ARM Industrial Module
AIM 711
Operating System eCos

26th November 2003

Contents

1 Overview

This Documentation should give a starting point of developing an embedded system with the AIM 711 using the realtime operating system *eCos*. The whole sources and tools of *eCos* are included on the AIM development CDROM.

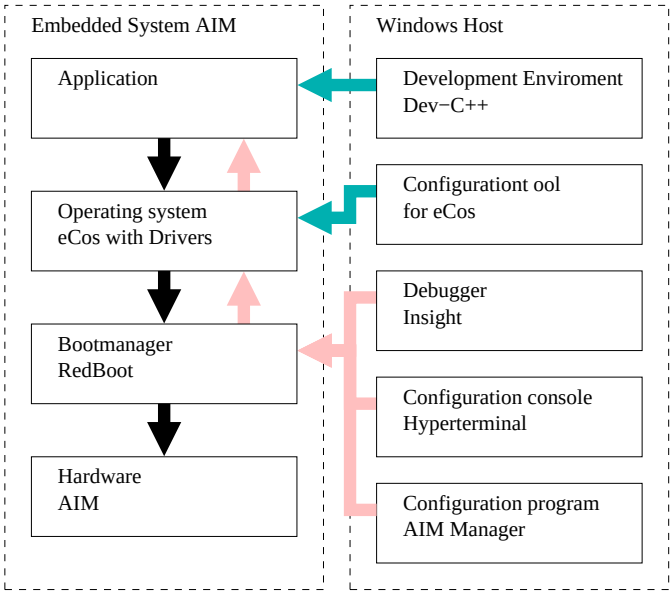

Figure 1: Overview

Special to embedded systems is that development have to be done on an host system, but application should run on the embedded system. So there have to be cross compiler which generates the code for the target system and a possibility to debug the application from the host. In the state of delivering the AIM contains the Bootmanager *RedBoot* which manages the FLASH and makes it possible to debug an application from remote. Figure 1 shows how the host tools communicate with the AIM.

This document describes how to use the components of the AIM with the possibilities of *eCos*.

2 Getting started

At first the bootmanager *RedBoot* have to be configured for your network. If the AIM is not reachable with its default IP-address it could be accessed over the serial interface on the service adapter. The default IP-address of *RedBoot* is set to **192.168.1.254** and the GDB connection port to **9000**.

The best way to start developing an application is to use the *AIM Project Templates* program as described in the AIM Software Documentation. The *Minimal* project is a simple „Hallo World" application, which is shown in Listing 1.

```
// ********************************************************
// * Example program for the ARM Industrial Modul "minimal" *
// ********************************************************

#include <stdio.h>


int main(void)
{
   printf("\nHello World!\n");

   return 0;
}
```

Listing 1: Simple Hello World application

The *AIM_Check* project is more complex does make use of every components contained on the AIM. In the Chapter 4 Components is described how to use these components as done in *AIM_Check*. Many of these features are only available if *eCos* is configured as in *AIM_Check* and will not work if *Minimal* is used as template. The *Minimal* project instead demonstrates how small the footprint of an application could be, although a whole multi threading kernel is integrated.

3 Bootmanager

As bootmanager *RedBoot* is used, which is based on *eCos* too. It is responsible for loading and starting an application and manages the FLASH content. Additionally many things like the date and the MAC-address[1] could be configured over it. *RedBoot* has a simple command line interface to interact with the user. This interface could be reached over the serial interface on the service adapter or over network with telnet[2].

3.1 Network settings

The network settings for *RedBoot* are saved in a special partition in FLASH. The following command `fconfig` could be used to set the network setting:

```
RedBoot> fconfig
Run script at boot:  false
Use BOOTP for network configuration:  false
Gateway IP address:  192.168.1.1
```

[1] The MAC-address is the 6 byte hardware address of the network interface, which must exist only once.
[2] Telnet is a terminal program to communicate over TCP, which exists on almost every host operating system.

```
Local IP address:  192.168.1.254
Local IP address mask:  255.255.255.0
Default server IP address:  192.168.1.1
DNS server IP address:  192.168.1.1
Network hardware address [MAC] for eth0:  0xEA:0x00:0x00:0x0E:0xE5:0x9F
GDB connection port:  9000
Force console for special debug messages:  false
Network debug at boot time:  false
Update RedBoot non-volatile configuration - continue (y/n)?  y
...  Erase from 0x020fe000-0x020ff000:  .
...  Program from 0x007f1000-0x007f2000 at 0x020fe000:  .
```

This includes the MAC-address which is also used later in the application. The all other network settings of the application have to be set by it self. But it is important that the application must have another IP-address as *RedBoot*, else remote debugging will not work.

