

# **RS232 to SD card Data Logger Documentation Version 1.0**

**Feb 15, 2007**

**Developer:**  
**Vikas Agrawal**

## Authenticity

This document is written with the help of various sources, which are covered under the [GNU Free Documentation License](#). The Software Algorithm and Software is designed by me, ofcourse with the help of the document released as per the resources in question. Excerpts from the pdf document from SanDisk is referred and rewritten for a better understanding of the SD Card.

## **Version History**

1.0 - dd-mm-yy – 15-02-2007 : First Publication of the Document.

## **Copyright Information**

Permission is granted to copy, distribute and/or modify this document under the terms of the [GNU Free Documentation License](#), Version 1.2 or any later version published by the [Free Software Foundation](#); with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "[GNU Free Documentation License](#)". The only condition being that the authors copyright information remains intact as a source for the changed document.

# **Table of Contents and Time Sheet**

## **PHASE I**

1. [Introduction.](#)
2. [Tutorials.](#)
3. [Requirements collection.](#)

Requirements for the Windows PC:

1. [Install WinARM.](#)
2. [Install RVDS.](#)

Requirements for the Linux PC:

1. [Install ELDK.](#)
2. [Install TFTP Server.](#)
3. [Install BDI2000.](#)

Requirements for the AT91RM9200-EK:

1. [Install Das U-Boot.](#)

## **PHASE II**

1. [Preparing the AT91RM9200-EK board for programming and debugging.](#)
2. [Using the BDI2000 to directly load the application into SDRAM.](#)
3. [Using the BDI2000 to directly program the application into FLASH.](#)
4. [About GNU Debugger \( GDB \).](#)

## **PHASE III**

1. [History of ARM.](#)
2. [What a SW developer needs to know about the Architecture.](#)
3. [Exception Handling in ARM.](#)

## **PHASE IV**

1. [Start Application Development.](#)
2. [About the SD Card.](#)
3. [The SOFTWARE Algorithm.](#)
4. [Constraints.](#)
5. [Result.](#)
6. [Future Applications.](#)
7. [Reference.](#)

## **PHASE I**

### **Introduction**

The SD Card Data Logger was created as a Master Thesis project during the 6 months training at S1nn Audio System, Esslingen, Germany from September 2005 to February 2006. For a while I had wanted to create a device in Embedded System and this project gave me a lot of insight in the procedure of developing software in the Embedded environment. Considerable amount of help was taken from the internet via the forums, commercial websites and their solutions, experienced colleagues in the company and also from the Professors at the university.

While at S1nn, I had the opportunity to work with AT91RM9200-EK an evaluation board from the Atmel family of microcontrollers based on ARM920T. Intrigued by its power and capability and after several weeks of heavy coding, The SD Card Data Logger was born.

Some quick shoutouts and thanks the following people, for their initial help and support:

- The [AT91 forum](#) where almost all the discussions about the Atmel Family of microcontroller takes place.
- To Martin Thomas and its wonderful work of creating a toolchain for ARM in Windows – WinARM.
- Das U-Boot community especially Mr. Wolfgang Denx for creating a generic Bootloader – Das U-Boot.
- Alex Bourgett, Vincent Devore for their unbelievable patience to answer my silly questions.
- My industry supervisor, Mr Heiko Henkelmann who provided me all the resources I wanted.
- My university supervisor, Mr Joerg Friedrich, for answering my emails and giving a hand of support whenever required.
- The innumerable number of websites and forums where the experts are promptly reply to the queries.

## Tutorials

Courtsey - O Reilly : Embedded system in C and C++

### Compiling:

In the embedded development, application programs are written in a high level language and compiled with a particular compiler. Since every embedded system has their own machine code format, the compiler has to be informed what the target code would look like. Each object file is developed in a particular format ( COFF, or ELF etc. ). There are proprietary format also thereby forcing the tools to be used manufactured only by the vendor. The object file format is laid in a particular structure, with a header, text section, data section ( initialised global variable and their initial values ) and the bss section ( uninitialised variables). The use of a particular format is useful when different compilers are used ( i.e the program is written in different source languages - assembly or C or ADA ). The output of all of them would be able to understand each other. In computing, the Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps. Today the ELF format has replaced executable formats such as a.out and COFF.

### Linking:

There is 'usually' a symbol table too somewhere in the object file that contains the name and location of all the variables within the source file. Some of the symbols are not resolved instantly because not all the variables come from the same file. At the linking stage, all the object file are combined and hence all the symbols are resolved. Apart from this all the text, data and bss section are merged with one another. Each symbol is given an address which assumes that the file started with an address 0x000. Even in the link stage these offsets are retained.

The input to the ld command is the object files which are to be linked. For embedded development a special object file that contains the compiled **startup code** must also be included within the list. The startup code is a small block of assembly language code that prepares the way for execution of software written in a

high level language. Each high level language has got its own expectations about the runtime environment.

Startup code is a small block of assembly language code. Some know startup files look like Startup.asm, crt0.c. They need to be linked together with the application object files. The presence of the startup file is available in the documentation supplied with the compiler for which we are developing the code for. The programmer might have to prevent getting the usual start up code ( the one which is for the default platform i.e host ) getting linked.

Sequence of startup Code:

1. Disable all interrupts,
2. Copy any init data from ROM to RAM
3. Zero the uninit'sed data area.
4. Allocate space for and init the stack.
5. Initialize the processors stack pointer
6. Create and init the heap
7. enable interrupts
8. Call main.

If the same symbol is declared in more than one object file, the linker is unable to proceed. If any of the reference is not resolved even after linking all the object files, the linker will try to resolve the reference on its own -

- the ref might be a fn which is part of a std libraries, in which case the inker looks for the standard library mentioned on the command line ( in the order provided ) and examine their symbol tables.

Standard C library can be changed ( for eg. after downloading from Cygnus), making a few target specific functions and compile the whole lot. The library can then be linked with the embedded software to resolve any previously unresolved standard library calls. The linker produces a spl relocatable copy of the program. But in the embedded system environment, the OS code and data are most likely within the relocatable program too.



Locating:

[http://www.gnu.org/software/binutils/manual/ld-2.9.1/html\\_chapter/ld\\_3.html](http://www.gnu.org/software/binutils/manual/ld-2.9.1/html_chapter/ld_3.html)

Tool that performs the conversion from the relocatable program to executable binary image is called linker. The user provides all the information about the memory on the target board as input to the linker. In GNU, the linker functionality is built right into the linker. This memory information required by the linker can be passed to it in the form of a linker script. Such scripts are sometimes used to control the exact order of the code and data sections within the relocatable program. But here, we want to do more. i.e we want to establish the location of each section in the memory.

Device Drivers:

Most of the actual **hardware initialization** takes place in the second stage ( i.e the stage after the the processor is informed about its environment, the interrupt controller and the critical peripherals are initialized ). The hardware initialization routine starts by initializing the chip select registers ( registers, which enables the memory and IO devices that are connected to the processor ). The association between particular chip select and hardware devices must be established by the hardware engineer. Because at reset, the processor has only the knowledge of 1024 bytes of ROM, it is important to initialize the other memories. After this chip select part is successful, the processor now knows that there is other memory locations too where the code and data can be located.

The base header files:

Creation of a Header file: (Board specific) A file that describes the boards most important features and a piece of software that initializes the hardware to a known state.

There are two types of components of the processor – peripherals and memories. Obviously, memories are for data and code storage and retrieval. Peripherals are specialised hardware devices that either coordinate interaction with the outside world ( IO ). Timer is also a kind of peripheral. It is a counter and hence does a specific hardware function.

Address Spaces: The range of address that can be used to address these memories and the peripherals. The first address space is called the memory space and is intended mainly for memory devices and the second address space is intended mainly for peripherals and is called IO space. However peripherals can also be located in the memory space at the discretion of hardware engineer and then they are called memory mapped IO. Basically the advantage in doing this is just as in a production cost, because a set of dedicated wires to the peripheral device can be removed and hence reduced cost. Other advantage in terms of programmer is that the peripheral can be used via pointers, data structures etc.

It is usually a good idea to associate a software module called a device driver with each of the external peripherals. Basically a device driver is a collection of software routines that control the operation of the peripheral and isolate the application software from the details of that particular hardware device.

Hide the hardware completely. In the end, you want the device driver module to be the only piece of software in the entire system that reads or writes that particular devices control and status registers directly. In addition, if a device generates any interrupts, the ISR that responds to them should be an integral part of the device driver. The benefits of good device driver design are threefold.

1. Easy to understand because of modularized structure.
2. because only one module interacts with the hardware, it is easy to track the hardware.
3. software changes that result from hardware changes are localized to the device driver.

This is how a device driver looks like in a 5 fold process:

1. A data structure that overlays the memory mapped control and status registers of the device. Encapsulate the register layout in a structure. Bind all the related registers to Timer ( ex. ) in a structure. For control registers, one can use #defines.
2. A set of variable to track the current state of the hardware and device driver. For ex. a variable which tracks the hardware initialization and also the hardware timer ( from which a set of software timers can be built ).

3. A routine to initialize the hardware to a known state. This way one can be familiar with the device interaction.
4. A set of routines that, taken together, provide an API for users of the device drivers. After the device has been initialized, one can start adding functionality to the driver.
5. One or more interrupt service routines. It is best to test most of the device driver routines before enabling interrupts for the first time. In the initial stage though it is better to use polling to know the guts of the driver working.

A small introduction about Memory in Embedded Systems:

RAM – SRAM and DRAM, the difference between two is that the DRAM is much less expensive per byte. ( DRAM is usually of the amount megabytes and SRAM is around kilobytes ) SRAM is more known for its 4 times faster access speed and hence are only used for critical purpose.

ROM – PROM, EPROM, and EEPROM.

Hybrid – Flash Memory is the most recent advancement in memory technology. From a software view point, Flash memory and EEPROM technologies are very similar. The major difference is that the Flash driver can be erased only one sector at a time, not byte by byte. Typically sector sizes are in the range of 256 bytes to 16 KB. NVRAM is battery backed up RAM.

Testing Memory : because there can be loose connections, or short circuits or missing memory chips. For this purpose, three types of test are made, Data Bus Test, Address Bus Test, Device Test ( if there are missing peripherals ).

All the programs written to test the memory must be in assembly unless there is already a known working memory which could provide the stack space for the high level language programs. Another option is to run the memory test program from an emulator.

Testing ROM: Since ROMS cannot be overwritten, hence they have a different procedure than testing RAM- checksum and cyclic redundancy. The simplest checksum is to add up all the data bytes ( discarding carriers all the way ). But the

problem in this type is that it cannot detect when all the location is written with 0. By inverting the calculated checksum in the end solves it.

Another problem is that if two bits in a column interchange, checksum remains the same. The error would thus remain undetected. But this can be solved by using CRC.

## **Requirements collection**

“Design of HW and SW for an RS232 to SD-Card Datalogger.

This will be a tool to record internal trace of our ECU's during test drives in the development phase. The unit needs to work with a car independant battery and needs to contain a real time clock to timestamp the trace data. The timestamp can be of a resolution of 1 millisecond“

As per the requirements, the **input** to the device is data in **ascii format** which is send at the rate of 115200 baud and the **output** is the timestamped information which is stored on the SD card for later reading.

## Resources Collection

1. A Windows PC ( optional )
2. A Linux PC with a Ethernet Port
3. BDI2000 Debugger
4. 20 Pin JTAG Cable
5. 2 Serial Cables
6. AT91RM9200-EK Board
7. Ethernet Cables
8. A RVDS Evaluation CD from ARM.

## **Install WinARM**

Follow the following steps to install WinARM.

1. Download WinARM - [http://www.siwawi.arubi.uni-kl.de/avr\\_projects/arm\\_projects/#winarm](http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/#winarm).
2. Unzip WinARM to a folder.

3. Make two entries in the Windows Path Variable. To create the entry, right click on MyComputer, click Properties, click Advanced, and then click Environment Variables. In the System Variables section double click on Path, and then add the following fields separated by a semicolon.
  - a. UnzippedWinARMFolderPath\WinARM\bin
  - b. UnzippedWinARMFolderPath\WinARM\utils\bin
4. Go to the WinARM directory and double click on “pn” folder. Double click pn.exe and it will open the editor – Programmers Notepad for you.
5. Open an existing project from the example directory and click on Tools->Make all.
6. The make should run, creating the required files and should not stop in between. In case of problems, look for troubleshooting.

#### Troubleshooting:

If you have any problems installing or with any of the above mentioned points. Go to [en.mikrocontroller.net](http://en.mikrocontroller.net) and you can search for the relevant answers. Other than that, support is available on #mikrocontroller.net on euIRC.com. To work with IRC channels, you must have an IRC Client for eg. X-Chat or mIRC or many others. Connect to the server by the following parameters if it is not available in the list of already present servers.

Name: choose any name to recognise the network

Server: irc.euirc.net

Port : 6667

More information is present on [www.searchirc.com](http://www.searchirc.com)

The main reason why WinARM is installed although in the next chapter we will force ourselves to use Linux is that WinARM had got plenty of source code examples to take it as a base and start working. This will avoid reinventing the wheel and hence make the job much easier. Once the programs are suitably compiled and linked the same MakeFile and the source code can be easily ported to be used with the below mentioned ToolChain. Another advantage to pre-build the code by using WinARM is that it is easier to work in Windows than in Linux atleast for people who are unfamiliar with the \*NIX environment.

## **Installing RVDS**

Before starting development, I had some general software examples from the Atmel Web Site, but unfortunately none of them were for GNU. They were either compiled for GHS or ADS. So I decided to go for Arm Development Suite and ordered a RVDS Evaluation CD from the following website - <http://www.arm.com/products/DevTools/RVDSEvalCD.html>. The response was prompt and I received the package in 5 days. The process to install was tedious and I would explain it from the beginning.

Evaluation Kits are code-limited and have the following restrictions:  
ARM Evaluation Tools

- \* You may not use the Evaluation Version of the  $\mu$ Vision IDE/Debugger to create commercial products.
- \* Programs that generate more than 16K Bytes of code and data will not compile, assemble, or link.
- \* The evaluation tools create Symbolic Output Format when the RealView compiler is selected. Fully licensed tools generate standard ELF/DWARF files.
- \* The debugger supports programs that are 16K Bytes or smaller.
- \* The RealView Linker does not accept scatter-loading description files for sophisticated memory layouts.
- \* The RealView Linker restricts the base address for code/constants to 0xXX000000, 0xXX800000, or 0x00080000 where XX is 00, 01, ..., FF. This allows memory start address like 0x00000000 and 0x12800000.
- \* It is not possible to generate position independent code or data.
- \* The RealView C/C++ Compiler does not generate a listing file.
- \* The CARM compiler, assembler, and linker are limited to 16K Bytes of object code. Source code may be of any size.
- \* The GNU ARM tools (compiler, assembler, and so on) that are provided are not limited or restricted in any way.

## **Install TFTP Server**

Trivial file transfer protocol - is installed via the Package Manager via a package called tftpd ( tftp daemon ). The entry is automatically entered in /etc/inetd.conf where the base directory can be configured for tftp. The directory mentioned in the inetd.conf file determines the place where the file will be picked from when the tftp

server is started. After configuring the inetd.conf it is important to restart the inet service via the command *'/etc/init.d/inetd reload'*

## Install BDI2000

Before you can use a BDI2000 the following steps must be executed:

- Get from your network administrator a free IP address for the BDI2000.
- Unzip bdisetup.zip in a folder in the home directory
- Enter the folder where bdisetup.zip was unzipped
- Type **make** on the command prompt.
- A new file will be generated called **bdisetup**.
- Connect the BDI2000 to one of the serial channels of your PC (e.g. ttyS0).
- Define the name and path of the configuration file. (e.g /home/guitart/bdi2000/rm9200ek.cfg)
- Power up BDI2000.

## Preparing the Abatron BDI2000 JTAG debugger:

Make a config file that does the minimum register initializations to get Uboot onto your board. If you already have Uboot in flash than you can use a blank [INIT] section that just holds the system at the reset vector.

; bdiGDB configuration file for AT91RM9200-DK

```
; This is the BDI2000 configuration file for AT91RM9200-EK.
```

```
[INIT]
```

```
;empty bcoz U-Boot already loaded.
```

```
[TARGET]
```

```
CPUTYPE      ARM920T
```

```
CLOCK        1                ;JTAG clock (0=Adaptive, 1=16MHz, 2=8MHz, 3=4MHz)
```

```
BDIMODE      AGENT
```

```
BREAKMODE    SOFT 0xDFFDFDFF ;SOFT or HARD, ARM / Thumb break code ( 0xDFFDFDFF )
```

```

STARTUP      STOP 5000          ; RUN , STOP millisec, RESET

[HOST]
IP           192.168.111.132
FILE        project.bin
FORMAT      BIN 0x20000000
LOAD        MANUAL      ;<AGENT> load VxWorks code MANUAL or AUTO after reset
PROMPT      DataLogger> ;new Telnet prompt
DEBUGPORT   2001
START       0x20000000

[FLASH]
WORKSPACE   0x200000 ;workspace in target RAM for fast programming algorithm
CHIPTYPE     AM29BX16 ;Abatron changed .....AT49X16 Flash type is Atmel AT49BV1614A in 16bit
mode
CHIPSIZE     0x800000 ;The size of one flash chip in bytes (e.g. AM29F010 = 0x20000)
BUSWIDTH     16       ;The width of the flash memory bus in bits (8 | 16 | 32)
FILE        project.bin ;The file to program ;The file to program
FORMAT      BIN 0x10020000
ERASE        0x10020000

[REGS]
FILE        reg9200.def

```

The BDI2000 must first be configured to allow communication with your local network, and configured with the parameters for interfacing with the target board. The following steps should be followed:

1. Prepare a PC to act as a host PC and start a TFTP server on it.
2. Connect the Abatron BDI2000 JTAG debugger via both serial and ethernet to the host PC and power it on. Use the serial cable supplied with the BDI2000.
3. Go to the folder where the Abatron BDI2000 bdiGDB is installed on the host PC.
4. Check the serial connection to the BDI by doing the following,



```
$ ./bdisetup -v -p/dev/ttyS0 -b57
BDI Type : BDI2000 Rev.C (SN: 92152150)
Loader    : V1.05
Firmware  : unknown
Logic     : unknown
MAC       : 00-0c-01-96-64-92
IP Addr   : 255.255.255.255
Subnet    : 255.255.255.255
Gateway   : 255.255.255.255
Host IP   : 255.255.255.255
Config    : ?????????????????????
```

#### 5. Load/Update the BDI Firmware Logic:

```
$ ./bdisetup -u -p/dev/ttyS0 -b57 -aGDB -tARM
Connecting to BDI loader
Erasing CPLD
Programming firmware with ./b20armgd.103
Programming CPLD with ./b20jed21.102
Erasing firmware flash ....
Erasing firmware flash passed
Programming firmware flash ....
Programming firmware flash passed.
```

#### 6. Transmit the initial configuration parameter:

```
$ ./bdisetup -c -p/dev/ttyS0 -b57 \
> -i192.168.111.202 \
> -i192.168.111.132 \
> -frm9200ek.cfg
Connecting to BDI Loader
Writing network configuration
```

Writing init list and mode  
Configuration passed

7. The above command uses the first serial port at 57,600 baud to set the BDI2000's IP address to 192.168.111.202, its default TFTP host to 192.168.111.132 and the configuration file to load to /rm9200ek.cfg
8. Check configuration and exit loader mode:

```
$ ./bdisetup -v -p/dev/ttyS0 -b57 -s  
BDI Type : BDI2000 Rev.C (SN: 92152150)  
Loader    : V1.05  
Firmware  : V1.03 bdiGDB for ARM  
Logic     : V1.02 ARM  
MAC       : 00-0c-01-96-64-92  
IP Addr   : 192.168.111.202  
Subnet    : 192.168.111.132  
Gateway   : 255.255.255.255  
Host IP   : 255.255.255.255  
Config    : rm9200ek.cfg
```

9. Locate the file reg9200.def within the installation of the BDI2000 bdiGDB support software.
10. Place the rm9200ek.cfg file in a location on the PC accessible to the TFTP server. Later you will configure the BDI2000 to load this file via TFTP as its configuration file. The BDI tool only requires the name of the file. The TFTP server when configured has to be given a directory path from where it will take the name of the file given at the BDI Setup tool i.e only rm9200ek.cfg is required and not the full path.
11. Similarly place the file reg9200.def in a location accessible to the TFTP server.
12. Open rm9200ek.cfg in an editor such as emacs or notepad and if necessary adjust the path of the reg9200.def file in the [REGS] section to match its location relative to the TFTP server root. Also comment out with a

;' the IP line in the [HOST] section, or update it to refer to the development PC.

