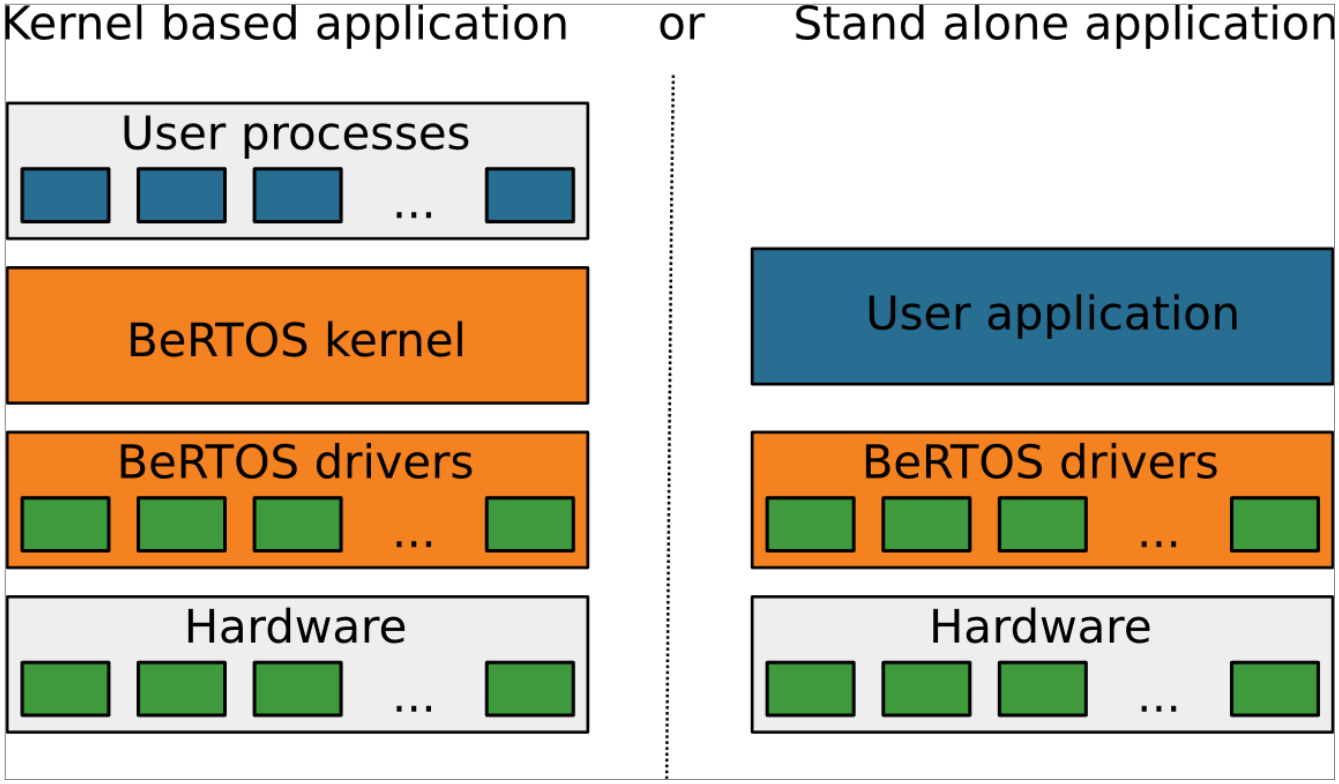
- to include BeRTOS within any product, distributed under any license (including commercial licenses and/or closed-source licenses)
- to modify BeRTOS as you want in any of its part under the following conditions:
    - Attribution: you must declare in a written statement that you are using BeRTOS in your application and offer to provide the (possibly modified) BeRTOS source code.
    - Share-alike: if you modify BeRTOS, you may distribute it only under the original license.

More information about licensing can be found on BeRTOS website: http://www.bertos.org/discover/license

4 Getting started with BeRTOS

This section explains the BeRTOS programming model. Applications may or may not use the kernel, depending on the specific needs.

- Kernel based application: when you create a kernel based application, CPU arbitration is done by the kernel while your application accesses the hardware through the driver's API. This is generally used with mid to high range MCUs;
- Standalone application: when you create a standalone application, you are in charge of splitting CPU time between the various functions of your program. This is generally used with low end MCUs.



