Ronetix Application Note AN001: Developing an Atmel AT45DB DataFlash Device Driver for eCos

Introduction:
It is essential for product development to have a low start-up cost; thus, it is best if proven free development tools like the GNU GCC toolchain and the royalty free RTOS eCos can be used for your project.

GNU GCC needs no introduction, since it is the official Linux development toolchain, but in a few words: GNU GCC tools are a high quality, free, open-source bundle that has support for a large number of processors, from the 8-bit AVR to the 32-bit ARM.

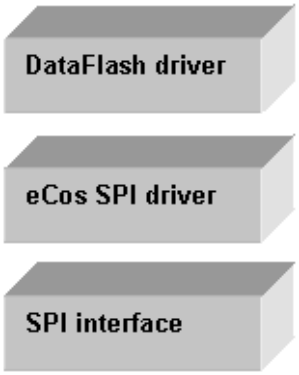
eCos has been ported to many platforms, including the recently popular ARM7TDMI core (a.k.a. the 8051 of the 21st century) that has been designed into most 32-bit microcontrollers today. eCos is so configurable, that its ROM footprint may vary from 20KB (kernel configuration only) to hundreds of kilobytes (including a fully functional TCP/IP stack, file system support, etc.). eCos comes with many ready to use packages, such as a FAT file system implementation, TCP/IP stack, Ethernet and other device drivers. But the volunteers working on that OS cannot possibly write every driver you may need, so you need to write your own driver if the device you use in your project is not already supported in eCos.

This article will show you how to build an eCos device driver for Atmel's SPI DataFlash chips family.

The piece of hardware that we are going to run eCos on Ronetix EB55800 evaluation board, which is powered by an Atmel AT91M55800 CPU. The board is equipped with 512KB external SRAM, 2MB of FLASH and an AT45DB011B – 1 megabit DataFlash for data storage.

Every debugger needs a hardware interface to debug embedded devices. I use Ronetix's PEEDI, which I designed as a 2-in-1 tool: a debugger interface supporting most debuggers and a FLASH programmer with its own database of more than 900 supported FLASH chips. PEEDI can also operate as a stand-alone device (no desktop PC connected to or any file servers, just a power cord) in a push-to-program manner, saving huge amounts of time when many boards need to be programmed.

Implementing the Driver:
The driver I will show you here is a DataFlash driver which lies over the AT91M55800 SPI driver, since the AT45DB011B an SPI device.



Fortunately, eCos provides an SPI driver for the AT91M55800 in its port, so we can just use it without any effort. Every eCos device driver must use two OS-provided macros to register itself, so its init function gets called every time the OS boots and its lookup function is called whenever the user application invokes cyg_io_lookup() to get the handler to the device, specifying the device name as an argument. These two macros are BLOCK_DEVIO_TABLE() and BLOCK_DEVTAB_ENTRY().
The eCos docs about the SPI driver say that we need to make sure a single SPI operation is taking part at that time, so a mutex is what we need to synchronize the ISP access:

```
cyg_mutex_t mutex;            // mutex used to synchronize the access
```

Every driver must export pointers to at least five mandatory functions:
– init function as mentioned before, called with the OS boot
– **lookup** function called each time the user application gets a handle to the device
– **write** and **read** functions used to actually transfer data
– **set** and **get** device configuration functions.

In our case the init function is only used to initialize the mutex and the SPI device handle:

```
cyg_mutex_init( &mutex );         // init the mutex we use
spi_device = cyg_spi_dataflash_dev0;
```

Later, the driver functions must lock the mutex on entry and release it on exit:

```
cyg_mutex_lock( &mutex );

cyg_mutex_unlock( &mutex );
```

Ok so far. Here comes the driver lookup function. This function typically sets up the device for actual use, turning on interrupts, configuring the controller, etc. But we will do nothing here because all the required work is done by the eCos SPI driver.

We will also need a function which will wait for the DataFlash to finish the previous internal operation. The idea is to poll the RDY flag for a predefined timeout period. This faces us for the first time with the issue of sending commands to the DataFlash chip. Every SPI device may send data while receiving, so when issuing a command we are going to use two data buffers, the first is for the outgoing data, in our case the command to the chip, and the second is the incoming data what the chip sends to us in the STATUS REGISTER READ command is the very status register value, which we will check

```
cyg_spi_transfer(spi_device, FLAG_POLLED, 2, tx_buff, rx_buff);
```

cyg_spi_transfer() function sends and receives given data via the SPI interface. The first argument is the handle to the interface, the second is the type of the transfer i.e. polled, not interrupt driven, follows the number of bytes to be sent/received and last are the TX and RX buffers.