13. Install and configure the Abatron BDI2000 in line with the bdiGDB instruction manual. Configure the BDI2000 to use the rm9200ek.cfg configuration file at the appropriate point of this process. For example, using the **bdisetup** utility:

The BDI2000 is now ready to work with the GNU Debugger. To check the communication you may establish a Telnet connection to the BDI2000.

Troubleshooting:

For more information, consult the bdiGDB User's manual.

## **Install ELDK**

ELDK is a short form of Embedded Linux Development Kit. ELDK does the same task as WinARM i.e it is also a toolchain to build GNU Project for ARM Kits but for Linux Operating System. The Embedded Linux Development Kit (ELDK) includes the GNU cross development tools, such as the compilers, binutils, gdb, etc., and a number of pre-built target tools and libraries necessary to provide some functionality on the target system.

It is provided for free with full source code, including all patches, extensions, programs and scripts used to build the tools.

It is known to work perfect in RedHat, Debian systems. As with time, there would be definitely changes and improvements to the sources, I will also mention the version I was using. I was using Debian Sarge 2.6 kernel on i386 architecture for my entire project and the ELDK version was 4.1. Since we are developing our project for ARM, we would need to traverse in the following way.

Steps of Installation:

1. <http://www.denx.de/wiki/view/DULG/ELDKAvailability>
2. Choose mirror -> 4.1 - > arm-linux-x86 -> iso -> arm-2007-01-21.iso and download it.
3. Burn the image onto a CD. You must have a ISO convertor which is usually bundled with Nero to burn it on a CD and install it.
4. Once burned do the following steps to install it on the harddisk in your Linux PC. Open a shell, for eg. Bash.

5. Type `mount -t iso9660 /dev/hdc /cdrom` OR you can use your own methods to mount the cdrom.
6. Go to the CDROM directory and type `./install -d/home/guitar/ELDK`
7. Open `.bashrc` file present in `/home/guitar` and edit it with the following two lines.

```
export  
PATH="${PATH}:/home/guitar/ELDK/usr/bin:/home/guitar/ELDK/bin":.  
export CROSS_COMPILE=arm-linux-
```

**Note :** The last entry in the PATH command is a ' .' (period). This is an important addition that you could make in case it is not present on your system. The period indicates the current directory in Linux. That means whenever you type a command, Linux would search for that program in all the directories that are in its PATH. Since there is a period in the PATH, Linux would also look in the current directory for program by the name (the directory from where you execute a command). Thus whenever you execute a program which is present in the current directory (maybe some scripts you have written on your own) you don't have to type a ' ./programname '. You can only type ' programname ' since the current directory is already in your PATH.

Now you are ready with the toolchain on Linux as well. The primary reason we shifted to Linux is that most of our programs will be based on OpenSource and especially the GNU Tool Chain. It is therefore recommended to develop the code using Linux and not a Windows wrap up for Linux for eg. Cygwin or MinGW. The other main reason ELDK will be used is that we will need from time to time compile Das U-Boot ( explained later ) and since both the Das U-Boot and ELDK is from the same company, they are already tested to work correctly. For our entire development, we would primarily use Linux.

## Install Das U-Boot

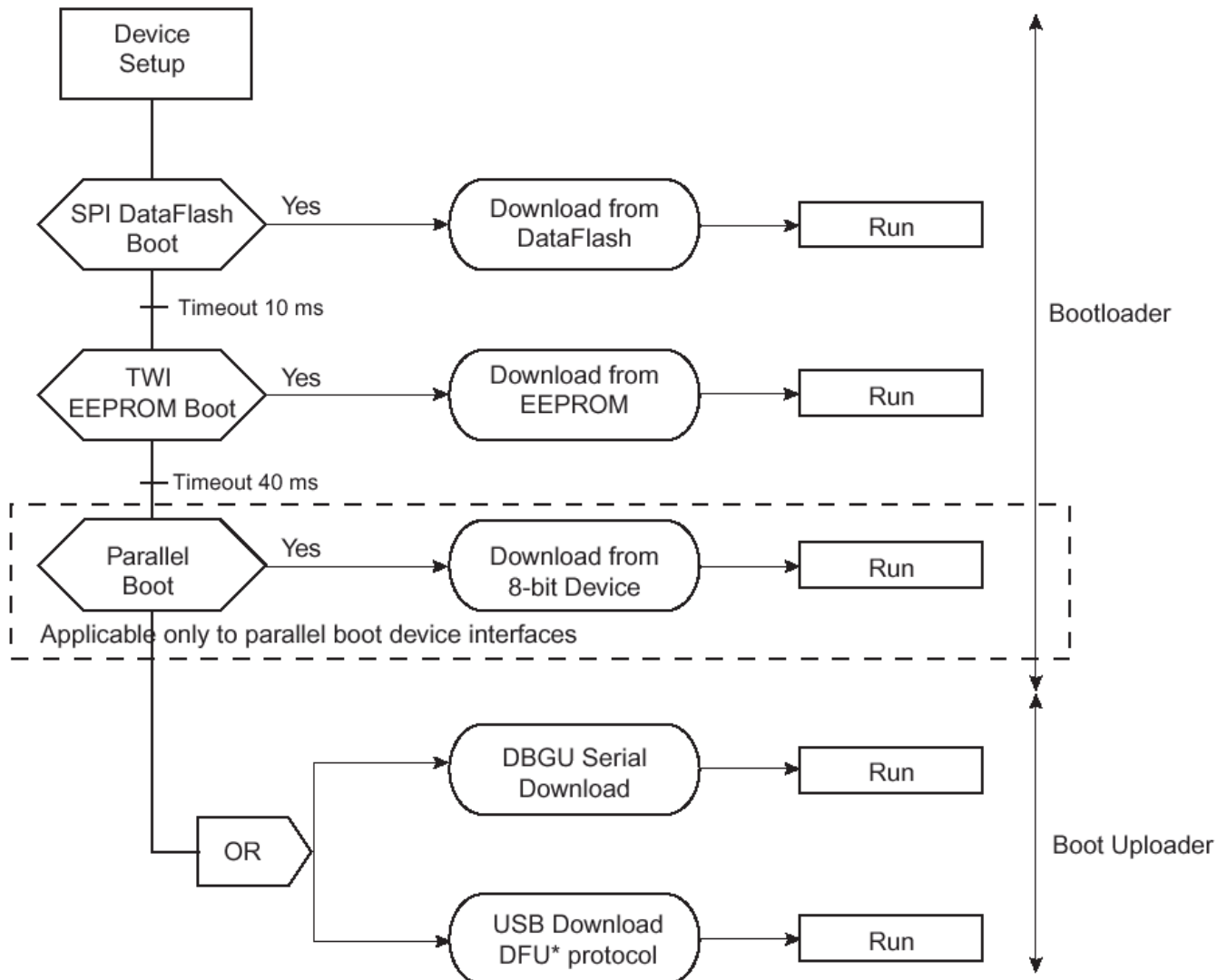
When the processor first starts up, it is suffering from amnesia; there is nothing at all in the memory to execute. Of course processor makers know this will happen, so they pre-program the processor to always look at the same place in the system BIOS ROM for the start of the BIOS boot program.

For example, the location FFFF0h just contains a "jump" instruction telling the processor where to go to find the real BIOS startup program.

This startup program in our case is referred to as BootLoader and the where to go part is called the "magic number".

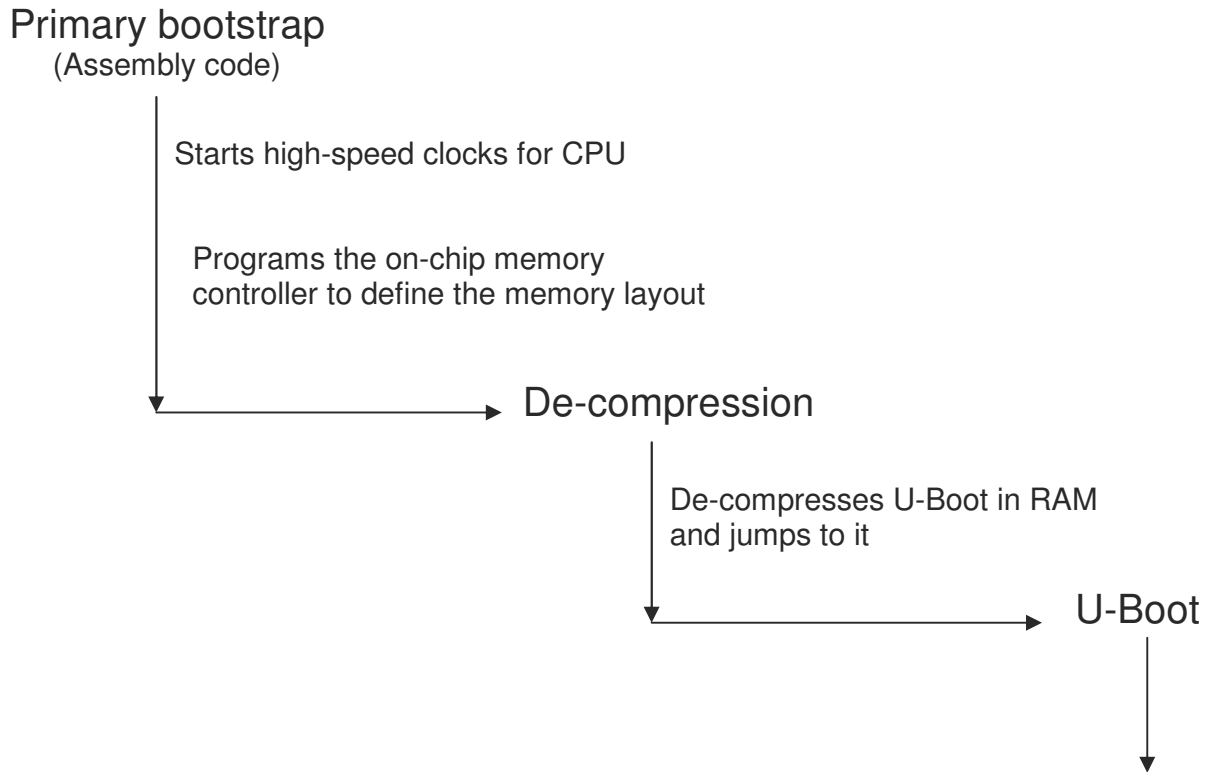
Many embedded system do not want the Bootloader to be the starting point of the execution, but actually an another tiny program which does the basic initialisation of the peripherals vis a vis SDRAM, FLASH memory, Master Clock, Peripheral Master Clock, Interrupt Disabling etc before giving control to the Bootloader. This program is called a Bootstrap.

Das U-Boot is a generic Bootloader. Bootloader is a software program which helps to load the OS or a simple while loop into the SDRAM and sets the PC to the starting location of the OS /simple while loop. The Bootloader sequence of AT91 can be shown in the following figure.



The following would be the series how the main application program would be loaded into the SDRAM in our case :

U-BOOT and the AT91RM9200-EK:



If External jumper is selected, the control is first given to the Bootstrap, which runs from Flash memory and does some basic initialisation and then

The control is given to the BootLoader sets up a correct environment for an operatin System and then

The control is given to the Application Program.

Coming back to the U-BOOT, you have to follow the below mentioned steps:

1. <http://sourceforge.net/projects/u-boot>
2. Click on Download.
3. Download the latest package. In my case it was u-boot1.1.6
4. Save it in a directory
5. Enter into the saved directory
6. Go to the file include/configs/at91rm9200dk.h
7. Look for CONFIG\_SKIP\_LOWLEVEL\_INIT. Comment the line and type CONFIG\_SKIP\_LOWLEVEL\_INIT in another line. We will use it because, the code inside CONFIG\_SKIP\_LOWLEVEL\_INIT repeats the same

process as the bootstrap from Atmel does. This breaks the proper loading of the Bootloader.

8. Type `make clobber; make clean; make distclean; make mrproper; make at91rm9200dk_config; make all` ( all in different lines ). You can skip some of the above cleaning methods but I prefer doing it in the beginning just to be on the safe side.
9. You will get a file called `u-boot.bin` as a result.
10. Type `gzip u-boot.bin` to create `u-boot.gz`. This program is ready to be loaded into the microcontroller.

If you have any problems installing it, a extensive set of troubleshooting is available. Troubleshooting:

1. <http://www.denx.de/wiki/DULG/WebHome>
2. <http://sourceforge.net/mailarchive/forum.php?forum=u-boot-users>
3. If you also want to ask questions apart from just looking for answers, <https://lists.sourceforge.net/lists/options/u-boot-users> . This would take you the registration form where you can complete your data and start asking questions.

Here we have succesfully compiled U-Boot and have got the resulting file as `u-boot.gz`. The next step would be to know how to load this program into the Flash Memory.

Now we would be again using the Windows PC as the BootUploader because Atmel is based on Xmodem Protocol and which is somehow broken in minicom. There is a solution at the following website, <http://www.koansoftware.com/en/art.php?art=68> however:

- If you need loading loader.bin into a **AT91RM9200** based board with minicom use the following instructions. The upload procedure works perfectly if you upload into the target using Xmodem protocol and HyperTerminal. If you try to use Xmodem with linux minicom you always get back 55 NAK errors and the upload fails. Use this code to solve the problem :-)  
<ftp://ftp.koansoftware.com/public/linux/sx-at91/sx-at91.c>



- build the source file with  
gcc sx-at91.c -o sx-at91

- Howto use this program with minicom/xminicom and AT91  
start minicom or xminicom  
edit Options / File transfer protocol,  
add a name (for example J) like the following example


	Name	Program	Name	U/D	FullScr	IO-Red.	Multi	
A	zmodem	/usr/bin/sz	-vv	-b	Y	U N	Y Y	
B	ymodem	/usr/bin/sb	-vv	Y	U	N	Y Y	
C	xmodem	/usr/bin/sx	-vv	Y	U	N	Y N	
D	zmodem	/usr/bin/rz	-vv	-b	-E	N D N	Y Y	
E	ymodem	/usr/bin/rb	-vv	N	D	N	Y Y	
F	xmodem	/usr/bin/rx	-vv	Y	D	N	Y N	
G	kermit	/usr/bin/kermit	-i -l %l	-s	Y	U Y	N N	
H	kermit	/usr/bin/kermit	-i -l %l	-r	N	D Y	N N	
I	ascii	/usr/bin/ascii-xfr	-dsv	Y	U	N	Y N	
J	at91	/home/koan/xmodem/sx-at91		Y	U	Y	N N	
		K		-				
L	-							

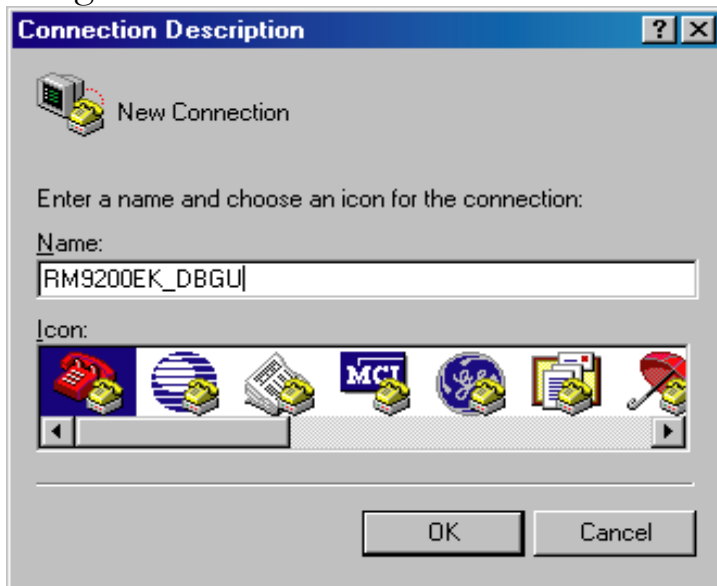
To keep it simple and fast, I would recommend using the HyperTerminal although it is one of the worst Terminals to show Data In and out. In case of uploading files via different protocols, it seems to work best.