The default IP-address is set to 192.168.1.254 and the GDB connection port to 9000.

3.2 Date

The AIM has an Real Time Clock, which could be accessed *RedBoot*. The actual date could be set by the command `date` as follows:

```
RedBoot> date 11:20:00 11/14/2003
```

Type only `date` to get the current date printed.

3.3 Store an application

To store an application in FLASH at first it have to be loaded in to RAM using the `load` command. There are many different possibilities to load from. The load command works for example over serial line with X-Modem protocol and over network from a TFTP server or HTTP server. The following example uses the default way a TFTP server:

```
RedBoot> load -m tftp -h 192.168.1.36 -r -b 0x40000 aim_check.bin.gz
Raw file loaded 0x00040000-0x0006a414, assumed entry at 0x00040000
```

This will load the compressed version of *AIM_Check* into RAM. To store the application to flash use:

```
RedBoot> fis create -r 0x40000 -e 0x40040 AIM_Check
An image named 'AIM_Check' exists - continue (y/n)?  y
...  Erase from 0x0202b000-0x02056000:  ....................................
......
...  Program from 0x00040000-0x0006a415 at 0x0202b000:  ....................
....................
...  Erase from 0x020ff000-0x02100000:  .
...  Program from 0x007fe000-0x007ff000 at 0x020ff000:  .
```

The option `-r` is the runtime address of the application and should be alway `0x40000` for the AIM 711 . The next option `-e` is the entry point and should be `0x40040`. With `-f` it is possible to specify where the image should be stored in FLASH, but if it is not used the image will be stored to the first large enough space. The length of an image

is taken from the last `load` command. In the example above does the image exists jet and will be overwritten.

It is also possible to load and store none compressed images, but this documentation restricted to use compressed ones to save FLASH space.

3.4 Run an application

To run an application it have to be first decompressed and then loaded to the runtime address with the following command:

```
RedBoot> fis load -d AIM_Check
Image loaded from 0x00040000-0x00091458
```

Now the application could just be started with `go`:

```
RedBoot> go
...
ARM Industrial Module Checker
...
```

To run an application automaticly after booting a boot script as described in chapter 3.5 should be used.

3.5 Boot script

The boot script in *RedBoot* is just a list of commands executed one after another like a batch file. To create a boot script `fconfig` must be called again and `Run scrip at boot` have to be set to `true`. As next step the commands could be entered:

```
RedBoot> fconfig
Run script at boot:  true
Boot script:
Enter script, terminate with empty line
>> fis load -d AIM_Check
>> go
>>
Boot script timeout (1ms resolution):  10
...
Update RedBoot non-volatile configuration - continue (y/n)?  y
...  Erase from 0x020fe000-0x020ff000:  .
...  Program from 0x007f1000-0x007f2000 at 0x020fe000:  .
```

In the above example the boot script timeout is set to 10 ms, which cause to run the application immediately. Now the AIM can be restarted by typing `reset`:

```
RedBoot> reset
...  Resetting.
+Ethernet eth0:  MAC address ea:00:00:0e:e5:9f
IP: 192.168.1.254/255.255.255.0, Gateway:  192.168.1.1
Default server:  192.168.1.1, DNS server IP: 192.168.1.1
RedBoot(tm) bootstrap and debug environment [ROMRAM]
Non-certified release, version VS 1.1.0 - built 18:25:55, Nov 10 2003
Platform:  AIM 711 (ARM 7TDMI)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.
RAM: 0x00000000-0x00800000, [0x00034428-0x007ef000] available
```

```
FLASH: 0x02000000 - 0x02100000, 256 blocks of 0x00001000 bytes each.
== Executing boot script in 0.010 seconds - enter ^C to abort
RedBoot> fis load -d AIM_Check
Image loaded from 0x00040000-0x00091458
RedBoot> go
...
ARM Industrial Modul Checker
...
```

To abort the boot script `Ctrl-C` can be entered while booting. For accessing *RedBoot* from the serial interface this is time enough, but to access it over network with telnet the boot script timeout have to be set at least to a couple of seconds.

