

The Insider's Guide To The STR73x ARM[®]7 Based Microcontroller

An Engineer's Introduction To The STR73x Series
Trevor Martin BSc. (hons) CEng. MIEE

www.hitex.de



Published by Hitex (UK) Ltd.

ISBN: 0-9549988 4

First Published April 2006

Hitex (UK) Ltd.

Sir William Lyons Road
University Of Warwick Science Park
Coventry, CV4 7EZ
United Kingdom

Credits

Author: Trevor Martin
Illustrator: Sarah Latchford

Editors: Michael Beach
Cover: Wolfgang Fuller

Acknowledgements

The author would like to thank Matt Saunders of ST Microelectronics and Joachim Klein of Hitex Development Tools GmbH. for their assistance in compiling this book

© Hitex (UK) Ltd., 02/06/2006

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical or photocopying, recording or otherwise without the prior written permission of the Publisher.

Contents

1	Chapter 1: The ARM7 CPU Core	9
1.1	Outline	9
1.2	The Pipeline	9
1.3	Registers	10
1.4	Current Program Status Register.....	11
1.5	Exception Modes.....	12
1.6	ARM 7 Instruction Set	14
1.6.1	Branching	16
1.6.2	Data Processing Instructions	17
1.6.2.1	Copying Registers.....	18
1.6.2.2	Copying Multiple Registers	18
1.7	Swap Instruction.....	19
1.8	Modifying The Status Registers	19
1.9	Software Interrupt.....	20
1.10	MAC Unit.....	21
1.11	THUMB Instruction Set	22
1.12	Summary.....	23
2	Chapter 2: Software Development	25
2.1	Outline	25
2.2	The Development Tools	26
2.2.1	StartEasy.....	26
2.2.2	HiTOP IDE	26
2.2.3	Which Compiler?.....	26
2.2.4	DA-C.....	26
2.2.5	TESSY.....	27
2.3	Tutorial	27
2.4	Startup Code	27
2.5	ARM Procedure Call Standard.....	30
2.6	Interworking ARM and THUMB.....	31
2.7	STDIO libraries.....	31
2.8	Accessing Peripherals.....	31
2.9	Interrupt Service Routines.....	32
2.10	Software Interrupt.....	33
2.11	In-Line Functions.....	34
2.11.1	Inline Assembler.....	34
2.12	Linker Script Files.....	34
2.13	C++ support	36
2.14	Hardware Debugging Tools	37
2.15	Important	38
2.16	Summary	39
3	Chapter 3: System Peripherals	40
3.1	Outline	40
3.2	Bus Structure	41
3.3	Memory Map	42
3.4	On-Chip FLASH	44
3.5	The System Memory Mode Bootloader	45
3.5.1	User FLASH Programming	46

3.5.2	FLASH Memory Protection	49
3.5.3	Code Enable And Disable Debug Protection	49
3.5.4	Low Power FLASH Modes	50
3.5.5	System Peripherals	50
3.5.6	Configuration Registers.....	50
3.5.7	Important Note	52
3.5.8	STR7 Power supplies.....	53
3.5.9	Low Voltage Protection	53
3.6	Clock Structure.....	54
3.7	Setting Up The Clocks	55
3.8	Clock Protection	57
3.9	Power, Reset and Clock Control Unit (PRCCU)	58
3.9.1	Low Power Modes.....	60
3.9.2	Slow Mode	60
3.9.3	Wait For Interrupt	60
3.9.4	STOP Mode and Wake-Up Unit.....	61
3.10	Wake-Up Timer	62
3.10.1	Halt.....	63
3.10.2	Low Power Modes: Important Note.....	64
3.11	Advanced Peripheral Bus Bridges	64
3.11.1	Software Reset.....	64
3.11.2	Error Registers	64
3.12	Interrupt Structure	65
3.12.1	General Purpose Interrupt.....	65
3.12.2	Leaving An IRQ Interrupt	67
3.12.3	Nested Interrupt Handling	68
3.12.4	FIQ Interrupt.....	69
3.12.5	Leaving an FIQ interrupt	70
3.12.6	External Interrupts	70
3.12.7	DMA Controller.....	71
3.13	Native Bus Arbitrator	75
3.14	Conclusion	75
4	Chapter 4: User Peripherals	77
4.1	Outline	77
4.2	General Purpose Peripherals.....	78
4.3	GPIO	78
4.4	Timebase Timer	80
4.5	Timer Counters	82
4.5.1	Input Capture	83
4.5.2	PWM Input Capture.....	84
4.5.3	Output Compare.....	84
4.5.4	PWM Output.....	85
4.5.5	One Pulse Mode	85
4.5.6	DMA Support.....	86
4.6	PWM Module.....	87
4.7	Real Time Clock.....	89
4.8	Analog To Digital Converter	92
4.8.1	Configuration	93
4.8.2	One-Shot Mode.....	94
4.8.3	Scan Mode	94
4.9	Injection Conversion.....	95
4.9.1	Analog Watchdog.....	95
4.9.2	Interrupts	96

4.9.3	DMA Support.....	97
4.9.4	Low Power Operation.....	97
4.10	Watchdog	98
4.11	Communications Peripherals	100
4.11.1	UART.....	100
4.12	Buffered SPI.....	103
4.12.1	SPI Master Mode	105
4.12.2	SPI Slave Mode	106
4.12.3	SPI Error Interrupts	106
4.12.4	DMA Support.....	106
4.12.5	The I2C Module.....	108
4.12.5.1	I2C Addressing	110
4.12.5.2	Slave Mode	110
4.12.5.3	I2C Master Mode	112
4.12.6	The CAN Controllers	114
4.12.6.1	ISO 7 Layer Model.....	114
4.12.6.2	CAN Node Design.....	115
4.12.6.3	CAN Message Objects.....	116
4.12.6.4	CAN Bus Arbitration.....	117
4.12.6.5	STR7 CAN Module	118
4.12.6.6	Deterministic CAN Protocols.....	129
4.13	Summary	129
5	Chapter 5: Tutorial Exercises	132
5.1	Introduction	132
5.2	Further STR730 Examples.....	132
5.3	Exercise 0: Installing the Software.....	133
5.4	Setting up the Hardware	134
5.5	Overview	134
5.6	Exercise 1: Setting Up Your First Project.....	135
5.6.1	StartEasy.....	135
5.6.2	Project Structure	140
5.6.3	HiTOP Debugger.....	142
5.6.3.1	Editing Your Project	145
5.6.3.2	Run Control.....	146
5.6.3.3	Viewing Data.....	150
5.6.3.4	HiTOP Project Settings	152
5.6.3.5	Advanced Breakpoints	153
5.6.3.6	Script Language	153
5.7	Exercise 2: Startup Code	156
5.7.1	Exercise 3: Interworking ARM & THUMB Instruction Sets	158
5.8	Exercise 4: Software Interrupt.....	161
5.9	Exercise 6: STR730 System Memory Mode Bootloader	162
5.9.1	Running The Bootloader Example	162
5.10	Exercise 5: C++.....	163
5.11	Exercise 7: FLASH Programming	164
5.12	Exercise 8: Clock Configuration.....	165
5.13	Exercise 9: Low Power Modes.....	166
5.14	Exercise 10: IRQ Interrupts.....	167
5.15	Exercise 11: FIQ Interrupt.....	170
5.16	Exercise 12: Memory to Memory DMA transfer	171
5.17	Exercise 13 : General Purpose IO (GPIO).....	172
5.18	Exercise 14 : Timebase timer	173
5.19	Exercise 15: Timer	174

5.20	Exercise 16: PWM Module.....	175
5.21	Exercise 17: Analog to Digital Converter	176
5.22	Exercise 18: Watchdog	177
5.23	Exercise 19: UART.....	178
5.24	Exercise 20: BSPI	179
5.25	Exercise 21: I2C.....	180
5.26	Exercise 22: CAN.....	181
6	Bibliography	184
6.1	Publications	184
6.2	Web URL.....	184

Introduction

This book is intended as a hands-on guide for anyone planning to use the STR7 family of microcontrollers in a new design. It is laid out both as a reference book and as a tutorial. It is assumed that you have some experience in programming microcontrollers for embedded systems and are familiar with the C language. The bulk of technical information is spread over the first four chapters, which should be read in order if you are completely new to the STR7 and the ARM7 CPU.

The first chapter gives an introduction to the major features of the ARM7 CPU. Reading this chapter will give you enough understanding to be able to program any ARM7 device. If you want to develop your knowledge further, there are a number of excellent books which describe this architecture and some of these are listed in the bibliography. Chapter Two is a description of how to write C programs to run on an ARM7 processor and, as such, describes specific extensions to the ISO C standard which are necessary for embedded programming.

Having read the first two chapters you should understand the processor and its development tools. Chapter Three then introduces the STR7 system peripherals. This chapter describes the system architecture of the STR7 family and how to set the chip up for its best performance. In Chapter Four we look at the on-chip user peripherals and how to configure them for our application code.

Throughout these chapters various exercises are listed. Each of these exercises is described in detail in Chapter Five, the Tutorial section. The Tutorial contains a worksheet for each exercise which steps you through an important aspect of the STR7. All of the exercises are based on the Hitex STR73x evaluation kit which comes with an STR7 evaluation board and a JTAG debugger, as well as the GCC ARM compiler toolchain. It is hoped that by reading the book and doing the exercises you will quickly become familiar with the STR73x family of microcontrollers.

1 Chapter 1: The ARM7 CPU Core

1.1 Outline

The CPU at the heart of the STR7 family is an ARM7. You do not need to be an expert in ARM7 programming to use the STR7, as many of the complexities are taken care of by the C compiler. You do need to have a basic understanding of how the CPU is working and its unique features in order to produce a reliable design.

In this chapter we will look at the key features of the ARM7 core along with its programmers' model and we will also discuss the instruction set used to program it. This is intended to give you a good feel for the CPU used in the STR7 family. For a more detailed discussion of the ARM processors, please refer to the books listed in the bibliography.

The key philosophy behind the ARM design is simplicity. The ARM7 is a RISC computer with a small instruction set and consequently a small gate count. This makes it ideal for embedded systems. It has high performance and low power consumption and it takes a small amount of the available silicon die area.

1.2 The Pipeline

At the heart of the ARM7 CPU is the instruction pipeline. The pipeline is used to process instructions taken from the program store. On the ARM 7 a three-stage pipeline is used.



A three-stage pipeline is the simplest form of pipeline and does not suffer from the kind of hazards seen in pipelines with more stages, such as read-before-write. The pipeline has hardware independent stages that execute one instruction while decoding a second and fetching a third. The pipeline speeds up the throughput of CPU instructions so effectively that most ARM instructions can be executed in a single cycle. The pipeline works most efficiently on linear code. As soon as a branch is encountered, the pipeline is flushed and must be refilled before full execution speed can be resumed. As we shall see, the ARM instruction set has some interesting features which help smooth out small jumps in your code in order to get the best flow of code through the pipeline. As the pipeline is part of the CPU, the programmer does not have any exposure to it. However, it is important to remember that the PC is running eight bytes ahead of the current instruction being executed, so care must be taken when calculating offsets used in PC relative addressing.

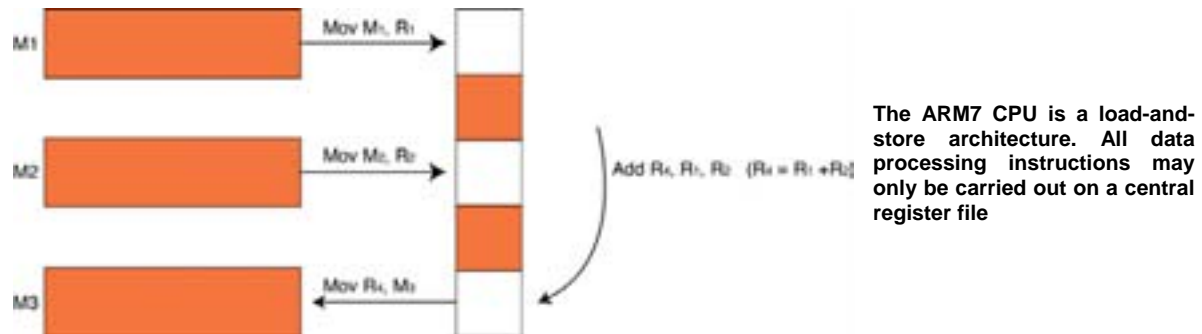
For example, the instruction:

```
0x4000 LDR PC, [PC, #4]
```

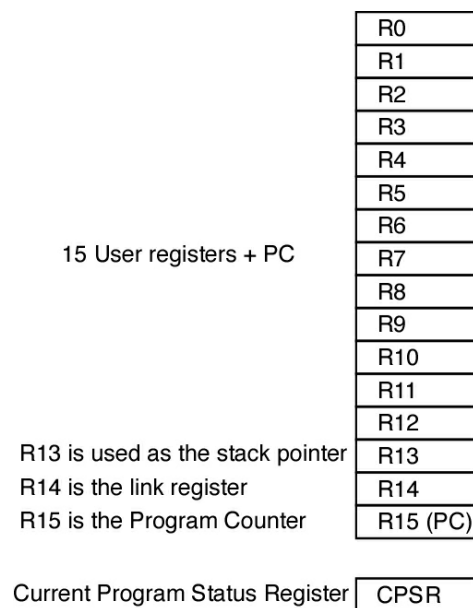
will load the contents of the address PC+4 into the PC. As the PC is running eight bytes ahead then the contents of address 0x400C will be loaded into the PC and not 0x4004 as you might expect on first inspection.

1.3 Registers

The ARM7 is a load-and-store architecture, so in order to perform any data processing instructions the data has first to be moved from the memory store into a central set of registers, the data processing instruction has to be executed and then the data is stored back into memory.



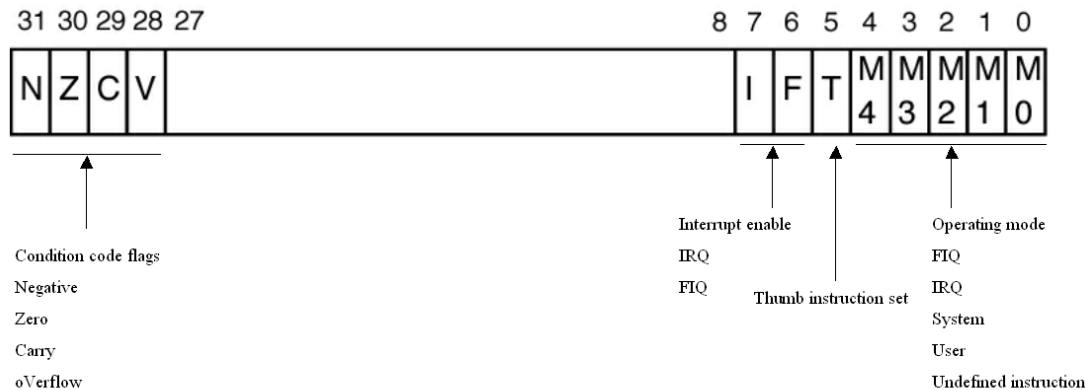
The central set of registers are a bank of 16 user registers R0 – R15. Each of these registers is 32 bits wide and R0 – R12 are user registers (in that they do not have any specific other function.) The Registers R13 – R15 do have special functions in the CPU. R13 is used as the stack pointer (SP). R14 is called the link register (LR). When a call is made to a function, the return address is automatically stored in the link register and is immediately available on return from the function. This allows quick entry and return into a 'leaf' function (a function that is not going to call further functions). If the function is part of a branch (i.e. it is going to call other functions) then the link register must be preserved on the stack (R13). Finally R15 is the program counter (PC). Interestingly, many instructions can be performed on R13 - R15 as if they were standard user registers.



The central register file has 16 word wide registers plus an additional CPU register called the current program status register. R0 – R12 are user registers R13 – R15 have special functions.

1.4 Current Program Status Register

In addition to the register bank there is an additional 32 bit wide register called the 'current program status register' (CPSR). The CPSR contains a number of flags which report and control the operation of the ARM7 CPU.



The Current Program Status Register contains condition code flags which indicate the result of data processing operations and User flags which set the operating mode and enable interrupts. The T bit is for reference only

The top four bits of the CPSR contain the condition codes which are set by the CPU. The condition codes report the result status of a data processing operation. From the condition codes you can tell if a data processing instruction generated a negative, zero, carry or overflow result. The lowest eight bits in the CPSR contain flags which may be set or cleared by the application code. Bits 7 and 8 are the I and F bits. These bits are used to enable and disable the two interrupt sources which are external to the ARM7 CPU. All of the STR7 peripherals are connected to these two interrupt lines as we shall see later. You should be careful when programming these two bits, because in order to disable either interrupt source the bit must be set to '1', not '0' as you might expect. Bit 5 is the THUMB bit.

The ARM7 CPU is capable of executing two instruction sets; the ARM instruction set which is 32 bits wide and the THUMB instruction set which is 16 bits wide. Consequently the T bit reports which instruction set is being executed. Your code should not try to set or clear this bit to switch between instruction sets. We will see the correct entry mechanism a bit later. The last five bits are the mode bits. The ARM7 has seven different operating modes. Your application code will normally run in the user mode with access to the register bank R0 – R15 and the CPSR as already discussed. However in response to an exception such as an interrupt, memory error or software interrupt instruction, the processor will change modes. When this happens the registers R0 – R12 and R15 remain the same, but R13 (LR) and R14 (SP) are replaced by a new pair of registers unique to that mode. This means that each mode has its own stack and link register. In addition, the fast interrupt mode (FIQ) has duplicate registers for R7 – R12. This means that you can make a fast entry into an FIQ interrupt without the need to preserve registers onto the stack.

Each of the modes except user mode has an additional register called the “saved program status register”. If your application is running in user mode when an exception occurs, the mode will change and the current contents of the CPSR will be saved into the SPSR. The exception code will run and on return from the exception the context of the CPSR will be restored from the SPSR allowing the application code to resume execution. The operating modes are listed below.

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7_fiq	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

The ARM7 CPU has six operating modes which are used to process exceptions. The shaded registers are banked memory that is “switched in” when the operating mode changes. The SPSR register is used to save a copy of the CPSR when the switch occurs

1.5 Exception Modes

When an exception occurs, the CPU will change modes and the PC will be forced to an exception vector. The vector table starts from address zero with the reset vector and then has an exception vector every four bytes.

Exception	Mode	Address
Reset	Supervisor	0x00000000
Undefined instruction	Undefined	0x00000004
Software interrupt (SWI)	Supervisor	0x00000008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C
Data Abort (data access memory abort)	Abort	0x00000010
IRQ (interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Each operating mode has an associated interrupt vector. When the processor changes mode the PC will jump to the associated vector.

NB. there is a missing vector at 0x00000014

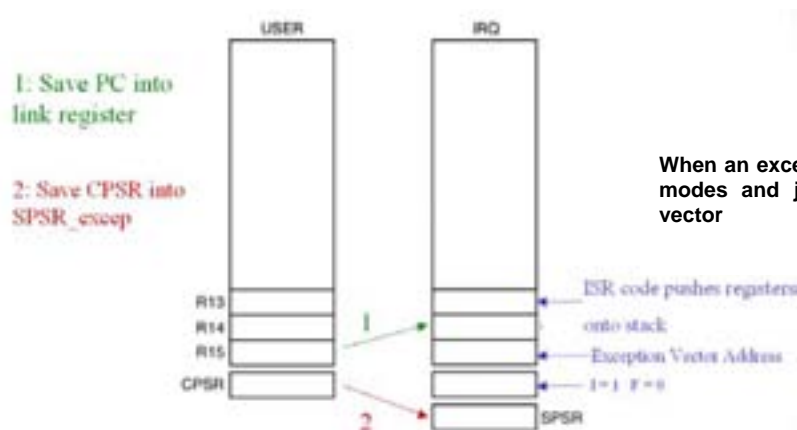
NB: There is a gap in the vector table because there is a missing vector at 0x00000014. This location was used on an earlier ARM architecture and has been preserved on ARM7 to ensure software compatibility between different ARM architectures. However in the STR7 family these four bytes are used for a very special purpose as we shall see later.

Priority	Exception
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction SWI

Each of the exception sources has a fixed priority. The on chip peripherals are served by FIQ and IRQ interrupts. Each peripheral's priority may be assigned within these groups

If multiple exceptions occur then there is a fixed priority as described below:

When an exception occurs, for example an IRQ exception, the following actions are taken: First the address of the next instruction to be executed (PC + 4) is saved into the link register. Then the CPSR is copied into the SPSR of the exception mode that is about to be entered (i.e. SPSR_irq). The PC is then filled with the address of the exception mode interrupt vector. In the case of the IRQ mode this is 0x00000018. At the same time the mode is changed to IRQ mode, which causes R13 and R14 to be replaced by the IRQ R13 and R14 registers. On entry to the IRQ mode, the I bit in the CPSR is set, causing the IRQ interrupt line to be disabled. If you need to have nested IRQ interrupts, your code must manually re-enable the IRQ interrupt and push the link register onto the stack in order to preserve the original return address. From the exception interrupt vector your code will jump to the exception ISR. The first thing your code must do is to preserve any of the registers R0-R12 that the ISR will use by pushing them onto the IRQ stack. Once this is done you can begin processing the exception.



Once your code has finished processing the exception it must return back to the user mode and continue where it left off. However the ARM instruction set does not contain a “return” or “return from interrupt” instruction so manipulating the PC must be done by regular instructions. The situation is further complicated by there being a number of different return cases. First of all, consider the SWI instruction. In this case the SWI instruction is executed, the address of the next instruction to be executed is stored in the Link register and the exception is processed. In order to return from the exception all that is necessary is to move the contents of the link register into the PC and processing can continue. However in order to make the CPU switch modes back to user mode, a modified version of the move instruction is used and this is called MOVS (more about this later). Hence for a software interrupt the return instruction is:

```
MOVS      R15,R14      ; Move Link register into the PC and switch modes.
```

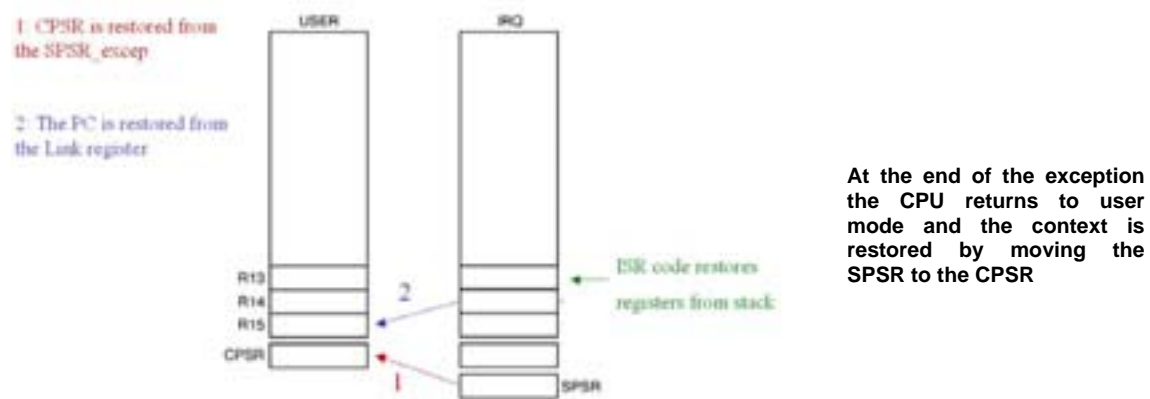
However, in the case of the FIQ and IRQ instructions, when an exception occurs the current instruction being executed is discarded and the exception is entered. When the code returns from the exception the link register contains the address of the discarded instruction plus four. In order to resume processing at the correct point we need to roll back the value in the Link register by four. In this case we use the subtract instruction to deduct four from the link register and store the results in the PC. As with the move instruction, there is a form of the subtract instruction which will also restore the operating mode. For an IRQ, FIQ or Prog Abort, the return instruction is:

```
SUBS R15, R14, #4
```

In the case of a data abort instruction, the exception will occur one instruction after execution of the instruction which caused the exception. In this case we will ideally enter the data abort ISR, sort out the problem with the memory and return to reprocess the instruction that caused the exception. This means we have to roll back the PC by two instructions, i.e. the discarded instruction and the instruction that caused the exception. In other words subtract eight from the link register and store the result in the PC. For a data abort exception the return instruction is:

```
SUBS R15, R14, #8
```

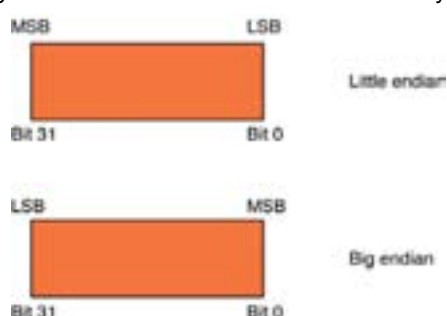
Once the return instruction has been executed, the modified contents of the link register are moved into the PC, the user mode is restored and the SPSR is restored to the CPSR. Also, in the case of the FIQ or IRQ exceptions, the relevant interrupt is enabled. This exits the privileged mode and returns to the user code ready to continue processing.



1.6 ARM 7 Instruction Set

Now that we have an idea of the ARM7 architecture, programmers' model and operating modes, we need to take a look at its instruction set (or rather sets.) Since all our programming examples are written in C, there is no need to be an expert ARM7 assembly programmer. However an understanding of the underlying machine code is very important in developing efficient programs. Before we start our overview of the ARM7 instructions it is important to set out a few technicalities. The ARM7 CPU has two instruction sets: the ARM instruction set which has 32-bit wide instructions and the THUMB instruction set which has 16-bit wide instructions. In the following section the use of the word ARM means the 32-bit instruction set and ARM7 refers to the CPU.

The ARM7 is designed to operate as a big-endian or little-endian processor. That is, the MSB is located at the high order bit or the low order bit. You may be pleased to hear that the STR7 family fixes the endianness of the processor as little endian (i.e. MSB at highest bit address), which does make it a lot easier to work with. However the ARM7 compiler you are working with will be able to compile code as little endian or big endian. You must be sure you have it set correctly or the compiled code will be back to front.



The ARM7 CPU is designed to support code compiler in big endian or little endian format. The ST silicon is fixed as little endian.

One of the most interesting features of the ARM instruction set is that every instruction may be conditionally executed. In a more traditional microcontroller the only conditional instructions are conditional branches and maybe a few others like bit test and set. However in the ARM instruction set the top four bits of the operand are compared to the condition codes in the CPSR. If they do not match, then the instruction is not executed and passes through the pipeline as a NOP (no operation).



Every ARM (32-bit) instruction is conditionally executed. The top four bits are ANDed with the CPSR condition codes. If they do not match the instruction is executed as a NOP

So it is possible to perform a data processing instruction which affects the condition codes in the CPSR. Then, depending on this result, the following instructions may or may not be carried out. The basic Assembler instructions such as MOV or ADD can be prefixed with sixteen conditional mnemonics, which define the condition code states to be tested for.

Suffix	Flags	Meaning
EQ	Z set	equal
NE	Z clear	not equal
CS	C set	unsigned higher or same
CC	C clear	unsigned lower
MI	N set	negative
PL	N clear	positive or zero
VS	V set	overflow
VC	V clear	no overflow
HI	C set and Z clear	unsigned higher
LS	C clear and Z set	unsigned lower or same
GE	N equals V	greater or equal
LT	N not equal to V	less than
GT	Z clear AND (N equals V)	greater than
LE	Z set OR (N not equal to V)	less than or equal
AL	Ignored	always

Each ARM (32- bit) instruction can be prefixed by one of 16 condition codes. Hence each instruction has 16 different variants.

So for example:

```
EQMOV R1, #0x00800000
```

will only move 0x00800000 into the R1 if the last result of the last data processing instruction was equal and consequently set the Z flag in the CPSR. The aim of this conditional execution of instructions is to keep a smooth flow of instructions through the pipeline. Every time there is a branch or jump, the pipeline is flushed and must be refilled and this causes a dip in overall performance. In practice, there is a break-even point between effectively forcing NOP instructions through the pipeline and a traditional conditional branch and refill of the pipeline. This break-even point is three instructions, so a small branch such as:

```
if( x<100)
{
    x++;
}
```

would be most efficient when coded using conditional execution of ARM instructions.

The main instruction groups of the ARM instruction set fall into six different categories, Branching, Data Processing, Data Transfer, Block Transfer, Multiply and Software Interrupt.

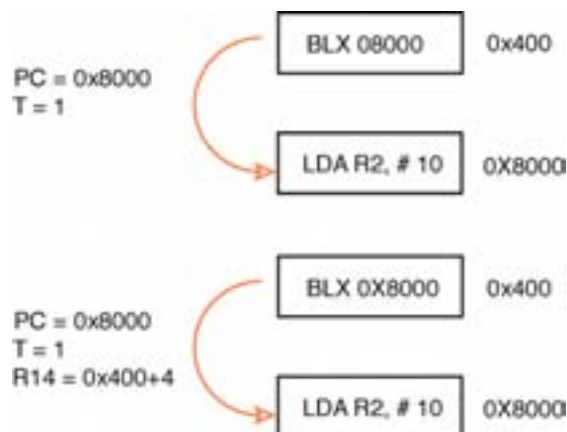
1.6.1 Branching

The basic branch instruction (as its name implies) allows a jump forwards or backwards of up to 32 MB. A modified version of the branch instruction, the branch link, allows the same jump but stores the current PC address plus four bytes in the link register.



The branch instruction has several forms. The branch instruction will jump you to a destination address. The branch link instruction jumps to the destination and stores a return address in R14.

So the branch link instruction is used as a call to a function storing the return address in the link register. The branch instruction can be used to branch on the contents of the link register to make the return at the end of the function. By using the condition codes we can perform conditional branching and conditional calling of functions. The branch instructions have two other variants called "branch exchange" and "branch link exchange". These two instructions perform the same branch operation, but also swap instruction operation from ARM to THUMB and vice versa.

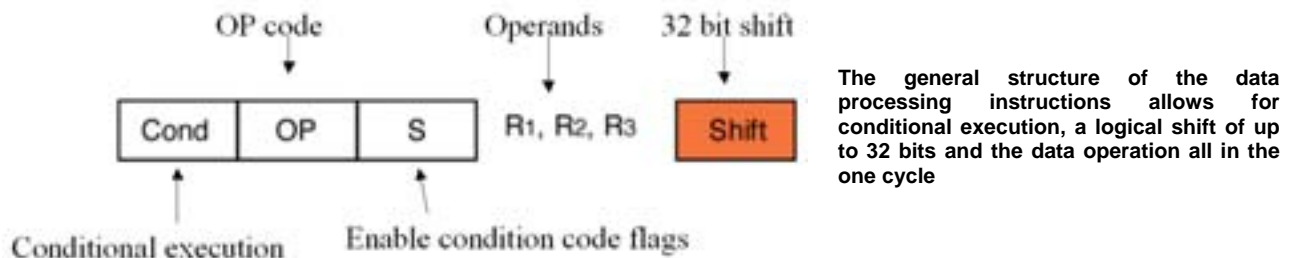


The branch exchange and branch link exchange instructions perform the same jumps as branch and branch link but also swap instruction sets from ARM to THUMB and vice versa.

This is the only method you should use to swap instruction sets, as directly manipulating the "T" bit in the CPSR can lead to unpredictable results.

1.6.2 Data Processing Instructions

The general form for all data processing instructions is shown below. Each instruction has a result register and two operands. The first operand must be a register, but the second can be a register or an immediate value.



In addition, the ARM7 core contains a barrel shifter which allows the second operand to be shifted by a full 32-bits within the instruction cycle. The “S” bit is used to control the condition codes. If it is set, the condition codes are modified depending on the result of the instruction. If it is clear, no update is made. If, however, the PC (R15) is specified as the result register and the S flag is set, this will cause the SPSR of the current mode to be copied to the CPSR. This is used at the end of an exception to restore the PC and switch back to the original mode. Do not try this when you are in the USER mode as there is no SPSR and the result would be unpredictable.

Mnemonic	Meaning
AND	Logical bitwise AND
EOR	Logical bitwise exclusive OR
SUB	Subtract
RSB	Reverse Subtract
ADD	Add
ADC	Add with carry
SBC	Subtract with carry
RSC	Reverse Subtract with carry
TST	Test
TEQ	Test Equivalence
CMP	Compare
CMN	Compare negated
ORR	Logical bitwise OR
MOV	Move
BIC	Bit clear
MVN	Move negated

These features give us a rich set of data processing instructions which can be used to build very efficiently-coded programs, or to give a compiler-designer nightmares. An example of a typical ARM instruction is shown below.

```
if(Z ==1)R1 = R2+(R3x4)
```

Can be compiled to:

```
EQADDS R1,R2,R3,LSL #2
```

1.6.2.1 Copying Registers

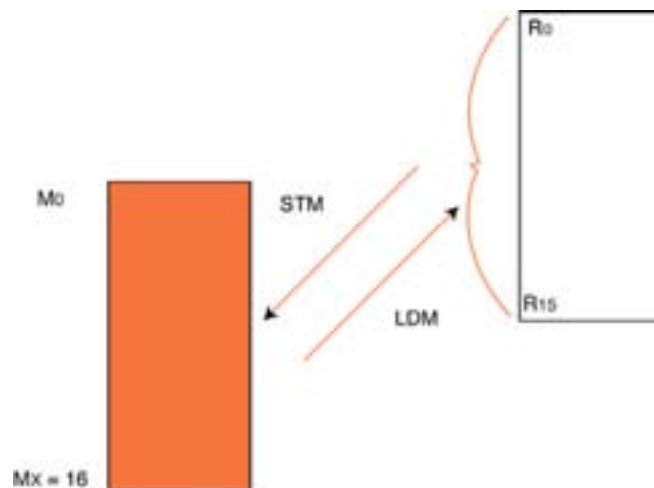
The next group of instructions are the data transfer instructions. The ARM7 CPU has load-and-store register instructions that can move signed and unsigned Word, Half Word and Byte quantities to and from a selected register.

Mnemonic	Meaning
LDR	Load Word
LDRH	Load Half Word
LDRSH	Load Signed Half Word
LDRB	Load Byte
LRDSB	Load Signed Byte
STR	Store Word
STRH	Store Half Word
STRSH	Store Signed Half Word
STRB	Store Byte
STRSB	Store Signed Half Word

Since the register set is fully orthogonal it is possible to load a 32-bit value into the PC, forcing a program jump anywhere within the processor address space. If the target address is beyond the range of a branch instruction, a stored constant can be loaded into the PC.

1.6.2.2 Copying Multiple Registers

In addition to load and storing single register values, the ARM has instructions to load and store multiple registers. So with a single instruction, the whole register bank or a selected subset can be copied to memory and restored with a second instruction.

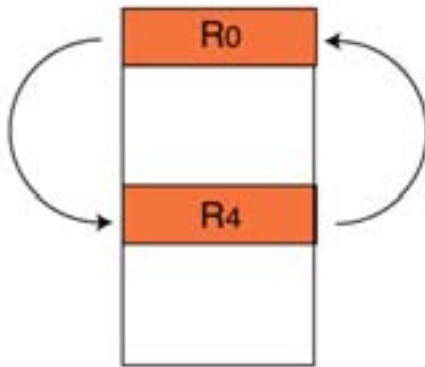


The load and store multiple instructions allow you to save or restore the entire register file or any subset of registers in the one instruction

1.7 Swap Instruction

The ARM instruction set also provides support for real time semaphores with a swap instruction. The swap instruction exchanges a word between registers and memory as one atomic instruction. This prevents crucial data exchanges from being interrupted by an exception.

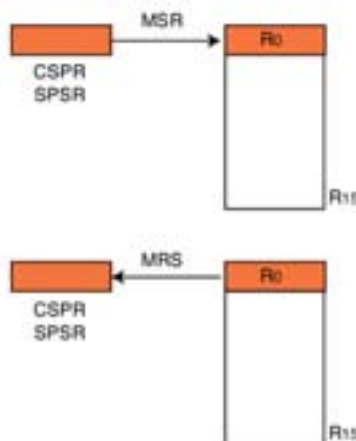
This instruction is not reachable from the C language and is supported by intrinsic functions within the compiler library.



The swap instruction allows you to exchange the contents of two registers. This takes two cycles but is treated as a single atomic instruction so the exchange cannot be corrupted by an interrupt.

1.8 Modifying The Status Registers

As noted in the ARM7 architecture section, the CPSR and the SPSR are CPU registers, but are not part of the main register bank. Only two ARM instructions can operate on these registers directly. The MSR and MRS instructions support moving the contents of the CPSR or SPSR to and from a selected register. For example, in order to disable the IRQ interrupts the contents of the CPSR must be moved to a register, the "I" bit must be set by ANDing the contents with 0x00000080 to disable the interrupt and then the CPSR must be reprogrammed with the new value.

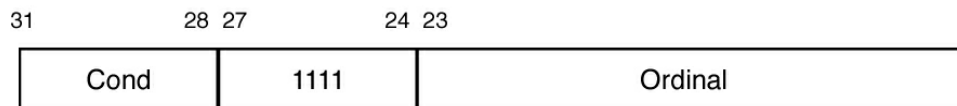


The CPSR and SPSR are not memory-mapped or part of the central register file. The only instructions which operate on them are the MSR and MRS instructions. These instructions are disabled when the CPU is in USER mode.

The MSR and MRS instructions will work in all processor modes except the USER mode. So it is only possible to change the operating mode of the process, or to enable or disable interrupts, from a privileged mode. Once you have entered the USER mode you cannot leave it, except through an exception, reset, FIQ, IRQ or SWI instruction.

1.9 Software Interrupt

The Software Interrupt Instruction generates an exception on execution, forces the processor into Supervisor mode and jumps the PC to 0x00000008. As with all other ARM instructions, the SWI instruction contains the condition execution codes in the top four bits followed by the op code. The remainder of the instruction is empty. However, it is possible to encode a number into these unused bits. On entering the software interrupt, the software interrupt code can examine these bits and decide which code to run. So it is possible to use the SWI instruction to make calls into the protected mode, in order to run privileged code or make operating system calls.



The Software Interrupt Instruction forces the CPU into SUPERVISOR mode and jumps the PC to the SWI vector. Bits 0-23 are unused and user defined numbers can be encoded into this space.

The Assembler Instruction:

```
SWI #3
```

will encode the value 3 into the unused bits of the SWI instruction. In the SWI ISR routine we can examine the SWI instruction with the following code pseudo code:

```
switch( *(R14-4) & 0x00FFFFFF) // roll back the address stored in link reg
// by 4 bytes
{
    // Mask off the top 8 bits and switch
    // on result
    case ( SWI-1)
        .....
```

Depending on your compiler, you may need to implement this yourself, or it may be done for you in the compiler implementation.

1.10 MAC Unit

In addition to the barrel shifter, the ARM7 has a built-in Multiply Accumulate Unit (MAC). The MAC supports integer and long integer multiplication. The integer multiplication instructions support multiplication of two 32-bit registers and place the result in a third 32-bit register (modulo32). A multiply-accumulate instruction will take the same product and add it to a running total. Long integer multiplication allows two 32-bit quantities to be multiplied together and the 64-bit result is placed in two registers. Similarly a long multiply and accumulate is also available.

Mnemonic	Meaning	Resolution
MUL	Multiply	32 bit result
MULA	Multiply accumulate	32 bit result
UMULL	Unsigned multiply	64 bit result
UMLAL	Unsigned multiply accumulate	64 bit result
SMULL	Signed multiply	64 bit result
SMLAL	Signed multiply accumulate	64 bit result

1.11 THUMB Instruction Set

Although the ARM7 is a 32-bit processor, it has a second 16-bit instruction set called THUMB. The THUMB instruction set is really a compressed form of the ARM instruction set.



This allows instructions to be stored in a 16-bit format, expanded into ARM instructions and then executed. Although the THUMB instructions will result in lower code performance compared to ARM instructions, they will achieve a much higher code density. So, in order to build a reasonably-sized application that will fit on a small single chip microcontroller, it is vital to compile your code as a mixture of ARM and THUMB functions. This process is called interworking and is easily supported on all ARM compilers. By compiling code in the THUMB instruction set you can get a space saving of 30%, while the same code compiled as ARM code will run 40% faster.

The THUMB instruction set is much more like a traditional microcontroller instruction set. Unlike the ARM instructions, THUMB instructions are not conditionally executed (except for conditional branches). The data processing instructions have a two-address format, where the destination register is one of the source registers:

ARM Instruction

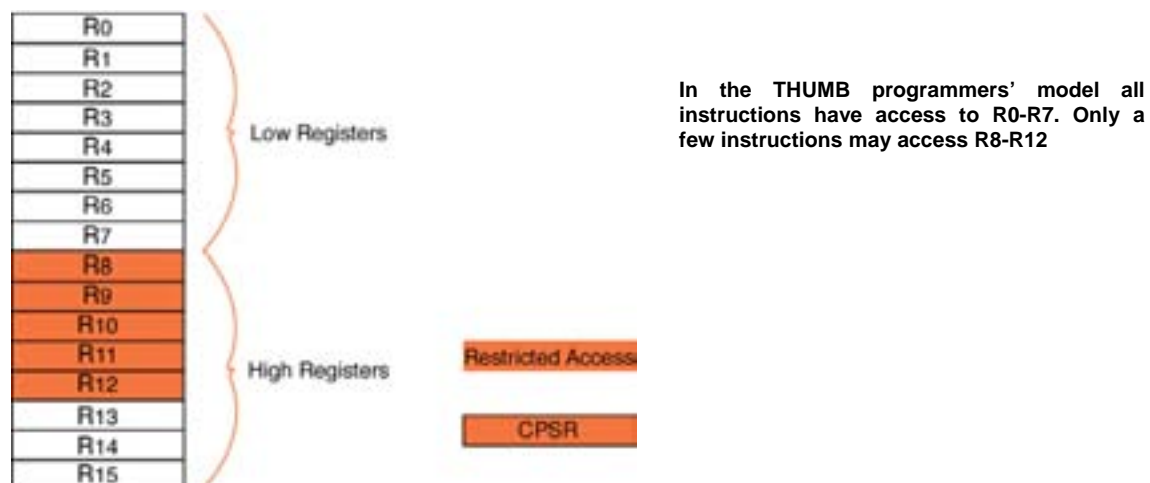
ADD R0, R0,R1

THUMB Instruction

ADD R0,R1

R0 = R0+R1

The THUMB instruction set does not have full access to all registers in the register file. All data processing instructions have access to R0 –R7 (these are called the “low registers”).

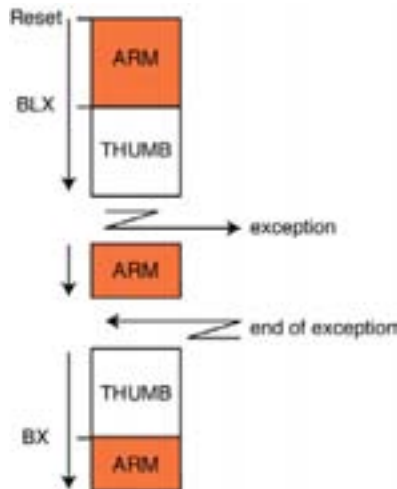


However access to R8-R12 (the “high registers”) is restricted to a few instructions:

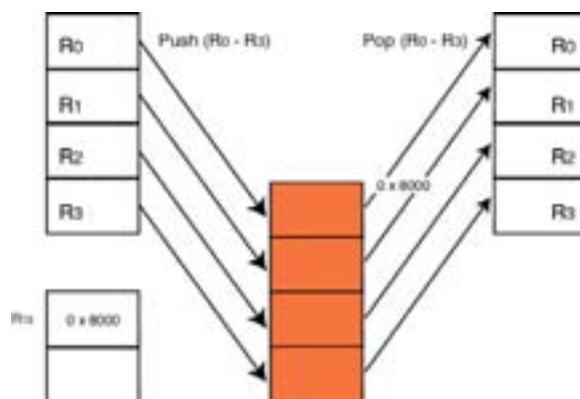
MOV, ADD, CMP

The THUMB instruction set does not contain MSR and MRS instructions, so you can only indirectly affect the CPSR and SPSR. If you need to modify any user bits in the CPSR you must change to ARM mode. You can

change modes by using the BX and BLX instructions. Also, when you come out of RESET, or enter an exception mode, you will automatically change to ARM mode.



The THUMB instruction set has the more traditional PUSH and POP instructions for stack manipulation. They implement a fully descending stack, hardwired to R13.



The THUMB instruction set has dedicated PUSH and POP instructions which implement a descending stack using R13 as a stack pointer

Finally, the THUMB instruction set does contain a SWI instruction which works in the same way as in the ARM instruction set, but it only contains 8 unused bits, to give a maximum of 255 SWI calls.

1.12 Summary

At the end of this chapter you should have a basic understanding of the ARM7 CPU. Please see the bibliography for a list of books that address the ARM7 in more detail. Also included on the CD is a copy of the ARM7 user manual.

2 Chapter 2: Software Development

2.1 Outline

Having read Chapter One, you should now have an understanding of the ARM7 CPU. In this chapter we will look at how to write and debug C code for the ARM7. The example programs in this chapter will concentrate on demonstrating how to use the unique features of the ARM7 discussed in the first chapter.

Since the ARM7 is rapidly becoming an industry standard CPU for general purpose microcontrollers, this will allow you to develop code on the STR73x and a number of other microcontrollers.

2.2 The Development Tools

The CD that comes with the STR7 starter kit includes an evaluation toolchain for the STR7. This is complete with a number of example programs demonstrating all the major features of the STR7 microcontroller. By reading through the book and running the exercises, you will be able to very quickly familiarise yourself with the essential features of the STR7. All of these exercises are detailed in Chapter 5. Once you have read the theory section and reached an exercise, turn to the Exercise Worksheet in Chapter Five and follow the instructions. The exercises expand on the basic theory and give you a practical handle on the operation of the STR7.

Exercise 0: Installing the software

Turn to the tutorial chapter and follow the instructions in Exercise 0 which describe setting up the software and hardware toolchain that we will use for the practical examples in this book

2.2.1 StartEasy

StartEasy is the first development tool used for a new STR7 project. StartEasy is an easy-to-use CASE tool that will auto-generate the necessary Assembler and C source code for a skeleton STR7 program. StartEasy also allows you to select the configuration of the basic STR7 peripherals, interrupt structure and CPU clocks through a few simple menus. As well as generating the source code StartEasy produces a fully configured project using the HiTOP IDE, so you can easily download and run your application code.