Please follow the steps to upload the files via HyperTerminal

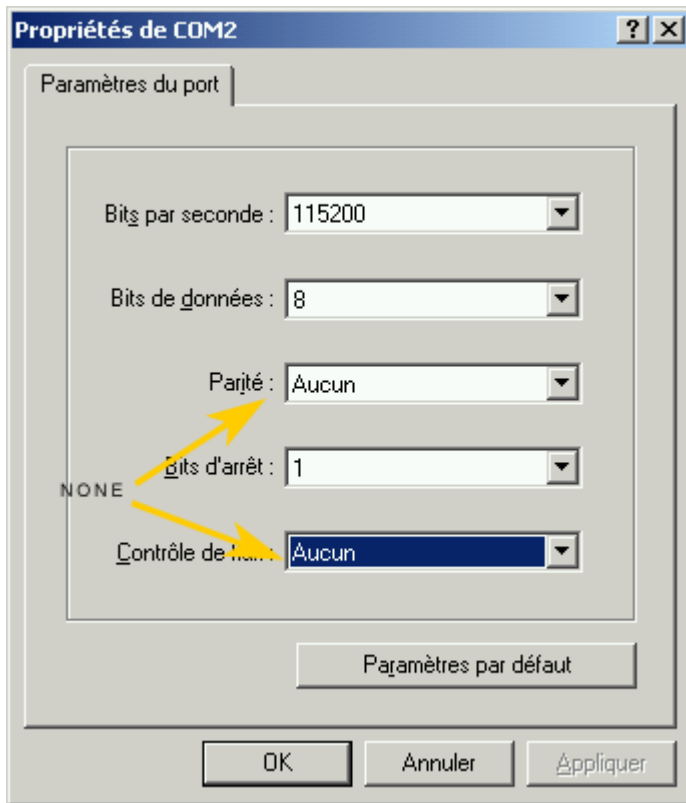
- Internal BootROM: For the AT91RM9200-EK there is a jumper called J15, that allow to boot in internal ROM (INT' position) or on the 16-bit external parallel flash (EXT' position). Now you will need two programs provided by Atmel, ( loader.bin and boot.bin ). These can be downloaded from [http://www.at91.com/Pages/products/EvaluationBoard/AT91RM9200EK/recovery/u-boot\\_recovery.html](http://www.at91.com/Pages/products/EvaluationBoard/AT91RM9200EK/recovery/u-boot_recovery.html). Or as stated earlier in the resources you would need, the files can also be found in the Folder

UbootFlashProgramming from the following location:  
[http://www.atmel.com/dyn/products/tools\\_card.asp?tool\\_id=3507](http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3507). Look for “AT91RM9200 U-Boot Flash Programming” in the above website.

- Connect the Evaluation Kit to your Host PC
  - By using the serial cable, supplied in the AT91RM9200-EK Evaluation Kit, connect the board to your PC through the J10 Serial Debug port Connector.
  - Power-up the Development Kit through the J1 connector,
  - Start the HyperTerminal application:  HyperTerminal
  - The connection can be called RM9200EK\_DBGU, for example. Valid by using the "OK" button



- From the "Connection to" window, select the COM port used and valid by using the "OK" button
- Set the serial parameters as described below:
  - Bit rate @115 kbps,
  - Data bit @8-bit,
  - Parity NONE,
  - Stop bit equal to 1,
  - Flux control NONE



The AT91RM9200-EK and your PC are connected now,

- Upload loader.bin through the Xmodem Protocol. You can choose different protocols from the Transfer Option in the Hyperterminal. Once the loader.bin is loaded you will see a prompt to upload u-boot.bin. Do not upload u-boot.gz. Upload u-boot.bin and then you would see the following picture of course according to the type of memory you are using.

```
AT91RM9200EK_DBGU - HyperTerminal (Unlicensed)
File Edit View Call Transfer Help

boot 1.0 (Aug 8 2003 - 12:29:00)

Uncompressing image...

U-Boot 1.1.1 (Jun 26 2004 - 02:38:02)

U-Boot code: 21F00000 -> 21F16DF0 BSS: -> 21F1B4AC
RAM Configuration:
Bank #0: 20000000 32 MB
Atmel: AT49BV6416 (64Mbit)
Flash: 8 MB
DataFlash: AT45DB642
Nb pages: 8192
Page Size: 1056
Size= 8650752 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C0007FFF (RO)
Area 1: C0008000 to C001FFFF (RO)
Area 2: C0020000 to C0027FFF
Area 3: C0028000 to C003FFFF
In: serial
Out: serial
Err: serial
Uboot> _

Connected 00:01:13 Auto detect 115200 8-N-1 SCROLL CAPS NUM Capture Print
```

Uptill now, the program is running in the SDRAM, which ofcourse as soon as you poweroff, will be erased and the entire process has to be repeated. To make U-Boot run everytime the board powers on we have to program the files in the Flash Memory.

### Some General Information:

**\*\*Loader.bin** needs to be updated according the custom board/software changes:

- PLLs initialisation (in main.c file)
- SMC\_CS0 settings according to the 16-bit parallel flash timings
- SDRAM initialisation

The loader.bin source code is [loader.tgz](#)

\*\* boot.bin code need to be updated according the custom board/software changes:

- PLLs initialisation
- SMC\_CS0 settings according to the 16-bit parallel flash timings
- SDRAM initialisation

The Boot.bin source code is [boot.tgz](#)

\*\*U-boot.bin (.gz) needs to be updated :

- flash ID and organisation, to allows the new flash support.
- MCK, PCK clock definition (in include/configs/at91rm9200dk.h file)

Following steps explain how to program the files to the flash memory.

How to proceed:

1)The first file to load is the boot image “boot.bin” .

To load this file by the kermit protocol, you need first to use the "send file " command of hyperterminal, select the kermit protocol and browse to the boot.bin file.

```
U-BOOT> loadb 20000000
## Ready for binary (Kermit) download ...
## Start Addr =0x20000000
```

At this step the boot image is loaded in SDRAM. The next command copies the SDRAM

(20000000) to flash(10000000).

The sectors details displayed is flash dependant.

```
U-BOOT>protect off 10000000 10005FFF
```

```
U-BOOT> erase 10000000 10005FFF
```

```
Erase Flash from 0x10000000 to 0x10005FFF...
```

```
Erasing sector 0 ... ok.
Erasing sector 1 ... ok.
Erasing sector 2 ... ok.
done.
Erased 3 sectors.
U-BOOT>cp.b 20000000 10000000 5FFF
Copy to flash... done.
U-BOOT>protect on 10000000 10005FFF
Protected 3 sectors
U-BOOT>
```

2) Once Primary Bootstrap is loaded, U-Boot gzipped image must be copied into Flash. The principle is exactly the same as for the primary bootstrap. The file to load is the uboot gzipped image “u-boot.gz” Note: Only copy the u-boot.gz file to a directory as for boot.bin. Be careful to no try to uncompressed it, it will crash your board. You can check that the u-boot.gz file you get is ~41K size before copy it into the flash..

```
U-BOOT> loadb 20000000
## Ready for binary (Kermit) download ...
## Start Addr =0x20000000
```

```
U-BOOT> protect off 10010000 1001FFFF
U-BOOT> erase 10010000 1001FFFF
Erase Flash from 0x10010000 to 0x1001FFFF...
Erasing sector 8 ... ok.
Erasing sector 9 ... ok.
done.
Erased 2 sectors.
U-BOOT>cp.b 20000000 10010000 FFFF
Copy to flash... done.
U-BOOT>protect on 10010000 1001FFFF
Protected 2 sectors
U-BOOT>
```

Reboot your board, you have upgraded your board with U-Boot.

When you reboot your board you will have this display:

After the upgrade of the flash, the display :

```
*** Warning - bad CRC, using default environment
```

```
Uboot>
```

This message means that there is no environment variables settled.

Type “bootdelay 3” on the U-Boot prompt and then “saveenv”. The next time you reboot the board there would not be any CRC error. It can also happen that there is no CRC error even for the first reboot. This error has nothing to do with the installation of U-Boot.

Congratulations: So now you have successfully installed U-BOOT on your board.

## PHASE II

### Preparing the AT91RM9200-EK board for programming and debugging

Follow the steps in this section in order to allow communication between the board and the host PC, and between the board and the JTAG device.

1. Check if the Mode LED is ON. If yes, there might be some problems and look for BDI troubleshooting.
2. Remove the Serial Cable from the Host PC to BDI2000.
3. Connect the Serial Cable from the Host PC to the DBGU Port.
4. Start a suitable terminal emulator on the host computer such as **minicom** or HyperTerminal. Set the communication parameters to 115200 baud, 8 data bits, no parity bit and 1 stop bit with no flow control.
5. Power up the AT91RM9200-EK board. You should see the message from the U-Boot and various LEDs blinking.
6. Connect the board to the BDI2000 using a 20-pin ARM/Xscale cable from the JTAG/ICE interface connector (J12) to the Target A port on the BDI2000.
7. Connect to the BDI2000's CLI interface via TCP/IP on the standard telnet port 23. The **telnet** application is suitable for this. You should see usage information followed by the prompt:

```
Core#0>
```

8. Confirm correct connection with the BDI2000 with the **reset halt** command as follows:

```
Core#0> reset halt
- TARGET: processing reset request
- TARGET: BDI asserts RESET and TRST
- TARGET: BDI removed TRST
- TARGET: Bypass check: 0x000000001 => 0x000000001
- TARGET: JTAG exists check passed
- Core#0: ID code is 0x19274013
```



```
- Core#0: BDI sets hold_rst and halt mode
- TARGET: BDI removes RESET
- Core#0: BDI sets hold_rst and halt mode again
- Core#0: BDI loads debug handler to mini IC
- Core#0: BDI clears hold_rst
- TARGET: resetting target passed
- TARGET: processing target startup ....
- TARGET: processing target startup passed
Core#0>
```

9. Locate the application image
10. Copy the application image file into a location on the host computer accessible to its TFTP server.

But when the power is switched off in BDI, the next time it jumps it looks for the tftp server and to load the .cfg file and the other file again from the tftp server. So the configuration file and definition file is not permanently stored in the BDI2000, but needs to be taken everytime the BDI2000 is restarted.

### **Using the BDI2000 to directly load the application into SDRAM**

During development the software would be changed a lot of times, so it is not required to program the application image into the flash memory. I programmed the application image only when the software was ready and it was working as expected. To simply load the program into the SDRAM, use the “**load**” command on the BDI Telnet prompt. It would look at the configuration file under the header [LOAD] for the details of the location and the name of the file to be loaded.

### **Using the BDI2000 to directly program the application into FLASH**

The BDI2000 supports direct programming of the Flash device and so that is the approach described here.

This is a three stage process. The relevant Flash blocks must first be unlocked, then erased, and finally programmed. This can be accomplished with the following steps:

1. Connect to the BDI2000 telnet port as before.
2. Use the following commands in the BDI2000 telnet session to unlock and then erase the relevant Flash blocks that will contain the application.

```
Core#0>unlock 0x50000000 0x20000 4
Unlocking flash at 0x50000000
Unlocking flash at 0x50020000
Unlocking flash at 0x50040000
Unlocking flash at 0x50060000
Unlocking flash passed
Core#0>erase 0x50000000 0x20000 4
Erasing flash at 0x50000000
Erasing flash at 0x50020000
Erasing flash at 0x50040000
Erasing flash at 0x50060000
Erasing flash passed
```

3. Program the application image into Flash with the following command. This again would look in the BDI configuration file under the header [PROG]:

```
Core#0>prog
Programming main.bin , please wait ....
Programming flash passed
Core#0>
```

4. This operation can take some time.

The application installation is now complete. This can be tested by powering off the board, disconnecting the JTAG, and then powering on the board again. To load the application automatically, type the following commands on the BDI prompt, bootm go 'the location of your application image' ( example bootm go 0x1000000 ) bootdelay 3

If it proves necessary to re-install the application, this may be achieved by repeating the above process from 1-4.

Some common BDI commands

telnet 192.168.111.202	Connects the IP address through telnet
Info	Provides the info of the current status of the Processor Running or Halted ( if halted then what caused the halt – a breakpoint or an exception )
load file	Loads the File to a memory location provided in the configuration file. For correct syntax type help.
ti	Step one instruction
md 0xFFFF0000	Displays the contents of the memory 0xFFFF0000 till the next 256 bytes
Help	Very useful command to always go and correct the syntax.
b 0xFFFF0000	Puts a hardware breakpoint at the memory location. The maximum breakpoints can be 2. Use GDB for elaborate debugging.
c 1	Clears the breakpoint number 1

## **GNU Debugger ( GDB )**

<http://csapp.cs.cmu.edu/public/docs/gdbnotes.txt>

Type the following commands once you have build the project using `-ggdb3` option. Assuming the linked file is `main.elf`

To load a newly built image and debug using GDB:

Load the program using BDI2000 into the target memory and set the PC to the starting address of the application program.

arm-linux-gdb main.elf - It will load the symbol table.

target remote bdi:2001 - This is a command for remote debugging. The debug program now rests on the starting address of the application program. By various simple GDB commands the behaviour of the software can be found out.

To restart the application.

Set the PC to the starting address through BDI.

Reload the symbol table and then type target remote bdi:2001

Following are some of the useful commands which are commonly used in debugging

-fno-schedule-insns -fno-schedule-insns2	to prevent jumping around code compiler options
set print array	
set print address	Prints the PC everytime the program breaks.
set print elements 1024	Prints 1024 elements of the array, default is 256
set print pretty	Prints the array in a pretty fashion. For example if there are 100 'A's, then it will print "100 times A" and not AAAAAA....
set print symbol-filename	Prints the name of the symbol file also with the path included.
set print union	
set verbose on	Displays every information what the GDB is doing.
set step-mode off	( no assembly )
set step-mode on	( assembly )
list size	configure the number of lines to display, default is 10

gdb elf-file	
target remote <a href="#">ip:port</a>	Remote debugging on the ip and port address.
Next	Goes “inside” the next line of Code
b symbol-name b lineno. b <a href="#">file:lineno.</a> b <a href="#">file:function</a>	Break points at the symbol name, lineno, lineno of a file, functionname of a file.
Continue	Continues to execute
Backtrace	To look at the various function calls through which the current position came from.
file symbol-file	file ( reloads the symboltable)
Detach	Detaches from remote debugging
Disconnect	Disconnects and also trashes the symbol file
monitor cmd	---- a gdb extension
print var	Prints the value of var in decimal
print/x var	Prints the value of var in hexadecimal
frame 0 frame 1	Jump to frame 0 Jump to frame 1
List list lineno. list <a href="#">file:lineno.</a> list <a href="#">file:function</a>	List the code from the current position lineno, lineno of a file, functionname of a file.

break if symbol == value , == ; != ; )= ; <	Conditional breaking
ptype	- what is the type of variable
print typecast symbol/address	
format o x d u t f a c	Print in the following format. o- octal, x – hexadecimal etc.
set myvar = 10 or set var myvar = 10	Set the value of the variable while the processor is halted.

Congratulations, you have successfully installed BDI2000 and GNU Debugger.

Till here we are all set with our environment, the next major step is to develop a program and to understand the ARM architecture. Before I go into the software logic, I would like to give an overview of the ARM architecture.

## **PHASE III**

### **History of ARM**

Courtesy: wikipedia.org

ARM Limited is a technology company headquartered in England. Founded in 1990, the company is best known for its processors, although it also designs, licenses and sells software development tools under the RealView and KEIL brands, systems and platforms, system-on-a-chip infrastructure and software. ARM is listed under its associated holding company (ARM Holdings PLC) on the London Stock Exchange (symbol: ARM) and NASDAQ (symbol: ARMHY). It is probably the best-known of the Silicon Fen companies.

The company was founded as a joint venture between Acorn Computers and Apple Computer (as Advanced RISC Machines), intended to further the development of the Acorn RISC Machine's RISC chip, which was originally used in the Acorn Archimedes and is now the processing core for many custom application-specific integrated circuits (ASICs). It has since expanded and now has offices and design centres across the world, including Sunnyvale, California, Austin, Texas, Olympia, Washington, Trondheim, Norway, Sophia Antipolis, France, Munich, Germany, Leuven, Belgium Taiwan, Shin Yokohama, Japan, China, and India.

A characteristic feature of ARM processors is their low electric power consumption, which makes them particularly suitable for use in portable devices. In fact, almost all modern mobile phones and personal digital assistants contain ARM CPUs, making them the most widely-used 32-bit microprocessor family in the world, more so than the better-known 32-bit Pentium 4 processors found in many PCs. Today ARMs account for over 75% of all 32-bit embedded CPUs.

ARM processors are used as the main CPU for most mobile phones, including those manufactured by Nokia, Sony Ericsson and Samsung; many personal digital assistants and handhelds, like the Apple iPod & iPhone [1], Nintendo Game Boy Advance and Nintendo DS, Gamepark GP32, and Gamepark Holdings GP2X; as well as many other applications, including GPS, digital cameras, digital televisions, network devices and storage.

Unlike other microprocessor corporations such as AMD, Intel, Freescale (formerly Motorola) and Renesas (formerly Hitachi and Mitsubishi), ARM only licenses its technology as intellectual property (IP), rather than manufacturing its own CPUs. Thus, there are a few dozen companies making processors based on ARM's designs. Intel, Freescale and Renesas have all licensed ARM technology. In 2005, 1.7 billion chips based on an ARM design were manufactured.