4 Components

4.1 Standard IO

The first things which are used in almost every application at least for debugging output are calls like `printf()` and `scanf()`. They use the *Standard IO* channels `stdin`, `stdout` and `stderr`, which on the AIM in default is the serial interface on the service adapter. While debugging the application with Insight or arm-elf-gdb[3] the standard output is routed to the console of the debugger.

4.2 Serial interfaces

```
int ret, ser_fd;
struct termios ser_termios;

// Open device
ret=open("/dev/termios1", O_RDWR|O_NONBLOCK);
if (ret < 0)
   return ret;
ser_fd=ret;

// Read line control parameter
ret=tcgetattr(ser_fd, &ser_termios);
if (ret < 0)
   return ret;

// Set line control parameter
cfmakeraw(&ser_termios);
ser_termios.c_cflag &= ~(CSIZE|CSTOPB|PARENB);
ser_termios.c_flags |= (CS8|CREAD);
cfsetispeed(&ser_termios, B57600);
cfsetospeed(&ser_termios, B57600);

ret=tcsetattr(ser_fd, TCSANOW, &ser_termios);
if (ret < 0)
   return ret;
```

Listing 2: Open and configure serial interface COM1

The AIM has three serial interfaces in all, whereas the first one is the one on the service adapter which is normally only used for interacting with RedBoot and for being the *Standard IO* interface as described in Chapter 4.1. The service port is the first internal serial interface of the processor. The second one called *COM1* is the external 16550 UART[4] for high speed communication. It is full features with all modem control signals and consequently able to do hardware handshake. The third one called *COM2* is the second internal serial interface of the processor and does not have the features of *COM1* like the modem control signals.

[3] Insight respectively arm-elf-gdb are part of the GNU compiler tool chain.
[4] The 16550 UART is a standard serial interface controller with 16 byte FIFO buffer.

The example code listed in Listing 2 open *COM1* and sets the line control parameter. Whereas `/dev/termios1` is *COM1* used with the *POSIX* compatible termios API.

```
static const char serial_test_string[]="+++ This is the test string +++";
int ret, received;

// Write data to serial interface
ret=write(ser_fd,serial_test_string,sizeof(serial_test_string));
if (ret < 0)
   return ret;

// Wait for incomming characters
received=0;
for (i=0;i<3;i++)
{
   // Try to read data from serial interface
   ret=read(ser_fd,&serial_test_buffer[received], \
         sizeof(serial_test_buffer)-received);
   if (ret < 0)
   {
      if (errno == EAGAIN) // No data available yet
         ret=0;
      else
         return ret;
   }
   received+=ret;
   if (received<sizeof(serial_test_string))
      sleep(1); // Wait one second
   else
      break; // Stop waiting if received the whole test string
}

// Close serial interface
close(ser_fd);
```

Listing 3: Send and receive an example string

In Listing 3 it will be send an example string and tried to receive it back. If a loopback is connected to *COM1* it will receive the echoed string, else it will stop waiting after 3 seconds. In this simple example this is implemented polling three times with an delay of one second each. *ECos* does also support the more powerful `select()` call which should be used in such case.

4.3 Ethernet

The standard way to initialize the network in *eCos* is to configure the *eCos* library with the *configtool* staticly to either use *DHCP*[5] or use a static IP-address. Then the application only have to call `init_all_network_interfaces()` to start the ethernet interface.