Hardware drivers are supported on many different architectures with a single API. This result is achieved using an abstraction over the physical hardware.

The Interface API is the entry point for BeRTOS' users. BeRTOS is committed to provide a stable API across minor releases, so that users can safely upgrade existing programs to get new feature and bug fixes.

It's a specific design decision to make the API hardware agnostic; this way, the API handles high level details only and it avoids irrelevant concepts. For example, the API for an SD card reader must provide only functions to initialize the card and read/write from it, but it must ignore which pins the card is connected to.

The driver itself must ignore the details of the underlying hardware connections, so that it's possible to use the same driver across different boards and devices.

Generic macros or functions are defined in places where it's necessary to access the CPU registers or pins. Such macros form the BeRTOS HAL layer and they must be implemented by the user.

5 Hello world with BeRTOS

Download BeRTOS and launch BeRTOS Wizard (http://www.bertos.org/use/tutorial-front-page/wizard-user-guide). From `cfg` select `debug`. This will automatically include debug support in BeRTOS. Set the correct output port for your cpu and then create the project. Let's call our project hello_world.

Each BeRTOS project is organized as follows:

`bertos/`: the whole bertos source tree;

`Makefile`: the makefile for the whole project. To build the project, simply launch `make`;

`<project_name>/`: our project directory

`cfg/`: directory with all the configuration files

`hw/`: hardware specific files

`main.c`: our application entry point

Open main.c and input:

```
#include "buildrev.h"
#include <cfg/debug.h>
int main(void)
{
    IRQ_ENABLE;
    kdbg_init();
    kprintf("Program build: %d\n", VERS_BUILD);
    kputs("Hello world!\n");
    return ;
}
```

Then compile the program; you will see some warnings but for now you can safely ignore them. Flash your board and reset. On the debugging serial you'll see these messages:

```
*** BeRTOS DBG START ***
Program build: 17
Hello world!
```

Congratulations! You have built your first BeRTOS program!

5.1 Line by line tutorial

```
#include <cfg/debug.h>
```

The first line includes function prototypes. Since this is often included by other files, you can omit it in larger projects.

```
IRQ_ENABLE;
```

This line enables IRQs on your CPU. We don't need it for this simple example, but you will need for almost every other module in BeRTOS, so it's a good habit to learn as soon as possible.

```
kdbg_init();
```

This line initializes the debug subsystem, opening a serial port for debugging. The parameters for this port are in `hello_world/cfg/cfg_debug.h`.

```
kprintf("Program build: %d\n", VERS_BUILD);
kputs("Hello world!\n");
```

Writes a debug string on the output. You can also use a `printf`-like function to format the output. Be aware, though, that the delay introduced by `printf` is high, so use it sparingly.

To reduce footprint and cpu usage, BeRTOS implements different types of formatting for `printf`. You can select your formatting options modifying `hello_world/cfg/cfg_formatwr.h`. Remember the warnings we got on the first build? These were due to using full formatting on an underpowered cpu. To avoid the warnings you can change printf format option to `PRINTF_NOFLOAT`.

6 Process management

If you have programmed with threads in a desktop operating system, you will find some similarities with BeRTOS processes. Every process is defined by:

- a function that is executed when the process is running;
- some user data that the process can use to communicate with other processes;
- a memory area used for the stack (remember that BeRTOS has a fully static memory model).

Unlike threads, however, processes have are independent from the parent process, they can have a priority (if enabled) and they can be monitored to detect stack overflows.

First example

In this example we are going to create three processes (one the main process plus two extra) that control some LEDs.

First, initialize the scheduler calling `proc_init()`, then create two processes using `proc_new()`. Remember that you need to provide some stack space to the process; usually KERN_MINSTACKSIZE should be enough, but read carefully the documentation of this variable if you're using a 16-bit architecture.

You can allocate the stack for both processes with the following lines:

```
PROC_DEFINE_STACK(stack1, KERN_MINSTACKSIZE);
PROC_DEFINE_STACK(stack2, KERN_MINSTACKSIZE);
```

This macro defines a memory buffer. The first parameter is the buffer name, while the second is the size in bytes.

Note:

It's not possible to determine the optimal stack size for a generic process before running it. `KERN_MINSTACKSIZE` is a good value to start working, but it's not the best value for every use case. For example, a process which uses `kprintf()` must have a bigger stack, at least `(KERN_MINSTACKSIZE * 2)`, because formatting requires lots of memory.