2.2.2 HiTOP IDE

HiTOP is the front end for all Hitex debuggers and in circuit emulators. In the case of the STR7, HiTOP connects the Tantino 7-9 JTAG debugger. The JTAG allows HiTOP to download code into the STR7 FLASH or RAM and then debug the code as it runs on the microcontroller. In addition to its debugging features, HiTOP includes a programmers' editor and support for various compiler tools and make utilities. Once you have defined your project with StartEasy you can edit, compile and debug within the HiTOP IDE.

2.2.3 Which Compiler?

The HiTOP development environment can be used with several different compiler tools. These include compilers from ARM, Keil, Greenhills and IAR. There is also a port of the GCC compiler available for the ARM series of CPUs. The GCC compiler has the advantage of being a free compiler which will compile C and C++ code for all of the ARM series of CPUs.

We can see from this simple analysis that the commercial compilers are streets ahead of the GNU tools in terms of code density and speed of execution. Increasingly, the commercial compilers include direct support for ARM-based microcontrollers in the form of debuggers with support for the STR7 and dedicated compiler switches. The reasons to use each of the given compilers can be summed up as follows: if you want the fastest code and standard tools use the ARM compiler, for best code density use the Keil, if you have no budget or a simple project use the GNU. Delivered with the starter kit is an installation of the GNU compiler which integrates with the HiTOP debugger IDE and StartEasy case tool. The examples given use the GNU compiler.

2.2.4 DA-C

Also included with the development toolchain is a second editor called Development Assistant for C. This is an advanced editor specifically targeted at embedded systems developers. As well as having all the features you would expect in a programmers' editor, DA-C includes a number of advanced features that help you to produce high quality well documented C source code. DA-C includes a static checker that will analyse your code for common programming errors, generate flow charts and calling hierarchies. DA-C also includes a code browser, so you can easily navigate your code and a metrics module so the source code may be analysed using quality standard measures.

2.2.5 TESSY

The final item of software included on the CD is a software testing tool suite called TESSY. The TESSY toolset automates the functional testing of embedded microcontrollers and their target hardware. In many industries (especially Aerospace and Medical), validation of microcontroller firmware is a lengthy and important process. TESSY is especially suitable for testing small footprint microcontrollers which have small amounts of on-chip memory. Rather than build a test harness that is downloaded into the target memory of the device under test, TESSY makes no changes to the code under test but builds its test harness in the HiSCRIPT language built into the debugger. This way the target application can be fully exercised without the loss of any on-chip resources.

2.3 Tutorial

Once you have installed the software and configured the target hardware as described in Exercise 0, it is best to run through the first real exercise to generate a simple application and gain some familiarity with the development tools.

Exercise 1: Configuring a new project

The first exercise demonstrates starting a new project in StartEasy and then using this project with the HiTOP IDE

2.4 Startup Code

In our example project we have a number of source files. In practice the .c files are your source code, but the file startup.s is an Assembler module provided by Hitex to support the STR7 microcontroller. As its name implies, the startup code is located to run from the reset vector. It provides the exception vector table, as well as initialising the stack pointer for the different operating modes. It also initialises some of the on-chip system peripherals and the on-chip RAM before it jumps to the main function in your C code. The startup code will vary, depending on which ARM7 device you are using and which compiler you are using, so for your own project it is important to make sure you are using the correct file.

First of all the startup code provides the exception vector table as shown below:

```
Reset_Vec:      B      ResetEntry
                B      Undef_Handler
                B      SWI_Handler
                B      PAbt_Handler
                B      DAbt_Handler
                NOP                               /* Reserved Vector */
                LDR     PC, [PC, #-0x408]
                B      FIQ_Handler
```

The vector table is located at 0x00000000 and provides a jump to interrupt service routines (ISR) on each vector. If your code is located to run from 0x00000000 then the vector table can be made from a series of branch instructions. You must remember to pad the unused interrupt vector with a NOP, also note the odd Assembler instruction on the IRQ interrupt vector. This will be discussed in Chapter 3 when we look at the interrupt structure in more detail.

Using the Branch instruction means that the entry point to our application software and the interrupt routines must be located within the first 32Mb on the STR7 memory map since this is the address range of the branch instruction. A more generic way to handle the vector table is to use the LDR instruction to load a 32 bit constant into the PC. This method uses more memory, but allows you to locate your code anywhere in the 4Gb address range of the ARM7.

```

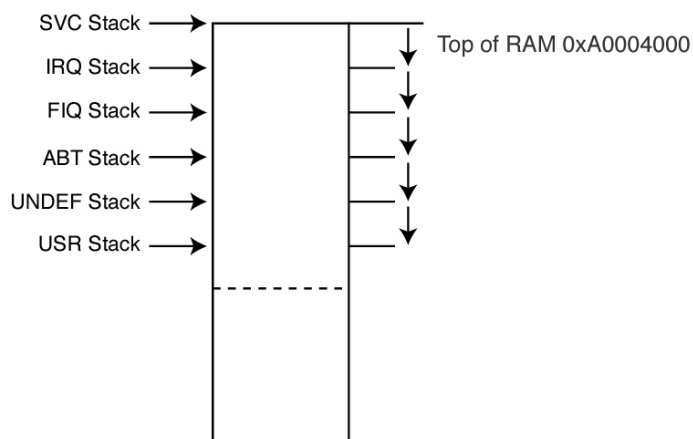
Vectors:      LDR      PC,Reset_Addr
               LDR      PC,Undef_Addr
               LDR      PC,SWI_Addr
               LDR      PC,PAbt_Addr
               LDR      PC,DAbt_Addr
               NOP                               /* Reserved Vector */
               LDR      PC,IRQ_Addr
               LDR      PC,[PC, #-0x0404]
               LDR      PC,FIQ_Addr

Reset_Addr:    DD      Reset_Handler
Undef_Addr:    DD      Undef_Handler
SWI_Addr:      DD      SWI_Handler
PAbt_Addr:     DD      PAbt_Handler

DAbt_Addr      DD      DAbt_Handler
               DD      0          /* Reserved Address */
IRQ_Addr:      DD      IRQ_Handler
FIQ_Addr:      DD      FIQ_Handler

```

The vector table and the constants table take up the first 64 bytes of memory. On the STR7 the memory at 0x00000000 may be mapped from a number of different sources either on-chip FLASH, RAM or external FLASH memory. (This is discussed more fully later on.) Whichever method you use, you are responsible for managing the vector table in the startup code, as it is not done automatically by the compiler. The startup code is also responsible for configuring the stack pointers for each of the operating modes.

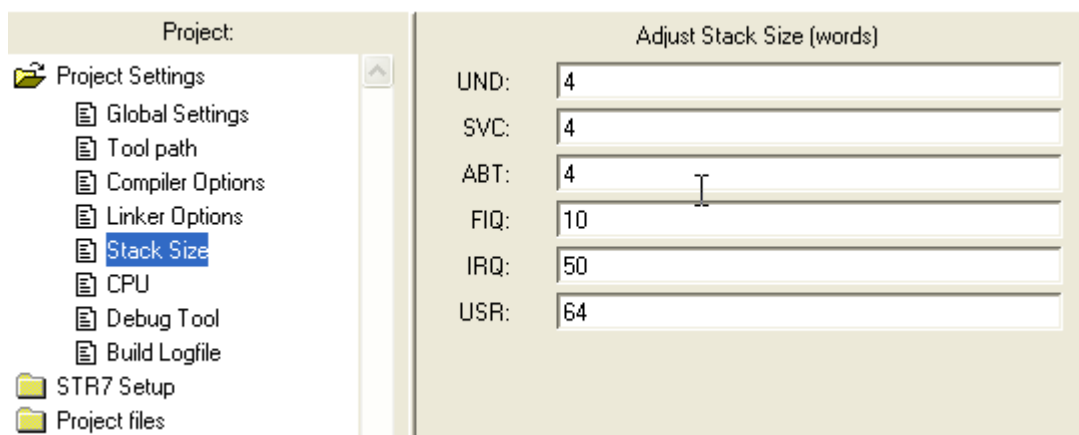


The six on-chip stack pointers (R13) are initialised at the top of on-chip memory. Care must be taken to allocate enough memory for the maximum size of each stack

Since each operating mode has a unique R13, there are effectively six stacks in the ARM7. The strategy used by the compiler is to locate user variables from the start of the on-chip RAM and grow upwards. The stacks are located at the top of memory and grow downwards. The startup code enters each different mode of the ARM7 and loads each R13 with the starting address of the stack.

		<code>// Setup Stack for each mode</code>
		<code>LDR R0, =Top_Stack</code>
		<code>// Enter Undefined Instruction Mode and set its Stack Pointer</code>
Switch mode and disable interrupts	→	<code>MSR CPSR_c, #Mode_UND I_Bit F_Bit</code>
Load address into the stack pointer	→	<code>MOV SP, R0</code>
Calculate start address of next stack	→	<code>SUB R0, R0, #UND_Stack_Size</code>
		<code>// Enter Abort Mode and set its Stack Pointer</code>
		<code>MSR CPSR_c, #Mode_ABT I_Bit F_Bit</code>
		<code>MOV SP, R0</code>
		<code>SUB R0, R0, #ABT_Stack_Size</code>
		 <code>.....FIQ,IRQ and supervisor stacks</code>
		<code>// Enter User Mode and set its Stack Pointer</code>
Finally switch to USER mode and enable interrupts	→	<code>MSR CPSR_c, #Mode_USR</code>
		<code>MOV SP, R0</code>
		<code>// Enter the C code</code>
		<code>LDR R0,=?C?INIT</code>
Check for ARM or Thumb mode.		<code>TST R0,#1 ; Bit-0 set: INIT is Thumb</code>
Load an Exit address into the link register	→	<code>LDREQ LR,=exit?A ; ARM Mode</code>
		<code>LDRNE LR,=exit?T ; Thumb Mode</code>
Jump to the C init routine		<code>BX R0</code>

Like the vector table, you are responsible for configuring the stack size. This can be done by editing the startup code directly, however the StartEasy tool provides a simple window that allows you to easily set the stack sizes.



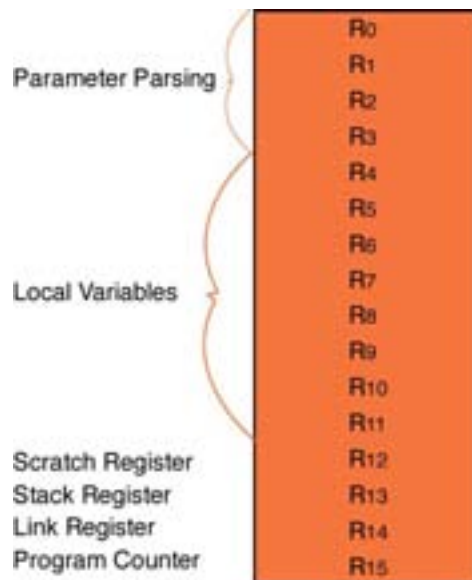
In addition, the graphical editor allows you to configure some of the STR730 system peripherals. We will see these in more detail later, but remember that they can be configured directly in the startup code.

Exercise 2: Configuring the ARM7 Startup code

This exercise focuses on the startup code. The CPU stacks are set up and checked in the debugger. The other critical areas of the startup code are also examined.

2.5 ARM Procedure Call Standard

The ARM procedure calling standard defines how the ARM7 register file is used by the compiler during runtime. In theory, the APCS allows code built in different toolsets to work together, so that you can take a library compiled by the ARM compiler and use it with the GCC toolset.



The ARM procedure call standard defines how the user CPU registers should be used by compilers. Adhering to this standard allows interworking between different manufacturers' tools

The APCS splits the register file into a number of regions. R0 to R3 are used for parameter passing between functions. If you need to pass more than 16 bytes then spilled parameters are passed via the stack. Local variables are allocated R4 – R11 and R12 is reserved as a memory location for the intra-call veneer code. As you select more options for the generated code such as re-entrancy and stack-checking, the compiler adds additional Assembler code to support these features. These Assembler veneers effectively add overhead to your code, so it is important to only enable features which you intend to use. The APCS also defines a stack frame which preserves the context of the CPU registers and also contains a pointer to the previous stack frame.

This is a very useful software debug task within an operating system such as Linux, but it isn't that useful for a JTAG debugger. The ARM procedure calling standard has a big impact on the speed of execution and stack size for the final application. Consequently for a small embedded microcontroller like the STR7 it is best to stop the compiler from using this standard. The compiler switch used to enable the APCS standard is:

`-mapcs-frame` or `-apcs`

And to disable it

`-mno-apcs-frame`

By default StartEasy generates a project which disables the use of the APCS.

2.6 Interworking ARM and THUMB

One of the most important features of the ARM7 CPU is its ability to run 16 bit THUMB code and 32 bit ARM code. In order to get a reasonably complex application to fit into the on-chip FLASH memory, it is very important to interwork these two instruction sets so that most of the application code is encoded in the THUMB instruction set and is effectively compressed to take minimal space in the on-chip FLASH memory. Any time-critical routines where the full processing power of the ARM 7 is required need to be encoded in the ARM 32 bit instruction set. When generating the code, the compiler must be enabled to allow interworking. This is achieved with the following switch:

`-mTHUMB-interwork`

The GCC compiler is designed to compile a given C module in either the THUMB or ARM instruction set. Therefore you must lay out your source code so that each module only contains functions that will be encoded as ARM or THUMB functions. By default the compiler will encode all source code in the ARM instruction set. To force a module to be encoded in the THUMB instruction set, use the following directive when you compile the code:

`-mTHUMB`

This option can be added to a give module either in StartEasy or within the HiTOP IDE.

The linker can then take both ARM and THUMB object files and produce a final executable that interworks both instruction sets.

Exercise 3: ARM-Thumb Interworking

The next exercise demonstrates setting up a project that interworks ARM and THUMB code.

2.7 STDIO libraries

The "Hello World" example in **Exercise 1** uses a simple `printf` routine which will print simple strings to a terminal. The GCC compiler has a full ANSI C library which includes the full high-level formatted I/O functions such as `scanf` and `printf`. These functions call low-level driver functions that can be modified to the I/O stream to a given peripheral. These functions are stored in a file called `syscalls.c`. If you intend to use the STDIO library you should add this module to your project and then tailor the driver functions to match your I/O device.

Bear in mind that the high level STDIO functions are quite bulky and should only be used if your application is very I/O driven.

2.8 Accessing Peripherals

Once we have built some code and got it running on an STR7 device, it will at some point be necessary to access the special function registers (SFR) in the peripherals. As all the peripherals are memory-mapped, they can be accessed as normal memory locations. Each SFR location can be accessed by 'hardwiring' a volatile pointer to its memory location as shown below.

```
#define SFR    (*((volatile unsigned long *) 0xFFFFF000))
```

StartEasy generates an include file which defines all the SFRs in the different STR7 variants. The include file is added to your skeleton project. This allows you to directly access any STR7 peripheral SFR directly using the data book naming convention.

2.9 Interrupt Service Routines

In addition to accessing the on-chip peripherals, your C code will have to service interrupt requests. It is possible to convert a standard function into an ISR as shown below:

```
void fiqint (void)    __attribute__ ((interrupt("FIQ")));
{
    IOSET1    = 0x00FF0000; //Set the LED pins
    EXTINT    = 0x00000002;  //Clear the peripheral interrupt flag
}
```

The keyword `__fiq` defines the function as an fast interrupt request service routine and will use the correct return mechanism. Other types of interrupt are supported by the keywords `__IRQ`, `__SWI` and `_UNDEF`.

As well as declaring a C function as an interrupt routine, you must link the interrupt vector to the function.

```
Vectors:      LDR      PC,Reset_Addr
               LDR      PC,Undef_Addr
               LDR      PC,SWI_Addr
               LDR      PC,PAbt_Addr
               LDR      PC,DAbt_Addr
               NOP                               /* Reserved Vector */
;             LDR      PC,IRQ_Addr
               LDR      PC,[PC, #-0x0404]        /* Vector from VicVectAddr */
               LDR      PC,FIQ_Addr

Reset_Addr:    DD      Reset_Handler
Undef_Addr:    DD      Undef_Handler?A
SWI_Addr:      DD      SWI_Handler?A
PAbt_Addr:     DD      PAbt_Handler?A
DAbt_Addr:     DD      DAbt_Handler?A
               DD      0                          /* Reserved Address */
IRQ_Addr:      DD      IRQ_Handler?A
FIQ_Addr:      DD      FIQ_Handler?A
```

The vector table is in two parts. First there is the physical vector table which has a load register instruction (LDR) on each vector. This loads the contents of a 32-bit wide memory location into the PC, forcing a jump to any location within the processor's address space. These values are held in the second half of the vector table, or constants table which follows immediately after the vector table. This means that the complete vector table takes the first 64 bytes of memory. The startup code contains predefined names for the Interrupt Service Routines (ISR). You can link your ISR functions to each interrupt vector by using the same name as your C function name. The table below shows the constants table symbols and the corresponding C function prototypes that should be used.

Exception Source	Constant	C Function Prototype
Undefined Instruction	Undef_Handler	void Undef_Handler(void) __attribute__ ((interrupt("undef")));
Software Interrupt	SWI_Handler	void SWI_Handler(void) __attribute__ ((interrupt("swi")));
Prefetch Abort	PAbt_Handler	void PAbt_Handler (void) __attribute__ ((interrupt("undef")));
Data Abort	DAbt_Handler	void DAbt_Handler (void) __attribute__ ((interrupt("undef")));
Fast Interrupt	FIQ_Handler	void FIQ_Handler (void) __attribute__ ((interrupt("fiq")));

As you can see from the table, there is no routine defined for the IRQ interrupt source. The IRQ exceptions are special cases as we will see later. Only the IRQ and FIQ interrupt sources can be disabled. The protection exceptions (Undefined instruction, Prefetch Abort, and Data Abort) are always enabled. Consequently these exceptions must always be trapped. As a minimum you should ensure that these interrupt sources are trapped in a tight loop, as shown below.

```
Pabt_Handler:      B      Pabt_Handler      ; Branch back to Pabt_Handler
```

If your code does encounter a memory error and ends up in one of these loops, you can examine the contents of the Abort Link Register to determine the address+4 of the instruction which caused the error. The SPSR will contain details of the operating mode which the CPU was in when the error occurred. From here you can backtrack and also examine the contents of the stack immediately before the application program crashed. So the CPU registers can produce some useful post mortem diagnostics if you know what you are looking for.

2.10 Software Interrupt

The Software Interrupt exception is a special case. As we have seen it is possible to encode an integer into the unused portion of the SWI opcode.

```
#define SoftwareInterrupt2 asm ("swi #02")
```

Now when we execute this line of code, it will generate a software interrupt and switch the CPU into Supervisor mode and vector the PC to the SWI interrupt vector, which will place the application into the SWI handler routine. Once we enter this routine, we need to determine what code to run. It would be possible to read the contents of a global variable and then use a switch statement to run a specific function depending on the contents of the global. However, there is a more elegant method. In the GCC compiler there is a register keyword that allows us to access a CPU register directly from C code. The declaration below declares a pointer to the link register.

```
register unsigned * link_ptr asm ("r14");
```

When we enter the software interrupt routine we can use this pointer to read the contents of the SWI instruction which generated the interrupt. This allows us to read the integer number encoded into the SWI instruction. We can then use this number to decide which function to run within the SWI handler.

```
temp = *(link_ptr-1) & 0x00FFFFFF;
```

The line above takes the contents of the link register and deducts one. Remember it is a word-wide pointer, so we are in fact deducting four bytes. This rolls the contents of the link register back by four so it is pointing at the address of the SWI instruction. Then the contents of the instruction with the top eight bits masked off (the condition codes and the SWI op code) are copied into the temp variable.

The result is that the encoded integer (in this case 2) is now stored in the temp variable and we can use its contents to decide which code to run. The SWI instruction is a convenient method of leaving User mode to enter the Supervisor mode and run some privileged code. The SWI calls can be used as part of a BIOS where all access to the special function registers is made via software interrupt calls. This in effect partitions your application code, so that all the hardware drivers run in Supervisor mode with their own stack and if necessary their own memory space. You do not have to build your code this way, but if you want to make use of it the mechanism is there.

Exercise 4: Software Interrupt

The SWI support in the GCC compiler is demonstrated in this example. You can easily partition code to run in either the User mode or in Supervisor mode.

2.11 In-Line Functions

It is also possible to increase the performance of your code by inlining your functions. The inline keyword can be applied to any function as shown below:

```
inline void NoSubroutine (void)
{
    ...
}
```

When the inline keyword is used, the function will not be coded as a subroutine, but the function code will be inserted at the point where the function is called, each time it is called. This removes the prologue and epilogue code which is necessary for a subroutine, making its execution time faster. However, you are duplicating the function every time it is called, so it is expensive in terms of your FLASH memory. The compiler will not inline functions unless you have set the optimiser to its "O2" level.

2.11.1 Inline Assembler

The compiler also allows you to use ARM or THUMB Assembler instructions within a C file. This can be done as shown below:

```
asm ( "mov r15,r2" );
```

This can be useful if you need to use features that are not supported by the C language, for example the MRS and MSR instructions.

2.12 Linker Script Files

Once you have written your source code and compiled it to object files, these files must be linked together to make the final absolute file. StartEasy generates the necessary linker switches and also a linker script file that describes the target memory layout to the linker, so it knows how to build the final application. The linker is invoked with the following switches:

```
ld ld_opt -o <project_name>.elf
```

where ld_opt is:

```
-T.\objects\<linker_file>.ld --cref -t -static -lgcc -lc -lm -nostartfiles -
Map=<project_name>.map
```

The linker script file is saved to your project directory and is given the extension .ld. It is important to understand the structure of this file as you may need to modify it as your application code grows. The linker script file is a structured text file with a number of sections that describe your project to the linker. First some search paths are defined for the compiler libraries. These search paths are defined in StartEasy and must point to the libraries of the GCC installation you are using:

```
SEARCH_DIR( "C:\Program Files\Hitex\GnuToolPackageARM\ARM-hitex-elf\lib\interwork"
)
SEARCH_DIR( "C:\Program Files\Hitex\GnuToolPackageARM\lib\gcc\ARM-hitex-
elf\4.0.0\interwork" )
```

The GCC compiler comes with several builds of the libraries. The version selected here supports ARM THUMB interworking. The next section in the script file defines the list of input object files that make up the complete project:

```
GROUP ( objects\startup.o
        objects\main.o
        objects\interrupt.o
        objects\THUMB.o )
```

If you add any additional source modules to your project you must update this list manually or regenerate the project with StartEasy.

Following the group section is the target memory layout:

```
MEMORY
{
    IntCodeRAM (rx) : ORIGIN = 0x00000000, LENGTH = 256k
    IntDataRAM (rw) : ORIGIN = 0x20000000, LENGTH = 64k
}
```

This section defines the size and location of the target memory. The description above describes the memory configuration of an STR7 used as a single chip device with 256k of FLASH memory starting from address 0x00000000 and 64K of RAM starting from 0x20000000. Later on, when we look at the external memory interface of the STR7, we will look at laying out code for targets with external memory. The final user section describes how the application code should be laid out within the target memory.

```
SECTIONS
{
    .text
    {
        ...
    }
    .data
    {
        ...
    }
}
```

The description of the application code is split into two basic sections, .text and .data. The .text section contains the executable code and the code constants, basically anything that should be located in the FLASH memory. The .data section allocates all of your volatile variables into the user RAM.

```
.text :
{
    __code_start__ = .;
    objects\startup.o (.text)          /* Startup code */
    objects\*.o (.text)
    . = ALIGN(4);
    __code_end__ = .;
    *(.glue_7t) *(.glue_7)

} >IntCodeRAM = 0
```

The text section ensures that the startup code is assigned first, so it will be placed on the reset vector. Then the remaining application code is assigned locations within the FLASH memory. The align command ensures that each relocatable section is placed on the next available word boundary.

```
.data : AT (_etext)
{
    /* used for initialized data */
    __data_start__ = .;
    PROVIDE (__data_start__ = .) ;
    *(.data)
    SORT(CONSTRUCTORS)
    __data_end__ = .;
    PROVIDE (__data_end__ = .) ;
} >IntDataRAM
. = ALIGN(4);
```

The .data section allocates the user variables into the on-chip RAM defined for the STR7.

2.13 C++ support

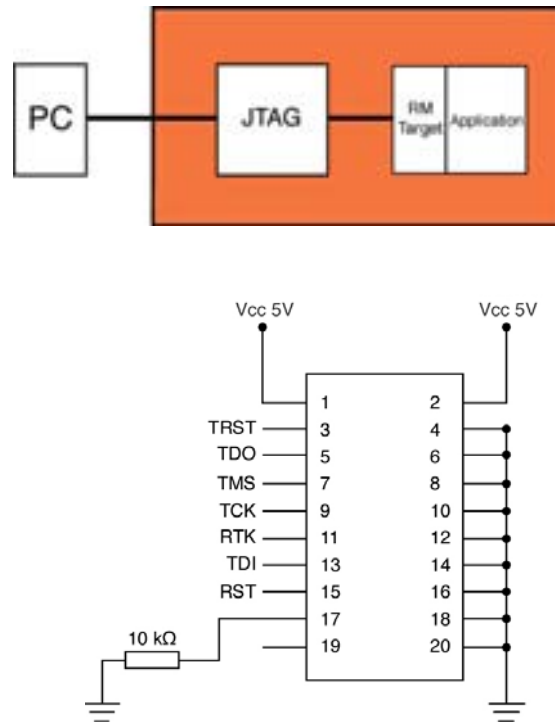
The GCC compiler is a C and C++ compiler, so while the examples in this book concentrate on using the C language, it is also possible to build a program using the C++ language and make use of its richer programming constructs. This is really outside the scope of this book, but an example is included here to get you started if you want to use the C++ language.

Exercise 5: C++

This example demonstrates how to define a C++ project with the GCC compiler and then builds and debugs it within the HiTOP IDE

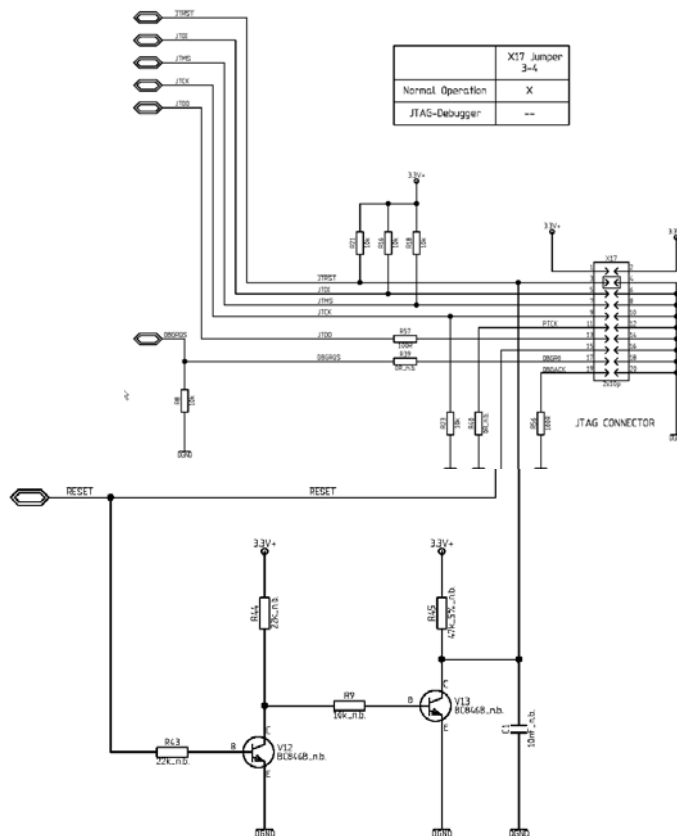
2.14 Hardware Debugging Tools

ST Microelectronics have designed the STR7 to have the maximum on-chip debug support. There are several levels of support. The simplest is a JTAG debug port. This port allows you to connect to the STR730 from the PC for a debug session. The JTAG interface allows you to have basic run control of the chip. That is you can single step lines of code, run halt and set breakpoints and also view variables and memory locations once the code is halted.



The JTAG connects to the target through a special debug port, which must be added to your target hardware. The JTAG socket has a standard layout which is common to all ARM7-based devices. The JTAG is brought out of the STR7 by seven dedicated debug pins. This means that the debug port does not compromise the GPIO ports or the on chip peripheral inputs and outputs.

It is also worth noting that the even pins on the JTAG connector are mostly ground connections. This means that if you accidentally plug your JTAG debugger in back to front, you will pull all of the signal lines to ground and possibly damage the Tantino. For this reason the evaluation board is fitted with a polarised socket to prevent this mishap. If you are designing your own hardware, it is recommended that you fit the same style of socket rather than just a bare header.



2.15 Important

As mentioned above, the JTAG port is a simple serial debug connection to the ARM7 device. It is very important to understand its behaviour during reset. If the ARM7 CPU is reset, all of the peripherals including the JTAG are reset. When this happens the Tantino debugger loses control of the chip and has to re-establish control once the STR7 device comes out of reset. This will take a finite number of clock cycles. While this is happening, any code that is on the chip will be run as normal. Once the Tantino gets back control of the chip, it performs a soft reset by forcing the PC back to address zero. However the on-chip peripherals are no longer in the reset condition, i.e. peripherals will be initialised, interrupt enabled etc. You must bear this in mind if the application you are developing could be adversely affected by this. A simple solution is to place a simple delay loop in the startup code, or at the beginning of main(). After a reset occurs, the CPU will be trapped in this loop until the Tantino regains control of the chip and none of the application code will have run, leaving the STR7 in its initialised condition. A macro is included in the startup code to provide this delay:

```
.macro StartupDelay delay_value

    ldr    R1, =\delay_value
    ldr    R2, =0
__StartDelay:
    sub    R1, R1, #1
    cmp    R1, R2
    bhi    __StartDelay

.endm
```

This macro is called at the entry point to your application. It is only required for debugging and should be removed on the released version of the code.

```
ResetEntry:
    StartupDelay 300000
```


2.16 Summary

So, by the end of this section you should be able to set up a project with StartEasy and HiTOP, select the compiler options and the variant you want to use, configure the startup code, be able to interwork the ARM and THUMB instruction sets, access the STR7 peripherals and write C functions to handle exceptions.

With this grounding we can now have a look at the STR7 system peripherals.

3 Chapter 3: System Peripherals

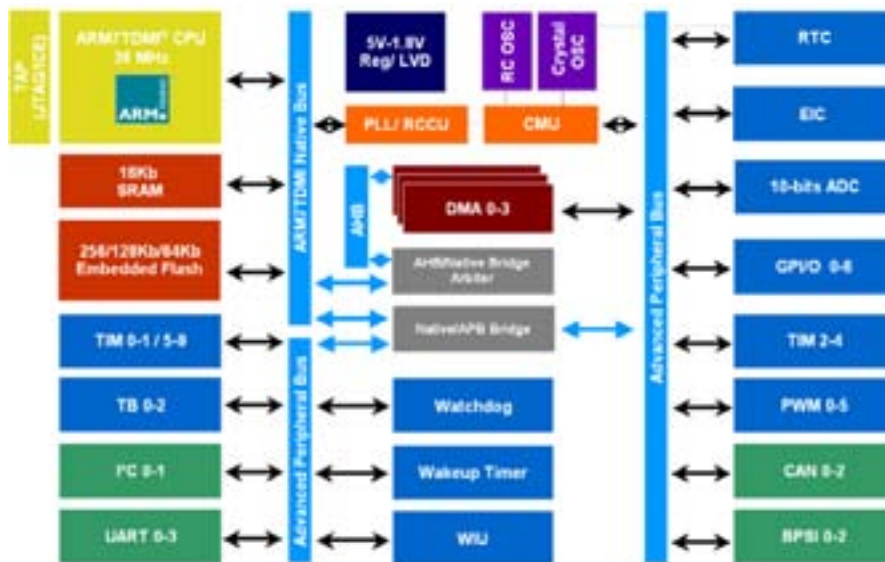
3.1 Outline

Now that we have some familiarity with the ARM7 core and the necessary development tools, we can begin to look at the STR73x device itself. In this section we will concentrate on the system peripherals, that is to say the features that are used to control the performance and functional elements of the device. This includes the on-chip FLASH and SRAM memory, clock and power control subsystem and the on-chip DMA control units. Finally we will take a look at the STR7 interrupt structure and the advanced support provided by ST to supplement the two CPU interrupt lines IRQ and FIQ. This is a key chapter in understanding how ST have integrated the ARM7 CPU into a standard microcontroller architecture.

Part number	Program memory Flash (KB)	RAM (KB)	A/D inputs	Timer functions		Serial Interface	I/Os (high current)	Packages	Supply voltage	Special features
				16-bit (IC/OC/PWM)	Others					
STR731FV0	64	16	12x10-bit	15 (12/12/12)	WDG, RTC	3xSPIs/ 2xI ² Cs/ 4xUARTs	72 (0)	TQFP100	4.5 to 5.5V	3xCANs, 16xDMA, on-chip RC oscillator
STR736FV0	64	16	12x10-bit	15 (12/12/12)			72 (0)	TQFP100		16xDMA, on-chip RC oscillator
STR730FZ1	128	16	16x10-bit	19 (20/20/16)			112 (0)	TQFP144/LFBGA144		3xCANs, 16xDMA, on-chip RC oscillator
STR731FV1	128	16	12x10-bit	15 (12/12/12)			72 (0)	TQFP100		3xCANs, 16xDMA, on-chip RC oscillator
STR735FZ1	128	16	16x10-bit	19 (20/20/16)			112 (0)	TQFP144/LFBGA144		16xDMA, on-chip RC oscillator
STR736FV1	128	16	12x10-bit	15 (12/12/12)			72 (0)	TQFP100		16xDMA, on-chip RC oscillator
STR730FZ2	256	16	16x10-bit	19 (20/20/16)			112 (0)	TQFP144/LFBGA144		3xCANs, 16xDMA, on-chip RC oscillator
STR731FV2	256	16	12x10-bit	15 (12/12/12)			72 (0)	TQFP100		3xCANs, 16xDMA, on-chip RC oscillator
STR735FZ2	256	16	16x10-bit	19 (20/20/16)			112 (0)	TQFP144/LFBGA144		16xDMA, on-chip RC oscillator
STR736FV2	256	16	12x10-bit	15 (12/12/12)			72 (0)	TQFP100		16xDMA, on-chip RC oscillator

3.2 Bus Structure

To the programmer the memory of all STR7 devices is one contiguous 32 bit address range. However the device itself is made up of a number of buses. The ARM7 core is connected to the “Advanced High performance Bus” (AHB) which is a bus structure defined by ARM. As its name implies, this is a high-speed bus running at the same frequency as the ARM7 CPU. The on-chip FLASH and RAM memory are connected to the CPU via this bus, along with the power and clock control unit, and the DMA units. All the remaining peripherals are located on a second bus called the “Advanced Peripheral Bus” (APB). This bus is again a standard bus structure defined by ARM. The APB bus is connected to the AHB bus by a bridge. This bridge contains a clock divider and access protection registers. The clock divider allows the APB busses to run at lower speeds than the main AHB bus. This is useful if you want to run the CPU at full speed but clock the peripherals at a slower speed to conserve power. Access to an unused region of memory within the APB region will cause the APB bridges to force a memory abort interrupt on the ARM7 CPU.

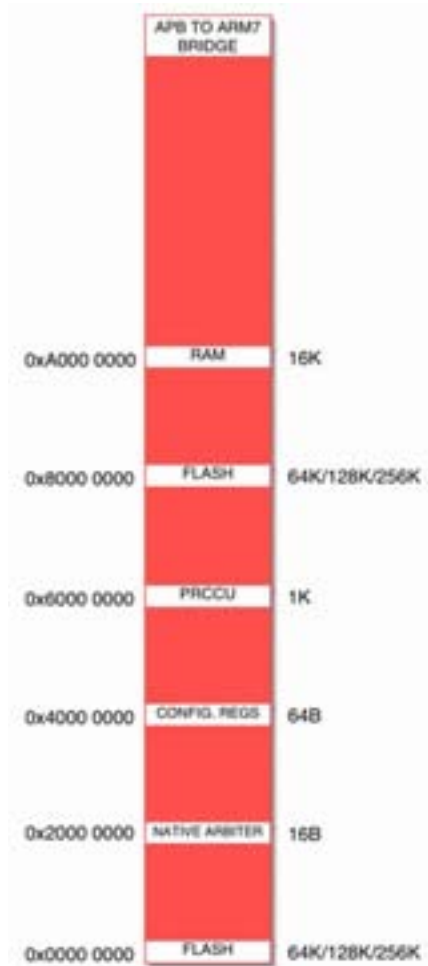


The STR7 has two internal busses. A high speed bus to connect the CPU with the FLASH and RAM and a peripheral bus. The busses are joined by a bridge which allows access to the user peripherals. The DMA units have their own dedicated bridge.

The STR730 has four DMA units which can transfer data between peripherals and memory, peripheral to peripheral or memory to memory. Although the DMA units are located on the high speed bus they have a dedicated bridge to the peripheral bus. When a DMA transfer is in operation it locks the bus preventing the CPU from executing instructions. However using the DMA units to transfer data is much faster by about a factor of four than software running on the CPU.

3.3 Memory Map

Despite the internal bus structure the STR7 has a completely linear memory map with the general layout shown below.



Despite having several separate internal busses, the STR7 memory map is completely linear and is defined by the CPU's 4Gbyte address space.

The on-chip RAM is located at 0xA0000000 and is 16K in size. The on-chip FLASH is at 0x80000000, with up to 256K of memory. Unlike the STR71x devices, the STR73x CPUs are true single-chip microcontrollers with no external bus. The main system peripherals are the CONFIG registers, located at 0x40000000 which control the boot mode of the STR730. Above these is the Power and Clock Control Unit at 0x60000000. The APB bus appears at 0xFFFF0000 to 0xFFFFFFFF. Each peripheral on the APB bus is allocated a 4k address space, giving plenty of room to layout the Special function registers (SFR) of complex peripherals. Finally the Enhanced interrupt controller appears at the very top of the ARM7 address space. Although this unit is on the AHB bus, its registers are fixed at the top of the ARM7 address space. We will see the reason for this when we examine the STR7 interrupt structure.

EXT	1K
ADC	1K
CMU	1K
RTC	1K
DMA 0-3	1K
TIM 4	1K
TIM 3	1K
TIM 2	1K
BSPI 2	1K
BSPI 1	1K
BSPI 0	1K
GPIO 0-6	1K
PWM 0-5	1K
CAN 2	1K
CAN 1	1K
CAN 0	1K
APB BRIDGE 1 REGS	1K
reserved	1K
TIM 5-9	1K
TIM 1	1K
TIM 0	1K
WAKEUP/TIM	1K
WDOG	1K
UART 3	1K
UART 1	1K
UART 2	1K
UART 0	1K
TB 0-2	1K
reserved	1K
reserved	1K
reserved	1K
reserved	1K
PC 1	1K
PC 0	1K
APB BRIDGE 1 REGS	1K

The peripheral busses provide 1K address spaces for the on chip peripherals.

In the first chapter we saw that the ARM7 CPU has its exception table fixed at 0x00000000 and that reset will cause the CPU to start executing from 0x00000000. The STR7 does not have any fixed memory at this address, however any memory resources (on-chip FLASH, RAM or system memory) may remapped to appear at 0x00000000. The type of memory mapped at zero is selected by two external processor pins, M1 and M0. This allows us to boot from on-chip FLASH or a factory programmed bootloader. During development, we can debug out of RAM, removing the need to continually burn the FLASH memory. The bootloader is discussed later but it is stored in a block of system memory and can be used to download a user bootstrap program into the on chip RAM via the serial port. Also these memory regions may be remapped on-the-fly, so it is possible to copy the interrupt vector table and interrupt routines into RAM and then map the RAM to zero. This allows us to run code from RAM at zero wait-states for very fast interrupt handling. Finally whenever any of the memory regions are remapped, they appear both at 0x00000000 and at their original locations. The boot configuration can be read via configuration register 0 and the remap bit in the same register allows the on-chip RAM to be mirrored at 0x00000000, replacing the FLASH memory which can still be read at its native location of 0x80000000.

M1	M0	Mode	Memory Mapping	Note
0	0	User1	B0F0 @ 0x0	All sectors visible except Sys Mem
0	1	User2	B0F0 @ 0x0	All sectors visible except Sys Mem & B0F1
1	0	SysMem	SysMem @ 0x0	-

The configuration of the external boot pins selects which memory resource is mirrored at the reset vector. The BOOT control register allows this to be changed dynamically.

The STR7 development board has two jumpers that allow you to select the state of these pins. The most common error when downloading a first program is simply having these jumpers incorrectly set.

3.4 On-Chip FLASH

The STR730 has a block of User FLASH located at 0x80000000. Depending on the STR730 variant, the size of the User FLASH can be 64K 128K or 256K. The FLASH memory is organised in one contiguous block but for programming purposes it is arranged as a number of sectors. These sectors vary in size from 8K up to 64K. The smaller sectors are located towards the bottom of memory and are intended to hold BIOS or bootloader code plus code constants that may be updated from time to time.



The STR730 has up to 256K of on chip FLASH memory. An additional 8K of system FLASH memory is factory programmed with a bootloader which can be used for production programming or field upgrades.

All of the FLASH sectors can be protected against erasure. Small 8K sectors are provided to hold permanent boot code.

As we saw in Chapter 2, the STR7 can be programmed in circuit using the JTAG debug port and this is the fastest method of programming the on-chip FLASH. However the STR730 contains a factory-programmed bootloader which provides an integral method of programming the on-chip FLASH.

3.5 The System Memory Mode Bootloader

In addition to the user FLASH memory the STR730 contains an 8K block of FLASH memory that is located at 0x10C0000 and is factory programmed with a bootloader program. When system mode is selected via the M0 and M1 pins, this block of memory is mirrored to 0x00000000 and after reset the STR730 will enter into the bootloader program.

The bootloader monitors the UART0 RX pin and waits for a terminal program to send the value “0x00” as a serial character, with the format eight data bits, no parity, one stop bit. The bootloader can then determine the baud rate by measuring the time between the start and the stop bits. Once the autobaud rate detection has been performed, the bootloader will expect a further 128 bytes of data which it will store in the on-chip RAM, starting from 0xA0000000. Once the 128 bytes have been downloaded, the bootloader will force a jump to 0xA0000000 and start to execute the downloaded data as a program. This program may be used to download a larger application into RAM, or may be a simple FLASH programming algorithm that is used to update the on-chip FLASH memory. Since the main delay in programming the on-chip FLASH is downloading the code into the chip via the serial port, it is possible to download the loader program using the bootloader and UART, and then load the application code by a faster serial interface such as the SPI port or even a custom parallel interface. For volume production programming, you have the option to use the JTAG port or the bootloader combined with a fast serial port.

Once the STR730 has been programmed with your application code, the bootloader is still resident in the system memory and could potentially be used to copy the on-chip FLASH memory. If you are concerned about the security of your code, part of the production process should be to erase the on-chip bootloader and disable the JTAG debug interface.

Exercise 6: Bootloader

This exercise demonstrates using the bootloader to download a program image into the on chip SRAM and then passing control to this application. This is the basis of a boot-loaded FLASH programmer

3.5.1 User FLASH Programming

The on-chip FLASH is arranged as word-wide (32-bit) memory. Each sector can be erased independently the FLASH can be written word- or double-word wide. Once written, each sector can be protected against an erase. It is also possible to disable the JTAG port to prevent the internal FLASH from being read. As the FLASH memory is arranged in one bank, it is necessary to load any FLASH programming algorithm into the on-chip RAM where it can run and reprogram the FLASH. The actual programming of the FLASH memory is controlled by the FLASH program erase controller (FPEC). The FPEC is a common unit between the STR710 and the STR730. Consequently it has some programming fields that refer to a second FLASH bank that is only available on the STR710 variants. In the diagrams below these are shown in the shaded boxes.



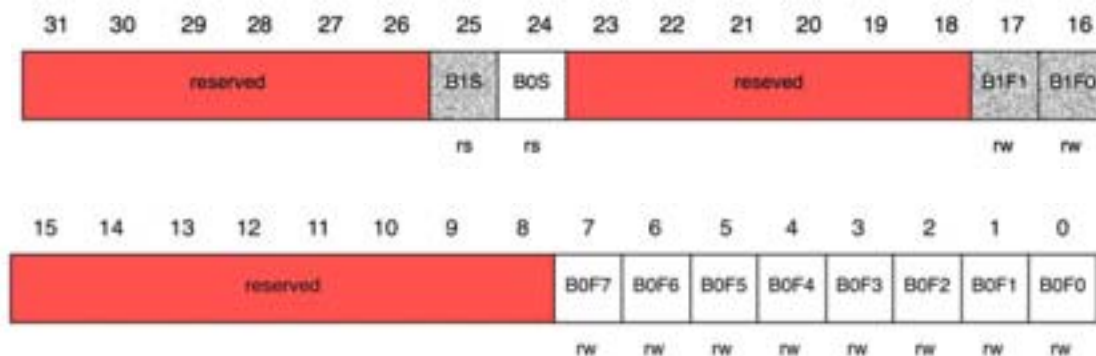
The FLASH program and erase controller (FPEC) allows in application programming of both banks of the STR7 on chip FLASH memory

Prior to programming the FLASH memory you must first perform an erase operation. Within the STR7 you can erase an individual sector or the entire FLASH.



FPEC Control register 0. This register allows you to easily perform a FLASH sector erase and either a 32 or 64 bit FLASH programming operation.

To perform the erase operation, set the sector erase (SER) bit in control register 0 then select the sectors to be erased in Control register 1. Finally start the erase operation by setting the WMS bit in control register 0. To erase an entire flash memory you must set all the sector bits.