If an application have to manage the network settings by it self, it could use the code of the *AIM_Check* example in net.c . It tries to get the network settings from *DHCP* and if it doesn't get a response it use its own fallback settings. Using this code makes it possible to use settings stored for example in FLASH.

```
int s;
struct hostent *hostinfo;
struct sockaddr_in sock_addr;

sock_addr.sin_port=htons((u_short)port);
sock_addr.sin_family = AF_INET;
sock_addr.sin_len = sizeof(sock_addr);

// Try to get the ip-address from an DNS server
if ((hostinfo=gethostbyname(hostname))!=NULL)
   memcpy((char*)&sock_addr.sin_addr, hostinfo->h_addr, hostinfo->h_length);
else
   return -1;

// Get a new socket for TCP/IP
if ((s=socket(AF_INET,SOCK_STREAM,0))<0)
   return -1;

// Connect to the destination
if (connect(s,(struct sockaddr*)&sock_addr, sizeof(sock_addr))<0)
{
   // Close the socket
   close(s);
   return -1;
}
```

Listing 4: Connect to destination using TCP/IP

As shown in Listing 4 in eCos it is possible to use the well known socket API which is also POSIX standard. In this simple example a destination will be connected over TCP/IP. If the host program just makes an echo of th incoming data, the same code as in Listing 3 of the Chapter 4.2 Serial Interfaces could be used to test a network connection.

4.4 FLASH

To access the FLASH there exist the functions `flash_read()`, `flash_erase()` and `flash_program()`. These functions could be used to access the FLASH directly with

[5]The Dynamic Host Configuration Protocol is used to get the network configuration from a central server.

knowledge of which range to use without overwriting the Bootmanager or other important data.

The best way to find out which region of FLASH could be used for application data ist to create a special partition with *RedBoot* called JFFS2. The following command could do that:

```
RedBoot> fis create -f 0x20bd000 -l 0x2000 -b 0x40000 Data
```

```
static char ws[FLASH_MIN_WORKSPACE];
static char flash_test_string[]="+++ This is the flash test string +++";
static char buf[1024];
char *flash_base;
unsigned long size;
void *erraddr;

if (flash_init( ws, FLASH_MIN_WORKSPACE, &printf ))
   return -1;

// Open the "Data" partition
if (!CYGACC_CALL_IF_FLASH_FIS_OP(CYGNUM_CALL_IF_FLASH_FIS_GET_FLASH_BASE,
                         "Data",
                         (CYG_ADDRESS*)&flash_base))
   return -1;
if (!CYGACC_CALL_IF_FLASH_FIS_OP(CYGNUM_CALL_IF_FLASH_FIS_GET_SIZE,
                         "Data",
                         &size))
   return -1;

if (size < sizeof(flash_test_string))
   return -1;

// Erase FLASH region
if (flash_erase( flash_base, sizeof(flash_test_string), &erraddr))
   return -1;

// Save test string to FLASH
if (flash_program( flash_base, flash_test_string,
      sizeof(flash_test_string), &erraddr))
   return -1;

// Read back the test string
if (flash_read( flash_base, buf, sizeof(flash_test_string), &erraddr ))
   return -1;

// Verify the read test string
if (strncmp(buf,flash_test_string,sizeof(flash_test_string)))
   return -1;
```

Listing 5: Write and read the FLASH content

The application could get the base address and the size of the partition over the special `CYGACC_CALL_IF_FLASH_FIS_OP()`functions. The example listed in Listin 5 shows how to do that. It is important to erase the FLASH region first which should be written. For complexer usage of FLASH a filesystem is supported, which is discribed in the next Chapter.

4.5 Filesystem

eCos supports the *Second journaling flash filsystem JFFS2*. It is a log-structured file system designed for use on flash devices in embedded systems. To use the filsystem a partiton called JFFS2 have to be created with *RedBoot*:

```
RedBoot> fis create -f 0x20c0000 -l 0x30000 -b 0x40000 JFFS2
```

```
int fd;
ssize_t wrote, len;
char test_string[] = "This␣is␣a␣test␣string";

// Mounting filesystem (JFFS2)
if (mount("/dev/flash1", "/", "jffs2") < 0)
   return -1;

// Change directory to '/'
if (chdir("/") < 0)
   return -1;

// Make a directory
if (mkdir("testdir", 0) < 0)
   return -1;

// Create a new file
fd = open("testdir/testfile.txt", O_WRONLY|O_CREAT);
if(fd < 0)
   return -1;

// Write a string to the file
wrote = write(fd, test_string, strlen(test_string));
if(wrote != strlen(test_string))
   return -1;

// Close file
close( fd );
```

Listing 6: Use the JFFS2 filesystem

After the first time mounting the filessytem with `mount()` it is empty. To create files and directorys the standard file access functions like open(), mkdir() etc could be used. In the Listing 6 a file are created after first making a directory.

