AT91 Library

Background

The AT91 library is a set of C and assembly source and project files aimed at helping AT91 customers get started quickly with the AT91 series microcontrollers.

The AT91 library contains:

- Header files for defining the AT91M40400 in C language
- Assembly Include files defining the AT91M40400 in ARM® Thumb® assembly language
- Examples of how to access the AT91M40400 peripherals
- Project Template for the AT91EB01 and AT91DB01
- Project examples to start up
- Application Notes source files
- Performance benchmarks

All delivered sources are free of charge and can be copied or modified without any authorization.

The software is delivered “as is” without warranty or condition of any kind, either express, implied or statutory. This includes without limitation any warranty or condition with respect to merchantability or fitness for any particular purpose, or against the infringements of intellectual property rights of others.

Getting Started with the AT91 Library

In the following, it is assumed that the ARM Software Development Toolkit V2.11a or later has been installed, that there are no default options for the APM (ARM Project Manager) tools, and that the directory “Bin\Config” of the ARM setup is empty (except “readme.txt”).

Building the Library

Copy the AT91 library (complete folder and sub-folder) onto your hard disk in a directory referred to in the following as <MyFolderAT91>.

Copy/move the files from the directory “<MyFolderAT91>\Template” into the directory “Template” of the ARM SDT setup.

Start the ARM Project Manager (APM).

Open the project “AT91_L16” in the directory “<MyFolderAT91>\Library” and build the 16-bit THUMB library.

Open the project “AT91_L32” in the directory “<MyFolderAT91>\Library” and build the 32-bit ARM library.

Creating a Project

Create a folder (labelled in the following as <MyProject>) in the directory “<MyFolderAT91>\Work”.

Open the ARM Project Manager.

AT91
ARM Thumb® Microcontrollers

Application Note

Rev. 1203A–11/98

Create a new project and select “EB01 Interworking Image” (or “DB01 Interworking Image” if applicable).

Locate the project in <MyProject>.

Name the project and press OK.

The “EB01 Interworking Image” has 4 variants:

- “EB01SramICE” to generate an application running in SRAM and debugged with the Embedded ICE.
- “EB01SramAngel” to generate an application running in SRAM and debugged with the Angel Debug Monitor.
- “EB01Flash” to generate an application running in Flash.
- “EB01FlashSymbol” to generate symbols corresponding to the “EB01Flash” binary image.

Identical variants are available in the “DB01 Interworking Image” template, but the target memory is SSRAM rather than SRAM.

Now add the required initialization files to the project that will start up the application depending on the variant environment. These files are in the directory <MyFolder\Library\Init> and their names are:

- “in_reset.s”
- “in_main.c”
- “in_eb01.s” or “in_db01.s” depending on the board targeted.

Add the application source files. The single constraint is that the main function of the project must be declared as:

```
int MainApplication ( void )
```

Build the project.

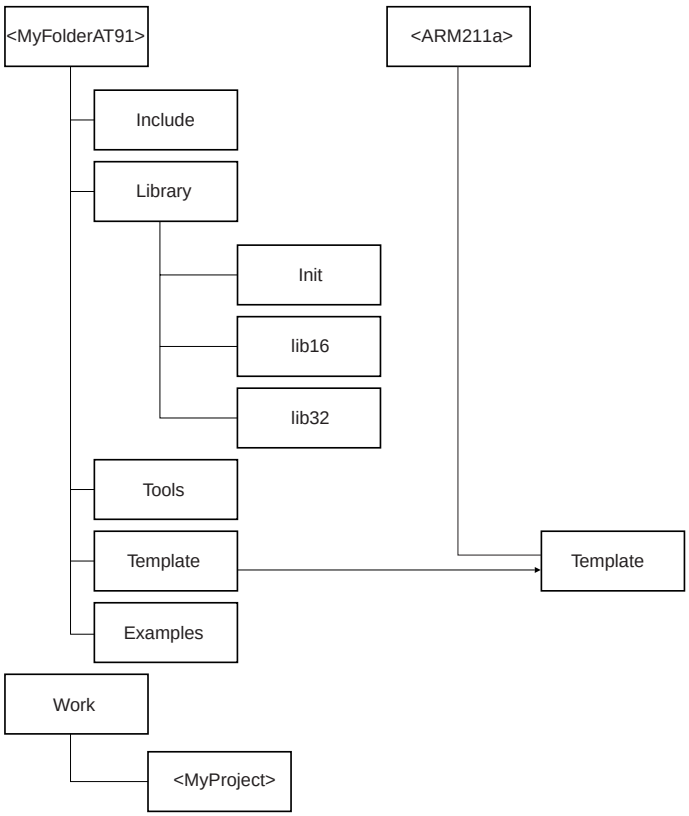
Start the debugger and configure it to communicate with the target using the Remote_A DLL (Add the driver if not already done).