Then let's define the entry points for the two processes:

```
void proc1(void)
{
```

```
        int times = 0;
        bool light_on = false;
        while (true)
        {
                light_on = !light_on;
                // light on or off led1
                TURN_LED_ON(1, !light_on);

                ++times;
                timer_delay(20);
                if (times > 30)
                        return;
        }
}

// proc2 is similar to proc1
void proc2(void)
{
        int times = 0;
        bool light_on = false;
        while (true)
        {
                light_on = !light_on;
                // light on or off led2
                TURN_LED_ON(2, !light_on);
                ++times;
                timer_delay(40);
                if (times > 30)
                        return;
        }
}
```

As you can see, you can define variables inside the process and they will be automatically saved: that's why we defined a stack space a bit larger. Of course you may need more stack space in your application, so feel free to create a larger stack for your processes. You can also create a larger stack for one process only, if you need that.

The macro `TURN_LED_ON(led, value)` turns the led `led` on or off depending on `value`. This macro must be defined in the `hw/` directory, which contains all hardware specific macros and functions, in the file `hw_leds.h`. See HAL tutorial for further explanations on `hw` files.

Remember:

If the entry point of the process reaches the end of the function, the process will quit (just like any plain program). Thus, if you want to create a process that will run forever in your application, you must enclose the function with a never-ending loop like `while (true) { ... }` above. You can still exit the process under certain conditions by using return.

Now in our main program we just need to create the two processes and set the priority for each of them.

```
void main(void)
{
        proc_init()
        // other initializations...
        // ...
        Process *p = proc_new(proc1, NULL,
sizeof(stack1), stack1);
        proc_setPri(p, -5);

        p = proc_new(proc2, NULL, sizeof(stack2),
stack2);
        proc_setPri(p, 10);

        // now wait for both processes to complete
        // note that "main" is itself a process
        ticks_t start = timer_clock();
        while (timer_clock() - start <
ms_to_ticks(3000))
        {
                kprintf("main\n");
                timer_delay(500);
        }
}
```

We create two processes with `proc_new()`, which takes the entry point, some user data (NULL in this case) and the stack to operate on.

We also define a *priority* for each of the two processes. Priorities are signed integers, where positive numbers have a higher priority over negative numbers. You should set priorities for your processes in the range -10,+10 (inclusive) to avoid interfering with other system tasks.

We're done. If you run the example, you will note that the second led will light before the first at the start. You can also add some debug prints to see process scheduling.

6.1 Process monitor

This process monitors all the other process to check whether a stack overflow has occurred; in this case you get a warning on the debug serial port.

The monitor is enabled with `CONFIG_KERN_MONITOR` flag in `cfg/cfg_monitor.h`, so you can enable it at debug time and remove it when releasing the software.

6.2 Synchronization example

You can synchronize processes using semaphores. In this example we will create one process that reads commands from the serial port and puts them in a queue and one process that reads such commands and lights the LEDs.

A common way to synchronize processes is to use semaphores. In our example you would lock the FIFO queue with the semaphore and then access it to read (or write) commands.

```
      static void serialRead(void)
      {
         while (1)
         {
            /* read data from serial line... */
            size_t count = kfile_read(&ser_port.fd, buf,
sizeof(buf));
            /* ...put the data in the queue */
            ...
         }
      }
```

The code for the second process (the "worker") has the same structure, but instead of reading from serial line, it will get some data from the queue, interpret it and it will finally light the LEDs.

As you can see, thanks to the cooperative kernel we know each time that memory access is exclusive, so there's no need to lock shared data structures. You can safely access each data without the risk of race conditions.

It's important to stress that all BeRTOS modules are designed to **automatically switch context** when there is a wait in progress, so there's no need to worry to release the CPU manually. In the above example, the function `kfile_read()` automatically releases the CPU when there are no more input chars on serial port. If you need to write new code which requires lots of CPU power, you must remember to release the CPU from time to time using the `cpu_relax()` function.