FPEC Control register 1. The second FPEC control register allows you to select the FLASH sector or sectors you wish to erase prior to starting the erase operation in control register 1.

```
void erase_FLASH (unsigned int sector)
{
    FLASH_CR0 |= 0x08000000;    // Set Sector Erase bit
    FLASH_CR1 |= sector;        // Select b1 sector 0
    FLASH_CR0 |= 0x80000000;    // Start the erase operation
    while (FLASH_CR0 & 0x00000010)    // wait for the lock bit to clear
    {
        ;
    }

    while (FLASH_CR0 & 0x00000004)    //wait for the busy bit to clear
    {
        ;
    }
}
```

Programming a FLASH cell is a similar simple operation. First you must select whether you are going to program a word or double word by either of the WPG or DWPG bits in FLASH control register0. Next write the word address of the location you want to program into the FLASH address register and the data to be programmed into the FLASH data register. Then to start the programming process, set the WMS bit in control register 0.

```
void write_FLASH_32 (unsigned int address, unsigned int data)
{
    FLASH_CR0 |= 0x20000000;    //Select word wide programming
    FLASH_AR = address;
    FLASH_DR0 = data;
    FLASH_CR0 |= 0x80000000;    //Start the FEPEC writing
    while (FLASH_CR0 & 0x00000010)    //wait for the lock bit to clear
    {
        ;
    }

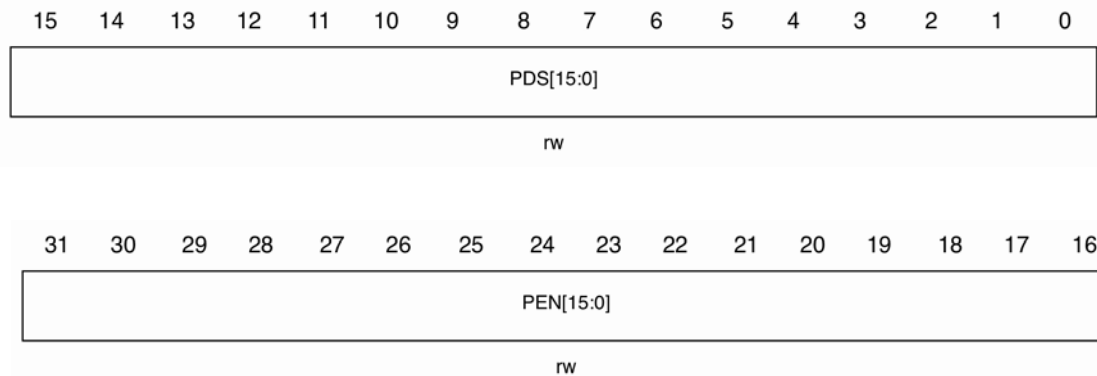
    while (FLASH_CR0 & 0x00000004)    //wait for the busy bit to clear
    {
        ;
    }
}
```

During either the FLASH erase or write operations, the busy bit in control register 0 will be set and then cleared when the hardware has finished the operation. While the write or erase operation is in progress, it is not possible to access the FLASH. So if you are updating one sector it is not possible to execute code in another sector. Since

this may cause problems, especially during sector erase operations which take a relatively long time, it is possible to suspend a FLASH programming operation then access the FLASH memory to execute some code and finally resume programming of the FLASH memory. To suspend FLASH programming operations, simply set the suspend bit in FLASH control register 0. You can then access the FLASH bank when the busy and lock bits in the same register have been reset by the FPEC. Once you have finished accessing the FLASH bank you can resume the FLASH programming operation by clearing the suspend bit and then reprogramming FLASH control register 0 with the same values used to start the operation. Finally set the WMS bit.

3.5.2 FLASH Memory Protection

The STR7 has two memory protection methods for the FLASH memory. It is possible to write protect sectors within either bank to protect against accidental write operations. You may also set debug protection, which is intended to prevent software piracy.



Debug protection is set by clearing the DBGP bit in the FLASH access protection register. This disables the JTAG preventing any development tool from connecting to the STR7. The JTAG interface may be re-enabled by setting this bit back to one. Generally you should not attempt to set and clear this bit during development, just clear it on production boards if you want to protect the FLASH. Since this register is a non-volatile you must write and erase it as if it were a standard FLASH cell with code running from the on-chip RAM.

3.5.3 Code Enable And Disable Debug Protection

Setting the DBGP bit directly will only cause a temporary unprotect. The JTAG will again be disabled after the next reset. The FLASH_nvaccess protection register allows you to permanently re-enable the JTAG peripheral. The register is split into two half words, both set to 0xFFFF in a new device. The bits in the lower half word are the protection disable bits PDS 15-0. The upper half word contains 16 protection enable bits PEN15-0. Each of these bits is a fuse so that once it has been burnt to zero, it cannot be reset back to one. So starting from the lowest order PDS bit we can write PDS bit 0 to zero and disable protection. Then protection may be re-enabled by writing its matching PEN bit to zero. We can repeat this cycle using the next lowest order pair of bits to disable and re-enable protection until all 16 pairs of fuses have been used.

You may also set write protection on the STR7 FLASH sectors. The FLASH non-volatile write protection register (FLASH_NVWPAR) contains a bit for each sector in bank zero and bank one. By writing these bits to zero you assert write protection on a given bank. If you wish to update a protected bank you must first write the protection bit back to one before accessing the sector with a write operation. This unprotection method is only a temporary unprotect since the sector will revert to being a protected sector as soon as the processor is reset.

Exercise 7: FLASH Programming

This exercise demonstrates programming a FLASH sector with program constants

3.5.4 Low Power FLASH Modes

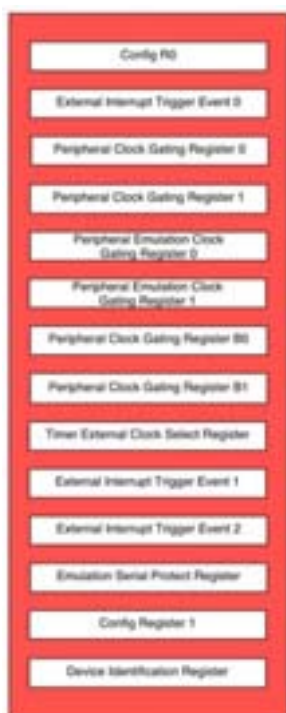
When the STR7 enters its low power mode (“Low power wait for interrupt” - see section 3.17), the FLASH memory can be configured to power down so that it consumes a minimal standby current. Once a peripheral generates an interrupt and wakes up the CPU there is an additional delay to allow the FLASH memory to re-charge before processing can resume. If the low power FLASH mode is not enabled, the FLASH memory power consumption is still automatically reduced when the STR730 enters its low power mode, it just does not reach its minimum consumption. However in this case the FLASH cells are still charged, so processing can resume immediately the CPU wakes up.

3.5.5 System Peripherals

The STR730 contains three blocks of registers that may be configured to control the overall performance of the device after it comes out of reset. These blocks are the configuration registers, the power reset and clock control registers and the clock monitor unit. They are to a certain extent interrelated and in the next section we will have a look at each of these peripherals in turn to see how they effect the operation of the STR730.

3.5.6 Configuration Registers

The system configuration registers are a group of 14 registers that control and report on the fundamental operation of the STR730.



The configuration registers enable the peripheral clocks. You must ensure each peripheral clock is enabled before a given peripheral can be used.

The configuration registers are also used to setup the external interrupts

The configuration registers may be split into four groups; system configuration, external interrupt configuration, peripheral clock configuration and emulation configuration.



Configuration Register 0 reports the device boot mode and debug state of the STR730. This register is also used to remap the ram and enable DMA support

The first two registers in this block are used to perform the system configuration functions. Configuration register 0 contains four read-only flags SYS, USER1, USER2 and JTBT that report the boot mode as either system, user 1 or user 2 (see the memory map section). The JTBT reports that the STR730 has booted in system mode and a JTAG debugger has downloaded a FLASH loaded program into the on-chip RAM. The DMA flags are used to configure two of the sixteen DMA data streams. These are discussed in more detail in the DMA section at the end of this chapter. Finally the REMAP bit is used to mirror the on-chip RAM starting at 0x00000000. This allows you to download code via the JTAG into the RAM, removing the need to repeatedly program the on-chip FLASH. This is useful in that this method of downloading is fastest and your JTAG tool can set multiple software breakpoints in the RAM rather than be limited to the two hardware breakpoints in the JTAG module. On the downside, your code is occupying the valuable on-chip RAM, limiting the amount of data space available to your application. Normally remapping the RAM to 0x00000000 is used in development. However the on-chip RAM has a much faster access time than the FLASH so code will execute fastest when loaded into the RAM. If you need a low interrupt latency, it would be possible to copy the vector table and interrupt routines into the RAM in and then remap the RAM to 0x00000000 so that the fastest executable memory is used to serve the STR730 interrupts



Configuration register 1 controls the FLASH power on delay, low power regulator and wake up configuration

The second configuration register allows you to configure a power-on delay for the on-chip FLASH. This is necessary if you have entered the wait-for-interrupt low power mode and disabled the main on-chip regulator. The power-on delay allows the STR730 to exit from the low power mode and the power supply from the regulator to stabilise before the CPU is allowed to access the FLASH. The power-on delay field consists of four bits that define the delay in clock cycles, as given below.

Power on delay = $FLOD \times 64 \times TCK2$ if the RC oscillator $TCK = 2MHz$

Power on delay = $FLOD \times TCK32$ if the RC oscillator = 32KHz

The main oscillator stabilisation time is given as 5msec in the data sheet hence suitable values for FLOD are;

$FLOD = 0x9C$ $TCK = 2MHz$

$FLOD = 0xA3$ $TCK = 32KHz$

The STR730 has an additional STOP low power mode, which has a dedicated wake-up unit. The wake-up unit has 32 input lines that are normally connected to GPIO pins that can be used trigger a wake-up of the STR730. The WUP0S bit in configuration register 1 is used to configure wake-up line 0. This input line to the wake-up unit may be connected to a dedicated wake-up timer or a dedicated WUP0 pin. When the STR730 enters a power

down mode, an additional low-power voltage regulator is used to maintain the peripherals, CPU and RAM contents. Control register 1 can control the maximum current output from this regulator by setting the bits in the LPVRCC field. The current output from the low power voltage regulator is set to 4mA after reset but it is possible to disable each of the user peripherals so that the power requirement can be lowered to 2mA.

```
CFG_R0 = 0x00000030;
CFG_R1 = 0x000000**
```

The next three registers in the configuration block are used to configure the external interrupts. These will be discussed later when we look at the STR73x interrupt structure but essentially the three registers define the trigger event for the external interrupt line. This may be edge-sensitive (rising or falling edge or both) or level sensitive.

The next seven registers are used to manage the peripheral clocks. With these registers you can enable and disable the clock to each peripheral. These registers also give you the option to freeze the peripheral in its current state or the stop the clock and reset the peripheral. If the peripheral is frozen the clock may simply be restarted and the peripheral will pick up from where it left off. Intelligent use of these registers will help you reduce both the overall power consumption of the STR730 and its electromagnetic emissions. The peripheral clock management registers also allow you to select the clock source for the ten general purpose timers. The clock source may be an external clock source or may come directly from the main external oscillator (via a prescaler) "Fext".

3.5.7 Important Note

After reset, the peripheral clock gating registers are initialised to disable all the peripheral clocks. You must first enable the clock for any peripheral you intend to use before you access any register within the peripheral.

The two peripheral clock gating registers contain a series of bits which are used to enable or disable the clock to a specific peripheral. If the bit is set the peripheral receives the clock signal. If it is reset the clock is switched off and the peripheral is held in reset. The peripheral clock gating B registers contain the same series of bits and are used to mask the reset function. If the bit for a given peripheral is set, when the clock gating registers disable the clock, the peripheral will be frozen rather than reset.

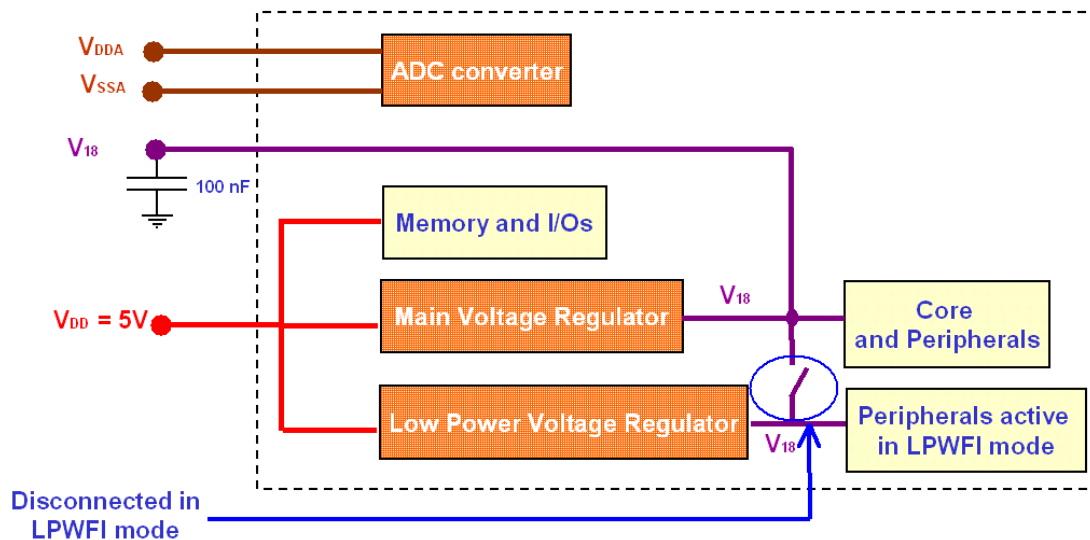
```
CFG_PCG0 = 0x7BFFFFDF7; // Enable all peripheral clocks
CFG_PCG1 = 0x60FF73F9;
CFG_PCGB0 = 0x00000000;
CFG_PCGB1 = 0x00000000;
CFG_TIMSR = 0x00000000; // Select the Fext clock source for each timer
```

Finally the timing external clock select register defines a bit for each of the ten timers. After reset, each of these bits is set to zero and the timers will take their clock source from an external clock via the input capture A pin (ICAPA). If you set these bits to one the timers will be clocked directly from the main oscillator, "Fext".

In addition to configuring how the STR730 will run for your application code, the configuration registers have some interesting features which can be used to extend the JTAG debug features. The peripheral emulation clock gating registers are used to control the behaviour of each peripheral clock when the CPU enters JTAG emulation mode. Like the peripheral clock gating registers discussed above, each peripheral has a dedicated bit, located within the 64 bits of these two registers. If this bit is cleared the peripheral clocks will be halted, when the chip is under JTAG control. This keeps activity of the peripherals "cycle-accurate" to the code running on the CPU. Finally the Emulation serial protection register provides additional debug support for the buffered SPI and UARTS. Both of these peripherals have receive FIFO's. If these FIFO's are read by the debugger the data is removed from the FIFO buffer and will be lost to the application software. If serial protection is enabled for these peripherals, the FIFO receive buffers can be read when the STR730 is in emulation mode, without removing the data.

3.5.8 STR7 Power supplies

The STR730 runs from a single 5V supply. This 5V is used to power the on-chip FLASH and RAM memories and to supply the IO pins, which are all 5V tolerant. The on-chip peripherals and the CPU are run from an internal 1.8V supply. The external V18 pin is not a supply pin but should be connected to ground via a capacitor in the range 100nF – 10 uF.



On chip regulators are used to derive the 1.8V core and peripheral supply from an external 5V source.

During normal operation this 1.8V is generated from an on-chip voltage regulator. When the STR730 enters its low power modes the main voltage regulator is disabled and the 1.8V supply is generated from an additional on-chip low power regulator. The low power regulator maintains the CPU registers and any peripherals that are running while the CPU is halted. The voltage regulator register in the Power reset and clock control unit is used to configure switching between the main and low power voltage regulators.



The on chip power supplies can be configured with the Voltage regulator register in the PRCCU.

This register allows the main regulator to be disabled by setting the VROFF_REG bit so that the STR730 is running from the low-power regulator. If the low-power regulator is selected, the PLL will automatically be disabled and the STR730 will run from the external oscillator divided by 16. If the main voltage regulator is used the VRLPW bit allows you to select which regulator is used in low power mode. This may be either the main regulator or the low power regulator and effects the power consumption and wake up time of the STR730. Depending on the configuration of the chip (operating speed and number of peripherals enabled) the maximum output of the low -power regulator can also be defined in the configuration registers (Configuration Register R1)

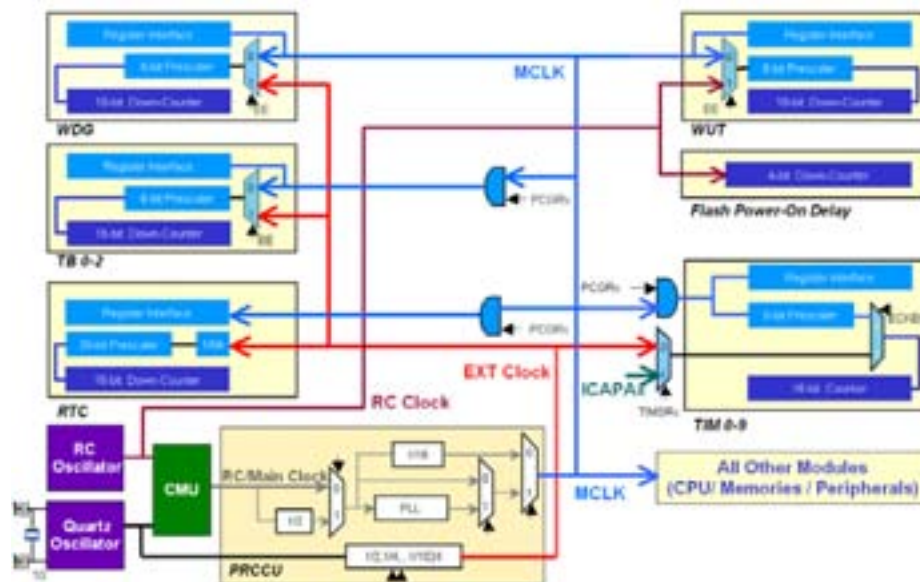
3.5.9 Low Voltage Protection

The STR730 has two low-voltage detect circuits built into its power supply. These are used to protect the CPU and peripheral supply plus the FLASH memory supply. If either supply falls below a safe operating level the chip

will be forced into reset until the power supply resumes its operating level. If the STR730 has reset because of a low power error, the LDV_INT bit will be set in the clock flag register of the power reset and clock control unit.

3.6 Clock Structure

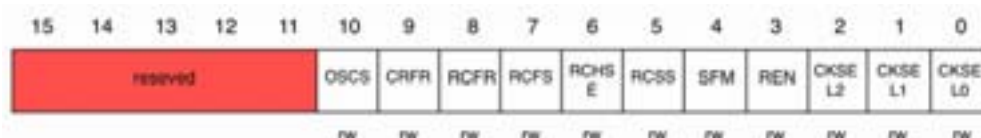
Before we look at these operating modes it is necessary to understand the clock structure of the STR7 and the different clocking domains that exist within the STR730.



The STR730 has several clock domains that are derived from the two external oscillators and the on chip RC oscillator.

The STR73x has two oscillators. One of these is an external crystal oscillator which must be between 4-8 MHz and the other is an internal backup oscillator, which is capable of running at either 2MHz or 32.768KHz. Both of these clocks are managed by the Clock Monitor Unit CMU. The CMU selects either of these clock sources as the input to the power reset and clock control unit (PRCCU). The clock control unit within the PRCCU derives the main system clock. The system clock is called Mclk and it is used to drive the CPU, memory and peripherals. Mclk is applied directly to the CPU and memory but passes through a further control block before it reaches the peripherals. Since the peripheral control block and the clock control unit may be used to dynamically change the frequency of Mclk, the timers, real time clock and watchdog are clocked by a separate clock called "Fext", which is derived directly from the main oscillator. This allows your application software to adjust the STR73x system clock without disturbing the timebase used by the various timer blocks. The backup oscillator will run at 2MHz when the STR730 comes out of reset. If you want the backup oscillator to run at 32KHz from reset this can be achieved by adding a 1.3Mohm resistor to the V_{bias} pin. Alternatively if the resistor is not present the STR730 will boot at 2MHz and may be adjusted to 32KHz by setting the RCFR bit in the control register

This register also controls the behaviour of the backup oscillator in the low power modes and we will look at this a little later.

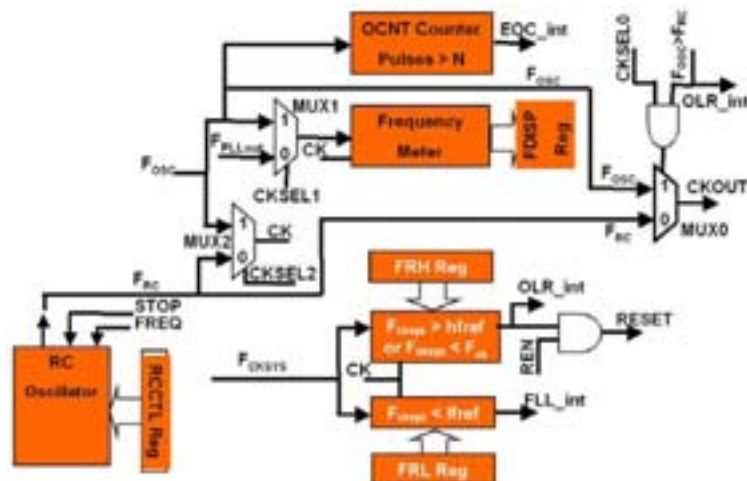


The STR730 builds on the low power consumption of the ARM7 CPU to provide additional power down modes and a flexible clocking scheme.

3.7 Setting Up The Clocks

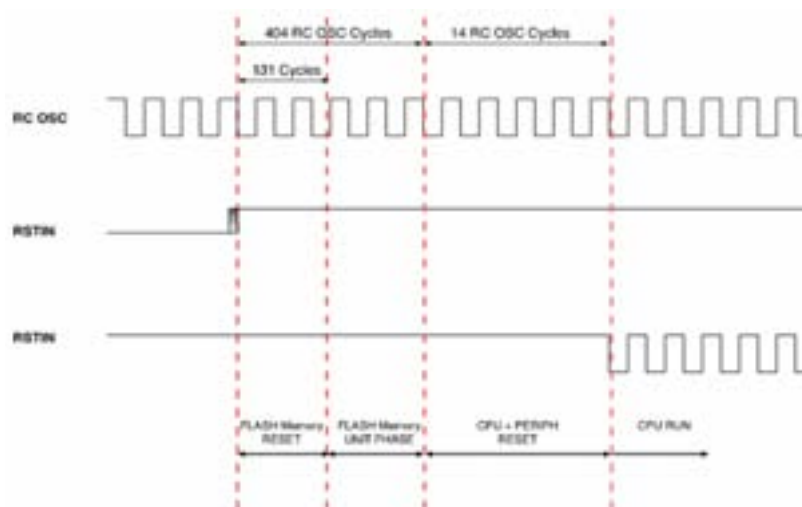
The two system clock oscillators available to the STR7 are controlled by the Clock Monitor Unit CMU. The CMU selects an oscillator to be an input to the power reset and clock control unit, which in turn provides the main system clock MCLK. The two oscillator sources are the external oscillator or the main clock and the on-chip oscillator called the “backup oscillator”.

After reset or an exit from a power down mode the CMU will select the backup oscillator as the STR73x clock source while the external main oscillator stabilises. Once the external crystal is stable, the CMU will switch to the main oscillator and The STR7 can run up to its maximum frequency of 38MHz. Once the main oscillator is providing the system clock, the CMU will continue to monitor it and will switch back to the backup oscillator if the main oscillator fails. This approach not only boosts the performance of the STR73x as it leaves reset but also helps save power as the CPU can start processing well before the main oscillator has stabilised. The same method is used when exiting the low power STOP mode. This ability to rapidly resume processing after a reset of exit from STOP mode is a big advantage for low power applications.



The Clock Monitor Unit (CMU) controls selection of the system oscillator source. It also contains a frequency meter to monitor the system clock.

When power is applied to the STR730 it will pass through a reset phase before it begins processing instructions. During the reset phase the backup oscillator will automatically be selected. The STR730 takes a total of 418 cycles or about 209μsec to initialise and start processing instructions from the reset vector. After the initialisation phase the STR7 will start to run your application code using the 2MHz backup oscillator.



The CMU is used to select the clock source as either the on chip RC oscillator or the external oscillator.

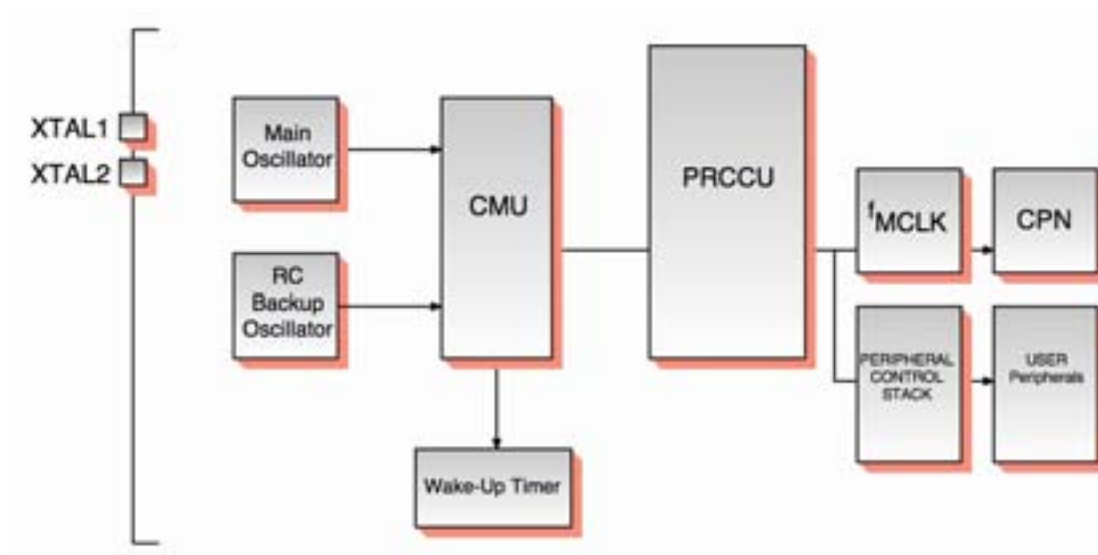
From reset or power down the STR730 takes around 209 usec to start processing instructions from the reset vector. This start-up time is achieved by using the on chip RC oscillator while the external crystal is still stabilising. Within the CMU is a dedicated oscillator counter which provides a time delay for the main oscillator to stabilise. The oscillator counter counts pulses from the main oscillator starting from the external reset signal. At the end of this time delay, which will be about 5msec for an 8 MHz crystal, the CMU will assume that the external oscillator has stabilised.



The control register in the CMU is used to configure the STR730 oscillators during running operation and low power modes

Once the oscillator counter reaches its end of count, an interrupt may be generated and the EOC bit will be set in the interrupt status register. When this event occurs, the application code can exchange the backup oscillator for the main oscillator by setting the CLKsel2 bit in the control register. Since the CMU registers effect the fundamental performance of the STR730 they are protected from accidental writes. Before any register within the CMU can be written to, a "key " value must be written to the Write enable register. The key value that unlocks access to the registers is 0x50FA followed by 0xAF05.

```
CMU_WE      = 0x50FA; // Write the key values to unlock the registers
CMU_WE      = 0xAF05;
CMU_CTRL    = 0x00000185; // Enable the external oscillator as the
                          // clock source
```



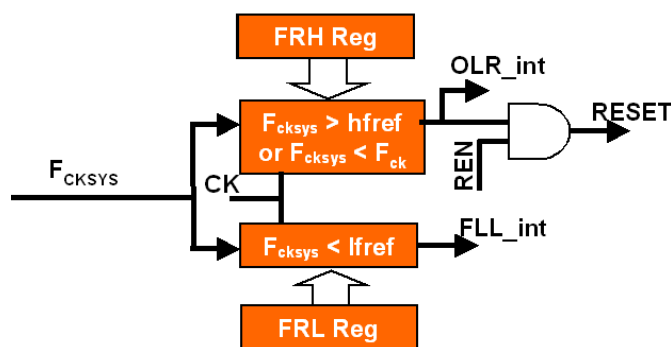
The CMU selects the external or on chip oscillator. This clock source is conditioned by the Power reset and clock control unit (PRCCU) to provide the CPU and peripheral clocks.

The clock output of the CMU is fed into the Power reset and clock control unit which conditions the oscillator clock and provides the system clock MCLK. The clock control unit within the PRCCU contains a Phase Locked Loop (PLL) that can multiply the main or backup oscillator up to the maximum system clock frequency of 36 MHz. The

clock control unit also contains alternate clock paths which may be selected to divide the oscillator frequency by 2, 16 or 32. As a rule, the divide by 2 path should be always be selected because this also acts as a filter and ensures an even mark space ratio for the system clock. The frequency of MCLK may be dynamically changed within the clock control unit by your application software. This is typically done to manage the power consumption of the STR730. The system clock is also used to provide the clock for each of the peripherals. Before it reaches each peripheral it is gated through a dedicated peripheral control block, which is used to enable/disable the clock for each peripheral. In the case of the timers it is possible to select the main oscillator or an external clock as the timer clock source via the Timer External Clock select register that is part of the configuration register block.

3.8 Clock Protection

As well as providing the back up oscillator the clock monitor unit also contains a frequency meter that can be used to monitor the external oscillator or the output of the PLL. If the monitored clock signal fails or exceeds a reference value the SRT730 can be reset or alternatively an interrupt may be generated. It is also possible to generate an interrupt if the monitored clock signal dips below a selected reference frequency. In this case an only an interrupt is generated.



The frequency meter in the CMU is able to provide an approximate frequency value for selected clock sources. Maximum and minimum operating limits can also be defined to provide an additional level of operating security.

The CKSEL1 bit in the control register allows you to select either the external oscillator or the PLL output as the clock source to be measured by the frequency meter. Once the clock source to be measured has been selected, setting the SFM bit will begin the frequency measurement. When a result is ready the SFM bit will be cleared and the approximate value of the measured frequency can be read in the frequency display register. The frequency held in the display register can be calculated with the following formula

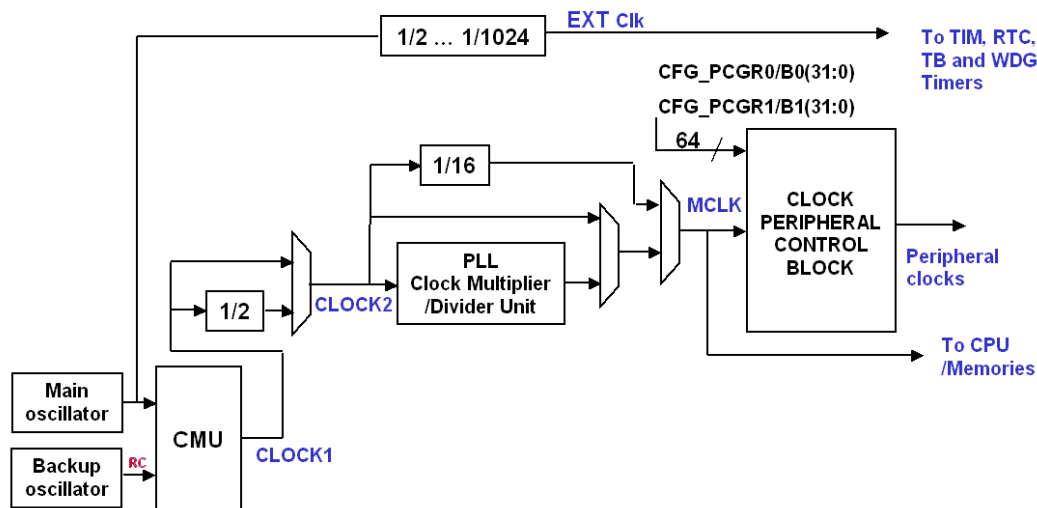
$$F_{in} = (F_{DISP}/16) \times MCLK$$

As well as displaying the clock frequencies the frequency meter can monitor the selected clock signals and trigger a reset or interrupt when the cross a pre-determined threshold. The upper frequency threshold is stored in the Frequency Reference High register and the lower threshold is held in the Frequency Reference Low register. The values to be stored in these registers to set the max and min limits may be calculated with the following formula

$$\text{Register value} = (F_{ref} \times MCLK)/16$$

3.9 Power, Reset and Clock Control Unit (PRCCU)

Once the clock source leaves the clock monitor unit it enters the Power Reset and Clock Control Unit. This unit contains a phase locked loop and several clock divides that allow you to dynamically select a wide range of clock frequencies, which can then be used to drive the CPU and peripherals.



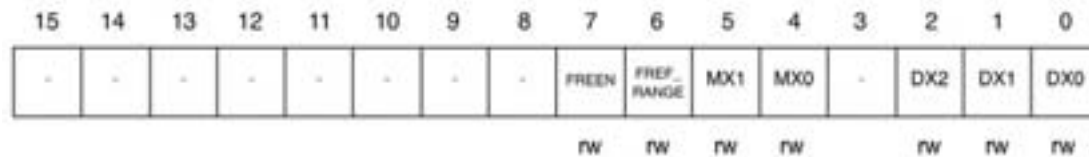
The Power reset and clock control unit contains a flexible clock scheme that allows an application to run the STR730 at full speed or run at low speed for power saving.

The main component of the clock module is used to derive the CPU AHB and APB (Peripheral) bus clock (MCLK). During normal operation you will want to derive a MCLK frequency of 36MHz. The evaluation board is fitted with an 8MHz crystal and the phase locked loop can be used to multiply this frequency to reach a suitable Mclk value. If the 8 MHz frequency was used as the input to the PLL the maximum value of Mclk that could be derived is 32MHz. However the external oscillator can be divided by two before it reaches the input of the PLL. If we use an input value of 4 MHz to the PLL then the maximum system clock of 36 MHz can be reached. This divider is controlled by bit 15 in the Clock Flag register. In addition to dividing the CK frequency by two, the divider also ensures an 50% duty cycle. So if you are using a low-cost oscillator, the divider will make the clock operation more stable.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV2	STOP_	CK2_	CKAF_	LOCK_	WKP_	LVD_		RTC_	WDG_	SOFT_		CK2_	CKAF_	LOCK_	CSU_
16_I	I	_J	_I	_J	RES	RES		ALARM	RES	RES		16	_ST		CKSEL
r/w	r/w	r/w	r/w	r/w	r	r		r	r	r		r/w	r	r	r/w

The Clock flag register contains the clock module status flags and also the clock selection configuration bits.

The PLL can operate from an external crystal as low as 1.5MHz and as high as 5MHz. The PLL is optimised to operate over two frequency ranges. The first range is between 1.5MHz and 3MHz and the second range is all frequencies above 3MHz, up to a maximum of 5MHz. Configuration of the PLL is made through the PLL Configuration register



The PLL configuration register sets the multiply and divide ration for the main PLL.

For a low frequency external crystal the Frequency Range bit must be cleared. The output frequency is determined by the multiplier and divider bits. The multiplier bits allow you to multiply up the crystal frequency by a factor of 12, 16, 20 or 24. The divider can be used to divide the resulting frequency by integer values between 1-7. Once the multiply and divider bits have been set the PLL will start and once it is stable, the lock bit in the clock flag register will be set. Before the PLL can be selected as the clock source we must also check that the CPU supply voltage is OK by checking the VR-OK bit in the Voltage Regulator Control register. In addition the PLL may be placed in a free running mode by setting the divider bits to one and setting the Free Running Enable bit (Freen). In this mode the PLL will generate a low frequency clock of 125KHz, 250KHz or 500KHz depending on the setting of the multiplier and frequency range bits. However you must not enter this mode when the PLL is being used as the system clock.

So at the output of the PRCCU (see fig. XXX clock structure) we have a choice of running from either external oscillators or the on-chip RC oscillator. The external oscillator frequency F_{ext} may be divided by 2, 16 or 32 as well as a high frequency and low frequency outputs from the PLL. These clock sources are selected in the clock flag register. After a power-on, the selected F_{ext} frequency is passed through to run the CPU as Mclk. The DIV2 bit enables the divide by 2 and CK2_16 bit enables the divide by 16. After a hardware reset the RC 2MHz oscillator is selected and the divide-by-two prescaler is enabled to give an Mclk of 1 MHz. If the PLL is enabled and stable it may be selected as the Mclk clock source by setting the CSU_SEL bit. The external watch crystal may be selected to provide Mclk by setting the Alternate function clock select bit (CKAF_SEL) in the Clock Control register.

Clock Signal	Description	Min	Max
F_{OSC}	Main oscillator	4 MHz	8 MHz
F_{RC}	Backup oscillator	Typical: 29KHz or 2.35MHz	
F_{EXT}	Real Time clock	3.9 KHz	4 MHz
F_{MCLK}	Main System Clock	-	36 MHz

So if the maximum frequency for MCLK = 36MHz and external crystal = 8MHz, the input to the PLL can be selected to be external crystal/2 = 4 MHz. So the PLL must multiply by 9 to give the maximum system clock. Since the multiplication factors for the PLL are 12, 16, 20 and 28 and the divide factors are integer values 1 – 7 the maximum value we can derive from the PLL is 32MHz

```
While(!(PRCCU_VRCTR & 0x00000004); // Wait until voltage regulator has stabilised
PRCCU_PLLCR = 0x000000F1           // Enable the PLL Fext x16 /2
While (!(PRCCU_CFQ & 0x00000002);  // Wait for PLL to lock
PRCCU_CFR |= 0x00000001;           // Select the PLL output as Mclk
```

Exercise 8: Clock Module

This exercise configures the PRCCU clock module to give a 32 MHz CPU clock and enable the peripheral clocks

3.9.1 Low Power Modes

We have seen how the STR7 has a very flexible clock generation unit, which can be used to conserve power by running peripherals or the ARM7 CPU at low frequencies during idle periods. However the STR7 has four specific power-down modes that greatly enhance its low power operation.

3.9.2 Slow Mode

As we have seen above, we can dynamically switch the clock frequency from a number of sources. By switching between the PLL and an alternate clock source we can reduce the STR7 power consumption to a few milliamps. To enter slow mode you can select either CLOCK2 or CLOCK2/16 with the CSU_CKSEL and CK2_16 bits in the configuration register. However once the clock source is changed the PLL will be disabled and must be restarted before you exit slow mode

```
RCCU_CFR          = 0x0000A800;          // Set Mclk to 0.250MHz
...

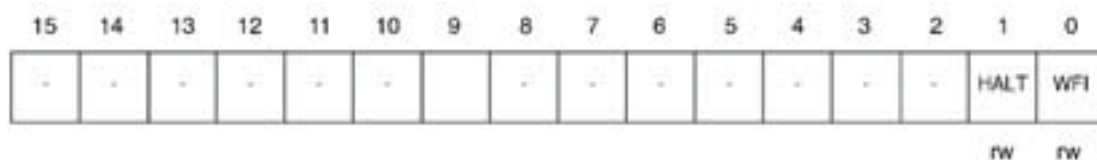
PPRCCU_PLLCR = 0x00000000;              // Clear the PLL register

While(!(PRCCU_VRCTR & 0x00000004);      // Re-select the PLL as the clock source
PRCCU_PLLCR = 0x000000F1

while (!(PRCCU_CFQ & 0x00000002);
PRCCU_CFR |= 0x00000001;
```

3.9.3 Wait For Interrupt

In wait for interrupt (WFI) mode, the CPU is halted but the peripherals continue to run. WFI mode is entered by clearing the WFI bit in the system mode register. In this mode Mclk is selected as Fosc/32.



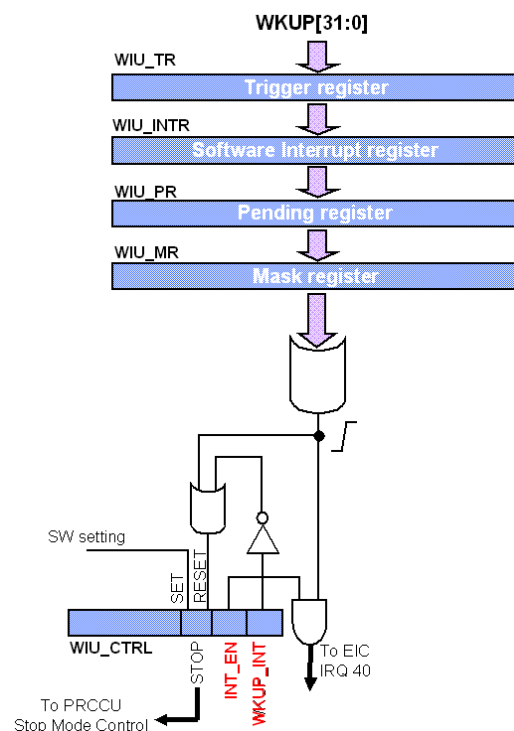
The system mode register allows you to enter the wait for interrupt mode or force system-wide reset.

It is also possible to enter a low-power wait for interrupt mode. In this mode the CPU is halted as described above and a slow clock is automatically selected to drive the peripherals. This means with an 8MHz external oscillator MCLK will be running at 250 KHz in low power mode, assuming the /2 prescaler is also in use. By setting the LPWFI bit in the clock control register, the FLASH memory will automatically be placed into its low power mode when WFI power saving is entered.

```
PRCCU_CCR = 0x00000001;    // Enable low power WFI mode
PRCCU_SMR = 0x00000000;    // Clear the WFI bit and enter wait for interrupt mode
                          // Peripherals are now clocked at Fosc/32
```

3.9.4 STOP Mode and Wake-Up Unit

In STOP mode all the STR730 oscillators are stopped but the configuration of the CPU, memory and peripherals is maintained. The STR730 may be restarted by a dedicated wake up unit allowing it to continue processing. When the chip comes out of STOP mode it will start-up with the backup oscillator and then switch to the main oscillator when it becomes stable. The backup oscillator will allow the STR730 to start processing instructions after around 200usec if the 2MHz on-chip oscillator is used. The chip will switch to the main oscillator when the count in the CMU oscillator counter expires. By default this is around 5msec with an 8MHz oscillator.



The wake-up unit allows 32 GPIO lines to be configured as wake-up pins to exit STOP mode. The wake-up function can be used in addition to the normal IO or alternate pin function

The wake-up unit has 32 dedicated inputs 31 of these lines are connected to the IO pins allowing a wide range of external activity to wake up the STR730. Wake-up line 0 can be connected to a dedicated wake-up pin WUP0 or to a wake up timer that can run from the RC oscillator and provide a periodic wake-up signal. To enter STOP mode you must configure at least one wake-up pin within the wake-up unit to bring the STR730 out of stop mode. Once the wake-up unit is configured the STR730 can be placed in the STOP mode by writing to the STOP bit in the wake-up control register.



To avoid accidentally placing the STR730 into STOP mode you must write the sequence “1, 0, 1” to the STOP bit to enter the low power mode. This bit sequence does not have to appear in consecutive cycles to take effect. However to make sure that the STOP sequence is fully reset it is best to write two dummy zeros at the beginning,. This resets the STOP mode state machine and guarantees that the STOP mode will be entered under all conditions.


```

WU_TR  = 0x00000001;    // Set the wakeup pin trigger level to rising edge
WU_MR  = 0x00000001 ;   // Enable wake up pin zero

WU_CTRL = 0x00000000;    // Write two zeros to clear the STOP mode state machine
WU_CTRL = 0x00000000;

WU_CTRL = 0x00000004;    // Write the STOP sequence 1,0,1
WU_CTRL = 0x00000000;
WU_CTRL = 0x00000004;

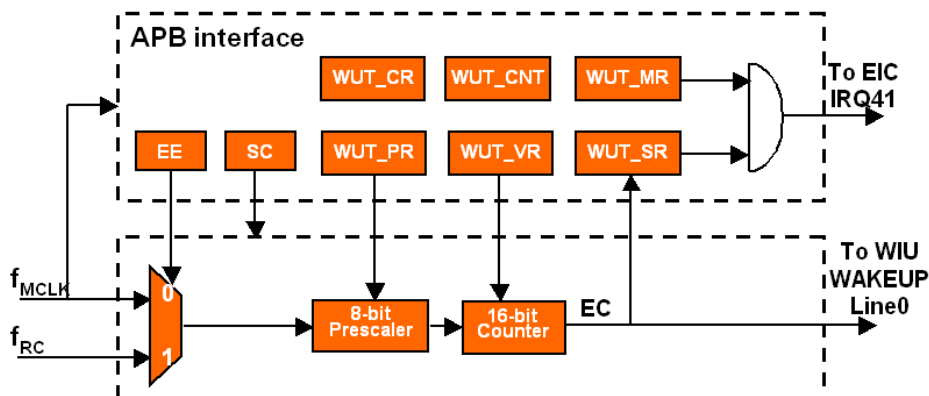
```

Once in STOP mode the CPU will consume minimal current and can resume processing in response to a signal on one of 32 wake up lines. Each wake up line can be individually enabled and can respond to a rising or falling edge. If a pin is used as a wake up pin it does not lose its normal functionality, so for example during normal processing a pin could be configured as the SPI clock. Then when the chip enters stop mode any activity on the SPI bus will wake up the STR730 allowing it to receive the SPI data.

As well as waking up the CPU each wake up line can also generate an interrupt to the CPU. This interrupt can be used independently of the STOP mode giving the STR730 32 external interrupts in addition to the 16 dedicated external interrupt lines. We will look at the interrupt structure in the next section but there are also an additional 16 dedicated external interrupt lines on the STR730. The wake-up interrupt is enabled by setting the INT_EN bit in the wake-up control register. This bit globally enables all the active wake-up pins to generate an interrupt once they are triggered. The interrupt can also be triggered by writing to the software interrupt register. This register has a dedicated bit for each of the wake-up lines, setting the bit corresponding to an active wake-up line will generate an interrupt to the CPU. This can be used as a software interrupt mechanism similar to the CPU SWI instruction or to simulate an external hardware during development or for power-on testing in the final application.

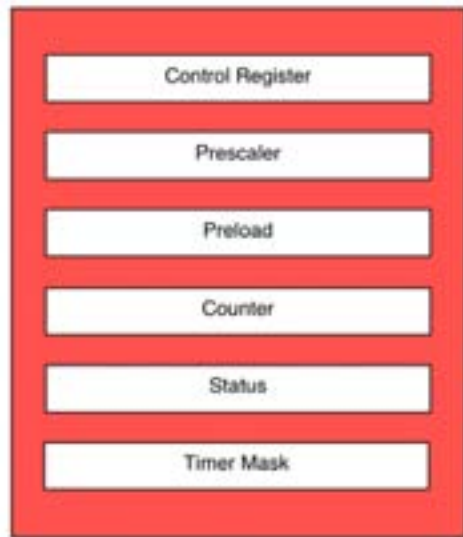
3.10 Wake-Up Timer

Wake up line 0 is configured with the WUP0S bit in configuration register 0. If this bit is set the line is connected to the wake up timer.



