

AT91SAM7 Serial Communications

Author:

**James P. Lynch
Grand Island, New York, USA
June 22, 2008**

Preface

I've been extremely gratified by the positive response I've received via email concerning my tutorials. Many readers have asked me for help with serial communications and interrupts on ARM microcontrollers. These requests have resulted in this tutorial "**AT91SAM7 Serial Communications**" which delves into interrupt-driven and DMA-based serial communications on Atmel AT91SAM7 microcontrollers. Readers will find helpful and detailed information on setting up a USART to be driven by character-by-character interrupts or DMA block transfer techniques.

This tutorial follows in the footsteps of my Atmel tutorial "**Using Open Source Tools for AT91SAM7 Cross Development**" and I'm assuming that you have read that one first. There is an appendix herein that shows how to set up an Eclipse project; there have been a few changes in the Eclipse and GNU Tool Chain since I authored the Atmel tutorial.

To the many generous people have emailed me with helpful suggestions and support, I'm very appreciative of their kindness.

Jim Lynch

Table of Contents

Preface.....	2
Introduction.....	5
Universal Synchronous Asynchronous Receiver Transmitter.....	6
Theory of Operation.....	7
Baud Rate Generation.....	8
Steps to Make USART0 Ready-to-Run.....	10
Turn on the USART0 Peripheral Clock.....	10
Give the USART0 Peripheral Control of the Pins.....	13
Set Up the USART0 Registers.....	16
Control Register - Reset then Disable the Receiver/Transmitter.....	16
Mode Register – Set up Character Format, etc.....	18
Interrupt Enable Register – Enable Desired USART0 Interrupt	20
Interrupt Disable Register – Disable Desired USART0 Interrupt	21
Baud Rate Generator Register – enter baud rate clock divider.....	22
Set Up the USART0 Registers that are not Used.....	23
Setting Up the Advanced Interrupt Controller (AIC).....	24
Final Preparations for USART0 Interrupt Processing.....	28
Assembly Language Part of the IRQ Handler.....	29
Designing the USART0 IRQ Handler.....	30
Flowchart – USART0 Interrupt Handler.....	31
Project Listings – Interrupt Version.....	32
AT91SAM7X256.H.....	32
BOARD.H.....	33
CRT.S.....	33
ISRSUPPORT.C.....	38
Lowlevelinit.c.....	39
Main.c.....	41
Usart0_isr.c.....	42
Usart0_Setup.c.....	43
Demo_sam7x256.cmd.....	50
Makefile.....	52
Openocd_program.cfg.....	55
Openocd.cfg.....	56
Script.ocd.....	56
Building the Project.....	57
Adding an LED to the Olimex SAM7-EX256 Board.....	57
Programming the Sample Application into Flash.....	58
Testing the Interrupt Driven Application.....	59
Direct Memory Access.....	62
USART0 DMA Registers.....	62
USART0 PDC Receive Pointer Register.....	63
USART0 PDC Receive Counter Register.....	63
USART0 PDC Transmit Pointer Register.....	64
USART0 PDC Transmit Counter Register.....	64
USART0 PDC Receive Next Pointer Register.....	65
USART0 PDC Receive Next Counter Register.....	65
USART0 PDC Transmit Next Pointer Register.....	66
USART0 PDC Transmit Next Counter Register.....	67
USART0 PDC Transfer Control Register.....	67

Set Up for DMA Interrupts.....	68
DMA Interrupt Handler.....	69
Project Listings – DMA Version.....	71
USART0_SETUP.C.....	71
USART0_ISR.C.....	79
Building the DMA Application.....	80
Other Possibilities.....	81
About the Author.....	82
Appendix	83
Download Yagarto Components.....	83
Install the YAGARTO Components.....	84
Install OpenOCD.....	84
Install Eclipse IDE.....	84
Install YAGARTO GNU ARM Tool Chain.....	84
Install YAGARTO Tools.....	84
Install the JTAG Device Drivers.....	84
Start Up Eclipse.....	85
Create an Eclipse Standard C Project.....	85
Import the Sample Project Files.....	87
Build the Project.....	89
Set Up a Second Make Target for Flash Programming.....	90

Introduction

Ten billion ARM microprocessor chips have been shipped to date. The ARM chips have a 75% market share in the 32-bit embedded marketplace. These ARM chips are in most of the cell phones including the new Apple iPhones. ARM Holdings is a British chip design company; other manufacturers such as Atmel, ST, Texas Instruments, NXP, Intel, etc. actually fabricate the chips. Since the architecture is common, the various chip vendors battle it out in terms of on chip memory and peripherals.

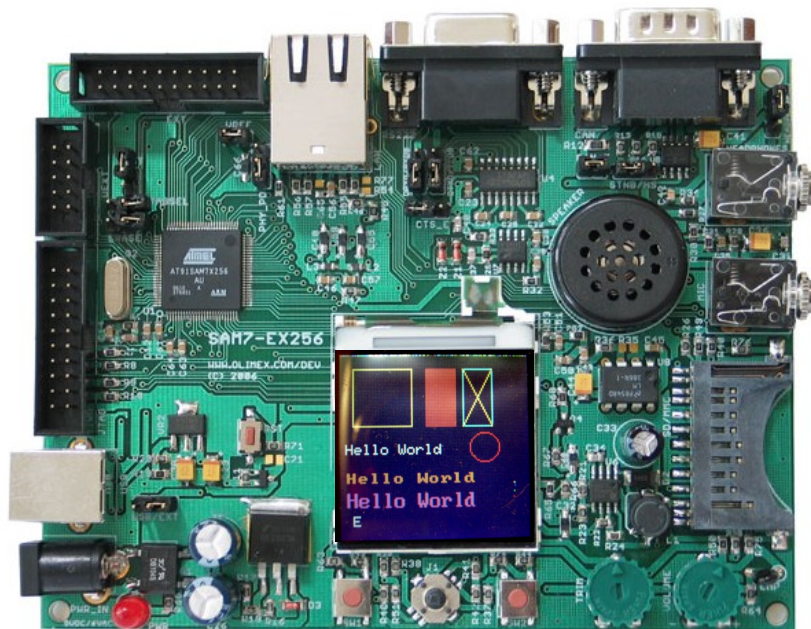
Atmel has a very nice line of ARM7 embedded controller chips called the AT91SAM7 family. The AT91SAM7X256 chip has three serial RS-232 peripherals. There are two USARTS (**U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter) called USART0 and USART1. The only difference between the two USART peripherals is that USART1 has more modem control signals. The third serial peripheral is called the Debug Unit which is a very simple UART (**U**niversal **A**synchronous **R**eceiver **T**ransmitter) with just a two-wire interface (send and receive).

Using the USART serial interface in interrupt mode is difficult for the novice; using the USART in DMA mode (**D**irect **M**emory **A**ccess) is even more mysterious. In this tutorial, I will cover in great detail design of a serial USART application that is interrupt-driven. I will also demonstrate the same application built with DMA block transfer techniques which is much more efficient.

In each case, the application will accept incoming characters and retransmit them back to the source. The interesting design variant is that the application will collect 10 incoming characters and retransmit the 10 characters back to the source only after the tenth incoming character has been received. This application can be tested with either the Open Source **RealTerm** utility or the Windows utility **Hyper Terminal** and a standard 9-pin serial cable.

The serial interface applications are built with the Eclipse IDE and the YAGARTO GNU ARM toolchain. Readers should download and read my tutorial "**Using Open Source Tools for AT91SAM7 Cross Development**". This document is hosted at the Atmel web site and can be downloaded from here:

http://www.atmel.com/dyn/resources/prod_documents/atmel_tutorial_source.zip



The completed serial communications application was tested on an Olimex SAM7-EX256 evaluation board which can be purchased from Spark Fun Electronics and other outlets for around \$120 (US funds).

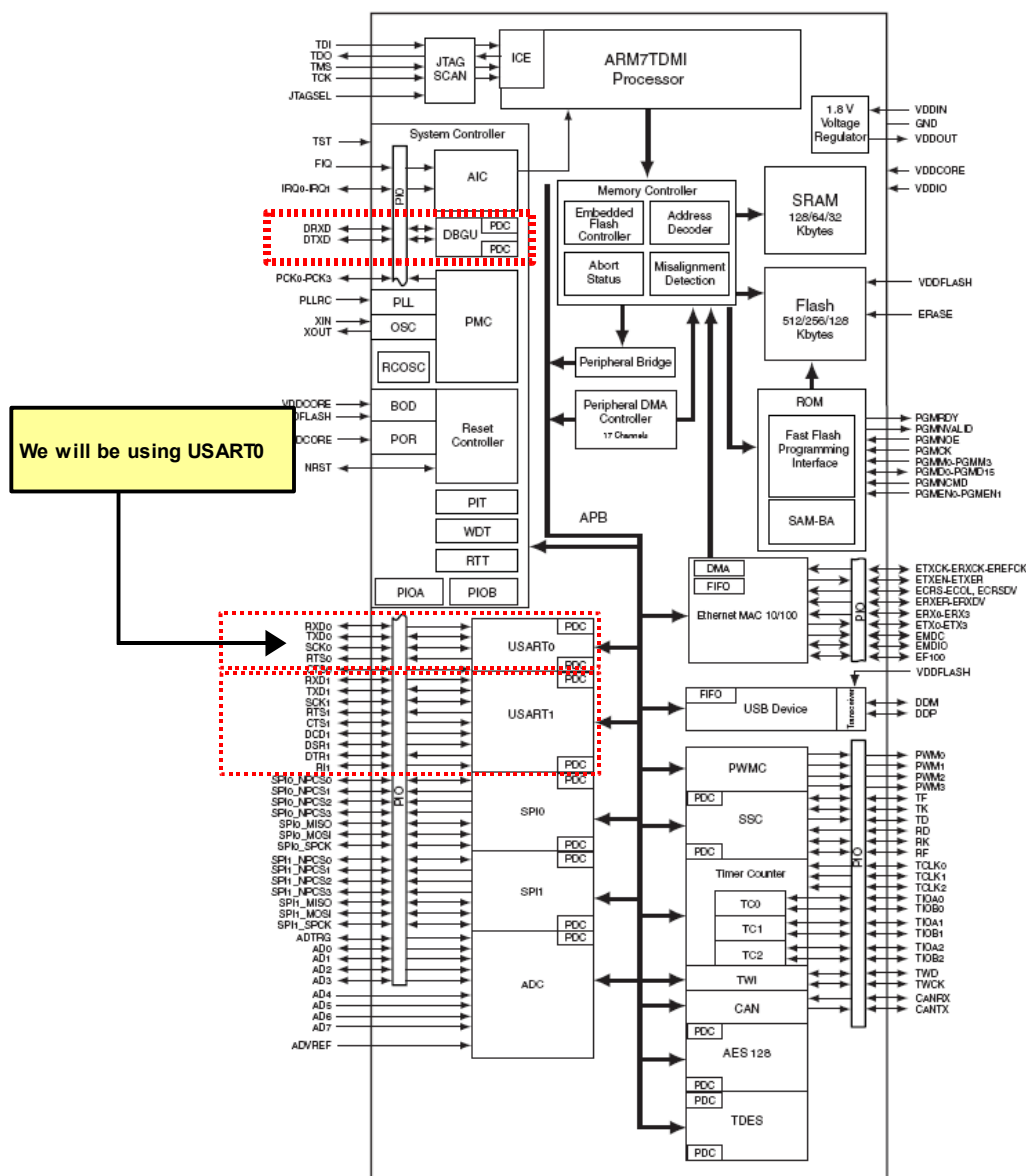
The Olimex board used here employs the Atmel **AT91SAM7X256** chip, which includes an Ethernet port, a USB interface, 256k of FLASH, 64k of RAM, and a large number of other peripherals.

Universal Synchronous Asynchronous Receiver Transmitter

The subject of USARTS and serial communications is so complex that one could write a book on it; indeed there are numerous books available. A particularly good reference is **“Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems”** by Jan Axelson, available from Amazon.

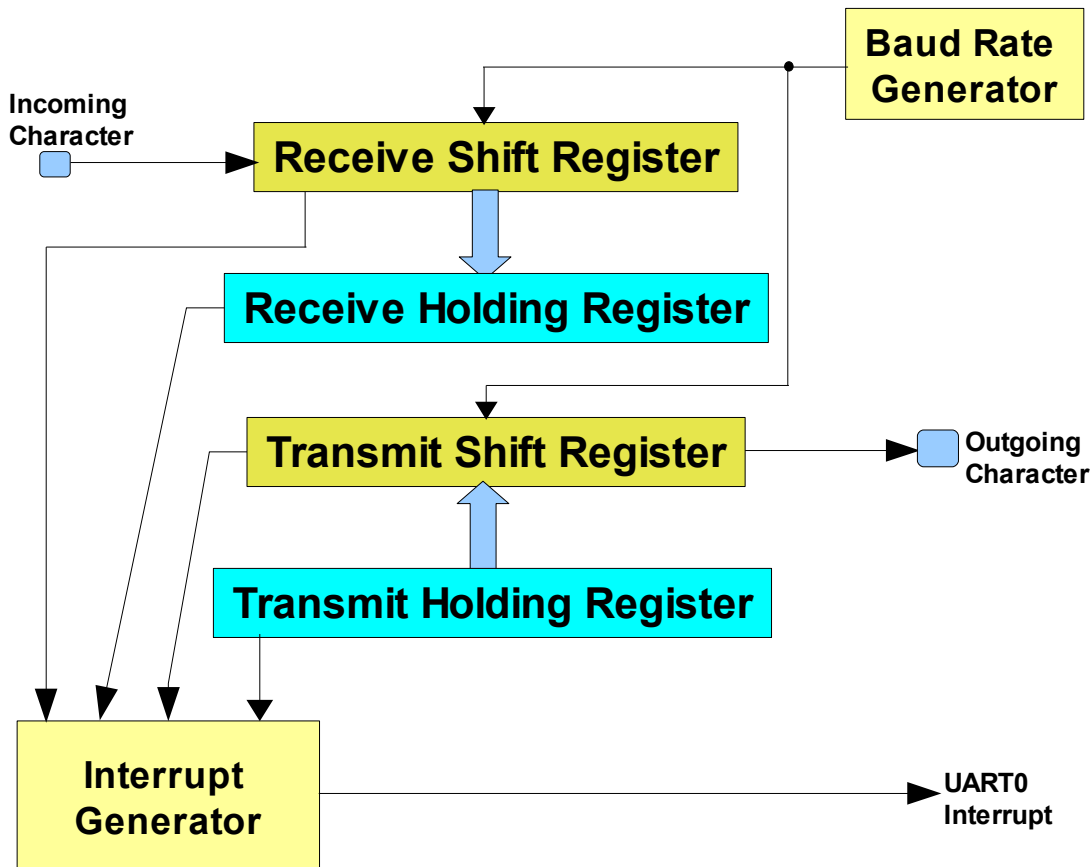
Atmel's USART peripheral is extremely sophisticated and can operate in many different modes, such as asynchronous, synchronous, RS-485, Smart Card protocol and Infra-red protocol. To keep things as simple as possible, we will be operating USART0 in “asynchronous” mode at 9600 baud with 1 start bit, 8 data bits, 1 stop bit, and no parity to be compatible with your PC's COM1 serial port. The Atmel diagram below shows the part of the AT91SAM7X256 chip we will be using.

Atmel AT91SAM7X256 - Serial Interfaces



Theory of Operation

While the description of this USART peripheral encompasses 45 pages in the Atmel data sheet, its basic operation is fairly simple. The primary thing to remember is that USART0 has just one interrupt. When it occurs, you have to read the Channel Status Register within your interrupt service routine to determine exactly what happened (Receiver Ready or Transmitter Empty, for example). There is a Receive Holding Register (32 bits) that holds the incoming character and a Transmit Holding Register (32 bits) where you can insert the next character to be transmitted. The following diagram illustrates these points.



To receive characters, you wait until there's a "Receiver Ready" (RXRDY) interrupt and then read the character out of the Receive Holding Register in the interrupt service routine. You have to do this fairly quickly if there's a stream of incoming characters expected.

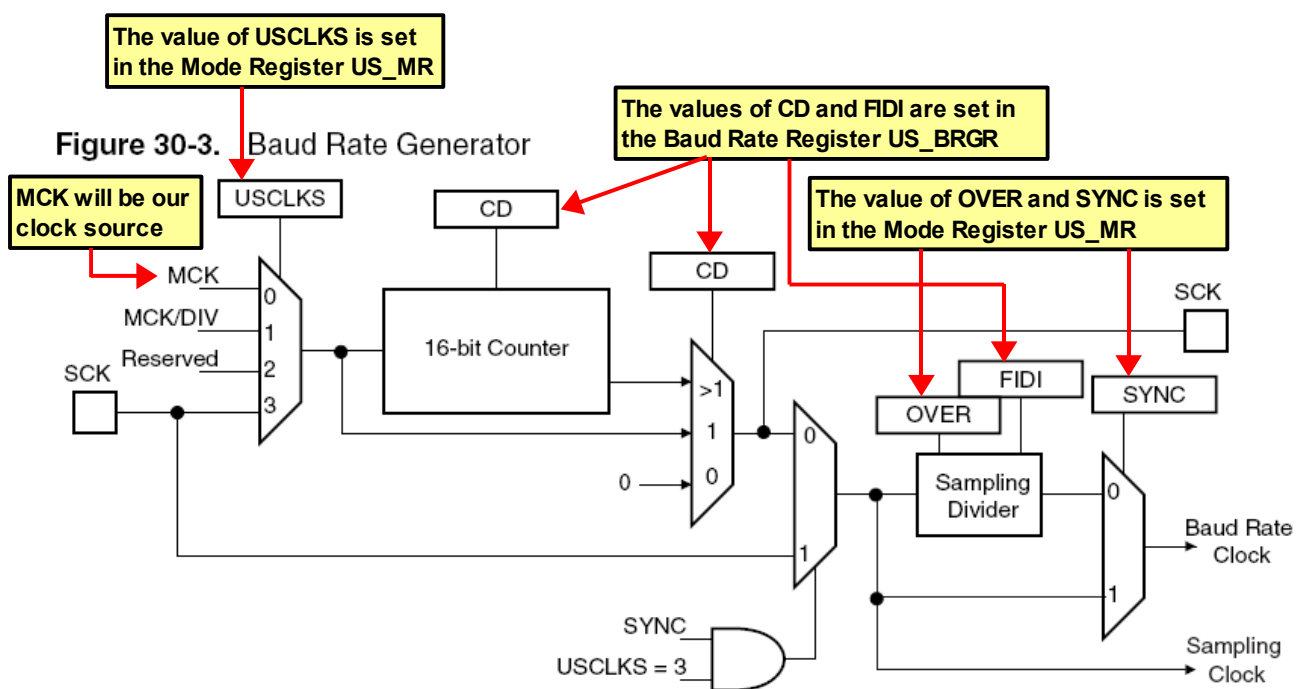
In the case of transmit, you have two choices. There are two basic interrupts available for transmit operations. First is the Transmitter Ready (**TXRDY**) interrupt. This occurs when you have placed an outgoing character into the Transmit Holding Register and it has immediately dropped down into the transmit shift register. This is a indication that you can now insert another outgoing character into the Transmit Holding Register even though the previous character may still be clocking out of the shift register. The second choice is the Transmitter Empty (**TXEMPTY**) interrupt. This is signaled when both the Transmit Holding Register and the transmit shift register are empty. In my view, this one indicates that you are really done transmitting – all bytes have been sent!

My preference is to use the **TXEMPTY** interrupt and this is what I will demonstrate in the upcoming examples.

There are many other USART interrupts available. The USART0 can indicate error conditions, such as overrun, framing error, break detection, and parity error. You can enable these interrupts and scan for them in the interrupt service routine. These error interrupts are latched in the Channel Status Register and you must clear them by setting the “Reset Status Bits” field in the USART0 Control Register. This error handling is not demonstrated in the tutorial examples.

Baud Rate Generation

In synchronous mode, the baud rate clock comes in over the SCK pin from the transmitting source. We are demonstrating asynchronous mode where we must generate the baud rate clock internally. The following diagram from the Atmel data sheet shows the logic of baud rate generation.



The Atmel data sheet gives the following formula for the baud rate (page 302).

$$\text{Baudrate} = \frac{\text{SelectedClock}}{(8(2 - \text{OVER}) \text{CD})}$$

First, we decide to use MCK as our clock source. To determine value of MCK, we have to refer to how the Olimex SAM7-EX256 board was initialized. The Olimex SAM7-EX256 board has a 18,432,000 Hz crystal oscillator.

MAINCK = 18432000 Hz (crystal frequency, from Olimex schematic)
 DIV = 14 (set up in lowlevelinit.c)
 MUL = 72 (set up in lowlevelinit.c)

$$\text{PLLCK} = (\text{MAINCK} / \text{DIV}) * (\text{MUL} + 1) = 18432000 / 14 * (72 + 1) \quad (\text{phase lock loop clock})$$

PLLCLK = 1316571 * 73 = 96109683 Hz
MCK = PLLCLK / 2 = 96109683 / 2 = 48054841 Hz (main clock MCK)

Baud Rate (asynchronous mode) = MCK / (8(2 - OVER)CD)

MCK = 48054841 hz (set bit field USCLKS = 00 in USART Mode Register US_MR to select "MCK only")

OVER = 0 (bit 19 of the USART Mode Register US_MR)
CD = divisor (USART Baud Rate Generator Register US_BRGR)
baudrate = 9600 (desired)

a little algebra:
$$\text{BaudRate} = \frac{48054841}{(8(2 - 0)CD)} = \frac{48054841}{16(CD)}$$

Plugging in 9600 for the baud rate and rearranging the equation will give an equation for the value of the CD counter.

$$CD = \frac{48054841}{9600(16)} = \frac{48054841}{153600} = 312.857037$$

CD = 313 (round up)

check the actual baud rate:
$$\text{BaudRate} = \frac{48054841}{(8(2 - 0)313)} = \frac{48054841}{5008} = 9595.6$$

what's the error:

$$\text{Error} = 1 - \frac{\text{desired baudrate}}{\text{actual baudrate}} = 1 - \frac{9600}{9595.6} = 1 - 1.00045854 = -.0004585$$

Error = -.0004585 (that's not very much!)

It should be very easy to calculate the required constants for other standard baud rates. Make a mental note that we will have to set up the following baudrate constants in the USART0 registers to get 9600 baud.

Constant	USART0 Register	Value	Description
USCLKS	Mode Register	00	Select MCK as the baud rate clock source
OVER	Mode Register	0	Select 16x over-sampling
CD	Baud Rate Generator Register	313	Clock divisor to get 9600 baud
SYNC	Mode Register	0	Select "asynchronous" mode
FIDI	FI DI Ratio Register	0	Only used in ISO7816 mode

Steps to Make USART0 Ready-to-Run

One of the reasons why all these ARM embedded controller chips are so affordable is that the chips “share” the external pins between peripherals or standard I/O points. This keeps the total number of pins on the AT91SAM7X256 package below 100.

The AT91SAM7X256 has 62 external pins that can be used as I/O ports or assigned to support the various peripherals. Page 32 of the AT91SAMX256 Data Sheet states that “At Reset, all I/O lines are automatically configured as input with the programmable pull-up enabled ...”.

The USART0 peripheral has five possible external pins (RXD0, TXD0, SCK0, RTS0, CTS0). On the edge of the chip package, these pins are labeled PA0, PA1, PA2, PA3, and PA4.

But these five pins could also be used as I/O ports and some of the same pins can be used as part of the second SPI peripheral. How do we sort this out?

Turn on the USART0 Peripheral Clock

The very first thing to do is to turn on the USART0's peripheral clock; forgetting to do this guarantees failure. In an attempt to reduce the chip's power consumption, every on-chip peripheral's clock source is turned off at power-on reset. Refer to the chapter in the Data Sheet titled “Power Management Controller (PMC)” to learn how turn the peripheral clocks “on”.

The specific register to turn on the peripheral clocks is the PMC Peripheral Clock Enable Register, as shown below (from page 192 of the Data Sheet).

25.9.4 PMC Peripheral Clock Enable Register

Register Name: PMC_PCER

Access Type: Write-only

31	30	29	28	27	26	25	24
PID31	PID30	PID29	PID28	PID27	PID26	PID25	PID24
23	22	21	20	19	18	17	16
PID23	PID22	PID21	PID20	PID19	PID18	PID17	PID16
15	14	13	12	11	10	9	8
PID15	PID14	PID13	PID12	PID11	PID10	PID9	PID8
7	6	5	4	3	2	1	0
PID7	PID6	PID5	PID4	PID3	PID2	-	-

• **PIDx: Peripheral Clock x Enable**

0 = No effect.

1 = Enables the corresponding peripheral clock.

The novice reader is probably asking at this point “which one of those PIDxx bits above is USART0”? The answer is Chapter 10, page 31 in the Data Sheet, in a table called “Peripheral Identifiers.”

Table 10-1. Peripheral Identifiers

Peripheral ID	Peripheral Mnemonic	Peripheral Name	External Interrupt
0	AIC	Advanced Interrupt Controller	FIQ
1	SYSC ⁽¹⁾	System	
2	PIOA	Parallel I/O Controller A	
3	PIOB	Parallel I/O Controller B	
4	SPI0	Serial Peripheral Interface 0	
5	SPI1	Serial Peripheral Interface 1	

Table 10-1. Peripheral Identifiers

Peripheral ID	Peripheral Mnemonic	Peripheral Name	External Interrupt
6	US0	USART0	
7	US1	USART 1	
8	SSC	Synchronous Serial Controller	
9	TWI	Two-wire Interface	
10	PWMC	Pulse Width Modulation Controller	
11	UDP	USB device Port	
12	TC0	Timer/Counter 0	
13	TC1	Timer/Counter 1	
14	TC2	Timer/Counter 2	
15	CAN	CAN Controller	
16	EMAC	Ethernet MAC	
17	ADC ⁽¹⁾	Analog-to Digital Converter	
18	AES	Advanced Encryption Standard 128-bit	
19	TDES	Triple Data Encryption Standard	
20-29	Reserved		
30	AIC	Advanced Interrupt Controller	IRQ0
31	AIC	Advanced Interrupt Controller	IRQ1

There it is, in the table above, listed as **US0**, bit 6. Therefore, we need to set bit 6 of the PMC Peripheral Clock Enable Register (PCER) to logic “one” (all the other bits will be undisturbed). To do this, we’ll need a pointer to this register. If we include the standard Atmel include file (**at91sam7x256.h**) in our project, this is fairly easy to do. Consider the following two lines of C code.

```
volatile AT91PS_PMC pPMC = AT91C_BASE_PMC; // pointer to PMC data structure
pPMC->PMC_PCER = (1<<AT91C_ID_US0); // enable usart0 peripheral clock
```

Just this one time, let’s analyze in detail the above two lines. The include file (at91sam7x256.h) has a C structure that describes the PMC registers. The structure **AT91S_PMC** consists of twenty eight 32-bit integers, each representing a register of the Power Management Controller (PMC) set.

```
typedef struct AT91S_PMC {
    AT91_REG PMC_SCER; // System Clock Enable Register
    AT91_REG PMC_SCDR; // System Clock Disable Register
    AT91_REG PMC_SCSR; // System Clock Status Register
    AT91_REG Reserved0[1]; //
    AT91_REG PMC_PCER; // Peripheral Clock Enable Register
    AT91_REG PMC_PCDR; // Peripheral Clock Disable Register
    AT91_REG PMC_PCSR; // Peripheral Clock Status Register
    AT91_REG Reserved1[1]; //
    AT91_REG PMC_MOR; // Main Oscillator Register
    AT91_REG PMC_MCFR; // Main Clock Frequency Register
    AT91_REG Reserved2[1]; //
    AT91_REG PMC_PLLR; // PLL Register
    AT91_REG PMC_MCKR; // Master Clock Register
    AT91_REG Reserved3[3]; //
    AT91_REG PMC_PCKR[4]; // Programmable Clock Register
    AT91_REG Reserved4[4]; //
    AT91_REG PMC_IER; // Interrupt Enable Register
    AT91_REG PMC_IDR; // Interrupt Disable Register
    AT91_REG PMC_SR; // Status Register
    AT91_REG PMC_IMR; // Interrupt Mask Register
} AT91S_PMC, *AT91PS_PMC;
```

Note in the C structure above, we also have a pointer to the PMC structure called ***AT91PS_PMC**.

Now we need to assign the actual physical base address of the PMC structure to the pointer, the handy Atmel include file gives us that too!

```
// *****  
//          BASE ADDRESS DEFINITIONS FOR AT91SAM7X256  
// *****  
#define AT91C_BASE_SYS      ((AT91PS_SYS) 0xFFFFF000) // (SYS) Base Address  
#define AT91C_BASE_AIC      ((AT91PS_AIC) 0xFFFFF000) // (AIC) Base Address  
#define AT91C_BASE_PDC_DBGU ((AT91PS_PDC) 0xFFFFF300) // (PDC_DBGU) Base Address  
#define AT91C_BASE_DBGU     ((AT91PS_DBGU) 0xFFFFF200) // (DBGU) Base Address  
#define AT91C_BASE_PIOA     ((AT91PS_PIO) 0xFFFFF400) // (PIOA) Base Address  
#define AT91C_BASE_PIOB     ((AT91PS_PIO) 0xFFFFF600) // (PIOB) Base Address  
#define AT91C_BASE_CKGR     ((AT91PS_CKGR) 0xFFFFFC20) // (CKGR) Base Address  
#define AT91C_BASE_PMC      ((AT91PS_PMC) 0xFFFFFC00) // (PMC) Base Address  
#define AT91C_BASE_RSTC     ((AT91PS_RSTC) 0xFFFFFD00) // (RSTC) Base Address  
#define AT91C_BASE_RTTC     ((AT91PS_RTTC) 0xFFFFFD20) // (RTTC) Base Address  
..... and so on .....
```

Note above that the #define constant AT91C_BASE_PMC identifies the physical location of the base address of the PMC registers to be 0xFFFFFC00. Therefore, the first line above creates a pointer pPMC to the PMC structure and assigns the base address to the pointer. It's a good idea to set the "volatile" attribute to prevent the compiler from attempting any optimization on this statement.

```
volatile AT91PS_PMC    pPMC = AT91C_BASE_PMC;
```

Two final bits of the puzzle remain. The PMC Peripheral Clock Enable Register is an element of the AT91S_PMC data structure; if pPMC is a pointer to the PMC structure, then the contents of the "Peripheral Clock Enable Register" itself can be expressed by:

```
pPMC->PMC_PCER = value;
```

Now we are almost done. If we can set the value of the "Peripheral Clock Enable Register" by a C statement as shown above, how do we set the bit associated with USART0? We know that it is bit 6, so the following statement below will work. Remember that a binary logic "one" will set a bit, binary logic "zero" will have no effect (in other words, it won't disturb the other bits).

```
pPMC->PMC_PCER = 0x00000040;
```

True, the above statement is OK, but coding purists will say that this construct involves a "magic constant". This is frowned upon because it is not clear where the constant came from. Once again, the Atmel include file has a handy constant to make things a bit clearer.

```
#define AT91C_ID_US0      ((unsigned int) 6) // USART 0
```

Now it becomes easy to access any of the PMC registers using our pPMC “pointer to a structure”. The following statement will set bit 6 of the PMC Peripheral Clock Enable Register. There are no “magic constants” and it is clear that we are setting bit 6 of the register and not disturbing any of the other bits.

```
pPMC->PMC_PCER = (1<<AT91C_ID_US0);
```

To repeat, the following two C statements will turn on the peripheral clock of USART0.

```
volatile AT91PS_PMC pPMC = AT91C_BASE_PMC;    // pointer to PMC data structure
pPMC->PMC_PCER = (1<<AT91C_ID_US0);    // enable usart0 peripheral clock
```

For the rest of this tutorial, the techniques described above will be used without any additional explanation.