The 7th bit of the status register, which is the second byte for the received data, is the RDY bit we are interested in, so let’s check it:

```
if ( rx_buff[1] & 0x80 ) break;     // check if the chip is ready
```

The read and write driver functions are slightly different when working with the SPI interface: they use cyg_spi_transaction_begin(), cyg_spi_transaction_transfer() and cyg_spi_transaction_end(), This is due to the complicated commands used to read:

```
// start SPI access
//=================
cyg_spi_transaction_begin( spi_device );

// send command
//=============
cyg_spi_transaction_transfer( spi_device, FLAG_POLLED, 8, cmd, NULL, 0);

// receive data
//=============
cyg_spi_transaction_transfer( spi_device, FLAG_POLLED, PAGE_SIZE, NULL, buffer, 0);

// end of SPI operation
//=====================
cyg_spi_transaction_end( spi_device );
```

In the driver write function, before every new page is written, the chip must have completed the previous write operation, so the chip status must be checked

We are almost done, but we need to implement the driver get/set configuration functions. There is not much to set the driver, so this function is kind of empty. I made the get config function to return the FlashData chip metrics:

```
int at45_get_config(cyg_io_handle_t handle, int key, void *buffer, cyg_uint32 *len)
{
    switch ( key )
    {
    case CYG_IO_CONFIG_SIZE:
      if (*len != sizeof(int)) return EINVAL;
       *((int*)buffer) = CHIP_SIZE;
       break;

    case CYG_IO_CONFIG_BLOCK_SIZE:
      if (*len != sizeof(int)) return EINVAL;
      *((int*)buffer) = PAGE_SIZE;
      break;

    default:
      return ENOSYS;
  }
  return ENOERR;
}
```

Bringing the Driver to Life

Sometimes this is the most difficult part to make it work. In the old days, it was not easy to debug code running on embedded targets. But things changed for the better when CPUs were made to include special debug circuitry which allows the developer to directly read and write target CPU registers and memory, or manage code execution using break and watch points using the JTAG interface tools like Ronetix's PEEDI. Now, it is just like debugging a desktop PC application!.

Generally the debugger used (in our case the GNU Insight debugger) connects to PEEDI and sends various debug commands, which PEEDI translates to JTAG commands appropriate for that special debug circuitry (called EmbeddedICE in ARM7 CPUs) to understand. Although our test application is small and the download-to-target speed is not so important here, a real embedded application may exceed 1MB, so the time to download may become noticeable. But by using PEEDI with its fast JTAG access up to 1MB/s, it is really fast to download large projects.

To test the driver I wrote the little test routine below, which simply gets the driver handle, writes something to the DataFlash and reads it back to verify that everything is OK. Here is how I used the eCos driver IO API functions:

```
// declare device handler
//=======================
cyg_io_handle_t handle;

// lookup the device
//==================
cyg_io_lookup( "/dev/at45_dataflash0", &handle );

// write data 'buff' long 'length' pages starting at 'page' number
//================================================
cyg_io_bwrite( handle, buff, &length, page );

// write data 'buff' long 'length' pages starting at 'page' number
//================================================
cyg_io_bread( handle, buff, &length, page );
```

To debug my test app first I need to start the Insight debugger pointing the compiled executable named example1_ram:

```
arm-elf-insight example1_ram
```

To connect to my PEEDI I have to type in the Insight's console window:

```
(gdb) target remote 192.168.1.10:2000
```

Where 192.168.1.10 is the IP address I assigned to my PEEDI, and 2000 is the TCP port to wait for connection from debuggers.

Now I can load my application into target's memory like this:

```
(gdb) load
```

This will load required application sections into target memory at addresses specified during the link process. You can manage these addresses using linker script files. While the load command is being executed, Insight sets PC to the entry point of the application.

I want to make sure that my application starts with all interrupts disabled, so I will set the CPSR register:

```
(gdb) set $cpsr=0xD3
```

Now my application is ready to be debugged, I will put some breakpoints where I need to know what exactly happens with the code I wrote. When ready, I will start execution issuing:

```
(gdb) continue
```

To make my life much easier, I defined my own commands in an Insight init file and told Insight to load that file when starting like this:

```
$ arm-elf-insight --command=my_gdb_init
```

my_gdb_init file contains this:

```
# this will tell gdb to connect to PEEDI using remote protocol
#-------------------------------------------------------------
target remote 192.168.1.10:2000
info target

# the following will define a user command
#-----------------------------------------
define ll
    set $cpsr=0xD3
    load
end
```

Finally, I have to mention that PEEDI may also be used with most of the well known commercial ARM debuggers developed by such companies as ARM, IAR, Green Hills, etc.

Let’s burn it into the FLASH

When testing my driver, I want to see how it works executed from the Flash. Here again, PEEDI gives me a hand revealing the other side in it fast and powerful, yet easy to use FLASH programmer. All I need to do to program my application is to connect to PEEDI via telnet and issue

```
flash program tftp://192.168.1.10/example1_rom.bin
```

Where tftp://192.168.1.10/example1_rom.bin is the path of the file on the TFTP server on my PC.

Additionally, PEEDI simplifies and speeds up the programming of large amounts of targets if I configure it to immediately start programming after a target is connected to the JTAG port, so the only thing I need to do is connect the next target to PEEDI

Future plans

Now having a driver for the data storage, you may wish to build another level of abstraction over it – a file system! A very good choice could be the YAFFS – a fast and lightweight FLASH file system.

Share on LinkedinShare on FacebookShare on Reddit on TwitterShare Via EmailShare on Whatsapp

Share