Load the image conforming to the board and the debug system. For example, if the user is working in the SRAM of the AT91EB01 and the Embedded ICE, the “<MyProject>\EB01SramICE\<MyProject>.axf” image needs to be loaded.

AT91 Library Contents

The AT91 Library hierarchy is described in Figure 1 below.

Figure 1. AT91 Library Hierarchy



Notes: 1. The directories “lib16” and “lib32” in “<MyFolderAT91>\Library” are duplicated in the directories “40400_16” and 40400_32”. This is in order to support future devices.

2. The folder <MyFolderAT91> contains 3 dummy files: “asm_path.s”, “c_path.c” and “c_path.c”. These can be added to a project in order to define the path of <MyFolderAT91> whatever the path of the project file may be.

Include Directory

This directory contains:

- C header files with the extension ".h"
- Assembly include files with the extension ".inc"
- Assembly macro definition files with the extension ".mac"

There are two files, one C header file and one assembly include file, to describe each of the following:

- The ARM7TDMI™ core
- Each AT91 peripheral
- The AT91 internal memories
- The AT91EB01 evaluation boards
- The AT91DB01 development board

For example, the USART is described in C by the file "usart.h" and in assembly by "usart.inc".

Only one macro file is present: "irq.mac". It defines the "IRQ_ENTRY" and "IRQ_EXIT" macro used by the assembly interrupt handlers.

The File "std_c.h"

The file "std_c.h" defines some standard data types (unsigned char, unsigned short, unsigned integer), the default values for Boolean types (TRUE, FALSE), and the type of the peripherals user interface registers as:

```
#define at91_reg volatile unsigned int
```

This means that all registers of the AT91 are 32 bits wide and their values can change at any time. This last point avoids the C compiler optimizer effect when reading the AT91 registers.

The "std_c.h" file must be included before using one of the peripheral header files.

Peripheral Definitions

Each peripheral of the AT91 is described by a C structure composed of registers and an assembly list of offsets corresponding to registers. Thus modular software architecture working on any peripheral of the same type can operate on a peripheral identified only by its base address.

At the end of the definition file, the peripheral base addresses are defined with conditional compilation depending on the device being used.

The File "prior_irq.h"

This file defines default priority levels for each interrupt source. These default levels are used by the C function initializing the interrupt of each peripheral. In case the priority levels of interrupt sources need to be redefined, modify this file.

Library Directory

This directory contains C and assembly files showing examples of how to use the AT91 peripherals.

For the most part, the peripheral access functions are written in C and associated with an assembly file when the peripheral needs interrupt handling.

The name of the C file has the format "lib_<periph_name>.c". If an assembly file exists, the name has the format "irq_<periph_name>.s". For example, the USART accesses are shown in the file "lib_usart.c" and the associated interrupt handling in the file "irq_usart.s".

There are two project files, one to create an ARM library file, the other one to create the same library in Thumb. The object files and the resulting library file (".alf") are respectively in the directory "lib32" and "lib16".

The "Library" directory also contains the "Init" sub-directory. This sub-directory contains source files to be included by project and which give a boot sequence depending on the debug system being used and the target board.

Library Inclusion

The AT91 library uses a C source file inclusion system. Thus it is not necessary to declare a function prototype in another place than the function declaration. Using the "_REFERENCE" and "CORPS" labels performs this system.