How it evolved: ( courtesy Wikipedia.org )

Family	Arch	Core	Feature	Cache (I/D)/MMU	typical MIPS @ MHz
<b>ARM1</b>	ARMv1	ARM1		None	
<b>ARM2</b>	ARMv2	ARM2	Architecture 2 added the MUL (multiply) instruction	None	4 MIPS @ 8MHz
	ARMv2a	ARM250	Integrated MEMC (MMU), Graphics and IO processor. Architecture 2a added the SWP and SWPB (swap) instructions.	None, MEMC1a	7 MIPS @ 12MHz
<b>ARM3</b>	ARMv2a	ARM2a	First use of a processor cache on the ARM.	4K unified	12 MIPS @ 25MHz
<b>ARM6</b>	ARMv3	ARM610	v3 architecture first to support addressing 32bits of memory (as opposed to 26bits)	4K unified	28 MIPS @ 33MHz
<b>ARM7TDMI</b>	ARMv4T	ARM7TDMI(-S)	3-stage pipeline	none	15 MIPS @ 16.8 MHz
		ARM710T		8KB MMU	unified, 36 MIPS @ 40 MHz
		ARM720T		8KB MMU	unified, 60 MIPS @ 59.8 MHz
		ARM740T		MPU	
	ARMv5TEJ	ARM7EJ-S	Jazelle DBX	none	
<b>ARM9TDMI</b>	ARMv4T	ARM9TDMI	5-stage pipeline	none	
		ARM920T		16KB/16KB, MMU	200 MIPS @ 180 MHz
		ARM922T		8KB/8KB, MMU	
		ARM940T		4KB/4KB, MPU	



<b>ARM9E</b>	ARMv5TE	ARM946E-S		variable, tightly coupled memories, MPU
		ARM966E-S		no cache, TCMs
		ARM968E-S		no cache, TCMs
	ARMv5TEJ	ARM926EJ-S	Jazelle DBX	variable, TCMs, 220 MIPS @ 200 MHz
<b>ARM10E</b>	ARMv5TE	ARM996HS	Clockless processor	no caches, TCMs, MPU
	ARMv5TE	ARM1020E	(VFP), 6-stage pipeline	32KB/32KB, MMU
		ARM1022E	(VFP)	16KB/16KB, MMU
	ARMv5TEJ	ARM1026EJ-S	Jazelle DBX	variable, MMU or MPU
<b>XScale</b>	ARMv5TE	80200/IOP310/IOP315 I/O Processor		
		80219		400/600MHz
		IOP321		600 BogoMips @ 600 MHz
		IOP33x		
		IOP34x	1-2 core, RAID Acceleration	32K/32K L1, 512K L2, MMU
		PXA210/PXA250	Applications processor, 7-stage pipeline	
		PXA255		32KB/32KB, MMU 400 BogoMips @ 400 MHz
		PXA26x		up to 400 MHz
		PXA27x		800 MIPS @ 624 MHz
		PXA800(E)F		
		Monahans		1000 MIPS @ 1.25 GHz
		PXA900		
		IXC1100	Control Plane Processor	
		IXP2400/IXP2800		
		IXP2850		
		IXP2325/IXP2350		
		IXP42x		
		IXP460/IXP465		
<b>ARM11</b>	ARMv6	ARM1136J(F)-S	SIMD, Jazelle DBX, (VFP), 8-stage pipeline	?? @ 532-665MHz (i.MX31 SoC)
		ARMv6T2 ARM1156T2(F)-S	SIMD, Thumb-2, variable, MPU	

			(VFP), 9-stage pipeline
	ARMv6KZ ARM1176JZ(F)-S		SIMD, Jazelle variable, DBX, (VFP) MMU+TrustZone
	ARMv6K ARM11 MPCore		1-4 core SMP, SIMD, Jazelle variable, MMU DBX, (VFP)
<b>Cortex</b>	ARMv7-A Cortex-A8		Application profile, VFP, NEON, Jazelle variable (L1+L2), RCT, Thumb-2, 13-stage pipeline up to 2000 (2.0 DMIPS/MHz in speed from 600 MHz to greater than 1 GHz)
	ARMv7-R Cortex-R4(F)		Embedded profile, (FPU) variable cache, MMU optional 600 DMIPS
	ARMv7-M Cortex-M3		Microcontroller profile no cache, (MPU) 120 DMIPS @ 100MHz

## What a SW developer needs to know about the Architecture

These are, arguably, the most useful instructions available. It is all very well being able to do stuff with the registers, but if you cannot load and store them to the main memory, then... <grin>

Instruction Set:

Courtsey : <http://www.heyrick.co.uk/assembler/qfinder.html>

### ***Single Data Transfer***

The single data transfer instructions (STR and LDR) are used to load and store single bytes or words of data from/to main memory. The addressing is very flexible.

First, we'll look at the instruction:

```
LDR R0, address
STR R0, address
LDRB R0, address
STRB R0, address
```

These instructions load and store the value of R0 to the specified address. If 'B' is also specified, as in the latter two instructions, then only a single byte is loaded or saved. The three unused bytes in the word are **zeroed** upon loading.

The address can be a simple value, or an offset, or a shifted offset. Write-back may be performed (to remove the need for adding/subtracting).

```
STR    R0, [Rbase]      Store R0 at Rbase.
STR    R0, [Rbase, Rindex] Store R0 at Rbase + Rindex.
STR    R0, [Rbase, #index] Store R0 at Rbase + index.
                        Index is an immediate value.
                        STR R0, [R1, #16] would load R0
                        from R1+16.
STR    R0, [Rbase, Rindex]! Store R0 at Rbase + Rindex, &
                        write back new address to Rbase.
STR    R0, [Rbase, #index]! Store R0 at Rbase + index, &
                        write back new address to Rbase.
STR    R0, [Rbase], Rindex Store R0 at Rbase, & write back
                        Rbase + Rindex to Rbase.
STR    R0, [Rbase, Rindex, LSL #2] will store R0 at the address
                        Rbase + (Rindex * 4)
STR    R0, place        Will generate a PC-relative offset
                        to 'place', and store R0 there.
```

You can, of course, use conditional execution on any of these instructions. Note, however, that the conditional flag comes before the byte flag, so if you wish to load a byte when the result is equal, the instruction would be LDREQB Rx, address (not LDRBEQ...).

If you specify pre-indexed addressing (where the base and index are both within square brackets), the write-back is controlled by the presence or absence of the '!'. The fourth and fifth examples above reflect this. Using this, you can automatically move forward or backward in memory. A string print routine could then become:

```
.loop
    LDRB  R0, [R1, #1]!
    SWI   "OS_WriteC"
```

```

    CMP    R0, #0
    BNE    loop
instead of:
.loop
    LDRB   R0, [R1]
    SWI    "OS_WriteC"
    ADD    R1, R1, #1
    CMP    R0, #0
    BNE    loop

```

The use of '!' is invalid for post-indexed addressing (where the index is outside of the square brackets, as in example six above) as write-back is implied.

As you can see, the offset may be shifted. Additionally, the index offset may be subtracted from the base. In this case, you might use code such as:

```

    LDRB   R0, [R1, #-1]

```

You cannot modify the PSR with a load or store instruction, though you can store or load the PC. In order to load a stored 'state' and correctly restore it, use:

```

    LDR    R0, [Rbase]
    MOVS   R15, R0

```

The MOVS will cause the PSR bits to be updated, provided that you are privileged.  
**Using MOVS with PC is not 32-bit compliant.**

According to the ARM assembler manual:  
*A byte load (LDRB) expects the data on bits 0 to 7 if the supplied address is on a word boundary, on bits 8 to 15 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeroes.*  
*A byte store (STRB) repeats the bottom 8 bits of the source register four times across the data bus. The external memory system should activate the appropriate byte subsystem to store the data.*  
*A word load (LDR) or word store (STR) should generate a word aligned address. Using a non-word-aligned addresses has non-obvious and unspecified results.*

The only thing of real note here is that you cannot use LDR to load a word from a non-aligned address.

## ***Multiple Data Transfer***

The multiple data transfer instructions (LDM and STM) are used to load and store multiple words of data from/to main memory.

The main use of LDM/STM is to dump registers that need to be preserved onto the stack. We've all seen **STMFD R13!, {R0-R12, R14}**.

The instruction is:

xxM type cond base write-back, {register list}

'xx' is LD to load, or ST to store.

'type' is:

Stack    Other

LDMED	LDMIB	Pre-incremental load
LDMFD	LDMIA	Post-incremental load
LDMEA	LDMDB	Pre-decremental load
LDMFA	LDMDA	Post-decremental load

STMFA	STMIB	Pre-incremental store
STMEA	STMIA	Post-incremental store
STMFD	STMDB	Pre-decremental store
STMED	STMDA	Post-decremental store

The assembler takes care of how to map the mnemonics. Note that ED is not IB; it is only the same for a pre-decremental load. When storing, ED is post-decrement.

FD, ED, FA, and EA refer to a Full or Empty stack which is either Ascending or Descending.

A full stack is where the stack pointer points to the last data item written, and empty stack is where the stack pointer points to the first free slot. A descending stack grows downwards in memory (ie, from the end of application space down) and an ascending stack is one which grows upwards in memory.

The other forms simply describe the behaviour of the instruction, and mean Increment After, Increment Before, Decrement After, Decrement Before.

RISC OS, by tradition, uses a Fully Descending stack. When writing in APCS assembler, it is common to set your stack pointer to the end of application space and then use a Full Descending stack. If you are working with a high level language (either BASIC or C), then you don't get a choice. The stack pointer (traditionally R13) points to the end of a fully descending stack. You must continue this format, or create and manage your own stack (if you're the sort of die-hard person that would do something like this!).

'base' is the register containing the address to begin with. Traditionally under RISC OS, the stack pointer is R13, though you can use any available register except R15.

If you would like the stack pointer to be updated with the new register contents, simply set the write-back bit by following the stack pointer register with an '!'.

The register list is given in {curly brackets}. It doesn't matter what order you specify the registers in, they are stored from lowest to highest. As a single bit determines whether or not a register is saved, there is no point to trying to specify it twice.

A side effect of this is that code such as:

```
STMFD R13!, {R0, R1}  
LDMFD R13!, {R1, R0}
```

will *not* swap the contents of two registers. A useful shorthand has been provided. To encompass a range of registers, simply say the first and the last, and put a dash between them. For example R0-R3 is identical to R0, R1, R2, R3, only tidier and saner...

When R15 is stored to memory, the PSR bits are also saved. When R15 is reloaded, the PSR bits are NOT restored unless you request it. The method of requesting is to follow the register list with a '^'.

```
STMFD R13!, {R0-R12, R14}  
...  
LDMFD R13!, {R0-R12, PC}
```

This saves all registers, does some stuff, then reloads all registers. PC is loaded from R14 which was probably set by a BL instruction or some-such. The PSR flags are untouched.

```
STMFD R13!, {R0-R12, R14}  
...  
LDMFD R13!, {R0-R12, PC}^
```

## **Exception Handling in ARM**

We discuss exceptions and interrupt handling techniques in ARM processors and see how the ARM architecture works in this area to know how are these techniques suitable for embedded systems to achieve the time constraints and safety requirements. Exception and interrupt handling is a critical issue since it affects directly the speed of the system and how fast does the system respond to external events and how does it deal with more than one external event at the same time by assigning priorities to these events.

### **Table of contents**

Abstract

Table of contents

List of Figures

Abbreviations

1 Introduction

1.1 ARM modes of operation

1.2 ARM Register set

2 ARM Exceptions

2.1 Vector Table

2.2 Exception priorities

2.3 Link Register Offset

2.4 Entering and exiting an exception handler

3 Interrupts

3.1 How are interrupts assigned?

3.2 Interrupt Latency

3.3 IRQ and FIQ exceptions

3.4 Interrupt stack

- 4 Interrupt handling schemes
  - 4.1 Non-nested interrupt handling
    - 4.1.1 Non-nested interrupt handling summary
  - 4.2 Nested interrupt handling
    - 4.2.1 Nested interrupt handling summary:
  - 4.3 Prioritized simple interrupt handling
    - 4.3.1 Prioritized simple interrupt handling summary
  - 4.4 Other schemes

## 5 Final remarks

Which interrupt handling scheme to use?

### **Abbreviations**

ISR Interrupt Service Routine

SWI Software Interrupt

IRQ Interrupt Request

FIQ Fast Interrupt Request

ARM Advanced RISC Machines

RISC Reduced Instruction Set Computers

SVC Supervisor

CPSR Current Program Status Register

SPSR Saved Program Status Register

LDR Load Register

STR Store Register

DMA Direct Memory Access

### **1 Introduction**

Exceptions are so important in embedded systems, without exception the development of systems would be a very complex task. With exceptions we can detect bugs in the application, errors in memory access and finally debug it by placing breakpoints and building the program with debugging information.

Interrupts which are kinds of exceptions are essential in embedded systems. It enables the system to deal with external events by receiving interrupt signals telling the CPU that there is something to be done instead of the alternative way of doing the same operation by the polling mechanism which wastes the CPU time in looping forever checking some flags to know that the event occurred.

In order for the processors to execute the correct ISR when an interrupt arrives, there must be a vector table known as Interrupt vector table. It is actually just an array of pointers to functions, located at some known memory address.



Due to the fact that systems are going more complex day after day, we have nowadays

systems with more than one interrupt source. That is why an interrupt handling scheme is needed to define how different cases will be handled. We may need priorities to be assigned to different interrupts and in some other cases we may need nested handling capabilities.

We introduce the ARM processor itself to see its different modes of operation and then we have an overview of the register set. This is because dealing with interrupts and exceptions causes the ARM core to switch between these modes and copy some of the registers into other registers to save the core state before switching to the new mode. In the next chapter we introduce exceptions and see how the ARM processor handles exceptions. In the third chapter we define interrupts and discuss mechanisms of interrupt handling on ARM. In the fourth chapter we provide a set of standard interrupt handling schemes. And finally some remarks regarding these schemes and which one is suitable to which application.

The main source of information provided in this paper is mainly the book “ARM System Developer’s Guide” [1].

### **1.1 ARM modes of operation**

The ARM processor internally has 7 different modes of operation, they are as follows; User mode: It is used for normal program execution state,

FIQ mode: This mode is used for interrupts requiring fast response and low latency like for example data transfer with DMA,

IRQ mode: This mode is used for general interrupt services,

Supervisor mode: This mode is used when operating system support is needed where it works as protected mode,

Abort mode: selected when data or instruction fetch is aborted,

system mode: Operating system privilege mode for users and

undefined mode: When undefined instruction is fetched. The following table summarizes the 7 modes:

#### **Processor Mode Description**

User (**usr**) Normal program execution mode

FIQ (**fiq**) Fast data processing mode

IRQ (**irq**) For general purpose interrupts

Supervisor (**svc**) A protected mode for the operating system

Abort (**abt**) When data or instruction fetch is aborted

Undefined (**und**) For undefined instructions

System (**sys**) Operating system privileged mode

## 1.2 ARM Register set

The ARM processor has twenty seven registers, some of which have conditions applied, so you only get to use sixteen at any one time...

- Register 0 to register 7 are general purpose registers and can be used for *ANY* purpose.  
Unlike 80x86 processors which require certain registers to be used for stack access, or the 6502 which places the result of mathematical calculations in the Accumulator, the ARM processor is highly flexible in its register use.
- Register 8 to register 12 are general purpose registers, but they have shadow registers which come into use when you switch to FIQ mode.
- Register 13 is typically the OS stack pointer, but can be used as a general purpose register. This is an operating system issue, not a processor issue, so if you don't use the stack you can corrupt this freely within your own code as long as you restore it afterwards. Each of the processor modes shadow this register.
- Register 14 is dedicated to holding the address of the return point to make writing subroutines easier. When you branch with link (BL) the return address is stored in R14. Likewise when the program is first run, the exit address is stored in R14. All instances of R14 must be preserved in other registers (not really efficient) or in a stack. This register is shadowed across all of the processor modes. This register can be used as a general purpose register once the link address has been preserved.
- Register 15 is the program counter. It holds the status of the processor as well as a twenty-six bit number which is the address of the program currently being used.

Register structure in ARM depends on the mode of operation. For example we have 16 (32-bit) registers named from R0 to R15 in ARM mode (**usr**).

Registers R0 to R12 are general purpose registers, R13 is stack pointer (**SP**), R14 is

subroutine link register and R15 is program counter (**PC**).

R16 is the current program status register (**CPSR**) this register is shared between all modes and it is used by the ARM core all the time and it plays a main role in the process of switching between modes.

In other modes some of these 16 registers are visible and can be accessed and some others are not visible and can't be accessed. Also some registers are available with the same name but as another physical register in memory which is called (**banked**), existence of such banked registers decreases the effort needed when context switching is required since the new mode has its own physical register space and no need to store the old mode's register values.

So in ARM7 we have a total of 37 physical registers and the following table shows the

ARM7 register set.

Type	User Mode	SVC Mode	IRQ Mode	FIQ Mode
General	<b>R0</b>	R0	R0	R0
General	<b>R1</b>	R1	R1	R1
General	<b>R2</b>	R2	R2	R2
General	<b>R3</b>	R3	R3	R3
General	<b>R4</b>	R4	R4	R4
General	<b>R5</b>	R5	R5	R5
General	<b>R6</b>	R6	R6	R6
General	<b>R7</b>	R7	R7	R7
Banked	<b>R8</b>	R8	R8	<b>R8_fiq</b>
Banked	<b>R9</b>	R9	R9	<b>R9_fiq</b>
Banked	<b>R10</b>	R10	R10	<b>R10_fiq</b>
Banked-fp	<b>R11</b>	R11	R11	<b>R11_fiq</b>
Banked-ip	<b>R12</b>	R12	R12	<b>R12_fiq</b>
Banked-sp	<b>R13</b>	<b>R13_svc</b>	<b>R13_irq</b>	<b>R13_fiq</b>
Banked-lr	<b>R14</b>	<b>R14_svc</b>	<b>R14_irq</b>	<b>R14_fiq</b>
General-pc	<b>R15</b>	R15	R15	R15

The Program Counter is built up as follows:

Bit	31	30	29	28	27	26	-----	1	0
	N	Z	C	V	I	F	Program Counter	S1	S0

### Figure 1: Register Organization in ARM [5]

As we can see the banked registers are marked with the gray colour. We can notice that in the FIQ mode there are more banked registers, this is to speed up the context switching since there will be no need to store many registers when switching to the FIQ mode. We may need only to store the values of registers r0 to r7 if the FIQ handler needs to use those registers, but registers r8\_fiq to r14\_fiq are specific registers for the FIQ mode and can't be accessed by any other mode (they become invisible in other modes).

## 2 ARM Exceptions

An exception is any condition that needs to halt normal execution of the instructions [1]. As an example of exceptions the state of resetting ARM core, the failure of fetching instructions or memory access, when an external interrupt is raised or when a software interrupt instruction is executed. There is always software associated with each exception, this software is called exception handler. Each of the ARM exceptions causes the ARM core to enter a certain mode automatically also we can switch between different modes manually by modifying the CPSR register. The following table summarises different exceptions and the associated mode of operation on ARM processor.

### Exception Mode

Fast Interrupt Request FIQ

Interrupt Request IRQ

SWI and RESET SVC

Instruction fetch or memory access failure ABT

Undefined Instruction UND

**More banked registers, so context switching is faster**

### 2.1 Vector Table

It is a table of instructions that the ARM core branches to when an exception is raised. These instructions are placed in a specific part in memory and its address is related to the exception type. It always contains a branching instruction in one of the following forms:

- **B <Add>**

This instruction is used to make branching to the memory location with address “Add”

relative to the current location of the pc.

- **LDR pc, [pc, #offset]**

This instruction is used to load in the program counter register its old value + an offset

value equal to “offset”.

- **LDR pc, [pc, #-0xff0]**

This instruction is used only when an interrupt controller is available, to load a specific ISR address from the vector table. The vector interrupt controller (VIC) is placed at memory address 0xfffff000 this is the base address of the VIC. The ISR address is always located at 0xfffff030.