The wake up timer can be connected to line 0 of the wake up unit to periodically wake up the STR730. Alternatively it can be used to raise a periodic interrupt.

The wake-up timer is a 16 bit free running count down timer with an 8 bit prescaler. The timer may be clocked from MCLK or from the backup oscillator. However since MCLK will be halted when the chip enters STOP mode the on-board RC oscillator should be used.



The wake up timer is a simple 16 bit counter which can generate a wake up event or interrupt at the end of count

3.10.1 Halt

In Halt mode all the STR730 oscillators are stopped. In this mode the only power loss is the leakage current, which is virtually negligible. To enable the STR730 to enter Halt mode you must set the EN_HALT bit and clear the SRESEN bit in the clock control register. Then to enter Halt mode you must set the Halt bit in the system mode register. It is only possible to exit halt mode with an external reset.

```
PRCCU_CCR = 0x00000800 ;           // Enable halt mode
PRCCU_SMR = 0x00000002 ;           // Enter halt mode
```

Mode	Quartz Oscillator	RC Oscillator	Main VReg	Low Power VReg	PLL	Flash	ARM7TDMI	Digital Modules
HALT	OFF ⁽¹⁾	OFF	OFF	ON ⁽⁶⁾	OFF	Power Down	OFF	OFF
STOP	OFF ⁽¹⁾	ON/OFF ⁽³⁾	OFF	ON ⁽⁶⁾	OFF	Power Down	OFF	OFF ⁽¹⁰⁾
WFI	ON/OFF ⁽²⁾	ON/OFF ⁽⁴⁾	ON	ON ⁽⁶⁾	ON ⁽⁷⁾	ON	OFF ⁽⁹⁾	ON/OFF ⁽¹¹⁾
LPWFI	ON/OFF ⁽²⁾	ON/OFF ⁽⁴⁾	ON/OFF ⁽⁵⁾	ON ⁽⁶⁾	OFF	Power Down ⁽⁶⁾	OFF ⁽⁹⁾	ON/OFF ⁽¹¹⁾

- 1) The Quartz Oscillator is not switched-off if the RTC module is running.
- 2) The Quartz Oscillator may be switched-off only if the CMU is currently selecting the internal RC Oscillator as system clock.
- 3) The RC-Oscillator is switched-off if bit RCHSE of CMU CTRL register is set to 1
- 4) The RC-Oscillator may be switched-off only if the CMU is currently selecting the Quartz Oscillator as system clock.
- 5) The Main Voltage Regulator is switched-off if bit VRLPW of Voltage Regulator VRCTR Register is cleared to 0.
- 6) The Low Power Voltage Regulator is never switched-off. To reduce the power consumption, you may reduce the Low Power Voltage Regulator output current through bits LPVRCC of CFG_R1 register.
- 7) The PLL may be switched-off by setting bit CK2_16 of PRCCU_CFR register
- 8) The FLASH module enters Power Down if bit PWD of FLASH FCR0 is set by software to 1.
- 9) ARM7TDMI input signal nWAIT is forced low.
- 10) If RC-Oscillator is on, the Wake-up Timer is running. If RTC module is running entering Stop mode does not stop RTC counter.
- 11) Digital modules status is defined by Configuration Registers PCGR0/1 and PCGRB0/1

3.10.2 Low Power Modes: Important Note

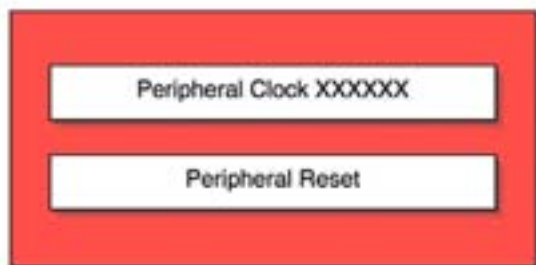
If you are experimenting with the low power modes remember that the chip will run your code before the JTAG gets control of the STR7. If your code places the STR7 into a power-down mode before the JTAG gets control of the CPU, you will have effectively lost control of the STR7. This is an easy mistake to make when experimenting with the STR7 clock sources and power down modes. To prevent this problem make sure the Low Power debug bit is set in the boot configuration register. If you do get caught out by this problem you can recover by changing the external boot mode pins to boot from RAM, since the RAM will not contain any valid code you, can get control back with the JTAG and erase the FLASH.

Exercise 9: Power Modes

In this exercise the STR7 is configured for low power operation and is then stepped through its different power modes and the resulting power reduction can be observed on a meter .

3.11 Advanced Peripheral Bus Bridges

Both of the Advanced Peripheral Busses have bridges connected the main Advanced High speed Bus. Within these bridges are a group of user-controlled registers that enable error detection on the peripheral busses as well as clock and reset control



The Clock Disable register can be used to disable the clock signal to any peripheral connected to the APB bus. This duplicates the peripheral enable register in the PRCCU.

3.11.1 Software Reset

In the APB bridge the Software Reset register allows a given peripheral to be reset by a system-wide reset or by setting an appropriate bit, an individual peripheral can manually be held in reset until it is released by your application software. A complete software reset can be forced by setting the Halt bit in the System Mode register. However you must first enable this function by setting the HALT_END and SRES_EN bits in the Clock Control register. The clock flag register contains a software reset flag so that you can differentiate between a hardware and software reset if necessary.

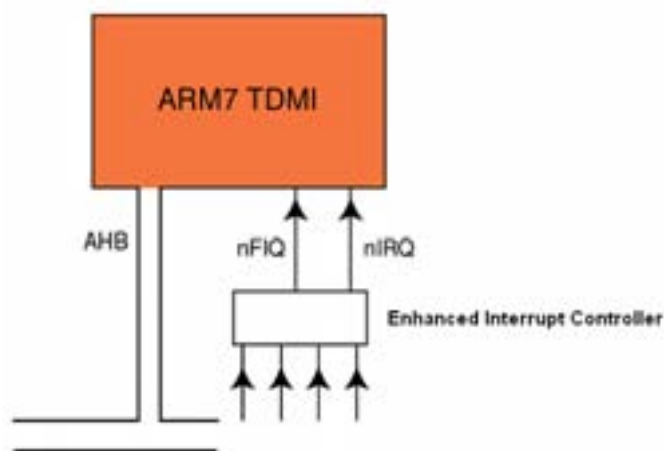
3.11.2 Error Registers

The three additional registers in the APB bridge allow you to detect an access to the unused memory window located within the APB1 and APB2 address range. This memory window is reflected over the address range (0x010h-0x3FFh) It is also possible to specify a maximum read time for any register on these two busses.

Access to an unused memory region or an access timeout can generate an abort interrupt. After reset both the memory abort and timeout mechanisms are disabled and it is probably best to leave them this way.

3.12 Interrupt Structure

The ARM7 CPU has two external interrupt lines for the fast interrupt request (FIQ) and general purpose interrupt IRQ request modes. As a generalisation in an ARM7 system there should only be one interrupt source that generates an FIQ interrupt so the processor can enter this mode and start processing the interrupt as fast as possible. This means that all the other interrupt sources must be connected to the IRQ interrupt. In a simple system they could be connected through a large OR gate. This would mean that when an interrupt was asserted the CPU would have to check each peripheral in order to determine the source of the interrupt. This could take many cycles and is inappropriate for real-time control. Clearly a more sophisticated approach is required. In order to handle the external interrupts, ST have designed an interrupt support peripheral called the “Enhanced Interrupt Controller” (EIC) that greatly speeds-up servicing of peripheral interrupts



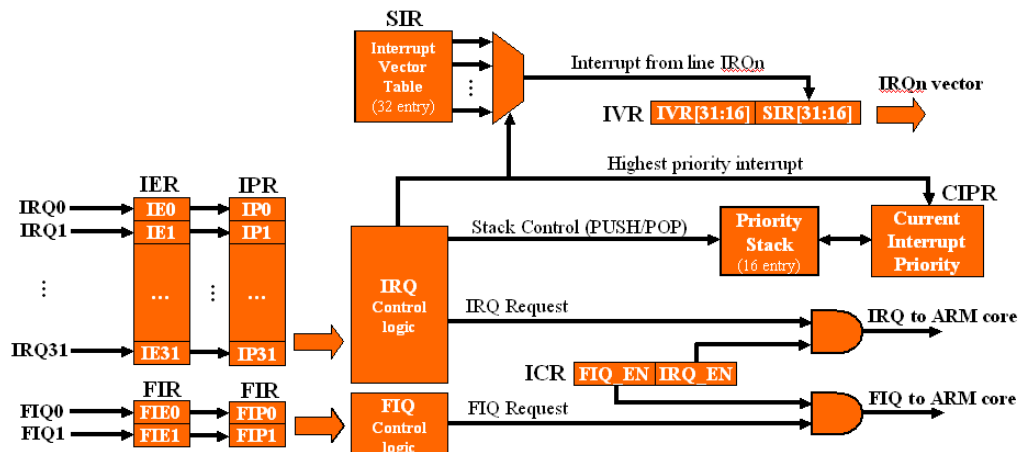
The EIC provides additional hardware support for the on chip peripheral interrupts. Without the VIC the interrupt response time would be very slow.

The EIC can be described in three parts. First it provides enhanced interrupt support for servicing the on-chip peripherals as general purpose interrupts, secondly it allows the watchdog and timer 0 global interrupt to be connected to the Fast interrupt line to create a high priority pair of interrupts and finally there is a dedicated external interrupt unit that allows up to 16 GPIO lines to be configured as external interrupt lines.

3.12.1 General Purpose Interrupt

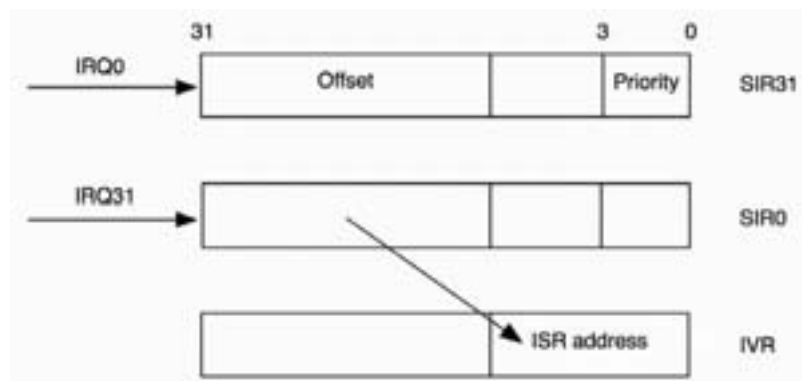
An individual interrupt source must be enabled both within the EIC by setting the appropriate bit in the Interrupt Enable register and globally enabling interrupt generation from the EIC by setting the IRQ output enable bit in the Interrupt Control register and finally within the peripheral itself by setting the required interrupt enable flag.

Since all of the on-chip peripherals have an interrupt channel which can be connected to the IRQ line to signal an interrupt to the ARM7 CPU and there are some 26 interrupt sources on-chip and only one IRQ line, it would be impossible to have efficient interrupt handling without additional hardware support.



The Enhanced interrupt controller (EIC) provides a vectored lookup table for the IRQ interrupts and a 16 level priority stack for nested interrupts. The FIQ interrupt may also be connected to the watchdog or Timer0

The EIC provides the necessary control logic to allow any interrupt source to assert an interrupt on the nIRQ line. However in addition it provides an internal interrupt vector table which acts as a hardware lookup table to provide the address of the interrupt service routine to match the current interrupt request. This interrupt vector table is made up of 32 source interrupt registers and one interrupt vector register.



When a peripheral generates an IRQ interrupt the offset value stored in the matching source interrupt registers (SIR) is transferred to the lower half word of the Interrupt vector register.

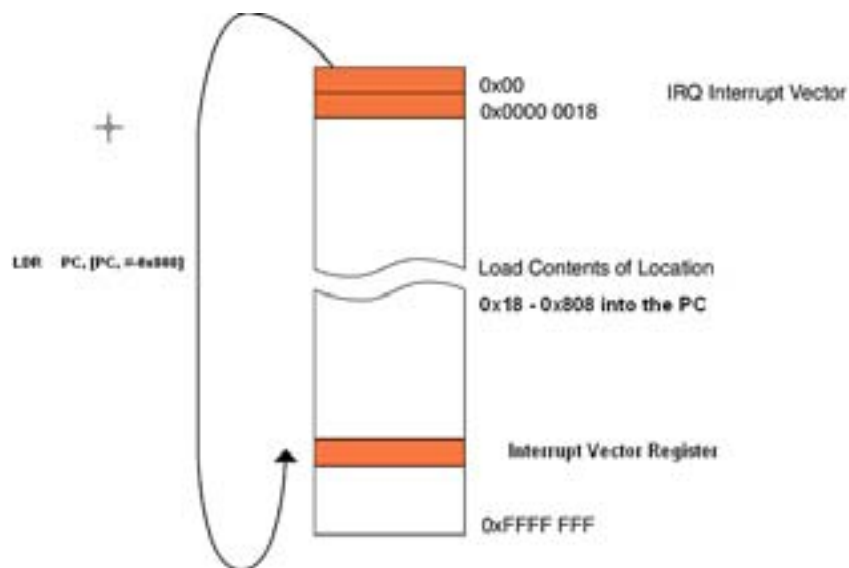
The upper half-word of each SIR register is programmed with the address of the matching interrupt service routine. When an interrupt occurs this half-word is transferred to the lower half-word of the interrupt vector register. This gives a full address to the required interrupt service routine. This mechanism may be used in one of two ways to allow low latency interrupt handling (i.e. serve the interrupt as fast as possible) or to support nested interrupt (i.e. support prioritised interrupt handling). In the first case the linker must be used to locate the entry to all the interrupt service routines within one 64K block. The address of this block is held in the upper half-word of the IVR and must be set when the EIC is configured. Next the address of each Interrupt service routine within the 64K block is stored in the upper half-word of the matching source interrupt register. The easiest way to configure the IVR and SIR registers is to program the upper half of the IVR register to zero and then to use the linker to locate all the interrupt routines in the first 64K of the on-chip FLASH. The address of each interrupt service routine can then be loaded into the SIR registers with the following line of C code:

```
EIC_SIR18 = ((unsigned int)ADC_IRQ_isr )<<16 | 1;
```

When the peripheral generates an interrupt, the full address of the matching interrupt service routine will be assembled in the Interrupt Vector register. While this is happening the ARM7 CPU will independently enter IRQ mode and vector to address 0x00000018, the IRQ interrupt vector. By placing the line of assembler shown below on the IRQ interrupt vector, we can load the interrupt service routine address into the program counter in one cycle and enter the interrupt routine.

```
LDR PC, [PC, #-0x407]
```

What this instruction does is take the current value of the program counter (0x00000018) deduct 0x407 and load the contents of this memory location into the program counter.



When an IRQ interrupt is generated the Address of the ISR will be loaded into the Interrupt vector register. This value can be loaded directly into the PC forcing a jump to the ISR.

By deducting 0x00000408 from 0x00000018 as unsigned integers, we are jumping off the bottom of the memory map and wrapping round to the top. The address we end up with is the interrupt vector register. This contains the address of the ISR, which is then loaded into the program counter. This jumps us straight from the IRQ interrupt vector into the dedicated interrupt service routine. Reading the Interrupt Vector register also signals to the EIC that the interrupt is being serviced. This allows the EIC to terminate its handling of the current interrupt and process any further pending interrupts.

```
EIC_SIR18 = ((unsigned int)ADC_IRQ_isr + 4)<<16 | 1; // Program IRQ vector with
                                                    // ISR address
EIC_IER0 = CHANNEL(18); // Enable the IRQ channel interrupt
EIC_ICR = EIC_IRQ_ENABLE; // Globally enable the EIC IRQ interrupts
```

3.12.2 Leaving An IRQ Interrupt

Before you quit the interrupt it is very important to clear the interrupt status flag in the underlying peripheral. If you fail to do this, the peripheral will continuously generate further interrupts until the flag is cleared. Secondly the interrupt channel in the EIC must also be cleared by writing to the appropriate channel bit in the Interrupt pending

register. When the pending bit for a given channel is cleared this signals the end of interrupt to the EIC which then begins its end of interrupt sequence. This sequence POPs the priority stack and serves and lower priority pending interrupt.

3.12.3 Nested Interrupt Handling

As mentioned in the first chapter the ARM7 does not inherently support nested interrupts. When we enter any IRQ interrupt the I bit in the CPSR is set preventing any further interrupts. This keeps the interrupt structure within the ARM7 simple and fast. However in some applications it may be necessary to have a hierarchy of interrupt sources which are able to pre-empt each other. To this end the EIC provides a priority field in each source interrupt register. This priority field allows fifteen levels of priority to be assigned among the IRQ interrupt sources. So if a number of interrupts are pending the EIC will select the interrupt source with highest value in the priority field of its associated source interrupt register. However the EIC also provides an internal priority stack. If the ARM7 CPU is serving an interrupt and a new interrupt occurs the EIC will compare the current interrupt priority to the priority on the new interrupt. If the new interrupt has a higher priority it will cause a fresh IRQ interrupt to be asserted to the ARM7 and the existing interrupt will be pushed onto the internal stack of the EIC.

So the on-chip interrupt structure is capable of supporting nested interrupts however we still have the problem that the ARM7 disables IRQ interrupts as soon as it starts to service a general purpose interrupt source. In order to allow the SRT7 to fully support nested interrupts we need to add some software support to re enable the IRQ interrupt once the interrupt has been entered. This software support is added in the form of two macros which are added to the entry and exit points on the interrupt function. The general form for a nested interrupt routine is shown below.

```
void EINT1_IRQ_isr ( void ) __attribute__ ((interrupt ("IRQ")));

void EINT1_IRQ_isr ( void )
{
    SWITCH_IRQ_TO_SYS;           // Entry Macro to support nested interrupts
/* BEGIN USER CODE XTI_IRQ */

/* END USER CODE XTI_IRQ */
    SWITCH_SYS_TO_IRQ;           // Exit macro to support nested interrupts
    EIC_IPR0 = CHANNEL(1);       // clear IRQ Pending bit
}
```

The function is declared with the IRQ attribute so the compiler will stack any registers between R0 and R12 that will be used during the interrupt routine and the IRQ mode link register. In addition the switch_IRQ_to_sys macro contains the following instructions

```
MRS      LR, SPSR                ; Copy SPSR_irq to LR
STMFD    SP!, {LR}               ; Save SPSR_irq
MSR      CPSR_c, #0x1F           ; Enable IRQ (Sys Mode)
STMFD    SP!, {LR}               ; Save the Sys mode LR onto the stack
```

First we need to protect any registers in the IRQ mode that may be corrupted. The compiler has saved the user registers and the link register but for nested interrupts we also need to save the IRQ SPSR onto the IRQ stack. Next by programming the CPSR we leave IRQ mode and enter system mode and at the same time re enable the general purpose interrupts. System mode is effectively the same as user mode except that the MSR and MRS instructions may be used. The final instruction in the macro saves the system mode link register onto the system stack. Since the system mode link register is the same as the user mode link register it must be protected or else the background code will be corrupted.

The exit macro reverses this process with the following assembly instructions

```
LDMFD    SP!, {LR}               ; Restore LR
MSR      CPSR_c, #0x92           ; Disable IRQ (IRQ Mode)
LDMFD    SP!, {LR}               ; Restore SPSR_irq to LR
MSR      SPSR_cxsf, LR           ; Copy LR to SPSR_irq
```

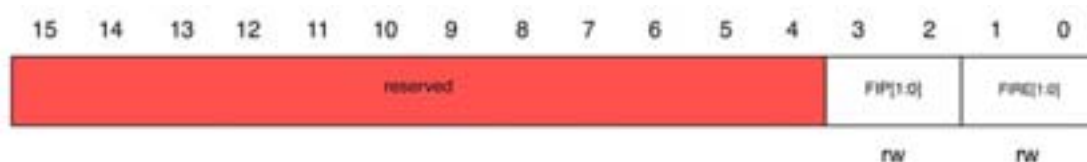
Once the user code has finished serving the interrupt the macro restores the system link register then programs the CPSR to force the CPU back into IRQ mode with general purpose interrupts enabled and finally the IRQ spsr is restored. After the macro has finished the interrupt pending bit for the current interrupt channel must be cleared to signal the end of interrupt to the EIC and then the compiler epilogue code will restore the user registers and return from the interrupt.

Exercise 10: IRQ/Nested interrupts

In this exercise the EIC is configured to handle nested interrupts

3.12.4 FIQ Interrupt

In addition to supporting the IRQ interrupts the EIC also allows the watchdog and timer0 global interrupt to be connected to the FIQ interrupt line. Unlike the IRQ interrupts there may only be two peripherals connected to the FIQ line and the EIC does not provide any vector table support. Either or both peripherals must have their interrupt sources enabled and in the EIC the Fast interrupt register provides two bits to allow these interrupt channels to be enabled.



The Fast Interrupt register is used to enable FIQ interrupt support in the EIC and provide flags of pending interrupts.

In addition the Interrupt control register has a global enable bit which must be set to enable any FIQ interrupts. Once the FIQ interrupts are enabled an interrupt on either source can generate an FIQ interrupt causing the ARM7 CPU to vector to 0x0000001C, the FIQ interrupt vector. Since the EIC does not provide any additional support for FIQ handling we can enter a dedicated FIQ ISR routine. If there is more than one FIQ interrupt source it will be necessary to read the FIQ pending bits in the FIR register to determine what code to run.

```

IOPORT1_PC0 = 0x00000100;    // Configure Port 1.8 as an input
IOPORT1_PC1 = 0x00000000;
IOPORT1_PC2 = 0x00000000;

CFG_EITE0 = 0x00000003;    // Configure external interrupt rising edge
CFG_EITE1 = 0x00000000;
CFG_EITE2 = 0x00000003;
EIC_FIR = 0x00000001;    // Enable the INT0 FIQ request

EIC_ICR = 0x00000002;

void FIQ_Handler (void)    __attribute__ ((interrupt ("FIQ")));

void FIQ_Handler (void)
{
    EIC_FIR    |= 0x00000008;    // Clear the Fast interrupt pending bit
}

```

3.12.5 Leaving an FIQ interrupt

It is important to remember to clear any interrupt status flags in the timer or watchdog before quitting the interrupt. Failure to do this will cause continual interrupts to be generated.

Exercise 11: FIQ interrupt

In this exercise the external interrupt 0 line on pin 1.8 is used to generate an FIQ.

3.12.6 External Interrupts

The STR730 has a dedicated external interrupt controller which allows up to 16 GPIO pins to generate an interrupt, in addition to acting as a standard GPIO pin or while performing their secondary function. Each external interrupt pin may be configured to be triggered by a high or low level on the pin, or by a rising or falling edge, or by both edges.

Some of these GPIO pins may also be configured as IO pins for peripherals such as the CAN or UART. The external interrupts may be used in conjunction with these peripherals so for example, an interrupt can be generated at the start of activity on the CAN bus.

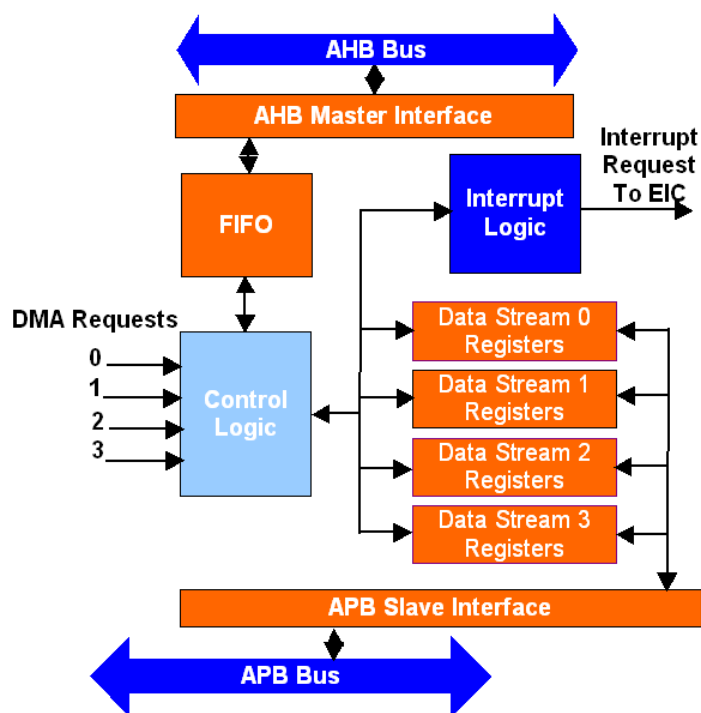
EITE2(n)	1	1	1	1	0	0	0	0
EITE1(n)	1	1	0	0	1	1	0	0
EITE0(n)	1	0	1	0	1	0	1	1
Event on INTn input channel triggered interrupt Request on IEC	INTn channel disabled	INTn rising and falling edges	INTn rising edge	INTn falling edge	INTn HIGH level	INTn LOW level	INTn HIGH level (*)	INTn LOW level

The external interrupts are configured with three registers in the configuration block. Each interrupt pin is enabled and configured by one dedicated bit in each register as shown below.

Once enabled each external interrupt is connected at a dedicated slot within the EIC. This allows each external interrupt to be handled by a dedicated interrupt service routine running within the ARM7 as a general-purpose interrupt.

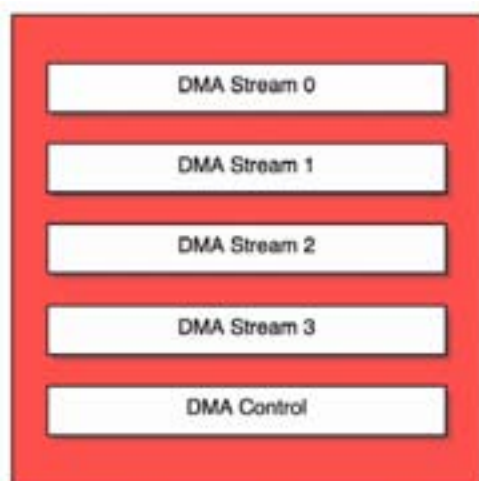
3.12.7 DMA Controller

The STR73x contains four DMA units each with four data streams. Each of these data streams is dedicated to a given peripheral and may transfer a block of data to or from a given peripheral, without any CPU intervention. In addition, on each DMA unit one of the data streams (stream 3) may be used for memory-to-memory copies. Each DMA unit has a single interrupt line that is connected to the IRQ interrupt via the EIC. In order to transfer the data, the DMA units share the internal busses with the CPU. Arbitration between the DMA units and the CPU is controlled by a dedicated arbitration unit called the “Native Bus Arbitrator”. After a reset this unit disables DMA transfers and must be initialised before any DMA activity can take place. This is discussed later in the Native Arbitrator section.



Each of the four DMA units has four DMA streams that can transfer data between memory and peripherals. In addition, stream 3 in each unit can perform memory to memory transfers.

The register set in each of the four DMA units is the same so once you have one DMA transfer working, all the others will work the same way. The programmer's interface for each DMA unit consists of a set of data stream registers for each DMA stream and a group of interrupt handling registers. In addition each peripheral that supports DMA transfers will have DMA configuration bits within its own registers which will also need to be configured.



Each DMA unit has an identical register set. Also each DMA stream is controlled in the same manner. So although there are a lot of registers the DMA units can be decomposed into a fairly simple subset of registers.

Each data stream is controlled by twelve registers. Four registers are used to store the memory source or destination address for the transfer. Each of these registers stores a half-word of the 32 bit address. The count register stores the number of DMA transfers required and finally the control register initialises and enables the DMA transfer.



Each DMA stream is controlled by twelve registers. These define the source and destination address and all of the configuration and control registers.

Each of the DMA data streams are connected to a peripheral register which acts as a source or sink of DMA data. When you initialise a DMA data stream you must configure the DIR bit in the control register to match the direction of data flow expected by the peripheral. The bit is zero if the peripheral is a data source or one if the peripheral is a data sink.



Once the data direction is enabled the matching source or destination base address registers can be initialised with the target memory address for the DMA transfer. This memory address can be a location in RAM or it can be a register within another peripheral. So it is possible to make a DMA transfer from the ADC results register directly into a BSPI transmit register, without any CPU overhead. In addition to setting the DMA direction you must also set the transfer unit size. This controls the data width that is transferred with each DMA cycle. The unit data size is controlled by the size bits in the control register DeSIZE for the destination size or SoSIZE for the source size. The sizes supported are 8,16 and 32 bits. The number of data units to transfer in each DMA session must be programmed into the maximum count register. When the DMA session starts, the contents of this register are loaded into the terminal count register, which is decremented on each transfer until it reaches zero and the DMA session ends.

During the DMA transfer the memory address in the active base address register will be used as the starting address for the DMA transfer. After each transfer the DMA unit allows you to increment this address by the size of the DMA unit data width. So for example, you can transfer the contents of an array or buffer to peripheral register or transfer the current contents of a peripheral register to a single fixed memory location.. This option to increment the base address is enabled by the Inc bits (DeInc, SoInc) in the control register. Finally a DMA data stream can be enabled as a circular buffer by setting the circular bit in the control register. In this mode when a DMA session ends it is automatically reset to its starting values and a new session begins. If the circular bit is not set, the DMA channel will be disabled at the end of the session and must be reinitialised by the CPU.

During a DMA transfer the data is moved from the source into the DMA unit and then from the DMA unit to the destination. Each of these transfers occurs as a burst of data over the internal busses, in contention with the CPU. You can also control the size of these bursts when a peripheral is the source of the DMA data. The burst size may be configured to be 1,4,8 or 16 data units in length. Once the DMA channel parameters are configured the session can be started by setting the control register enable bit.

In addition to supporting data transfers between peripherals and memory stream, stream three of each DMA unit can be used to transfer data between two memory locations. Again these can be RAM or registers within a peripheral. In the control register for stream 3 in each DMA unit there is an additional Mem2Mem bit that is used to enable this memory transfer feature.

```

ARB_PRIOR      = 0x00000003;           // Enable DMA arbitration
DMA0_SOURCEL3  = 0x00000000;           // Set the source address
DMA0_SOURCEH3  = 0x0000A000;           // 0xA0000000
DMA0_DESTL3    = 0x00000800;           // Set the destination address
DMA0_DESTH3    = 0x0000A000;           // 0xA0000800
DMA0_MAX3      = 0x00000400;           // Set the transfer size to 1K
DMA0_CTRL3     = 0x00000917;           // Start the transfer
  
```



Each DMA unit has a dedicated interrupt line that can be used to generate an interrupt at the end of a DMA transfer for a given stream within the unit. An error interrupt can also be enabled. This interrupt source will be triggered if the DMA stream loses arbitration on the AHB bus to the CPU. If the DMA controller attempts to access an invalid memory location, an abort exception will be generated within the CPU.



Each DMA stream will take one cycle to transfer data into the DMA unit and a further cycle to transfer data to the destination. However each transfer to and from the DMA unit is at the data size defined for that transfer. For example you could transfer a word (4 bytes) of data into the DMA unit in one cycle and then transfer each byte to an eight-bit location in four cycles. In addition the first transfer into the DMA unit takes 7 cycles and the first transfer out of the DMA unit takes an additional 5 cycles. There is also a latency within the DMA unit of one cycle.

This gives us an overhead of 11 cycles. To transfer four words to a destination of eight half words would take $11 + 4 \times 8 = 23$ cycles.

The STR73x has sixteen DMA streams and the ARM7 CPU which all use the same internal busses to send data. The DMA units arbitrate between themselves for access to the bus using a simple round-robin scheme that ensures equal access to the STR73x bus. Once a DMA unit has won arbitration, it will arbitrate its own four DMA streams in the same fashion. Once the DMA controller has internally selected a DMA stream to begin its transfer, this stream will contend with the CPU to gain the bus. This second level of arbitration is controlled by the Native Bus Arbitrator.

Exercise 12: DMA Transfer

In this exercise the DMA system is used to transfer a block of data between two address ranges.

3.13 Native Bus Arbitrator

The Native Bus Arbitrator is used to resolve simultaneous bus requests with the DMA unit.



The Native bus arbitrator is used to arbitrate between the DMA units and the CPU.

The arbitration scheme between the CPU and the DMA units is controlled by the priority register. This register has only two options; by clearing the lowest two bits DMA transfers are halted and the CPU has sole access to the internal busses. By setting these bits the DMA units are enabled with priority over the CPU. When DMA transfers are enabled in the Native Bus Arbitrator, any pending DMA transfers will win the bus and start transferring data. The Native Bus Arbitrator has an internal count-down timer that will count down to zero. When this timer reaches zero, the currently active DMA transfer will be halted and the CPU will gain access to the bus. The timer will be reloaded and will again count down to zero. When it reaches zero, once more control will be handed back to the DMA controller and transfers will restart. The countdown timer is a 16-bit timer which is clocked by Mclk and the reload value is stored in the Time Out register. On reset, this register contains 0xFFFF, which freezes the timer, giving full control of the bus to the DMA controller. If one of the DMA streams attempts to access a memory location that is not a peripheral register or memory location the Native Bus Arbitrator will generate an interrupt to the ARM7 CPU, causing an Abort interrupt and the DMA controller will stop transfers and enter a lock state. Once the CPU has entered the abort ISR it must correct the problem in the DMA unit and unlock the Native Bus Arbitrator by writing "1" to the Abort bit in the control register.

3.14 Conclusion

This is an important chapter! You must be familiar with the system architecture of the STR7 in order to use it successfully.

4 Chapter 4: User Peripherals

4.1 Outline

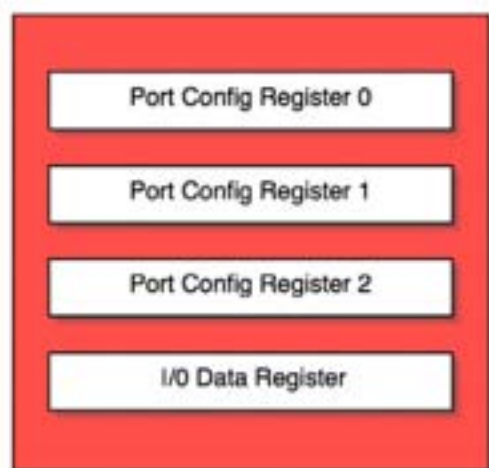
This chapter presents each of the user peripherals in turn. The examples show how to configure and operate each peripheral. By working through each peripheral in turn you will gain a good insight into the capabilities of the STR73x. The example programs can be used as the basis for more complex programs or additionally there is a driver library available from the ST Microelectronics website.

4.2 General Purpose Peripherals

As we have seen in Chapter 3, all the user peripherals are located on the Advanced Peripheral Bus (APB) and are clocked by the system clock Mclk. Before reaching the peripherals the system clock passes through the configuration registers peripheral control registers. After reset these registers disable the individual clocks to each peripheral. This ensures that the STR730 starts up in a low power configuration. You must enable each peripheral clock source before the peripheral can be accessed.

4.3 GPIO

The GPIO ports that are provided by ARM are fairly simple TTL-style IO ports. However ST Microelectronics have designed their own far more sophisticated port structure for the STR7. Depending on the variant of STR7 you are using there are up to 112 GPIO lines which can each sink up to 2mA. On reset the microcontroller pins will default to input pins. The application software can configure the GPIO to be output pins or enable the pins' alternate function. The alternate function allows a user peripheral such as a timer or UART to be connected to a specific external pin. In addition to the alternate function, the external interrupt controller allows up to 16 pins to be independently configured as external interrupts and 32 can act as wake-up pins. This is in addition to the pin acting as a simple GPIO or peripheral pin. If you are using the STR7 in a low-power application this means you can put the device into a low-power mode and then wake it up when there is an activity on a peripheral. For example, you can trigger a wake-up when a character is sent over the UART.



The extended GPIO ports on the STR730 are configured by three control registers. A single data register provides read/write access to each IO port.

The GPIO pins are controlled by three configuration registers and one data register per 16 pin port. Each IO pin is configured by setting a bit in each of the configuration registers i.e. bit zero in each of the configuration registers controls the configuration of pin zero. The table below summarises the possible drive configurations for each pin.

Configuration	Hi-Z Analog Input	Input	Reserved	Input / Output	Output	Output	Alternate Function	Alternate Function
Output Mode	Tristate	Tristate	Reserved	Weak Push-Pull	Open Drain	Push-Pull	Open Drain	Push-Pull
Input Mode	Analog Input	TTL Levels	Reserved	TTL Levels	TTL Levels	TTL Levels	TTL Levels	TTL Levels

Once the GPIO pins have been configured you can read and write data directly to and from the port latches via the IO data register. The IO data register and the configuration registers are word-wide registers but they only

have active bits in the lower half word. The GPIO port registers must be accessed as 32-bit or 16-bit words. Byte-wide access is not allowed. The include file generated by StartEasy defines the port registers as 32-bit words and this will force the compiler to generate the correct addressing mode. If you need to write to a single pin you will need to do a read-modify-write operation in the IO Data register. This will take several cycles and may be interrupted. When writing your code you must be careful that the port data is not corrupted in the interrupt service routine.

```
{
    unsigned int delay,count;    // Define local variables

    IOPORT2_PC0 = 0x0000;        // Configure port2 bits 8-15 as open drain output
    IOPORT2_PC1 = 0x0000;
    IOPORT2_PC2 = 0xFF00

    while(1)
    {
        count = 0x00000080;      // Set the LED tracer start value
        while((count= count<<1) <= 0x00010000) // Shift the set bit to enable the
                                                // next LED
        {
            for(delay = 0;delay<=0x0000f000;delay++)
            {
                ;
            }

            IOPORT2_PD =  count; // Write the LED pattern to the Port data register
        }
    }
}
```

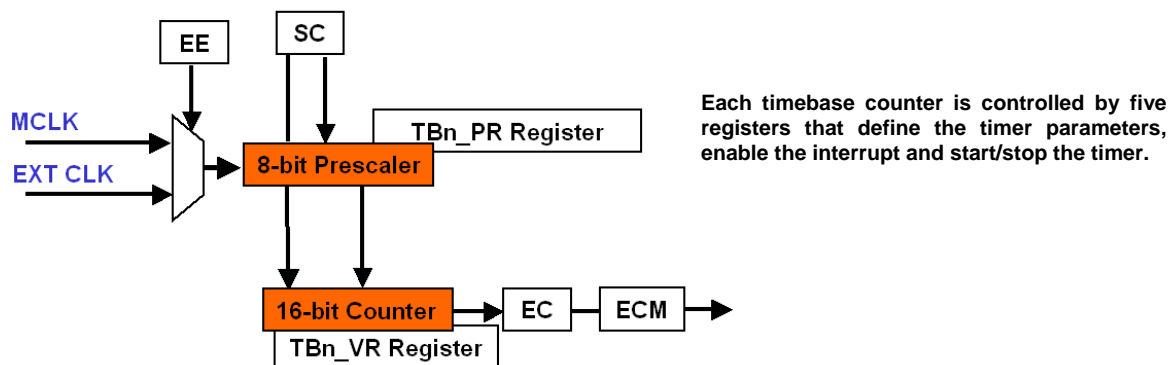
Exercise 13: GPIO

This exercise demonstrates programming the GPIO pins to control the seven segment LED displays on the evaluation board.

4.4 Timebase Timer

The STR73x has three simple sixteen bit timers with eight bit prescalers that can be used to generate periodic interrupts for scheduling software events. These may be simple timers but they provide necessary hardware support for operating systems, without having to loose one of the main timer blocks.

The timebase timers are countdown counters with an automatic preload register. The value stored in the preload register will be loaded into the timer which counts down to 0x0000. Once the count reaches 0x0000, an interrupt may be generated and the preload value is reloaded and the count down cycle restarts.



The STR730 has three timebase timers. These are 16-bit autoreload timers which can generate an interrupt when the timer reaches its end of count.

Each of the timers may be clocked from either the system clock M_{CLK} or from F_{ext} which is derived directly from the external oscillator. Since you can use the PRCCU to vary the frequency of M_{CLK} the option to use F_{ext} is useful since this will guarantee a regular timebase.

The timebase counter control register is used to select the clock source from either M_{CLK} or F_{ext} . Once selected, the counting frequency can be set by loading the prescaler with an eight bit value. This simply divides the clock source by the load value plus 1. Once the tick rate has been set the value stored in the preload register will determine the count period by the formula below:

$$\text{Timeout (in microseconds)} = (\text{Preload value} + 1) \times (\text{reload value} + 1) \times T_{ck} / 1000$$

Where T_{ck} = selected clock period in nanoseconds

Once the count values are initialised, the timebase interrupt may be enabled by setting the ECM bit in the mask register. Finally the count can be started by setting the SC bit in the control register

```
TB0_MR = 0x00000001;    // Enable interrupt
TB0_PR = 0x000000FF;    // Start the prescaler to maximum divide value
TB0_VR = 0x0000FFFF;    // Set the reload register to the maximum count value
TB0_CR = 0x00000002;    // Start the counter
```

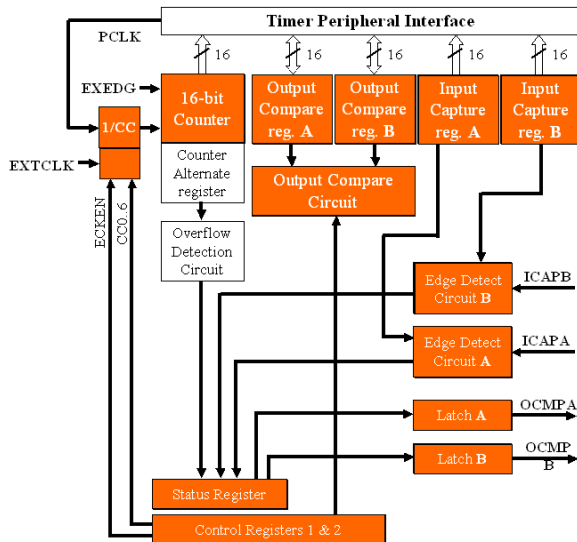
When the interrupt occurs a general purpose interrupt will be generated and end of count flag will be set in the status register. This flag must be cleared before leaving the interrupt routine.

Exercise 14: Timebase Timer

This exercise configures timebase timer 0 with its maximum count. At the end of count an interrupt is generated . In the interrupt the elements of the seven segment display are toggled.

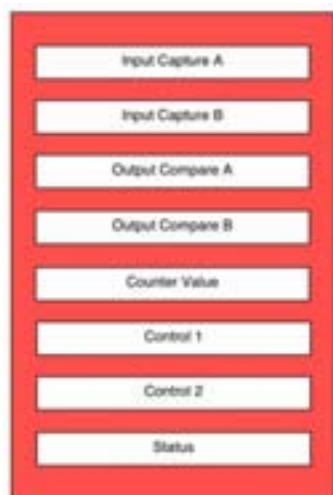
4.5 Timer Counters

The STR7 has up to nine timer counter blocks. These timers consist of a free running 16 bit counter with prescaler and a series of capture and compare registers. The timer counters have several dedicated operating modes. These modes allow easy configuration of the timer counters for common functions such as pulse measurement or PWM generation.



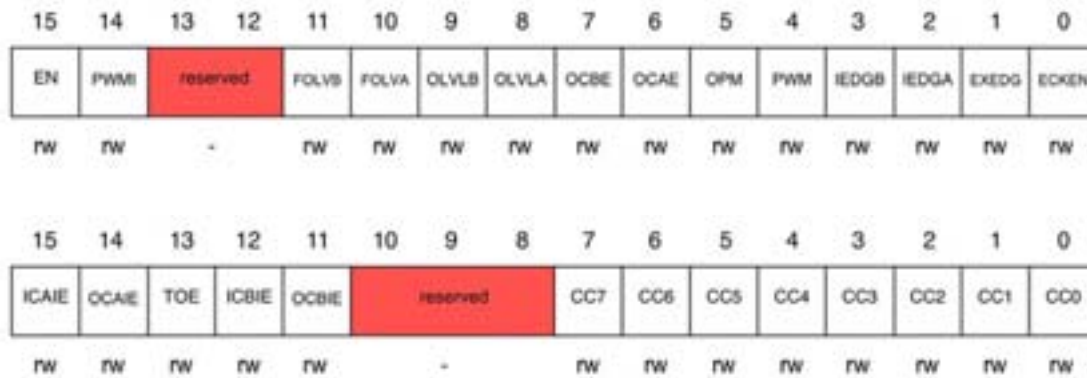
The STR730 has three independent 16 bit timer counters with dedicated capture and compare registers.

The timer clock may be selected from either the M_{CLK} bus clock or from an external clock source, depending on the value of the ECKEN bit in control register 1. If the timer is clocked from an external clock source you can link the external clock pin to the output of a second timer block, making it possible to cascade the timers to build more complex timer arrays. Each timer counter is controlled by eight registers, which allow it to be configured for input capture and output waveform generation.



Each timer has a regular programmers interface that allows easy configuration of each operating mode.

The timer block is a 16-bit counter with an eight bit prescaler and is managed by two control registers.

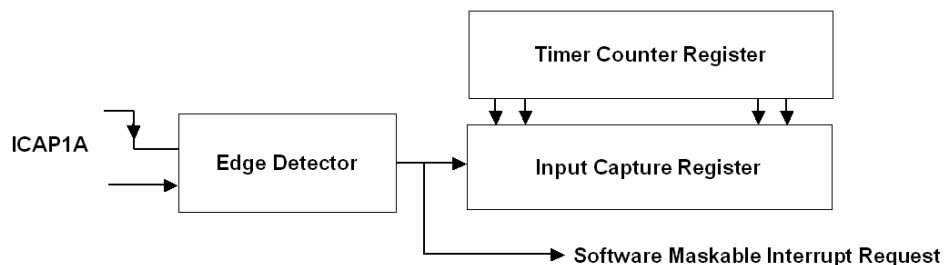


The timer control registers configure the prescaler CC0-CC7 and the various capture compare operating modes.

The clock source is selected with the ECKEN bit in control register 1. By default this bit is set at zero to select the system clock. The timer prescaler value is held in the lower eight bits of control register 2. This prescaler is only applied to the system clock. If an external clock source is selected, it is fed directly to the timer. Should the external clock be used, an additional bit EXEDG in Control Register 1 allows you to determine if the rising or falling edge will increment the counter.