Each function is declared with the "_REFERENCE" label and the body of each function is conditionally compiled if "CORPS" label is defined.

When the C library file is compiled, the "_REFERENCE" is removed and "CORPS" is declared. The complete functions are generated.

When another file needs to use the functions from a library file, it includes this C file by declaring "_REFERENCE" to be replaced by "extern" and by un-defining "CORPS". The result is only prototyping the functions.

The main advantage of this system is that it enables the source to be maintained. It is used for the library C functions, but can be generalized for all other C source files.

Tools Directory

This directory contains command files for the debugger and a binary file transmitter from the PC to a target board.

It also contains subdirectories:

- "FlashPgm" which contains a Flash downloader for the target boards
- "booteb01" which contains the EB01 boot sources
- "bootdb01" which contains the DB01 boot sources

The boot source files are provided as reference.

Command Files for the Debugger

It is useful to perform complex operations on the chip peripherals from the debugger. The most useful ones are:

- Chip reset using the watchdog to return in Reboot mode. This tests the boot sequence after having downloaded it.

- EBI configuration and remap command. For example, this allows access to external memories on virgin boards (first programming while out of production).

The command file “reset_wd” performs an internal reset by using the watchdog.

To run it, enter the following line in the command window of the debugger:

```
obey <MyFolderAT91>\Tools\reset_wd
```

The command files “ebi_eb01” and “ebi_db01” perform default EBI configuration followed by the remap command for the AT91EB01 and the AT91DB01 respectively . Values programmed are those used for the standard boot sequence.

To run it, enter one of the following lines in the command window of the debugger:

```
obey <MyFolderAT91>\Tools\ebi_eb01
obey <MyFolderAT91>\Tools\ebi_db01
```

One of these two files may be edited to be adapted to the application board devices.

Binary Transmitter

The AT91EB01 standard boot can run an SRAM downloader from a serial port. In this case, the data received are directly saved in SRAM at address 0x0200 0000 and then the control is given at this address (refer to the AT91EB01 User Guide for a complete sequence description if you need to use this feature). Note that the same feature is provided by the AT91DB01 at the address 0x0010 0000.

As data are directly stored in the SRAM, they have to be sent in a binary format. This is not possible from the debugger nor from other standard PC tools. For this reason, users are provided with the PC program “Bincom.exe” which enables transmission of a binary image on a programmable serial COM at a programmable speed.

Flash Downloader

The AT91EB01 has one 64K bytes x 16 Flash device.

The AT91DB01 has two 256K bytes x 8 Flash devices, emulating a 16-bit device.

The subdirectory “FlashPgm” contains a Flash downloader project. It supports one variant for each target board.

This Flash downloader is a standard application running on the target and using the semihosting feature to read the image file to be programmed on the host hard disk drive. It must have as arguments the name of this file and the address where programming should begin.

For example, to program the project “led_blink” in the AT91EB01 Flash, enter the following lines in the command window:

```
$top_of_memory=0x02200000
load <MyFolderAT91>\Tools\FlashPgm\At91eb01\
    FlashPgm.axf
<MyFolderAT91>\Exaples\led_blink\Binary\
    led_blink.axf 0x01100000
```

Projects and Variants Description

The AT91 Library proposes two project templates:

- EB01 Interworking Image
- DB01 Interworking Image

These project templates can be chosen when you create a project from the ARM Project Manager in the “Type” field of the “New Project” window. For this, you need to copy the delivered project template (files “at91eb01.apj” and “at91db01.apj”) from the directory “<MyFolderAT91>\Template” to the “Template” folder of the ARM Software Toolkit.

Both templates are made from the standard template “THUMB/ARM Interworking Image” and are adapted for the AT91-based board use.

Variants and Link Addresses

The templates propose one variant per target memory available. Each variant defines a link address of the image to be generated.

The SRAM (on AT91EB01) or SSRAM (on AT91DB01) variants depend on the debug systems used, because the Angel Debug Monitor occupies an address space in these memories.

The template “EB01 Interworking Image” proposes the following variants:

Table 1.