Give the USART0 Peripheral Control of the Pins

We mentioned earlier that the AT91SAM7X256 boots up at reset with all pins set as I/O in input mode. So we need to specify that pins PA0, PA1, PA2, PA3, and PA4 are to be connected to the peripheral USART0.

The Atmel include file has a C structure for the PIO controller registers.

```
typedef struct _AT91S_PIO {
    AT91_REG PIO_PER;        // PIO Enable Register
    AT91_REG PIO_PDR;        // PIO Disable Register
    AT91_REG PIO_PSR;        // PIO Status Register
    AT91_REG Reserved0[1];   //
    AT91_REG PIO_OER;        // Output Enable Register
    AT91_REG PIO_ODR;        // Output Disable Register
    AT91_REG PIO_OSR;        // Output Status Register
    AT91_REG Reserved1[1];   //
    AT91_REG PIO_IFER;       // Input Filter Enable Register
    AT91_REG PIO_IFDR;       // Input Filter Disable Register
    AT91_REG PIO_IFSR;       // Input Filter Status Register
    AT91_REG Reserved2[1];   //
    AT91_REG PIO_SODR;       // Set Output Data Register
    AT91_REG PIO_CODR;       // Clear Output Data Register
    AT91_REG PIO_ODSR;       // Output Data Status Register
    AT91_REG PIO_PDSR;       // Pin Data Status Register
    AT91_REG PIO_IER;        // Interrupt Enable Register
    AT91_REG PIO_IDR;        // Interrupt Disable Register
    AT91_REG PIO_IMR;        // Interrupt Mask Register
    AT91_REG PIO_ISR;        // Interrupt Status Register
    AT91_REG PIO_MDER;       // Multi-driver Enable Register
    AT91_REG PIO_MDDR;       // Multi-driver Disable Register
    AT91_REG PIO_MDSR;       // Multi-driver Status Register
    AT91_REG Reserved3[1];   //
    AT91_REG PIO_PPUDR;      // Pull-up Disable Register
    AT91_REG PIO_PPUER;      // Pull-up Enable Register
    AT91_REG PIO_PPUSR;      // Pull-up Status Register
    AT91_REG Reserved4[1];   //
    AT91_REG PIO_ASR;        // Select A Register
    AT91_REG PIO_BSR;        // Select B Register
    AT91_REG PIO_ABSR;       // AB Select Status Register
    AT91_REG Reserved5[9];   //
    AT91_REG PIO_OWER;       // Output Write Enable Register
    AT91_REG PIO_OWDR;       // Output Write Disable Register
    AT91_REG PIO_OWSR;       // Output Write Status Register
} AT91S_PIO, *AT91PS_PIO;
```

First, we can set up a pointer to the Parallel Input/Output Controller (PIO) structure.

```
volatile AT91PS_PIO pPIO = AT91C_BASE_PIOA;
```

We can simplify things a bit by noting that we only intend to use the RXD0 and TXD0 pins of the full USART0 pin set (RXD0, TXD0, SCK0, RTS0, CTS0). In the setup below, you should note that we are ignoring the pins (SCK0, RTS0, CTS0). There are no ill effects from this decision; we don't normally send the baud rate clock SCK0 to the outside world in an asynchronous serial application and the "request to send" pin RTS0 and "clear to send" pin CTS0 aren't normally used in a simple RS-232 hookup. This leaves us with just two pins to set up, pin PA0 (RXD0) and pin PA1 (TXD0).

First, we disable PIO control of these two pins. Use the PIO Controller PIO Disable Register (PIO_PDR) to do this. This is effectively telling the PIO controller that we will not be using these pins as simple I/O points.

27.7.2 PIO Controller PIO Disable Register

Name: PIO_PDR
Access Type: Write-only

31	30	29	28	27	26	25	24
P31	P30	P29	P28	P27	P26	P25	P24
23	22	21	20	19	18	17	16
P23	P22	P21	P20	P19	P18	P17	P16
15	14	13	12	11	10	9	8
P15	P14	P13	P12	P11	P10	P9	P8
7	6	5	4	3	2	1	0
P7	P6	P5	P4	P3	P2	P1	P0

• **P0-P31: PIO Disable**

0 = No effect.

1 = Disables the PIO from controlling the corresponding pin (enables peripheral control of the pin).

I'll leave it to the reader to use the Eclipse **F3** key to find the definitions of these constants in the include file.

```
pPIO->PIO_PDR = AT91C_PA0_RXD0 | AT91C_PA1_TXD0;
```

Now we have to assign these two pins to a peripheral. You have two choices, Peripheral Set A or Peripheral Set B. The Atmel Data Sheet, Table 10-2 on page 33 shows the choices; below is a portion of that table.

Table 10-2. Multiplexing on PIO Controller A

PIO Controller A				Application Usage	
I/O Line	Peripheral A	Peripheral B	Comments	Function	Comments
PA0	RXD0		High-Drive		
PA1	TXD0		High-Drive		
PA2	SCK0	SPI1_NPCS1	High-Drive		
PA3	RTS0	SPI1_NPCS2	High-Drive		
PA4	CTS0	SPI1_NPCS3			
PA5	RXD1				
PA6	TXD1				
PA7	SCK1	SPI0_NPCS1			
PA8	RTS1	SPI0_NPCS2			
PA9	CTS1	SPI0_NPCS3			

Note that pins PA0 and PA1 (RXD0 and TXD0, respectively) are listed above as being part of Peripheral Set A. The PIO Peripheral A Select Register (PIO_ASR) lets us select Peripheral Set A for those two pins.

27.7.24 PIO Peripheral A Select Register

Name: PIO_ASR

Access Type: Write-only

31	30	29	28	27	26	25	24
P31	P30	P29	P28	P27	P26	P25	P24
23	22	21	20	19	18	17	16
P23	P22	P21	P20	P19	P18	P17	P16
15	14	13	12	11	10	9	8
P15	P14	P13	P12	P11	P10	P9	P8
7	6	5	4	3	2	1	0
P7	P6	P5	P4	P3	P2	P1	P0

• P0-P31: Peripheral A Select.

0 = No effect.

1 = Assigns the I/O line to the Peripheral A function.

```
pPIO->PIO_ASR = AT91C_PIO_PA0 | AT91C_PIO_PA1;
```

While not required, we'll set the PIO Peripheral B Select Register (PIO_BSR) to zero ("no effect") just for the sake of clarity.

27.7.25 PIO Peripheral B Select Register

Name: PIO_BSR

Access Type: Write-only

31	30	29	28	27	26	25	24
P31	P30	P29	P28	P27	P26	P25	P24
23	22	21	20	19	18	17	16
P23	P22	P21	P20	P19	P18	P17	P16
15	14	13	12	11	10	9	8
P15	P14	P13	P12	P11	P10	P9	P8
7	6	5	4	3	2	1	0
P7	P6	P5	P4	P3	P2	P1	P0

• P0-P31: Peripheral B Select.

0 = No effect.

1 = Assigns the I/O line to the peripheral B function.

```
pPIO->PIO_BSR = 0;
```

At this point, we have the USART0 peripheral clock turned on and the two pins (RXD0 and TXD0) are associated with the USART0 peripheral.

Set Up the USART0 Registers

To set up the USART0 registers, we'll need a pointer to the USART data structure. The same data structure will suffice for both USART0 and USART1. Here is the USART data structure from the Atmel include file:

```
typedef struct _AT91S_USART {
    AT91_REG US_CR;          // Control Register
    AT91_REG US_MR;          // Mode Register
    AT91_REG US_IER;         // Interrupt Enable Register
    AT91_REG US_IDR;         // Interrupt Disable Register
    AT91_REG US_IMR;         // Interrupt Mask Register
    AT91_REG US_CSR;         // Channel Status Register
    AT91_REG US_RHR;         // Receiver Holding Register
    AT91_REG US_THR;         // Transmitter Holding Register
    AT91_REG US_BRGR;        // Baud Rate Generator Register
    AT91_REG US_RTOR;        // Receiver Time-out Register
    AT91_REG US_TTGR;        // Transmitter Time-guard Register
    AT91_REG Reserved0[5];   //
    AT91_REG US_FIDI;        // FI_DI_Ratio Register
    AT91_REG US_NER;         // Nb Errors Register
    AT91_REG Reserved1[1];   //
    AT91_REG US_IF;         // IRDA_FILTER Register
    AT91_REG Reserved2[44];  //
    AT91_REG US_RPR;         // Receive Pointer Register
    AT91_REG US_RCR;         // Receive Counter Register
    AT91_REG US_TPR;         // Transmit Pointer Register
    AT91_REG US_TCR;         // Transmit Counter Register
    AT91_REG US_RNPR;        // Receive Next Pointer Register
    AT91_REG US_RNCR;        // Receive Next Counter Register
    AT91_REG US_TNPR;        // Transmit Next Pointer Register
    AT91_REG US_TNCR;        // Transmit Next Counter Register
    AT91_REG US_PTCR;        // PDC Transfer Control Register
    AT91_REG US_PTSR;        // PDC Transfer Status Register
} AT91S_USART, *AT91PS_USART;
```

As shown several times before, a pointer to the USART data structure can be created as shown below. Note that we assigned the base memory address specific to USART0.

```
volatile AT91PS_USART pUsart0 = AT91C_BASE_US0;
```

Control Register - Reset then Disable the Receiver/Transmitter

A prudent first move in initializing the USART0 is to “reset” the USART0 receiver and transmitter and then disable both to prevent any surprises while we set up the rest of the USART0 registers. We can use the USART0 Control Register to do this.

30.7.1 USART Control Register

Name: US_CR

Access Type: Write-only

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	RTSDIS	RTSEN	DTRDIS	DTREN
15	14	13	12	11	10	9	8
RETTO	RSTNACK	RSTIT	SENDA	STTTO	STPBRK	STTBK	RSTSTA
7	6	5	4	3	2	1	0
TXDIS	TXEN	RXDIS	RXEN	RSTTX	RSTRX	-	-

Shown directly below are the meanings of the various bits in the USART0 Control Register.

- **RSTRX: Reset Receiver**
0: No effect.
1: Resets the receiver.
- **RSTTX: Reset Transmitter**
0: No effect.
1: Resets the transmitter.
- **RXEN: Receiver Enable**
0: No effect.
1: Enables the receiver, if RXDIS is 0.
- **RXDIS: Receiver Disable**
0: No effect.
1: Disables the receiver.
- **TXEN: Transmitter Enable**
0: No effect.
1: Enables the transmitter if TXDIS is 0.
- **TXDIS: Transmitter Disable**
0: No effect.
1: Disables the transmitter.
- **RSTSTA: Reset Status Bits**
0: No effect.
1: Resets the status bits PARE, FRAME, OVRE, and RXBRK in US_CSR.
- **STTBK: Start Break**
0: No effect.
1: Starts transmission of a break after the characters present in US_THR and the Transmit Shift Register have been transmitted

- **STPBK: Stop Break**
0: No effect.
1: Stops transmission of the break after a minimum of one character length and transmits a high level during 12-bit periods. No effect if no break is being transmitted.
- **STTTO: Start Time-out**
0: No effect.
1: Starts waiting for a character before clocking the time-out counter. Resets the status bit TIMEOUT in US_CSR.
- **SENDA: Send Address**
0: No effect.
1: In Multidrop Mode only, the next character written to the US_THR is sent with the address bit set.
- **RSTIT: Reset Iterations**
0: No effect.
1: Resets ITERATION in US_CSR. No effect if the ISO7816 is not enabled.
- **RSTNACK: Reset Non Acknowledge**
0: No effect
1: Resets NACK in US_CSR.
- **RETO: Rearm Time-out**
0: No effect
1: Restart Time-out
- **DTREN: Data Terminal Ready Enable**
0: No effect.
1: Drives the pin DTR at 0.
- **DTRDIS: Data Terminal Ready Disable**
0: No effect.
1: Drives the pin DTR to 1.
- **RTSEN: Request to Send Enable**
0: No effect.
1: Drives the pin RTS to 0.
- **RTSDIS: Request to Send Disable**
0: No effect.
1: Drives the pin RTS to 1.

In the C statement below, we set the USART0 Control register to reset the transmitter and receiver and then disable both of them.

```
pUsart0->US_CR = AT91C_US_RSTRX | // reset receiver
                 AT91C_US_RSTTX | // reset transmitter
                 AT91C_US_RXDIS | // disable receiver
                 AT91C_US_TXDIS; // disable transmitter
```

We will “enable” the receiver as one of the last steps in setting up the USART.

Mode Register – Set up Character Format, etc.

The USART0 Mode Register is a “catch-all” for setting up various communications parameters. We know we want to set up 1 start bit, 8 data bits, 1 stop bit and no parity. We also want to set up “asynchronous” mode and select MCK to drive the baud rate clock. The layout of the USART0 Mode Register is as follows:

30.7.2 USART Mode Register

Name: US_MR

Access Type: Read/Write

31	30	29	28	27	26	25	24
–	–	–	FILTER	–	MAX_ITERATION		
23	22	21	20	19	18	17	16
–	–	DSNACK	INACK	OVER	CLKO	MODE9	MSBF
15	14	13	12	11	10	9	8
CHMODE		NBSTOP		PAR			SYNC
7	6	5	4	3	2	1	0
CHRL		USCLKS		USART_MODE			

There are a number of fields in this Mode Register, here are some of the settings available.

• USART_MODE

USART_MODE				Mode of the USART
0	0	0	0	Normal
0	0	0	1	RS485
0	0	1	0	Hardware Handshaking
0	0	1	1	Modem
0	1	0	0	ISO7816 Protocol: T = 0
0	1	0	1	Reserved
0	1	1	0	ISO7816 Protocol: T = 1
0	1	1	1	Reserved
1	0	0	0	IrDA
1	1	x	x	Reserved

We want “normal” mode for this field.

Since this is 0000 and is the default reset condition, we won't have to set this field.

• USCLKS: Clock Selection

USCLKS		Selected Clock
0	0	MCK
0	1	MCK / DIV
1	0	Reserved
1	1	SCK

We want “MCK” clock for this field (see previous baud rate calculation).

Since this is 00 and is the default reset condition, we won't have to set this field.

• CHRL: Character Length.

CHRL		Character Length
0	0	5 bits
0	1	6 bits
1	0	7 bits
1	1	8 bits

We want “8 bits” character length for this field.

Since this is 11, we will have to set this field.

- **SYNC: Synchronous Mode Select**
0: USART operates in Asynchronous Mode.
1: USART operates in Synchronous Mode.
- **PAR: Parity Type**

We want "Asynchronous" mode for this field.
Since this is 0 and is the default reset condition, we won't have to set this field.

PAR			Parity Type
0	0	0	Even parity
0	0	1	Odd parity
0	1	0	Parity forced to 0 (Space)
0	1	1	Parity forced to 1 (Mark)
1	0	x	No parity
1	1	x	Multidrop mode

We want "No Parity" for this field.
Since this is 100, we will have to set this field.

- **NBSTOP: Number of Stop Bits**

NBSTOP		Asynchronous (SYNC = 0)	Synchronous (SYNC = 1)
0	0	1 stop bit	1 stop bit
0	1	1.5 stop bits	Reserved
1	0	2 stop bits	2 stop bits
1	1	Reserved	Reserved

We want "1 stop bit" for this field.
Since this is 00 and is the default reset condition, we won't have to set this field.

- **CHMODE: Channel Mode**

CHMODE		Mode Description
0	0	Normal Mode
0	1	Automatic Echo. Receiver input is connected to the TXD pin.
1	0	Local Loopback. Transmitter output is connected to the Receiver Input..
1	1	Remote Loopback. RXD pin is internally connected to the TXD pin.

We want "Normal" mode for this field.
Since this is 00 and is the default reset condition, we won't have to set this field.

- **MSBF: Bit Order**

0: Least Significant Bit is sent/received first.
1: Most Significant Bit is sent/received first.

We want "Least Significant Bit" order for this field. Since this is 0, we won't have to set this field.

- **MODE9: 9-bit Character Length**

0: CHRL defines character length.
1: 9-bit character length.

We don't want "9-bit char" length for this field. Since this is 0, we won't have to set this field.

- **CLKO: Clock Output Select**

0: The USART does not drive the SCK pin.
1: The USART drives the SCK pin if USCLKS does not select the external clock SCK.

We want "USART does not drive SCK" pin for this field. Since this is 0, we won't have to set this field.

- **OVER: Oversampling Mode**

0: 16x Oversampling.
1: 8x Oversampling.

We want "16X Oversampling" mode for this field. Since this is 0, we won't have to set this field.

- **INACK: Inhibit Non Acknowledge**

0: The NACK is generated.
1: The NACK is not generated.

This is not applicable in simple "Normal" mode – so leave as 0. Thus we won't have to set this field.

- **DSNACK: Disable Successive NACK**

0: NACK is sent on the ISO line as soon as a parity error occurs in the received character (unless INACK is set).
1: Successive parity errors are counted up to the value specified in the MAX_ITERATION field. These parity errors generate a NACK on the ISO line. As soon as this value is reached, no additional NACK is sent on the ISO line. The flag ITERATION is asserted.

This is not applicable in simple "Normal" mode – so leave as 0. Thus we won't have to set this field.

- **MAX_ITERATION**

Defines the maximum number of iterations in mode ISO7816, protocol T= 0.

This is not applicable in simple "Normal" mode – so leave as 0. Thus we won't have to set this field.

- **FILTER: Infrared Receive Line Filter**

0: The USART does not filter the receive line.
1: The USART filters the receive line using a three-sample filter (1/16-bit clock) (2 over 3 majority)

This is not applicable in simple "Normal" mode – so leave as 0. Thus we won't have to set this field.

Based on the notes in the USART0 Mode Register description above, we can tabulate the following Mode Register settings:

USART0 Mode Register Settings		
Field	Setting	Description
USART_MODE	000	Select Normal mode of the USART
USCLKS	00	Select MCK as source for baud rate generation
CHRL	11	Select 8 bit character length
SYNC	0	Select USART operates in Asynchronous mode
PAR	100	Select no parity
NBSTOP	00	Select 1 stop bit
CHMODE	00	Select Normal Channel Mode
MSBF	0	Select Least Significant Bit transmitted first
MODE9	0	Select CHRL (above) defines char length (no 9-bit chars)
CLK0	0	Select USART does NOT drive SCK pin
OVER	0	Select 16X Oversampling (used in baud rate calculation)
INACK	0	Not applicable in “normal” mode – leave as zero
DSNACK	0	Not applicable in “normal” mode – leave as zero
MAX_ITERATION	0	Not applicable in “normal” mode – leave as zero
FILTER	0	Select USART does NOT filter the receive line

Based on the settings shown in the table above, we can set up the Mode register with the following C language statement:

```
pUsart0->US_MR = AT91C_US_PAR_NONE | // no parity
                0x3 << 6;           // 8-bit characters
```

Interrupt Enable Register – Enable Desired USART0 Interrupt

While we are configuring the USART0, it behooves us to **not** enable any possible UART0 interrupts.

30.7.3 USART Interrupt Enable Register

Name: US_IER
Access Type: Write-only

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	CTSIC	DCDIC	DSRIC	RIIC
15	14	13	12	11	10	9	8
–	–	NACK	RXBUFFER	TXBUFE	ITERATION	TXEMPTY	TIMEOUT
7	6	5	4	3	2	1	0
PARE	FRAME	OVRE	ENDTX	ENDRX	RXBRK	TXRDY	RXRDY

- RXRDY: RXRDY Interrupt Enable
- TXRDY: TXRDY Interrupt Enable
- RXBRK: Receiver Break Interrupt Enable
- ENDRX: End of Receive Transfer Interrupt Enable
- ENDTX: End of Transmit Interrupt Enable
- OVRE: Overrun Error Interrupt Enable
- FRAME: Framing Error Interrupt Enable
- PARE: Parity Error Interrupt Enable
- TIMEOUT: Time-out Interrupt Enable
- TXEMPTY: TXEMPTY Interrupt Enable
- ITERATION: Iteration Interrupt Enable
- TXBUFE: Buffer Empty Interrupt Enable
- RXBUFF: Buffer Full Interrupt Enable
- NACK: Non Acknowledge Interrupt Enable
- RIIC: Ring Indicator Input Change Enable
- DSRIC: Data Set Ready Input Change Enable
- DCDIC: Data Carrier Detect Input Change Interrupt Enable
- CTSIC: Clear to Send Input Change Interrupt Enable

The following C language statement will **not** select any of the USART0 interrupts for activation.

```
pUsart0->US_IER = 0x0000; // no usart0 interrupts enabled (no effect)
```

Interrupt Disable Register – Disable Desired USART0 Interrupt

While we are configuring the USART0, it also behooves us to temporarily disable all possible UART0 interrupts

30.7.4 USART Interrupt Disable Register

Name: US_IDR
Access Type: Write-only

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	CTSIC	DCDIC	DSRIC	RIIC
15	14	13	12	11	10	9	8
–	–	NACK	RXBUFF	TXBUFE	ITERATION	TXEMPTY	TIMEOUT
7	6	5	4	3	2	1	0
PARE	FRAME	OVRE	ENDTX	ENDRX	RXBRK	TXRDY	RXRDY

- RXRDY: RXRDY Interrupt Enable
- TXRDY: TXRDY Interrupt Enable
- RXBRK: Receiver Break Interrupt Enable
- ENDRX: End of Receive Transfer Interrupt Enable
- ENDTX: End of Transmit Interrupt Enable
- OVRE: Overrun Error Interrupt Enable
- FRAME: Framing Error Interrupt Enable
- PARE: Parity Error Interrupt Enable
- TIMEOUT: Time-out Interrupt Enable
- TXEMPTY: TXEMPTY Interrupt Enable
- ITERATION: Iteration Interrupt Enable
- TXBUFE: Buffer Empty Interrupt Enable
- RXBUFF: Buffer Full Interrupt Enable
- NACK: Non Acknowledge Interrupt Enable
- RIIC: Ring Indicator Input Change Enable
- DSRIC: Data Set Ready Input Change Enable
- DCDIC: Data Carrier Detect Input Change Interrupt Enable
- CTSIC: Clear to Send Input Change Interrupt Enable

The following C language statement will disable **all** of the USART0 interrupts.

```

pUsart0->US_IDR = 0xFFFF; // all usart0 interrupts disabled

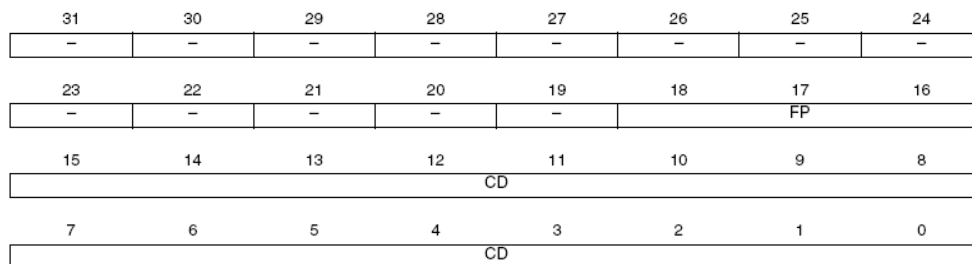
```

Baud Rate Generator Register – enter baud rate clock divider

In the baud rate analysis given previously, we determined that the clock divider **CD** should be set to 313 (0x139) to achieve a 9600 baud rate.

30.7.9 USART Baud Rate Generator Register

Name: US_BRGR
Access Type: Read/Write



• **CD: Clock Divider**

CD	USART_MODE ≠ ISO7816			USART_MODE = ISO7816
	SYNC = 0		SYNC = 1	
	OVER = 0	OVER = 1		
0	Baud Rate Clock Disabled			
1 to 65535	Baud Rate = Selected Clock/16/CD	Baud Rate = Selected Clock/8/CD	Baud Rate = Selected Clock /CD	Baud Rate = Selected Clock/CD/FL_DI_RATIO

- **FP: Fractional Part**
- 0: Fractional divider is disabled.
- 1 - 7: Baudrate resolution, defined by FP x 1/8.

The following C language statement will set the baud rate clock divider to 0x139. In this baud rate setup, the Fractional Divider is disabled.

```
pUSART0->US_BRGR = 0x139;    // CD = 0x139  (313 from above calculation)
                             // FP=0 (not used)
```

Set Up the USART0 Registers that are not Used

Since we elected to run the USART0 in “normal” mode, there are four setup registers that are not applicable. These registers are: Receiver Time-out Register, Transmitter Timeguard Register, FI DI Ratio register, and IrDA Filter Register. These registers can simply be set to zero.

```
pUSART0->US_RTOR = 0;        // receiver time-out (disabled)
pUSART0->US_TTGR = 0;        // transmitter timeguard (disabled)
pUSART0->US_FIDI = 0;        // FI over DI Ratio Value (disabled)
pUSART0->US_IF = 0;         // IrDA Filter value (disabled)
```

Note that the only USART0 register that is either read/write or write-only that I didn't access is the Transmit Holding Register. This is not actually a “setup” register. Anytime you write to this register, the contents are immediately dumped into the transmit shift register and the character starts clocking out (assuming that the transmitter is enabled, of course).

Setting Up the Advanced Interrupt Controller (AIC)

The issue of interrupts in a AT91SAM7 is fairly complex for a novice, so a detailed discussion of handling interrupts may be helpful. As mentioned before, there are 32 possible interrupts in a Atmel AT91SAM7X256 chip as shown below.

Table 10-1. Peripheral Identifiers

Peripheral ID	Peripheral Mnemonic	Peripheral Name	External Interrupt
0	AIC	Advanced Interrupt Controller	FIQ
1	SYSC ⁽¹⁾	System	
2	PIOA	Parallel I/O Controller A	
3	PIOB	Parallel I/O Controller B	
4	SPI0	Serial Peripheral Interface 0	
5	SPI1	Serial Peripheral Interface 1	
6	US0	USART 0	
7	US1	USART 1	
8	SSC	Synchronous Serial Controller	
9	TWI	Two-wire Interface	
10	PWMC	Pulse Width Modulation Controller	
11	UDP	USB device Port	
12	TC0	Timer/Counter 0	
13	TC1	Timer/Counter 1	
14	TC2	Timer/Counter 2	
15	CAN	CAN Controller	
16	EMAC	Ethernet MAC	
17	ADC ⁽¹⁾	Analog-to Digital Converter	
18	AES	Advanced Encryption Standard 128-bit	
19	TDES	Triple Data Encryption Standard	
20-29	Reserved		
30	AIC	Advanced Interrupt Controller	IRQ0
31	AIC	Advanced Interrupt Controller	IRQ1

Keep in mind that the “system” interrupt (Peripheral ID = 1) above actually covers seven shared interrupts; periodic interval timer PIT, real time timer RTT, watchdog timer WDT, debug-UART DBGU, power management controller PMC, reset controller RSTC, and embedded flash controller EFC. If you enable one or more of these system controller interrupts, then you will have to query within the IRQ handler the status registers for each enabled “system” peripheral to determine exactly what happened. Fortunately, these will not be used in our serial communications examples to follow.

The Atmel AT91SAM7X256 microcontroller has a Advanced Interrupt Controller (AIC) to assist in quickly processing IRQ interrupts. The basic idea is that when you have an interrupt asserted, the AIC will quickly supply you with the IRQ Interrupt Handler address to jump to. Common sense tells us that the AIC will need to know the following things about any potential interrupt: **address of the interrupt handler**, its **priority**, and if you want it **edge-triggered or level-sensitive**.

To specify the handler address, there are 32 **AIC Source Vector Registers**, one for each possible interrupt. If the interrupt is unused, program it to a “default handler” address (usually a endless loop). This is actually accomplished at the end of the **lowlevelinit.c** module in the project. In our example, we will be programming the handler address **Usart0IrqHandler()** into AIC_SVR6. That will be the seventh SVR register, assuming zero-base addressing of the array of AIC Source Vector Registers (**AIC_SVR6**).

23.8.4 AIC Source Vector Register

Register Name: AIC_SVR0..AIC_SVR31

Access Type: Read/Write

Reset Value: 0x0

31	30	29	28	27	26	25	24
VECTOR							
23	22	21	20	19	18	17	16
VECTOR							
15	14	13	12	11	10	9	8
VECTOR							
7	6	5	4	3	2	1	0
VECTOR							

• VECTOR: Source Vector

The user may store in these registers the addresses of the corresponding handler for each interrupt source.

There is also a companion set of 32 **AIC Source Mode Registers** that allow you to specify the priority and the “level-sensitive” or edge-sensitive” trigger characteristic of any potential interrupts.