4.5.1 Input Capture

Each of the STR7 timers has two input capture registers each with a matching capture pin. Each of these registers may be configured to capture the timer count when there is a transition on a matching input capture pin



Each timer counter has two capture registers which may be used for custom edge measurement

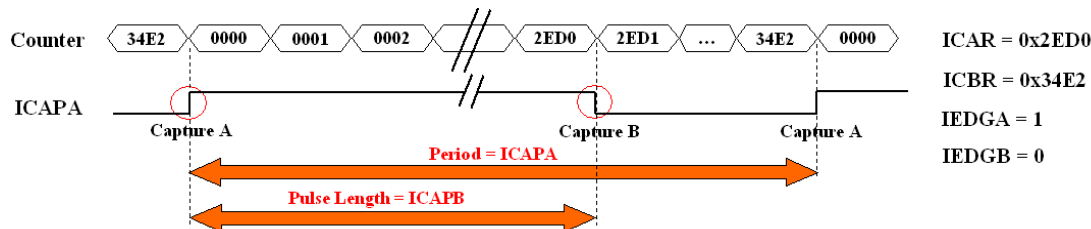
Each capture pin may be configured to trigger a capture event on a rising or falling edge by programming the IEDGA and IEDGB bits in control register 1. When a capture occurs, either capture register may generate an interrupt by enabling the ICAIE and ICBIE bits in control register 2.

```

TIM0_CR2 = 0x000090FF; // Enable input capture interrupts and set the prescaler
TIM0_CR1 = 0x00008004; // Timer enable ICAPA rising edge, ICAPB falling edge
  
```

4.5.2 PWM Input Capture

Additionally there is a PWD capture mode that allows the measurement of pulse width and the period of a signal applied to the ICAPA pin. By setting the PWMI pin in Control Register 1, the edge detector for channel B is routed to the ICAPA pin. This allows us to trigger on rising and falling edges for the same signal.



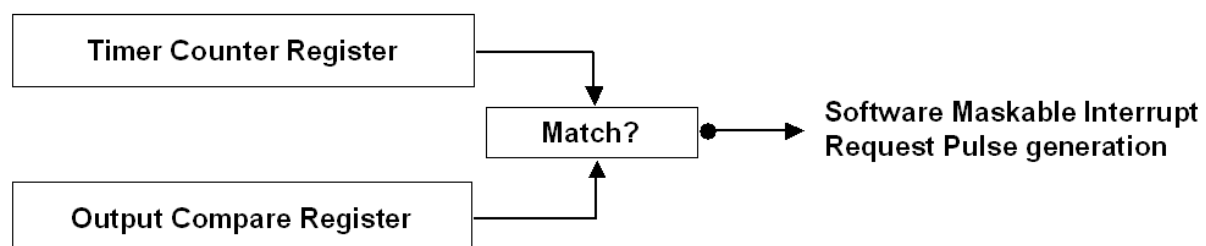
The PWM capture mode is synchronised on a pulse edge to reset the timer (Capture A). The next pulse edge (Capture A) will be captured into capture register B. This is the duty cycle. The next capture A event restarts the cycle and the capture A register contains the PWM period

In this mode the count for rising edge of the signal is stored in Capture Register A and the falling edge is stored in Capture Register B. Thus Capture Register B contains the pulse width and Capture Register A contains the signal period. By enabling the capture A interrupt we can read both these values on each cycle and reset the counter to start a fresh cycle.

```
TIM1_CR2 = 0x000008FF; // Enable interrupt on capture A event
TIM1_CR1 = 0x0000C004; // Timer enable, Enable PWMI mode,
                        // Capture A rising edge, capture B falling edge
```

4.5.3 Output Compare

In addition to the capture registers, each of the STR7 timers has two compare registers with matching output pins. When a match is made between the free running counter and the contents of the compare register, a compare event is triggered. This event can be used to generate an interrupt or change the logic on the compare pin. Once the timer is initialised and a compare value has been loaded into the Output Compare A and Output Compare B registers, the compare interrupts may be enabled by setting the OCAIE and OCBIE bits in Control Register two.



Each timer has two compare registers that may be used for custom pulse generation

Now whenever the free running timer matches the contents of either compare register, an interrupt will be generated. If you want to generate a waveform in addition to switching the GPIO pins to their secondary function, you must enable the output compare pins by setting the OCAE and OCBE bits in control register 1. Now whenever a compare event is made, the contents of the OVLVA and OVLVB bits will be applied to the output pins.

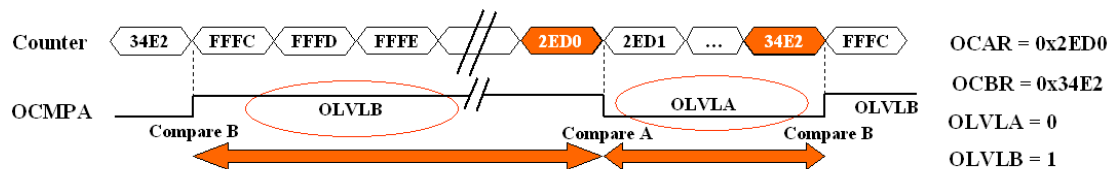
```

TIM1_OCAR = 0x0000FF00 // Set the compare count
TIM1_CR2 = 0x000040FF; // Enable the compare A interrupt and set the prescaler
TIM1_CR1 = 0x0008100; // Timer enable, set the output level on OCMPA pin

```

4.5.4 PWM Output

Like the PWM input mode, the STR7 timers have a special mode to allow easy generation of PWM output waveforms. By setting the PWM bit in Control Register 1 both of the compare registers are used to control the output compare A pin.



In PWM mode the timer starts from 0xFFFFC. A compare event on Compare A register forces the output compare A pin low. A compare match on compare B register forces the same pin high and reloads the timer with 0xFFFFC to restart the cycle.

At the start of a cycle the timer is reset to 0xFFFFC and the OCA pin is set high. The value loaded into the compare A register is the period of the pulse. When this match is made, the compare event loads logic zero stored in OLVA onto the output compare pin. The PWM period is stored in the compare B register. When this count matches the timer contents, the logic one value stored in OLVB is applied to the output pin and the timer is reset not to zero but to 0xFFFFC and the cycle is restarted. So in order to modulate the PWM signal, it is only necessary to write to output capture register A.

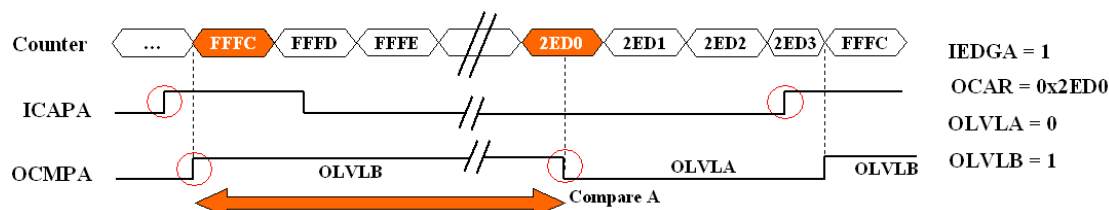
```

TIM1_OCAR = 0x00000080; // Set the capture A count ( pulse falling edge)
TIM1_OCBR = 0x000000FF; // Set the PWM period
TIM1_CNTR = 0x000000000; // initialise the count
TIM1_CR2 = 0x000008FF; // enable the capture B interrupt and set the prescaler
TIM1_CR1 = 0x00008150; // Timer enable,
                        // set output levels on compare A,
                        // enable compare A and PWM mode

```

4.5.5 One Pulse Mode

The STR7 timers have an additional useful mode that can generate a pulse that is triggered by an external signal.



One pulse mode uses the two capture registers to define the rise and fall edges of a pulse which can then be triggered by an input capture pin.

One-Pulse mode is enabled by setting the OPM bit in control register 1. In this mode the pulse generation is triggered by a capture event on ICAPA pin. This can be a rising or falling edge, which is programmed by the IEDGA bit. Once this event is triggered the timer will output a pulse whose characteristics are determined by the

two compare registers. When the trigger event occurs, the timer is reset to 0xFFFC and the contents of OLVLB bit is loaded onto output compare pin A. When the timer matches the contents of output compare register A, the logic level in OLVLB bit is loaded into output compare pin A and the timer then waits for the next capture event.

```
TIM1_OCAR    = 0x0000FF00 // Set the pulse length
TIM1_CR2     = 0x0000FF;  // Set the prescaler
TIM1_CR1     = 0x0008265; // Timer enable, set the pulse output level on OCPA pin
                        // enable one pulse mode, set capture edge
```

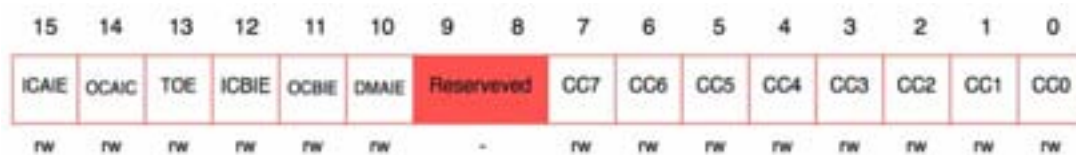
4.5.6 DMA Support

Each of the timer units has a dedicated DMA channel. This channel allows the capture and compare registers to act as a DMA source. The DMAS0 and DMAS1 bits in control register 1 allow you to select the source register for the DMA transfer. The source register can be either of the input capture registers (ICAPA, ICAPB) or the output compare registers (OCMPA,OCMPB).



Control register 1 allows you to select the source register for a DMA transfer.

Once the DMA source has been selected, the DMA transfer can be enabled by setting the DMAIE bit in control register.



Once the DMA source is selected the DMAIE bit in control register 2 enables the DMA transfer.

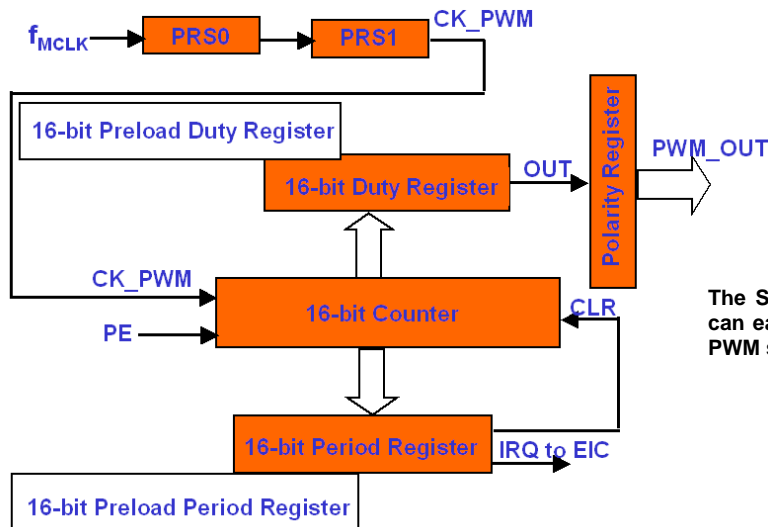
This configures the DMA channel in the timer unit. However the DMA transfer has to be fully configured, as discussed in Chapter 3.

Exercise 15: Timer

This exercise configures timer0 in PWM mode which is modulated by the output from the ADC. The PWM signal may be observed by an oscilloscope and the compare interrupt routines are used to modulate an LED in software.

4.6 PWM Module

In addition to the general purpose timer, the STR73x has a six dedicated PWM modules which may be used to generate a channel of PWM, of a single edge-aligned waveform.



The STR730 has six independent PWM units which can each generate a channel of single edge aligned PWM square wave.

The PWM module consists of a pair of prescaler registers that are used to divide down M_{CLK} to give the PWM timer tick frequency. This clock feeds a sixteen bit free-running counter which has two compare registers. One compare register holds the total period of the PWM signal and the second holds the duty period for the PWM pulse. The compare registers control the state of a dedicated PWM output pin. When the PWM timer is reset to zero this pin is forced high until the timer matches the contents of the PWM duty register. At this point it will be forced low. The timer will continue counting until it matches the contents of the period register. At this point the timer will be reset, forcing the pin high and the cycle will start over. After a hardware reset, the PWM module is disabled by having the count and duty registers initialized to zero. The resolution of the PWM signal is defined by the counter frequency. As each channel has its own dedicated counter it is possible to have a unique frequency and modulation for each PWM channel.

$$F_{ckpwm} = F_{mclk} / (2 \times \text{pow}(\text{PWM_PR0}) \times \text{PR1} + 1)$$

Once the resolution is known the PWM period can be independently set for each channel using the formula:

$$T_{pwm} = \text{PER} + 1$$

$$F_{ckpwm}$$

The duty cycle can be set as a percentage of the PWM period using:

$$DC_{pwm} = \frac{\text{PWMn_DUT}}{\text{PWMn_PER} + 1}$$

The module may then be initialised as follows

```

IOPORT2_PC0 = 0x01;           // Set IO port2.0 alternate function output
IOPORT2_PC1 = 0x01;
IOPORT2_PC2 = 0x01;

PWM0_PER    = 0xFFFF;         // Set the PWM period
PWM0_DUT    = 0x8000;         // Set the PWM Duty
PWM0_PEN    = 0x0001;         // Start the count
  
```

In addition the Output Polarity register allows the logic of the PWM signal to be inverted so that a match on the duty register sets the PWM pin and a match on the period register clears the pin.

```
PWM0_PLS = 0x00000001;           // Invert the PWM mark space periods
```

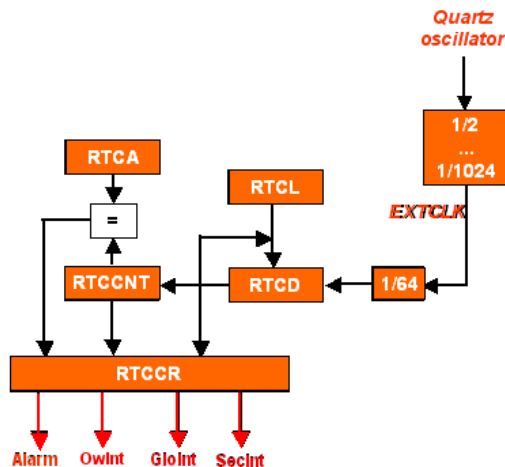
An interrupt may also be generated when a match is made with the duty register so that the PWM signal can be modulated under interrupt control. This also allows the PWM module to be used a simple periodic interrupt if the PWM function is not required.

Exercise 16: PWM

This exercise configures the PWM module and plays a tune on the buzzer on the evaluation board

4.7 Real Time Clock

The Real Time Clock peripheral consists of two main blocks. Firstly there is a prescaler which divides the external oscillator to give a seconds tick, this timebase is fed to the second block which is a set of timers and comparator registers that provide the clock and alarm functions.



The real time clock is a 32-bit counter which is incremented every second. A set of comparator registers provides an alarm function.

The RTC programmers interface consists of 10 registers as shown below. Like the other peripherals, these registers are accessed via the Advanced Peripheral Bus and are clocked by the system clock. However the RTC has its own dedicated clock source which is derived from the main oscillator. Because of these two separate clock domains there is a configuration sequence that must be followed to ensure the user registers are updated correctly.



The RTC user interface allows you to configure a prescaler to match the external clock source, set and read the time and set and read the alarm registers.

The prescaler load registers provide the count value that is used to divide the external oscillator to provide an accurate second tick. After reset the prescaler load registers are preloaded with the correct value to give a seconds tick for a 32.768KHz crystal. If you are using anything other than this frequency, the prescaler load registers must be reprogrammed with a divide value to match the external oscillator. However before any write can be made to the prescaler, alarm or counter register the RTC must first be placed in configuration mode. This is done by two bits in the control register



To configure the RTC registers you must set the CNF flag in the control low register. A write to the RTC registers is only complete when the RTOFF flag returns high.

Before entering configuration mode you must check that any write to the RTC registers has been completed. This is done by checking that the RTOFF flag is high. If this is the case, you can then set the CNF flag to enter configuration mode. In configuration mode we can modify the prescaler if necessary, zero the count current count and set an alarm value. Once configuration is complete the CNF bit must be cleared to make the RTC active.

The Real Time Clock has one interrupt channel to the EIC but three internal interrupt sources. An interrupt can be generated each time the RTC counter is incremented to give a nominal one second interrupt. An interrupt can be generated when the RTC counter overflows or after 2³² seconds. An alarm interrupt can also be generated when the contents of the Alarm register match the Counter register. To enable an interrupt source you must set both the specific interrupt enable bit and the global interrupt enable bit .

```

RTC_CRL |= 0x00000010;           // Set RTC into config mode
while(!(RTC_CRL & 0x20));

RTC_CNTL = 0x00000000;           // Set the count to zero
while(!(RTC_CRL & 0x20));

RTC_CNTH = 0x00000000;           // Set the count to zero
while(!(RTC_CRL & 0x20));

RTC_CRH |= 0x00000001;           // Enable the seconds interrupt
while(!(RTC_CRL & 0x20));

RTC_ALRL = 0x0000000A;           // Set an Alarm for 10 seconds
while(!(RTC_CRL & 0x20));

```

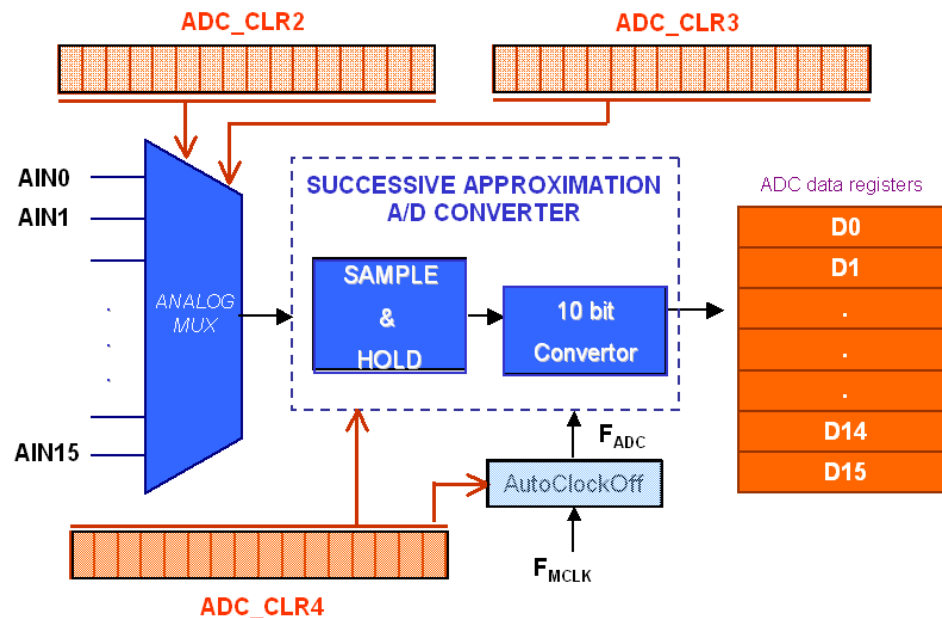
```
RTC_CRH |= 0x00000001;           // Enable the alarm interrupt
while(!(RTC_CRL & 0x20));

RTC_CRL &= 0x000000EF;           // Set RTC into running mode
while(!(RTC_CRL & 0x20));
```

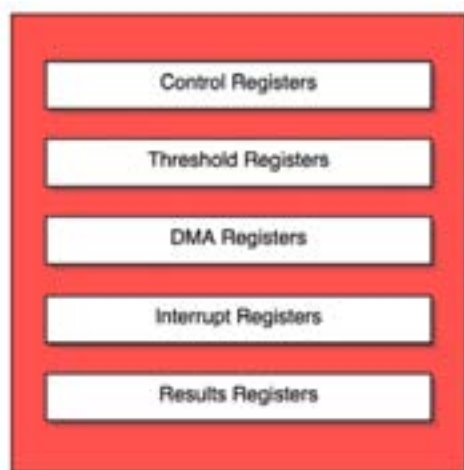
The Real Time Clock will remain active during the power down modes but care should be taken when reading the RTC registers immediately after coming out of a power down mode. Because the real time clock has its own oscillator, the registers on the APB2 bus will not be updated for one RTC clock tick after coming out of power down. You must ensure that your application code does not read the registers for 31.25 usec after resuming from a power down mode.

4.8 Analog To Digital Converter

The STR73x has an on-board Analog to Digital Converter. This converter has maximum resolution of ten bits with sixteen multiplexed input channels. Each with its own results register. All conversion results are guaranteed to be monotonic. The ADC has a prescaler which divides down M_{CLK} to run the converter at 10MHz. Each analog to digital conversion takes a minimum of 40 cycles or can be made a rate of 250K conversions per second.



The analog to digital converter is a 16 channel 10 bit converter. It supports single channel conversion or a continuous round robin “scan” conversion of selected channels. An additional “injection” mode allows interruption of the scan mode to read selected channels on demand.



The ADC registers can be split into five main blocks. These consist of the control configuration registers, threshold registers for the analog watchdog, DMA and interrupt configuration registers and a dedicated results register for each channel

The conversion time is split between the sample-and-hold circuit and the actual conversion. Each of these sections may be clocked at different rates so it is possible to have a slow clock on the sample-and-hold and then do a fast conversion. This allows you to tune the ADC characteristics to the external analog circuit. Once configured the ADC has a number of conversion modes. There is a basic one-shot mode which makes a single conversion of selected channels when triggered by software. A scan mode will make continual round-robin conversion of the selected channels. The scan mode also incorporates an injection mode which halts the chain of scan conversions then performs a one-shot conversion on a group of selected channels and then resumes the scan conversion. Finally four of the analog channels may be used as analog watchdogs. In this mode the

analog signal is monitored and if it strays outside a defined maximum or minimum range, a warning interrupt may be generated.

4.8.1 Configuration

Once the ADC clock has been enabled in the PRCCU, the prescalers for the sample-and-hold and the separate converter circuitry must be configured.



Control register 1 is used to configure the prescalers for the sample-and-hold circuit and the ADC.

Both prescalers are configured in control register 1. Both prescalers divide down Mclk and must provide an output to the ADC which is a maximum of 10 MHz. If the sample-and-hold prescaler is set to zero the main ADC clock will be used for both the sample-and-hold and the converter circuits. The sample-and-hold process takes 10 clock cycles and the main converter takes a further 30 cycles. The full conversion time may be calculated by the formula:

$$T_c = 10 \times (Mclk+1) \times SMPP + 20 \times Mclk \times CNVP$$

Where: SMPP = sample and hold prescaler factor
 CNVP = Main ADC clock prescaler factor
 Tc = Total conversion time

You must ensure that the conversion frequency for both the sample and hold and the conversion circuits must not exceed 10 MHz

$$10 < Fmclk/SMPP \quad \text{hence at 36MHz SMPP and CNVP} \Rightarrow 4$$

$$10 < Fmclk/CNVP$$

Once the prescalers have been configured the ADC may be placed into a self-calibration mode. This is done by setting the CAL bit in Control Logic register 0.



Control register 0 is used to start the ADC self calibration procedure and also start the ADC conversions

The calibration time may be shortened by around 16 cycles by disabling noise filtering. This is done by setting the NOAVRG bit in control register 4. During the calibration procedure the CAL bit stays high and will be cleared once calibration is finished.

The ADC is then ready to begin conversions in either one shot, scan injection or watchdog modes.

```
ADC_CLR4 = 0x0000000;           // Clear the powerdown bit
ADC_CLR1 = 0x00000020;          // Set ADC clock 5.3MHz
ADC_CLR0 = 0x00000002;          // Start calibration
while (ADC_CLR0)                 // Wait till end of calibration
{
    ;
}
```

4.8.2 One-Shot Mode

The ADC conversion modes are configured in Control Register 2. The two basic conversion modes, one-shot and scan are selected with the Mode bit and the channels to be converted are defined in the NCH and FCH fields.



The ADC conversion modes are configured through the NCH and FCH fields in Control Register 2.

To enter one-shot mode, the mode bit should be set to zero and the number of the first channel to be converted placed into the FCH field. The number of channels to be converted is placed in the NCH field and finally the start bit in Control Register 0 is set. One conversion will be made on each of the channels and at the end of the chain of conversions, the start bit will be cleared

```
ADC_CLR2 = 0x0000000;           // Single conversion ch:0
ADC_CLR0 = 0x0000001;           // Start conversion

while(!(ADC_PBR & 0x00000002)) // Wait for end of conversion
{
    ;
}
```

4.8.3 Scan Mode

If the mode bit in Control Register 2 is set to 1 the ADC will operate in scan mode. In this mode, the ADC will repeatedly perform the defined chain of conversions until it is halted by clearing the START bit in Control Register 0. The start bit will actually read zero when the last conversion has been completed.

```
ADC_CLR2 = 0x00081C0;           // Chain conversion start ch:0 convert 8 channels
                                   // Scan mode
ADC_CLR0 = 0x0000001;           // Start conversion
```


4.9 Injection Conversion

If the ADC is operating in scan mode it is possible to interrupt the chain of conversions and perform a single one-shot chain of conversions on a different set of channels and then resume the original scan mode. This is called an injection conversion and is defined in Control Register 3



The application software can interrupt a scan conversion to perform a series of conversions defined by the JNCH and JFCH fields in control register 3. This is called injection mode.

In Control Register 3 the JFCH and JNCH define the starting channel number and the number of channels to convert in the same way as the FCH and NCH fields in Control Register 2. Once the injection conversion has been defined, the conversion chain can be started by setting the JSTART bit .

```
ADC_CLR3      =      0x000001C8;    // Injection conversion from CH:8 for 8 channels
ADC_CLR3      |=      0x00008000;    // Start the injection conversion
while(!(ADC_PBR & 0x00000004))
{
;
}
```

4.9.1 Analog Watchdog

In addition to the conversion modes the ADC has an “analog watchdog” mode that allows channels 0 – 4 to be guarded with maximum and minimum values.



The analog threshold registers define maximum and minimum results values for an ADC channel. If the ADC result is out of range an interrupt can be generated.

Each of the analog watchdogs has a dedicated maximum and minimum threshold register which contains a 10-bit user-programmable value. The watchdog is then linked to a specific ADC channel by programming the THRCH bits. The watchdogs may only be connected to the lower four analog channels. Once configured the THREN bit enables the watchdog. At the end of each conversion an active watchdog will check the conversion result against the threshold registers can generate an interrupt if the conversion value is out of range.

```
ADC_TRA0 = 0x000000FF;    // Set threshold high at 0x00ff ch:0
ADC_TRB0 = 0x0000800F;    // Set low threshold 0x000f Enable watchdog
ADC_IMR  = 0x00000030;    // Enable threshold interrupts
```

4.9.2 Interrupts

The ADC has a single IRQ interrupt channel connected the EIC. Internally the ADC can raise an interrupt at the end of each conversion or at the end of each chain of conversions in one-shot, scan and injection modes. Additionally, as mentioned above, the analog watchdogs can generate an interrupt if the analog signal goes outside of a user-defined range.



The interrupt pending register has status bits for each conversion mode. Each analog watchdog channel has two status bits which indicate if the high or low threshold has been crossed.

When an interrupt occurs, the pending bit register contains the status flags for each of the interrupt sources. Like the status registers in the other peripherals, this register must be read to clear the flags and cancel the interrupt. The flags for each of the analog watchdogs consist of two bits, which allow you to determine if the upper or lower threshold has been crossed. If the upper bit is set the upper threshold has been crossed; if the lower bit is set the lower threshold has been crossed.

```
void ADC_IRQ_isr ( void ) __attribute__ ((interrupt ("IRQ")));

void ADC_IRQ_isr ( void )
{
    SWITCH_IRQ_TO_SYS;

    /* BEGIN USER CODE XTI_IRQ */
    switch ( ADC_PBR & 0x00000030)    // read the threshold 0 bits
    {
        case ( 0x00000010):
            // Threshold low error
            break;

        case (0x00000020):
            //Threshold high error
            break;

        default:
            }
    }

    /* END USER CODE ADC_IRQ */

    SWITCH_SYS_TO_IRQ;
    // Reading the ADC_PBR clears the status flags
    EIC_IPR0 = CHANNEL(62); // clear the EIC channel
}
```

4.9.3 DMA Support

In addition to the interrupt support, the ADC has a dedicated DMA channel which can be programmed to make a DMA transfer from the ADC results register to a memory or peripheral register at the end of each conversion. The DMA conversions are enabled within the ADC by selecting the required channels in the DMA Enable register and then setting the DMAEN bit in the DMA global enable register.



The DMA Global Enable register switches on the ADC DMA support. The channel number of the first result transferred is stored in the DENCH bits.

If more than one analog channel has its DMA mode enabled the channel number of the first DMA transfer is stored in the DENCH bits. This can be used to synchronise the receive buffer to the incoming data. In addition to enabling the DMA support in the ADC, the DMA unit must be configured to make the transfers, as discussed in Chapter 3. The code below demonstrates a full DMA transfer for the ADC that takes an ADC result and transfers it to an IO port, without any CPU overhead.

```
ADC_CLR4      = 0x00000000;      // Clear the ADC powerdown bit
ADC_DMAE      = 0x00008000;      // Enable the ADC DMA support
ADC_DMAR      = 0x00000007;      // Enable channel 0-2 DMA transfer

ARB_PRIOR     = 0x00000003;      // Enable DMA arbitration
DMA3_SOURCEL0 = 0xF850;          // Set the source address
DMA3_SOURCEH0 = 0xFFFF;          // ADC channel 0 results register
DMA3_DESTL0   = 0x0800;          // Set the destination address
DMA3_DESTH0   = 0xA000;          // 0xA0000800
DMA3_MAX0     = 0x3;             // Set the transfer size to 3 data units
DMA3_CTRL0    = 0x28F;           // Enable the transfer the transfer

ADC_CLR1      = 0x00000020;      // Set ADC clock 5.3MHz
ADC_CLR0      = 0x00000002;      // Start calibration
while (ADC_CLR0)                // Wait till end of calibration
{
    ;
}
ADC_CLR2      = 0x00008080;      // Scan mode conversion ch:0 three channels
ADC_CLR0      = 0x00000001;      // Start conversion
```

4.9.4 Low Power Operation

For low-power designs it is possible to fully power down the ADC if you are not using it by setting the PWDN bit in Control Logic register 4



The ADC is designed to support low power designs. In control register 4 the ADC can be fully powered down and the ACKO bit can enable automatic switch off of the ADC clock when the converter is idle.

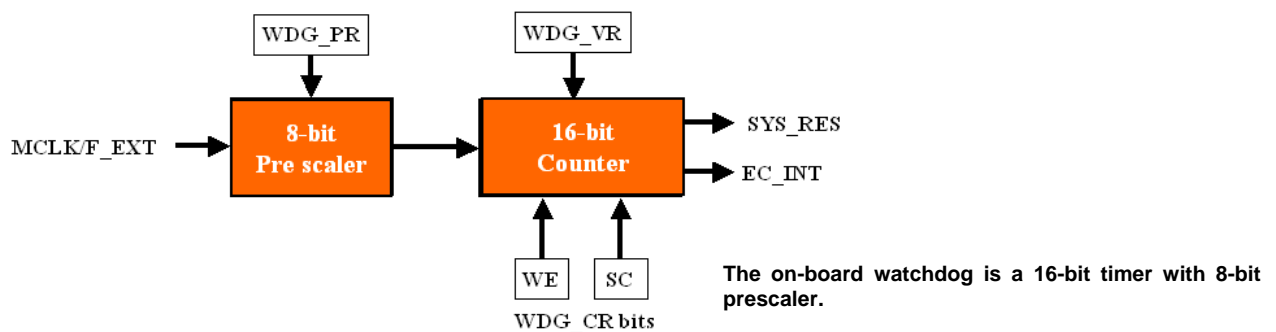
In addition to powering down the ADC, setting the ACKO bit will automatically disable the ADC clock whenever the converter is idle.

Exercise 18 : Analog to Digital converter

This exercise enables the ADC in round robin mode and writes the conversion result to the LEDs on the evaluation board

4.10 Watchdog

In common with most microcontrollers the STR7 has an on-board watchdog that provides a hardware recovery mechanism in the event of the application software crashing, or being disturbed by a hardware failure such as a power brownout.



However if you do not require a this form of protection in your application, the watchdog may be used as free running timer that can generate a periodic interrupt for an operating system or scheduler, freeing up one of the more complex timer blocks.



The watchdog interface allows the watchdog to be configured as an on-chip watchdog or periodic timer with an end of count interrupt.

The watchdog timer is a 16-bit count down counter with an eight bit prescaler. The prescaler and watchdog counter reload have dedicated registers and the timeout period may be calculated with the following formula:

$$\text{Timeout in micro secs} = (\text{Prescaler} + 1) \times (\text{reloadvalue} + 1) \times \text{Tpclk2}/1000$$

Once the timer registers have been configured, the watchdog may be started by setting the watchdog enable bit in the control register. Once this bit is set there is no way other than a hardware reset to disable the watchdog. Unlike the other peripherals you cannot stop the peripheral clock to the watchdog in the PRCCU. To stop the watchdog from timing out and causing a reset you must write to the watchdog key register. This forces a reload of the watchdog timer. To ensure that the application software is still running properly, the watchdog reload will only occur if the value 0xA55A followed by 0x5AA5 are written consecutively to the Key Register. It is up to your application software to ensure that the Key register is written frequently enough to stop the watchdog from timing out. If the watchdog does timeout, a reset will be forced in the STR7 and your code will once again start from the reset vector. However the watchdog status register contains an end-of-count flag that is set if a timeout occurs. This flag may only be cleared by software so it is possible to tell if the STR7 is starting from a hardware reset or a watchdog timeout.

```
WDG_PR =      0x000000FF ; // Set maximum divide on the prescaler (Reset value)
WDG_VR =      0x0000FFFF ; // Set maximum timeout ( Reset value)
WDG_CR =      0x00000003;  // Start the count and enable watchdog
while(1)
{
    WDG_KR      = 0x0000A55A; // Refresh the watchdog
    WDG_KR      = 0x00005AA5;
}
```

If your application does not need to use the watchdog for software protection it can be used as an additional 16-bit timer, which can generate a periodic interrupt. If the watchdog enable bit is not set in the control register the counter will count down to zero and can generate an interrupt provided that the ECM bit is set in the Watchdog Mask register. When the counter reaches zero the interrupt is generated and the watchdog counter is reloaded to begin the next countdown.

```
WDG_MR      =      0x00000001;          // enable the IRQ interrupt
WDG_CR      =      0x00000002;          // start the count
```

The only other register available in the watchdog peripheral is the Counter register which is a read-only register containing the current count of the watchdog timer.

Exercise 18: Watchdog

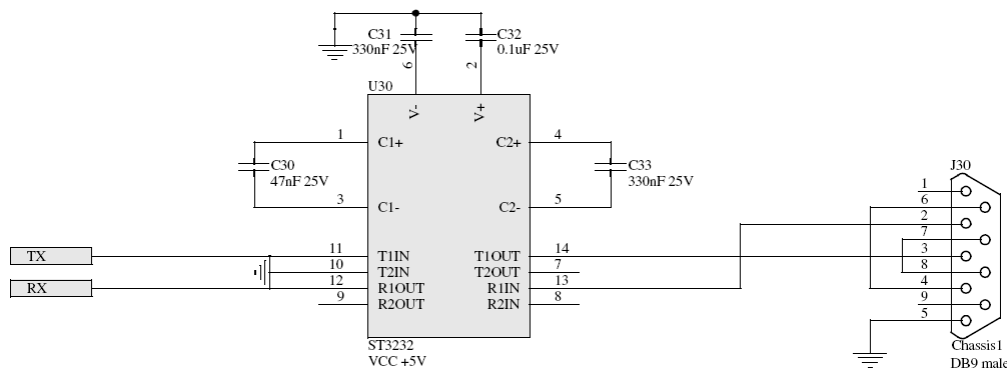
This exercise configures the watchdog as a periodic timer and enables the end of count interrupt. Each interrupt is used to drive LED arrays on the evaluation board.

4.11 Communications Peripherals

All the remaining peripherals within the STR730 are the serial communications peripherals and there are a lot of them. These include four UARTS, three SPI interfaces, two I2C interfaces and up to three CAN controllers. For the remainder of this chapter we will have a look at each in turn.

4.11.1 UART

Most small microcontrollers have at least one UART and some have two. However the STR7 is almost unique in having four UARTs as peripherals. As we will see later, the buffered SPI port can also be used as an additional UART. These features combined with either the CAN SPI or I2C peripheral to make the STR7 the ideal hardware for a protocol converter or RS232 concentrator.



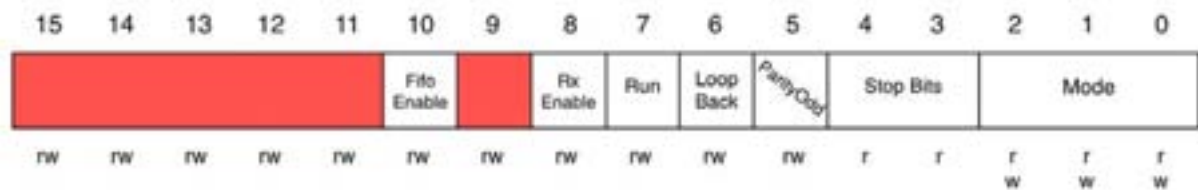
On the evaluation board, each UART is brought out to a 9 way D type, with RS232 level driver.

The four UART peripherals each support full-duplex asynchronous communication and have the same register set. Thus code that runs on UART one can be reused for the either of the other two. Each UART contains its own baud rate generator in the form of a 16-bit timer. There are also two FIFO buffers, each 16 words deep, for the transmit and receive data. The UART data format is configurable to be eight or nine bits long, with various combinations of start, stop and parity bits.



One pulse mode uses the two capture registers to define the rise and fall edges of a pulse which can then be triggered by an input capture pin.

The principle features of each UART are configured in the UART control register.



The UART control register configures the operating mode and data format used for serial communication. The control register may enable the USART FIFO and receive functions as well as selecting the loopback test mode.

After reset, the UART is disabled as the RUN bit is set to zero. The UART must be configured and then enabled by setting the RUN bit to one. The bit fields in the control register allow us to configure the format of the serial data word (seven eight or nine bits long) select the logic of the parity bit if it is being used, configure the length of the stop bit and enable reception and the UART FIFOs. Once the UART has been configured we must set the baud rate by programming the BAUD rate register. This register is in effect the reload register for the dedicated 16-bit timer within the UART. The timer must be configured to generate an overflow tick at 16 times the desired baud rate. Hence the reload value can be determined by the following calculation.

Reload value = $MCLK / (16 \times \text{BAUDRATE})$

When selecting an external oscillator you should remember that a typical RS232 connection to a PC can tolerate about a 5% error. However the bit timing for the CAN peripheral must be exact. So if you are going to use both peripherals, ensure that you select the oscillator frequency to give an accurate CAN bit timing and take the error on the RS232 Baud rate. Once the UART has been configured and the baud rate set, you must finally configure the GPIO pins to their alternate function before setting the run bit to enable the UART. If you are using the FIFO mode it is also necessary to reset the TX and RX FIFOs prior to transmitting or receiving the first character to ensure the FIFO buffer pointers are correctly initialised. This is done by writing any value to the FIFO reset registers. Either FIFO can be reset at any time placing the FIFO pointer to zero and destroying the contents of the reset FIFO.

```

UART0_CR      = 0x00000001 ; // Reset UART
UART0_RXRSTR  = 0x00000001 ; // Clear Transmit and Receive Buffers
UART0_TXRSTR  = 0x00000001 ;
UART0_BR      = 0x00000021 ; // Set Baudrate to 9600 Bps
UART0_IER     = 0x00000005 ; // Enable TX Half empty and RX full
                                // interrupts
UART0_CR      = 0x00000509 ; // Normal mode 8 N
UART0_CR      |= 0x00000080 ; // Start the UART running

```

Once the UART is configured, data can be written and received via the transmit and receive buffers. Simply writing data into the TX buffer will place the data into the UART transmit shift register which asynchronously transmits the data over the serial connection in the format defined by the configuration register. A common error is the write more data to the TX buffer before the transmission is complete. This will in effect crash the buffer and corrupt the data currently under transmission. Before writing fresh data you, should poll the UART status register and check that the TX full bit is clear before writing to the transmit register.

```

void put_char ( char character )
{
    while ( UART0_SR & UART_TXFULL )    // check if transmit buffer is free
    { ; }
    UART0_TXBUFR = (unsigned char)character; // write character
}

```

If the FIFOs are enabled, the data will be placed in 16 word-deep buffer and each data word will be transmitted in turn. In this case, the TX full bit in the status register will indicate that there are 16 pending words in the FIFO and no further characters should be written. There is also a half empty flag (or half full depending on your viewpoint) that indicates when the TX FIFO contains eight characters or more.

When a character is received it is copied into the RXbuffer register and the RX full bit in the status register is set. If the FIFOs are enabled, a 16 word-deep receive FIFO, similar to the TX FIFO, is enabled. Any received characters will be written into the FIFO, where they may be read from the RXbuffer, in received order. The RX full bit will be set whenever there are any characters in the receive FIFO. There is also a RX half full flag in the status register which is set whenever there are eight or more characters in the RX FIFO.

```
char character get_char ( void )
{
    while ( !(UART0_SR & 0x00000001) )    // wait until RX FIFO contains a character
    { ; }
    UART0_TXBUFR = (unsigned char)character; // write character

    return (character);
}
```

Each UART has a single interrupt channel connected to the EIC. However internally the UART has nine interrupt sources connected to this one channel, which may individually be controlled in the interrupt enable register.

Three of the interrupt sources are used to detect errors that have occurred on the serial line. These include parity error, framing error and overrun error. The remaining interrupt lines are used to make the most efficient use of the transmit and receive FIFOs for maximum character throughput.

The transmit interrupts may be used with or without the FIFOs enabled. However if you are trying to achieve maximum transmission rate it is best to use have the FIFOs enabled. In this case you would need the TX half empty and TX empty interrupts enabled. The transmission would begin by filling the TX FIFO with 16 characters and then waiting for the half full interrupt. This would allow a further eight characters to be written to the TX FIFO. This would ensure a continual back-to-back transmission of characters, with minimal software overhead. Once all the necessary characters had been written into the TX FIFO, the final sixteen characters would be sent and the end of interrupt would be signalled by a TX empty interrupt.

It is also possible to use the TX interrupts when the FIFO is disabled. In this case it is important to understand at what point in the transmission cycle each interrupt is generated. The TX half empty interrupt is generated when the character is moved from the TX buffer into the TX transmission shift register. The TX empty interrupt is generated as the stop bit is about to be transmitted. You can use either interrupt source to reload the TXbuffer. The TX half empty will give back-to-back transmission of characters as the TX buffer can be reloaded while the previous character is still being transmitted.

The receive interrupts operate much the same way as the transmit interrupts. The RXbuf full and RXhalf full flags can both generate an interrupt. The RXbufFull interrupt will generate an interrupt every time a character is received and is best used when the FIFOs are disabled. This allows the CPU to read the RXreceive register each time a character is received. If data is arriving back-to-back, this method will be too slow to cope with all the traffic, especially at high bit rates. For high bandwidth serial links, it is best to enable the FIFOs and use the RX half full interrupt. This allows you to receive eight characters into the FIFO before an interrupt is generated. Once the interrupt is generated, you have a further eight characters worth of time to read the data out of the FIFO before it overflows.

This method has a problem in that if you receive less than eight characters, no interrupt is generated and the data remains in the FIFO. For this reason the UART has a receive timeout register. This is a counter with a programmable time delay that will generate an interrupt if there is data in the RX FIFO and its timeout has expired. Each time the FIFO is read, the timer is reloaded so it acts as a watchdog for the receive FIFO and ensures you will catch the remaining few characters of a transmission that have not triggered the RXhalf full interrupt. The UART timeout register is a 16-bit counter, similar to the baud rate generator and should be set to 1 ½ bit periods, using the following formula:

$$\text{Timeout} = \text{MCLK} / \text{BAUDRATE}$$

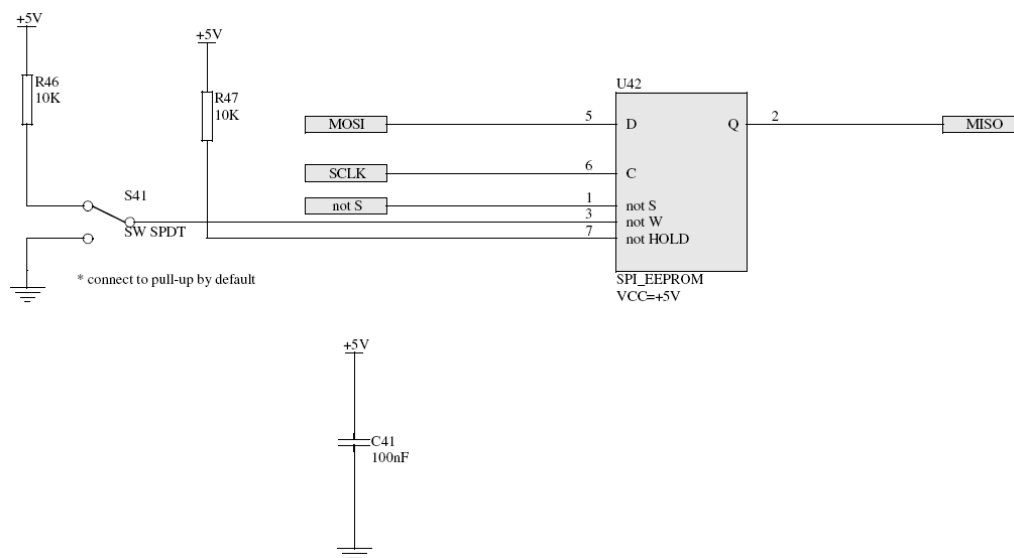
Exercise 19: UART

This exercise configures the UART0 and enables the interrupts to echo back data sent from a terminal on the PC using the UART FIFO buffers

4.12 Buffered SPI

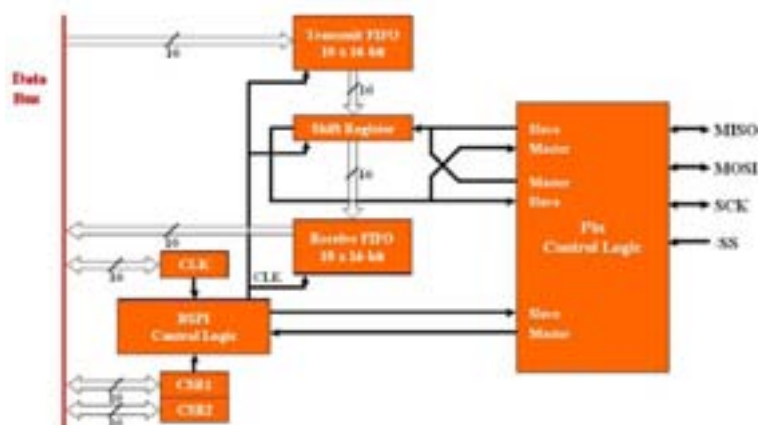
The STR7 has three fully independent SPI peripherals that support full duplex communication at speeds up to 6 Mb/sec. The SPI peripherals provide a fast communications channel for serial communication between the STR7 and other devices on the same circuit board with the STR7 configured as a master or slave device. The SPI protocol is a master slave protocol designed for point-to-point communication between two devices. Each SPI slave device has a slave select pin that is used to enable a selected device prior to communication from the master. If there are multiple devices on the SPI bus, the master is responsible for ensuring that only a single slave device is selected before communication begins. Once a slave is selected data can be shifted out from the master with a clock provided by the master. Since the SPI channel is a full duplex channel as the data from the master is shifted out a word of data from the slave is shifted into the master. Only data is transferred on the SPI bus so there is no protocol overhead making for a fast, easy-to-use serial connection.