- **MOV pc, #immediate**

Load in the program counter the value “immediate”.

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abor: (prefetch)	Abort
0x00000010	Abor: (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

**Figure 2 An exact vector table with the branching instruction[5]**

We can notice in the vector table that the FIQ exception handler is placed at the end of the vector table, so no need for a branching instruction there; we can place the exception handler directly there so handling begins faster by eliminating the time of branching. At these addresses we find a jump instruction like that:

**ldr pc, [pc, #\_IRQ\_handler\_offset]**

## 2.2 Exception priorities

Since exceptions can occur simultaneously so we may have more than one exception raised at the same time, the processor has to have a priority for each

exception so it can decide which of the currently raised exceptions is more important. The following table shows various exceptions that occur on the ARM and their associated priorities.

Exception	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Pre-fetch	5	1	-
SWI	6	1	-
Undefined instruction	6	1	-

We should notice the difference between prioritization of exceptions (when multiple exceptions are valid at the same time), and the actual exception handler code. Exception

handlers are themselves liable to interruption by exceptions, and so we have the two bits

called F-bit and I-bit. The F-bit determines if exceptions can be ranked at all or not, when it is

1 so no other exceptions can be raised. And the I-bit is the same but for IRQ exceptions. The Undefined Instruction and SWI cannot occur at the same time because they are both caused by an instruction entering the execution stage of the ARM instruction pipeline, so are mutually exclusive and thus they have the same priority.

### 2.3 Link Register Offset

The link register is used to return the *PC* (after handling the exception) to the appropriate place in the interrupted task. It is modified based on the current *PC* value and the type of exception occurred. For some cases it should point to the next instruction after the exception handling is done and in some other cases it should return to one or 2 previous instructions to repeat those instructions after the exception handling is done. For example, in the case of IRQ exception, the link

register is pointing initially to the last executed instruction + 8, so after the exception is handled we should return to the old **PC** value + 4 (next instruction) which equals to the old **LR** value – 4. Another example is the data abort exception, in this case when the exception is handled, the **PC** should point to the same instruction again to retry accessing the same memory location again.

## 2.4 Entering and exiting an exception handler

Here are the steps that the ARM processor does to handle an exception [5]:

- Preserve the address of the next instruction.
- Copy **CPSR** to the appropriate **SPSR**, which is one of the banked registers for each mode of operation.
- Force the **CPSR** mode bits to a value depending on the raised exception.
- Force the **PC** to fetch the next instruction from the exception vector table.
- Now the handler is running in the mode associated with the raised exception.
- When handler is done, the **CPSR** is restored from the saved **SPSR**.
- **PC** is updated with the value of (**LR** – offset) and the offset value depends on the type of the exception.

And when deciding to leave the exception handler, the following steps occurs:

- Move the Link Register **LR** (minus an offset) to the **PC**.
- Copy **SPSR** back to **CPSR**, this will automatically changes the mode back to the previous one.
- Clear the interrupt disable flags (if they were set).

## 3 Interrupts

There are two types of interrupts available on ARM processor. The first type is the interrupt caused by external events from hardware peripherals and the second type is the SWI instruction.

The ARM core has only one FIQ pin, that is why an external interrupt controller is always used so that the system can have more than one interrupt source which are prioritized with this interrupt controller and then the FIQ interrupt is raised and the handler identifies which of the external interrupts was raised and handle it.

### 3.1 How are interrupts assigned?

It is up to the system designer who can decide which hardware peripheral can produce which interrupt request. By using an interrupt controller we can connect multiple external interrupts to one of the ARM interrupt requests and distinguish between them.

There is a standard design for assigning interrupts adopted by system designers:

- SWIs are normally used to call privileged operating system routines.
- IRQs are normally assigned to general purpose interrupts like periodic timers.
- FIQ is reserved for one single interrupt source that requires fast response time, like

DMA or any time critical task that requires fast response.

### 3.2 Interrupt Latency

It is the interval of time between from an external interrupt signal being raised to the first fetch of an instruction of the ISR of the raised interrupt signal.

System architects must balance between two things, first is to handle multiple interrupts

simultaneously, second is to minimize the interrupt latency.

Minimization of the interrupt latency is achieved by software handlers by two main methods, the first one is to allow nested interrupt handling so the system can respond to new interrupts during handling an older interrupt. This is achieved by enabling interrupts immediately after the interrupt source has been serviced but before finishing the interrupt handling. The second one is the possibility to give priorities to different interrupt sources; this is achieved by programming the interrupt controller to ignore interrupts of the same or lower priority than the interrupt being handled if there is one.

### 3.3 IRQ and FIQ exceptions

Both exceptions occur when a specific interrupt mask is cleared in the *CPSR*. The ARM

processor will continue executing the current instruction in the pipeline before handling the interrupt. The processor hardware go through the following standard procedure:

- The processor changes to a specific mode depending on the received interrupt.
- The previous mode **CPSR** is saved in **SPSR** of the new mode.
- The **PC** is saved in the **LR** of the new mode.
- Interrupts are disabled, either IRQ or both IRQ and FIQ.
- The processor branches to a specific entry in the vector table.

Enabling/Disabling FIQ and IRQ exceptions is done on three steps; at first loading the

contents of **CPSR** then setting/clearing the mask bit required then copy the updated contents back to the **CPSR**.

### 3.4 Interrupt stack



Exception handling uses stacks extensively because each exception has a specific mode of operation, so switching between modes occurs and saving the previous mode data is required before switching so that the core can switch back to its old state successfully. Each mode has a dedicated register containing a stack pointer. The design of these stacks depends on some factors like operating system requirements for stack design and target hardware physical limits on size and position in memory. Most of ARM based systems has the stack designed such that the top of it is located at high memory address. A good stack design tries to avoid stack overflow because this causes instability in embedded systems.

## 4 Interrupt handling schemes

Here we introduce some interrupt handling schemes with some notes on each scheme about its advantages and disadvantages.

### 4.1 Non-nested interrupt handling

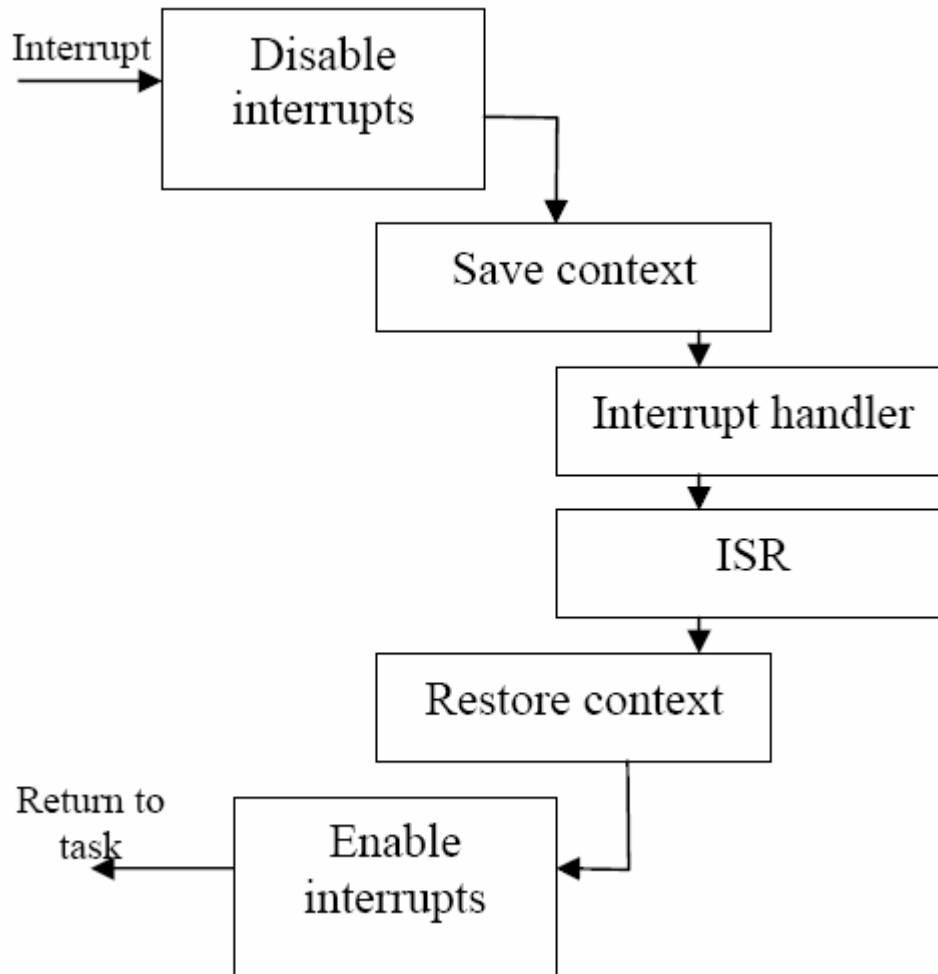
This is the simplest interrupt handler. Interrupts are disabled until control is returned back to the interrupted task. So only one interrupt can be served at a time and that is why this scheme is not suitable for complex embedded systems which most probably have more than one interrupt source and require concurrent handling. Figure 5 shows the steps taken to handle an interrupt:

Initially interrupts are disabled, When IRQ exception is raised and the ARM processor

disables further IRQ exceptions from occurring. The mode is changed to the new mode

depending on the raised exception. The register **CPSR** is copied to the **SPSR** of the new

mode. Then the **PC** is set to the correct entry in the vector table and the instruction there will direct the **PC** to the appropriate handler. Then the context of the current task is saved a subset of the current mode non banked register. Then the interrupt handler executes some code to identify the interrupt source and decide which ISR will be called. Then the appropriate ISR is called. And finally the context of the interrupted task is restored, interrupts are enabled again and the control is returned to the interrupted task.



**Figure 4 Simple non nested interrupt handlers**

#### **4.1.1 Non-nested interrupt handling summery:**

- Handle and service individual interrupts sequentially.
- High interrupt latency.
- Relatively easy to implement and debug.
- Not suitable for complex embedded systems.

#### **4.2 Nested interrupt handling**

In this handling scheme handling more than one interrupt at a time is possible. This is

achieved by re-enabling interrupts before the handler has fully served the current interrupt.

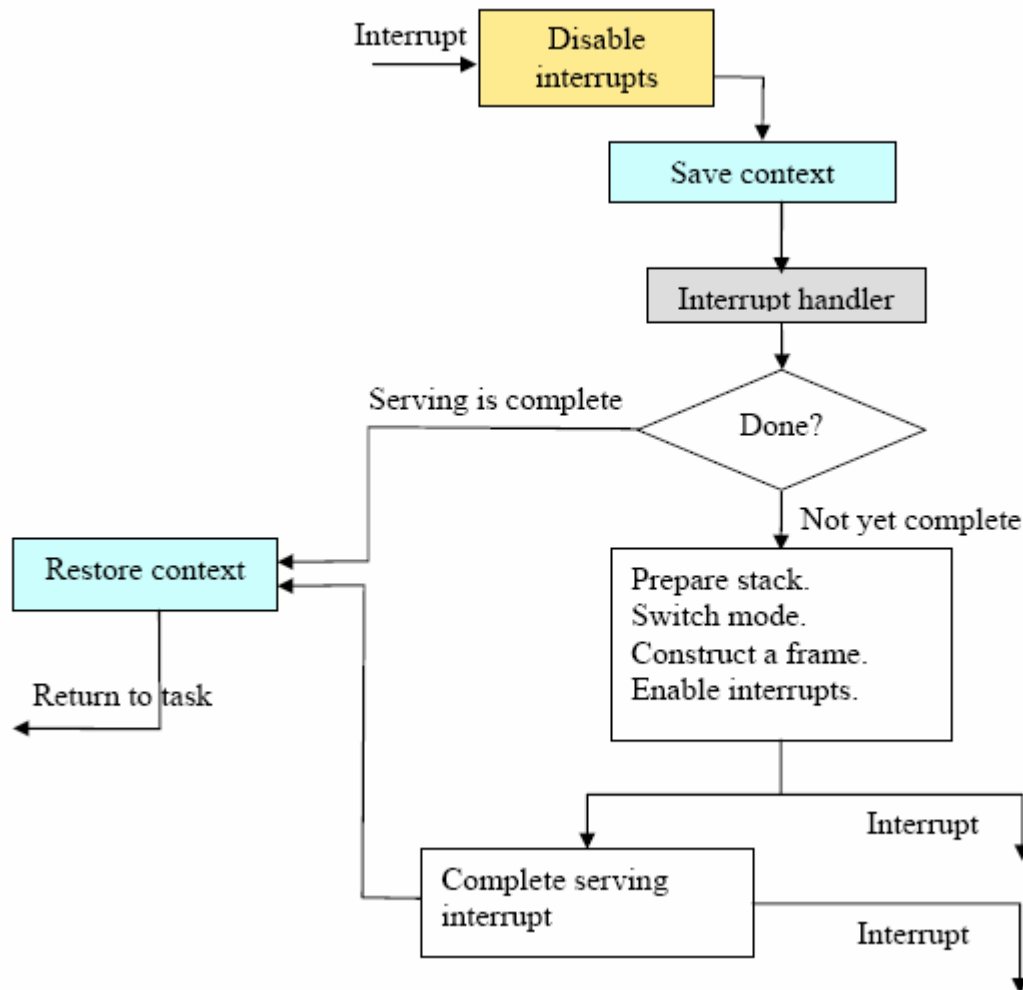
This feature increases the complexity of the system but improves the latency. The scheme should be designed carefully to protect the context saving and restoration

from being interrupted. The designer should balance between efficiency and safety by using defensive coding style that assumes problems will occur.

The goal of nested handling is to respond to interrupts quickly and to execute periodic tasks without any delays. Re-enabling interrupts requires switching out of the IRQ mode to user mode to protect link register from being corrupted. Also performing context switch requires emptying the IRQ stack because the handler will not perform switching if there is data on the IRQ stack, so all registers saved on the IRQ stack have to be transferred to task stack. The part of the task stack used in this process is called **stack frame**.

The main disadvantage of this interrupt handling scheme is that it doesn't differ between

interrupts by priorities, so lower priority interrupt can block higher priority interrupts.



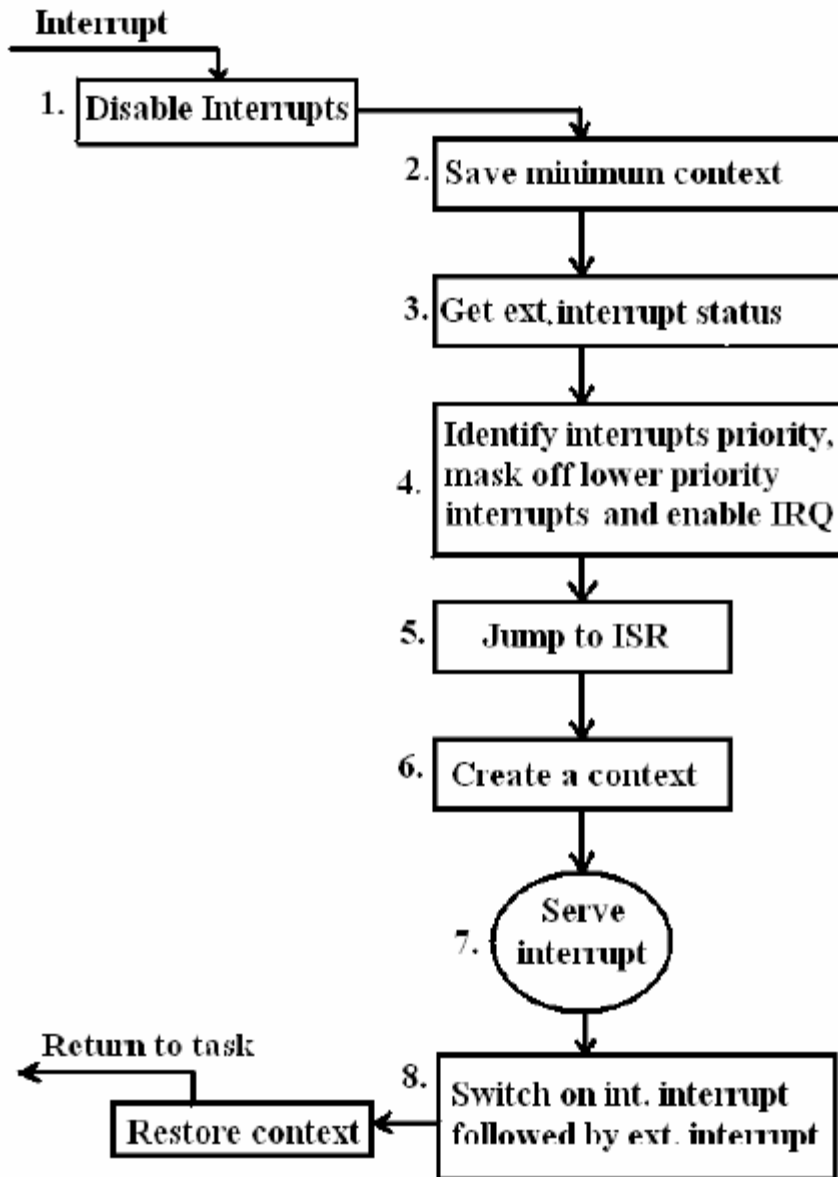
**Figure 5 Nested Interrupt Handling**

#### 4.2.1 Nested interrupt handling summery:

- Handle multiple interrupts without a priority assignment.
- Medium or high interrupt latency.
- Enable interrupts before the servicing of an individual interrupt is complete.
- No prioritization, so low priority interrupts can block higher priority interrupts.

#### **4.3 Prioritized simple interrupt handling**

In this scheme the handler will associate a priority level with a particular interrupt source. A higher priority interrupt will take precedence over a lower priority interrupt. Handling prioritization can be done by means of software or hardware. In case of hardware prioritization the handler is simpler to design because the interrupt controller will give the interrupt signal of the highest priority interrupt requiring service. But on the other side the system needs more initialization code at start-up since priority level tables have to be constructed before the system being switched on.



**Figure 6 Priority Interrupt Handler [1]**

When an interrupt signal is raised, a fixed amount of comparisons with the available set of priority levels is done, so the interrupt latency is deterministic but at the same point this could be considered a disadvantage because both high and low priority interrupts take the same amount of time.

#### **4.3.1 Prioritized simple interrupt handling summery:**

- Handle multiple interrupts with a priority assignment mechanism.
- Low interrupt latency.
- Deterministic interrupt latency.
- Time taken to get to a low priority ISR is the same as for high priority ISR.