There are 8 priority levels available. “Why not 32 levels, you ask”? It is not discussed in the data sheet, but I suspect that determination of the highest priority interrupt is a logic circuit that would get too complex for 32 priority levels. Normally, eight priority levels are more than sufficient, but if your design has a large number of interrupts, then some interrupts may end up having the same priority. In that case, the interrupt with the lowest source number is serviced first. From the data sheet diagram below, zero is the lowest priority while seven is the highest priority.

23.8.3 AIC Source Mode Register

Register Name: AIC_SMR0..AIC_SMR31

Access Type: Read/Write

Reset Value: 0x0

31	30	29	28	27	26	25	24	
–	–	–	–	–	–	–	–	
23	22	21	20	19	18	17	16	
–	–	–	–	–	–	–	–	
15	14	13	12	11	10	9	8	
–	–	–	–	–	–	–	–	
7	6	5	4	3	2	1	0	
–	SRCTYPE		–	–	PRIOR			–

- **PRIOR: Priority Level**

Programs the priority level for all sources except FIQ source (source 0).

The priority level can be between 0 (lowest) and 7 (highest).

The priority level is not used for the FIQ in the related SMR register AIC_SMRx.

- **SRCTYPE: Interrupt Source Type**

The active level or edge is not programmable for the internal interrupt sources.

When the AIC detects an interrupt and resolves the competing priorities, if any, it will copy the handler address you programmed beforehand into the AIC Interrupt Vector Register (AIC_IVR). This is the “winner”, so to speak.

23.8.5 AIC Interrupt Vector Register

Register Name: AIC_IVR

Access Type: Read-only

Reset Value: 0

31	30	29	28	27	26	25	24
IRQV							
23	22	21	20	19	18	17	16
IRQV							
15	14	13	12	11	10	9	8
IRQV							
7	6	5	4	3	2	1	0
IRQV							

- **IRQV: Interrupt Vector Register**

The Interrupt Vector Register contains the vector programmed by the user in the Source Vector Register corresponding to the current interrupt.

We can, of course, individually enable and disable interrupts in the Advanced Interrupt Controller (AIC). To enable any of the 32 interrupts, use the AIC Interrupt Enable Command Register shown below. Note that Bit0 is for the FIQ interrupt, Bit1 is for the System interrupt - that's the one that shares seven peripherals.

23.8.11 AIC Interrupt Enable Command Register

Register Name: AIC_IECR
Access Type: Write-only

31	30	29	28	27	26	25	24
PID31	PID30	PID29	PID28	PID27	PID26	PID25	PID24
23	22	21	20	19	18	17	16
PID23	PID22	PID21	PID20	PID19	PID18	PID17	PID16
15	14	13	12	11	10	9	8
PID15	PID14	PID13	PID12	PID11	PID10	PID9	PID8
7	6	5	4	3	2	1	0
PID7	PID6	PID5	PID4	PID3	PID2	SYS	FIQ

- **FIQ, SYS, PID2-PID31: Interrupt Enable**

0 = No effect.

1 = Enables corresponding interrupt.

Likewise, we can disable any interrupt in the AIC by using the AIC Interrupt Disable Command Register shown below.

23.8.12 AIC Interrupt Disable Command Register

Register Name: AIC_IDCR
Access Type: Write-only

31	30	29	28	27	26	25	24
PID31	PID30	PID29	PID28	PID27	PID26	PID25	PID24
23	22	21	20	19	18	17	16
PID23	PID22	PID21	PID20	PID19	PID18	PID17	PID16
15	14	13	12	11	10	9	8
PID15	PID14	PID13	PID12	PID11	PID10	PID9	PID8
7	6	5	4	3	2	1	0
PID7	PID6	PID5	PID4	PID3	PID2	SYS	FIQ

- **FIQ, SYS, PID2-PID31: Interrupt Disable**

0 = No effect.

1 = Disables corresponding interrupt.

You can clear any edge-triggered interrupt in the AIC by setting the appropriate bit in the AIC Interrupt Clear Command Register (AIC_ICCR) as shown below. This is an extra step you will have to do for any “edge-triggered” interrupts. For “level-sensitive” interrupts, such as our USART0 interrupt, this has no effect. For level-sensitive interrupts, the interrupt in the AIC is automatically cleared when you read the AIC Interrupt Vector Register (AIC_IVR). That operation is done in the assembly language part of interrupt handling.

23.8.13 AIC Interrupt Clear Command Register

Register Name: AIC_ICCR
Access Type: Write-only

31	30	29	28	27	26	25	24
PID31	PID30	PID29	PID28	PID27	PID26	PID25	PID24
23	22	21	20	19	18	17	16
PID23	PID22	PID21	PID20	PID19	PID18	PID17	PID16
15	14	13	12	11	10	9	8
PID15	PID14	PID13	PID12	PID11	PID10	PID9	PID8
7	6	5	4	3	2	1	0
PID7	PID6	PID5	PID4	PID3	PID2	SYS	FIQ

- **FIQ, SYS, PID2-PID31: Interrupt Clear**

0 = No effect.

1 = Clears corresponding interrupt.

You can “software trigger” any of the AIC interrupts by setting a bit in the AIC Interrupt Set Command Register, shown below. It would be rare for someone to employ this, but the capability might be useful in some software debugging situations. We won’t be using this register in our examples.

23.8.14 AIC Interrupt Set Command Register

Register Name: AIC_ISCR

Access Type: Write-only

31	30	29	28	27	26	25	24
PID31	PID30	PID29	PID28	PID27	PID26	PID25	PID24
23	22	21	20	19	18	17	16
PID23	PID22	PID21	PID20	PID19	PID18	PID17	PID16
15	14	13	12	11	10	9	8
PID15	PID14	PID13	PID12	PID11	PID10	PID9	PID8
7	6	5	4	3	2	1	0
PID7	PID6	PID5	PID4	PID3	PID2	SYS	FIQ

• **FIQ, SYS, PID2-PID31: Interrupt Set**

0 = No effect.

1 = Sets corresponding interrupt.

If there were multiple interrupts asserted simultaneously, you could look at them using the AIC Interrupt Pending Register (AIC_IPR) shown below. In most cases, looking at this register is superfluous since the whole purpose of the AIC is to feed you the handler address of every pending interrupt, one after another.

23.8.8 AIC Interrupt Pending Register

Register Name: AIC_IPR

Access Type: Read-only

Reset Value: 0

31	30	29	28	27	26	25	24
PID31	PID30	PID29	PID28	PID27	PID26	PID25	PID24
23	22	21	20	19	18	17	16
PID23	PID22	PID21	PID20	PID19	PID18	PID17	PID16
15	14	13	12	11	10	9	8
PID15	PID14	PID13	PID12	PID11	PID10	PID9	PID8
7	6	5	4	3	2	1	0
PID7	PID6	PID5	PID4	PID3	PID2	SYS	FIQ

• **FIQ, SYS, PID2-PID31: Interrupt Pending**

0 = Corresponding interrupt is not pending.

1 = Corresponding interrupt is pending.

The Atmel data sheet is very insistent that, at the end of your IRQ interrupt processing, you should signal “**End of Interrupt**” by writing any value (such as zero) to the AIC End of Interrupt Command Register shown below. Shortly you will see that this is done in the assembler language part of IRQ interrupt handling.

23.8.15 AIC End of Interrupt Command Register

Register Name: AIC_EOICR

Access Type: Write-only

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–
23	22	21	20	19	18	17	16
–	–	–	–	–	–	–	–
15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–
7	6	5	4	3	2	1	0
–	–	–	–	–	–	–	–

The End of Interrupt Command Register is used by the interrupt routine to indicate that the interrupt treatment is complete. Any value can be written because it is only necessary to make a write to this register location to signal the end of interrupt treatment.

Using the AIC setup registers just described above, we can properly set up the AIC for USART0 interrupts by the following code snippet.

```
// Set up the Advanced Interrupt Controller (AIC) registers for USART0
void Usart0IrqHandler(void); // function prototype for USART0 handler

volatile AT91PS_AIC pAIC = AT91C_BASE_AIC; // pointer to AIC data structure
pAIC->AIC_IDCR = (1<<AT91C_ID_US0); // Disable USART0 interrupt in AIC
pAIC->AIC_SVR[AT91C_ID_US0] = (unsigned int)Usart0IrqHandler; // Set the USART0 IRQ handler address in AIC
// Source Vector Register[6]
pAIC->AIC_SMR[AT91C_ID_US0] = (AT91C_AIC_SRCTYPE_INT_HIGH_LEVEL | 0x4 ); // Set the int source type and pri
// in AIC Source Mode Register[6]
pAIC->AIC_IECR = (1<<AT91C_ID_US0); // Enable the USART0 interrupt in AIC
```

Final Preparations for USART0 Interrupt Processing

Just a few more lines of code are required to have everything ready for USART0 interrupts. First, we enable both the receiver and transmitter. Since this application is half-duplex, so to speak, we will never have the transmitter and receiver running at the same time. In some RS-485 2-wire applications, you could run the risk of the receiver seeing everything that is being transmitted and thus interrupting on it. That is not the case here, so we can leave both of them on at all times.

We will enable the USART0 receive interrupt since we will be waiting for received characters to come in. We will also set up a pointer to the buffer and zero the number of characters (nChars) count. Remember that we intend to count ten incoming characters before transmitting them all back to the source in a burst.

Finally, we enable global IRQ interrupts (these were turned off in the `crt.s` assembly language module).

```
// external global variables
extern char Buffer[]; // holds received characters
extern unsigned long nChars; // counts number of received chars
extern char *pBuffer; // pointer into Buffer

// enable the USART0 receiver and transmitter
pUSART0->US_CR = AT91C_US_RXEN | AT91C_US_TXEN;

// enable the USART0 receive interrupt
pUSART0->US_IER = AT91C_US_RXRDY; // enable RXRDY usart0 receive interrupt
pUSART0->US_IDR = ~AT91C_US_RXRDY; // disable all other interrupts except RXRDY

// set up buffer pointer and character counter
pBuffer = &Buffer[0];
nChars = 0;

// enable IRQ interrupts
enableIRQ();

// at this point, only the USART0 receive interrupt is armed!
```

Assembly Language Part of the IRQ Handler

The sample project, evolved from the samples in the tutorial “Using Open Source Tools for Atmel AT91SAM7 Cross Development”, uses an assembler language start-up routine called “`crt.s`”. The IRQ handler of this routine is extracted from an Atmel example; this assembly language snippet does two great things: it supports “nested” interrupts and it allows us to develop our interrupt handlers as a pure C language function. The vector table and IRQ handler are shown below.

```
// Vector Table
vec_reset:  b      _init_reset          /* RESET vector - must be at 0x00000000 */
vec_undef:  b      AT91F_Undef_Handler /* Undefined Instruction vector */
vec_swi:    b      _vec_swi            /* Software Interrupt vector */
vec_pabt:   b      AT91F_Pabt_Handler  /* Prefetch abort vector */
vec_dabt:   b      AT91F_Dabt_Handler  /* Data abort vector */
vec_rsv:    nop
vec_irq:    b      AT91F_Irq_Handler   /* Interrupt Request (IRQ) vector */
vec_fiq:    b      _vec_fiq           /* Fast interrupt request (FIQ) vector */

AT91F_Irq_Handler:
/* Manage Exception Entry */
/* Adjust and save LR_irq in IRQ stack */
    sub     lr, lr, #4
    stmfd  sp!, {lr}

/* Save r0 and SPSR (need to be saved for nested interrupt) */
    mrs    r14, SPSR
    stmfd  sp!, {r0,r14}

/* Write in the IVR to support Protect Mode */
/* No effect in Normal Mode */
/* De-assert the NIRQ and clear the source in Protect Mode */
    ldr    r14, =AT91C_BASE_AIC
    ldr    r0, [r14, #AIC_IVR]
    str    r14, [r14, #AIC_IVR]

/* Enable Interrupt and Switch in Supervisor Mode */
    msr    CPSR_c, #ARM_MODE_SVC

/* Save scratch/used registers and LR in User Stack */
    stmfd  sp!, { r1-r3, r12, r14}

/* Branch to the routine pointed by the AIC_IVR */
    mov    r14, pc
    bx     r0

/* Manage Exception Exit */
/* Restore scratch/used registers and LR from User Stack */
    ldmia  sp!, { r1-r3, r12, r14}

/* Disable Interrupt and switch back in IRQ mode */
    msr    CPSR_c, #I_BIT | ARM_MODE_IRQ

/* Mark the End of Interrupt on the AIC */
    ldr    r14, =AT91C_BASE_AIC
    str    r14, [r14, #AIC_EOICR]

/* Restore SPSR_irq and r0 from IRQ stack */
    ldmia  sp!, {r0,r14}
    msr    SPSR_cxsf, r14

/* Restore adjusted LR_irq from IRQ stack directly in the PC */
    ldmia  sp!, {pc}^
```

USART0 interrupt executes this instruction! (points to `bx r0`)

Here the ISR code jumps to the address placed in the AIC Interrupt Vector Register (the winner!). This will be the handler: **Usart0IrqHandler()**

The beauty of this is that the handler can be a normal C language function!

Here is where we signal “end of interrupt” to the AIC. We can write anything to do this; here we write the AIC base address itself to the AIC_EOICR register.

All done!

Designing the USART0 IRQ Handler

The intention is to read 10 characters and after the 10th character has been received, transmit all ten received characters back to the source. This just seemed a little more interesting than retransmitting every incoming character as it comes in.

If there is a receive interrupt, we immediately copy the incoming character from the receive holding register (US_RHR) to the buffer and advance the buffer pointer and nChars.

If we have received ten characters, we reset the buffer pointer to the start of the Buffer and clear nChars. We enable the transmitter interrupt and disable the receiver interrupt and then send the first character in the buffer by loading the transmit holding register (US_THR) and advancing the pointer and nChars count. The first character in the buffer will now start clocking out and we will be interrupted next when the TXEMPTY interrupt occurs.

Processing a Receive Interrupt (RXRDY)

```
// we have a receive interrupt,
// remove it from Receiver Holding Register and place into buffer[]
*pBuffer++ = pUsart0->US_RHR;
nChars++;

// check if 10 characters have been received
if (nChars >= 10) {

    // yes, redirect buffer pointer to beginning
    pBuffer = &Buffer[0];
    nChars = 0;

    // disable the receive interrupt, enable the transmit interrupt
    pUsart0->US_IER = AT91C_US_TXEMPTY;      // enable TXEMPTY usart0 transmit interrupt
    pUsart0->US_IDR = ~AT91C_US_TXEMPTY;    // disable all interrupts except TXEMPTY

    // send first received character, TXEMPTY interrupt will send the rest
    pUsart0->US_THR = *pBuffer++;
    nChars++;
}
```

The transmit operation is very similar. We check if 10 characters has been already sent and, if so, set up for reception. If less than 10 characters have been sent, we fetch the next character and place it into the transmit holding register (US_THR) and advance the buffer pointer and character count.

```
// we have a transmit interrupt (previous char has clocked out)
// check if 10 characters have been transmitted
if (nChars >= 10 ) {

    // yes, redirect buffer pointer to beginning
    pBuffer = &Buffer[0];
    nChars = 0;

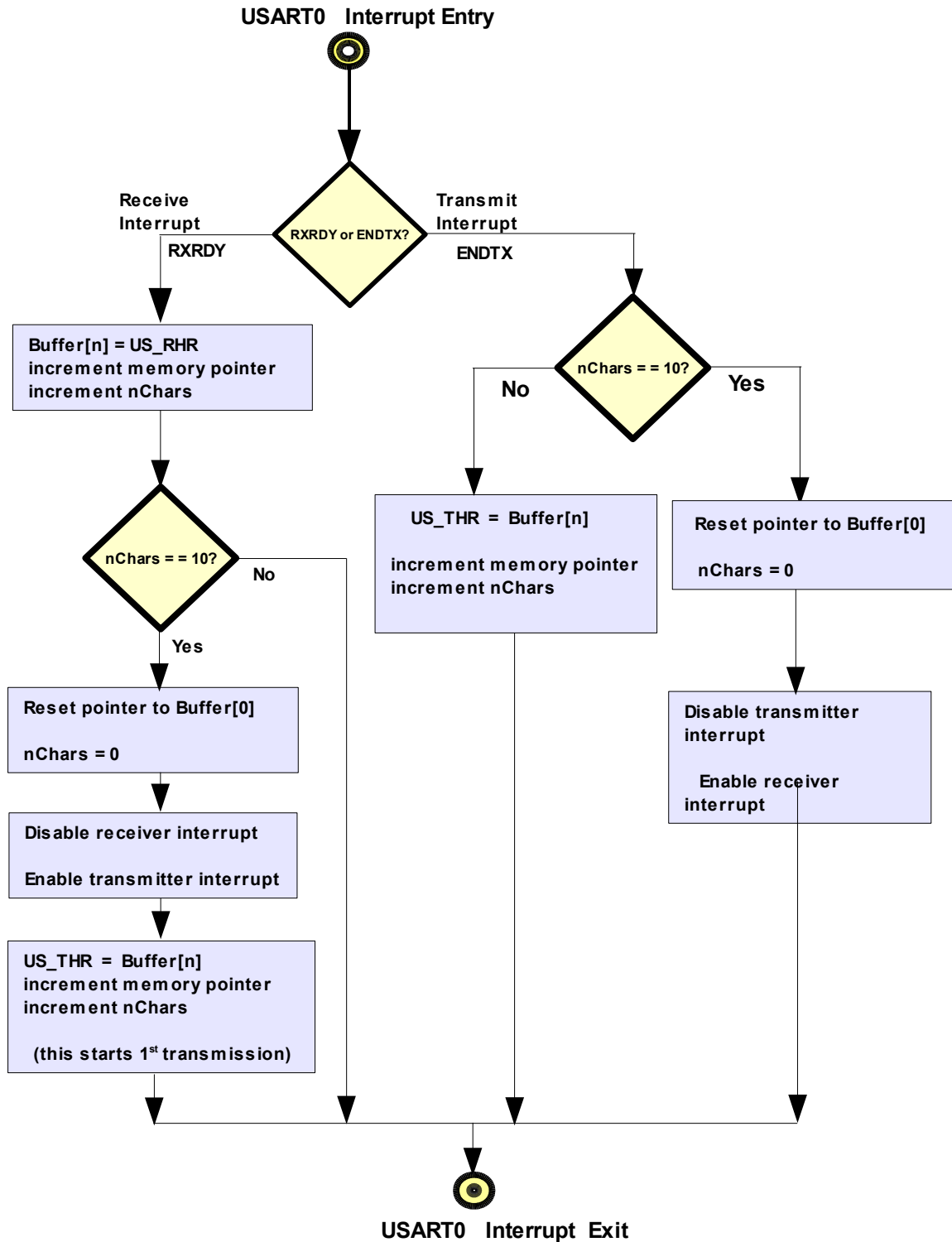
    // enable receive interrupt, disable the transmit interrupt
    pUsart0->US_IER = AT91C_US_RXRDY;      // enable RXRDY usart0 receive interrupt
    pUsart0->US_IDR = ~AT91C_US_RXRDY;    // disable all interrupts except RXRDY

} else {

    // no, send next character
    pUsart0->US_THR = *pBuffer++;
    nChars++;
}
```

A flow chart of the USART0 IRQ Handler is shown below.

Flowchart – USART0 Interrupt Handler



Project Listings – Interrupt Version

AT91SAM7X256.H

This is a standard Atmel include file that can be found on their web site. This file includes data structures and memory addresses for peripheral registers and other useful constants.

```
// -----
//          ATMEL Microcontroller Software Support  -  ROUSSET  -
// -----
//  DISCLAIMER:  THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS OR
//  IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
//  MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT ARE
//  DISCLAIMED.  IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT,
//  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
//  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
//  OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
//  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
//  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
//  EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
// -----
// File Name      : AT91SAM7X256.h
// Object         : AT91SAM7X256 definitions
// Generated      : AT91 SW Application Group 01/16/2006 (16:36:21)
//
#ifndef AT91SAM7X256_H
#define AT91SAM7X256_H

typedef volatile unsigned int AT91_REG; // Hardware register definition

// *****
//          SOFTWARE API DEFINITION FOR System Peripherals
// *****
typedef struct AT91S_SYS {
    AT91_REG  AIC_SMR[32];    // Source Mode Register
    AT91_REG  AIC_SVR[32];    // Source Vector Register
    AT91_REG  AIC_IVR;        // IRQ Vector Register
    AT91_REG  AIC_FVR;        // FIQ Vector Register
    AT91_REG  AIC_ISR;        // Interrupt Status Register
    AT91_REG  AIC_IPR;        // Interrupt Pending Register
    AT91_REG  AIC_IMR;        // Interrupt Mask Register
    AT91_REG  AIC_CISR;       // Core Interrupt Status Register
    AT91_REG  Reserved0[2];   //
    AT91_REG  AIC_IECR;       // Interrupt Enable Command Register
    AT91_REG  AIC_IDCR;       // Interrupt Disable Command Register
    AT91_REG  AIC_ICCR;       // Interrupt Clear Command Register
    AT91_REG  AIC_ISCR;       // Interrupt Set Command Register
    AT91_REG  AIC_EOICR;      // End of Interrupt Command Register
    AT91_REG  AIC_SPU;        // Spurious Vector Register
    AT91_REG  AIC_DCR;        // Debug Control Register (Protect)
    AT91_REG  Reserved1[1];   //
    AT91_REG  AIC_FFER;       // Fast Forcing Enable Register
    AT91_REG  AIC_FFDR;       // Fast Forcing Disable Register
    AT91_REG  AIC_FFSR;       // Fast Forcing Status Register
    AT91_REG  Reserved2[45];  //
    AT91_REG  DBGU_CR;        // Control Register
    AT91_REG  DBGU_MR;        // Mode Register
    AT91_REG  DBGU_IER;       // Interrupt Enable Register
    AT91_REG  DBGU_IDR;       // Interrupt Disable Register
    :
    :
    :
};
```

This is a very long file!

BOARD.H

It is traditional to have a “board support” include file which sets memory limits, etc. For this project, we only use the LED definition.

```
-----
//          ATMEL Microcontroller Software Support  -  ROUSSET  -
-----
// 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.
-----
// File Name:      Board.h
// Object:         AT91SAM7S Evaluation Board Features Definition File.
//
// Creation:       JPP   16/June/2004
-----
#ifndef Board_h
#define Board_h

#include "at91sam7x256.h"
#define __inline inline

#define true      1
#define false    0

//-----
// SAM7Board Memories Definition
//-----
// The AT91SAM7S2564 embeds a 64-Kbyte SRAM bank, and 256 K-Byte Flash

#define INT_SRAM          0x00200000
#define INT_SRAM_REMAP   0x00000000

#define INT_FLASH        0x00000000
#define INT_FLASH_REMAP 0x01000000

#define FLASH_PAGE_NB    1024
#define FLASH_PAGE_SIZE  256

//-----
// Leds Definition
//-----
#define LED4              (1<<3)           // PA3 (pin 1 on EXT connector)
#define LED_MASK          (LED4)

//-----
// Master Clock
//-----
#define EXT_OC             1843200          // Exetrnal ocilator MAINCK
#define MCK                47923200      // MCK (PLLRC div by 2)
#define MCKKHz              (MCK/1000)   //

#endif // Board_h
```

CRT.S

The assembler language start-up routine is similar to the one in the tutorial “*Using Open Source Tools for Atmel AT91SAM7 Cross Development*”. At completion, the start-up routine branches to main with the CPU in “system” mode with the global interrupts off. The IRQ interrupt processing part, designed by Atmel, Rousett, France supports nested interrupts and permits the rest of the IRQ handler to be just a simple C function.

```

/* ***** */
/*
/*                               CRT.S                               */
/*
/*                               Assembly Language Startup Code for Atmel AT91SAM7X256
/*
/*
/*
/*
/*
/*
/*
/* Author: James P Lynch      June 22, 2008
/* ***** */

/* Stack Sizes */
.set  UND_STACK_SIZE, 0x00000010 /* stack for "undefined instruction" interrupts is 16 bytes */
.set  ABT_STACK_SIZE, 0x00000010 /* stack for "abort" interrupts is 16 bytes */
.set  FIQ_STACK_SIZE, 0x00000080 /* stack for "FIQ" interrupts is 128 bytes */
.set  IRQ_STACK_SIZE, 0x00000080 /* stack for "IRQ" normal interrupts is 128 bytes */
.set  SVC_STACK_SIZE, 0x00000080 /* stack for "SVC" supervisor mode is 128 bytes */

/* Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs (program status registers) */
.set  ARM_MODE_USR, 0x10 /* Normal User Mode
/*
.set  ARM_MODE_FIQ, 0x11 /* FIQ Processing Fast Interrupts Mode */
.set  ARM_MODE_IRQ, 0x12 /* IRQ Processing Standard Interrupts Mode */
.set  ARM_MODE_SVC, 0x13 /* Supervisor Processing Software Interrupts Mode */
.set  ARM_MODE_ABT, 0x17 /* Abort Processing memory Faults Mode */
.set  ARM_MODE_UND, 0x1B /* Undefined Processing Undefined Instructions Mode */
.set  ARM_MODE_SYS, 0x1F /* System Running Priviledged Operating System Tasks Mode */
.set  I_BIT, 0x80 /* when I bit is set, IRQ is disabled (program status registers) */
.set  F_BIT, 0x40 /* when F bit is set, FIQ is disabled (program status registers) */

/* Addresses and offsets of AIC and PIO */
.set  AT91C_BASE_AIC, 0xFFFFF000 /* (AIC) Base Address */
.set  AT91C_PIOA_CODR, 0xFFFFF434 /* (PIO) Clear Output Data Register */
.set  AT91C_AIC_IVR, 0xFFFFF100 /* (AIC) IRQ Interrupt Vector Register */
.set  AT91C_AIC_FVR, 0xFFFFF104 /* (AIC) FIQ Interrupt Vector Register */
.set  AIC_IVR, 256 /* IRQ Vector Register offset from base above */
.set  AIC_FVR, 260 /* FIQ Vector Register offset from base above */
.set  AIC_EOICR, 304 /* End of Interrupt Command Register */

/* identify all GLOBAL symbols */
.global _vec_reset
.global _vec_undef
.global _vec_swi
.global _vec_pabt
.global _vec_dabt
.global _vec_rsv
.global _vec_irq
.global _vec_fiq
.global AT91F_Irq_Handler
.global AT91F_Fiq_Handler
.global AT91F_Default_FIQ_handler
.global AT91F_Default_IRQ_handler
.global AT91F_Spurious_handler
.global AT91F_Dabt_Handler
.global AT91F_Pabt_Handler
.global AT91F_Undef_Handler

/* GNU assembler controls */
.text /* all assembler code that follows will go into .text section */
.arm /* compile for 32-bit ARM instruction set */
.align /* align section on 32-bit boundary */

```

```

/* ===== */
/*          VECTOR TABLE          */
/*          Must be located in FLASH at address 0x00000000 */
/*          Easy to do if this file crt.s is first in the list */
/*          for the linker step in the makefile, e.g. */
/*          $(LD) $(LFLAGS) -o main.out crt.o main.o */
/* ===== */

_vec_reset:    b    _init_reset        /* RESET vector - must be at 0x00000000 */
_vec_undef:   b    AT91F_Undef_Handler /* Undefined Instruction vector */
_vec_swi:     b    _vec_swi           /* Software Interrupt vector (endless loop) */
_vec_pabt:    b    AT91F_Pabt_Handler /* Prefetch abort vector */
_vec_dabt:    b    AT91F_Dabt_Handler /* Data abort vector */
_vec_rsv:     nop                    /* Reserved vector */
_vec_irq:     b    AT91F_Irq_Handler  /* Interrupt Request (IRQ) vector */
_vec_fiq:     b    _vec_fiq           /* Fast interrupt request (FIQ) vector (endless loop) */

/* ===== */
/*          _init_reset Handler     */
/*          RESET vector 0x00000000 branches to here. */
/*          ARM microprocessor begins execution after RESET at address 0x00000000 */
/*          in Supervisor mode with interrupts disabled! */
/*          _init_reset handler:  creates a stack for each ARM mode. */
/*                               sets up a stack pointer for each ARM mode. */
/*                               turns off interrupts in each mode. */
/*                               leaves CPU in SYS (System) mode. */
/*                               block copies the initializers to .data section */
/*                               clears the .bss section to zero */
/*                               branches to main( ) */
/* ===== */

.text          /* all assembler code that follows will go into .text section */
.align        /* align section on 32-bit boundary */

_init_reset:
    /* Setup a stack for each mode with interrupts initially disabled. */
    ldr    r0, =_stack_end                /* r0 = top-of-stack */

    msr    CPSR_c, #ARM_MODE_UND|I_BIT|F_BIT /* switch to Undefined Instruction Mode */
    mov    sp, r0                        /* set stack pointer for UND mode */
    sub    r0, r0, #UND_STACK_SIZE       /* adjust r0 past UND stack */

    msr    CPSR_c, #ARM_MODE_ABT|I_BIT|F_BIT /* switch to Abort Mode */
    mov    sp, r0                        /* set stack pointer for ABT mode */
    sub    r0, r0, #ABT_STACK_SIZE       /* adjust r0 past ABT stack */

    msr    CPSR_c, #ARM_MODE_FIQ|I_BIT|F_BIT /* switch to FIQ Mode */
    mov    sp, r0                        /* set stack pointer for FIQ mode */
    sub    r0, r0, #FIQ_STACK_SIZE       /* adjust r0 past FIQ stack */

    msr    CPSR_c, #ARM_MODE_IRQ|I_BIT|F_BIT /* switch to IRQ Mode */
    mov    sp, r0                        /* set stack pointer for IRQ mode */
    sub    r0, r0, #IRQ_STACK_SIZE       /* adjust r0 past IRQ stack */

    msr    CPSR_c, #ARM_MODE_SVC|I_BIT|F_BIT /* switch to Supervisor Mode */
    mov    sp, r0                        /* set stack pointer for SVC mode */
    sub    r0, r0, #SVC_STACK_SIZE       /* adjust r0 past SVC stack */

    msr    CPSR_c, #ARM_MODE_SYS|I_BIT|F_BIT /* switch to System Mode */
    mov    sp, r0                        /* set stack pointer for SYS mode */
    /* we now start execution in SYSTEM mode */
    /* This is exactly like USER mode (same stack) */
    /* but SYSTEM mode has more privileges */