Like the UARTS, both SPI peripherals are exact duplicates so code developed to run on one SPI peripheral will also work on the other peripheral.



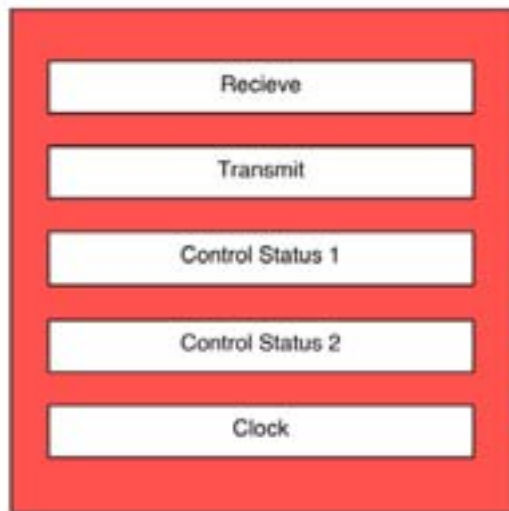
External SPI devices may be easily interfaced to either BSPI peripheral. The master slave select pin should be pulled high during operation.

The SPI peripherals have a standard four-wire interface to the SPI bus that consists of the Master-in, Slave-out (MISO) and Slave-out, Master in (MOSI) data bus, a serial clock pin and a slave select pin.



The STR730 contains two buffered SPI peripherals. Each SPI peripheral contains its own baud rate generator and 10 word deep FIFOs for receive and transmit.

Each SPI peripheral is controlled by five special function registers located in the APB peripheral block.

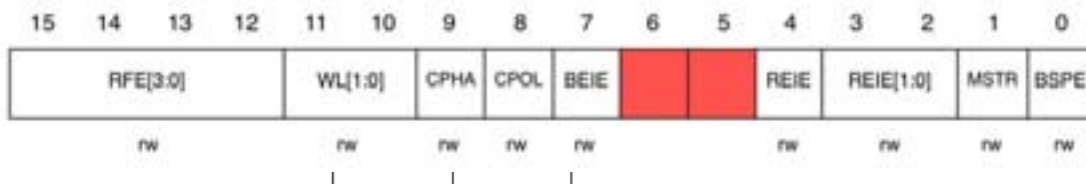


The SPI interface consists of only five registers. Its performance is enhanced by TX/RX FIFOs and DMA support

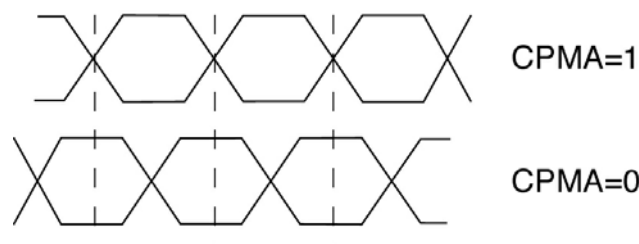
After a hardware reset the SPI peripheral will be configured as a slave device. If you intend to use it as a master device you must configure the Master Clock Divide register, prior to switching from a slave to a master device. This divides down the APB clock to give the SPI clock. The value programmed into the clock divide register must be even and a minimum of six in master mode and a minimum of eight in slave mode.

SPI bit rate = $APB/BSPI_CLK$

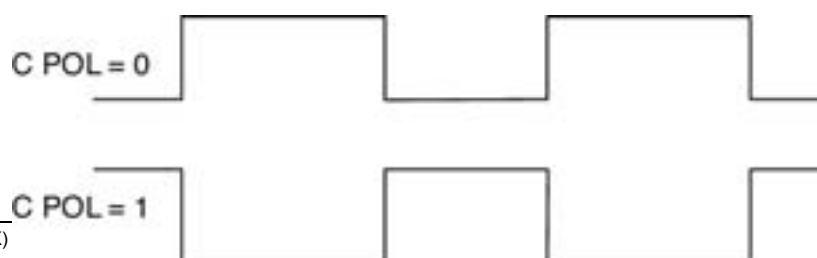
The remaining configuration bits are in the Control/Status Register_1 register. Here the BSPI peripheral can be configured to match the transmission characteristics for the other devices on the bus.



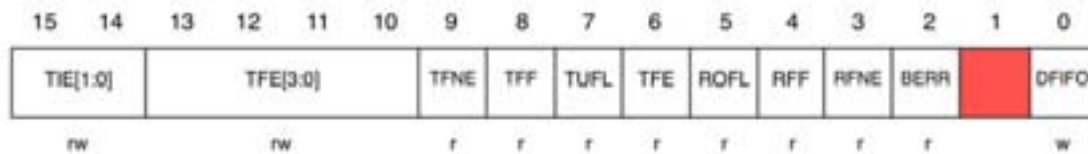
Control status register 1 configures the operating mode of the BSPI peripheral and sets the clock parameters to match those of the external SPI device. Here you can also set the word length and enable the FIFOs.



The SPI protocol allows for different clock phase and polarities. If you are using master mode, the CPHA bit allows you to select the clock phase to capture data on the leading or trailing edge of the clock and the CPOL bit allows you to define the clock polarity as active high or active low



After configuring the clock characteristics the word length for data transmission can be selected. The STR7 SPI peripheral is particularly suitable for high-speed data transfer.



Control Register 2 contains the BSPI status flags and interrupt enable bits. The DFIFO bit is used to reset the FIFOs and should be used during the BSPI initialisation process.

Firstly most microcontrollers only support transmission and reception of eight bit words. The STR7 SPI peripheral supports both an 8- or 16-bit word length. Secondly both the receive and transmit shift registers have FIFO buffers of up to ten words deep. Both these features are configured in status/control registers. Finally MSTR bit allows you to select master or slave mode and the BSPE bit enables the SPI peripheral for operation.

```
PCU_BOOTCR    = 0x00000004;           // enable SPI0 oscillator
for (delay = 0; delay < 0x200000; delay++); // wait for the clocks to stabilise
BSPI0_CLK     = 0x3C;                 // set clock divider 30 MHz/60 = 500000
BSPI1_CSR1    = 0x01;                 // enable SPI1
```

4.12.1 SPI Master Mode

Once the SPI peripheral has been configured in master mode, data can be transmitted. When the transmission starts, the slave select pin will go low to enable the external SPI peripheral for the duration of the transmission. If you have more than one external peripheral then additional GPIO pins must be used to select the desired peripheral.

Transmission is actually started by writing into the TX data register. Writing into the lower half word of this register will transfer data into the TX FIFO where it will be queued for transmission or if the FIFO is disabled it will be transferred directly into the SPI transmit shift register. If you are using an eight bit word the data is left justified so you must shift the data eight bits to the left so that it is located in the upper byte of the half word.



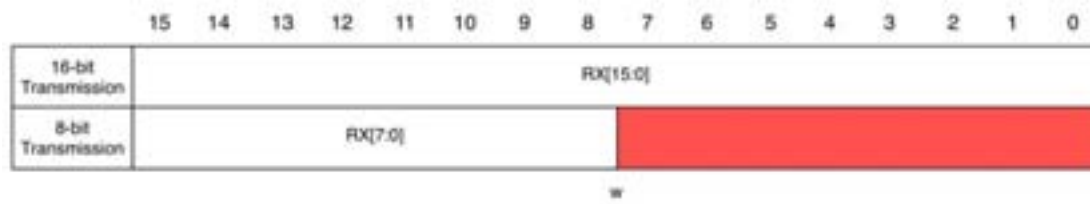
A 16-bit half-word may be written directly to the BSPI TX register. However for 8-bit operation the data must be shifted 8-bits to the left.

In the control registers there are flags for Transmit FIFO full, Transmit FIFO empty, Transmit FIFO not empty all of which can be used to control the flow of transmitted data. There is also a transmit underflow flag which is set when the transmit FIFO is empty and the transmit shift register has finished sending data. A transmit interrupt can be generated when either the FIFO full, FIFO empty or FIFO underflow flag is set.

```
BSPI0_CSR1    |= 0x02;           // set master mode
BSPI0_CSR2    |= 0x01;           // clear the FIFO

IOPORT2_PD    &= 0x7FFF;         // Pull a GPIO line used as a slave select line low
BSPI0_TXR     = 0xF900;         // Write to the TX buffer
```

As data is shifted into the SPI peripheral it will be transferred into the receive FIFO and can be read directly from the receive register. Like the TX register, the data can be read from the lower half-word if using 16-bit mode or the upper byte of the lower half-word if using 8-bit mode.



A 16-bit half word may be read directly from the BSPI RX register. However for 8-bit operation the data must be shifted 8-bits to the right.

The control registers contain flags for Receive FIFO full and Receive FIFO empty conditions, as well as Receive FIFO overflow, which is set if the CPU has not read data from a full FIFO and more data has been received. A receive interrupt can be enabled for either the FIFO not empty, or FIFO full condition.

4.12.2 SPI Slave Mode

In slave mode the slave select and serial clock pins act as an inputs to the SPI peripheral. When the slave select pin is pulled low the data bus will be activated and the SPI bus clock will provide the peripheral clock. Data on the bus will be shifted into the RX data register and any transmit data in the TX FIFO will be shifted out to the SPI bus master. The same transmit and receive interrupt structure is available in slave mode and it is up to the CPU to respond to the bus master in a timely fashion.

```
BSPI1_CSR1 = 0x01;           // enable SPI1
BSPI1_CSR1 |= 0x04;         // enable receive interrupt

BSPI1_CSR2 |= 0x01;         // clear the FIFO

BSPI1_TXR = 0xAF00;         // load the TX buffer, this will be sent when a master
                             // communicates to this SPI device
```

4.12.3 SPI Error Interrupts

In addition to the transmit and receive interrupts, the SPI peripheral has a bus error interrupt. In master mode this interrupt will be generated if the slave select pin is pulled low by another device. This indicates that there is a second master on the bus and that there is a bus contention.

4.12.4 DMA Support

Each of the three BSPI peripherals has a dedicated DMA channel. This channel can be used to send or receive data to and from the BSPI peripheral, depending whether it is in transmit or receive mode. Control Register 3 is used to configure the DMA channel within each DMA channel.



The BSPI peripheral supports DMA transfers to and from the transmit and receive registers. The DMA support is enabled within Control Register 3.

The DMA channel can be configured to burst transfer blocks of data between the peripheral registers and memory. The burst sizes are configured in the RBURST_LEN and TBURST_LEN fields and can be configured to transfer packet sizes of 1, 4, 8 and 16 words where the word size corresponds to the SPI word length configured in Control Status register 1. Since the data is stored in the transmit and receive FIFOs, the FIFOs must be enabled with a size that matches or exceeds the DMA packet size. Once the burst length has been selected you must enable DMA transmit and receive transfers with the RREQ_EN and TREQ_EN bits, as required, and then globally enable the BSPI transfer by setting the DMA_EN bit. In addition, the DMA channel must be configured as discussed in Chapter 3. The code below demonstrates how to configure a BSPI transmit and receive DMA transfer

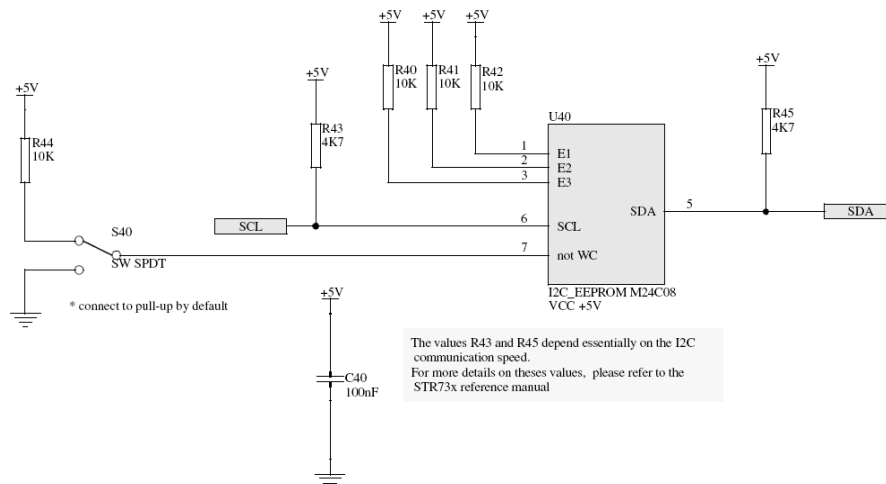
If you are using a burst transfer size of greater than one word you must be aware that a DMA receive transfer will only be made when the FIFO fills up to the burst size. For example, if the burst size is four words and seven words are received into the FIFO, one DMA transfer will be made and three words will remain in the FIFO until another word is received and the burst transfer size is reached.

Exercise 20: BSPI

This example connects the two BSPI peripherals together and configures them as master and slave. The master is then used to initiate the transfer of data between the two peripherals.

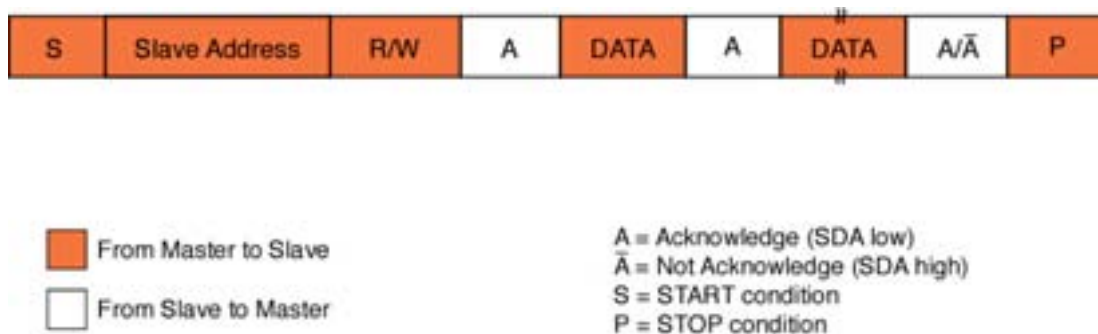
4.12.5 The I2C Module

The STR7 has two I2C communications peripherals. Like the UARTs and the SPI peripherals the two I2C modules are both fully independent modules and are code-compatible. The I2C modules support I2C communications up to 400KHz and can operate in both master and slave mode. The I2C peripheral also supports multi-master communication. Each I2C peripheral is interfaced to other I2C devices by two pins. One of these pins is used to carry the serial clock and the other the serial data.

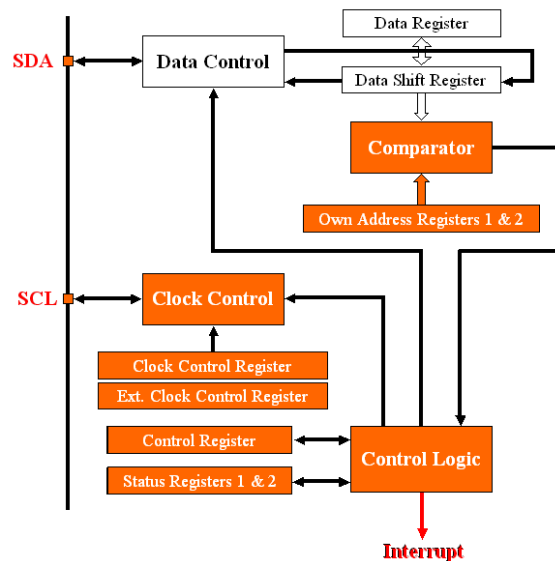


Like the SPI bus the I2C bus allows easy interfacing to external devices.

Like the SPI protocol I2C is a master slave protocol. In a typical system a single master controls all communication and the slaves can only respond to its commands. A typical I2C bus transaction is shown below.



The transaction begins with the master generating a start condition, followed by the address of the node which is to be the destination of the message data. The I2C protocol uses a 7-bit address, which supports a network of 127 nodes and one global "General call" address. The eighth bit is used to signal a read or write request to the selected device. Once the master has sent the start condition and the address byte, the slave device will send a handshake in the form of an acknowledge. In the case of an error a NotAcknowledge will be sent and the master will start again. Once the slave has been selected and the direction of data transfer has been established, the actual data transfer will take place, with each byte being acknowledged by the receiving node. Finally the master will end the transaction by placing a stop condition on the bus.



The STR73x contains two I2C peripherals with internal baud rate generators which support master and slave mode .

The STR7 I2C peripheral may be thought of as an I2C engine in that it can generate the clock and data signals to send and receive data and also generate the start, stop and handshake signals. Your software is responsible for managing the data content of the I2C transaction to successfully communicate with other I2C devices on the network.



The I2C module is controlled by 7 registers but has two interrupt channels. One interrupt is used for data handling the other for control and error containment.

After a reset the I2C peripheral is disabled. To configure the peripheral you must first set the bit rate by programming the clock control registers. The clock control register and extended clock control register contain a 12-bit prescaler, which is used to divide the APB1 clock to give the desired clock rate. In addition, the clock control register allows you to select either standard I2C mode or Fast I2C mode. If you plan to run the bus faster than 100Kb/sec, you should select fast mode. Once the bit rate has been set, the peripheral can be enabled by writing to the PE bit in the control register. You must write twice to the PE bit in order to enable the I2C peripheral.

As the I2C peripheral has to respond to every event on the bus it is generally best make this device interrupt-driven. Internally each I2C peripheral has eleven interrupt sources. These are split into the two categories of "Bus events" and "data Transfer" . Each group of interrupts has an interrupt channel connected to the EIC. The interrupts are enabled by setting the ITE bit in the control register. This bit enables both interrupt channels so you must provide two interrupt service routines.

```

I2C_OAR2 = 0x00000080; // Set frequency bits for 50K bit rate
I2C0_CR  = 0x00000020; // enable the I2C module
I2C0_CR  = 0x00000020; // write twice to enable
I2C0_CR |= 0x00000001; // enable the ITE interrupt
I2C0_CR |= 0x00000004; // enable Ack
I2C0_CCR = 0x0000002C; // set the bit rate at 50K
I2C0_ECCR = 0x00000003; // set the upper clock register bits
I2C0_OAR1 = 0x00000002; // Set own address

```

4.12.5.1 I2C Addressing

The default addressing mode in the I2C module is a 7-bit address. This address must be placed in the highest seven bits of the own address 1. Register bit zero is not used. When a master addresses a node, it will place a matching address in the data register and bit zero is used to signify a read or write transaction. If bit zero is set to 1 it is a read and zero is a write. The I2C peripherals are also capable of using a 10-bit addressing mode. In this case, the address byte sent after the start bit must start with the pattern '11110' followed by the first two bits of the address. The least significant bit is again used to denote a read or write transaction. Transmission of the '11110' pattern will cause the ADD10 flag to be set as a request to send a second address byte containing the remaining eight bits of the address. Once the address has been sent, the data transmission carries on as in the 7-bit mode.

4.12.5.2 Slave Mode

After reset the STR7 I2C peripherals will be in slave mode and if you intend to make the STR7 a slave device, a unique network address must be programmed into own address register. Here you can set the local 7-bit address or if you are using 10-bit addressing, the additional address bits are placed in the own address register 2.

Once the I2C peripheral is fully configured it will wait for a master device to start a transaction. As soon as the master has placed the start condition on the bus and written the node address of the STR7 onto the bus, the STR7 peripheral will respond with an acknowledge handshake and generate a bus event interrupt. In this interrupt the STR7 can read the flags in the status registers.

7	6	5	4	3	2	1	0
EVF	ADD10	TRA	BUSY	BTf	ADSL	M/SL	SB
r	r	r	r	r	r	r	r

Depending what part of the I2C transaction has been reached different combinations of the flags will be set and either the bus event or data transfer interrupt will be generated. The data register is used to send and receive data directly into the I2C bus as data is received you can read it from this register or write to it when you need to send data.

When the ITE bit is set in the control register, two interrupt channels are enabled to the EIC. The ITERR channel will generate an interrupt when any flag in status register 2 is set and when the start bit (SB), address match (ADSL) or 10 bit addressing flags (ADD10) in status register 1 are also set. The second interrupt channel will generate an interrupt when the Byte transfer finished flag in status register 1 is set. This means that the handling of the I2C receive is split over two interrupt routines. The code below demonstrates the minimum handler you need to receive a single byte.

The ITERR routine waits for the I2C peripheral to be addressed and reading status register 1 clears the flags. When the data has been sent the, stop flag will trigger the ITERR interrupt, which reads status register 2 and clears the flags.

```
void I2C1_ITERR_isr ( void )
{
    unsigned int status;
        SWITCH_IRQ_TO_SYS;

    if(I2C1_SR1 & 0x84)
    {
        ;          // node address detected
    }
    if(I2C1_SR2 & 0x08)
    {
        ;          // stop bit detected
    }
        SWITCH_SYS_TO_IRQ;
        // clear IRQ Pending bit
        EIC_IPR0 = CHANNEL(8);
    }
```

Once the node has been addressed the next byte send will be interpreted as data and will trigger the TX_RX interrupt and the data can be read out from the data register.

```
void I2C1_IRQ_isr ( void )
{
    unsigned int dummy;

    SWITCH_IRQ_TO_SYS;

    while(!(I2C1_SR1 & 0x88))          // Wait for slave to detect data
    {
        ;
    }
    dummy = I2C1_DR;                  // read the received data

    SWITCH_SYS_TO_IRQ;

    // clear IRQ Pending bit
    EIC_IPR0 = CHANNEL(16);
}
```

4.12.5.3 I2C Master Mode

After the I2C peripheral has been configured you can enter Master mode by writing to the start bit in the control register. This places a start condition on the bus and places the STR7 in the role of bus master until the end of the transaction, when a stop condition is generated by setting the stop bit in the control register. Once the transaction has ended the I2C peripheral will revert back to a slave device. This is intended to allow another network device to act as a master and initiate a bus transaction. The same interrupt framework can be used in master mode - you simply need to respond to the additional combination of status bits. In master mode the background code must write to the control register to send the start bit and place the I2C peripheral into master mode.

```
I2C0_CR |= 0x08; // send start bit
```

Once the start bit has been sent an ITERR interrupt will be generated. This interrupt is used to send the node address you wish to send data too and then send the first byte of data.

```
void I2C0_ITERR_isr ( void )
{
    SWITCH_IRQ_TO_SYS;

    switch(TX_state)
    {
        case (0):
            while(!(I2C0_SR1 & 0x81) );           // Wait for bit Start condition to be sent
            I2C0_DR = 0x04;
            TX_state = 1;
            break;

        case(1):
            while ( !(I2C0_SR2 & 0x20));           // Wait for address to be sent

            I2C0_CR |= 0x20;                       // Write to Control register
            I2C0_DR = 0xAA;                       // Send data
            TX_state = 2;
            break;

        default :
            break;
    }
}
```

Once you start to send data bytes the TX_RX_INT interrupt will be generated. This can be used to send further data bytes, or you can end the transaction by sending a stop bit.

```
void I2C0_IRQ_isr ( void )
{
    SWITCH_IRQ_TO_SYS;

    /* BEGIN USER CODE I2C0_IRQ */

    while ( !(I2C0_SR1 & 0x88)) // Wait for data to be sent
    {
        ;
    }
    I2C0_CR |= 0x02;           // Send stop condition
    TX_state = 0;

    /* END USER CODE I2C0_IRQ */

    SWITCH_SYS_TO_IRQ;

    // clear IRQ Pending bit
    EIC_IPR0 = CHANNEL(15);
}
```

Exercise 21: I2C Loop Back

This exercise configures both I2C modules as master and slave and transfers data between them.

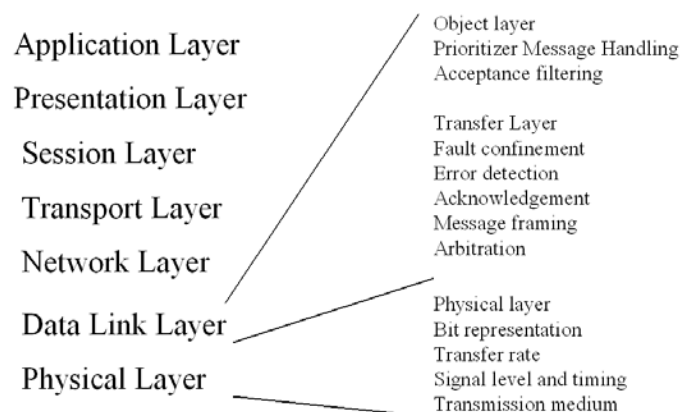
4.12.6 The CAN Controllers

Variants of the STR7 are available with an integrated CAN controller on-chip. The STR730 covered here has two identical but independent CAN modules. The CAN controller is one of the more complicated peripherals on the STR7. In this section we will have a look at the CAN protocol and the STR7 CAN peripherals.

The Controller Area Network (CAN) Protocol was developed by Robert Bosch for Automotive Networking in 1982. Over the last 22 Years CAN has become a standard for Automotive networking and has had a wide uptake in non -automotive systems where it is required to network together a few embedded nodes. CAN has many attractive features for the embedded developer. It is a low cost, easy to implement peer-to-peer network with powerful error checking and a high transmission rate of up to 1 Mbit/sec. Each CAN packet is quite short and may hold a maximum of eight bytes of data. This makes CAN suitable for small embedded networks that have to reliably transfer small amounts of critical data between nodes.

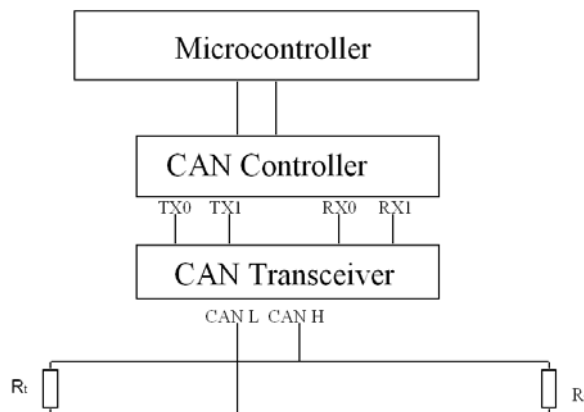
4.12.6.1 ISO 7 Layer Model

In the ISO seven layer model the CAN protocol covers the layer two 'data link layer', that is forming the message packet, error containment, acknowledgment and arbitration. CAN does not rigidly define the layer 1 'Physical layer' so can messages may be run over many different physical media. However the most common physical layer is a twisted pair and standard line drivers are available. The other layers in the ISO model are effectively empty and the application code directly addresses the registers of the CAN peripheral. In effect the CAN peripheral can be used as a glorified UART, without the need for an expensive and complex protocol stack. Since CAN is also used in industrial automation, there are a number of software standards that define how the CAN messages are used to transfer data between different manufacturers' equipment. The most popular of these application layer standards are CANopen and DeviceNET. The sole purpose of these standards is to provide interoperability between different OEM equipment. If you are developing your own closed system you do not need these application layer protocols and are free to implement you own proprietary protocol, which is what most people do in practice.



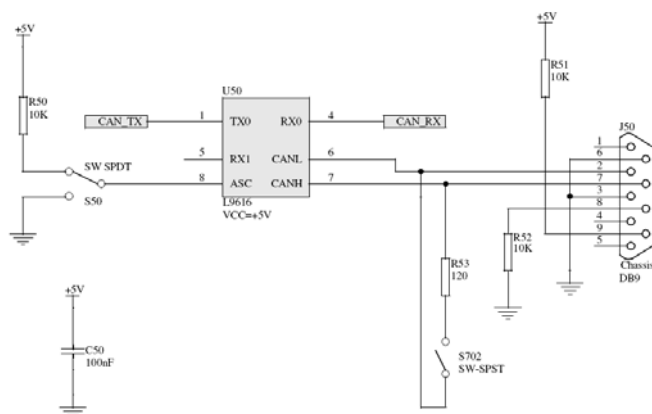
4.12.6.2 CAN Node Design

A typical CAN node is shown below. Each node consists of a microcontroller and a separate CAN controller. The CAN controller may, as in the case of the STR7, be fabricated on the same silicon as the microcontroller, or it may be a stand-alone controller, in a separate chip to the microcontroller. The CAN controller is interfaced to the twisted pair by a line driver and the twisted pair is terminated at either end by a 120 Ohm resistor. The most common mistake with a first CAN network is to forget the terminating resistors and then nothing works!



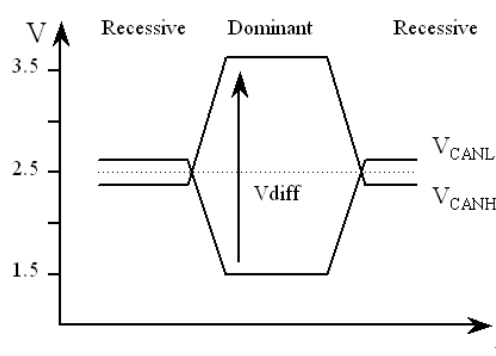
CAN node hardware: A typical CAN node has a microcontroller, CAN controller, physical layer and is connected to a twisted pair terminated by 120 Ohm resistors.

One important feature about the CAN node design is that the CAN controller has separate transmit and receive paths to and from the physical layer device. So as the node is writing onto the bus it is also listening back at the same time. This is the basis of the message arbitration and for some of the error detection. The physical layer is implemented with a dedicated 8-pin line driver



The CAN physical layer is implemented in an external 8 pin package available from ST and second sourced by a number of other manufacturers.

The two logic levels are written onto the twisted pair as follows, a logic one is represented by bus idle with both wires held half way between 0 and Vcc. A logic Zero is represented by both wires being differentially driven.

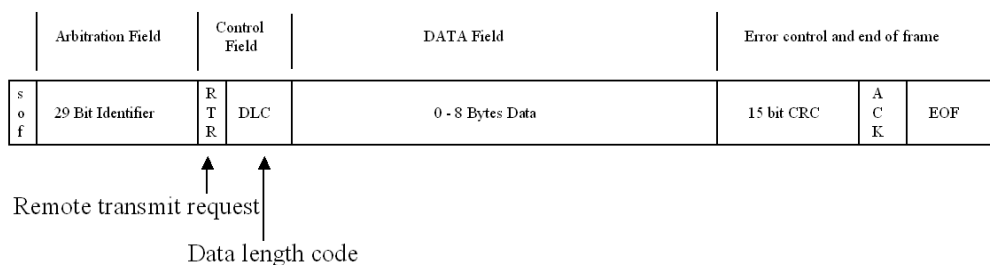


CAN Physical layer signals: On the CAN bus logic zero is represented by a maximum voltage difference called dominant, logic 1 by bus idle called recessive. A dominant bit will overwrite a recessive bit.

In CAN speak, a logic one is called a “recessive” bit and a logic zero is called “dominant” bit. In all cases a dominant bit will overwrite a recessive bit. So if ten nodes write recessive and one writes dominant then each node will read back a dominant bit. The CAN bus can achieve bit rates up to a maximum of 1 Mbit/sec. Typically this can be achieved over about 40 metres of cable. By dropping the bit rate longer cable runs may be achieved. In practice you can get at least 1500 metres with the standard drivers at 10 Kbit/sec.

4.12.6.3 CAN Message Objects

The CAN bus has two message objects that may be generated by the application software. The message object is used to transfer data around the network. The message packet is shown below:



CAN message packet : The message packet if formed by the CAN controller, the application software provides the data bytes the message identifier and the RTR bit.

The message packet starts with a dominant bit to mark the start of frame. Next comes the message identifier. This may be up to 29 bits long. The message identifier is used to label the data being sent in the message packet. CAN is a producer-consumer protocol. A given message is produced from one unique node and then may be consumed by any number of nodes on the network simultaneously. It is also possible to do point-to-point communication by making only one node interested in a given identifier. Then a message can be sent from the producer node to one given consumer node on the network. In the message packet the RTR bit is always set to zero - this field will be discussed in a moment. The DLC field is the data length code and contains an integer between 0 and 8 that indicates the number of data bytes being sent in this message packet. So although you can send a maximum of 8 bytes in the message payload, it is possible to truncate the message packet in order to save bandwidth on the CAN bus. After the 8 bytes of data there is a 15-bit cyclic redundancy check. This provides error detection and correction from the start of frame up to the beginning of the CRC field. After the CRC there is an acknowledge slot. The transmitting node expects the receiving nodes to assert an acknowledge in this slot within the transmitting CAN packet. In practice the transmitter sends a recessive bit and any node that has received the CAN message up to this point will assert a dominant bit on the bus, thus generating the acknowledge. This means that the transmitter will be happy if just one node acknowledges its message or if 100 nodes generate the acknowledge. So when developing your application layer care must be taken to treat the acknowledge as a weak acknowledge rather than confirmation that the message has reached all its destination nodes. After the acknowledge slot there is an end of frame message delimiter.

It is also possible to operate the CAN bus in a master slave mode. A CAN node may make a remote request onto the network by sending a message packet which contains no data but has the RTR bit set. The remote frame is requesting a message packet to be transmitted with a matching identifier. On receiving a remote frame the node which generates the matching message will transmit the corresponding message frame.

As mentioned the CAN message identifier can be up to 29 bits long. There are two standards of CAN protocol, the only difference being the length of the message identifier.

S O F	Identifier	RTR	DLC	CRC	ACK	EOF
-------------	------------	-----	-----	-----	-----	-----

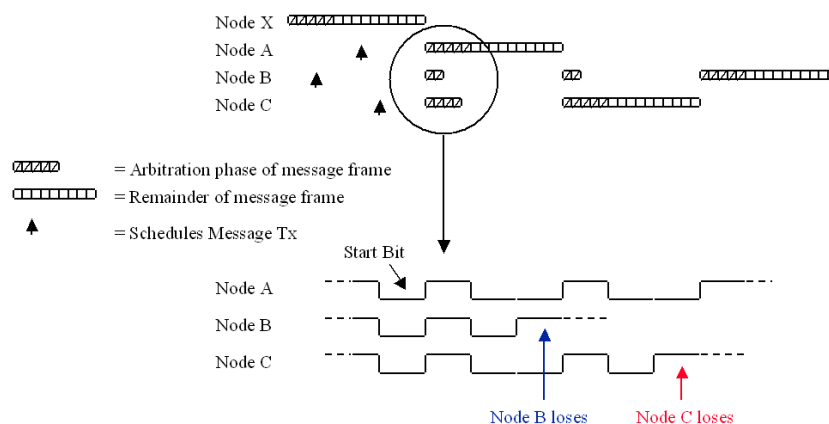
Remote Transmit Request: The RTR frame is used to request message packets from the network as a master slave transaction.

It is possible to mix the two protocol standards on the same bus but you must not send a 29-bit message to a 2.0A device.

	Frame with 11 bit ID	Frame with 29 bit ID	CAN Type	Identifier Type
V2.0B Active CAN Module	Tx/Rx OK	Tx/Rx OK	CAN 2.0A	11-bit identifier
V2.0B Passive CAN Module	Tx/Rx OK	Ignored	CAN 2.0B Passive	11-bit identifier
V2.0A CAN Module	Tx/Rx OK	Bus ERROR	CAN 2.0B Active	29-bit identifier

4.12.6.4 CAN Bus Arbitration

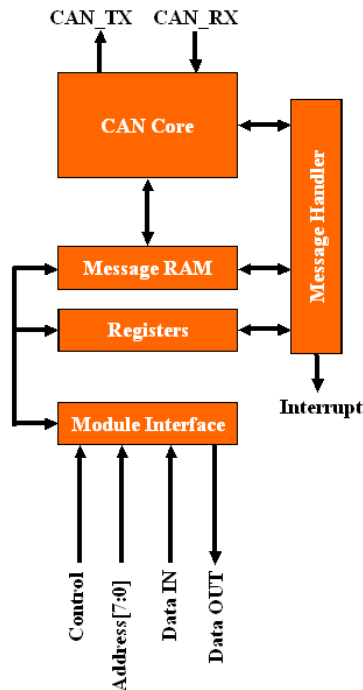
If a message is scheduled to be transmitted onto the bus and the bus is idle it will be transmitted and may be picked up by any interested node. If a message is scheduled and the bus is active it will have to wait until the bus is idle before it can before transmission. If several messages are scheduled while the bus is active they will start transmission simultaneously, being synchronised by the start of frame bit, once the bus becomes idle. When this happens the CAN bus arbitration will take place to determine which message wins the bus and is transmitted.



CAN arbitration: Message arbitration guarantees that the most important message will win the bus and be sent without any delay. Stalled messages will then be sent in order of priority, lowest value identifier first.

CAN arbitrates its messages by a method called non-destructive bit-wise arbitration. In the diagram above, three messages are pending transmission. Once the bus is idle and they are synchronised by the start bit, they will start to write their identifiers onto the bus. For the first two bits all three messages write the same logic and hence read back the same logic, so each node continues transmission. However on the third bit, node A and C write dominant bits and node B writes recessive. At this point node B wrote recessive but read back dominant. In this case it will back off the bus and start listening. Node A and C will continue transmission until node C write recessive and node A writes dominant. Now node C stops transmission and starts listening. Now node A has won the bus and will send its message. Once A has finished nodes B and C will transmit and node C will win and send its message. Finally node B will send its message. If node A is scheduled again it will win the bus, even though the node B and C messages have been waiting. In practice the CAN bus will transmit the message with the lowest value identifier.

4.12.6.5 STR7 CAN Module



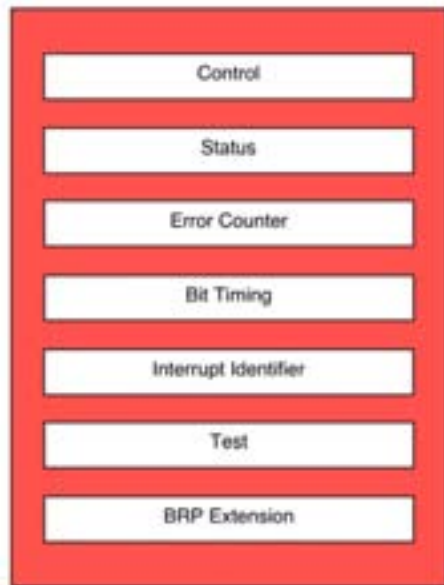
The STR7 CAN module is a full CAN module with an integral message RAM that supports 32 user - configurable receive/transmit buffers.

The STR7 CAN module will support CAN 2.0A and B with bit rates up to the full 1Mbit/S. It can operate in a basic CAN mode which is easy to use for simple CAN networks but also has a full CAN mode which enables 32 transmit and receive buffers stored in a block of message RAM located within the CAN peripheral. The full CAN mode greatly reduces the CPU overhead when you are serving a heavily-loaded CAN network. The CAN controller register set may split into two halves, the protocol registers and the message interface registers.



The special function registers may be split into two halves. The protocol registers which configure the CAN module and the message interface registers which are used to access the message buffers.

The CAN protocol registers are principally concerned with configuring the CAN controller and with error containment.



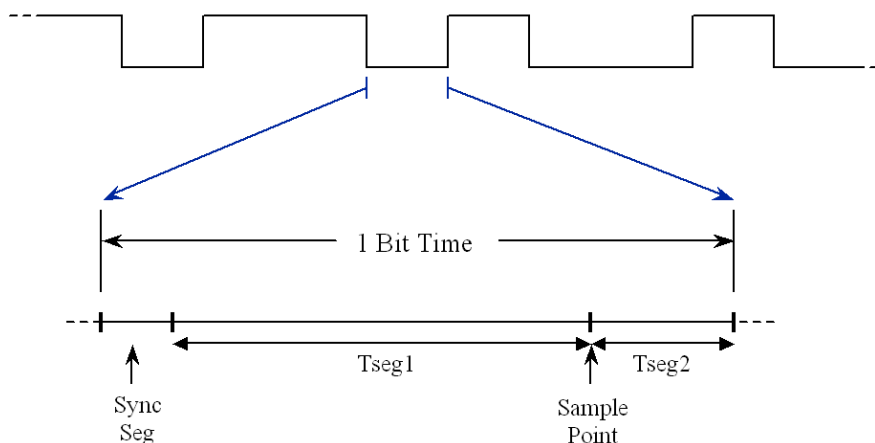
The CAN protocol registers configure the CAN bus parameters and manage the interrupt handling and error containment.

4.12.6.5.1 Initialising The CAN Controller

First we will look at the initialising the CAN controller and then later we will look at the error registers.

4.12.6.5.1.1 Bit Timing

Unlike many other serial protocols the CAN bit rate is not just defined by a baud rate prescaler. The CAN peripheral contains a Baud rate prescaler but it is used to generate a time quanta i.e. a time slice. A number of these time quanta are added together to get the overall bit timing.



CAN bit timing: Unlike other serial protocols the CAN bit period is constructed as a number of segments that allow you to tune the CAN data transmission to the channel being used.

The Bit period is split into three segments. First is the sync segment, which is fixed at one time quanta long. The next two segments are Tseg1 and Tseg2 and the user defines the number of time quanta in these regions. The minimum number of time quanta in a bit period is 8 and the maximum is 25. The receiving sample point is at the end of Tseg1, so changing the ratio of Tseg1 to Tseg2 adjusts the sample point. This allows the CAN protocol to be tuned to the transmission channel. If you are using long transmission lines, the sample point can be moved backwards. If you have drifting oscillators you can bring the sample point forward. In addition, the receivers can adjust their bit rate to lock onto the transmitter. This allows the receivers to adjust to small variations in the transmitter bit rate. The amount that each bit can be adjusted is called the synchronous jump width and may be set to between 1 – 4 time quanta and is again, user-definable.

To calculate the bit timing, the formula is:

$$\text{Bit rate} = \frac{F_{\text{CLK}}}{\text{BRP} \times (1 + \text{Tseg1} + \text{Tseg2})}$$

Where BRP = Baud rate prescaler and F_{CLK} is the APB1 clock

All of the calculated timing values are stored into the STR7 CAN registers as the calculated value – 1. This is in order to prevent any timing value being set to zero time quanta.

This calculation has a lot of unknowns. If we assume that we want to reach a bit rate of 125K with a 24 MHz F_{CLK} and a sample point of about 70%.

The total number of time quanta in a bit period is given by $(1 + \text{Tseg1} + \text{Tseg2})$. If we call this term QUANTA and rearrange the equation in terms of the baud rate prescaler.:

$$\text{BRP} = \frac{F_{\text{CLK}} \text{ Hz}}{\text{Bit rate} \times \text{QUANTA}}$$

Using our known values:

$$\text{BRP} = \frac{24\text{MHz}}{125\text{K} \times \text{QUANTA}}$$

Now we know that we can have between 8 and 25 time quanta in the bit period, so using a spreadsheet we can substitute in integer values between 8 and 25 for QUANTA until we get an integer value for BRP.

In this case when QUANTA = 16 BRP = 12:

$$\text{Then } 16 = \text{QUANTA} = (1 + \text{Tseg1} + \text{Tseg2})$$

So we can adjust the ratio between Tseg1 and Tseg2 to give us the desired sample point:

$$\text{Sample point} = \frac{\text{QUANTA} \times 70}{100}$$

$$\text{Hence } 16 \times 0.7 = 11.2$$

Round this to the nearest integer gives the sample point at 11 time quanta. The sync segment is always equal to 1 so Tseg1 = 11-1 = 10 and Tseg 2 will be equal to 5. Using these values the sample point will be at 68.8% of bit period.

The value for the synchronous jump width may be calculated by the following rule of thumb.

Tseg2 \geq 5 * Tq, then program SJW to 4

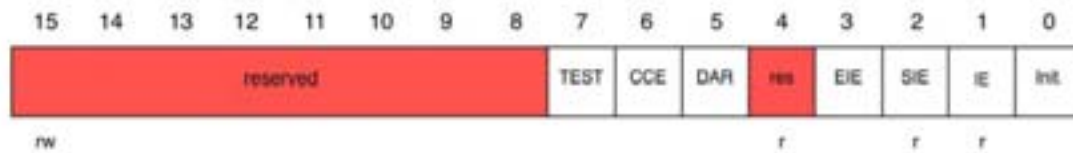
Tseg2 < 5 * Tq, then program SJW to (Tseg2 - 1) * Tq

In this case SJW = 4.

Remember that the actual values programmed into the timing register are the calculated values minus 1 hence the bit timing register is equal to 0x49CB. If the calculated value for the baud rate prescaler is greater than 64 the first six bits are programmed into the CAN_BRT register and the upper bits (minus 1) are programmed into the BRP extension register. This allows you to divide the F_{CLK} by a maximum of 1023.

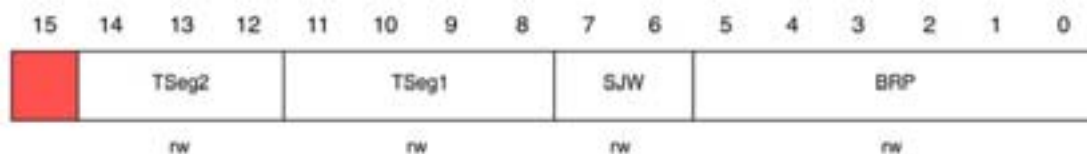
4.12.6.5.2 Configuring The CAN Module

Now that we have calculated the values for the CAN bit timing values we can perform the initial configuration of the Can module.