## 4.4 Other schemes

There are some other schemes for handling interrupts, designers have to choose the suitable one depending on the system being designed.

### 4.4.1 Re-entrant interrupt handler:

The basic difference between this scheme and the nested interrupt handling is that interrupts are re-enabled early on the re-entrant interrupt handler which can reduce interrupt latency. The interrupt of the source is disabled before re-enabling interrupts to protect the system from getting infinite interrupt sequence. This is done by using a mask in the interrupt controller.

By using this mask, prioritizing interrupts is possible but this handler is more complex.

### 4.4.2 Prioritized standard interrupt handler:

It is the alternative approach of prioritized simple interrupt handler; it has the advantage of low interrupt latency for higher priority interrupts than the lower priority ones. But the disadvantage now is that the interrupt latency is non-deterministic.

### 4.4.3 Prioritized grouped interrupt handler:

This handler is designed to handle large amount of interrupts by grouping interrupts together and forming a subset which can have a priority level. This way of grouping reduces the complexity of the handler since it doesn't scan through every interrupt to determine the priority. If the prioritized grouped interrupt handler is well designed, it will improve the overall system response times dramatically, on the other hand if it is badly designed such that interrupts are not well grouped, then some important interrupts will be dealt as low priority interrupts and vice versa. The most complex and possibly critical part of such scheme is the decision on which interrupts should be grouped together in one subset.

## 5 Final remarks

### Which interrupt handling scheme to use?

We can't decide on one interrupt handling scheme to be used as a standard in all systems, it depends on the nature of the system and how many interrupts are there, how complex is the system and so on. For example; when our system has only periodic tasks then no need for prioritized handling scheme, since all of our tasks have equal importance. And when our system has a hardware interrupt from an external source that has hard real time determinism and must be processed quickly, using prioritized schemes is better. Another point is the number of interrupt

sources, when we have large amount of interrupts, using grouped priority handling scheme is a good choice.

## References

- [1] A.N. Sloss, D.Symes, C. Wright: *ARM System Developer's Guide*, **Publisher:** Morgan Kaufmann, March 25, 2004.
- [2] Steve Furber: *ARM System-On-Chip Architecture*, 2nd Edition, **Publisher:** Addison-Wesley, August 25, 2000.
- [3] David Seal: *ARM Architecture Reference Manual*, 2nd Edition, **Publisher:** Addison-Wesley, December 27, 2000.
- [4] *ARM710T Datasheet*, **Publisher:** ARM, August 1998, retrieved from [http://www.arm.com/documentation/ARMProcessor\\_Cores/index.html](http://www.arm.com/documentation/ARMProcessor_Cores/index.html).
- [5] *ARM Technical Reference Manual*, **Publisher:** ARM, March 2006, retrieved from [http://www.arm.com/documentation/ARMProcessor\\_Cores/index.html](http://www.arm.com/documentation/ARMProcessor_Cores/index.html).

## PHASE IV

### Start Application Development

Writing a Makefile:

<http://www.gnu.org/software/make/manual/make.html>

```
#Modified version of Makefile from WinARM Projects. Copyright Martin Thomas.
#Modified code. Copyright Vikas Agrawal, Published under GPL.
TOPDIR      = $(CURDIR)
OBJECT_DIR  = object-files
SOURCE_DIR  = source
OBJTREE     := $(if $(OBJECT_DIR),$(OBJECT_DIR),$(CURDIR))
SRCTREE     := $(if $(SOURCE_DIR),$(SOURCE_DIR),$(CURDIR))

# load other configuration
# Define programs and commands.

## Create ROM-Image (final)
RUN_MODE=ROM_RUN
## Create RAM-Image (debugging)
#RUN_MODE=RAM_RUN

#VECTOR_LOCATION=VECTORS_IN_ROM ## - Exception vectors in ROM:
VECTOR_LOCATION=VECTORS_IN_RAM ## - Exception vectors in RAM:

#Format
FORMAT = binary

# target file name
TARGET = project

MCU      = arm920t
SUBMDL   = AT91RM9200
THUMB_IW = -mthumb-interwork

# Optimization level, can be [0, 1, 2, 3, s].
OPT      = 0
#Debugging Information
DEBUG    = gdb3
```



```

#-----
# Compiler flag to set the C Standard level.
# c89 - "ANSI" C
# gnu89 - c89 plus GCC extensions
# c99 - ISO C99 standard (not yet fully implemented)
# gnu99 - c99 plus GCC extensions
#-----

CSTANDARD      = -std=gnu99
CDEFS          = -D$(RUN_MODE) # Place -D or -U options for C here
CINCS          = -I./include/ # Place -I options here

ifdef VECTOR_LOCATION
CDEFS          += -D$(VECTOR_LOCATION)
ADEFS          += -D$(VECTOR_LOCATION)
endif

#-----
# Compiler flags.
# -Wa,...: tell GCC to pass this to the assembler.
# -adhlns...: create assembler listing
#-----
CFLAGS        = -g$(DEBUG)
CFLAGS        += $(CDEFS) $(CINCS)
CFLAGS        += -O$(OPT)
CFLAGS        += -Wall -Wimplicit -Wcast-align
CFLAGS        += -Wpointer-arith -Wswitch
CFLAGS        += -Wredundant-decls -Wreturn-type -Wshadow -Wunused
CFLAGS        += -Wa,-adhlns=$(subst $(suffix $<),.lst,$<)
CFLAGS        += -finline-functions
CFLAGS        += -Wno-implicit
#-----

#-----
# flags only for C
ONLYFLAGS += -Wnested-externs
ONLYFLAGS += $(CSTANDARD)
#-----

#-----
# Assembler flags.
# -Wa,...: tell GCC to pass this to the assembler.
# -adhlns: create listing
# -g$(DEBUG):have the assembler create line number information
ASFLAGS = $(ADEFS) -O1 -Wa,-adhlns=$(<:.S=.lst),-g$(DEBUG)
#-----

```

```

MATH_LIB = -lm

#-----
# Linker flags.
# -Wl,...      :   tell GCC to pass this to linker.
# -Map         :   create map file
# --cref       :   add cross reference to  map file
#-----

LDFLAGS = -nostartfiles -Wl,-Map=$(TARGET).map,--cref
LDFLAGS += $(MATH_LIB)
LDFLAGS += -lc -lgcc
# Set Linker-Script Depending On Selected Memory and Controller
ifeq ($(RUN_MODE),RAM_RUN)
LDFLAGS += -T$(SUBMDL)-RAM.ld
else
LDFLAGS += -T$(SUBMDL)-ROM.ld
endif

#-----

#-----
# Compiler flags to generate dependency files.
# this options just outputs the stage to a file for later reading. nothing to do with compilation.
# Passing -M to the driver implies -E, and suppresses warnings with an implicit -w.
# -MP adds phony target helps in avoiding errors when header file is removed but makefile not
# updated
# -MD #Passing -MD to the driver does not implies -E, and suppresses warnings with an implicit -w.
# Combine all necessary flags and optional flags.
# Add target processor to flags.
#-----
GENDEPFLAGS = -MD -MP -MF .dep/$(@F).d

# Combine all necessary flags and optional flags.
# Add target processor to flags.
ALL_CFLAGS = -mcpu=$(MCU) $(THUMB_IW) -I. $(CFLAGS) $(GENDEPFLAGS)
ALL_ASFLAGS = -mcpu=$(MCU) $(THUMB_IW) -I. -x assembler-with-cpp $(ASFLAGS)

TOOLCHAIN = linux
# Define programs and commands.
SHELL      = sh
CC         = arm-$(TOOLCHAIN)-gcc

```

```

OBJCOPY = arm-$(TOOLCHAIN)-objcopy
OBJDUMP = arm-$(TOOLCHAIN)-objdump
SIZE    = arm-$(TOOLCHAIN)-size
NM       = arm-$(TOOLCHAIN)-nm
REMOVE  = rm -f
COPY    = cp

```

makevariables:

```

    @echo The Makefile inbuilt Variables
    @echo MAKEFILES = $(MAKEFILES)
    @echo MAKEFILE_LIST = $(MAKEFILE_LIST)
    @echo .DEFAULT_GOAL = $(DEFAULT_GOAL)
    @echo MAKE_RESTARTS = $(MAKE_RESTARTS)
    @echo .FEATURES = $(FEATURES)
    @echo .INCLUDE_DIRS = $(INCLUDE_DIRS)
.PHONY : .VARIABLES
    @echo .VARIABLES = $(VARIABLES)

```

# Define Messages

```

MSG_ERRORS_NONE           = Errors: none
MSG_BEGIN                  = "----- begin (mode: $(RUN_MODE)) -----"
MSG_END                    = ----- end -----
MSG_SIZE_BEFORE            = Size before:
MSG_SIZE_AFTER             = Size after:
MSG_FLASH                  = Preparing load file for Flash:
MSG_EXTENDED_LISTING       = Creating Extended Listing:
MSG_SYMBOL_TABLE           = Creating Symbol Table:
MSG_LINKING                = Linking:
MSG_COMPILING_ARM          = "Compiling C (ARM-only):"
MSG_ASSEMBLING_ARM         = "Assembling (ARM-only):"
MSG_CLEANING               = Cleaning project:

```

```

# Attempt to create a output directory. source/main.c      source/object-files/main.o
$(shell [ -d ${OBJECT_DIR} ] || mkdir -p ${OBJECT_DIR})

```

# List Assembler source files here which must be assembled in ARM-Mode..

```

ASRCARM    = cstartup_gnu.S isr.S
SRCARM     = main.c init.c mci_device.c usart_device.c st_device.c led_device.c

```

# Define all object files.

```

AOBJARM    = $(addprefix $(OBJTREE)/,$(ASRCARM:.S=.o))

```

```

COBJARM    = $(addprefix $(OBJTREE)/,$(SRCARM:.c=.o))

# Define all listing files.
LST        = $(ASRCARM:.S=.lst) $(SRCARM:.c=.lst)

# Default target.
all: makevariables begin gccversion sizebefore build sizeafter finished end

begin:
    @echo
    @echo $(MSG_BEGIN)

# Display compiler version information.
gccversion :
    @$(CC) --version

# Display size of file.
BINSIZE    = $(SIZE) --target=$(FORMAT) $(TARGET).bin
ELFSIZE    = $(SIZE) -A $(TARGET).elf

sizebefore:
    @if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_BEFORE); $(ELFSIZE);
$(BINSIZE); echo; fi

build: elf bin obd sym

elf: $(TARGET).elf
bin: $(TARGET).bin
obd: $(TARGET).obd
sym: $(TARGET).sym

# Link: create ELF output file from object files.
.SECONDARY : $(TARGET).elf
.PRECIOUS : $(AOBJARM) $(COBJARM)

%.elf: $(AOBJARM) $(COBJARM)
    @echo
    @echo $(MSG_LINKING) $$@
    $(CC) $(ALL_CFLAGS) $(AOBJARM) $(COBJARM) --output $$@ $(LDFLAGS)
#    $(OBJCOPY) --strip-debug $$@ $$@

# Compile: create object files from C source files. ARM-only
$(COBJARM) : $(OBJTREE)/%.o : $(SRCTREE)/%.c

```

```

@echo
@echo $(MSG_COMPILING_ARM) $<
$(CC) -c $(ALL_CFLAGS) $(ONLYFLAGS) $< -o $$@
$(OBJDUMP) -h -S -C $$@ > $(patsubst %.o,%.obd,$@)
$(NM) -n $$@ > $(patsubst %.o,%.sym,$@)

# Assemble: create object files from assembler source files. ARM-only
$(AOBJARM) : $(OBJTREE)/%.o : $(SRCTREE)/%.S
    @echo
    @echo $(MSG_ASSEMBLING_ARM) $<
    $(CC) -c $(ALL_ASFLAGS) $< -o $$@
    $(OBJDUMP) -h -S -C $$@ > $(patsubst %.o,%.obd,$@)
    $(NM) -n $$@ > $(patsubst %.o,%.sym,$@)

# Create final output file (.bin) from ELF output file.
%.bin: %.elf
    @echo
    @echo $(MSG_FLASH) $$@
    $(OBJCOPY) -O $(FORMAT) $< $(patsubst %.bin,%.bin,$@)      # just to show the use
of patsubst

# Create extended listing file from ELF output file.
# testing: option -C
%.obd: %.elf
    @echo
    @echo $(MSG_EXTENDED_LISTING) $$@
    $(OBJDUMP) -h -S -C $< > $$@
#    $(OBJDUMP) -i > $(patsubst %.obd,%.lsq,$@)

# Create a symbol table from ELF output file.
%.sym: %.elf
    @echo
    @echo $(MSG_SYMBOL_TABLE) $$@
    $(NM) -n $< > $$@

sizeafter:
    @if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_AFTER); $(ELFSIZE);
$(BINSIZE); echo; fi

finished:
    @echo $(MSG_ERRORS_NONE)

end:
    @echo $(MSG_END)
    @echo

```

```

# Include the dependency files.
-include $(shell mkdir .dep 2>/dev/null) $(wildcard .dep/*)

# Target: clean project.
clean: begin clean_list finished end

clean_list :
    @echo
    @echo $(MSG_CLEANING)
    $(REMOVE) $(TARGET).bin
    $(REMOVE) $(TARGET).elf
    $(REMOVE) $(TARGET).map
    $(REMOVE) $(TARGET).sym
    $(REMOVE) $(TARGET).obd
    $(REMOVE) $(addprefix $(SRCTREE)/,$(LST))
    $(REMOVE) -r .dep
    $(REMOVE) -r $(OBJTREE)

# Listing of phony targets.
.PHONY : all makevariables begin gccversion sizebefore build sizeafter finished end elf bin obd sym
clean clean_list

```

## **About the SD Card**

<http://www.digitalspirit.org/file/?aff=,./docs/sd/ProductManualSDCardv2.2final.pdf>

### **1 Introduction**

#### **1.1 General Description**

The SanDisk Secure Digital (SD) Card is a flash-based memory card specifically designed to meet the security, capacity, performance and environmental requirements inherent in next generation mobile phones and consumer electronic devices. The SanDisk SD Card includes a copyright protection mechanism that complies with the security of the SDMI standard, and is faster and capable of higher

memory capacity. The SD Card security system uses mutual authentication and a “new cipher algorithm” to protect against illegal usage of the card content. Unsecured access to the user’s own content is also available. The physical form factor: pin assignment and data transfer protocol, with some additions, are forward compatible with the SD Card.

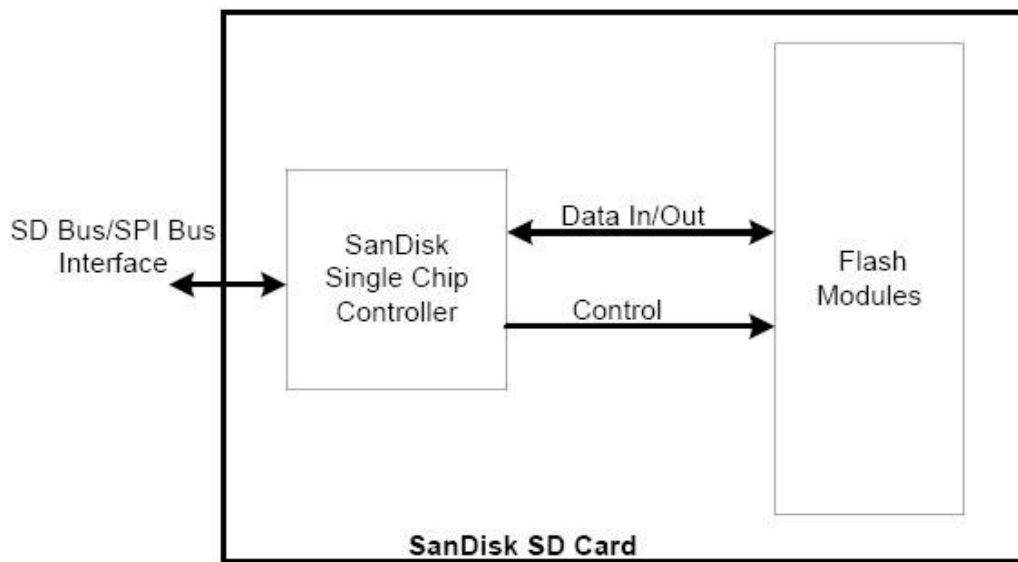
SanDisk SD Card communication is based on an advanced nine-pin interface (clock,

command, 4xData and 3xPower lines) designed to operate in a low voltage range. The

communication protocol is defined as part of this specification. The SD Card host interface supports regular MultiMediaCard operation as well. In other words, MultiMediaCard forward compatibility was kept. The main difference between the SD Card and MultiMediaCard is the initialization process. Matsushita Electric Company (MEI), Toshiba Corporation, and SanDisk Corporation defined the SD Card Specification originally.

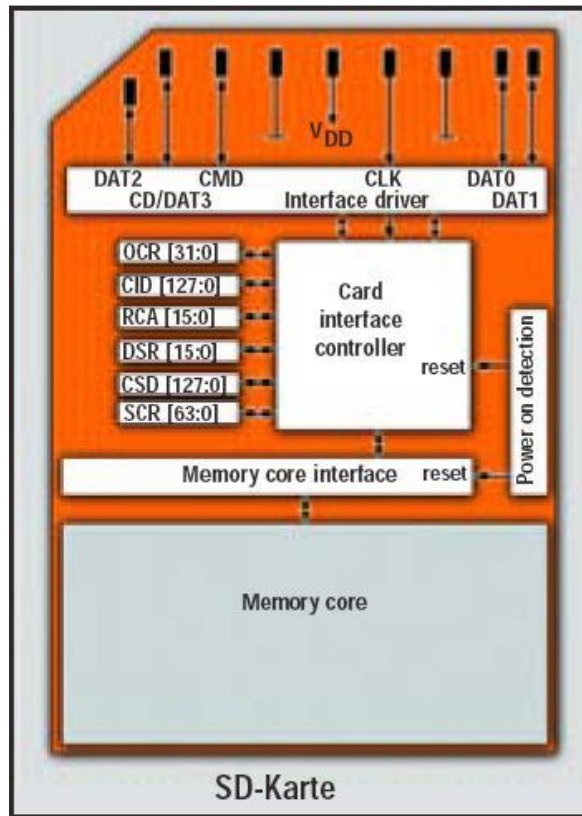
Currently, the Secure Digital Association (SDA) controls the specifications. The SanDisk SD Card was designed to be compatible with the SD Card Physical Specification. The SD Card Interface allows for easy integration into any design, regardless of microprocessor used. For compatibility with existing controllers, the SanDisk SD Card offers, in addition to the SD Card Interface, an alternate communication protocol based on the SPI standard.