Target Memory	Debug System	Variant to Use	Link Address
SRAM	Angel	EB01SramAngel	0x0201 8000
SRAM	ICE	EB01SramICE	0x0200 0000
Flash	-	EB01Flash	0x0100 0000

The template “DB01 Interworking Image” proposes the following variants:

Table 2.

Target Memory	Debug System	Variant to Use	Link Address
SSRAM	Angel	DB01SramAngel	0x0011 8000
SSRAM	ICE	DB01SramICE	0x0010 0000
Flash	-	DB01Flash	0x0200 0000

The Flash variant generates a binary image. In this case, another template (with name finishing by “FlashSymbol”) generates the corresponding symbol table.

Initialization Labels

The initialization sequence defined in the directory “<MyFolderAT91>\Library\Init” depends on the target board and on the debug system used. These codes are conditionally assembled/compiled depending on labels defined by the template and the variants.

Target Boards

The two target boards supported and their main features are:

- **The AT91EB01**
  - Contains a Flash connected at Chip Select 0.
  - Chip Select 1 drives a Static RAM device of up to 4 Mbits depending on the version of the board.
  - A standard boot sequence is programmed in the lower 64K bytes Flash page, which is write-protected. It is executed only if SW1 is on the “LOWER_MEM” position.
  - The user can download his own boot sequence in the upper 64K bytes Flash page and execute it by switching SW1 to the “UPPER MEM” position and then by generating a reset.
- **The AT91DB01**
  - Contains an EPROM connected at Chip Select 0.
  - Chip Select 1 drives a Flash device.
  - The boot sequence branches to the Flash start address if LK9 is on.
  - The user cannot test his boot sequence in any other way than by programming a new Boot ROM device.
  - The main advantage of this board is that the user is able to emulate internal memory extensions with an external Synchronous Static RAM and to connect a Logic Analyzer to the internal busses.
  - The applications are debugged in the SSRAM address space.

Debug System

The boot sequence provided with the AT91 Library supports three debug systems:

- **The Angel Debug Monitor**
  - EBI is configured and Angel itself performs the remap.
  - Angel’s code is stored in Flash and is copied into the SRAM at startup. Additional memory is required to store its data.
  - The undefined Instruction exception is used to handle breakpoints.
  - Communications with the debugger requires the USART 0.
  - The USART driver uses the Advanced Interrupt Controller.
  - ARM interrupt vectors are initialized.
  - Stacks are defined. Only the User/System stack can be redefined by the user. Other stacks must not be modified unless Angel operations are disturbed.
- **The Embedded ICE**
  - No address space is needed. Only the semihosting features needs some stack space while operating.
  - If a valid boot sequence has been run before the core is stopped by the ICE, the EBI is configured and the remap is performed.
  - The ARM interrupt vector is not initialized to support vectoring features.
  - The stacks are undefined with the exception of the User/System one which is set to “$top_of_memory”.
- **Debug on Binary**
  - The application is programmed in non-volatile memory. It must be generated with a binary format in order to be programmed.
  - A complete startup sequence must be run. This means EBI configuration, stacks and vectors setting and C data initialization must be performed.
  - The debug information is not included in the binary image and has to be loaded separately from the Symbol variant.

Device Labels

The project templates define a device identification label, **AT91M40400**. It is defined for both C compilers and ARM/Thumb assembler. This will allow support for future devices in the AT91 family.

Target Board Labels

The project templates define a target board identification label.

For the “EB01 Interworking Image” project template, this label is **AT91EB01**.

For the “DB01 Interworking Image” project template, this label is **AT91DB01**.

Debug System Labels

The AT91-based board project templates define 3 variants each depending on the debug system. For each of these variants, a debug system identification label is defined.

The label **AT91_DEBUG_NONE** is defined when the user wants to generate a final binary version. This is the case for the “EB01Flash” and “DB01Flash” variants. To make sure symbols are corresponding, it is also defined for the variants “EB01FlashSymbol” and “DB01FlashSymbol”.