```

```

/* copy initialized variables .data section (Copy from ROM to RAM) */
    ldr    R1, =_etext
    ldr    R2, =_data
    ldr    R3, =_edata
1:      cmp    R2, R3
    ldrlo  R0, [R1], #4
    strlo  R0, [R2], #4
    blo   1b

/* Clear uninitialized variables .bss section (Zero init) */
    mov    R0, #0
    ldr    R1, =_bss_start
    ldr    R2, =_bss_end
2:      cmp    R1, R2
    strlo  R0, [R1], #4
    blo   2b

/* Enter the C code */
    b      main

/* ===== */
/* Function:          AT91F_Irq_Handler */
/* */
/* This IRQ_Handler supports nested interrupts (an IRQ interrupt can itself
/* be interrupted). */
/* */
/* This handler re-enables interrupts and switches to "Supervisor" mode to
/* prevent any corruption to the link and IP registers. */
/* */
/* The Interrupt Vector Register (AIC_IVR) is read to determine the address
/* of the required interrupt service routine. The ISR routine can be a
/* standard C function since this handler minds all the save/restore
/* protocols. */
/* */
/* Programmers: */
/*-----*/
/*          ATMEL Microcontroller Software Support  -  ROUSSET  -
/*-----*/
/* DISCLAIMER: THIS SOFTWARE IS PROVIDED BY ATMEL "AS IS" AND ANY EXPRESS
/* OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
/* WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
/* NON-INFRINGEMENT ARE DISCLAIMED. IN NO EVENT SHALL ATMEL BE LIABLE FOR
/* ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
/* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
/* OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
/* BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
/* WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
/* OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
/* EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
/* */
/* File source      : Cstartup.s79
/* Object           : Generic CStartup to AT91SAM7S256
/* 1.0 09/May/06 JPP : Creation
/* */
/* Note: taken from Atmel web site (www.at91.com)
/* Keil example project: AT91SAM7S-Interrupt_SAM7X
/* ===== */
AT91F_Irq_Handler:

/* Manage Exception Entry */
/* Adjust and save LR_irq in IRQ stack */
    sub    lr, lr, #4
    stmfd  sp!, {lr}

/* Save r0 and SPSR (need to be saved for nested interrupt) */
    mrs    r14, SPSR
    stmfd  sp!, {r0,r14}

/* Write in the IVR to support Protect Mode */
/* No effect in Normal Mode */
/* De-assert the NIRQ and clear the source in Protect Mode */
    ldr    r14, =AT91C_BASE_AIC
    ldr    r0, [r14, #AIC_IVR]
    str    r14, [r14, #AIC_IVR]

```

```

/* Enable Interrupt and Switch in Supervisor Mode */
    msr    CPSR_c, #ARM_MODE_SVC

/* Save scratch/used registers and LR in User Stack */
    stmfD  sp!, { r1-r3, r12, r14}

/* Branch to the routine pointed by the AIC_IVR */
    mov    r14, pc
    bx    r0

/* Manage Exception Exit */
/* Restore scratch/used registers and LR from User Stack */
    ldmia  sp!, { r1-r3, r12, r14}

/* Disable Interrupt and switch back in IRQ mode */
    msr    CPSR_c, #I_BIT | ARM_MODE_IRQ

/* Mark the End of Interrupt on the AIC */
    ldr    r14, =AT91C_BASE_AIC
    str    r14, [r14, #AIC_EOICR]

/* Restore SPSR_irq and r0 from IRQ stack */
    ldmia  sp!, {r0,r14}
    msr    SPSR_cxsf, r14

/* Restore adjusted LR_irq from IRQ stack directly in the PC */
    ldmia  sp!, {pc}^

/* ===== */
/* Function:          AT91F_Dabt_Handler          */
/*                   AT91F_Dabt_Handler          */
/*                   AT91F_Dabt_Handler          */
/* Entered on Data Abort exception.              */
/*                   AT91F_Dabt_Handler          */
/* ===== */
AT91F_Dabt_Handler:    b    AT91F_Dabt_Handler

/* ===== */
/* Function:          AT91F_Pabt_Handler          */
/*                   AT91F_Pabt_Handler          */
/*                   AT91F_Pabt_Handler          */
/* Entered on Prefetch Abort exception.          */
/*                   AT91F_Pabt_Handler          */
/* ===== */
AT91F_Pabt_Handler:    b    AT91F_Pabt_Handler

/* ===== */
/* Function:          AT91F_Undef_Handler        */
/*                   AT91F_Undef_Handler        */
/*                   AT91F_Undef_Handler        */
/* Entered on Undefined Instruction exception.    */
/*                   AT91F_Undef_Handler        */
/* ===== */
AT91F_Undef_Handler:  b    AT91F_Undef_Handler

AT91F_Default_FIQ_handler:    b    AT91F_Default_FIQ_handler
AT91F_Default_IRQ_handler:    b    AT91F_Default_IRQ_handler
AT91F_Spurious_handler:       b    AT91F_Spurious_handler

.end

```

ISRSUPPORT.C

This module, written by Bill Knight, provides the ability for a C function to turn the global interrupts on and off.

```
// *****
// File Name : isrsupport.c
// Title      : interrupt enable/disable functions
//
// This module provides the interface routines for setting up and
// controlling the various interrupt modes present on the ARM processor.
// Copyright 2004, R O Software
// No guarantees, warranties, or promises, implied or otherwise.
// May be used for hobby or commercial purposes provided copyright notice remains intact.
//
// Note from Jim Lynch:
// This module was developed by Bill Knight, RO Software and used with his permission.
// Taken from the Yahoo LPC2000 User's Group - Files Section 'UT050418A.ZIP'
// *****

#define IRQ_MASK 0x00000080
#define FIQ_MASK 0x00000040
#define INT_MASK (IRQ_MASK | FIQ_MASK)

static inline unsigned __get_cpsr(void) {
    unsigned long retval;
    asm volatile ("mrs %0, cpsr" : "=r" (retval) : /* no inputs */ );
    return retval;
}

static inline void __set_cpsr(unsigned val) {
    asm volatile ("msr cpsr, %0" : /* no outputs */ : "r" (val) );
}

unsigned disableIRQ(void) {
    unsigned _cpsr;
    _cpsr = __get_cpsr();
    __set_cpsr(_cpsr | IRQ_MASK);
    return _cpsr;
}

unsigned restoreIRQ(unsigned oldCPSR) {
    unsigned _cpsr;

    _cpsr = __get_cpsr();
    __set_cpsr((_cpsr & ~IRQ_MASK) | (oldCPSR & IRQ_MASK));
    return _cpsr;
}

unsigned enableIRQ(void) {
    unsigned _cpsr;

    _cpsr = __get_cpsr();
    __set_cpsr(_cpsr & ~IRQ_MASK);
    return _cpsr;
}

unsigned disableFIQ(void) {
    unsigned _cpsr;

    _cpsr = __get_cpsr();
    __set_cpsr(_cpsr | FIQ_MASK);
    return _cpsr;
}

unsigned restoreFIQ(unsigned oldCPSR) {
    unsigned _cpsr;

    _cpsr = __get_cpsr();
    __set_cpsr((_cpsr & ~FIQ_MASK) | (oldCPSR & FIQ_MASK));
    return _cpsr;
}

unsigned enableFIQ(void) {
    unsigned _cpsr;

    _cpsr = __get_cpsr();
    __set_cpsr(_cpsr & ~FIQ_MASK);
    return _cpsr;
}
}
```

Lowlevelinit.c

When the Atmel AT91SAM7256 boots up, the CPU starts running with a simple 32 Khz RC oscillator – not very fast. This module, adapted from Atmel examples, sets up the phased lock loop circuits to use the crystal oscillator (18.432 Mhz) and multiply it to ≈48 Mhz.

```
// *****
//                                     lowlevelinit.c
//
// Basic hardware initialization
//
// SLCK = 42000 hz (worst case) 32768 hz is the nominal slow clock frequency
// SLCK_PERIOD = 1 / 42000 = 23.8 usec
//
// MAINCK = 18432000 hz crystal on Olimex SAM7-EX256 board)
// PLLCK = (MAINCK / DIV) * (MUL + 1) = 18432000/14 * (72 + 1)
// PLLCLK = 1316571 * 73 = 96109683 hz
// MCK = PLLCLK / 2 = 96109683 / 2 = 48054841 hz
//
// Note: see page 5 - 6 of Atmel's "Getting Started with AT91SAM7X Microcontrollers" for details.
//
// Author: James P Lynch June 22, 2008
// *****

// include files
#include "at91sam7x256.h"
#include "Board.h"

// external references
extern void AT91F_Spurious_handler(void);
extern void AT91F_Default_IRQ_handler(void);
extern void AT91F_Default_FIQ_handler(void);

void LowLevelInit(void)
{
    int          i;
    AT91PS_PMC   pPMC = AT91C_BASE_PMC;

    // Set Flash Wait state
    //
    // Note: MCK period = 1 / 48054841 hz = 20.0809 nsec
    // FMCN = number of Master clock cycles in 1 microsecond = 1.0 usec/ 20.08095 nsec = 50 (rounded up)
    //
    // FWS = flash wait states = 1 for 48 Mhz operation (FWS = 1)
    // note: see page 656 of AT91SAM7XC512/256/128 Preliminary User Guide
    //
    // result: 0xFFFFF60 = 0x00300100 (AT91C_BASE_MC->MC_FMR = MC Flash Mode Register)
    AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN)&(50 <<16)) | AT91C_MC_FWS_1FWS;

    // Watchdog Disable
    //
    // result: 0xFFFFD44 = 0x00008000 (AT91C_BASE_WDTC->WDTC_WDMR = Watchdog Mode Register)
    AT91C_BASE_WDTC->WDTC_WDMR = AT91C_WDTC_WDDIS;

    // Enable the Main Oscillator
    //
    // Give the Main Oscillator 1.5 msec to start up
    // Main oscillator startup time = SlowClockPeriod * 8 * OSCOUNT
    // SlowClockPeriod = 1 / 42000 = .0000238 sec (worst case RC clock)
    // OSCOUNT = 8
    // MOS startup time = 23.8 usec * 8 * OSCOUNT = .0000238 * 8 * 8 = 1.5 msec
    //
    // MOSCEN = 1 (enables main oscillator)
    //
    // result: 0xFFFFC20 = 0x00000801 (pPMC->PMC_MOR = Main Oscillator Register)
    pPMC->PMC_MOR = (( AT91C_CKGR_OSCOUNT & (0x08 <<8) | AT91C_CKGR_MOSCEN ));

    // Wait the startup time (until PMC Status register MOSCEN bit is set)
    // result: 0xFFFFC68 bit 0 will set when main oscillator has stabilized
    while(!(pPMC->PMC_SR & AT91C_PMC_MOSCS));
}
```

```

// PMC Clock Generator PLL Register setup
//
// The following settings are used:  DIV = 14
//                                  MUL = 72
//                                  PLLCOUNT = 10
//
// Main Clock (MAINCK from crystal oscillator) = 18432000 hz (see AT91SAM7-EK schematic)
// Note: input freq to PLL must be 1 Mhz to 32 Mhz so 18.432 Mhz is OK
//
// MAINCK / DIV = 18432000/14 = 1316571 hz
// PLLCK = 1316571 * (MUL + 1) = 1316571 * (72 + 1) = 1316571 * 73 = 96109683 hz
//
// PLLCOUNT = number of slow clock cycles before the LOCK bit is set in PMC_SR after CKGR_PLLR is written.
//
// PLLCOUNT = 10
//
// OUT = 0 (sets allowable range of PLL output freq from 80 Mhz to 160 Mhz ---> 96.109683 Mhz is OK)
//
// result: 0xFFFFFC2C = 0x0000000048200E   (pPMC->PMC_PLLR = PLL Register)
pPMC->PMC_PLLR = ((AT91C_CKGR_OUT_0) |
                 (AT91C_CKGR_DIV & 14) |
                 (AT91C_CKGR_PLLCOUNT & (40<<10)) |
                 (AT91C_CKGR_MUL & (72<<16)));

// Wait the startup time (until PMC Status register LOCK bit is set)
// result: 0xFFFFFC68 bit 2 will set when PLL has locked
while(!(pPMC->PMC_SR & AT91C_PMC_LOCK));

// PMC Master Clock (MCK) Register setup
//
// CSS = 3 (PLLCK clock selected)
//
// PRES = 1 (MCK = PLLCK / 2) = 96109683/2 = 48054841 hz
//
// Note: Master Clock MCK = 48054841 hz (this is the CPU clock speed)
// result: 0xFFFFFC30 = 0x00000004 (pPMC->PMC_MCKR = Master Clock Register)
pPMC->PMC_MCKR = AT91C_PMC_PRES_CLK_2;

// Wait the startup time (until PMC Status register MCKRDY bit is set)
// result: 0xFFFFFC68 bit 3 will set when Master Clock has stabilized
while(!(pPMC->PMC_SR & AT91C_PMC_MCKRDY));

// result: 0xFFFFFC30 = 0x00000007 (pPMC->PMC_MCKR = Master Clock Register)
pPMC->PMC_MCKR |= AT91C_PMC_CSS_PLL_CLK;

// Wait the startup time (until PMC Status register MCKRDY bit is set)
// result: 0xFFFFFC68 bit 3 will set when Master Clock has stabilized
while(!(pPMC->PMC_SR & AT91C_PMC_MCKRDY));

// Set up the default interrupts handler vectors
AT91C_BASE_AIC->AIC_SVR[0] = (int) AT91F_Default_FIQ_handler;
for (i=1; i < 31; i++) {
    AT91C_BASE_AIC->AIC_SVR[i] = (int) AT91F_Default_IRQ_handler;
}

```


Main.c

This very simple main program sets up pin PA3 to drive a LED to act as a background activity indicator.

It also sets up the USART0 for interrupt-driven operations and enables global interrupts so that the appearance of a incoming serial character will cause a USART0 interrupt.

The main program then falls into an endless idle loop blinking the LED.

```
// *****
//
//                               main.c
//
//   Interrupt-driven USART0 demonstration program for Olimex SAM7-EX256 Evaluation Board
//
//   This simple demo reads 10 characters from USART0 (9600 baud, 8 data bits, 1 stop bit, no parity)
//   When 10 characters are read, they are transmitted back to the source.
//
//   Use standard RS-232 serial cable and the Windows HyperTerm program to test.
//
//   Blinks LED (pin PA3) with an endless loop
//   PA3 is pin 1 on the EXT 20-pin connector (3.3v is pin 18)
//
//   The Olimex SAM7-EX256 board has no programmable LEDs.
//   Added a simple test LED from Radio Shack as shown below (can be attached to the 20-pin EXT connector.)
//
//   3.3 volts |-----|                anode |----|                PA3
//   EXT 0-----| 470 ohm |-----| LED |-----| 0 EXT
//   Pin 18    |-----|                |----| cathode         pin 1
//
//                               Radio Shack Red LED
//                               276-026 T-1 size (anode is the longer wire)
//
//   LED current:  $I = E/R = 3.3/470 = .007$  amps = 7 ma
//   Note: most PIO pins can drive 8 ma on the AT91SAM7X256, so we're OK
//
//
// Author: James P Lynch June 22, 2008
// *****

// *****
//                               Header Files
// *****
#include "AT91SAM7X256.h"
#include "board.h"

// *****
//                               External References
// *****
extern void LowLevelInit(void);
extern void USART0Setup(void);
extern unsigned enableIRQ(void);

int main (void) {

    unsigned long    j;
    unsigned int     IdleCount = 0;

    // Initialize the Atmel AT91SAM7S256 (watchdog, PLL clock, default interrupts, etc.)
    LowLevelInit();

    // Set up the LED (PA3)
    volatile AT91PS_PIO    pPIO = AT91C_BASE_PIOA; // pointer to PIO data structure
    pPIO->PIO_PER = LED_MASK; // PIO Enable Register - allow PIO to control pin PP3
    pPIO->PIO_OER = LED_MASK; // PIO Output Enable Register - sets pin P3 to outputs
    pPIO->PIO_SODR = LED_MASK; // PIO Set Output Data Register - turns off the LED

    // set up USART0
    USART0Setup();

    // enable global interrupts
    enableIRQ();
}
```

```

// *****
// * endless blink loop *
// *****
while (1) {
    if ((pPIO->PIO_ODSR & LED4) == LED4) // read previous state of LED4
        pPIO->PIO_CODR = LED4; // turn LED4 (DS1) on
    else
        pPIO->PIO_SODR = LED4; // turn LED4 (DS1) off

    for (j = 1000000; j != 0; j-- ); // wait 1 second 1000000

    IdleCount++; // count # of times through the idle loop
}
}

```

Usart0_isr.c

This C language interrupt handler is called by the IRQ handler in the startup routine “crt.s”. It reads incoming characters and saves them in a common buffer. When ten characters have been received, the handler switches to “transmit” mode and sends the collected ten characters back to the source.

When that sequence finishes, the handler sets itself up for “receive” mode and waits for the next incoming character to appear. Receive ten characters, transmit same ten characters – it does that forever.

```

// *****
//                               usart0_isr.c
//
//  USART0 Interrupt Service Routine
//
//  This demonstration is designed to read 10 characters into a buffer.
//  After the 10th character arrives, transmit the 10 characters back.
//
//  The application is interrupt-driven.
//
//  Author: James P Lynch June 22, 2008
//  *****

// *****
//                               Header Files
// *****
#include "at91sam7x256.h"
#include "board.h"

// *****
//                               Global Variables
// *****
char      Buffer[32]; // holds received characters
unsigned long nChars = 0; // counts number of received chars
char      *pBuffer = &Buffer[0]; // pointer into Buffer

void Usart0IrqHandler (void) {

    volatile AT91PS_USART pUsart0 = AT91C_BASE_US0; // create a pointer to USART0 structure

    // determine which interrupt has occurred
    // assume half-duplex operation here, only one interrupt type at a time
    if ((pUsart0->US_CSR & AT91C_US_RXRDY) == AT91C_US_RXRDY) {

        // we have a receive interrupt,
        // remove it from Receiver Holding Register and place into buffer[]
        *pBuffer++ = pUsart0->US_RHR;
        nChars++;

        // check if 10 characters have been received
    }
}

```

```

        if (nChars >= 10) {

            // yes, redirect buffer pointer to beginning
            pBuffer = &Buffer[0];
            nChars = 0;

            // disable the receive interrupt, enable the transmit interrupt
            pUsart0->US_IER = AT91C_US_TXEMPTY;    // enable TXEMPTY usart0 transmit interrupt
            pUsart0->US_IDR = ~AT91C_US_TXEMPTY;   // disable all interrupts except TXEMPTY

            // send first received character, TXEMPTY interrupt will send the rest
            pUsart0->US_THR = *pBuffer++;
            nChars++;

        }

    } else if ((pUsart0->US_CSR & AT91C_US_TXEMPTY) == AT91C_US_TXEMPTY) {

        // we have a transmit interrupt (previous char has clocked out)
        // check if 10 characters have been transmitted
        if (nChars >= 10 ) {

            // yes, redirect buffer pointer to beginning
            pBuffer = &Buffer[0];
            nChars = 0;

            // enable receive interrupt, disable the transmit interrupt
            pUsart0->US_IER = AT91C_US_RXRDY;    // enable RXRDY usart0 receive interrupt
            pUsart0->US_IDR = ~AT91C_US_RXRDY;   // disable all interrupts except RXRDY

        } else {

            // no, send next character
            pUsart0->US_THR = *pBuffer++;
            nChars++;

        }

    }

}

```

Usart0_Setup.c

This initialization module sets up the USART0 for interrupt operation and sets up the AIC to process a USART0 interrupt.

```

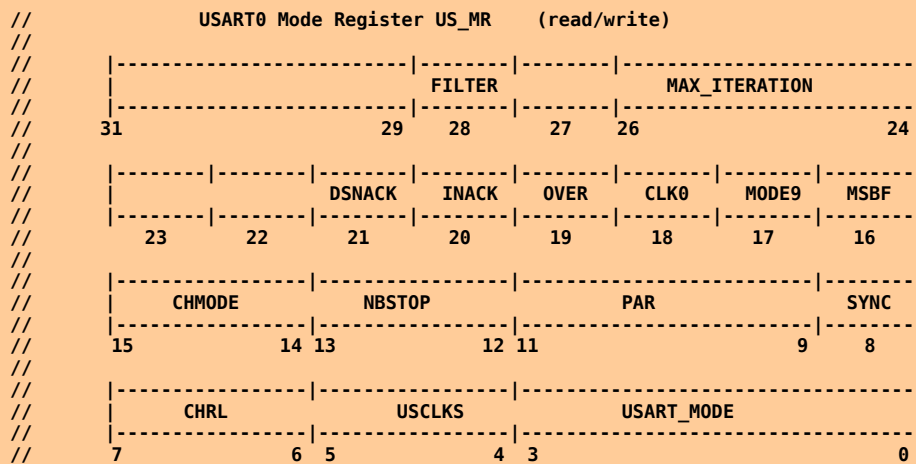
// *****
//                                     usart0_setup.c
//
// Purpose: Set up USART0 (peripheral ID = 6) 9600 baud, 8 data bits, 1 stop bit, no parity
//
// We will use the onboard baud rate generator to specify 9600 baud
//
// The Olimex SAM7-EX256 board has a 18,432,000 hz crystal oscillator.
//
// MAINCK = 18432000 hz (from Olimex schematic)
// DIV = 14 (set up in lowlevelinit.c)
// MUL = 72 (set up in lowlevelinit.c)
//
// PLLCK = (MAINCK / DIV) * (MUL + 1) = 18432000/14 * (72 + 1)
// PLLCLK = 1316571 * 73 = 96109683 hz
// MCK = PLLCLK / 2 = 96109683 / 2 = 48054841 hz
//
// Baud Rate (asynchronous mode) = MCK / (8(2 - OVER)CD)
//
// MCK = 48054841 hz (set USCLKS = 00 in USART Mode Register US_MR - to select MCK only)
// VER = 0 (bit 19 of the USART Mode Register US_MR)
// CD = divisor (USART Baud Rate Generator Register US_BRGR)
// baudrate = 9600 (desired)
//
//

```



```
// RSTRX = 1          (reset receiver)
// RSTTX = 1          (reset transmitter)
// RXEN = 0           (receiver enable - no effect)
// RXDIS = 1          (receiver disable - disabled)
// TXEN = 0           (transmitter enable - no effect)
// TXDIS = 1          (transmitter disable - disabled)
// RSTSTA = 0         (reset status bits - no effect)
// STTBK = 0          (start break - no effect)
// STPBK = 0          (stop break - no effect)
// STTTO = 0          (start time-out - no effect)
// SENDA = 0          (send address - no effect)
// RSTIT = 0          (reert iterations - no effect)
// RSTNACK = 0        (reset non acknowledge - no effect)
// RETTO = 0          (rearm time-out - no effect)
// DTREN = 0          (data terminal ready enable - no effect)
// DTRDIS = 0         (data terminal ready disable - no effect)
// RTSEN = 0          (request to send enable - no effect)
// RSTDIS = 0         (request to send disable - no effect)
```

```
pUSART0->US_CR = AT91C_US_RSTRX | // reset receiver
                  AT91C_US_RSTTX | // reset transmitter
                  AT91C_US_RXDIS | // disable receiver
                  AT91C_US_TXDIS; // disable transmitter
```



```
// USART_MODE = 0000 normal
// USCLKS = 00 choose MCK for baud rate generator
// CHRL = 11 8-bit characters
// SYNC = 0 asynchronous mode
// PAR = 100 no parity
// NBSTOP = 00 1 stop bit
// CHMODE = 00 normal mode, no loop-back, etc.
// MSBF = 0 LSB sent/received first
// MODE9 = 0 CHRL defines character length
// CLK0 = 0 USART does not drive SCK pin
// OVER = 0 16 x oversampling (see baud rate equation)
// INACK = 0 NACK (not used)
// DSNACK = 0 not used since NACK is not generated
// MAX_ITERATION = 0 max iterations not used
// FILTER = 0 filter is off
```

```
pUSART0->US_MR = AT91C_US_PAR_NONE | // no parity
                 0x3 << 6; // 8-bit characters
```

```

//          USART0 Interrupt Enable Register US_IER    (write only)
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          31      30      29      28      27      26      25      24
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          23      22      21      20      19      18      17      16
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          15      14      13      12      11      10      9      8
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          7        6        5        4        3        2        1        0
//
pUSART0->US_IER = 0x00;    // no usart0 interrupts enabled (no effect)

```

```

//          USART0 Interrupt Disable Register US_IDR   (write only)
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          31      30      29      28      27      26      25      24
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          23      22      21      20      19      18      17      16
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          15      14      13      12      11      10      9      8
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          7        6        5        4        3        2        1        0
//
pUSART0->US_IDR = 0xFFFF; // disable all USART0 interrupts

```

```

//          USART0 Interrupt Mask Register US_IDR     (write only)
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          31      30      29      28      27      26      25      24
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          23      22      21      20      19      18      17      16
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          15      14      13      12      11      10      9      8
//
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|-----|-----|
//          7        6        5        4        3        2        1        0
//
// read only, nothing to set up here

```

```

//      USART0 Receive Holding Register US_RHR    (read only)
//
//      |-----|
//      |-----|
//      |-----|
//      31                                     24
//
//      |-----|
//      |-----|
//      23                                     16
//
//      |-----|-----|-----|-----|
//      | RXSYNH |-----|-----|-----| RXCHR |
//      15    14                                     9    8
//
//      |-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----| Y
//      7                                                     0
//      this is where any incoming character will be

//      USART0 Transmit Holding Register US_THR    (write only)
//
//      |-----|
//      |-----|
//      |-----|
//      31                                     24
//
//      |-----|
//      |-----|
//      23                                     16
//
//      |-----|-----|-----|-----|
//      | TXSYNH |-----|-----|-----| TXCHR |
//      15    14                                     9    8
//
//      |-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----| Y
//      7                                                     0
//      this is where we place characters to be transmitted

//      USART0 Baud Rate Generator Register US_BRGR    (read/write)
//
//      |-----|
//      |-----|
//      |-----|
//      31                                     24
//
//      |-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----| FP
//      23                                     19 18                                     16
//
//      |-----|-----|-----|-----|
//      |-----|-----|-----|-----|
//      15                                     8
//
//      |-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----| Y
//      7                                                     0
//
pUSART0->US_BRGR = 0x139;           // CD = 0x139 (313 from above calculation) FP=0 (not used)

```

```

//          USART0 Receiver Time-out Register US_RTOR    (read/write)
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  31      30      29      28      27      26      25      24
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  23      22      21      20      19      18      17      16
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  15                                                    9
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  7                                                    0

```

```

pUSART0->US_RTOR = 0;          // receiver time-out (disabled)

```

```

//          USART0 transmitter TimeGuard Register US_TTGR  (read/write)
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  31      30      29      28      27      26      25      24
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  23      22      21      20      19      18      17      16
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  15      14      13      12      11      10      9      8
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  7                                                    0

```

```

pUSART0->US_TTGR = 0;          // transmitter timeguard (disabled)

```

```

//          USART0 FI DI Ratio Register US_FIDI    (read/write)
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  31      30      29      28      27      26      25      24
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  23      22      21      20      19      18      17      16
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  15      14      13      12      11      10      9      8
//
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  |-----|-----|-----|-----|-----|-----|-----|-----|
//  7                                                    0

```

```

// not used, nothing to set up here

```



```

//          USART0 Number of Errors Register   US_NER   (read only)
//
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          31                                     24
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          23                                     16
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          15                                     8
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          7                                     0
//          NB_ERRORS
// Read-only, nothing to set up here

```

```

//          USART0 IrDA Filter Register   US_IF   (read/write)
//
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          31                                     24
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          23                                     16
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          15                                     8
//
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          |-----|
//          7                                     0
//          IRDA_FILTER
// not used, nothing to set up here

```

```

// Set up the Advanced Interrupt Controller (AIC) registers for USART0
volatile AT91PS_AIC pAIC = AT91C_BASE_AIC; // pointer to AIC data structure

pAIC->AIC_IDCR = (1<<AT91C_ID_US0); // Disable USART0 interrupt in AIC

pAIC->AIC_SVR[AT91C_ID_US0] = // Set the USART0 IRQ handler address in AIC Source
    (unsigned int)Usart0IrqHandler; // Vector Register[6]

pAIC->AIC_SMR[AT91C_ID_US0] = // Set the interrupt source type(level-sensitive) and
    (AT91C_AIC_SRCTYPE_INT_HIGH_LEVEL | 0x4 ); // priority (4) in AIC Source Mode Register[6]

pAIC->AIC_IECR = (1<<AT91C_ID_US0); // Enable the USART0 interrupt in AIC

// enable the USART0 receiver and transmitter
pUSART0->US_CR = AT91C_US_RXEN | AT91C_US_TXEN;

// enable the USART0 receive interrupt
pUSART0->US_IER = AT91C_US_RXRDY; // enable RXRDY usart0 receive interrupt
pUSART0->US_IDR = ~AT91C_US_RXRDY; // disable all interrupts except RXRDY

// set up buffer pointer and character counter
pBuffer = &Buffer[0];
nChars = 0;

// enable IRQ interrupts
enableIRQ();

// at this point, only the USART0 receive interrupt is armed!
}