Semaphores can be useful in some corner cases, such as:

- two processes access the same serial: `kfile_read()` executes a context switch, so the serial must be protected with a semaphore;
- two processes access the same structure and there can be a context switch in the meanwhile. Example:

```
uint8_t buf[10];


void proc1(void)
{
   ...
   /* this will switch context if there are no
chars to read */
   kfile_read(&ser.fd, buf, sizeof(buf));
   ...
}


void proc2(void)
{
   ...
   /* whoops, race condition */
   kfile_read(&ser.fd, buf, sizeof(buf));
}
```

Have a look at semaphore API for further explanations.

6.3 Messages

Synchronizing processes using semaphores can be tricky, since some code paths can lead to deadlocks and race conditions. For example, if you forget to release a semaphore after doing your work, no other process can access the data the semaphore protects. Also, you must remember to explicitly release the CPU when done, which is error prone.

To avoid such problems, BeRTOS provides a very lightweight yet powerful communication method: **Message Queues**. The concept behind this tool is easy: define your custom message, declare an input port for each process and wait for signals coming from the port.

6.3.1 Defining custom messages

The Msg struct is minimalistic, so you need to expand it to contain whatever data you want.

```
#include <kern/msg.h>
typedef struct Command
{
   Msg msg;
   int cmd_id;
   int parameters[MAX_PARMS_CNT];
   int result;
} Command;
```

We have defined a Command which has an id and some parameters to operate on, together with the return code. Now we can send it to a process and let it do the work.

6.3.2 Declare the input port

Each process that wants to communicate with other processes must have an **input port** and, optionally, a **reply port**. In our example the input port will be used by other processes to schedule tasks on the worker process. Remember that tasks in the message queue are processed in FIFO order.

Let's declare a process with its ports:

```
MsgPort proc1_in_port;
```

```
MsgPort main_reply_port;

void proc1_main(void)
{
   // we will see this in a moment
   msg_initPort(...);
   while (1)
   {
      // wait for signals, see below
   }
}

int main(void)
{
   // init everything here
   //...
   // also init reply port
   msg_initPort(...);

   Command cmd;
   cmd.cmd_id = CMD_WASTE_TIME;
   cmd.parameters[0] = 1000;     //secs
   cmd.msg.replyPort = &main_reply_port;
   msg_put(&proc1_in_port, &cmd.msg);
   // ...
   // wait for reply signals, see below
}
```

The main process creates a worker process then sends commands through the message port. The command structure must be filled with valid data (of course); we also fill in the reply port to get the answer. When everything is done, just put it in the port and you're (almost) done.

To init a port, you must declare which signal will be emitted when a message is received. Signals are a kernel feature which is very cheap to use but it carries almost no information; see signals documentation. However, in this case signals are just enough!

So, how do we init a port? This is the code:

```
/* rename standard signal to something more useful */
```

```
#define SIG_CMD_ARRIVED SIG_USER0
msg_initPort(&proc1_in_port,
event_createSignal(proc_current(), SIG_CMD_ARRIVED);
```

First, rename one of the standard signals SIG_USERn to something sensible; then init the port with the above line. Whenever a new message arrives, a signal is sent to the process. How do you wait for a signal? It's just one call:

```
/* wait just one signal, but you can OR more signals
at the same time */
sigmask_t sigs = sig_wait(SIG_CMD_ARRIVED);
```

This call will put the process to sleep, waiting for a signal. Note that the process will **not be blocked in a spinlock**, polling the signal somewhere, **nor it will be in the ready queue** in the kernel; it will be frozen without wasting CPU resources. When the signal arrives, there's a context switch to the waiting process, which will wake up and resume execution.

Once the command is executed, you may want to return a code to let the calling process know what happened. To this end you can use the reply port we defined earlier.

```
...
/* get the first message from port */
Command *cmd = (Command *)msg_get(&proc1_in_port)
// execute the command
switch (cmd->cmd_id)
{
   ...
}
/* res is the return code of the above commands */
cmd.result = res;
msg_reply(cmd);
```

See that the message already knows which is its reply port, so you don't need to specify it. Note that when using `msg_get()` without putting the message into another port, you must take care of recycle the message, by `free()`ing it or, with static memory, by putting it again in a free message pool.

One final remark: you can put really everything inside a message, be it strings, function pointers, arrays or other structures. **There won't be any performance hit** depending on the size of message structure because messages are simply moved from one port (a linked list) to another, so moving a message between ports is just a matter of swapping a few pointers.

7 Known issues

8 Contact Information

For any support on the SAM3X Demo, please send request to at91@atmel.com

9 Copyright and disclaimer