Currently, the SanDisk SD Card provides up to 1024 million bytes of memory using flash memory chips, which were designed especially for use in mass storage applications. In addition to the mass storage specific flash memory chip, the SD Card includes an on-card intelligent controller which manages interface protocols, security algorithms for copyright protection, data storage and retrieval, as well as Error Correction Code (ECC) algorithms, defect handling and diagnostics, power management and clock control.



Pin No. SD Mode	Name	Type1	Description
1	CD/DAT32	I/O3, PP	Card detect/Data line [Bit 3] Command/Response Command is a bi-directional signal. Host and card drivers are operating in push- pull mode.
2	CMD	I/O, PP	Supply voltage ground Two ground lines.
3	VSS1	S	Supply voltage Power supply line for all cards.
4	VDD	S	Clock Clock is a host to card signal.
5	CLK	I	CLK operates in push-pull mode.
6	VSS2	S	Supply voltage ground Data line [Bit 0] Data lines are bi-directional signals. Host and card drivers are operating in push-pull mode.
7	DAT0	I/O, PP	Data line [Bit 1] Data lines are bi-directional signals. Host and card drivers are operating in push-pull mode.
8	DAT1	I/O, PP	Data line [Bit 2] Data lines are bi-directional signals. Host and card drivers are operating in push-pull mode.
9	DAT2	I/O, PP	





Name	SD Card Registers Width	Description
CID	128	Card identification number: individual card number for identification.
RCA6	16	Relative card address: local system address of a card, dynamically suggested by the card and approved by the host during initialization.
CSD	128	Card specific data: information about the card operation conditions.
SCR	64	SD Configuration Register: information about the SD Card's special features capabilities.
OCR	32	Operation Condition Register

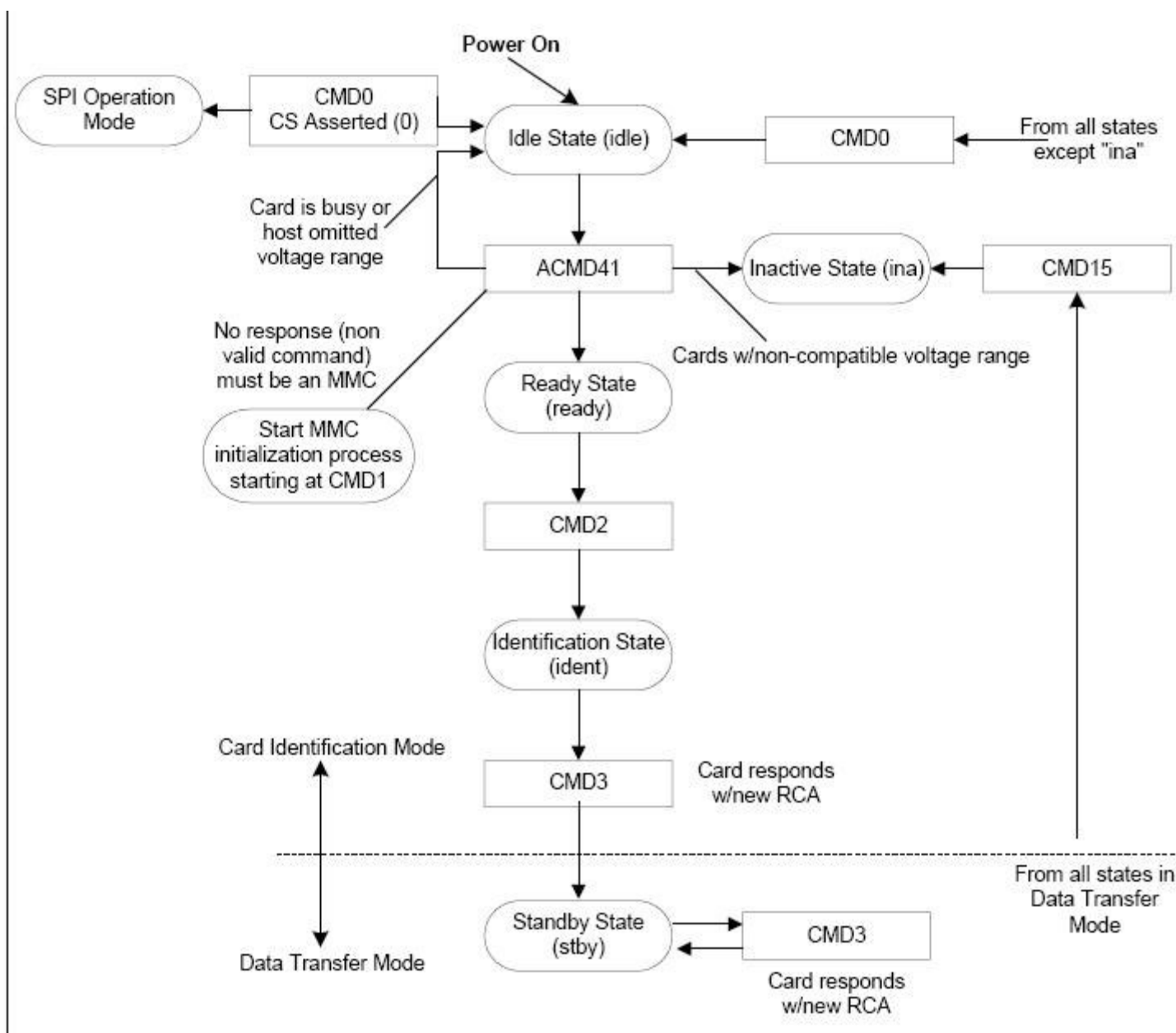
## COMMANDS in SD Card

### Class 0 & 1 commands: Basic commands and Read Stream commands

CMD0\_GO\_IDLE\_STATE  
 CMD1\_SEND\_OP\_COND  
 CMD2\_ALL\_SEND\_CID  
 CMD3\_SET\_RELATIVE\_ADDR

CMD3_MMC_SET_RELATIVE_ADDR CMD4_SET_DSR CMD7_SEL_DESEL_CARD CMD9_SEND_CSD CMD10_SEND_CID CMD11_MMC_READ_DAT_UNTIL_STOP CMD12_STOP_TRANSMISSION CMD13_SEND_STATUS CMD15_GO_INACTIVE_STATE
<b>Class 2 commands: Block oriented Read commands</b>
CMD16_SET_BLOCKLEN CMD17_READ_SINGLE_BLOCK CMD18_READ_MULTIPLE_BLOCK
<b>Class 4 commands: Block oriented write commands</b>
CMD24_WRITE_BLOCK CMD25_WRITE_MULTIPLE_BLOCK CMD27_PROGRAM_CSD
<b>Class 6 commands: Group Write protect</b>
CMD28_SET_WRITE_PROT CMD29_CLR_WRITE_PROT CMD30_SEND_WRITE_PROT
<b>Class 5 commands: Erase commands</b>
CMD32_TAG_SECTOR_START CMD33_TAG_SECTOR_END CMD34_MMC_UNTAG_SECTOR CMD35_MMC_TAG_ERASE_GROUP_START CMD36_MMC_TAG_ERASE_GROUP_END CMD37_MMC_UNTAG_ERASE_GROUP CMD38_ERASE
<b>Class 7 commands: Lock commands</b>
CMD42_LOCK_UNLOCK
<b>Application Specific Commands</b>
ACMD55_NEXTCMD_APP_CMD ACMD56_GEN ACMD6_SET_BUS_WIDTH ACMD13_SDCARD_STATUS ACMD22_SEND_NUM_WR_BLOCKS ACMD23_SET_WR_BLK_ERASE_COUNT ACMD41_SEND_APP_OP_COND ACMD42_SET_CLR_CARD_DETECT ACMD51_SEND_SCR

Initialisation State Diagram:



At power-up, the SD card defaults to the proprietary SD bus protocol. The host issues command 0 (GO\_IDLE\_STATE). The card responds with response format R1 (see the table below). The idle state bit is set high to signify that the card has entered idle state. The SD specification requires that the host issue an initialization command before any other requests can be processed. Sending command 55 (APP\_CMD) followed by application command 41 (SEND\_OP\_COND) to the

card completes this important step. MMC cards do not respond to command 55, which can be used to reject MMC cards as invalid media. This command sequence is repeated until all bits in the R1 response from the card are zero (i.e., the IDLE bit goes low).

## Examining the SD Card Responses

To read card registers or blocks from the card, we must first understand how the card responds to our inquiries. In SPI mode, the SD card replies to the commands SEND\_CSD (9), SEND\_CID (10), and READ\_SINGLE\_BLOCK (17) with a R1 format reply. A start token, the requested data, and finally a CRC-16 checksum over the data follow. We must not assume that the R1 reply and the data start token occur immediately one after the other, as the bus can go to the idle state for some time between these two events.

Command	Mnemonic	Argument	Reply	Description
0 (0x00)	GO_IDLE_STATE	none	R1	Resets the SD card.
9 (0x09)	SEND_CSD	none	R1	Sends card-specific data.
10 (0x0a)	SEND_CID	none	R1	Sends card identification.
17 (0x11)	READ_SINGLE_BLOCK	address	R1	Reads a block at byte address.
24 (0x18)	WRITE_BLOCK	address	R1	Writes a block at byte address.
55 (0x37)	APP_CMD	none	R1	Prefix for application command.
59 (0x3b)	CRC_ON_OFF	Only Bit 0	R1	Argument sets CRC on (1) or off (0).
41 (0x29)	SEND_OP_COND	none	R1	Starts card initialization.

The SD card contains several important registers that provide information about the SD card. The most important register is the Card Specific Data register (CSD). For our sample application we are interested in the block size and total size of the memory. We must also pay attention to the Card Identification Data register (CID), as it contains details about the cards' manufacturer and serial number. **Below table** shows the layout of the CID and CSD registers.

## Reading the CSD and CID Register Meta-Data

The SEND\_CSD and SEND\_CID commands send back register contents used to determine the SD card parameters. These commands return a fixed number of bytes, which corresponds to the size of the CSD or CID register, respectively. The argument contained within the command bytes is ignored by the SD card for these SEND commands.

### Card Identification Register

Name		CID Register Definitions			CID Value	Comments
		Type	Width	CID-Slice		
Manufacturer ID (MID)	Binary	8		[127:120]	0x03	Manufacturer IDs are controlled and assigned by the SD Card Association. Identifies the card OEM and/or the card contents. The OID is assigned by the 3C.13
OEM/Application ID (OID)	ASCII	16		[119:104]	SD Code 0x44 SD02G SD01G SD512 SD256 SD128 SD64 SD32 SD16	Five ASCII characters long
Product Name (PNM)	ASCII	40		[103:64]	Product Revision xx Product Serial Number	Two binary-coded decimal digits
Product Revision <sup>14</sup>	BCD	8		[63:56]	32-bit unsigned integer	
Serial Number (PSN)	Binary	32		[55:24]	---	---
Reserved	---	4		[23:20]	Manufacture date (for ex. April 2001= 0x014)	Manufacturing date—yy/mm (offset from 2000)
Manufacture Date Code (MDT)	BCD	12		[19:8]	CRC7*	Calculated
CRC7 checksum (CRC)	Binary	7		[7:1]	---	---
Not used, always “1”	---	1		[0:0]		

### Card Specific Data Register

Field	Width	Cell	CSD	CSD Value	CSD Code	Description
		Type	Slice			
CSD_STRUCTURE	2	R	[127:126]	1.0	0	CSD structure
---	6	R	[125:120]	---	000000b	Reserved

TAAC	8	R	[119:112]	1.5 msec	00100110	Data read access time-1
NSAC	8	R	[111:104]	0	00000000b	Data read access time-2 in CLK cycles (NSAC*100)
TRANS_SPEED	8	R	[103:96]	High-speed 50MHz	0110010 01011010	Max. data transfer rate
CCC	12	R	[95:84]	All (inc. WP, lock/unlock)	5F5	Card command Classes
READ_BL_LEN	4	R	[83:80]	2G Up to 1G	Ah 9h	Max. read data block length
READ_BL_PARTIAL	1	R	[79:79]	Yes	1b	Partial blocks for read allowed
WRITE_BLK_MISALIGN	1	R	[78:78]	No	0b	Write block Misalignment
READ_BLK_MISALIGN	1	R	[77:77]	No	0b	Read block Misalignment
DSR_IMP	1	R	[76:76]	No	0b	DSR Implemented
---	2	R	[75:74]	---	00b	Reserved
				2 GB	F24h	
				1 GB	F22h	
				512 MB	F1Eh	
				256 MB	F13h	
				128 MB	F03h	
				64 MB	EDFh	
				32 MB	74Bh	
C_SIZE	12	R	[73:62]	16 MB	383h	Device size
VDD_R_CURR_MIN	3	R	[61:59]	100 mA	111b	Max. read current @VDD min.
VDD_R_CURR_MAX	3	R	[58:56]	80 mA	110b	Max. read current @VDD max.
VDD_W_CURR_MIN	3	R	[55:53]	100 mA	111b	Max. write current @VDD min.
VDD_W_CURR_MAX	3	R	[52:50]	80 mA	110b	Max. write current @VDD max.
				2G=2048	0x07	
				1G=1024	0x07	
				512=512	0x06	
				256=256	0x05	
				128=128	0x04	
				64=64	0x03	
				32=32	0x03	
				16=16	0x03	
C_SIZE_MULT						Multiplier
ERASE_BLK_EN	1	R	[46:56]	Yes	1b	Erase single block enable
SECTOR_SIZE	7	R	[45:39]	32 blocks	0011111b	Erase sector Size
WP_GRP_SIZE	7	R	[38:32]	128 sectors	1111111b	Write protect group size
WP_GRP_ENABLE	1	R	[31:31]	Yes	1b	Write protect group enable
Reserved	2	R	[30:29]	---	0b	Reserved for MMC compatibility
R2W_FACTOR	3	R	[28:26]	x16 2G	0100b Ah	Factor
WRITE_BL_LEN	4	R	[25:22]	Up to 1G	9h	Max. write data block length

WRITE_BL_PARTIAL	1	R	[21:21]	No	0	Partial blocks for write allowed
---	5	R	[20:16]	---	00000b	Reserved
FILE_FORMAT_GRP	1	R/W (1)	[15:15]	0	0b	File format group
COPY	1	R/W (1)	[14:14]	Not original	1b	Copy flag (OTP)
PERM_WRITE_PROTECT	1	R/W (1)	[13:13]	Not protected	0b	Permanent write protection
TMP_WRITE_PROTECT	1	R/W (1)	[12:12]	Not protected	0b	Temporary write Protection
FILE_FORMAT	2	R/W (1)	[11:10]	HD w/partition	00b	File format
Reserved	2	2	R/W [9:8]	---	---	Reserved
CRC	7	R/W	[7:1]	---	CRC7	CRC
---	1	---	[0:0]	---	1b	Not used, always 1

## Relative Card Address Register

The 16-bit Relative Card Address (RCA) Register carries the card address that is published by the card during the card identification. This address is used for the addressed host-card communication after the card identification procedure.

## DATA transfer State Diagram:

### Reading a Block of Data from the SD Card

Reading a block of data from the SD card is quite simple. The host issues the READ\_SINGLE\_BLOCK command with a starting byte address as the argument. This address must be aligned with the beginning of a block on the media. The SD card then evaluates this byte address and responds back with an R1 command reply. An out-of-range address is indicated in the command reply.

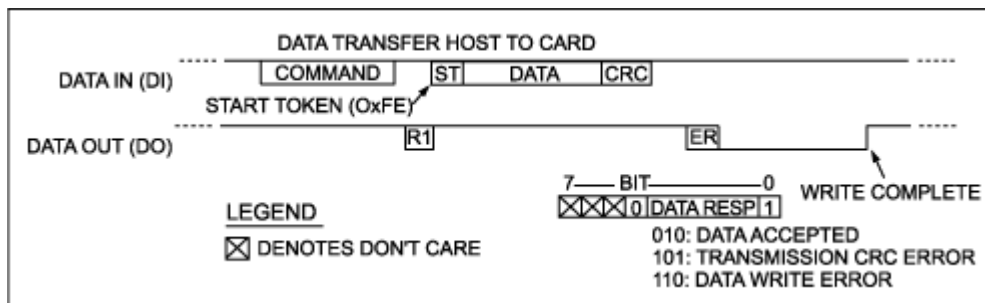
If the read is completed from the SD media without error, a start data token is sent followed by a fixed number of data bytes and two bytes for the CRC-16 checksum. The start data token is not sent if the SD card encounters a hardware failure or media read error. Rather, an error token is sent and the data transfer is aborted.

### Writing a Block of Data to the SD Card

Writing a block of data is similar to reading, as the host must supply a byte address that is aligned with the SD card block boundaries. The write block size must equal READ\_BL\_LEN, which is typically 512 bytes. A write is initiated by issuing the WRITE\_BLOCK (24) command, to which the SD card responds with the R1 command response format. If the command response indicates that the write can

proceed, the host transmits the data start token followed by a fixed number of data bytes, and ends with a CRC-16 checksum of the sent data. The SD card returns a data response token indicating the acceptance or rejection of the data to be written.

If the data is accepted, the SD card holds the DO line low continuously while the card is busy. The host is not obligated to keep the card select low during the busy period, and the SD card releases the DO line if CS is deasserted. This process is useful when more than one device is connected to the SPI bus. The host can wait for the SD card to release the busy indication, or check the card by periodically asserting the chip select. If the card is still busy, it will pull the DO line low to indicate this state. Otherwise, the card returns the DO line to the idle state (see **Figure 7**).