The label **AT91_DEBUG_ICE** is defined when the user wants to use an Embedded ICE-based debug system. This is the case for the “EB01SramIce” variant of the “EB01 Interworking Image” or “DB01SsramIce” variant of the “DB01 Interworking Image”.

The label **AT91_DEBUG_ANGEL** is defined when the user wants to use an Angel-based debug system. This is the case for the “EB01SramAngel” variant of the “EB01 Interworking Image” or “DB01SsramAngel” variant of the “DB01 Interworking Image”.

Interrupt Handling

Scope

Interrupt Entry and Exit Macros

Interrupt handlers are given as an example. All of them use the IRQ_ENTRY and IRQ_EXIT routines defined in the file “irq.mac” stored in the directory <MyFolderAT91>\Include.

The sequence defined by <IRQ_ENTRY> is:

- Adjust the Link Register.
- Save it in the IRQ stack.
- Save the SPSR (Saved Program Status Register) and r0 in the IRQ stack.
- Switch to System Mode and Clear I.
- Save the registers used(at least r0 - r3 and r12) and r14(user).

The sequence defined by <IRQ_EXIT> is:

- Restore the registers used and r14(user).
- Switch back to IRQ mode and Set I.
- Perform a write to the End of Interrupt Command Register.
- Restore r0 and SPSR.
- Restore Link Register directly in the PC.

Between “IRQ_ENTRY” and “IRQ_EXIT”, a C handler is called. Its address is saved by the C function enabling the interrupt. Depending on the interrupt to be handled, the search for the corresponding handler can be performed in assembler.

Switch to System Mode

The assembler interrupt handler calls a C handler. At this moment, the Link Register is used to save the return address to the assembler handler. However, if a nested interrupt occurs (this may occur because of priority management), the Link Register is overwritten by the save of the Program Counter.

If the user wishes to support nested interrupts, a switch to System Mode must be made. The System Mode defined by the ARM processor uses the same register bank as the User Mode. Thus the Link Register must be saved because it may be the return address of a Branch with Link instruction. Therefore the assembler handler saves the register r0 in order to be used during “IRQ_EXIT” to modify the CPSR (Current Program Status Register).

The switch to System Mode enables the IRQ stack size to be known. This is equivalent to the number of priority levels of the AIC(8) multiplied by the number of words saved by the assembler handler(3). The User Stack is limited to 3 x 4 x 8 = 96 bytes.

Peripheral Interrupts Handling

The interrupt management at the AIC level is described in the file “lib_aic.c”. Interrupt sources can be enabled with the C function “enable_interrupt” and disabled with the C function “disable_interrupt”.

USART Interrupt Handling

The USART interrupt management is described in the file “irq_usart.s”. This defines an assembler interrupt handler for each USART. These interrupt handlers allow management of three different types of interrupt from each USART: error, receive and transmit. A mask and a handler pointer stored in the table “USARTHandlerTable” identify each.

When an interrupt occurs, the Status Register of the USART is read and then masked with the Interrupt Mask Register. The result is then masked with each of the three masks read from the table. When a result is not zero, the corresponding handler is executed.

Masks and handlers are definable with the C function “enable_usart_irq”, defined in the file “lib_usart.c”.

PIO Interrupt Handling

The PIO interrupt management is described in the file “irq_pio.s”. This defines the assembler interrupt handler of

the PIO controller. The C handlers for each PIO line are stored in the table “PIOHandlerTable”. The assembly interrupt handler searches for each PIO line interrupt. If present, the corresponding handler is called with the PIO Controller base address as argument and a mask identifying the PIO line.

The PIO lines are looked for byte by byte to increase latency time on the highest bits.

The C PIO interrupt handlers are definable with the C function “enable_pio_interrupt” defined in the file “lib_pio.c”.

Timer Counter Interrupt Handling

The Timer Counter interrupt management is described in the file “irq_tc.s”. This defines one assembler interrupt handler for each Timer Counter channel that directly calls a C handler.

The C handler addresses are stored in the table “TCHandlerTable” and are definable with the C function “enable_timer_irq” defined in the file “lib_tc.c”.