Before the CAN timing parameters can be programmed the CAN controller must be placed into reset by setting the init bit in the control register

After reset the CAN controller is held in its initialising mode with the init bit in the Can control register set to one. This allows the access to the timing registers once the init bit is set to zero the CAN controller enters its operating mode and the bit timing registers become read only. The values calculated above can be programmed into the bit timing register and the baud rate extension register.



The Timing register has fields for each of the bit timing parameters calculated in the bit timing example

Here it is important to note that the values programmed into the timing registers are the calculated values, minus one. This ensures that a timing segment cannot be programmed to zero length.

```

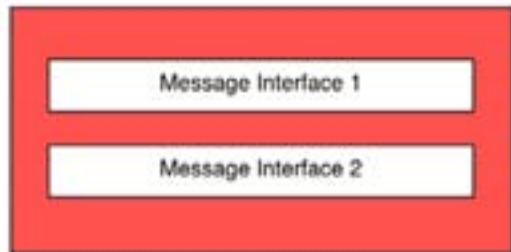
CAN0_CR      = 0x000000C1;           // init config change and test enable
CAN0_BRPR    = 0x00000000;           // set extended Baud rate prescaler to 0
CAN0_BTR     = 0x000045C3;           // set bit rate to 500K for FCLK = 24 MHz
CAN0_TESTR   = 0x00000004;           // set into basic mode
CAN0_CR      = 0x00000080;           // set into running mode, test bit set

```

The CAN TX pin (P1.12) should be set to alternate function, however the CAN RX pin (1.12) should be set to CMOS tristate input for the CAN peripheral to work.

4.12.6.5.3 Using The CAN Module

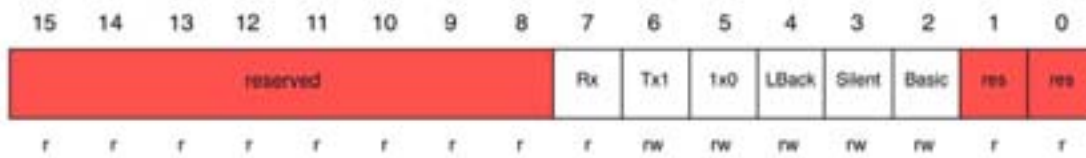
Once the CAN module is fully configured, it may be used to send and receive packets of data. The STR7 CAN module has two operating modes. For simple CAN networks with low data rates, there is an easy-to-use basic CAN mode which has a single transmit and receive buffer. For more demanding networks with higher data rates there is a full CAN mode which has multiple transmit and receive buffers. In either of these modes the CAN data is accessed via the message interface registers.



The CAN module has two sets of message interface registers. In Basic mode these act as transmit and receive buffers. In Full CAN mode they act as windows to the message RAM

4.12.6.5.3.1 Basic Mode

The CAN module enters the basic CAN mode during initialisation by setting the Basic bit in the test register.



The test register contains the BASIC bit which switches the CAN controller between Full and Basic CAN modes.

When basic mode is entered the message interface registers are configured as a transmit and receive buffer.



The message interface registers are used as transmit and receive buffers in basic mode. In full CAN mode they are used to program the 32 message objects in the CAN message RAM.

In the basic mode the IF1 registers act as the transmit buffers and the IF2 registers act as the receive buffers. However both interface blocks have the same register layout. To transmit a message in basic mode the message identifier must be programmed into the arbitration registers. Next the message data is placed in the data registers and the data length code is programmed into the control register. Finally to send the message the busy bit in the command register must be set. Now the CAN message will be scheduled for transmission and will begin transmission as soon as the bus enters an idle state. When a message is received, its identifier is placed in the arbitration registers and the data is available from the data registers and a receive interrupt is generated. This potentially means that in basic mode, all messages on the network will be received by the STR7 CAN controller. This would cause an interrupt on the CPU everytime there was data on the CAN network, rapidly loading the CPU. To avoid this problem the receive buffer has a message filter that can be set to only accept a specific message or range of messages.

```
CAN0_IF1_A1R = 0x00000000; // Clear the arbitration register
CAN0_IF1_A2R = 0x0000A001; // Set the ID for a standard message
CAN0_IF1_MCR |= 0x00000001; // Set the data length code to send just one
                                // byte
CAN0_IF1_DA1R = 0x00000055; // Write the data in data0 register
CAN0_IF1_CRR = 0x00008001; // Set the busy bit to start transmission
```

In basic mode the IF2 message registers become the receive buffer. However it must be enabled by setting the message valid bit in the upper arbitration register.

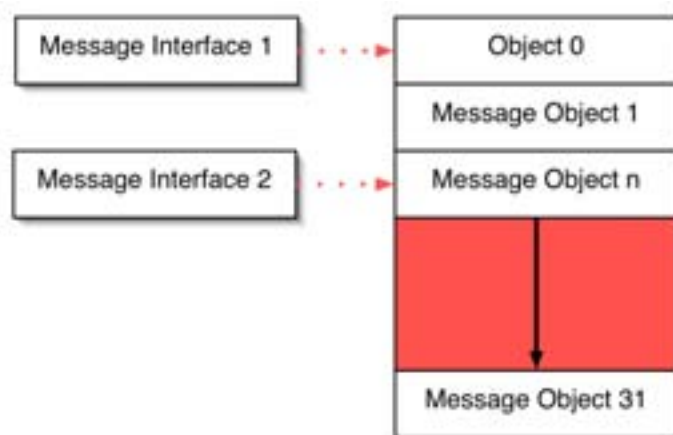
```
CAN0_IF2_A2R = 0x8000;
```

When a message is received the new data flag in the message control register will be set and the new message packet may be read from the IF2 registers. The new data flag must be cleared to unlock the IF2 registers so another message may be received.

```
if(CAN0_IF2_MCR & 0x8000)
{
    id = (CAN0_IF2_A2R >>2) &0x000007FF; // read the ID 11 bit standard
    dlc = CAN0_IF2_MCR & 0x0x0000000F // read the data length code
    test = CAN0_IF2_DA1R; // read the message data
    CAN0_IF2_MCR &= ~0x8000; // clear the new data flag
}
```

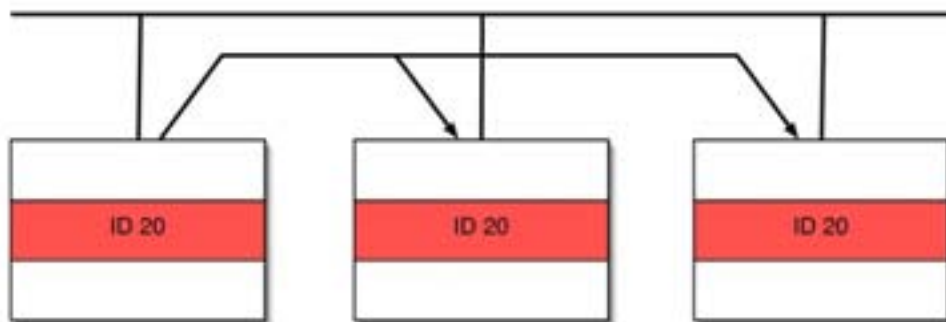
4.12.6.5.3.2 Full CAN Mode

While basic CAN mode is easy to use, it does mean that an interrupt will be generated for every message received and even with the use of the message masks in a medium to heavily loaded network, this will put a significant loading on the CPU. The STR7 CAN module has a full CAN mode that enables a block of message RAM. This message RAM is configured as 32 message objects which may be individually configured as transmit or receive objects.



The message RAM is arranged as 32 message buffers that can be individually configured as transmit or receive buffers. The two sets of message interface registers can access any of these buffers to send and receive CAN messages.

The message RAM is accessed by the transmit and receive message buffer registers that were used in the basic CAN mode. These two registers sets each become a floating window which can be set to interface to any selected message object in the message RAM to read and write data. In full CAN mode the two interface register sets both act as interface “windows” and unlike Basic CAN mode, do not have specific transmit or receive functions. They can be used to perform either task. Each of the 32 message objects may be configured as a transmit or receive buffer when the CAN controller is initialised. The transmit objects can be configured with the message identifier and DLC initialised, so to send a given message, the CPU simply has to update the data and schedule the message for transmission. So if your CAN application generates say 10 different messages onto the bus, each message may have its own transmit buffer. The remaining message objects may be configured as receive objects. The message mask and arbitration registers allow each message object to receive a specific CAN identifier or range of identifiers. This allows you to use the remaining message objects as dedicated receive buffers. When a message arrives at the CAN controller, the enabled receive objects are scanned to find an message object that matched the message identifier. If a match is found, the message is stored and a receive interrupt is generated. When the CPU responds to the message it can examine the New data registers to locate the updated message buffer. The data can then be read directly, without having to examine the message identifier. The message objects are particularly suitable if you are running a “loose” network. By this we mean it is not necessary to capture every CAN message. For example, you may have a CAN message that is sent every 100ms and contains a temperature reading. In a process control application, it is not necessary to capture every temperature reading (i.e. the trend) but it is necessary to know the temperature now. In such a system a message object may be configured to receive the temperature message and the data will be refreshed each time the message appears on the CAN network. The STR7 can then read the current value of the temperature variable whenever it needs it by accessing the data in the message data registers. In effect, the receive message data buffers can be thought of as virtual network memory.



In full CAN mode a message transmitted from one node will update all the message buffers configured to receive this message. The CAN controller message RAM is like a page of virtual memory shared across the network.

However if your application must capture every message, a receive interrupt can be generated for every message that arrives. If you have a heavily loaded network, particularly if the message traffic occurs in bursts, this can create a lot of interrupts for the CPU to service. The STR7 CAN module allows several of the message objects to be combined to create a message FIFO for a given message identifier, so we can have “lossy” message objects for process control messages alongside FIFO buffers for critical lossless messages, all combined in the same CAN controller.

The message interface registers are used in the same way as for the basic CAN configuration but once the data has been written to the registers, the command register allows you to select the message object to update by writing to the message number field.



Data may be transferred to and from the CAN message objects by configuring the message mask register and then writing the message object number into the command request register. During the data transfer the busy bit is set.

The command mask register is also used to select which fields within the message object will be written to, or read from. This allows you to initialise the message objects for transmit or receive and initialise the message parameters. Then during operation you can selectively read/write the data portion of the message objects.

```

CAN0_CR      = 0x00000041;          // init and config change enable
CAN0_BRPR    = 0x00000000;          // Clear the extended BRP
CAN0_BTR     = 0x000045C3;          // set bit rate to 500K

//Configure a transmit object ( could use the IF2 registers)
CAN0_IF1_CMR = 0x000000F0;          // enable write of mask,arb and control
registers
CAN0_IF1_M1R = 0x00000000;          // Clear the lower mask registers
CAN0_IF1_M2R = 0x00000000;          // Clear the upper
CAN0_IF1_A1R = 0x00000000;          // Clear lower arbitration register
CAN0_IF1_A2R = 0x0000A004;          // Message valed, std frame,transmit, ID1
CAN0_IF1_MCR = 0x00000001;          // DLC = 1
CAN0_IF1_CRR = 0x00000001;          // write to message object one
while(CAN0_IF1_CRR & 0x8000);       // wait for the message object to be
// written to

//configure a receive object ( could use the IF2 registers)
CAN0_IF1_CMR = 0x000000F0;          // enable write of mask,arb and control
// registers
CAN0_IF1_M1R = 0x00000000;          // Clear the lower mask registers
CAN0_IF1_M2R = 0x00000000;          // Clear the upper
CAN0_IF1_A1R = 0x00000000;          // Clear lower arbitration register
CAN0_IF1_A2R = 0x00008004;          // Message valed, std frame,receive, ID1
CAN0_IF1_MCR = 0x00000000;          // clear the message control register
CAN0_IF1_CRR = 0x00000002;          // write to message object two

while(CAN0_IF1_CRR & 0x8000);       // Wait for the message object to be
// written to
CAN0_CR      = 0x00000080;          // set into running mode, test bit set

```

To configure a a group of message objects as a FIFO, you must simply set the message arbitration and mask registers to the same value for a contiguous block of message objects. In addition, the end of buffer bit (EoB) in the message object control register for the message object with the highest message object number must be set to one. The EoB bit in all the other message objects that are part of the FIFO must be set to zero.

Once the message objects are configured you can transmit a message by writing data into the message data registers and setting the TXRequest bit in the message control register.

```

CAN0_IF1_CMR = 0x00000086;          // Write data and control register set
// TXrequest bit
CAN0_IF1_DA1R = 0x00000055;          // load some data
CAN0_IF1_CRR = 0x00000001;          // write to message object one

while(CAN0_IF1_CRR & 0x00008000);   // Wait for the message object to be written
to

```

When CAN messages are received, the message identifier will be compared to the message identifiers in the valid message objects. If the ID matches, the message data will be stored in the message object data registers and the flag corresponding to the message object in the New data register will be set. The received data can then be read into the message interface registers and then be made available to the application software.

```

if(CAN0_ND1R & 0x02                // if message object 2 has received new data
{
    CAN0_IF1_CMR = 0x00000003;          // Mark all data registers for read
    CAN0_IF1_CRR = 0x00000002;          // read message object two

    while(CAN0_IF1_CRR & 0x00008000);   // Wait for the busy bit to clear

    CAN_data_0_1 = CAN0_IF1_DA1R;        // read the data from the message registers
    CAN_data_2_3 = CAN0_IF1_DA2R;

```



```

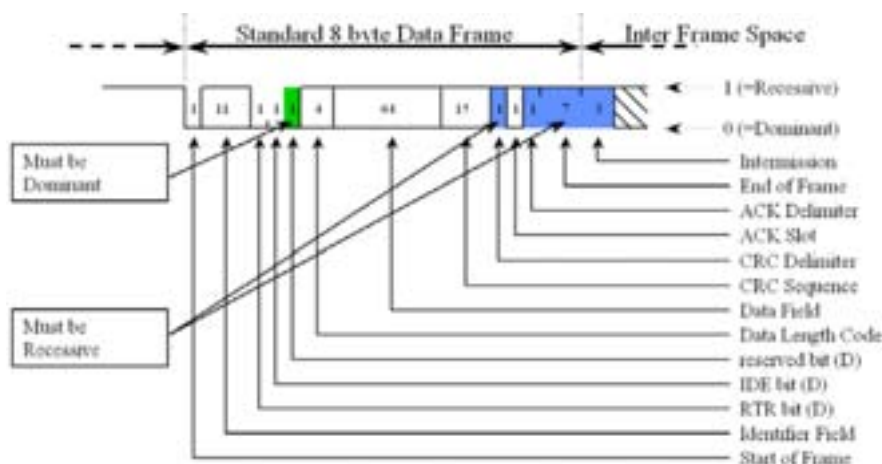
CAN_data_4_5 = CAN0_IF1_DB1R;
CAN_data_6_7 = CAN0_IF1_DB2R;
}

```

It is important to wait for the busy bit to clear. This takes several cycles and if you read the data registers too soon you, will get old data.

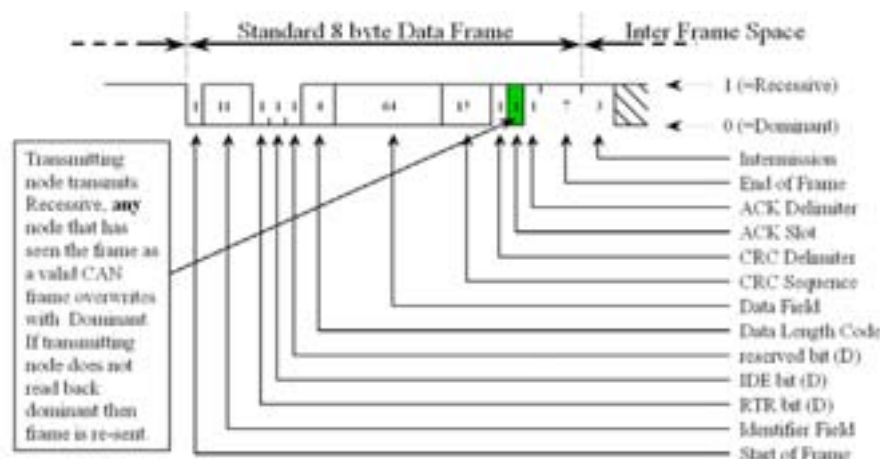
4.12.6.5.4 CAN Error Containment

The CAN protocol has five methods of error containment built into the silicon. If any error is detected it will cause the transmitter to resend the message so the CPU does not need to intervene unless there is a gross error on the bus. There are three error detection methods at the packet level – form check, CRC, and acknowledge and two at the bit level – bit check error and bit stuffing error. Within the Can message there are a number of fields that are added to the basic message. On reception the message telegram is checked to see if all these fields are present if not the message is rejected and an error frame is generated. This ensures that a full, correctly formatted message has been received.



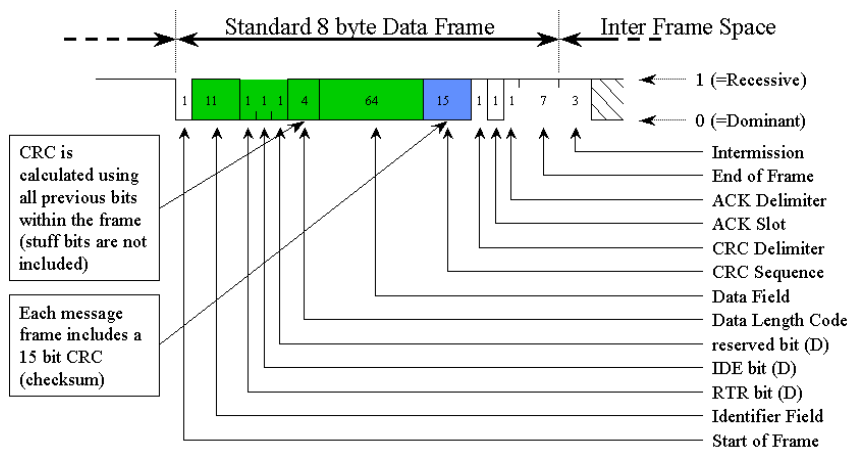
Frame Check: The frame check tests that a correctly formatted Can message has been received.

Each message must be acknowledged by having a dominant bit inserted in the acknowledge field. If no acknowledge is received, the transmitter will continue to send the message until an acknowledge is received.



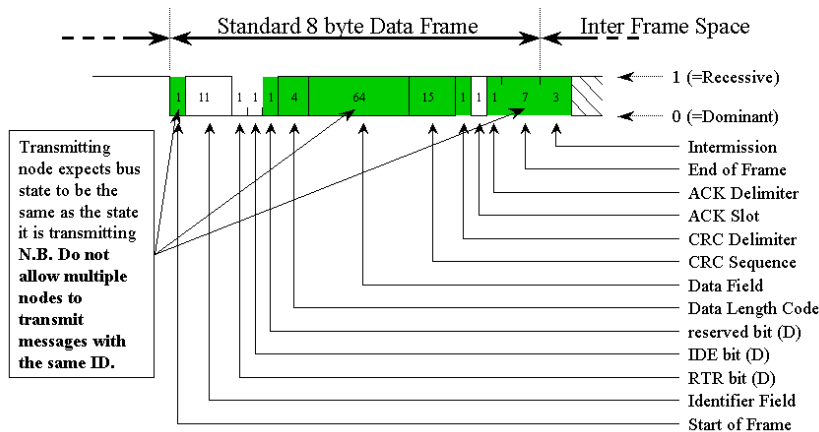
Acknowledge: All CAN frames must be acknowledged. If there is no handshake the message will be re sent.

The CAN message packet also contains a 15 bit CRC which is automatically generated by the transmitter and checked by the receiver. This CRC can detect and correct 4 bits of error in the region from the start of frame to the beginning of the CRC field. If the CRC fails and the message is rejected an error frame is generated onto the bus.



CRC: A 15-bit CRC is automatically generated which is a weighted polynomial checksum that provides error detection and correction across the message packet.

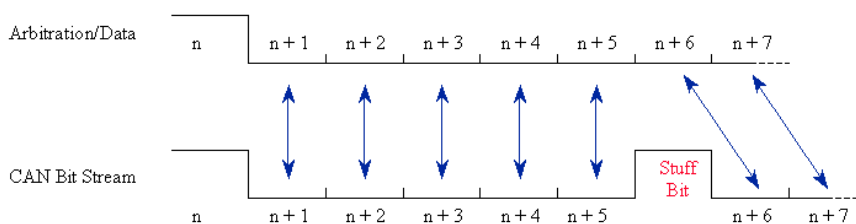
Once a node has won arbitration it will start to write its message onto the bus. As during arbitration as each bit is written onto the bus the CAN controller is reading back the level written onto the bus. As the node has won arbitration nothing else should be transmitting so each bit level written onto the bus must match the level read back. If the wrong level is read back the transmitter generates an error frame and reschedules the message. The message is sent in the next message slot but must still go through the arbitration process with any other scheduled message.



Bit check error: Once the arbitration has finished the write and read back mechanism is used for bitwise error checking.

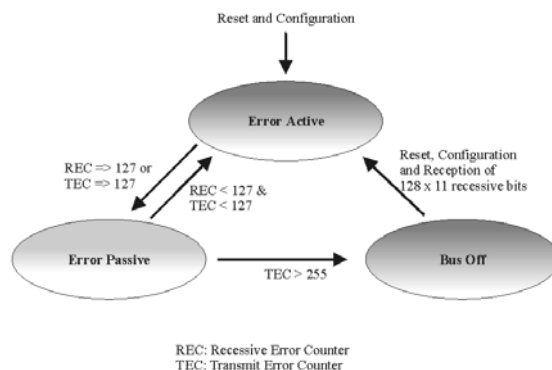
This leads to one of the golden rules in developing a CAN network. In a CAN network every identifier must be uniquely generated. So you must not have the same identifier sent from two different nodes. If this happens it is possible two messages with the same ID are scheduled together, both messages will fight for arbitration and both will win as they have the same ID, once they have won arbitration they will both start to write their data onto the bus at some point this data will be different and this will cause a bit check error, both messages will be rescheduled, win arbitration and go into error again. Potentially this 'deadly embrace' can lock up the network so beware!

At the bit level CAN also implements a bit stuffing scheme. For every five dominant bits in a row a recessive bit is inserted. This helps to break up DC levels on the bus and provides plenty of edges in the bitstream, which are



Bit Stuffing: For every five bits of logic '1' in a row, a stuff bit of the opposite logic is inserted. The error frame breaks this rule by being six dominant bits in a row.

used for resynchronisation. An error frame in the CAN protocol is simply six dominant bits in a row. This allows any CAN controller to assert an error onto the bus as soon as the error is detected without having to wait until the end of a message.



Error counters: The CAN controller moves between a number of error states that allow a node to fail in an elegant fashion without blocking the bus.

Internally each CAN controller has two counters - a receive error counter and a transmit error counter. These counters will count up when receiving or transmitting an error frame. If either counter reaches 128 then the CAN controller will enter an 'error passive' mode. In this mode it still responds to error frames but if it generates an error frame, it writes recessive bits in place of dominant bits. If the transmit error counter reaches 255 then the CAN controller will go into a bus-off condition and take no further part in CAN communication. The CPU must intervene to reinitialise the controller and put it back onto the bus. Both these mechanisms are to ensure that if a node goes faulty, it will fail gracefully and not block the bus by continually generating error frames.

4.12.6.5.4.1 STR7 Error Handling

Error conditions within the STR7 Can controller are reported in the status register .

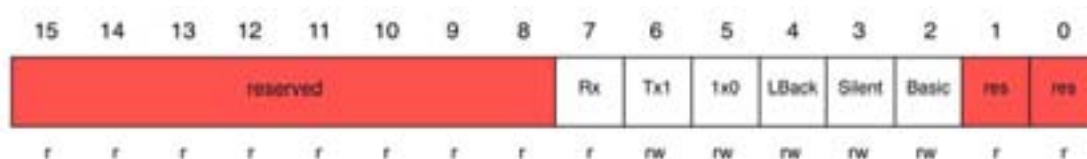


The status register contains the error warning flags and a Last error code that details the last fault that occurred on the CAN bus.

This register contains flags for busoff and error passive conditions, as well as an early warning error limit, which is set when the error counters reach a count of 96. In addition to the error flags, there is a last error code field which reports the type of error condition that was last encountered. In addition, the error counter register allows you to read the current values held in the transmit and receive error counters.

4.12.6.5.5 STR7 CAN Test Modes

The test register within the Can module allows the module to be configured into a number of special modes. However before you can write to this register you must set the test bit in the CAN control register. The loopback mode can be used for self test routines. As its name implies in this mode the TX pin is internally connected to the RX pin so the Can module receives its own message. This allows you to check the integrity of the Can controller.



The test register can set the CAN controller into loopback mode and is also able to manually control the TX pin and read the RX pin for physical layer testing. The silent mode prevents the controller generating an acknowledge to any CAN message.

It is also possible to test the integrity of the physical layer since the TX pin may be controlled by software and the bus level on the RX pin may be read directly. The TX0 and TX1 bits may write the bus level to dominant or recessive and this can be read back via the RX bit. The test register also allows you to configure the CAN controller into “silent” mode. In this mode, the CAN controller is not able to transmit any messages and will not generate any acknowledges or error frames. It is able to receive messages but does not take any active part on the bus. You can also use the combination of the loopback and silent modes to do a self-test on a live network, without any danger of upsetting the operation of that network.

4.12.6.6 Deterministic CAN Protocols

Although it is possible to guarantee delivery of a CAN message within a certain maximum delay, it is difficult to schedule CAN messages to arrive at regular intervals. Most systems that need exchange data on a regular basis rely on having enough bandwidth available to allow any message arbitration to be resolved and still have enough time for a delayed message to reach its target within its time budget. In an effort to make CAN more deterministic, an extension to the standard CAN protocol called Time Triggered CAN (TTCAN) uses a time-division multiplexing approach to ensure all CAN messages are schedulable. In the TTCAN protocol, a start of frame message is sent and then each CAN message is allocated a slot within which it must send its data. Using this approach there is no arbitration and each message is guaranteed to be delivered at a regular and known rate. However the TTCAN protocol cannot be implemented with a standard CAN controller because if there is a bus error, a CAN message would automatically resent. This would cause a message to be sent in the next slot and destroy the determinism of the system. The STR7 CAN controller has been designed to support the TTCAN protocol by allowing the automatic retransmission of messages to be switched off. This ensures that if a bus error occurs, the message will be abandoned for this frame and will be resent by software in its correct slot in the next frame. The automatic retransmission can be setting by clearing the DAR bit in the control register.

Exercise 22: CAN loop back

This exercise configures the CAN controller in both Basic and Full CAN mode and uses the loop-back test mode to simulate sending and receiving CAN message packets.

4.13 Summary

The STR73x has a comprehensive set of easy to use peripherals that meet the requirement of today's developers. If you have worked through the proceeding chapters accessing the peripherals will be straight forward. The tutorial programs help you to get each of the peripherals working as quickly as possible. In addition the bibliography list additional tools and resources that assist development of an STR73x based application.

5 Chapter 5: Tutorial Exercises

5.1 Introduction

Having read chapter one, you should now have an understanding of the ARM7 CPU. In this chapter we will look at how to write and debug C code for the ARM7. The example programs given will concentrate on demonstrating how to use the unique features of the ARM7 discussed in first chapter.

Since the ARM7 is rapidly becoming an industry-standard CPU for general-purpose microcontrollers, this will allow you to develop code on the STR7 and a number of other devices.

5.2 Further STR730 Examples

The latest versions of these tutorial exercises plus further STR730 example programs can be found at <http://www.hitex.co.uk/str730>.

5.3 Exercise 0: Installing the Software

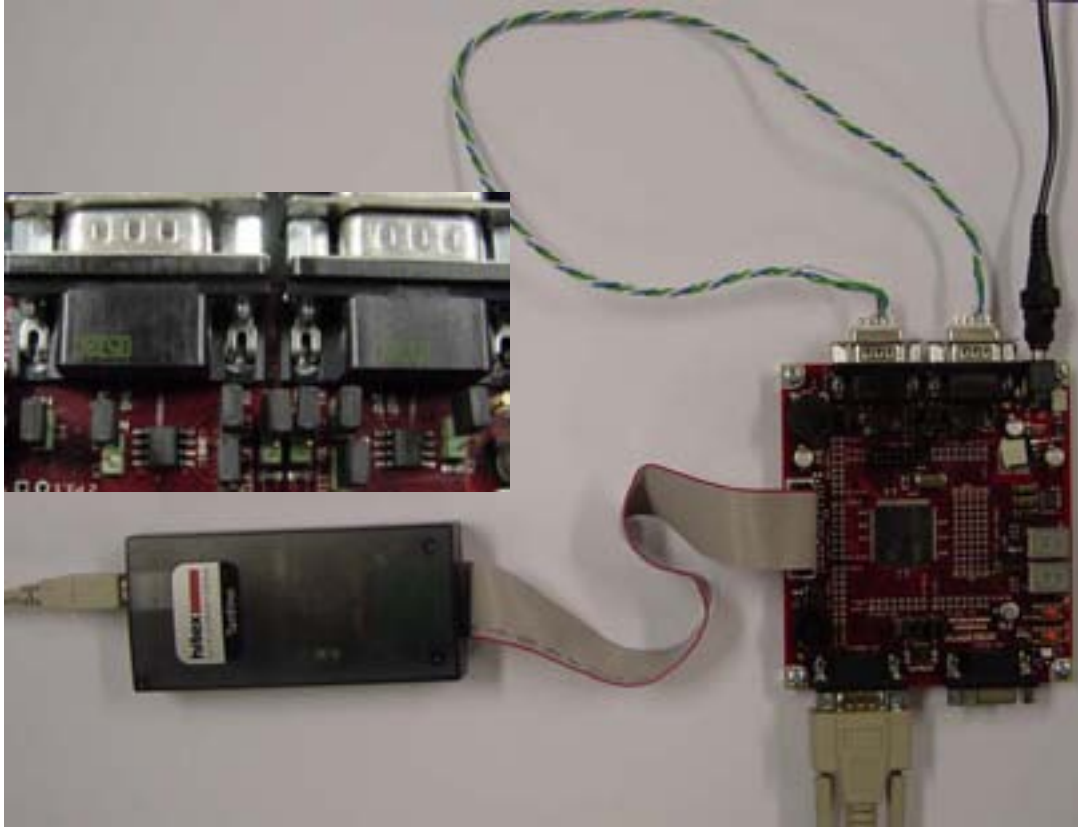
Once you have inserted the starter kit disk, it will AUTORUN and display the screen shown below:



From this screen, install the StartEasy software, GNU C /C++ compiler and the HiTOP debugger software for the Tantino 7-9 (NOT the standard Tantino). Finally from the Insider's Guide CD, copy the example programs onto your PC to a convenient directory. When you have finished installing the software you will need to reboot your PC.

5.4 Setting up the Hardware

Once the hardware is installed you will need to setup the evaluation board, as illustrated.



The boot config jumpers on the STR7 evaluation board should be set to the factory default setting shown below. This ensures that the STR7 boots from its on-chip FLASH memory. This is discussed further in Chapter 3.

The Tantino7-9 JTAG debugger should be plugged into the debug socket and the power supply plugged into the power socket on the evaluation board. Next the USB cable should be plugged into the Tantino 7-9 to connect the JTAG debugger to the PC. Finally a RS232 serial cable should be added to connect the socket labelled “RS232 – A” to a PC serial port.

5.5 Overview

The installed software discussed in this chapter includes StartEasy, which is a free CASE tool for ST and a number of other ARM7-based microcontrollers. StartEasy allows you to rapidly define a skeleton project that configures the most commonly-used peripherals and the interrupt structure. The GCC compiler is an open source C/C++ compiler that generates code to run on ARM7-based microcontrollers. The HiTOP IDE is an editing and debugging environment for all Hitex emulators and debuggers. It will work as a stand-alone ARM7 CPU simulator (HiSIM-ARM) or as the front-end for the Tantino7-9 JTAG debugger. The evaluation version is limited to working with an application program that has a maximum size of 16KB, although it is possible to upgrade to a full licence that allows you to work with any size of code. These three software applications make a complete evaluation toolchain for the STR7 and will be used throughout the remainder of this chapter. The remaining software on the disk is an evaluation version of a software testing program called “TESSY” and a software quality toolset called “Development Assistant for C”.

5.6 Exercise 1: Setting Up Your First Project

In order to familiarise ourselves with the toolset, we will work through generating a simple “Blinky” LED chaser example and run this on the evaluation board.

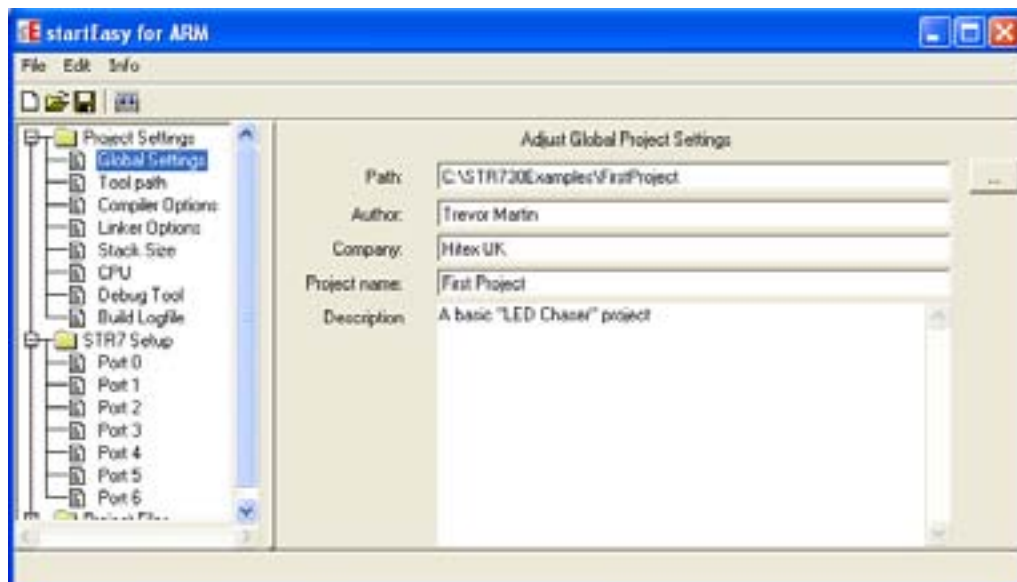
5.6.1 StartEasy

StartEasy is an easy-to-use case tool which will generate a bare-bones project that contains the necessary startup code for the STR7, as well as being able to auto-generate C-code to initialise selected peripherals. It also generates a HiTOP project that allows you to edit, compile and debug this basic project.

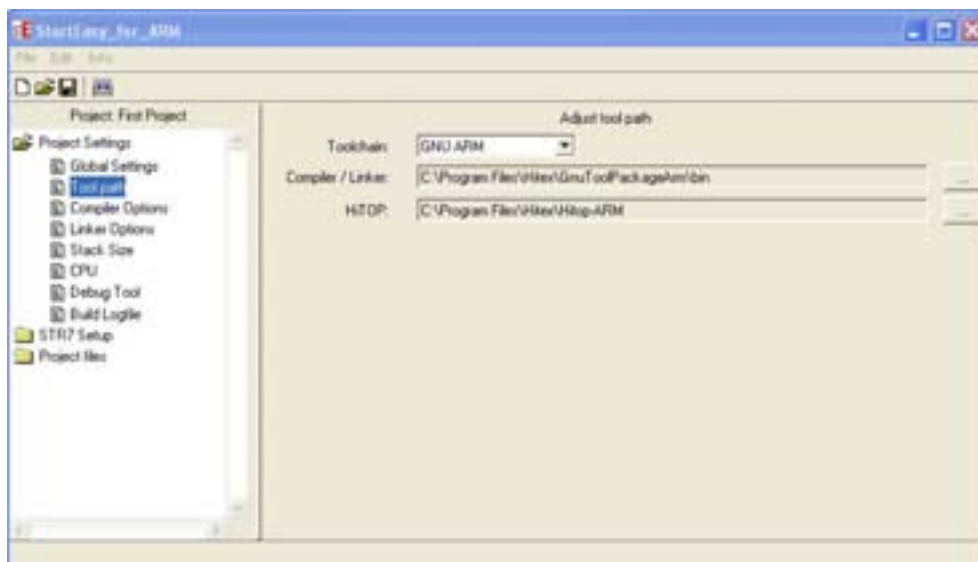
To start StartEasy, select and double-click the StartEasy for ARM icon:



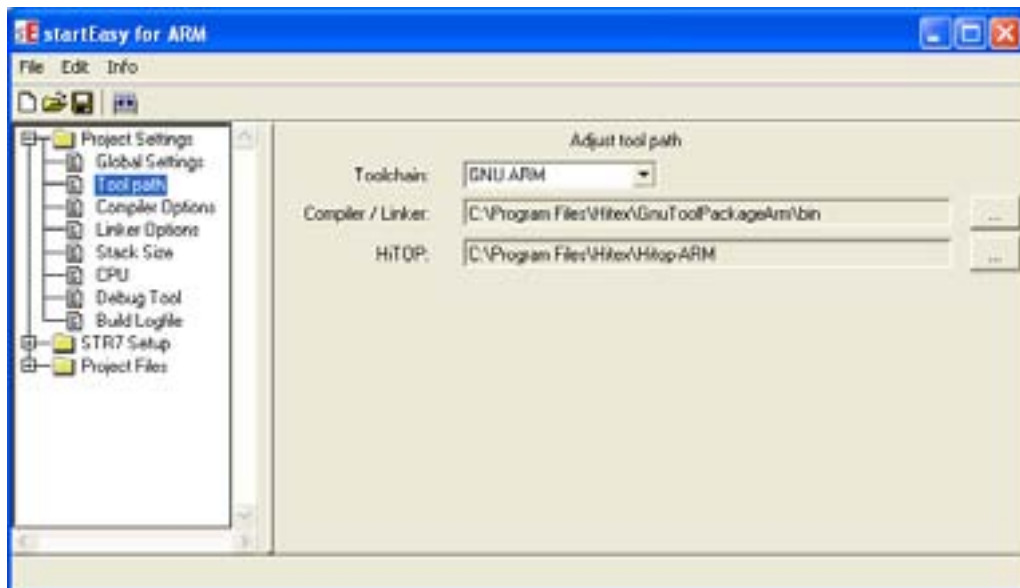
This will open the project settings window, which should be filled-in with the basic project information, as shown below. The project name will also be the name of the debug file produced by the GCC linker.



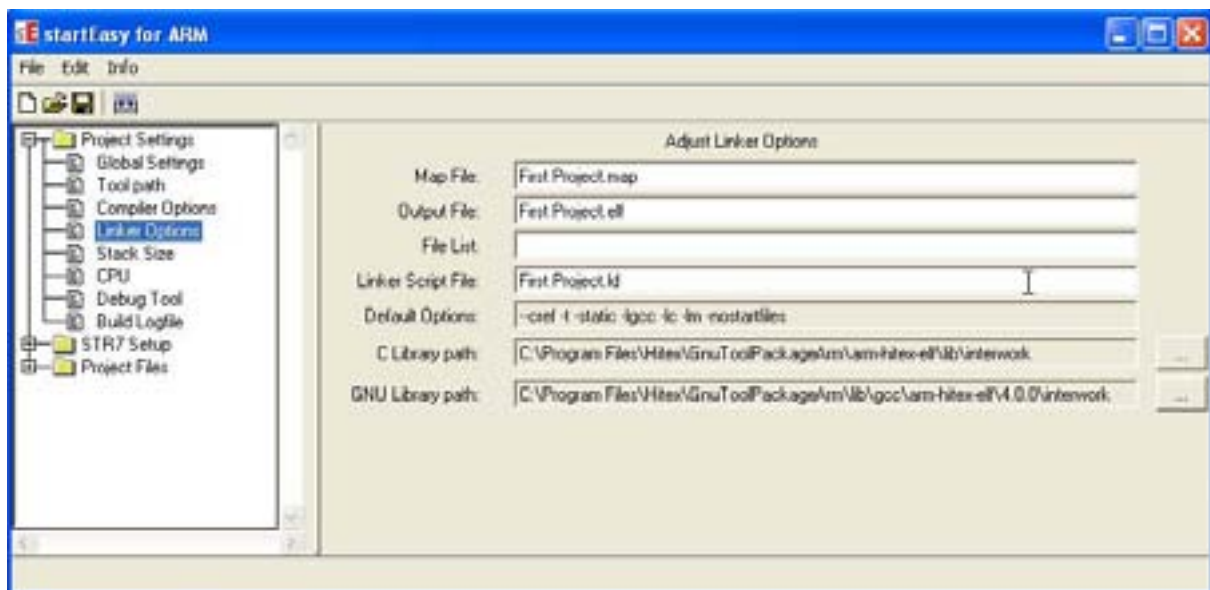
Next, click on the project settings folder to expand the project settings options and select tool path. There are a number of different installation packages for the GCC compiler but the package installed here requires the tool path to be setup as shown below:



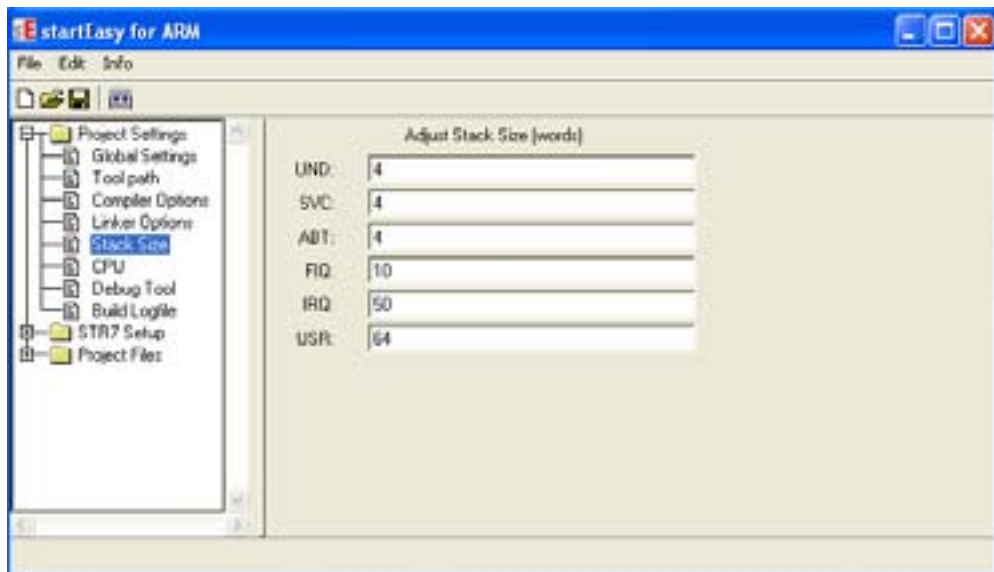
For the current example the compiler options can be left at their default settings but these will be discussed in more detail later on in this chapter.



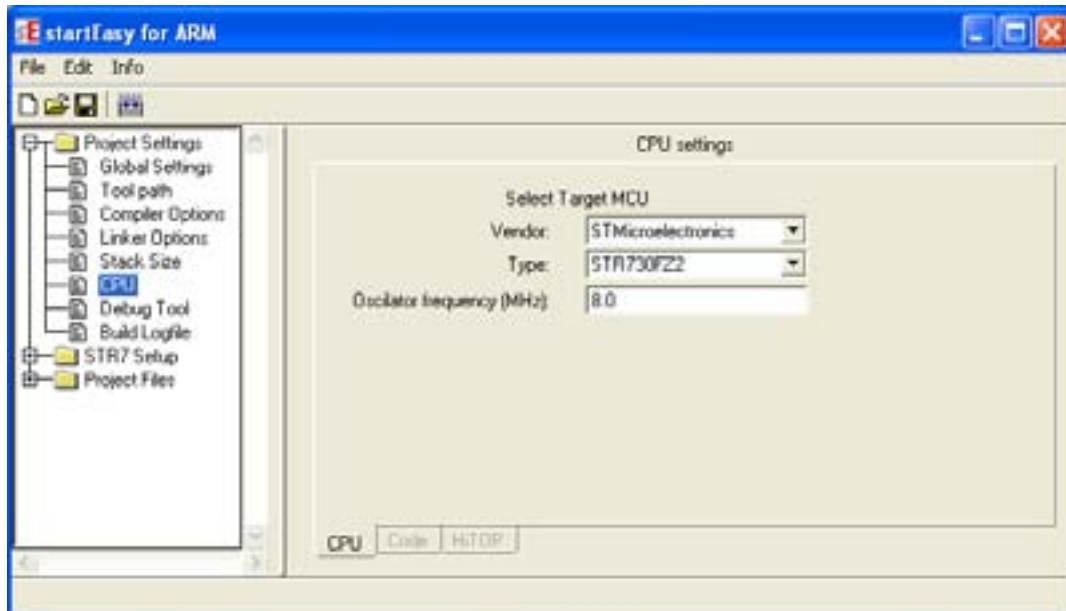
Next open the linker options dialog. The project output files and the linker script file default to the name of your project. Again, because the GCC compiler can be installed to various locations, check that the linker library paths are configured to match the current installation.



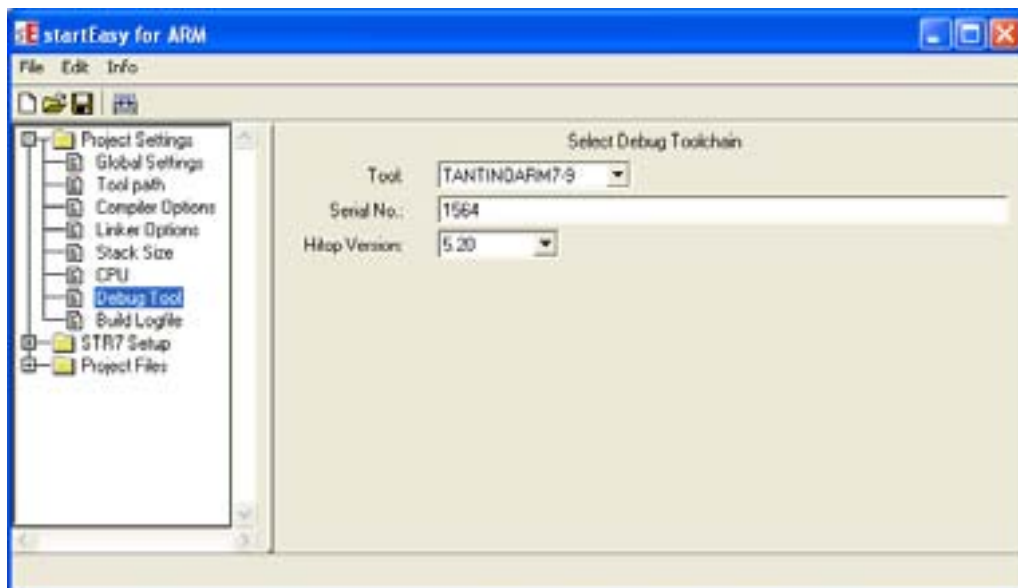
Once you have checked the linker options, open the “Stack Size” dialogue. The stack size window allows you to define how many bytes are allocated to each of the stacks available in the different operating modes of the ARM7. The stack configuration is done in the startup code. This is held in an Assembler file that is used to configure the STR7 before it enters the main() function in your C code. For a simple project, we can use the default settings.