```

Demo_sam7x256.cmd

This linker script file is almost exactly the same as the linker script in the tutorial “Using Open Source Tools for AtmelAT91SAM7 Cross Development”. It basically instructs the linker where to place the code and data in memory.

```
/* ***** */
/* demo_sam7x256.cmd LINKER SCRIPT */
/* */
/* The Linker Script defines how the code and data emitted by the GNU C compiler and assembler are */
/* to be loaded into memory (code goes into FLASH, variables go into RAM). */
/* */
/* Any symbols defined in the Linker Script are automatically global and available to the rest of the */
/* program. */
/* */
/* To force the linker to use this LINKER SCRIPT, just add the -T demo_sam7ex256.cmd */
/* directive to the linker flags in the makefile. For example, */
/* */
/* LFLAGS = -Map main.map -nostartfiles -T demo_sam7ex256.cmd */
/* */
/* The order that the object files are listed in the makefile determines what .text section is */
/* placed first. */
/* */
/* For example: $(LD) $(LFLAGS) -o main.out crt.o main.o lowlevelinit.o */
/* */
/* crt.o is first in the list of objects, so it will be placed at address 0x00000000 */
/* */
/* The top of the stack (_stack_end) is (last_byte_of_ram + 1) - 4 */
/* */
/* Therefore: _stack_end = (0x00020FFFF + 1) - 4 = 0x00021000 - 4 = 0x0020FFFC */
/* */
/* Note that this symbol (_stack_end) is automatically GLOBAL and will be used by the crt.s */
/* startup assembler routine to specify all stacks for the various ARM modes */
/* */
/* MEMORY MAP */
/* */
/* -----> |-----| 0x00210000 */
/* . | UDF Stack 16 bytes | 0x0020FFFC <----- _stack_end */
/* . | | 0x0020FFEC */
/* . | ABT Stack 16 bytes | */
/* . | | 0x0020FFDC */
/* . | FIQ Stack 128 bytes | */
/* . | | 0x0020FF5C */
/* RAM | IRQ Stack 128 bytes | */
/* . | | 0x0020FEDC */
/* . | SVC Stack 16 bytes | */
/* . | | 0x0020FECC */
/* . | | */
/* */
```

```

/*
/*      .      |      stack area for user program      |
/*      .      |      |
/*      .      |      free ram      |
/*      .      |      |
/*      .      |      .....|0x002006D8 <----- _bss_end      |
/*      .      |      |
/*      .      |      .bss uninitialized variables      |
/*      .      |      .....|0x002006D0 <----- _bss_start, _edata      |
/*      .      |      |
/*      .      |      .data initialized variables      |
/*      .      |      |
/*      .----->|      |0x00200000      |
/*
/*
/*      .----->|      |0x00040000      |
/*      .      |      |
/*      .      |      free flash      |
/*      .      |      |
/*      .      |      .....|0x000006D0 <----- _bss_start, _edata      |
/*      .      |      |
/*      .      |      .data initialized variables      |
/*      .      |      |
/*      .      |      .....|0x000006C4 <----- _etext      |
/*      .      |      |
/*      .      |      C code      |
/*      .      |      |
/*      .      |      .....|0x00000118 main()      |
/*      .      |      |
/*      .      |      Startup Code (crt.s)      |
/*      .      |      (assembler)      |
/*      .      |      |
/*      .      |      .....|0x00000020      |
/*      .      |      |
/*      .      |      Interrupt Vector Table      |
/*      .      |      32 bytes      |
/*      .----->|      |0x00000000 _vec_reset      |
/*
/*
/* Author: James P. Lynch   June 22, 2008
/*
/* *****
/*
/* identify the Entry Point (_vec_reset is defined in file crt.s) */
ENTRY(_vec_reset)

/* specify the AT91SAM7X256 memory areas */
MEMORY
{
    flash    : ORIGIN = 0,           LENGTH = 256K    /* FLASH EPROM      */
    ram      : ORIGIN = 0x200000,    LENGTH = 64K     /* static RAM area  */
}

/* define a global symbol _stack_end (see analysis in annotation above) */
_stack_end = 0x20FFFC;

/* now define the output sections */
SECTIONS
{
    . = 0;                               /* set location counter to address zero */

    .text :                               /* collect all sections that should go into FLASH after startup */
    {
        *(.text)                          /* all .text sections (code) */
        *(.rodata)                        /* all .rodata sections (constants, strings, etc.) */
        *(.rodata*)                       /* all .rodata* sections (constants, strings, etc.) */
        *(.glue_7)                        /* all .glue_7 sections (no idea what these are) */
        *(.glue_7t)                       /* all .glue_7t sections (no idea what these are) */
        _etext = .;                       /* define a global symbol _etext just after the last code byte */
    } >flash                             /* put all the above into FLASH */

```

```

.data :                               /* collect all initialized .data sections that go into RAM */
{
    _data = .;                         /* create a global symbol marking the start of the .data section */
    *(.data)                           /* all .data sections */
    _edata = .;                       /* define a global symbol marking the end of the .data section */
} >ram AT >flash                       /* put all the above into RAM (but load the LMA initializer copy into
FLASH) */

.bss :                                 /* collect all uninitialized .bss sections that go into RAM */
{
    _bss_start = .;                   /* define a global symbol marking the start of the .bss section */
    *(.bss)                           /* all .bss sections */
} >ram                                  /* put all the above in RAM (it will be cleared in the startup code */

. = ALIGN(4);                          /* advance location counter to the next 32-bit boundary */
_bss_end = .;                          /* define a global symbol marking the end of the .bss section */
}

_end = .;                              /* define a global symbol marking the end of application RAM */

```

Makefile

The makefile is similar to the one in the “*Using Open Source Tools...*” tutorial. One helpful change is that “implicit” rules are used to do the assemble and multiple compiles. Normally, to compile a C file, you specify a “target: prerequisites” line followed by a “command” line that has been indented by a tab.

```

main.o:    main.c at91sam7x256.h board.h
           arm-elf-gcc -I./ -c -fno-common -O0 -g main.c

```

The Make manual explains on page 77 that you can skip the specification of the “command” line and let Make deduce the operation needed by just inspecting the file extensions in the dependency line. This being the case, all we need is the dependency line above by itself.

```

main.o:    main.c at91sam7x256.h board.h

```

The Make utility deduces that because you have a C object file and a C source file (looking at the file extensions), you need to run the C compiler.

Now, there are a couple of things to remember.

For compiling C programs, Make creates a command line like this:

```
$(CC) -c $(CPPFLAGS) $(CFLAGS) source.c.
```

For an assembler program, Make creates a command line like this:

```
$(AS) $(ASFLAGS) source.s
```

These are built-in variables used by Make in utilizing explicit rules. If you look at this makefile's variables, we use the very same variable names to identify the compiler, assembler, etc. This makes it very easy to add files to the makefile – just update the OBJECTS list and the “dependency” lines.

```

# *****
# Makefile for Atmel AT91SAM7X256 - flash execution
#
# Description of Compiler Flags (CFLAGS)
# -mcpu=arm7tdmi identifies the target ARM microprocessor
# -I./ search current working directory for include files
# -c do not invoke the linker
# -fno-common compiler gives each global variable space in .data segment
# -O0 set lowest optimization level (best for debugging)
# -g include debugging information in output file
# -fomit-frame-pointer don't store frame pointer for functions that don't need it
# -Wcast-align emit warning if casting pointer causes alignment problems
#
# -MD emits a one-line file such as main.d file with dependency line like this:
# main.o: main.c at91sam7x256.h board.h
# note: use -MD once to get your dependency lines set up - then remove.
#
# Description of Assembler Flags (ASFLAGS)
# -mapcs-32 select apcs-32 ARM procedure calling convention
# -g include debugging information in output file
#
# Description of Linker flags (LFLAGS)
# -omain.out set the output filename to "main.out"
# -Tdemo_sam7ex256.cmd identifies the linker script file
# -Map main.map create a map file with the name "main.map"
# --cref add cross reference table to the map file
#
# Description of ObjCopy flags (CPFLAGS)
# --output-target=binary convert main.out to a binary file (main.bin)
#
# Description of ObjDump flags (ODFLAGS)
# -x display header information, symbol table etc.
# --syms display the symbol table
#
# James P Lynch June 22, 2008
# *****

NAME = demo_sam7ex256

# variables
CC = arm-elf-gcc
AS = arm-elf-as
LD = arm-elf-ld -v
CP = arm-elf-objcopy
OD = arm-elf-objdump

CFLAGS = -mcpu=arm7tdmi -I./ -c -fno-common -O0 -g -fomit-frame-pointer -Wcast-align
ASFLAGS = -mapcs-32 -g
LFLAGS = -omain.out -Tdemo_sam7ex256.cmd -Map main.map --cref
CPFLAGS = --output-target=binary
ODFLAGS = -x --syms

OBJECTS = crt.o \
          main.o \
          lowlevelinit.o \
          usart0_setup.o \
          usart0_isr.o \
          isrsupport.o

# ALL - make target called by Eclipse (Project -> Build Project)
all: main.out
    @ echo "...create binary file"
    $(CP) $(CPFLAGS) main.out main.bin
    @ echo "...create dump file"
    $(OD) $(ODFLAGS) main.out > main.dmp

```

Add any additional source files to this list



```

main.out: $(OBJECTS)
    @ echo "...linking"
    $(LD) $(LFLAGS) -omain.out $(OBJECTS) libgcc.a

# list of dependencies for each C and ASM file in the project
# Note: Implicit Rules will deduce using source file extension which to run: C compiler or ARM assembler

crt.o:          crt.s
main.o:         main.c at91sam7x256.h board.h
lowlevelinit.o: lowlevelinit.c at91sam7x256.h Board.h
usart0_setup.o: usart0_setup.c at91sam7x256.h board.h
usart0_isr.o:   usart0_isr.c at91sam7x256.h board.h
isrsupport.o:  isrsupport.c

# CLEAN - make target called by Eclipse (Project -> Clean ...)
clean:
    -rm $(OBJECTS) main.out main.bin main.map main.dmp

# *****
#                               FLASH PROGRAMMING
# *****
#
# Alternate make target for flash programming only
#
# You must create a special Eclipse make target (program) to run this part of the makefile
# (Project -> Create Make Target... then set the Target Name and Make Target to "program")
#
# OpenOCD is run in "batch" mode with a special configuration file and a script file containing
# the flash commands. When flash programming completes, OpenOCD terminates.
#
# Note that the script file of flash commands (script.ocd) is part of the project
#
# Programmers: Martin Thomas, Joseph M Dupre, James P Lynch
# *****

# specify output filename here (must be *.bin file)
TARGET = main.bin

# specify the directory where openocd executable resides
OPENOCD_DIR = 'c:/Program Files/openocd-r657/bin/'

# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debuggers)
#OPENOCD = $(OPENOCD_DIR)openocd-pp.exe
OPENOCD = $(OPENOCD_DIR)openocd-ftd2xx.exe

# specify OpenOCD configuration file (must be in your project folder)
OPENOCD_CFG = openocd_program.cfg

# program the AT91SAM7S256 internal flash memory
program: $(TARGET)
    @echo "Flash Programming with OpenOCD..."           # display a message on the console
    $(OPENOCD) -s $(OPENOCD_DIR) -f $(OPENOCD_CFG)      # program the onchip FLASH here
    @echo "Flash Programming Finished."                 # display a message on the console

```

Add any additional source files to this list

Openocd_program.cfg

In this project, I elected to place both OpenOCD configuration files in the project folder. The most recent revisions of OpenOCD seem to have trouble with a space character in the path to the configuration file (c:\Program Files\, for example). The safest thing to do is to place these files in your project (assuming that its path doesn't have embedded space characters) since we know that Windows has a proper path to our project folder.

This OpenOCD configuration file is for Flash Programming. If you create an alternate Make Target called “**program**”, clicking on “**program**” in the make targets view will start the flash programming operation.

This particular OpenOCD configuration file is for the Olimex ARM-USB-OCD JTAG debugger interface. Check my original tutorial for configuration files for the other Amontec and Olimex devices.

```
#define our ports
telnet_port 4444
gdb_port 3333

#commands specific to the Olimex ARM-USB-OCD
interface ft2232
ft2232_device_desc "Olimex OpenOCD JTAG A"
ft2232_layout "olimex-jtag"
ft2232_vid_pid 0x15BA 0x0003
jtag_speed 2
jtag_nsrst_delay 200
jtag_ntrst_delay 200

#reset_config <signals> [combination] [trst_type] [srst_type]
reset_config srst_only srst_pulls_trst

#jtag_device <IR length> <IR capture> <IR mask> <IDCODE instruction>
jtag_device 4 0x1 0xf 0xe

#daemon_startup <'attach' | 'reset'>
daemon_startup reset

#target <type> <endianess> <reset_mode> <jtag#> [variant]
target arm7tdmi little run_and_init 0 arm7tdmi_r4

#run_and_halt_time <target#> <time_in_ms>
run_and_halt_time 0 30

# commands below are specific to AT91sam7 Flash Programming
# -----

#target_script specifies the flash programming script file
target_script 0 reset script.ocd

#working_area <target#> <address> <size> <'backup' | 'nobackup'>
working_area 0 0x00200000 0x4000 nobackup

#flash bank at91sam7 0 0 0 0 <target#>
flash bank at91sam7 0 0 0 0 0
```

Openocd.cfg

This OpenOCD configuration file is used when you wish to run OpenOCD as a debugging agent (daemon). This file is also imported into the project's folder so as to guarantee a proper Windows path to it.

```
#define our ports
telnet_port 4444
gdb_port 3333

#commands specific to the Olimex ARM-USB-OCD
interface ft232
ft232_device_desc "Olimex OpenOCD JTAG A"
ft232_layout "olimex-jtag"
ft232_vid_pid 0x15BA 0x0003
jtag_speed 2
jtag_nsrst_delay 200
jtag_nrst_delay 200

#reset_config <signals> [combination] [trst_type] [srst_type]
reset_config srst_only srst_pulls_trst

#jtag_device <IR length> <IR capture> <IR mask> <IDCODE instruction>
jtag_device 4 0x1 0xf 0xe

#daemon_startup <'attach'|'reset'>
daemon_startup reset

#target <type> <endianess> <reset_mode> <jtag#> [variant]
target arm7tdmi little run_and_init 0 arm7tdmi_r4

#run_and_halt_time <target#> <time_in_ms>
run_and_halt_time 0 30
```

Script.ocd

When you run the Makefile target “**program**”, the openocd_program.cfg file calls this programming script file to manage flash programming. Note the use of the “**flash write_image**” command below. This is a new command for OpenOCD and they removed the “**flash program**” command I used in the previous tutorial.

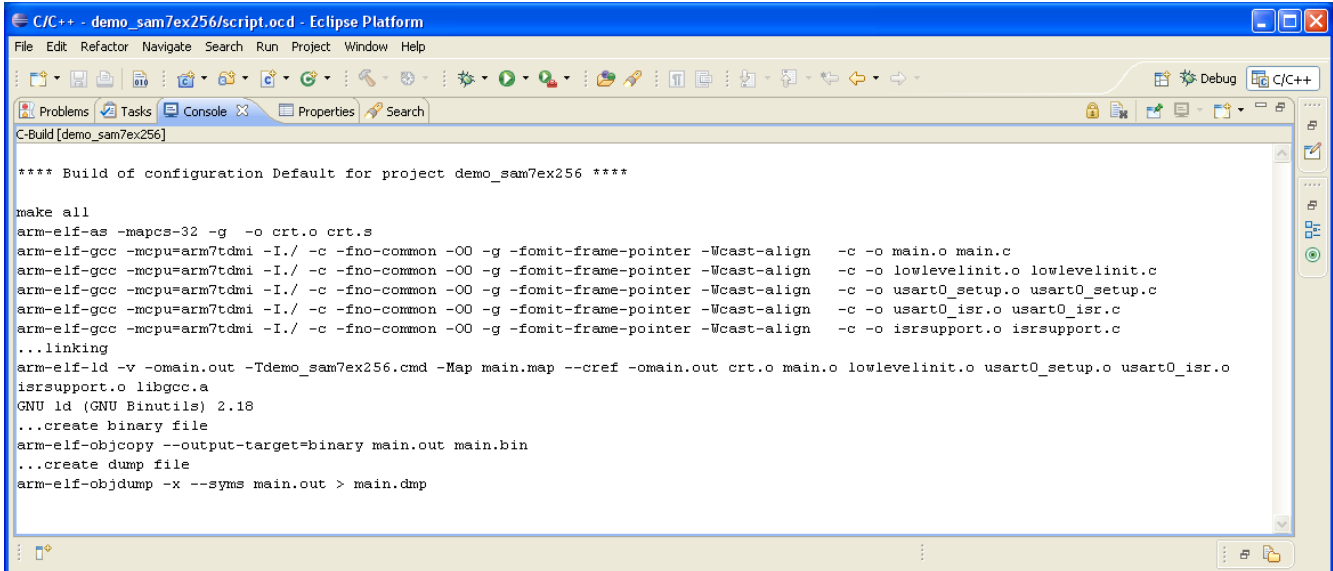
```
# OpenOCD Target Script for Atmel AT91SAM7S256
#
# Programmer: James P Lynch, Martin Thomas
#

wait_halt # halt the processor and wait
armv4_5 core_state arm # select the core state
mww 0xffffffff60 0x00320100 # set flash wait state (AT91C_MC_FMR)
mww 0xfffffd44 0xa0008000 # watchdog disable (AT91C_WDTC_WDMR)
mww 0xfffffc20 0xa0000601 # enable main oscillator (AT91C_PMC_MOR)
sleep 100 # wait 100 ms
mww 0xfffffc2c 0x00480a0e # set PLL register (AT91C_PMC_PLLR)
sleep 200 # wait 200 ms
mww 0xfffffc30 0x7 # set master clock to PLL (AT91C_PMC_MCKR)
sleep 100 # wait 100 ms
mww 0xfffffd08 0xa5000401 # enable user reset AT91C_RSTC_RMR
sleep 10 # wait 10 msec
flash write_image main.bin 0x100000 bin # program the onchip flash
reset run # reset, then let target run
shutdown # stop OpenOCD
```


Building the Project

If you use Eclipse/CDT to create a project named “**demo_sam7ex256**” and import the source files from the attachment to this tutorial, you will have a serial communications project that should build without errors.

When you click the “**Build All**” button, the following console display will reflect building of the project.



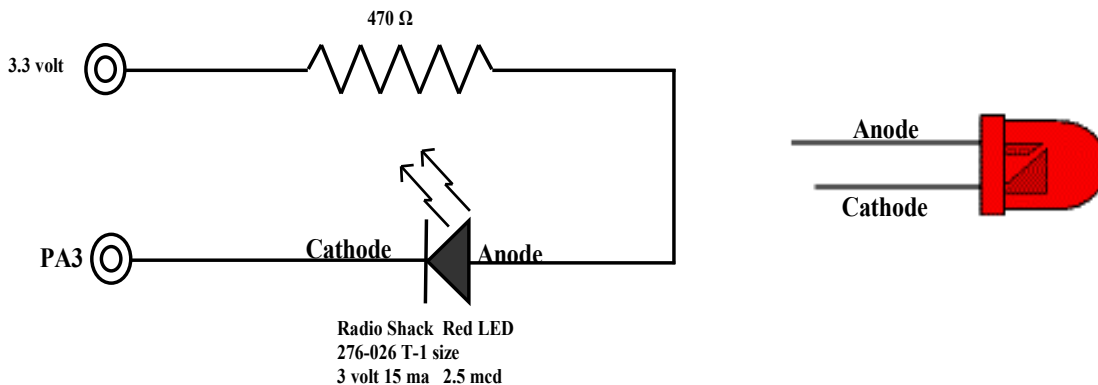
```
C/C++ - demo_sam7ex256/script.ocd - Eclipse Platform
File Edit Refactor Navigate Search Run Project Window Help
C-Build [demo_sam7ex256]

**** Build of configuration Default for project demo_sam7ex256 ****

make all
arm-elf-as -mapcs-32 -g -o crt.o crt.s
arm-elf-gcc -mcpu=arm7tdmi -I./ -c -fno-common -O0 -g -fomit-frame-pointer -Wcast-align -c -o main.o main.c
arm-elf-gcc -mcpu=arm7tdmi -I./ -c -fno-common -O0 -g -fomit-frame-pointer -Wcast-align -c -o lowlevelinit.o lowlevelinit.c
arm-elf-gcc -mcpu=arm7tdmi -I./ -c -fno-common -O0 -g -fomit-frame-pointer -Wcast-align -c -o usart0_setup.o usart0_setup.c
arm-elf-gcc -mcpu=arm7tdmi -I./ -c -fno-common -O0 -g -fomit-frame-pointer -Wcast-align -c -o usart0_isr.o usart0_isr.c
arm-elf-gcc -mcpu=arm7tdmi -I./ -c -fno-common -O0 -g -fomit-frame-pointer -Wcast-align -c -o isrsupport.o isrsupport.c
...linking
arm-elf-ld -v -omain.out -Tdemo_sam7ex256.cmd -Map main.map --cref -omain.out crt.o main.o lowlevelinit.o usart0_setup.o usart0_isr.o
isrsupport.o libgcc.a
GNU ld (GNU Binutils) 2.18
...create binary file
arm-elf-objcopy --output-target=binary main.out main.bin
...create dump file
arm-elf-objdump -x --syms main.out > main.dmp
```

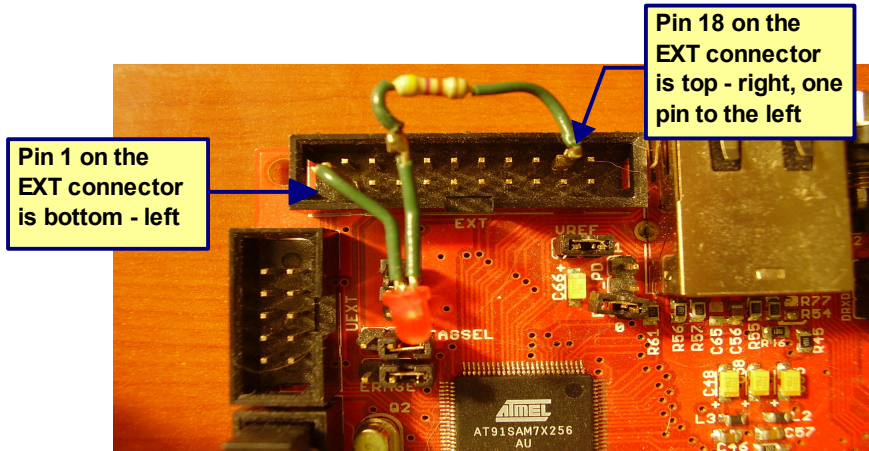
Adding an LED to the Olimex SAM7-EX256 Board

For some unknown reason, the Olimex SAM7-EX256 board doesn't have a user-programmable LED to serve as a background activity indicator. It's fairly easy to add one.



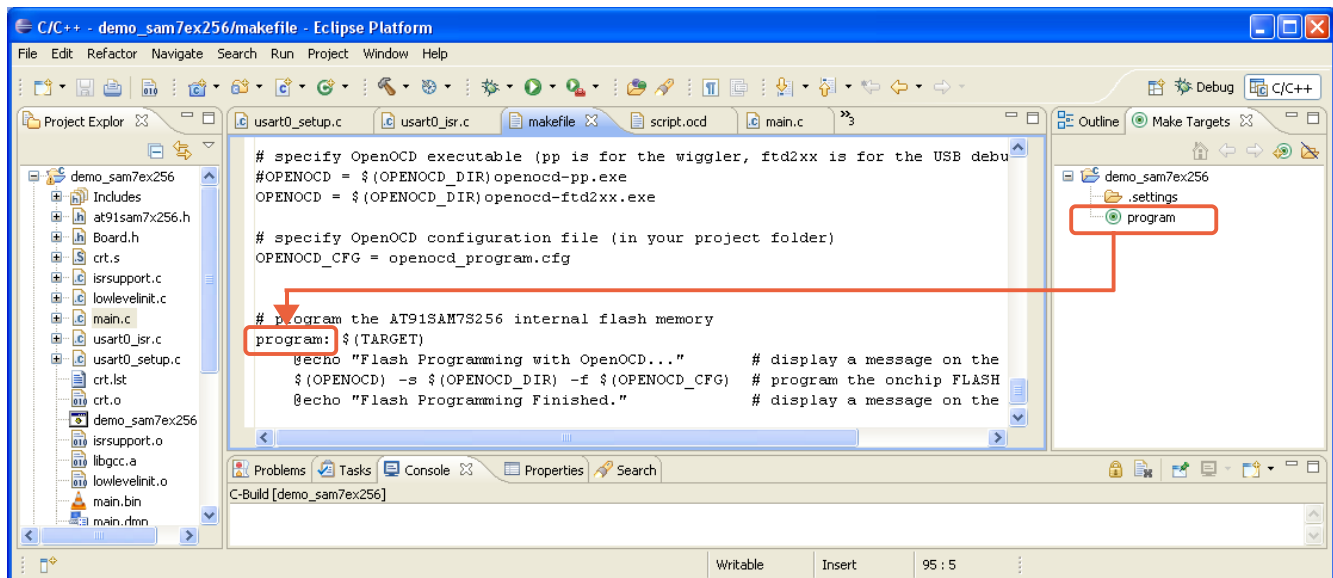
We can use Port PA3 to drive the LED; it can supply 8 ma which is just right for a cheap Radio Shack red LED called out above. You'll need a 470Ω resistor to limit the current to 7ma.

Pin 18 on the EXT connector is 3.3 volts; pin 1 is port PA3 as shown below.



Programming the Sample Application into Flash

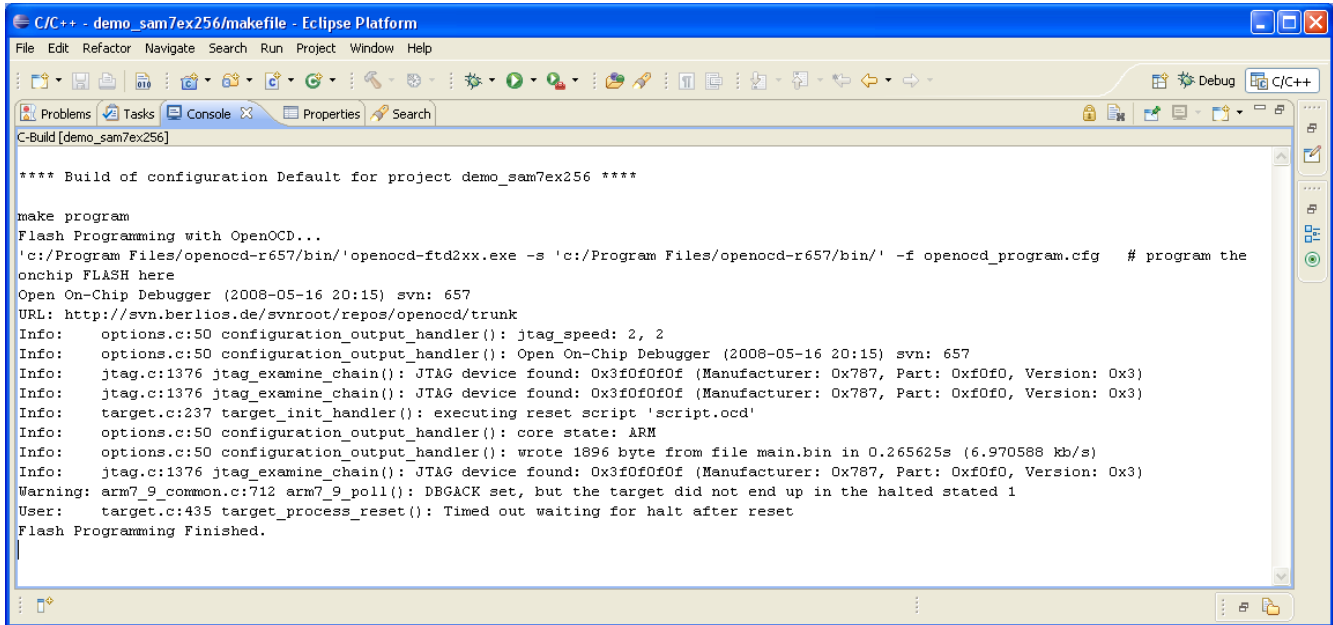
Remember that the Makefile has an alternate target “**program:**” that runs the OpenOCD utility in one-shot mode to program the flash. Note that in the “Make Targets” view on the upper right below, there is a “**program**” target that you can click on. Doing that will execute the “program” target in the makefile and program your onchip flash. See the “**Using Open Source Tools ...**” tutorial for information on how to set up the OpenOCD flash programming facility.



You should see something like the console view shown below. If things are looking good, there will be the LED blinking about once a second due to the main program idle background loop. The thing to look for is:

Info: options.c:50 configuration_output_handler(): wrote 1896 byte from file main.bin in 0.265625s (6.970588 kb/s)

This does indicate that we wrote the **main.bin** file into onchip flash in 1/4 second.



```
C/C++ - demo_sam7ex256/makefile - Eclipse Platform
File Edit Refactor Navigate Search Run Project Window Help
C-Build [demo_sam7ex256]

**** Build of configuration Default for project demo_sam7ex256 ****

make program
Flash Programming with OpenOCD...
'c:/Program Files/openocd-r657/bin/'openocd-ftd2xx.exe -s 'c:/Program Files/openocd-r657/bin/' -f openocd_program.cfg # program the
onchip FLASH here
Open On-Chip Debugger (2008-05-16 20:15) svn: 657
URL: http://svn.berlios.de/svnroot/repos/openocd/trunk
Info: options.c:50 configuration_output_handler(): jtag_speed: 2, 2
Info: options.c:50 configuration_output_handler(): Open On-Chip Debugger (2008-05-16 20:15) svn: 657
Info: jtag.c:1376 jtag_examine_chain(): JTAG device found: 0x3f0f0f0f (Manufacturer: 0x787, Part: 0xf0f0, Version: 0x3)
Info: jtag.c:1376 jtag_examine_chain(): JTAG device found: 0x3f0f0f0f (Manufacturer: 0x787, Part: 0xf0f0, Version: 0x3)
Info: target.c:237 target_init_handler(): executing reset script 'script.ocd'
Info: options.c:50 configuration_output_handler(): core state: ARM
Info: options.c:50 configuration_output_handler(): wrote 1896 byte from file main.bin in 0.265625s (6.970588 kb/s)
Info: jtag.c:1376 jtag_examine_chain(): JTAG device found: 0x3f0f0f0f (Manufacturer: 0x787, Part: 0xf0f0, Version: 0x3)
Warning: arm7_9_common.c:712 arm7_9_poll(): DBGACK set, but the target did not end up in the halted stated 1
User: target.c:435 target_process_reset(): Timed out waiting for halt after reset
Flash Programming Finished.
```

Testing the Interrupt Driven Application

Note again that the application waits for you to type 10 characters and then retransmits them back as a burst. You will need to connect a standard 9-pin serial cable from your desktop computer's COM1 port to the RS-232 connector on the Olimex board and use a serial terminal emulator program to type the ten characters and view the results.