## **THE SOFTWARE Algorithm**

Algorithm Diagram

Write StartUP Code

Create Device Driver for SDcard

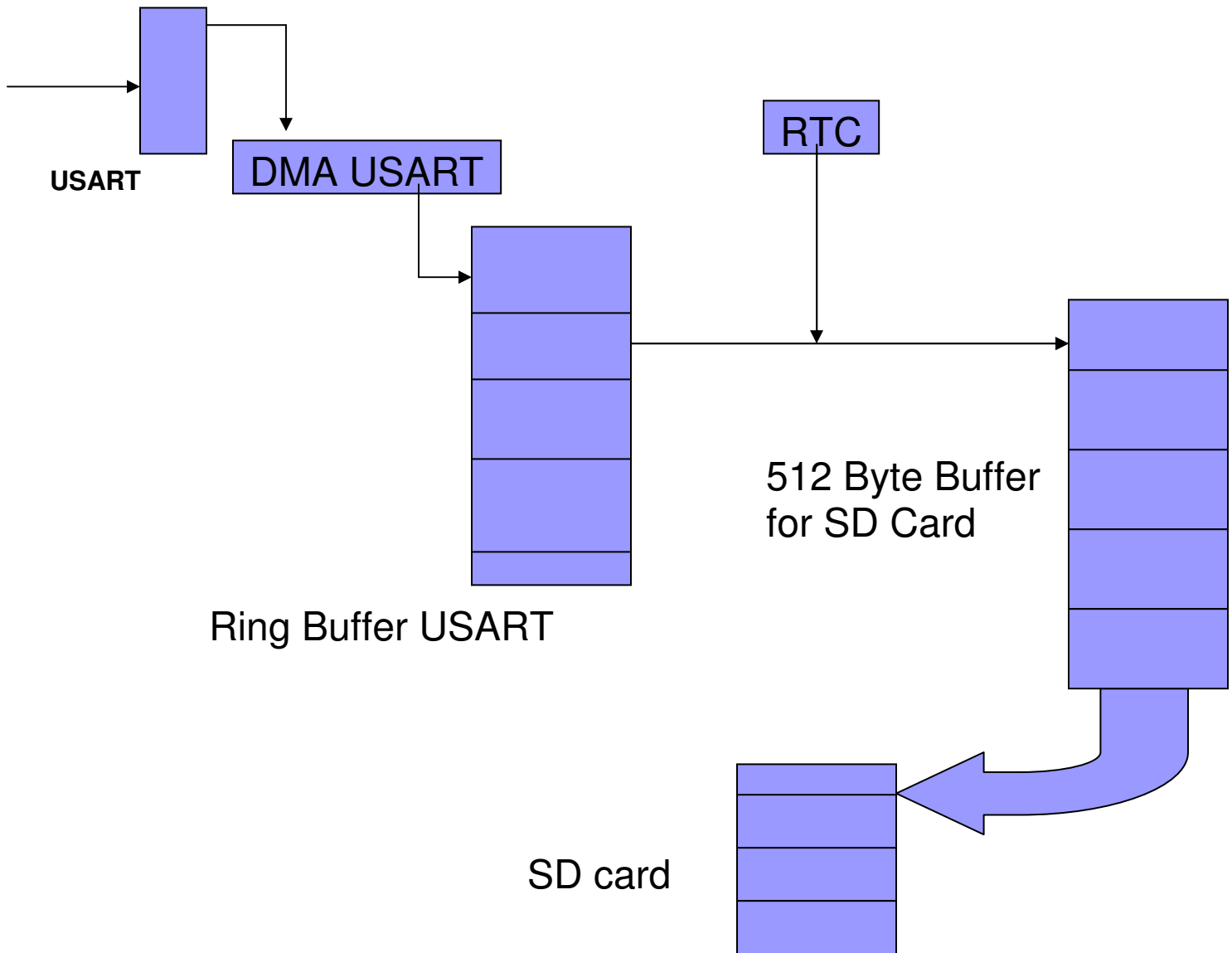
Create Device Driver for USART

Create the Application Program

- Configure Peripheral DMA Controller for USART.
- Configure Peripheral DMA Controller for SDCard.
- Store Incoming bytes into RingBuffer.
- Pick the Bytes from the RingBuffer into another Buffer ( BufferSD ).
- Insert Time stamp if byte is a line feed.
- Copy to SDCard if BufferSD is 512 bytes full.

Write Linker Script

## Algorithm Diagram



## C Startup Files for AT91RM9200-EK

All software package projects use a reduced C startup that is as basic as possible. It calls C functions as soon as possible to improve code reuse and readability. It is only assembler-dependent.

The AT91 software packages' minimum C startup proceeds as follows:

- sets the ARM exception vectors
- calls an `AT91F_LowLevelInit()` C function (defined by the application)

- sets stack pointers
- initializes C variables
- calls the C function main()

Most of the hardware initialization of the device should be done in the AT91F\_LowLevelInit() function. However, it is important to note that this function cannot use the stack-to-store variables or arguments as they are not initialized. Obviously, the C startup can/must be modified according to the application requirements.

The software packages provide an example of C startup to the user to get a quick start

using AT91 products.

**Main Files** All AT91 software packages projects use a main file that is as basic as possible. It is called by C startup after C initialization.

After the main ( high level language) is terminated, the startup code again takes control.

In Atmel board all the peripherals are memory mapped.

The address can be assigned as such – int \*pRegister = (int \*) 0x10002345 or

#define pRegister (void \*) 0x10002345

## AT91RM9200.h

Below is a extract from the header file **AT91RM9200.h**, All the below mentioned codes are examples to just make it easier to understand what the basic naming conventions are.

A constant represented by **AT91C\_xxx**

#define AT91C\_PIO\_PA18 ((unsigned int) 1 << 18) // Pin Controlled by PA18

#define AT91C\_PA18\_RXD0 ((unsigned int) AT91C\_PIO\_PA18 //USART0 Receive Data

Symbol to the peripheral ID.

#define AT91C\_ID\_US0 ((unsigned int) 6) // USART 0

C structures representing the peripheral user interface

Each peripheral is controlled by a set of registers mapped in the microcontroller address space. These registers are defined as members of a C structure. Using these structures allows the compiler to work with only one base address and indirect addressing.

These structures are used by the C functions of the hardware API C functions to improve code reuse. For example, the same handler can be used to drive two USARTs.

```
typedef struct _AT91S_PIO {  
    AT91_REG PIO_PER; // PIO Enable Register  
    AT91_REG PIO_PDR; // PIO Disable Register  
    AT91_REG PIO_PSR; // PIO Disable Register  
    AT91_REG PIO_OER; // PIO Output Enable Register  
    AT91_REG PIO_ODR; // PIO Output Disable Register  
    AT91_REG PIO_OSR; // PIO Output Status Register  
    AT91_REG PIO_IFER; // PIO Input Filter Enable Register  
    AT91_REG PIO_IFDR; // PIO Input Filter Disable Register  
    AT91_REG PIO_IFSR; // PIO Input Filter Disable Register  
    AT91_REG PIO_MDSR; // PIO Multi-drive Status Register  
} AT91S_PIO, *AT91PS_PIO;
```

- Using direct register access only

```
(*AT91C_US0_THR) = 'a';  
I = (*AT91C_US0_RHR);
```

- Using direct register access through a pointer. This helps C compiler optimization and improves code readability.

```
AT91_REG *pRhr = AT91C_US0_RHR, *pThr = AT91C_US0_THR;  
*pThr = 'a';  
I = (*pRhr);
```

- Using a C structure API

This improves code re-use among ARM-based products.

After compilation, the code size should not be larger than when using direct register access as described above.

Debug is easier, as dumping the pUs0 variable provides all the USART0 settings.

```
AT91PS_USART pUs0 = AT91C_BASE_US0;  
pUs0->US_RHR = 'a';  
I = pUs0->US_RHR;
```

- Using the hardware API function layer

This improves code re-use between ARM-based products.

After compilation, code size should not be larger than when using direct register access as described above.

```
AT91PS_USART pUs0 = AT91C_BASE_US0;
```

```
AT91F_US_Configure(pUs0, ...);
```

### Peripheral Base Address

A symbol is associated with each peripheral base address. This is to improve code reuse.

```
#define AT91C_BASE_AIC ((AT91PS_AIC) 0xFFFFF000) // (AIC) Base Address
```

### Peripheral Register Absolute Address

A symbol is defined for each register absolute address. This offers another way to access a register without using the peripheral C structure and the peripheral base address associated.

```
#define AT91C_PIOB_PER ((AT91_REG *) 0xFFFFF600) // (PIOB) PIO Enable Register
```

### Bit Field and Masks

A symbol is defined for each bit field mask. Reference to this symbol can be found in the HTML documentation. Using a symbol instead of the hard-coded immediate value results in better code reuse.

```
#define AT91C_PMC_PCK ((unsigned int) 0x1 << 0) // (PMC) Processor Clock
```

## lib\_AT91RM9200.h

### Hardware API Function Layer

The hardware API functions layer is a set of C inline functions. The goal of these functions is to provide a software API for the peripheral hardware. These functions can be used to improve code reuse between different AT91 products. They also illustrate recommended techniques for programming the peripheral. These functions consist of only a few instructions. The C preprocessor replaces the inline function called by the body of this function. Arguments are checked as with a standard function. The ARM-based product software package uses inline functions for two purposes:

- To avoid passing of arguments and branch instructions that are too long compared to the function size.

- To prevent an increase in the size of the resulting binary code when not used.

```

/*-----
/* \fn AT91F_DBGU_InterruptEnable
/* \brief Enable DBGU Interrupt
/*-----
__inline void AT91F_DBGU_InterruptEnable(
AT91PS_DBGU pDbgu, // \arg pointer to a DBGU controller
unsigned int flag) // \arg dbggu interrupt to be enabled
{
pDbgu->DBGU_IER = flag;
}

```

## PIO

The I/O line can either be a GPIO or Peripheral Controller I/O. This can be configured by Enabling the Peripheral ( PER ) or Disabling a Peripheral ( PDR ). When a pin is multiplexed with one or two peripheral functions, the selection is controlled with the registers PIO\_PER (PIO Enable Register) and PIO\_PDR (PIO Disable Register). - Scenarios where multiplexing of a single pin ( if one needs both peripheral on that pin )

When an I/O line is general-purpose only, i.e. not multiplexed with any peripheral I/O, programming of the PIO Controller regarding the assignment to a peripheral has no effect and only the PIO Controller can control how the pin is driven by the product.

### PIO or Peripheral :

Use PDR to disable peripheral fn ( rem it is automatically disabled at reset ) . Use PUDR to pull the pin low or high. .A value of 0 indicates that the pin is controlled by the corresponding on-chip peripheral selected in the PIO\_ABSR (AB Select Status Register). A value of 1 indicates the pin is controlled by the PIO controller.

If a pin is used as a general purpose I/O line (not multiplexed with an on-chip peripheral), PIO\_PER and PIO\_PDR have no effect and PIO\_PSR returns 1 for the corresponding bit. So PIO\_PER is only valid in the case of multiplexing and

when the line corresponding pin has to be run by the peripheral PIO\_PSR shows which functionality ( Peripheral or PIO ) PIO\_ABSE shows which peripheral ( A or B ) is enabled. '0' indicates Peripheral A is selected.

When driven by PIO controller :

When the I/O line is controlled by the PIO controller, the pin can be configured to be driven. This is done by writing PIO\_OER (Output Enable Register) and PIO\_PDR (Output Disable Register). The results of these write operations are detected in IO\_OSR (Output Status Register). When a bit in this register is at 0, the corresponding I/O line is used as an input only. When the bit is at 1, the corresponding I/O line is driven by the PIO controller.

### Data Transfers from USART to RingBuffer

The peripheral triggers PDC transfers using transmit (TXRDY) and receive (RXRDY) signals. When the peripheral receives an external character, it sends a Receive Ready signal to the PDC which then requests access to the system bus. When access is granted, the PDC starts a read of the peripheral Receive Holding Register (RHR) and then triggers a write in the memory. After each transfer, the relevant PDC memory pointer is incremented and the number of transfers left is decremented. When the memory block size is reached, a signal is sent to the peripheral and the transfer stops. The same procedure is followed, in reverse, for transmit transfers.

As soon as the counter reaches 0, the peripheral END flag is set. The peripheral END Flag is automatically cleared when one of the counter-registers (Counter or Next Counter Register) is written. Before that it is set to 1.

Lastly Running from Flash:

```
bootm go 0x10100000
bootdelay 3
copy the Variables from Flash to SDRAM by inserting the following code
#ifdef ROM_RUN
// Relocate .data section (Copy from ROM to RAM)
    LDR    R1,=_etext
    LDR    R2,=_data
```



```

        LDR    R3,=_edata
LoopRel:    CMP    R2,R3
            LDRLO R0,[R1],#4
            STRLO R0,[R2],#4
            BLO   LoopRel
#endif

// Clear .bss section (Zero init)
        MOV    R0,#0
        LDR    R1,=__bss_start__
        LDR    R2,=__bss_end__
LoopZI:    CMP    R1,R2
            STRLO R0,[R1],#4
            BLO   LoopZI

```

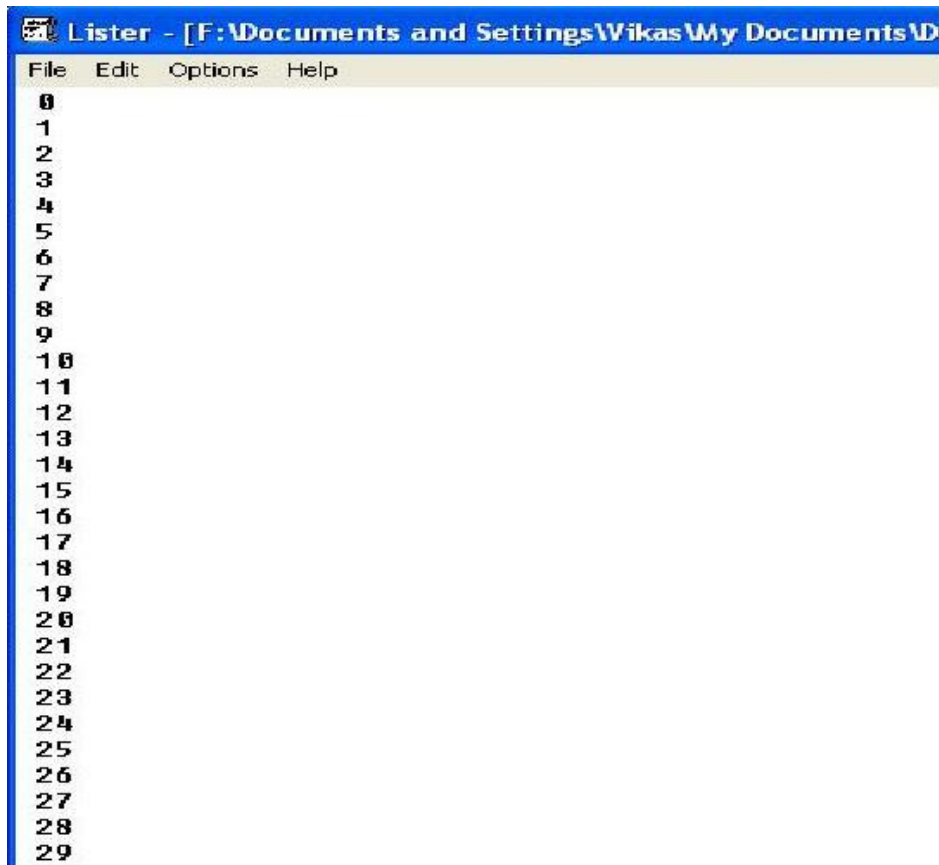
## Constraints

During the development of the software, one potential constraint was the timing issue. Since the bytes to be stored is received around every 100 microsecond approx., the processor should check the byte for linefeed, store it into another buffer and also if the buffer is 512 bytes full, raise an interrupt to store it in the SD Card. All these constraints in the worst case were much more than 100 microseconds ( around 1 millisecond ), and hence the bytes received on the USART were simply lost subsequently raising an Overflow Error.

To solve this problem, an another approach was taken. Previously as stated above, every byte received on the USART was checked for its type, before the next byte is allowed to be received. In the new approach, which was successfully implemented and suggested by my industry supervisor, a Ring Buffer was created which was made to work independent of the type of the byte. The bytes were stored in the buffer as they came via a **DMA Controller**. At the same time, the bytes received were copied to a SD Card buffer, timestamped and when this buffer was full were transferred to the SD Card. During the worst case scenarios ( when byte is a linefeed **and** when the buffer has to be stored in the SD card ), the maximum jitter that the transfer from USART Buffer to the SD Card Buffer was around 10 bytes. This is duly made up in the next cycles when there is no worst case scenario. Another advantage of this method is that the processor has a lot more time to do things apart from just polling the incoming byte on the USART.

## Result

Input File: This is an example input file with a set of numbers followed by a carriage return.



When the data above is fed into the Software then the resulting data stored in the SD card will look like as mentioned below. The time stamp is made at every line-feed.

Lister - [F:\Documents and Settings\Wikas\My Documents\Documentation\lister

File Edit Options Help

Welcome to reading form SD card

i> to get information about the card

n> to read SD card one by one

a> to read from the starting block to last block saved

q> to quit tests

Welcome to the RS232 to SDCard Datalogger

|-----09,01,2007-----||-----  
8-----|

03:16:24.674-0  
03:16:24.674-1  
03:16:24.674-2  
03:16:24.674-3  
03:16:24.675-4  
03:16:24.675-5  
03:16:24.675-6  
03:16:24.675-7  
03:16:24.676-8  
03:16:24.676-9  
03:16:24.676-10  
03:16:24.677-11  
03:16:24.677-12  
03:16:24.677-13  
03:16:24.678-14  
03:16:24.678-15  
03:16:24.678-16  
03:16:24.679-17  
03:16:24.679-18  
03:16:24.679-19  
03:16:24.680-20

## **Future Applications**

At present, this DataLogger is only suitable for data those are in ASCII format. In future, by changing the software, any other kind of patterns can also be taken care of.

Currently, the Software Design is of just a single main loop. It is possible in future to use an operating system by which various tasks can be scheduled and that the processor has not only one single task to perform.

The DataLogger can be made to work generic on the ARM Processors of ARM920T family and hence not get dependant on the architecture of the Board manufacturer ( for ex. Atmel ).

## **Reference**

AT91RM9200 Datasheet and Manual – [www.atmel.com](http://www.atmel.com)

SanDisk SD Card Product Manual – [www.sdcard.org](http://www.sdcard.org)

Arm Architecture reference manual – [www.arm.co.uk](http://www.arm.co.uk)

AT91RM9200DK U-Boot User Manual – [www.at91.com](http://www.at91.com)

BDI2000 Manual – [www.abatron.ch](http://www.abatron.ch)

ARM Exception Handling - [www.iti.uni-stuttgart.de/~radetzki/Seminar06/08\\_report.pdf](http://www.iti.uni-stuttgart.de/~radetzki/Seminar06/08_report.pdf)

AT49BV6416 Flash Memory DataSheet – [www.atmel.com](http://www.atmel.com)

ARM Instruction Set - <http://www.heyrick.co.uk/assembler/qfinder.html>

OverView of ARM – [www.wikipedia.org](http://www.wikipedia.org)