The *AIM_Check* example includes a simple commandline interface to manage the filsystem. Additionaly it is possible to use TFTP[6] to up- and download files. In the file `tftp.c` of *AIM_Check* the TFTP server is started.

4.6 Real Time Clock

The Real Time Clock of the AIM is fully integrated in *eCos*. To get the actual date the POSIX standard function `time()` could be used. It returns the seconds since

[6]TFTP Trivial File Transfer Protocol is a very simple file transfer protocol

00:00:00, January 1, 1970. These seconds could be converted to a readable string using `ctime()`. To set the date such a string could be converted to a special structure using `strptime()`, which then could be converted to seconds back using `mktime()`.

```
time_t t, orig_t;
char *s;
struct tm tm;

// Read seconds since 00:00:00 UTC, January 1, 1970
t=time(NULL);
if (t < 0)
   return -1;

// Save original time
orig_t=t;

// Convert seconds in to real date
s=ctime(&t);
if (s == NULL)
   return -1;

// Print current date
printf("Current date is: %s",s);

// Set date structure
strptime("Fre Nov  7 18:32:12 2003", "%a %b %d %H:%M:%S %Y", &tm);

// Convert date structure to seconds
t=mktime(&tm);
if (t < 0)
   return -1;

// Set date
if (cyg_libc_time_settime(t))
   return -1;
```

Listing 7: Get and set the date and time

In Listing 7 such procedure is used. For writing back the new date to the Real Time Clock the special eCos function `cyg_libc_time_settime()` would be called.

The date could also be set over the *RedBoot* command `date` and with the *AIM Manager*.

4.7 EEPROM

To access the EEPROM which is connected over the I^2C bus there are two AIM specific functions called `hal_aim711_eeprom_read()` and `hal_aim711_eeprom_write()`. This functions are part of the *AIM HAL*[7]. To use them cyg/hal/plf_aux.h have to be included.

Listing 8 how to read and write to the EEPROM. The second argument of the EEPROM access functions is the offset address to access. This makes is possible to read or write

[7]The Hardware Abstraction Layer makes the adaptation of the AIM hardware to *eCos*.

```
static char eeprom_buf[256];
static char eeprom_test_string[]="+++ This is the eeprom test string +++";
int ret;

// Save test string to EEPROM
ret=hal_aim711_eeprom_write(eeprom_test_string, 0, sizeof(eeprom_test_string));
if (ret < sizeof(0))

// Read back the test string
ret = hal_aim711_eeprom_read(eeprom_buf, 0, sizeof(eeprom_buf));
if (ret < 0)
   return -1;

// Verify the read test string
if (strncmp(eeprom_buf,eeprom_test_string,sizeof(eeprom_test_string)))
   return -1;
```

Listing 8: Write and read the EEPROM content

for example only one byte at any place of the EEPROM.

4.8 Service adapter

The service adapter has the following units:

- Serial interface
- Three LEDs
- JTAG interface

As described in Chapter 4.1 Standard IO the serial interface is used for standard input and output of RedBoot and an application. Additionally it is possible to let Insight or arm-elf-gdb connect over it.

```
#define BIT0 (1<<0)
#define BIT1 (1<<1)
#define BIT2 (1<<2)

hal_diag_led(BIT0|BIT2);
```

Listing 9: Switch LED 0 and 2 on

The three LED are mainly used in the startup code. If something is wrong the value of the LEDs would indicate the state where it stopped working. To make usage of the LEDs in an application it exists the call `hal_diag_led(int mask)`included from cyg/hal/hal_diag.h. In Listing 9 LED 0 and 2 will be set.

5 Resources

- The latest eCos documentation could be found at:
  http://sources.redhat.com/ecos/docs-latest
- Firmware updates and newer documentation could be found at:
  http://www.visionsystems.de