Now it's true that this application can be tested with the standard Windows **Hyper Terminal** utility, located in the "accessories" folder. This Windows program is a good example of "crapware"; programs placed into Windows that are teaser editions of commercial programs that they want you to purchase. The Hyper Terminal program has no screen "clear" button, you have to restart it to get a clear screen. In my view, that disqualifies it!

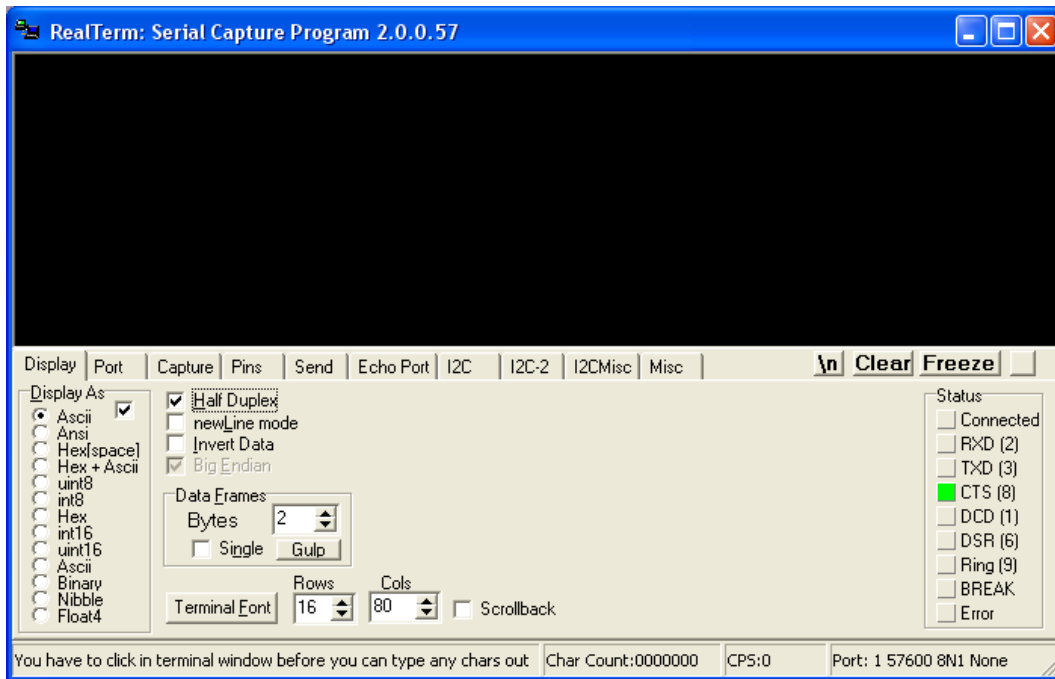
There is a perfectly acceptable Open Source terminal emulation program called "**Realterm**" that can do the job and it has a "clear" button (now your author is happy!). You can download the **Realterm** program from SourceForge using this link:

http://downloads.sourceforge.net/realterm/Realterm_2.0.0.57_setup.exe?modtime=1204263582&big_mirror=0

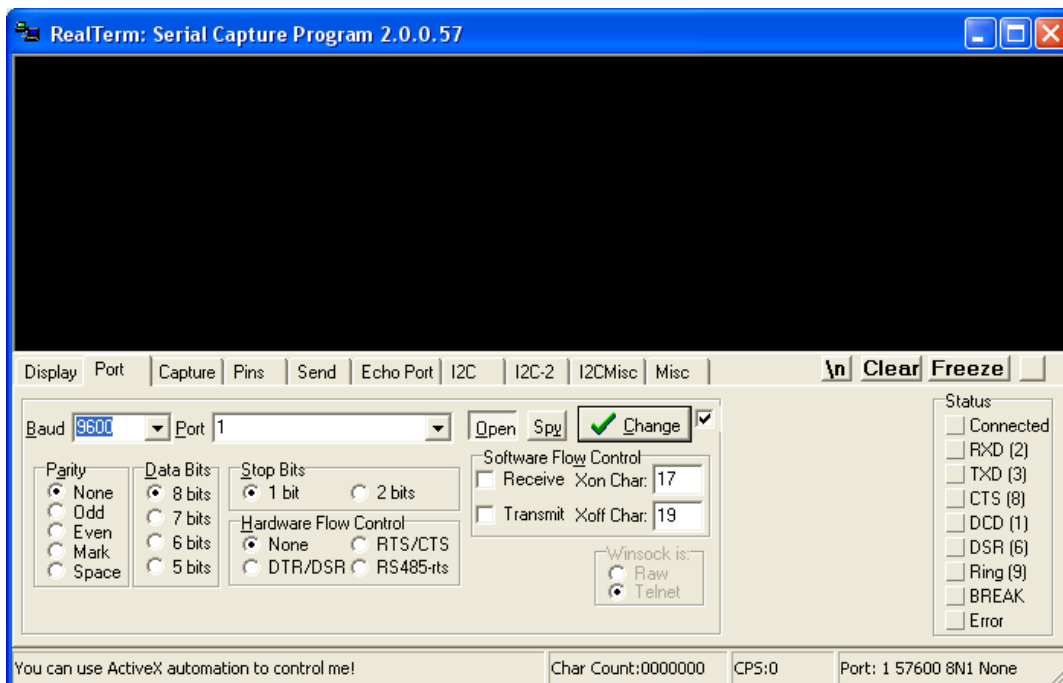
Realterm downloads as an installer executable that will unpack and install **Realterm** on your computer.

When you run **Realterm** for the first time, there will be a couple of settings to deal with to configure it for 9600 baud, 1 start bit, 8 data bits, 1 stop bit and no parity plus half-duplex.

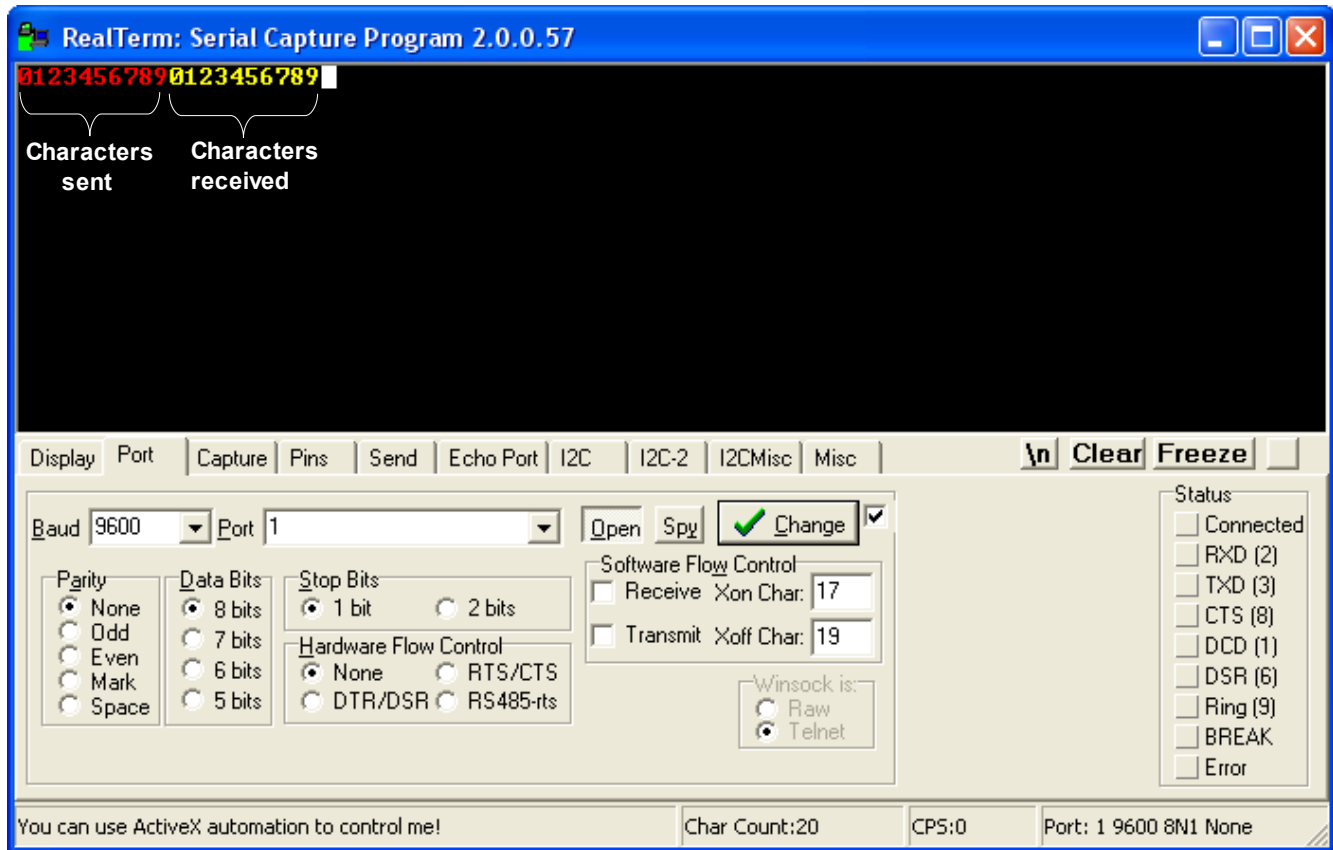
In the opening screen below, Within the “**Display**” tab, click on the “**Half Duplex**” check box to turn on local echo (so you can see what you are typing).



Within the “**Port**” tab, set the baud rate to “**9600**”, set the Port to “**1**” for COM1, and hit the “**Change**” button. On my version, the parity, data bits and stop bits were already set properly.



Now reset the Olimex SAM7-EX256 board and type 10 characters. After the 10th one is entered, the application will send the ten characters back in a burst. You can do this over and over!



This application is "interrupt-driven" where each character, send or receive, causes an interrupt. We put the incoming characters into a common buffer and transmitted the collected characters from the same buffer. You can use your imagination to change this scenario; you could put incoming characters into a circular buffer with put and get pointers, put each character into an RTOS queue, the sky is the limit.

This application caused 20 interrupts; ten for received characters and 10 for transmitted characters. If you use a Direct Memory Access technique to do this, there are only two interrupts (one interrupt for receive "end-of-message" and one interrupt for "transmit" end-of-message. You will be pleased to see that it's not very difficult to modify this example to support DMA operation of the USART0.

Direct Memory Access

Direct memory access (DMA) has been around since the time of the first mainframe computers such as the IBM System 360. Engineers noticed that there were clock cycles where the memory wasn't being accessed (during instruction decode, for example). They designed circuits that would use those spare memory cycles to transfer bytes from the RAM memory to peripherals such as a printer (this was called "cycle stealing in those days). Just about everything we see in microprocessor development today evolved from discoveries made in the heyday of mainframe computers.

The Atmel AT91SAM7X256 chip also has a DMA capability. It allows transfer of bytes from peripherals to memory or vice versa. All you have to do is set up a pointer and a byte count and turn it on. You'll get an interrupt when its finished.

It's even more sophisticated – there can be two pointers and byte counts. This allows use of a ping-pong buffer scheme where when one buffer gets filled up, the DMA controller will automatically switch to the second buffer using the secondary pointer and byte count. This allows you to empty the first buffer while the second one is being filled up – clearly necessary when you high speed communications. For this demonstration, we will not use the second buffer.

DMA capability is provided for the debug serial port, both USART serial ports, both synchronous serial controller (SSI) ports, and both serial peripheral interface (SPI) ports. The Ethernet controller has its own DMA and the USB unit uses a dual-port memory.

USART0 DMA Registers

While there is a chapter in the Atmel AT91SAM7X256 data sheet on DMA, it is very generic. The DMA registers are integrated into the register set of each peripheral supported by the DMA controller. The memory map of the USART0 registers, shown below, refers to the DMA setup registers at the bottom of the list.

Table 30-12. USART Memory Map

Offset	Register	Name	Access	Reset State
0x0000	Control Register	US_CR	Write-only	-
0x0004	Mode Register	US_MR	Read/Write	-
0x0008	Interrupt Enable Register	US_IER	Write-only	-
0x000C	Interrupt Disable Register	US_IDR	Write-only	-
0x0010	Interrupt Mask Register	US_IMR	Read-only	0x0
0x0014	Channel Status Register	US_CSR	Read-only	-
0x0018	Receiver Holding Register	US_RHR	Read-only	0x0
0x001C	Transmitter Holding Register	US_THR	Write-only	-
0x0020	Baud Rate Generator Register	US_BRGR	Read/Write	0x0
0x0024	Receiver Time-out Register	US_RTOR	Read/Write	-
0x0028	Transmitter Timeguard Register	US_TTGR	Read/Write	-
0x2C - 0x3C	Reserved	-	-	-
0x0040	FI DI Ratio Register	US_FIDI	Read/Write	-
0x0044	Number of Errors Register	US_NER	Read-only	-
0x0048	Reserved	-	-	-
0x004C	IrDA Filter Register	US_IF	Read/Write	0x0
0x5C - 0xFC	Reserved	-	-	-
0x100 - 0x128	Reserved for PDC Registers	-	-	-

Here are the DMA registers.

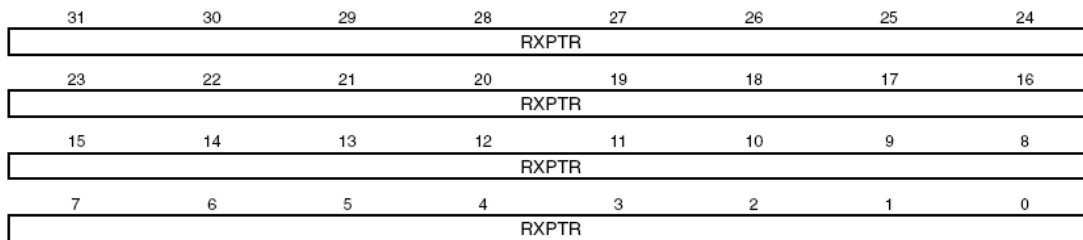
USART0 PDC Receive Pointer Register

The Receive Pointer Register is where you load the address of the buffer that is to receive the incoming characters.

22.4.1 PDC Receive Pointer Register

Register Name: PERIPH_RPR

Access Type: Read/Write



- RXPTR: Receive Pointer Address

Address of the next receive transfer.

The buffer is defined in the file "usart0_isr.c" and can hold 32 bytes. Here we set up a pointer to the USART0 data structure and thus load the Receive Pointer Register with the address of the buffer.

```
extern char      Buffer[];           // holds received characters
extern unsigned long nChars;        // counts number of received chars
extern char      *pBuffer;         // pointer into Buffer

volatile AT91PS_USART pUsart0 = AT91C_BASE_US0; // create a pointer to USART0 structure

pUsart0->US_RPR = (unsigned int)Buffer; // address of DMA input buffer
```

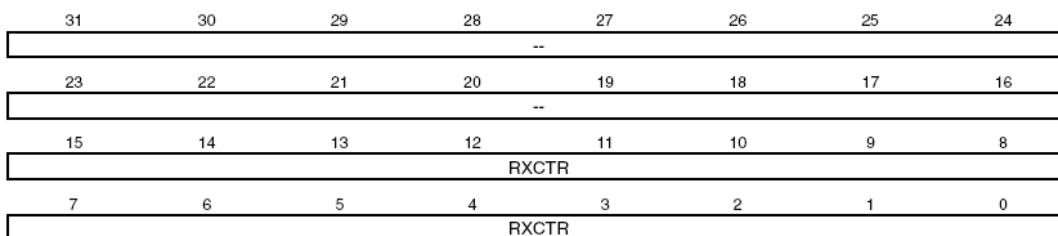
USART0 PDC Receive Counter Register

We place the number of expected receive characters in the Receive Counter Register. The DMA controller will decrement this count each time a character arrives and transfer the character to the buffer. When the count hits zero, we get a ENDRX interrupt.

22.4.2 PDC Receive Counter Register

Register Name: PERIPH_RCR

Access Type: Read/Write



- RXCTR: Receive Counter Value

Number of receive transfers to be performed.

Since we want to read in 10 characters, we set the DMA receive count to ten.

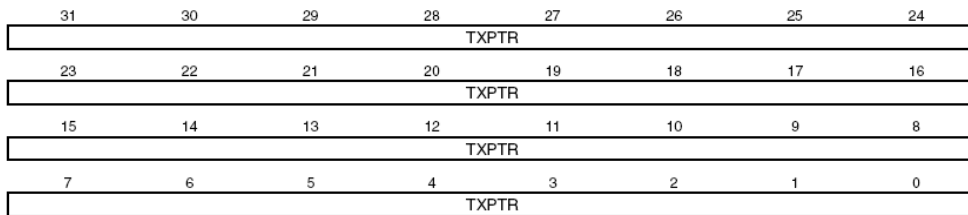
```
pUsart0->US_RCR = 10; // we'll read in 10 chars via DMA
```

USART0 PDC Transmit Pointer Register

The Transmit Pointer Register is where you load the address of the buffer that contains the outgoing characters.

22.4.3 PDC Transmit Pointer Register

Register Name: PERIPH_TPR
Access Type: Read/Write



- TXPTR: Transmit Pointer Address
Address of the transmit buffer.

The 32-byte buffer is defined in the file "usart0_isr.c" and serves both the transmit and the receive operation. Here we load the Transmit Pointer Register with the address of the buffer.

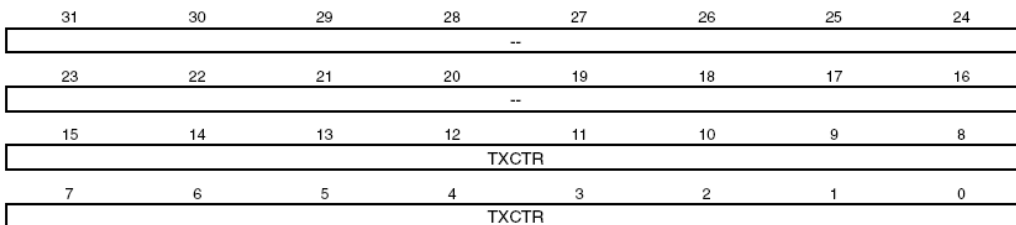
```
pUsart0->US_TPR = (unsigned int)Buffer; // address of DMA output buffer (use same one)
```

USART0 PDC Transmit Counter Register

We place the number of expected transmit characters in the Transmit Counter Register. The DMA controller will decrement this count each time a character is sent. When the count hits zero, we get a ENDTX interrupt.

22.4.4 PDC Transmit Counter Register

Register Name: PERIPH_TCR
Access Type: Read/Write



- TXCTR: Transmit Counter Value
TXCTR is the size of the transmit transfer to be performed. At zero, the peripheral DMA transfer is stopped.

Since we want to transmit 10 characters, we set the DMA transmit count to ten.

```
pUsart0->US_TCR = 10;           // we'll transmit 10 chars via DMA
```

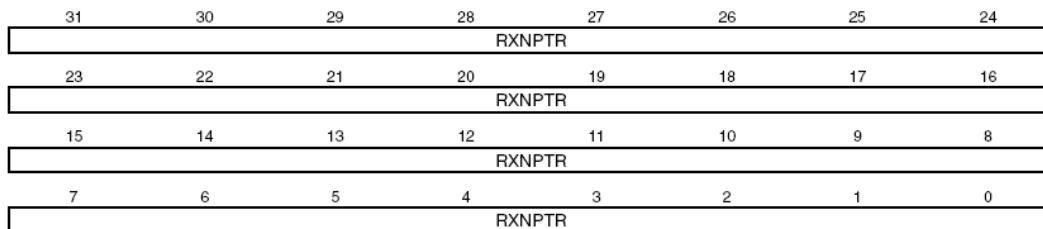
USART0 PDC Receive Next Pointer Register

The Receive Next Pointer Register is where you load the address of the secondary buffer that is to receive the incoming characters after the primary buffer fills up. When the primary buffer fills up, this pointer is copied into the USART_RPR and reception continues.

22.4.5 PDC Receive Next Pointer Register

Register Name: PERIPH_RNPR

Access Type: Read/Write



- RXNPTR: Receive Next Pointer Address

RXNPTR is the address of the next buffer to fill with received data when the current buffer is full.

Since we're not planning to use this buffer chaining feature in this example, we just set the pointer to zero.

```
pUsart0->US_RNPR = (unsigned int)0; // next DMA receive buffer address
// if set to zero, it is not used
```

USART0 PDC Receive Next Counter Register

We place the number of expected receive characters destined for the alternate buffer in the Receive Next Counter Register. When the primary buffer fills up, this count is copied into the USART0_RCD register. The DMA controller will decrement this count each time a character arrives and transfer the character to the alternate buffer. When the count hits zero, we get a RXBUFF interrupt.

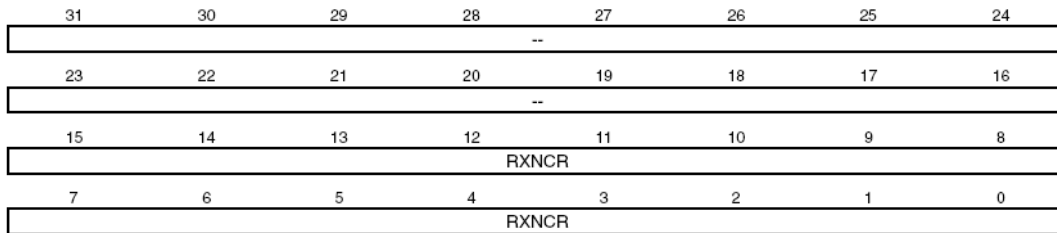
The Atmel data sheet states that “if the Next Counter Register is equal to zero, the PDC disables the trigger while activating the related peripheral end flag”. This means that setting the “next receive count register” to zero disables the “chained” buffers feature.

This is a good place to point out that the ENDRX and ENDTX interrupts signify that the current buffer has either been filled (receive) or emptied (transmit). Also, the RXBUFF and TXBUFE interrupts signify that both the primary and secondary buffer has either been filled (receive) or emptied (transmit). We are not using the secondary buffer feature in the example project, but it is fairly straightforward to set up.

22.4.6 PDC Receive Next Counter Register

Register Name: PERIPH_RNCR

Access Type: Read/Write



- **RXNCR: Receive Next Counter Value**

RXNCR is the size of the next buffer to receive.

Below we set the Receive Next Counter register to zero to disable the chaining feature.

```
pUsart0->US_RNCR = (unsigned int)0;           // next DMA receive counter
                                                // if set to zero, it is not used
```

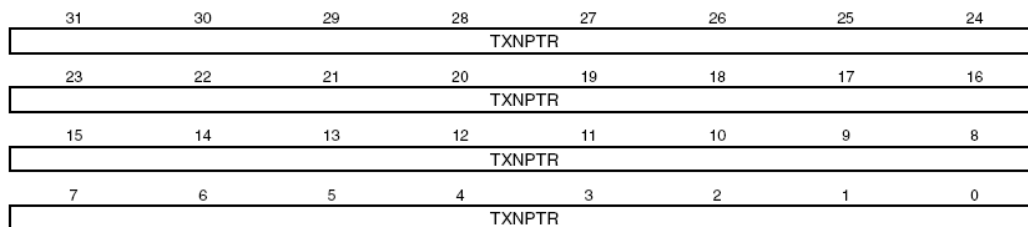
USART0 PDC Transmit Next Pointer Register

The Transmit Next Pointer Register is where you load the address of the secondary buffer that be transmitted after the primary buffer has emptied.

22.4.7 PDC Transmit Next Pointer Register

Register Name: PERIPH_TNPR

Access Type: Read/Write



- **TXNPTR: Transmit Next Pointer Address**

TXNPTR is the address of the next buffer to transmit when the current buffer is empty.

Since we're not planning to use this chained buffer feature in the example, we just set the pointer to zero.

```
pUsart0->US_TNPR = (unsigned int)0;           // next DMA transmit buffer address
                                                // if set to zero, it is not used
```

USART0 PDC Transmit Next Counter Register

The Transmit Next Counter Register is where you specify the number of characters to be transmitted using the secondary buffer.

22.4.8 PDC Transmit Next Counter Register

Register Name: PERIPH_TNCR

Access Type: Read/Write

31	30	29	28	27	26	25	24
--							
23	22	21	20	19	18	17	16
--							
15	14	13	12	11	10	9	8
TXNCR							
7	6	5	4	3	2	1	0
TXNCR							

- **TXNCR: Transmit Next Counter Value**

TXNCR is the size of the next buffer to transmit.

Since we are not using the chained buffer feature in this example, we set the value to zero to disable the “chained buffer” feature.

```
pUsart0->US_TNCR = (unsigned int)0; // next DMA transmit counter
// if set to zero, it is not used
```

USART0 PDC Transfer Control Register

The Transfer Control Register allows us to enable and disable receiver and transmitter DMA operations

22.4.9 PDC Transfer Control Register

Register Name: PERIPH_PTCR

Access Type: Write-only

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	TXTDIS	TXTEN
7	6	5	4	3	2	1	0
-	-	-	-	-	-	RXTDIS	RXTEN

- **RXTEN: Receiver Transfer Enable**

0 = No effect.

1 = Enables the receiver PDC transfer requests if RXTDIS is not set.

- **RXTDIS: Receiver Transfer Disable**

0 = No effect.

1 = Disables the receiver PDC transfer requests.

- **TXTEN: Transmitter Transfer Enable**

0 = No effect.

1 = Enables the transmitter PDC transfer requests.

- **TXTDIS: Transmitter Transfer Disable**

0 = No effect.

1 = Disables the transmitter PDC transfer requests

Since we intend to receive 10 characters first, we enable the receive transfer and disable the transmit transfer.

```
pUsart0->US_PTCR = AT91C_PDC_RXTEN |           // enable receive transfer,  
                  AT91C_PDC_TXTDIS;         // disable transmit transfer
```

Set Up for DMA Interrupts

For the normal interrupt-driven example given previously, we used the RXRDY and TXEMPTY interrupts. For the DMA example, we will use the ENDRX and ENDTX interrupts instead. Thus, the code to set up the interrupts is exactly the same excepting for the ENDRX and ENDTX interrupts.

```
// Set up the Advanced Interrupt Controller (AIC) registers for USART0  
volatile AT91PS_AIC    pAIC = AT91C_BASE_AIC; // pointer to AIC data structure  
  
pAIC->AIC_IDCR = (1<<AT91C_ID_US0);          // Disable USART0 interrupt in AIC  
  
pAIC->AIC_SVR[AT91C_ID_US0] =                // Set the USART0 IRQ handler address in AIC Source  
    (unsigned int)Usart0IrqHandler;          // Vector Register[6]  
  
pAIC->AIC_SMR[AT91C_ID_US0] =                // Set the interrupt source type and priority  
    (AT91C_AIC_SRCTYPE_INT_HIGH_LEVEL | 0x4 ); // in AIC Source Mode Register[6]  
    pAIC->AIC_IECR = (1<<AT91C_ID_US0);      // Enable the USART0 interrupt in AIC  
  
// enable the USART0 receiver and transmitter  
pUsart0->US_CR = AT91C_US_RXEN | AT91C_US_TXEN;  
  
// enable the USART0 end-of-receive interrupt  
pUsart0->US_IER = AT91C_US_ENDRX;           // enable ENDRX usart0 end-of-receive interrupt  
pUsart0->US_IDR = ~AT91C_US_ENDRX;         // disable all interrupts except ENDRX
```

DMA Interrupt Handler

To process characters using the DMA, we only have two USART0 IRQ interrupts to deal with; the ENDRX interrupt that occurs when the 10th character has arrived and the ENDTX interrupt that occurs after the 10th character has been transmitted.

The DMA interrupt service routine is conceptually simpler since when either interrupt has occurred, all characters have been either received or transmitted. All that has to be done is reset the buffer pointers and character count and enable for either transmission or reception.

When the ENDRX “end-of-message” interrupt asserts, we do the following steps to turn things around for transmission.

```
// we have a end-of-receive interrupt (ENDRX)
pUsart0->US_RCR = 10;           // restore the receive count - clears ENDRX flag

// point the transmit buffer pointer to beginning of buffer, set count to 10
pUsart0->US_TPR = (AT91C_REG)&Buffer[0]; // address of DMA output buffer (use same one)
pUsart0->US_TCR = 10;           // we'll transmit 10 chars via DMA

// disable the end-of-receive interrupt, enable the end-of-transmit interrupt
pUsart0->US_IER = AT91C_US_ENDTX; // enable usart0 end-of-transmit interrupt
pUsart0->US_IDR = ~AT91C_US_ENDTX; // disable all interrupts except ENDTX

// enable transmit DMA transfers, disable receive DMA transfers
// note: this will START the transmission of whatever is in the Buffer[32]!
pUsart0->US_PTCR = AT91C_PDC_TXTEN | // enable transmit transfer,
                  AT91C_PDC_RXTDIS; // disable receive transfer
```

Likewise, when the ENDTX “end-of-message” interrupt asserts, we do the following steps to turn things around for reception (so the application runs over and over).

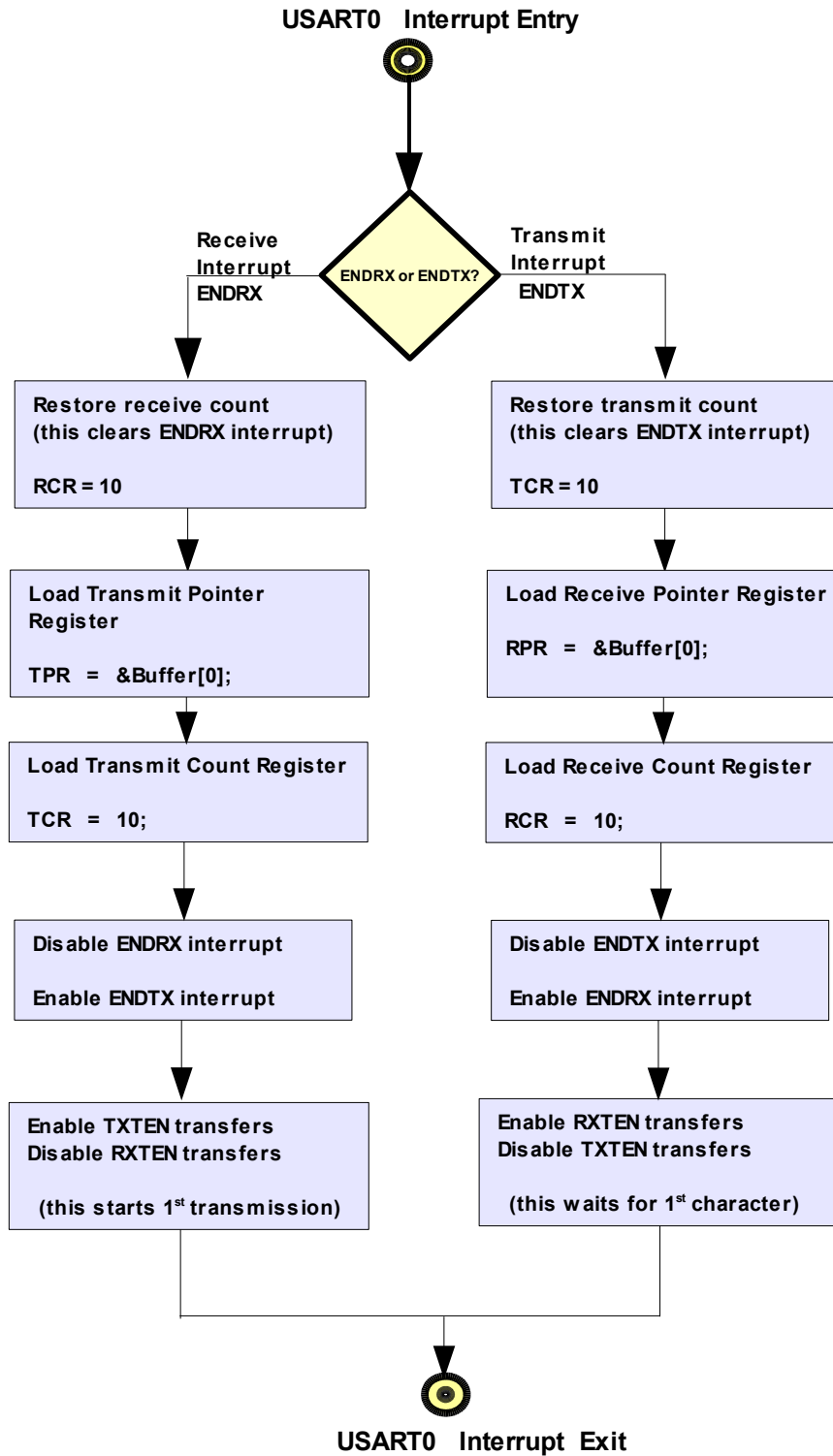
```
// we have a end-of-transmit interrupt (10 characters have clocked out)
pUsart0->US_TCR = 10;           // restore the transmit count - clears ENDTX flag

// point the receive buffer pointer to beginning of buffer, set count to 10
pUsart0->US_RPR = (AT91C_REG)&Buffer[0]; // address of DMA output buffer (use same one)
pUsart0->US_RCR = 10;           // we'll receive 10 chars via DMA

// enable receive interrupt, disable the transmit interrupt
pUsart0->US_IER = AT91C_US_ENDRX; // enable usart0 end-of-receive interrupt
pUsart0->US_IDR = ~AT91C_US_ENDRX; // disable all interrupts except ENDRX

// enable receive DMA transfers, disable transmit DMA transfers
// note: the DMA transfer will start when the first character arrives!
pUsart0->US_PTCR = AT91C_PDC_RXTEN | // enable receive transfer,
                  AT91C_PDC_TXTDIS; // disable transmit transfer
```

A simplified flow chart of the process is given below.



Project Listings – DMA Version

Only two source files are different for the DMA version of the project: “`usart0_isr.c`” and “`usart0_setup.c`”. All other files are unchanged but are included in this tutorial's attachment.

USART0_SETUP.C

All we do in the DMA version of “`usart0_setup.c`” is to initialize the DMA registers and set up for ENDRX and ENDTX interrupts.

```
// *****
//
//          usart0_setup.c
//
// Purpose: Set up USART0 (peripheral ID = 6) 9600 baud, 8 data bits, 1 stop bit, no parity
//          This example uses DMA control of the USART0
//
//
// We will use the onboard baud rate generator to specify 9600 baud
//
// The Olimex SAM7-EX256 board has a 18,432,000 hz crystal oscillator.
//
// MAINCK = 18432000 hz (from Olimex schematic)
// DIV = 14 (set up in lowlevelinit.c)
// MUL = 72 (set up in lowlevelinit.c)
//
// PLLCK = (MAINCK / DIV) * (MUL + 1) = 18432000/14 * (72 + 1)
// PLLCLK = 1316571 * 73 = 96109683 hz
// MCK = PLLCLK / 2 = 96109683 / 2 = 48054841 hz
//
// Baud Rate (asynchronous mode) = MCK / (8(2 - OVER)CD)
//
// MCK = 48054841 hz (set USCLKS = 00 in USART Mode Register US_MR - to select MCK only)
// OVER = 0 (bit 19 of the USART Mode Register US_MR)
// CD = divisor (USART Baud Rate Generator Register US_BRGR)
// baudrate = 9600 (desired)
//
//
// a little algebra: BaudRate =  $\frac{48054841}{(8(2 - 0)CD)}$  =  $\frac{48054841}{16(CD)}$ 
//
//
//  $CD = \frac{48054841}{9600(16)}$  =  $\frac{48054841}{153600}$  = 312.857037
//
// CD = 313 (round up)
//
//
// check the actual baud rate: BaudRate =  $\frac{48054841}{(8(2 - 0)313)}$  =  $\frac{48054841}{5008}$  = 9595.6
//
//
// what's the error: Error = 1 -  $\frac{\text{desired baudrate}}{\text{actual baudrate}}$  = 1 -  $\frac{9600}{9595.6}$  = 1 - 1.0004585 = -.0004585
// (not much)
//
//
// Author: James P Lynch June 22, 2008
// *****
//
// *****
//          Header Files
// *****
#include "at91sam7x256.h"
#include "board.h"
```

```

// *****
//                               External Globals
// *****
extern char Buffer[];              // holds received characters
extern unsigned long nChars;     // counts number of received chars
extern char *pBuffer;            // pointer into Buffer

// *****
//                               Function Prototypes
// *****
void Usart0IrqHandler(void);

void USART0Setup(void) {
    // enable the usart0 peripheral clock
    volatile AT91PS_PMC pPMC = AT91C_BASE_PMC; // pointer to PMC data structure
    pPMC->PMC_PCER = (1<<AT91C_ID_US0); // enable usart0 peripheral clock

    // set up PIO to enable USART0 peripheral control of pins
    volatile AT91PS_PIO pPIO = AT91C_BASE_PIOA; // pointer to PIO data structure
    pPIO->PIO_PDR = AT91C_PA0_RXD0 | AT91C_PA1_TXD0; // enable peripheral control of PA0 and PA1
    pPIO->PIO_ASR = AT91C_PIO_PA0 | AT91C_PIO_PA1; // assigns the 2 I/O lines to peripheral A function
    pPIO->PIO_BSR = 0; // peripheral B function set to "no effect"

    // set up the USART0 registers
    volatile AT91PS_USART pUsart0 = AT91C_BASE_US0; // create a pointer to USART0 structure

    //
    //                               USART0 Control Register US_CR (read/write)
    //
    //
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // 31                                           24
    //
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // -----|-----|-----|-----|-----|-----|-----|-----|
    //             RTSDIS  RTSSEN  DTRDIS  DTREN
    // 23     22     21     20     19     18     17     16
    //
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // RETTO  RSTNACK  RSTIT  SENDA  STTTO  STPBRK  STTBRK  RSTSTA
    // 15     14     13     12     11     10     9     8
    //
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // -----|-----|-----|-----|-----|-----|-----|-----|
    // TXDIS  TXEN  RXDIS  RXEN  RSTTX  RSTRX  -      -
    // 7      6      5      4      3      2      1      0
    //
    // RSTRX = 1             (reset receiver)
    // RSTTX = 1             (reset transmitter)
    // RXEN = 0              (receiver enable - no effect)
    // RXDIS = 1            (receiver disable - disabled)
    // TXEN = 0              (transmitter enable - no effect)
    // TXDIS = 1            (transmitter disable - disabled)
    // RSTSTA = 0           (reset status bits - no effect)
    // STTBRK = 0           (start break - no effect)
    // STPBRK = 0           (stop break - no effect)
    // STTTO = 0            (start time-out - no effect)
    // SENDA = 0            (send address - no effect)
    // RSTIT = 0            (reer iterations - no effect)
    // RSTNACK = 0          (reset non acknowledge - no effect)
    // RETTO = 0            (rearm time-out - no effect)
    // DTREN = 0            (data terminal ready enable - no effect)
    // DTRDIS = 0           (data terminal ready disable - no effect)
    // RTSSEN = 0           (request to send enable - no effect)
    // RSTDIS = 0           (request to send disable - no effect)

    pUsart0->US_CR = AT91C_US_RSTRX | // reset receiver
                    AT91C_US_RSTTX | // reset transmitter
                    AT91C_US_RXDIS | // disable receiver
                    AT91C_US_TXDIS; // disable transmitter

```



```

//          USART0 Mode Register US_MR      (read/write)
//
//          |-----|-----|-----|-----|-----|-----|
//          |-----|-----|-----|-----|-----|-----|
//          |          FILTER          |          MAX_ITERATION          |
//          |          31          |          29          |          28          |          27          |          26          |          24          |
//          |-----|-----|-----|-----|-----|-----|
//          |          DSNACK          |          INACK          |          OVER          |          CLK0          |          MODE9          |          MSBF          |
//          |          23          |          22          |          21          |          20          |          19          |          18          |          17          |          16          |
//          |-----|-----|-----|-----|-----|-----|
//          |          CHMODE          |          NBSTOP          |          PAR          |          SYNC          |
//          |          15          |          14          |          13          |          12          |          11          |          9          |          8          |
//          |-----|-----|-----|-----|-----|-----|
//          |          CHRL          |          USCLKS          |          USART_MODE          |
//          |          7          |          6          |          5          |          4          |          3          |          0          |
//
//          US_MR = 0x0000          normal
//          USCLKS = 0x00          choose MCK for baud rate generator
//          CHRL = 0x11          8-bit characters
//          SYNC = 0x00          asynchronous mode
//          PAR = 0x100          no parity
//          NBSTOP = 0x00          1 stop bit
//          CHMODE = 0x00          normal mode, no loop-back, etc.
//          MSBF = 0x00          LSB sent/received first
//          MODE9 = 0x00          CHRL defines character length
//          CLK0 = 0x00          USART does not drive SCK pin
//          OVER = 0x00          16 x oversampling (see baud rate equation)
//          INACK = 0x00          NACK (not used)
//          DSNACK = 0x00          not used since NACK is not generated
//          MAX_ITERATION = 0x00          max iterations not used
//          FILTER = 0x00          filter is off

```

```

pUsart0->US_MR = AT91C_US_PAR_NONE |          // no parity
                0x3 << 6;                   // 8-bit characters

```

```

//          USART0 Interrupt Enable Register US_IER      (write only)
//
//          |-----|-----|-----|-----|-----|-----|
//          |          31          |          30          |          29          |          28          |          27          |          26          |          25          |          24          |
//          |-----|-----|-----|-----|-----|-----|
//          |          CTSIC          |          DCDIC          |          DSRIC          |          RIIC          |
//          |          23          |          22          |          21          |          20          |          19          |          18          |          17          |          16          |
//          |-----|-----|-----|-----|-----|-----|
//          |          NACK          |          RXBUFF          |          TXBUFE          |          ITERATIO          |          TXEMPTY          |          TIMEOUT          |
//          |          15          |          14          |          13          |          12          |          11          |          10          |          9          |          8          |
//          |-----|-----|-----|-----|-----|-----|
//          |          PARE          |          FRAME          |          OVRE          |          ENDTX          |          ENDRX          |          RXBRK          |          TXRDY          |          RXRDY          |
//          |          7          |          6          |          5          |          4          |          3          |          2          |          1          |          0          |
//
//          US_IER = 0x00;          // no usart0 interrupts enabled (no effect)

```

```

//          USART0 Interrupt Disable Register US_IDR      (write only)
//
//          |-----|-----|-----|-----|-----|-----|
//          |          31          |          30          |          29          |          28          |          27          |          26          |          25          |          24          |
//          |-----|-----|-----|-----|-----|-----|
//          |          CTSIC          |          DCDIC          |          DSRIC          |          RIIC          |
//          |          23          |          22          |          21          |          20          |          19          |          18          |          17          |          16          |
//

```



```

//      7                                0
//      this is where we place characters to be transmitted

//      USART0 Baud Rate Generator Register US_BRGR   (read/write)
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      31                                24
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      23                                19 18                                16
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      15                                8
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      7                                Y                                0
//
pUsart0->US_BRGR = 0x139;          // CD = 0x139 (313 from above calculation for 9600 baud)
//                                // FP=0 (not used)

```

```

//      USART0 Receiver Time-out Register US_RTOR   (read/write)
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      31      30      29      28      27      26      25      24
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      23      22      21      20      19      18      17      16
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      15                                T0                                9
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      7                                T0                                0
//
pUsart0->US_RTOR = 0;          // receiver time-out (disabled)

```

```

//      USART0 transmitter TimeGuard Register US_TTGR   (read/write)
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      31      30      29      28      27      26      25      24
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      23      22      21      20      19      18      17      16
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      15      14      13      12      11      10      9      8
//
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      |-----|-----|-----|-----|-----|-----|-----|-----|
//      7                                TG                                0
//
pUsart0->US_TTGR = 0;          // transmitter timeguard (disabled)

```

```

//          USART0 FI DI RatioRegister    US_FIDI    (read/write)
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 31       30       29       28       27       26       25       24
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 23       22       21       20       19       18       17       16
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
//                                FI_DI_RATIO
// 15       14       13       12       11       10       9       8
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
//                                FI_DI_RATIO
// 7                                               0
// not used, nothing to set up here

```

```

//          USART0 Number of Errors Register    US_NER    (read only)
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 31                                               24
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 23                                               16
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 15                                               8
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
//                                NB_ERRORS
// 7                                               0
// Read-only, nothing to set up here

```

```

//          USART0 IrDA Filter Register    US_IF    (read/write)
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 31                                               24
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 23                                               16
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
// 15                                               8
//
// |-----|-----|-----|-----|-----|-----|-----|-----|
// |-----|-----|-----|-----|-----|-----|-----|-----|
//                                IRDA_FILTER
// 7                                               0
// not used, nothing to set up here

```


USART0_ISR.C

The DMA USART0 interrupt service routine is less complicated than the normal interrupt-driven version. Interrupts occur at the end of a 10 character reception or transmission.

```
// *****
//                               usart0_isr.c
//
//   USART0 Interrupt Service Routine - DMA version
//
//   This demonstration is designed to read 10 characters into a buffer.
//   After the 10th character arrives, transmit the 10 characters back.
//
//   The application is interrupt-driven but uses the DMA.
//
//   Author: James P Lynch June 22, 2008
//   *****

// *****
//                               Header Files
// *****
#include "at91sam7x256.h"
#include "board.h"

// *****
//                               Global Variables
// *****
char          Buffer[32];           // holds received characters
unsigned long nChars = 0;         // counts number of received chars
char          *pBuffer = &Buffer[0]; // pointer into Buffer

void Usart0IrqHandler (void) {

    volatile AT91PS_USART pUsart0 = AT91C_BASE_US0; // create a pointer to USART0 structure

    // determine which interrupt has occurred (end-of-receive DMA or end-of-transmit DMA)
    if ((pUsart0->US_CSR & AT91C_US_ENDRX) == AT91C_US_ENDRX) {

        // we have a end-of-receive interrupt (ENDRX)
        pUsart0->US_RCR = 10; // restore the receive count - clears ENDRX flag

        // point the transmit buffer pointer to beginning of buffer, set count to 10
        pUsart0->US_TPR = (AT91_REG)&Buffer[0]; // address of DMA output buffer (use same one)
        pUsart0->US_TCR = 10; // we'll transmit 10 chars via DMA

        // disable the end-of-receive interrupt, enable the end-of-transmit interrupt
        pUsart0->US_IER = AT91C_US_ENDTX; // enable usart0 end-of-transmit interrupt
        pUsart0->US_IDR = ~AT91C_US_ENDTX; // disable all interrupts except ENDTX

        // enable transmit DMA transfers, disable receive DMA transfers
        // note: this will START the transmission of whatever is in the Buffer[32]!
        pUsart0->US_PTCR = AT91C_PDC_TXTEN | // enable transmit transfer,
            AT91C_PDC_RXTDIS; // disable receive transfer

    } else if ((pUsart0->US_CSR & AT91C_US_ENDTX) == AT91C_US_ENDTX) {

        // we have a end-of-transmit interrupt (10 characters have clocked out)
        pUsart0->US_TCR = 10; // restore the transmit count - clears ENDTX flag

        // point the receive buffer pointer to beginning of buffer, set count to 10
        pUsart0->US_RPR = (AT91_REG)&Buffer[0]; // address of DMA output buffer (use same one)
        pUsart0->US_RCR = 10; // we'll receive 10 chars via DMA
    }
}
```

```

// enable receive interrupt, disable the transmit interrupt
pUsart0->US_IER = AT91C_US_ENDRX;           // enable usart0 end-of-receive interrupt
pUsart0->US_IDR = ~AT91C_US_ENDRX;         // disable all interrupts except ENDRX

// enable receive DMA transfers, disable transmit DMA transfers
// note: the DMA transfer will start when the first character arrives!
pUsart0->US_PTCR = AT91C_PDC_RXTEN |       // enable receive transfer,
                  AT91C_PDC_TXTDIS;       // disable transmit transfer
}
}

```

Building the DMA Application

Create a new Eclipse C project and give it the name “**demo_sam7ex256_dma**”. Import the source files from the attachment to this tutorial and build the project. The screen capture below shows the build operation.

```

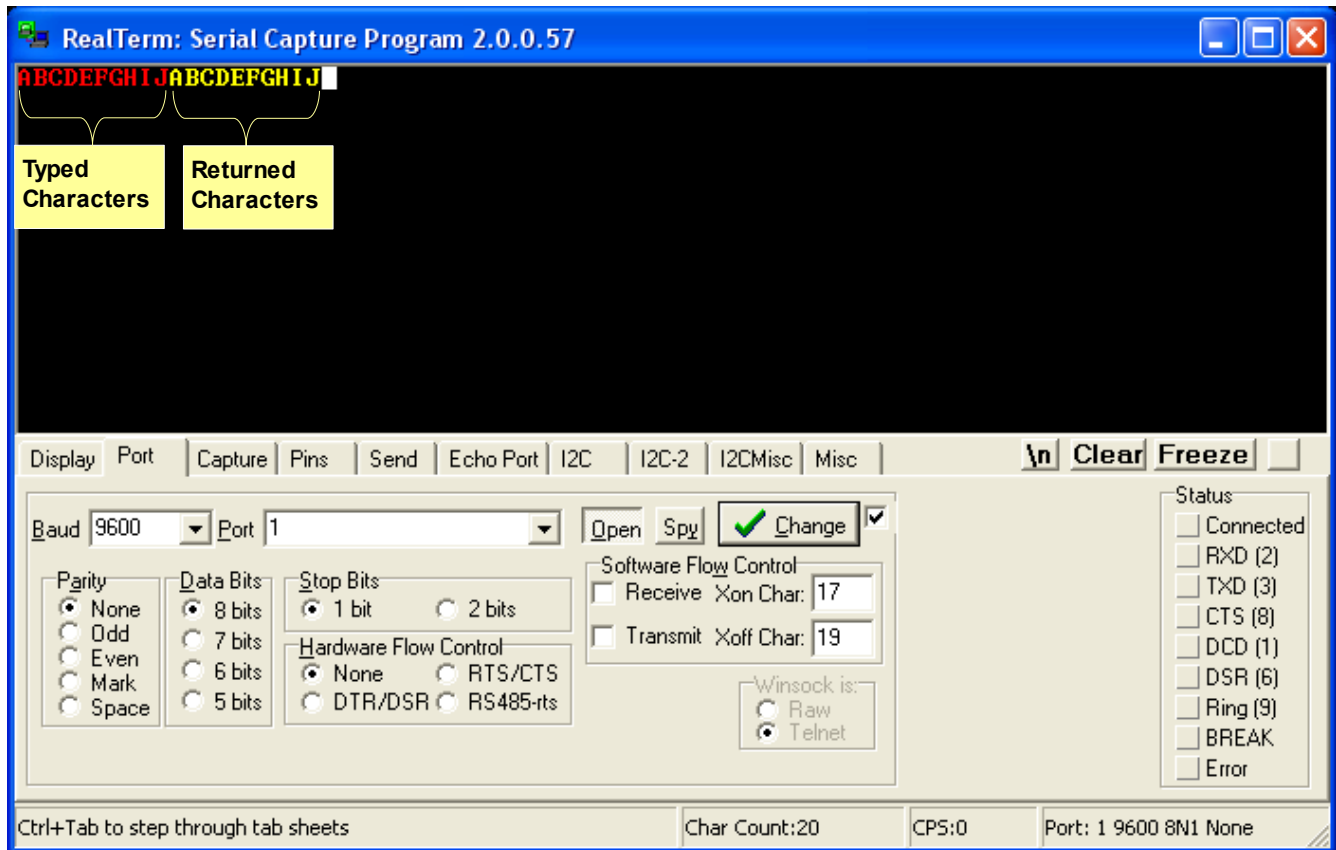
C/C++ - demo_sam7ex256_dma/usart0_isr.c - Eclipse Platform
File Edit Refactor Navigate Search Run Project Window Help
C-Build [demo_sam7ex256_dma]
**** Build of configuration Default for project demo_sam7ex256_dma ****

make all
.assembling crt.s
arm-elf-as -ahls -mapcs-32 -o crt.o crt.s > crt.lst
.compiling main.c
arm-elf-gcc -I./ -c -fno-common -O0 -g main.c
.compiling lowlevelinit.c
arm-elf-gcc -I./ -c -fno-common -O0 -g lowlevelinit.c
.compiling usart0_setup.c
arm-elf-gcc -I./ -c -fno-common -O0 -g usart0_setup.c
.compiling usart0_isr.c
arm-elf-gcc -I./ -c -fno-common -O0 -g usart0_isr.c
.compiling isrsupport.c
arm-elf-gcc -I./ -c -fno-common -O0 -g isrsupport.c
..linking
arm-elf-ld -v -Map main.map -Tdemo_sam7ex256_dma.cmd -o main.out crt.o main.o lowlevelinit.o usart0_setup.o usart0_isr.o
isrsupport.o libgcc.a
GNU ld (GNU Binutils) 2.18
..copying
arm-elf-objcopy --output-target=binary main.out main.bin
arm-elf-objdump -x --syms main.out > main.dmp

```

After successfully building the project and programming it into flash memory, the application can be tested with RealTerm. You will notice that the application works just the same as the interrupt-driven version.

The screen shot below shows the operation of the application, this time in DMA mode.



Other Possibilities

Thoughtful readers should be wondering: “What if the incoming message is of undetermined length, but is terminated by a carriage-return”? This can be handled by thoughtful design.

As each character comes in, the character is written to your buffer in RAM and the DMA receive buffer pointer is incremented and the DMA receive character count is decremented. You can at any time read the DMA receive pointer register (USART0_RPR) and the DMA receive counter register (USART0_RCR) and inspect what has come in. Be aware the the pointer points to the next free byte in the buffer, so you will have to look back one byte in the buffer to see the last character entered. You could check if that byte is a carriage-return.

If you used an RTOS such as FreeRTOS, you could embed this check in the kernel tick interrupt handler and make this check every 10 msec, for example. If the last entered character is not a carriage-return, do nothing. If a carriage-return is detected, then you can disable the DMA receive operation and set a semaphore to wake up a special task to process the completed receive message.

If you are not using a RTOS, then a counter-timer could be set up to run at high speed (1 msec) and make this check in the timer IRQ handler.

For transmission, you usually know the message length, so normal DMA setup would apply.

About the Author

Jim Lynch lives in Grand Island, New York and is a software developer for Control Techniques, a subsidiary of Emerson Electric. He develops embedded software for the company's industrial drives (high power motor controllers) which are sold all over the world.



Mr. Lynch has previously worked for Mennen Medical, Calspan Corporation and the Boeing Company. He has a BSEE from Ohio University and a MSEE from State University of New York at Buffalo. Jim is a single father and has two grown children who now live in Florida and Nevada. He has two brothers, one is a Viet Nam veteran in Hollywood, Florida and the other is the Bishop of St. Petersburg, also in Florida. Jim plays the guitar (search for **lynchzilla** on YouTube), enjoys woodworking and hopes to write a book very soon that will teach students and hobbyists how to use these high-powered ARM microcontrollers.

Lynch can be reached via e-mail at: lynch007@gmail.com

Appendix

There have been changes to Eclipse and YAGARTO since I authored the *“Using Open Source Tools for Atmel AT91SAM7 Cross Development”* a year ago. In this appendix, abbreviated instructions to download and install YAGARTO and create and build the Serial Communications Eclipse project are given that reflect these changes.

Download Yagarto Components

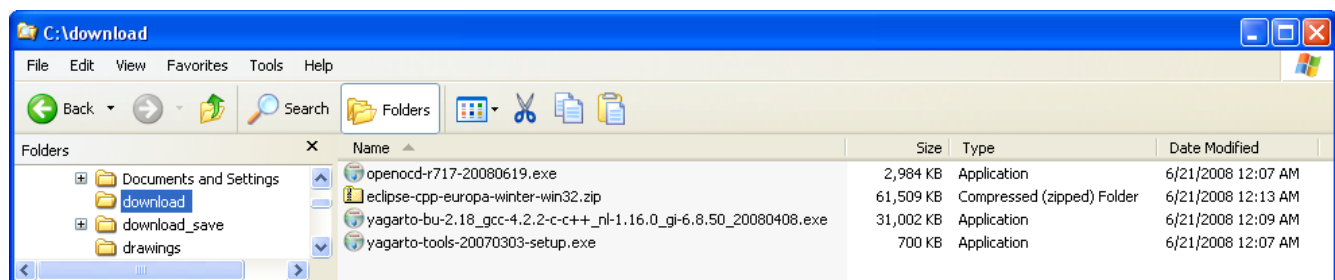
Go to the Yagarto web site (www.yagarto.de) and download the latest components; there are three install files and the Eclipse zip file to download. In the latest YAGARTO, Michael Fischer has you download the Eclipse zip file directly from the Eclipse web site. He also directs you to the SourceForge web site to get the GNU ARM Tool Chain. I suggest that you create a **“c:\download”** folder to receive these components.

Download

The packages of YAGARTO can be found here:

Package	Version	Last Version	
Open On-Chip Debugger (2.91 MB) (md5sum: 696e80452e76714439ed98bef6a2f6d) This version of OpenOCD supports the following JTAG interfaces .	r717	19.06.2008	Download OpenOCD from the YAGARTO web site. Double-click to run the installer.
YAGARTO Tools (700 KB) (md5sum: a1c654d6704bd3c1e109a73ce22eee2a) Include tools like make, sh, touch and more. You only need these tools if you do not have installed the Open On-Chip Debugger, and want to use e.g. J-Link / SAM-ICE.	20070303	03.03.2007	Download YAGARTO Tools from the YAGARTO web site. Double-click to run the installer.
YAGARTO GNU ARM toolchain (31 MB) (md5sum: 37bd97b9a45f4fd44d9f789ac4a3b6ad) Many thanks to the devkitPro project, from whom I took the source for Insight.	Binutils-2.18 Newlib-1.16.0 GCC-4.2.2 GDB-6.8.50-20080308 Insight-6.8.50-20080308	08.04.2008	Download GNU ARM Toolchain from the SourceForge web site. Double-click to run the installer.
Integrated Development Environment You must download the IDE from eclipse.org, but the link above will give you some instructions.	Eclipse Eclipse CDT Zylin plugin		Download Eclipse IDE from the Eclipse web site. Unzip into the c:\ folder

When finished, your download folder should have the following components:



Install the YAGARTO Components

Once you have the YAGARTO components downloaded, it only takes a few minutes to set it all up.

Install OpenOCD

Double-click on the file “**openocd-r717-20080619.exe**” to start the OpenOCD installer. Take the defaults on every question.

Install Eclipse IDE

Eclipse is extremely easy to install. Just unzip the file “**eclipse-cpp-europa-winter-win32.zip**” directly to the “**c:**” folder (your main drive root folder). This will create a “**c:\eclipse**” folder. Eclipse is a simple executable – it does not make any entries into the Windows registry!

Look into the Eclipse directory and send the Eclipse executable (**eclipse.exe**) to the desktop as an icon. You can click on that icon to start up Eclipse.

Install YAGARTO GNU ARM Tool Chain

Double-click on the file “**yagarto-bu-2.18_gcc-4.2.2-c-c++_nl-1.16.0_gi-6.8.50_20080408.exe**” to start the GNU ARM Tool Chain installer. Take the defaults on every question.

Install YAGARTO Tools

Do not install this if you plan to use and have installed OpenOCD (whose directory has a copy of the **make.exe** utility). If you need to install this, double-click on the file “**yagarto-tools-20070303-setup.exe**” to start the Yagarto Tools installer. Take the defaults on every question.

The very necessary utility “**make.exe**” is not part of the YAGARTO GNU ARM tool chain. There is a copy of it in the OpenOCD folder you just installed. If you didn't install OpenOCD because you plan to use the JLink, then the YAGARTO Tools installation will give you a copy of the “**make.exe**” utility.

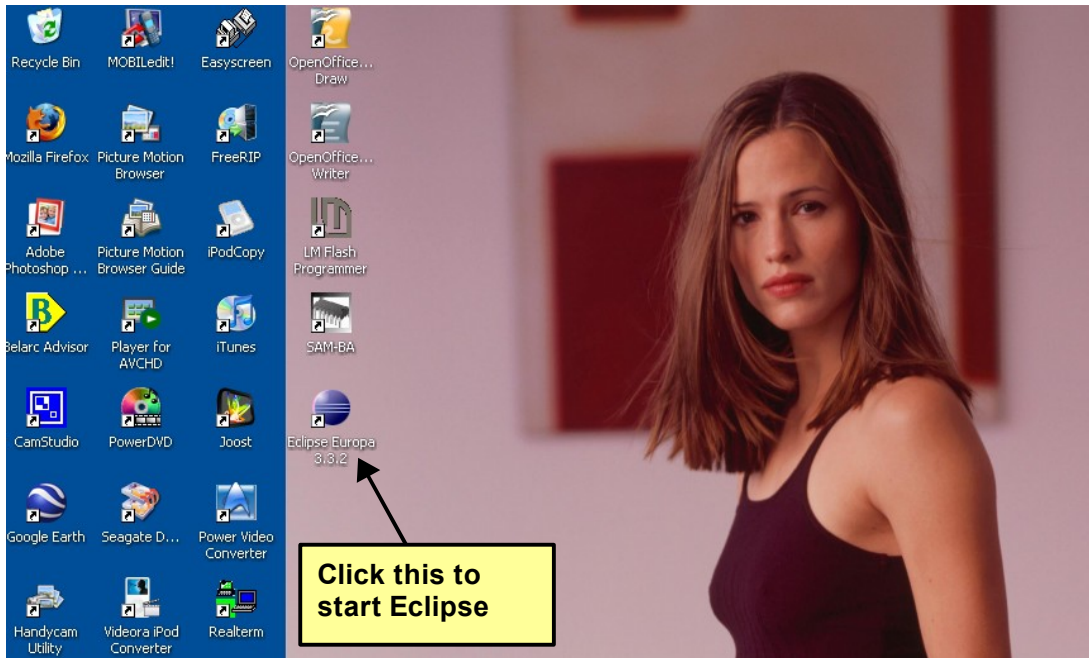
Install the JTAG Device Drivers

The very first time you plug in your JTAG debugger, you will hear the USB “beep” and will have to install the USB or parallel port drivers to support your JTAG debugging hardware. The instructions in the tutorial “*Using Open Source Tools for AT91SAM7 Cross Development*” are still valid.

You will have to direct the driver installer to the YAGARTO folder that has the drivers. For example, the Olimex ARM-USB-OCD drivers are in the folder: `c:\Program Files\openocd-r717\driver\arm-usb-ocd\`

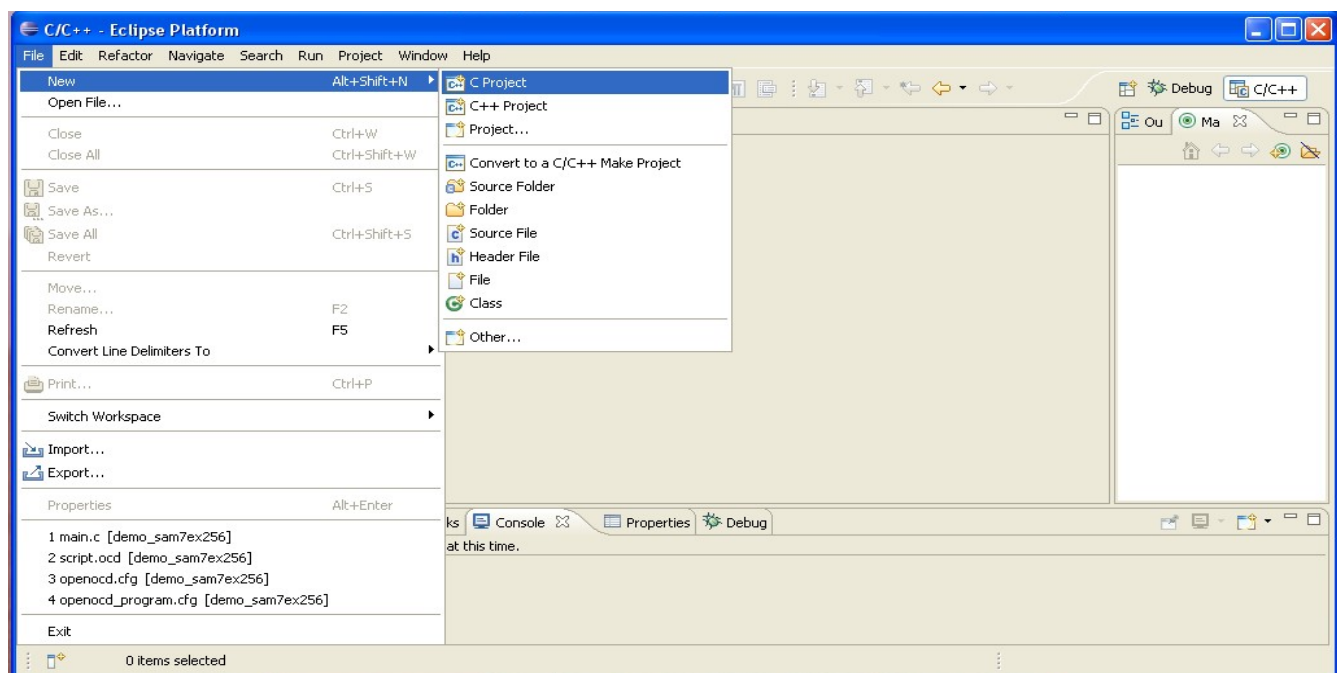
Start Up Eclipse

If you created a screen icon for Eclipse, click on it to start the Eclipse IDE.

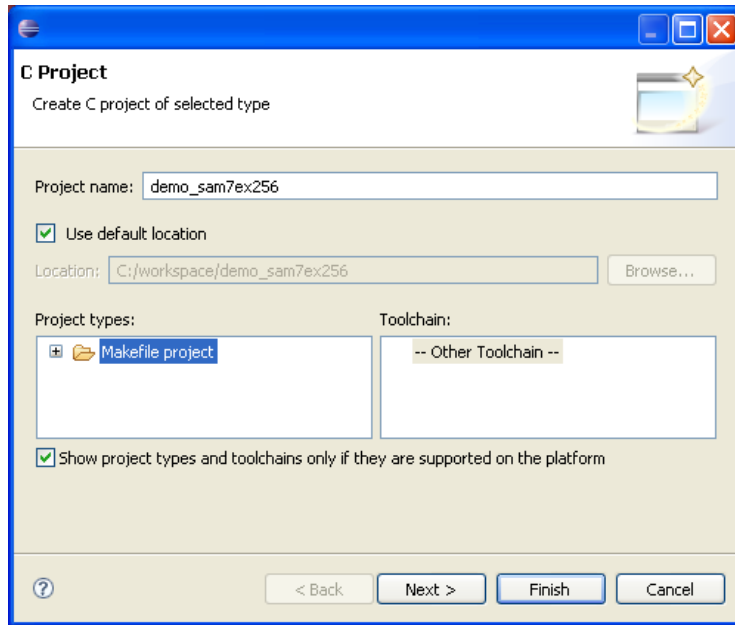


Create an Eclipse Standard C Project

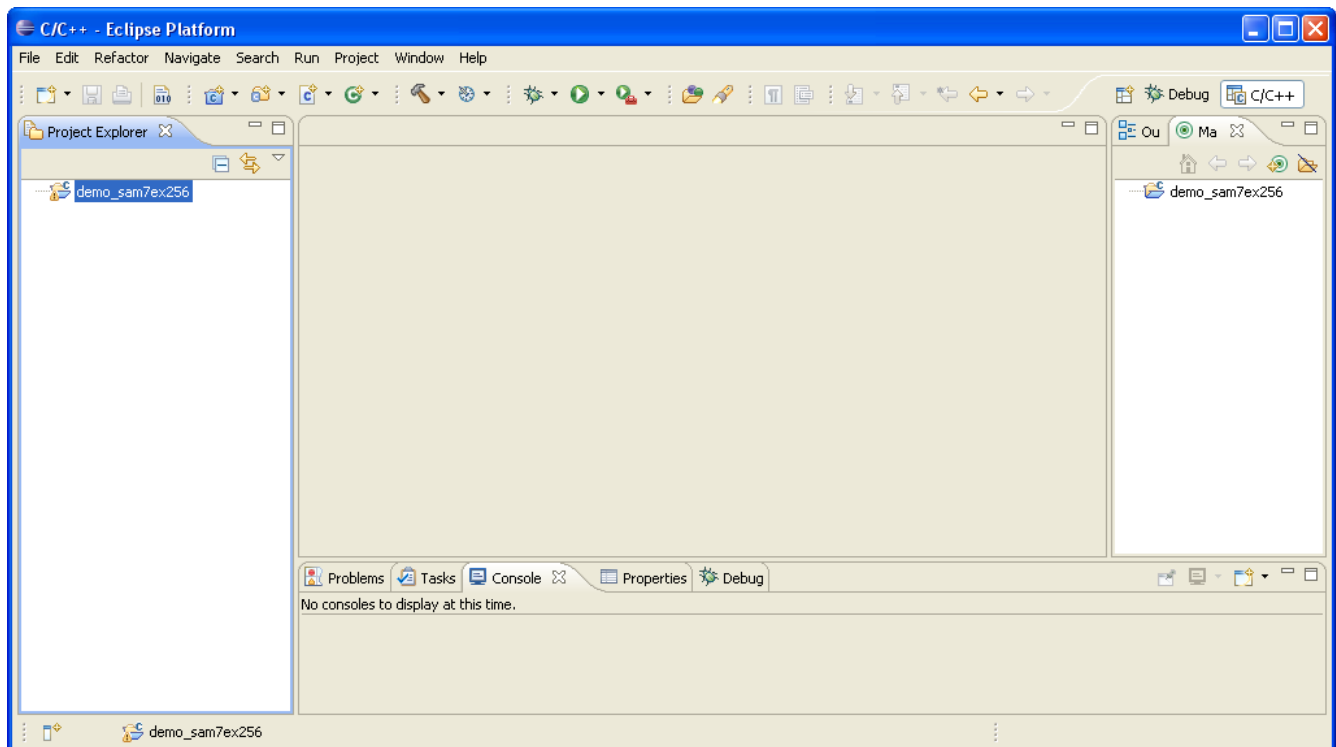
Click on “File – New – C Project”.



Type in a project name (**demo_sam7ex256**) and click on **“Makefile Project”**. Make sure that **“--Other Toolchain--”** is visible in the Toolchain: dialog box. Click **“Finish”** to exit and create the new project.

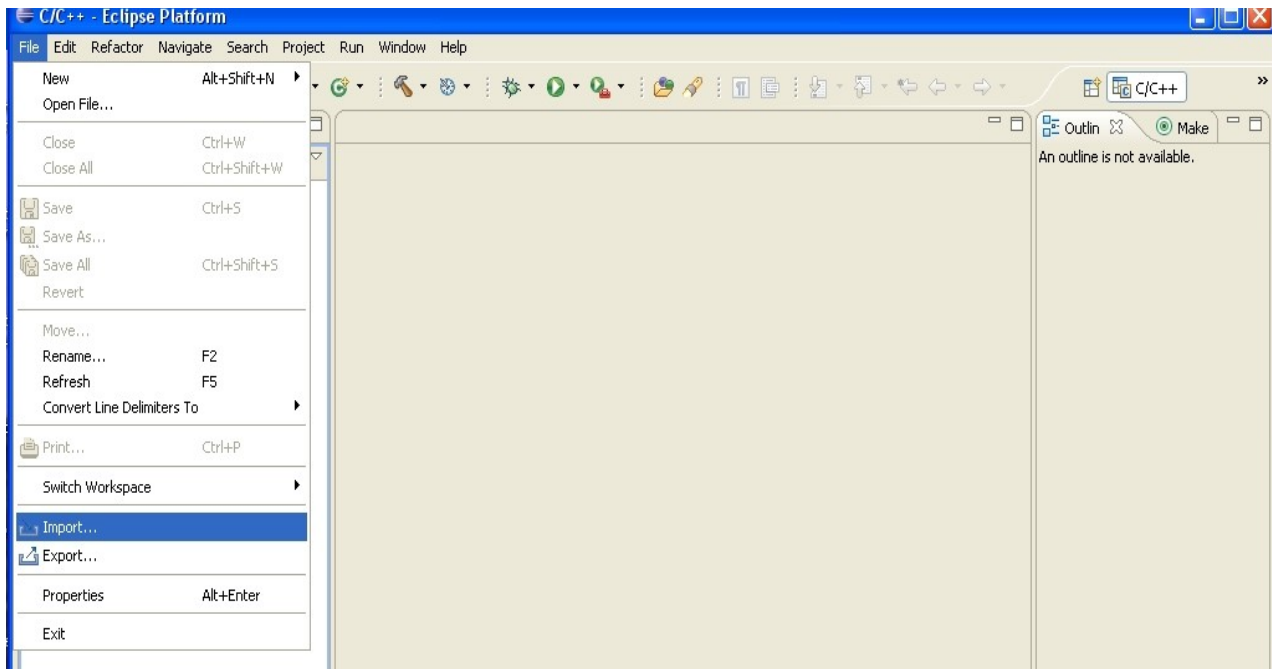


Now you have an empty project named **“demo_sam7ex256”**. There are currently no files in this project.

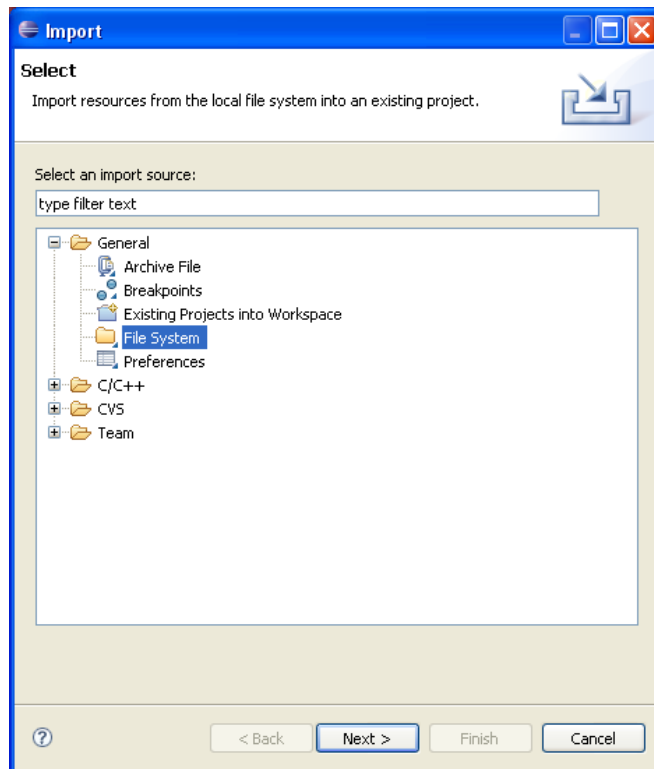


Import the Sample Project Files

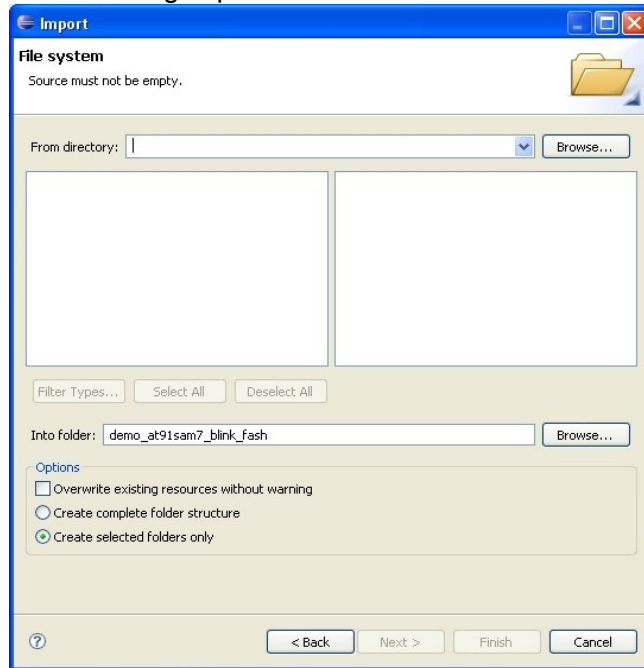
Click on “**File – Import**” to retrieve the sample files.



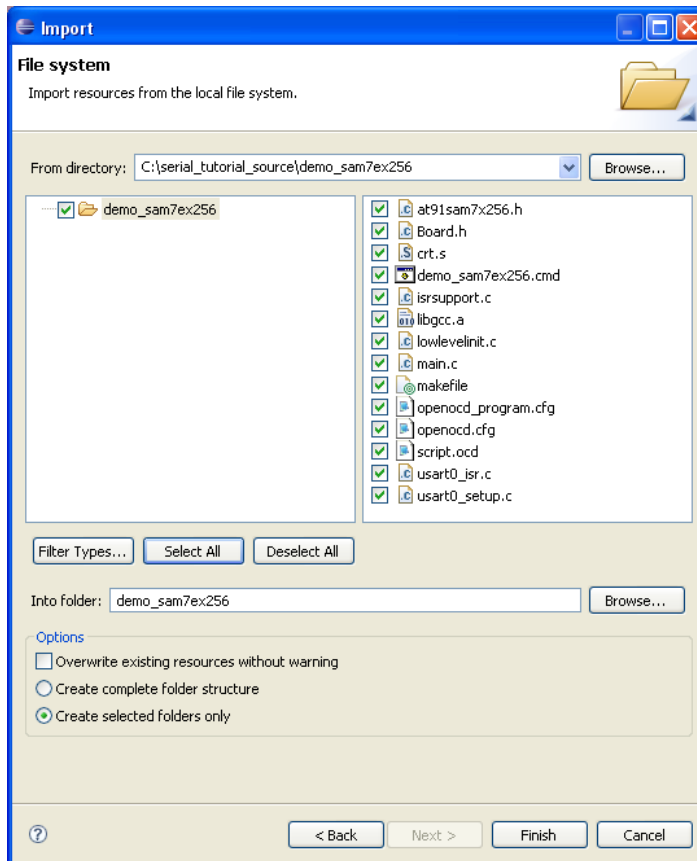
You should see this import screen. Click on “**General**” and then “**File System**” to get started. Click “**Next**” to continue with the import operation.



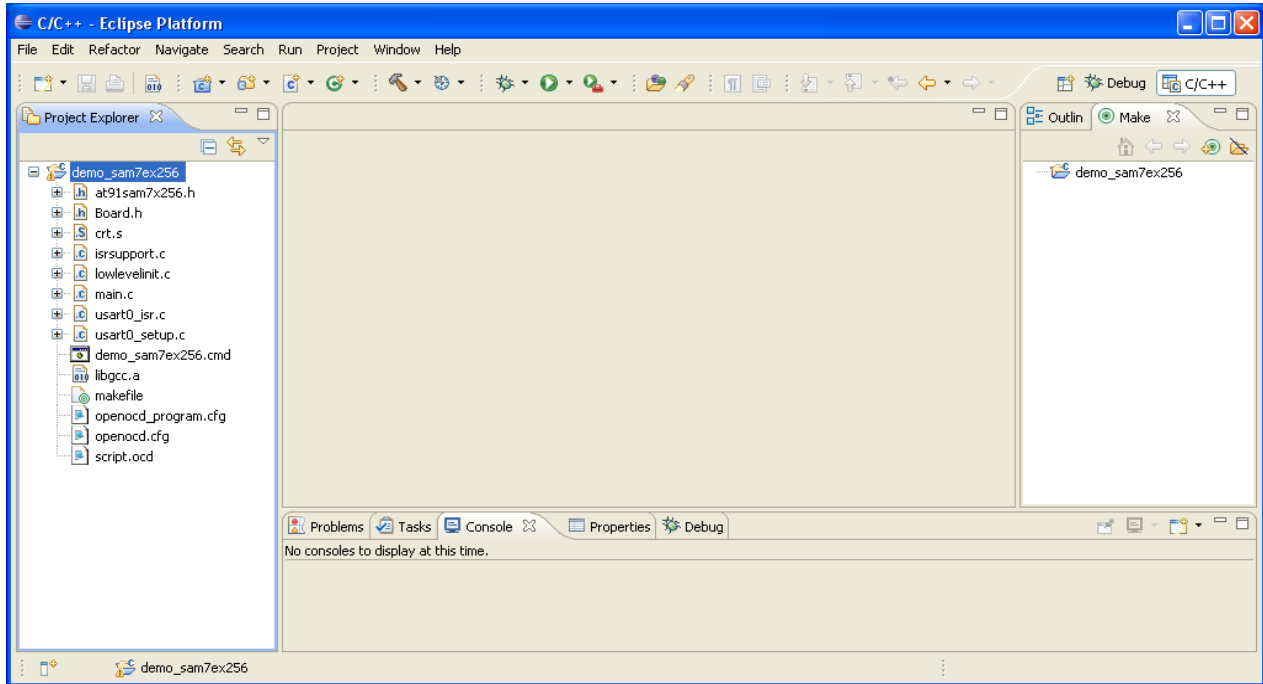
You should now see the following import screen.



Let's assume that the tutorial sample project files are in the folder "c:\serial_tutorial_source\demo_sam7ex256". Click on the "Browse" button above to search for this folder. Select all the files in this folder and click "Finish" shown below to import the project files.

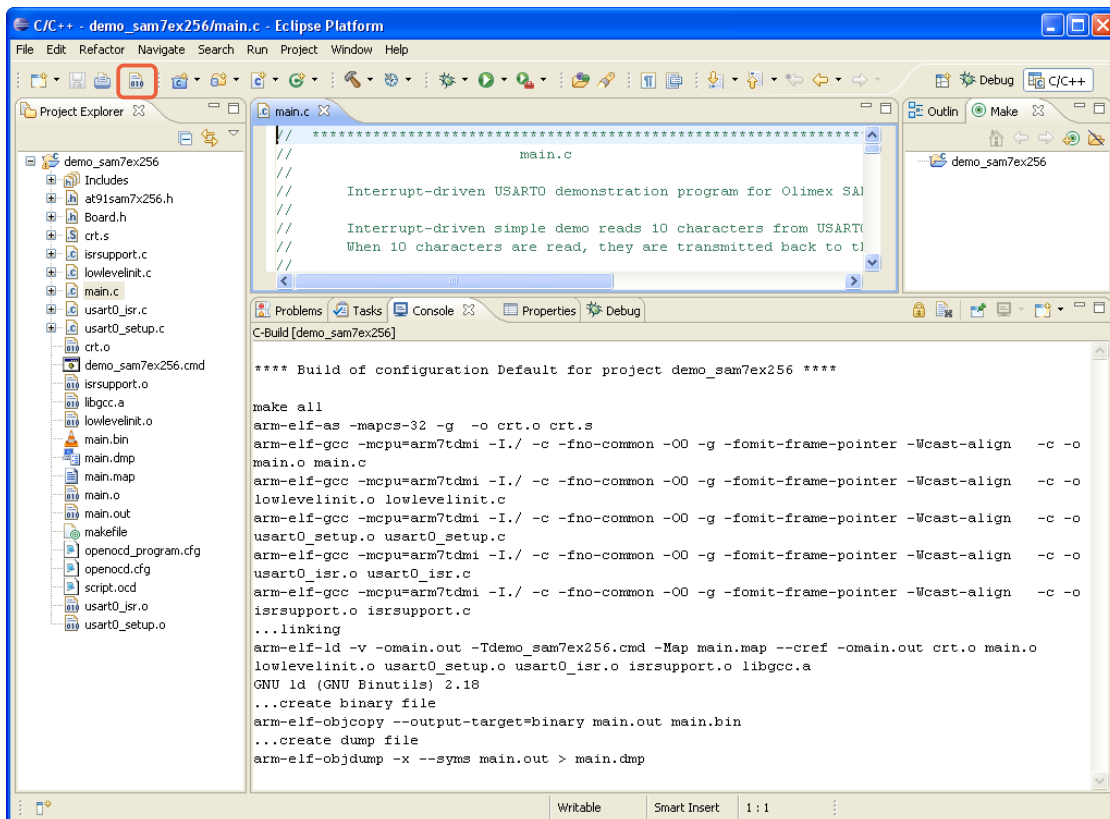


Now the Eclipse project shows the necessary files.



Build the Project

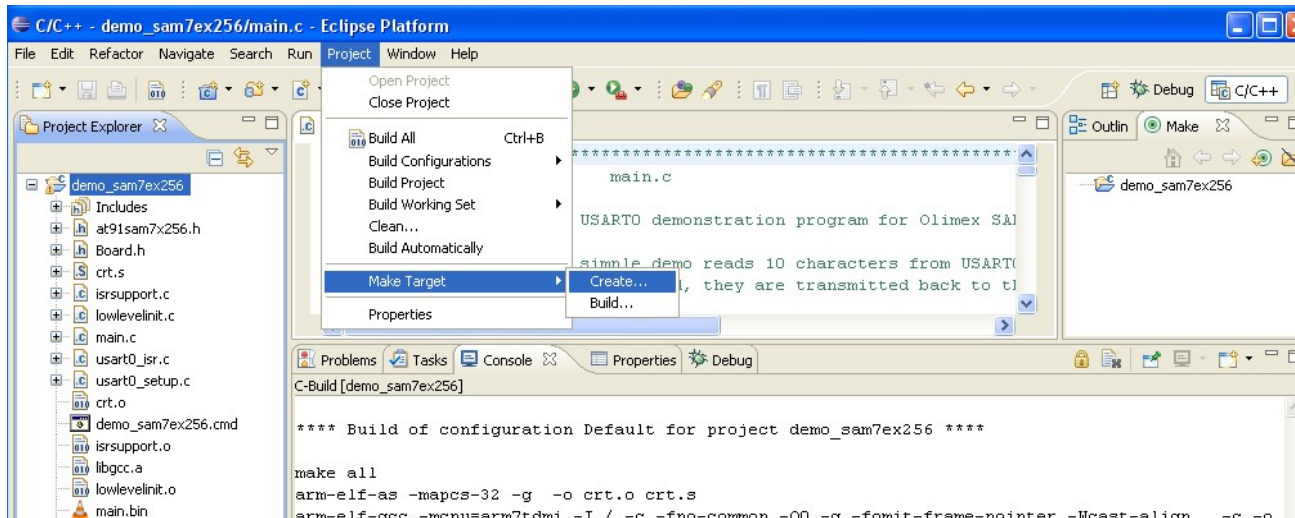
Now click on the “Build All” button as shown below. The console view will show the results of the build.



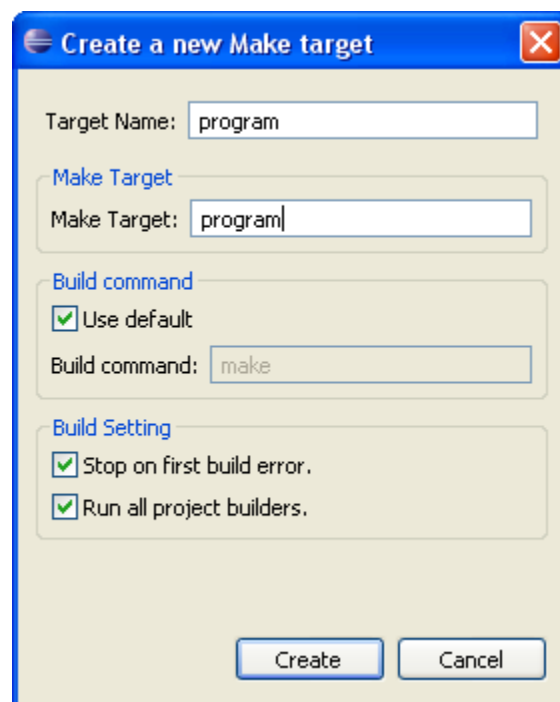
Set Up a Second Make Target for Flash Programming

The makefile has a secondary target designed to run OpenOCD in flash programming mode. To activate this from within Eclipse, we need to create an alternate Make target.

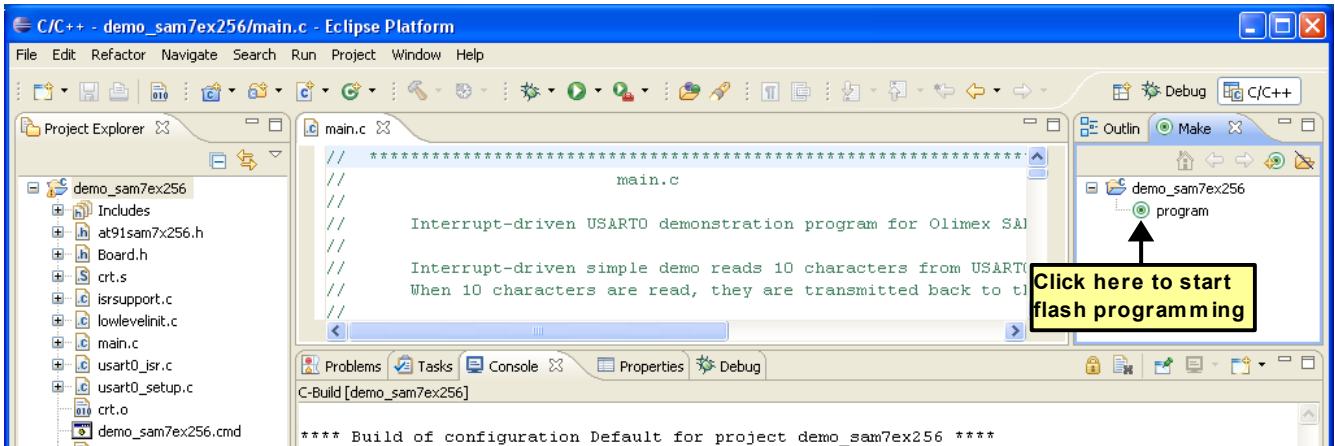
In the Project Explorer view, make sure that the project is selected (highlighted). Click on “**Project – Make Target – Create..**” as shown below.



In the “Create a new Make Target” screen shown below, type “**program**” into both the “Target Name:” text box and into the “Make Target:” text box as shown below. Click on “**Create**” to finish.



If you expand the “demo_sam7ex256” entry in the Make Target view on the upper right shown below, you'll see that we created an alternate make target called “program”.



If you double-click on the “program” target, the flash memory will be programmed as shown below. When it completes, your application should start running.