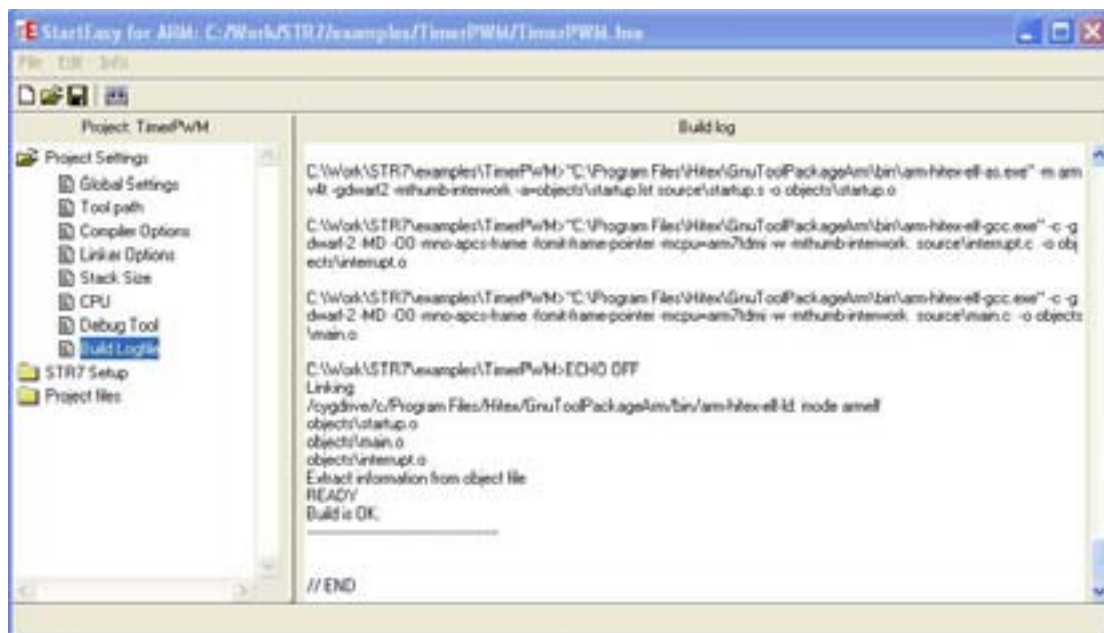
After you have examined the stack window, open the CPU dialog. In this window you must select the vendor to be “ST Microelectronics” and the Type to be the “STR730FZ2”. You must also set the Oscillator frequency to be 8.0MHz



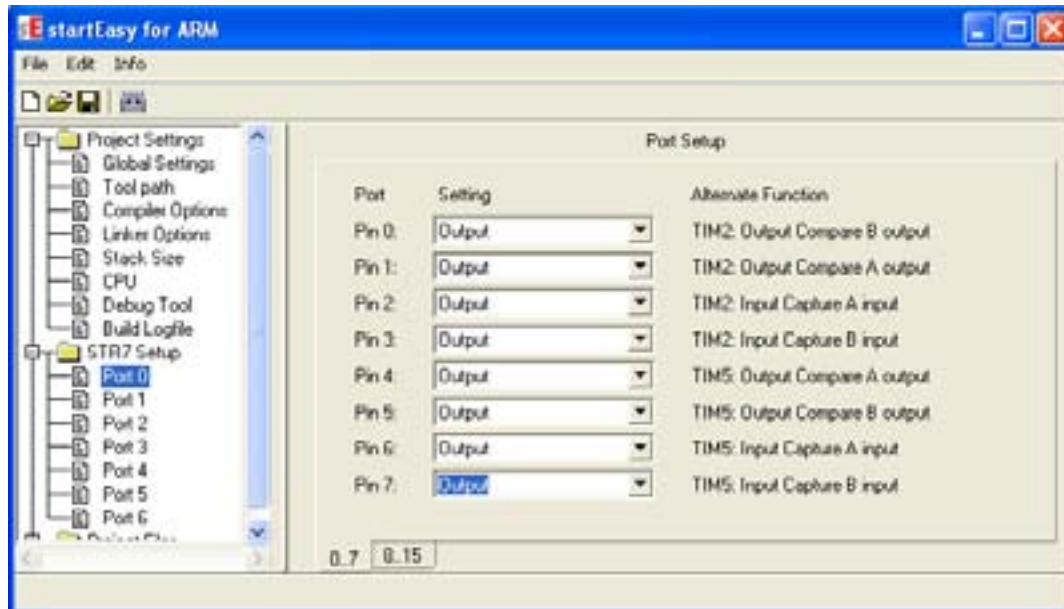
Now open the Debug dialog. Here the option must be set to specify the serial number of your Tantino7-9. You will find it on the base of the Tantino case. You must also select the version of the HiTOP IDE you are using. Here it is currently 5.20.



The final option, Build Logfile, allows you to review the build diagnostic messages generated when you create the project.



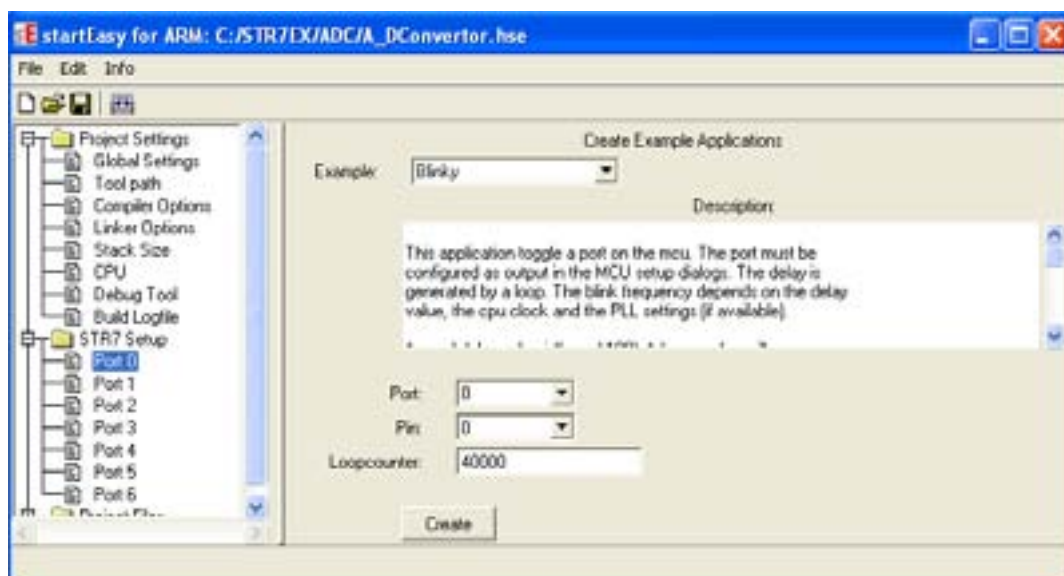
Once you have finished configuring the project settings, close the project setting folder and open the STR7 Setup folder. The options in this folder allow us to configure the STR7 bus, CPU clocks, the interrupt structure and the basic on-chip peripherals. For our “LED Chaser” project, we are going to illuminate the segments on the LED display on the evaluation board. To do this, we must configure the General Purpose IO. First open the PORT 0 dialog and select the tab for pins 0 to 7. In each dialog box select “Output”. Next click on the 8-15 tab and configure pins 8 and 9 also as output.



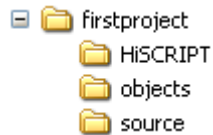
This finishes configuring the project and the STR7 for our simple “LED chaser” application. To generate the project, select Edit and Example Applications from the menu toolbar.



From the example drop-down box select the “Blinky” example. Then press the Create button.



This will generate all the project files into your nominated project directory:



Inside the project directory are three subdirectories. The source directory contains all the generated Assembler and C source files. The object directory contains the linker script file and the files generated by the compiler and linker. The compiler generates listing and object files and the linker produces a map file and an absolute file in ELF/DWARF format. The final directory is called the HiSCRIPT directory and is used to store any script files that are used within the debugger. Currently this directory is empty.

5.6.2 Project Structure

The project consists of three source files and three include files, as shown below:

Startup.s	These files contain the Assembler startup
Startup_stmicelectronics.s	code which configures the processor
Startup_generic.s	before you enter your C code
Main.c	This file contains the auto generated
	code and the main function
Interrupt.c	This file contains the default interrupt
	handlers
Main.h	Include file for Main.c
STR730.h	Include file to define the STR730 SFR's

The C source files contains a shell `void main(void)` function and the auto-generated code in a function called "init". There is also a set of comments that define sections where you should put your code such as declarations and functions. If you place your code within these comments you can change the options in StartEasy and update the project without overwriting your source code.

```

int main(void)
{
/* BEGIN USER CODE MAIN VARIABLES */

/* END USER CODE MAIN VARIABLES */

/* BEGIN USER CODE MAIN INIT1 */

/* END USER CODE MAIN INIT1 */

    init();

/* BEGIN USER CODE MAIN INIT2 */

/* END USER CODE MAIN INIT2 */

    while(1)
    {
/* BEGIN USER CODE MAIN LOOP */

        IO_SETPORT = IO_ON;
        for(i=0; i!=BLINKY_DELAY; i++);
        IO_CLRPORT = IO_OFF;
        for(i=0; i!=BLINKY_DELAY; i++);

/* END USER CODE MAIN LOOP */
    }
}
/* BEGIN USER CODE FUNCTIONS2 */

/* END USER CODE FUNCTIONS2 */

```

The include files are also structured to allow easy maintenance of external declarations. In each include file declare a function or global, as shown below.

```

#ifdef _MAIN_H_

/* BEGIN USER CODE LOCAL VARIABLES */
unsigned int UserVariable;
/* END USER CODE LOCAL VARIABLES */

/* BEGIN USER CODE LOCAL FUNCTIONS */
void UserFunction(void);
/* END USER CODE LOCAL FUNCTIONS */

#else

/* BEGIN USER CODE GLOBAL VARIABLES */
extern unsigned int UserVariable;
/* END USER CODE GLOBAL VARIABLES */

/* BEGIN USER CODE GLOBAL FUNCTIONS */
extern void UserFunction(void);
/* END USER CODE GLOBAL FUNCTIONS */

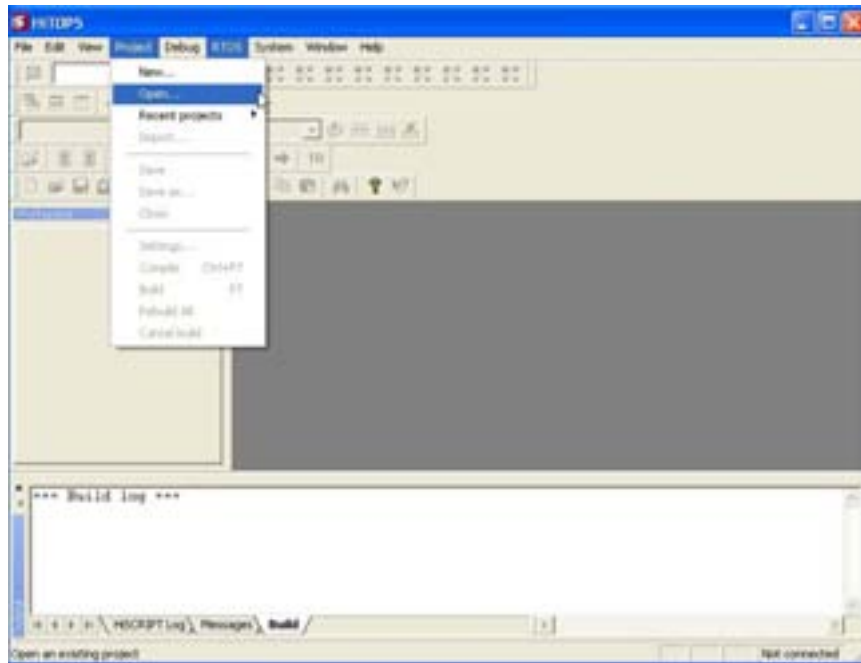
#endif

```

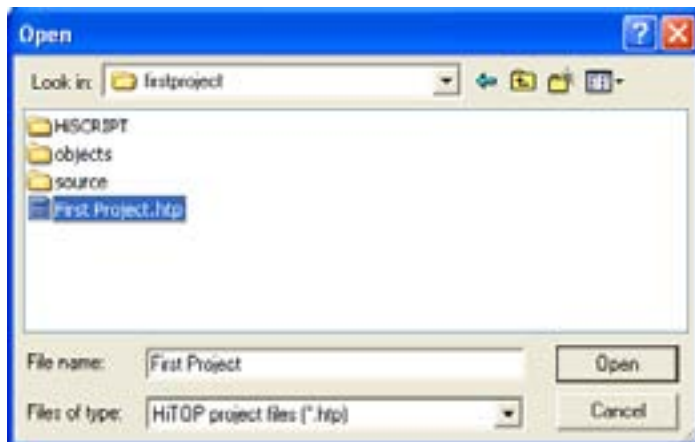
The `#ifdef` symbol is declared once in the home module. When the project is built the declarations are made when the home module is compiled and will be declared as externals in any other module where the include file is used.

5.6.3 HiTOP Debugger

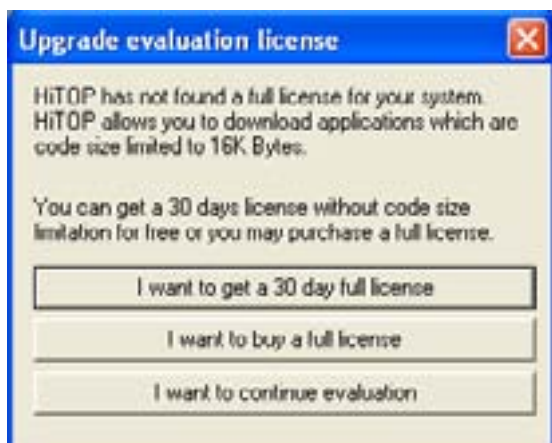
Once you have generated the example project, start HiTOP by double-clicking on the HiTOP icon:



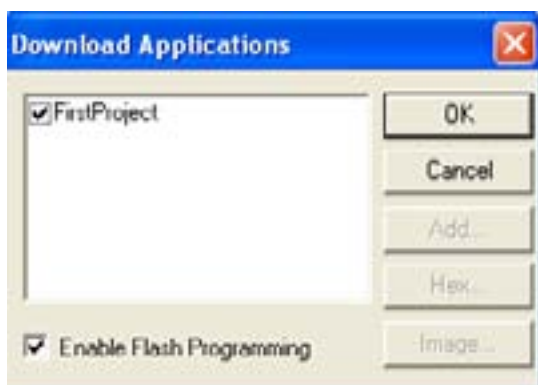
When the HiTOP program has launched, select “project/open project” and select the HiTOP project in the “Hello World” project directory. The project file extension is “.htp”. When you do this, make sure the evaluation board and the Tantino7-9 are connected.



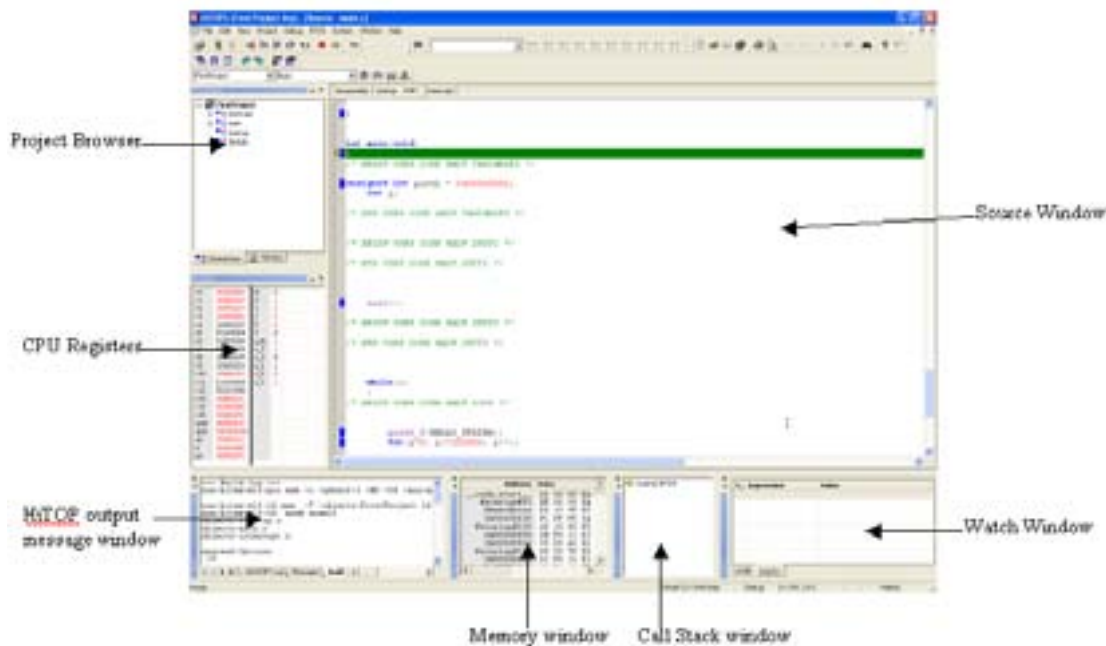
When you open the project, a reminder screen will be displayed. Click the “I want to continue evaluation” button. You also have the option to purchase a full licence or get a 30 day trail licence that has no code limitations.



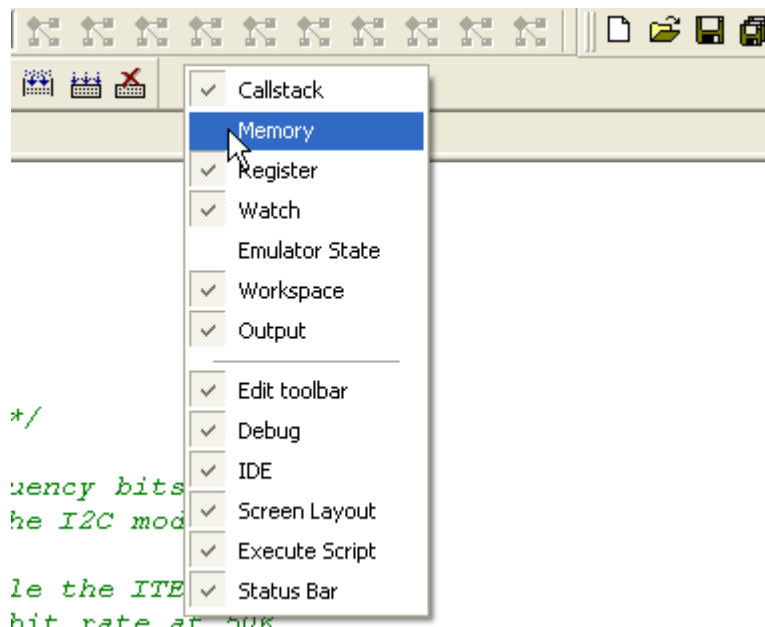
Once you have passed the reminder screen, HiTOP will start and ask if you want to download your application to the STR7 FLASH memory. Click OK to begin the FLASH download. If an error occurs at this point it will most likely be that the boot jumpers are incorrectly set on the evaluation board.



After the FLASH download has finished, your project will be fully loaded and ready for a debug session. The debugger and its main windows are shown below



If you close a HiTOP window it can be reopened by moving the mouse cursor onto an unused section of the toolbar, right-clicking the mouse and then selecting the window you wish to reopen. This menu also allows you to enable and disable the various toolbars.



The next sections are a tutorial on how to use the basic features of HiTOP to debug and edit your project.

5.6.3.1 Editing Your Project

To edit the code in your project, do the following:

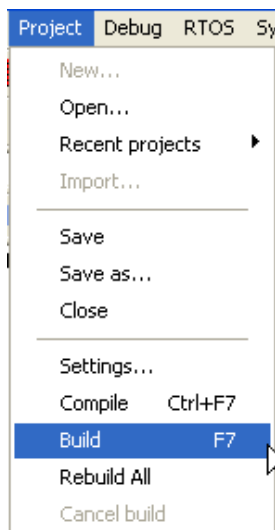
1. Select the main tab in the source window to display the code in this module.
2. Next right-click and select "Switch to edit mode"
3. So that we can flash each segment of the LED displays, edit the defines at the beginning of the main.c file to match the code shown below.

```
/* BEGIN USER CODE DEFINES */

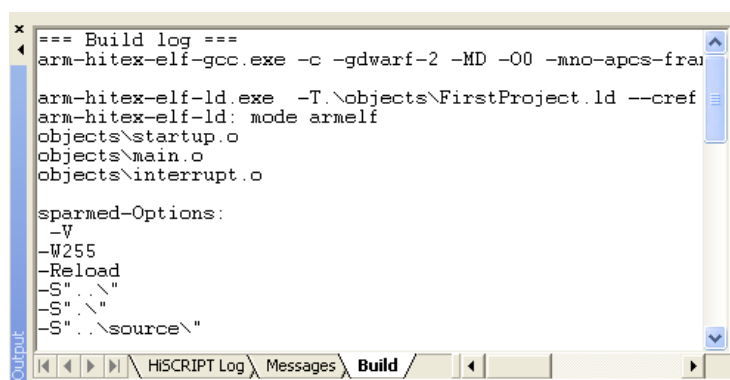
#define IO_SETPORT    IOPORT0_PD
#define IO_CLRPORT    IOPORT0_PD
#define IO_ON         0x000
#define IO_OFF        0x3FF
#define BLINKY_DELAY  40000

/* END USER CODE DEFINES */
```

4. To rebuild the code, select project/build on the main toolbar.



Reporting of the build progress is shown in the output/build window. If there are errors when you build the project you can click on the error report and the offending line of code will be displayed in the source window.

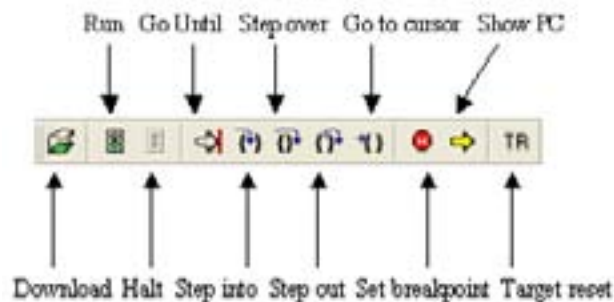


After the build has finished, the new code will be downloaded into the evaluation board.

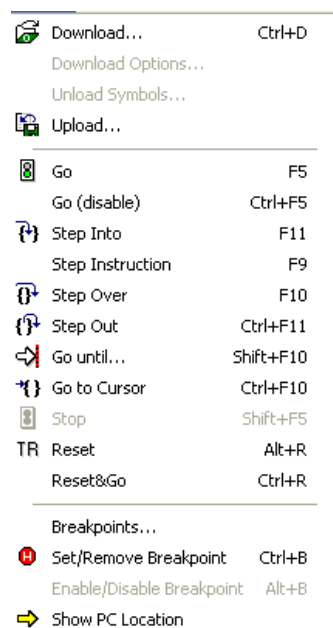
Once this has finished, right-click again in the source window and switch back into debug mode. Now you are ready to use HiTOP in its debugging mode.

5.6.3.2 Run Control

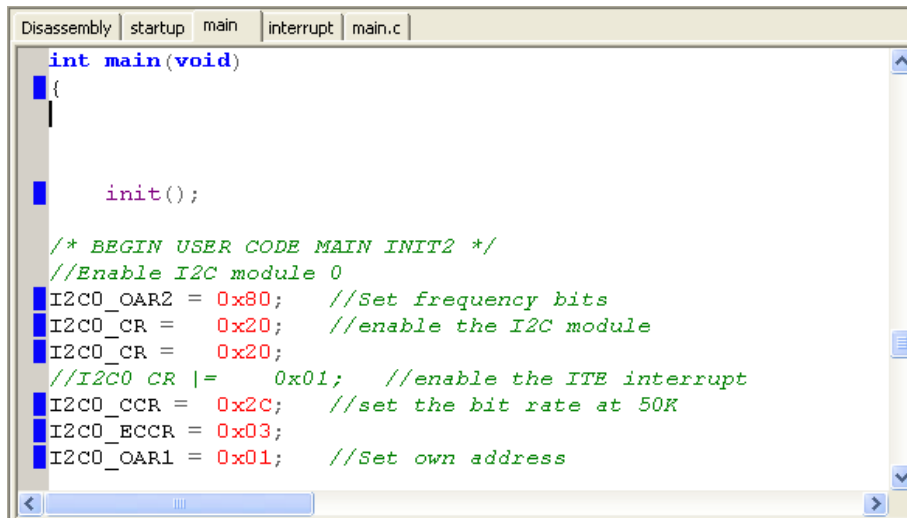
Once the project is loaded, the STR7 is reset and the program counter is forced to the reset vector. From here it is possible to execute code at full speed or single-step line-by-line. The debug toolbar has specific buttons to control execution of code on the ARM7 CPU.



These functions can also be accessed via the debug menu, which also displays the keyboard shortcuts. It is worth learning the keyboard shortcuts as these are the fastest way to control the execution of your code.



The source window allows you to browse your C code for any module in the project. There is also a disassembly window that will show you the actual contents of the program memory.



```

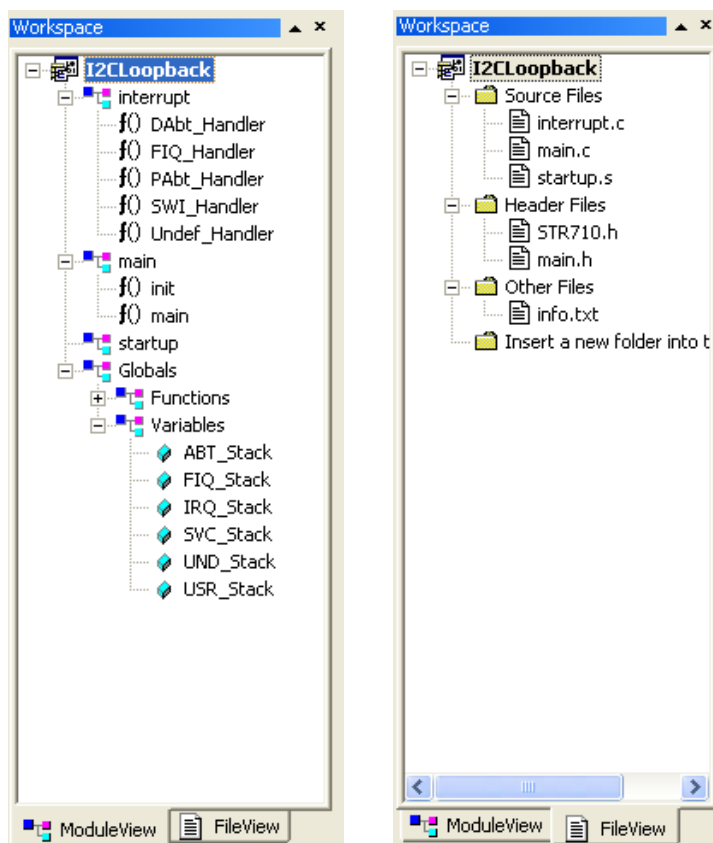
Disassembly | startup | main | interrupt | main.c
int main(void)
{

    init();

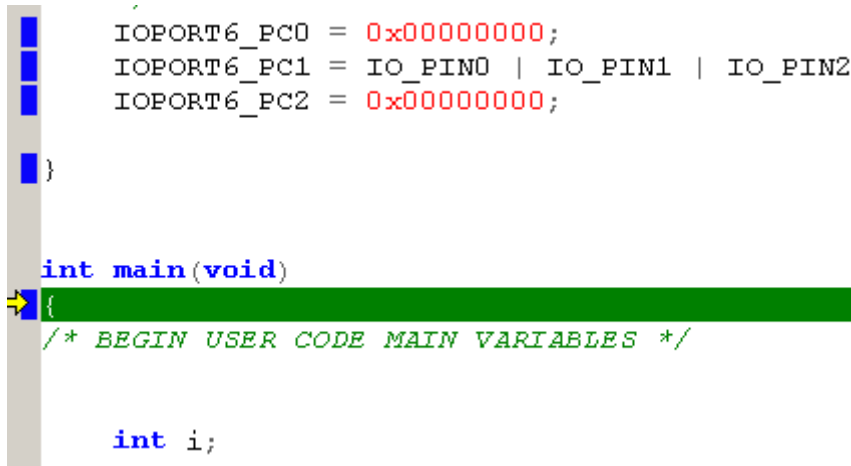
    /* BEGIN USER CODE MAIN INIT2 */
    //Enable I2C module 0
    I2C0_OAR2 = 0x80; //Set frequency bits
    I2C0_CR = 0x20; //enable the I2C module
    I2C0_CR = 0x20;
    //I2C0 CR |= 0x01; //enable the ITE interrupt
    I2C0_CCR = 0x2C; //set the bit rate at 50K
    I2C0_ECCR = 0x03;
    I2C0_OAR1 = 0x01; //Set own address

```

The Project Window allows you to browse your project. Double-clicking on a module or function name will open the selected file in the source code window.



The current location of the program counter is shown as a yellow arrow in the source window:



```

IOPORT6_PC0 = 0x00000000;
IOPORT6_PC1 = IO_PIN0 | IO_PIN1 | IO_PIN2
IOPORT6_PC2 = 0x00000000;

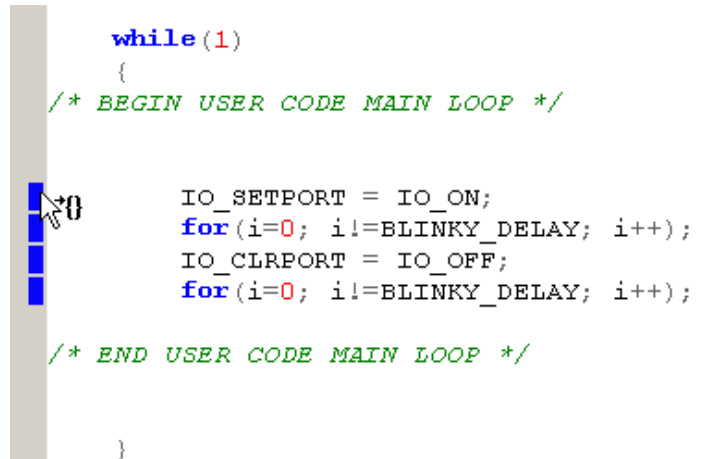
}

int main(void)
{
/* BEGIN USER CODE MAIN VARIABLES */

    int i;

```

The blue squares at the edge of the source window indicate an executable line of code. If there is no blue square, there is no code at this location. If you place the mouse icon over a blue square, a pair of braces is displayed.



```

        while(1)
        {
/* BEGIN USER CODE MAIN LOOP */

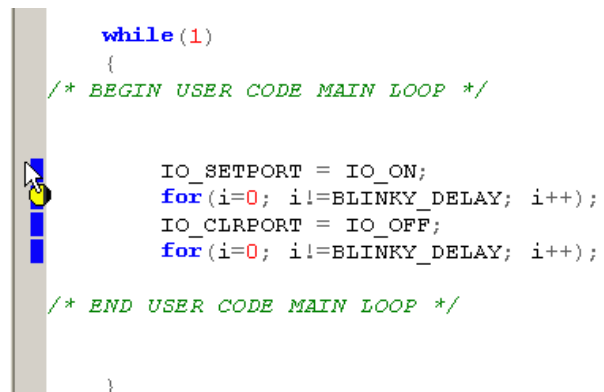
        IO_SETPORT = IO_ON;
        for(i=0; i!=BLINKY_DELAY; i++);
        IO_CLRPORT = IO_OFF;
        for(i=0; i!=BLINKY_DELAY; i++);

/* END USER CODE MAIN LOOP */

        }

```

If you now left-click, the code will be executed until it reaches this point. If you move the mouse pointer further to the left, the braces are replaced by a circle. If you left-click again, a breakpoint will be set and will be shown graphically as a red bar across the source code and a red circle in the margin.



```

        while(1)
        {
/* BEGIN USER CODE MAIN LOOP */

        IO_SETPORT = IO_ON;
        for(i=0; i!=BLINKY_DELAY; i++);
        IO_CLRPORT = IO_OFF;
        for(i=0; i!=BLINKY_DELAY; i++);

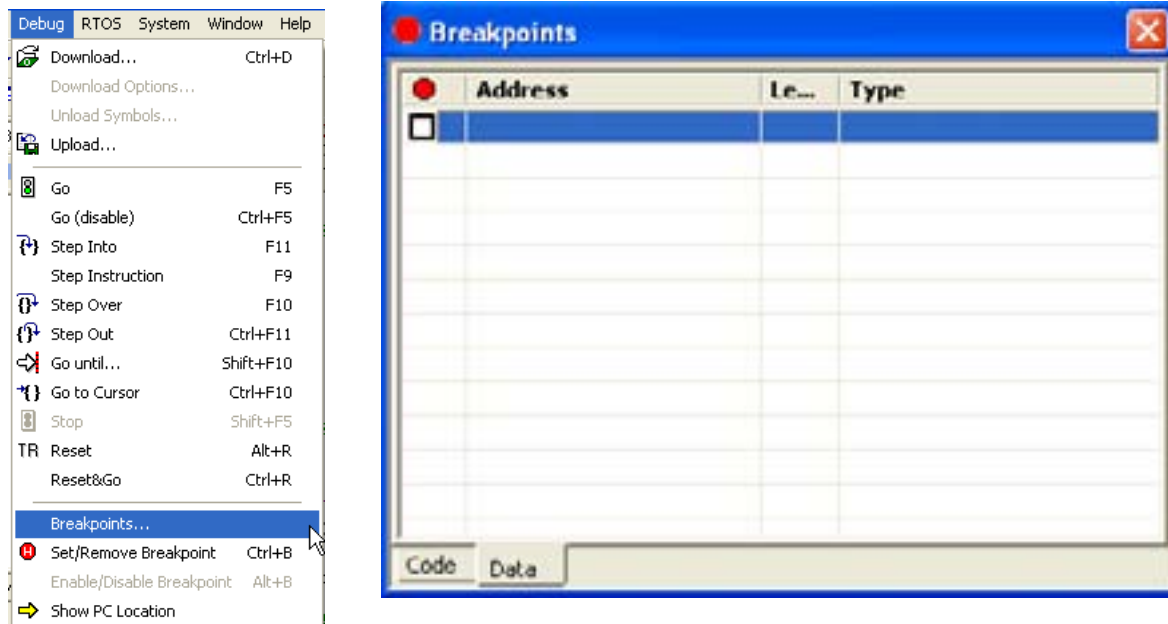
/* END USER CODE MAIN LOOP */

        }

```

The ARM7 TDMI JTAG module supports two hardware breakpoints. When you are debugging from FLASH, this requires careful management. However the Tantino also supports software breakpoints so building your application to run from RAM will allow additional breakpoints to be set.

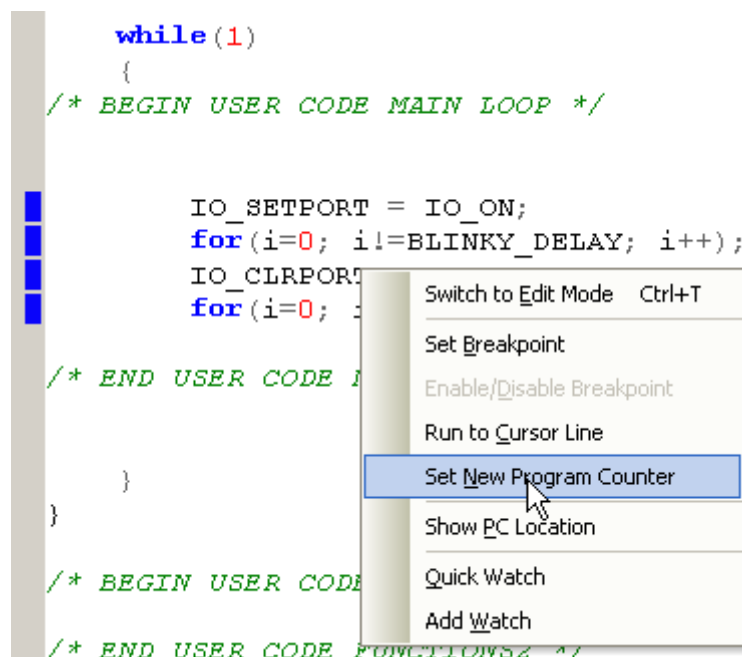
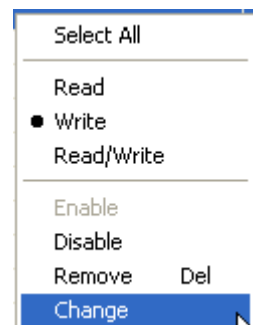
It is also possible to set breakpoints on data variables. This allows you to halt code when a variable is read or written too. Open the breakpoint menu under debug/breakpoint and select the data tab:



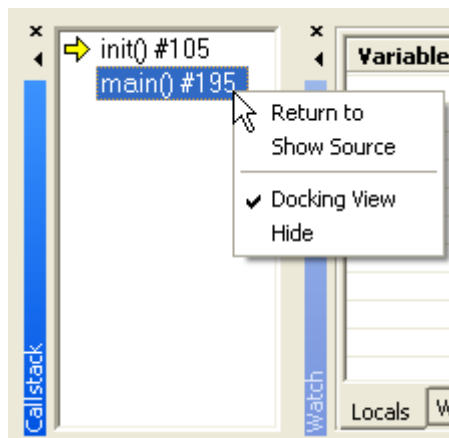
Now locate the variable you want to break on, either in the source window or the project browser, then drag-and-drop it to the breakpoint window. By default, the break condition is on a write to the variable. If you select the local options this can be changed to read or read/write. The change option gives you full access to programming the breakpoint condition

There are some more advanced breakpoint settings that allow the setting of breakpoints “on-the-fly” and conditional breakpoints and these are discussed in the “HiTOP project settings” section at the end of this first tutorial.

You may also position the program counter on any line of code. Locate the line of code where you want to place the program counter with the mouse pointer and left-click to locate the cursor. Next right-click and select “set new program counter”. This will force the Program Counter to this location. No other registers are affected so you must use this option with care.



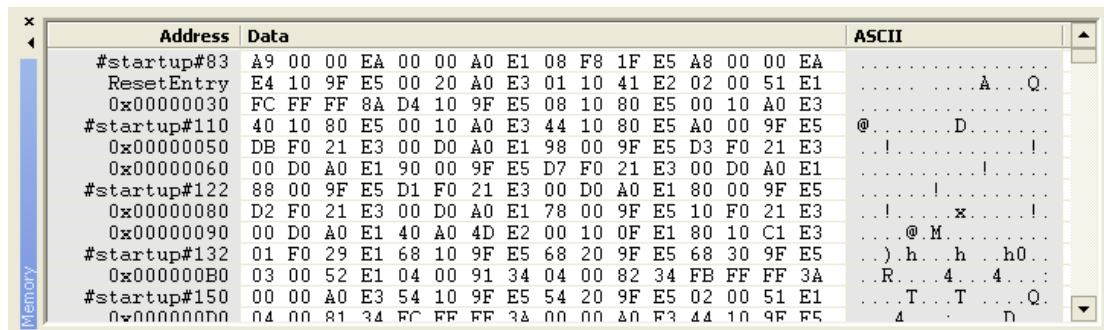
The callstack window displays the calling hierarchy of the functions pushed onto the stack. If you double-click on a function name, the source will be displayed at the point that the program will return to. The local options displayed by a right-click also allow you to run the program up to a selected return point.



Take some time to become proficient with running the code! You should be able to reset the target, run until main(), single-step the code, set breakpoints and reposition the program counter

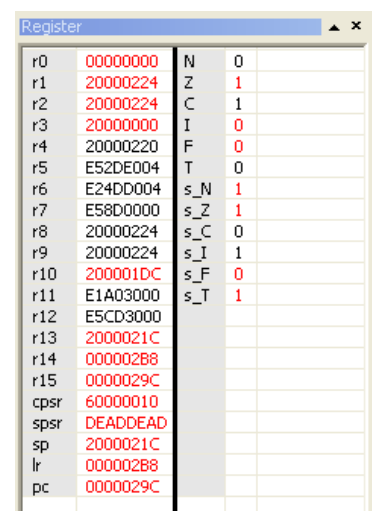
5.6.3.3 Viewing Data

As well as controlling the program execution, it is also possible to view the contents of any memory location within ARM7 address range. The memory window is the most basic method of viewing and changing the contents of any memory location.

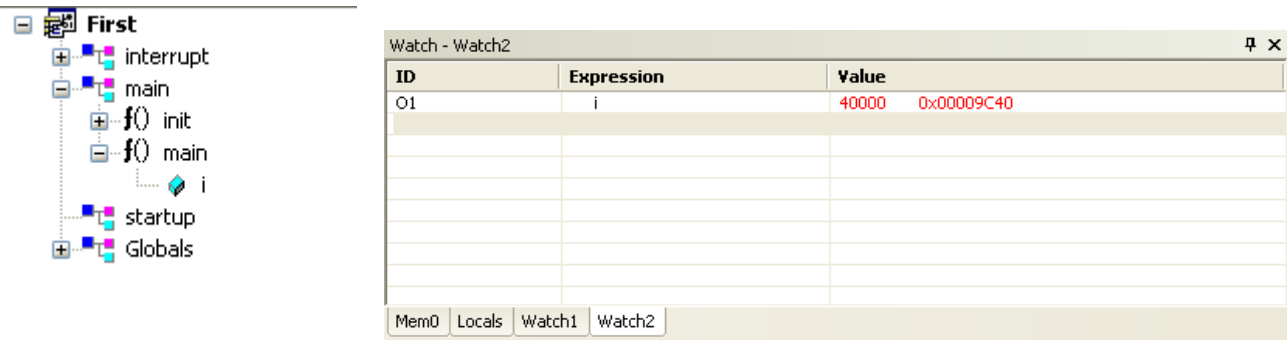


You can set the address range of the window by double-clicking on an entry in the address column and entering an absolute address or a symbolic name. The contents of memory locations can be changed by overtyping the values in the data or ASCII window. The more advanced options are available by right-clicking the mouse.

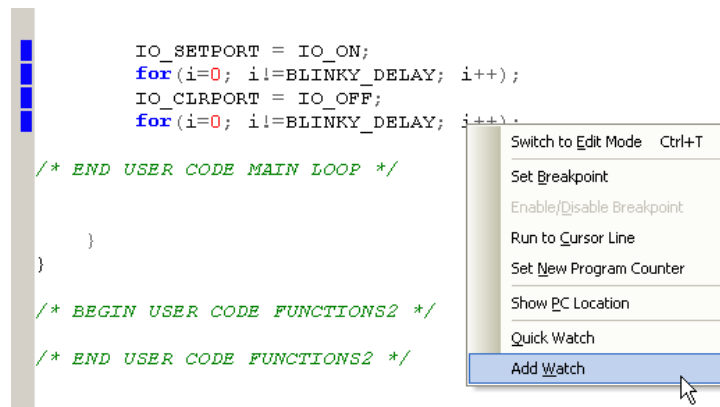
The register window gives you access to the active register bank along with the CPSR and SPSR. In this window you can modify the register contents and change the state of the CPSR/SPSR flags.



The watch window allows you to view program variables. You can edit the current value contained in the variable by simply double-clicking on the current value and entering a new value.



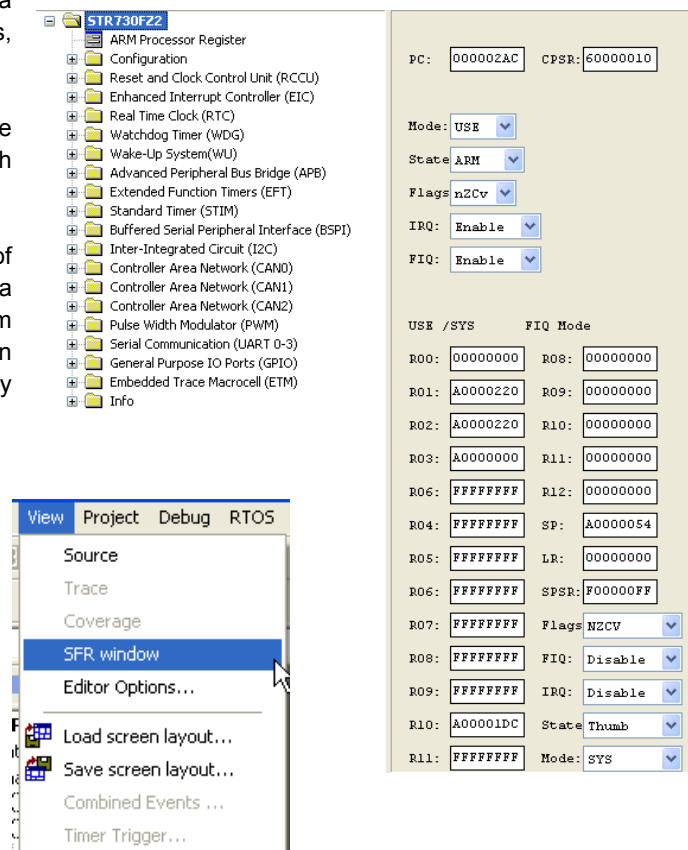
You can drag-and-drop variables into this window from the source code and from the project explorer, or use the right mouse button and select “Add Watch”.



The Watch window supports all C and C++ data types including complex objects such as classes, arrays, structures and unions.

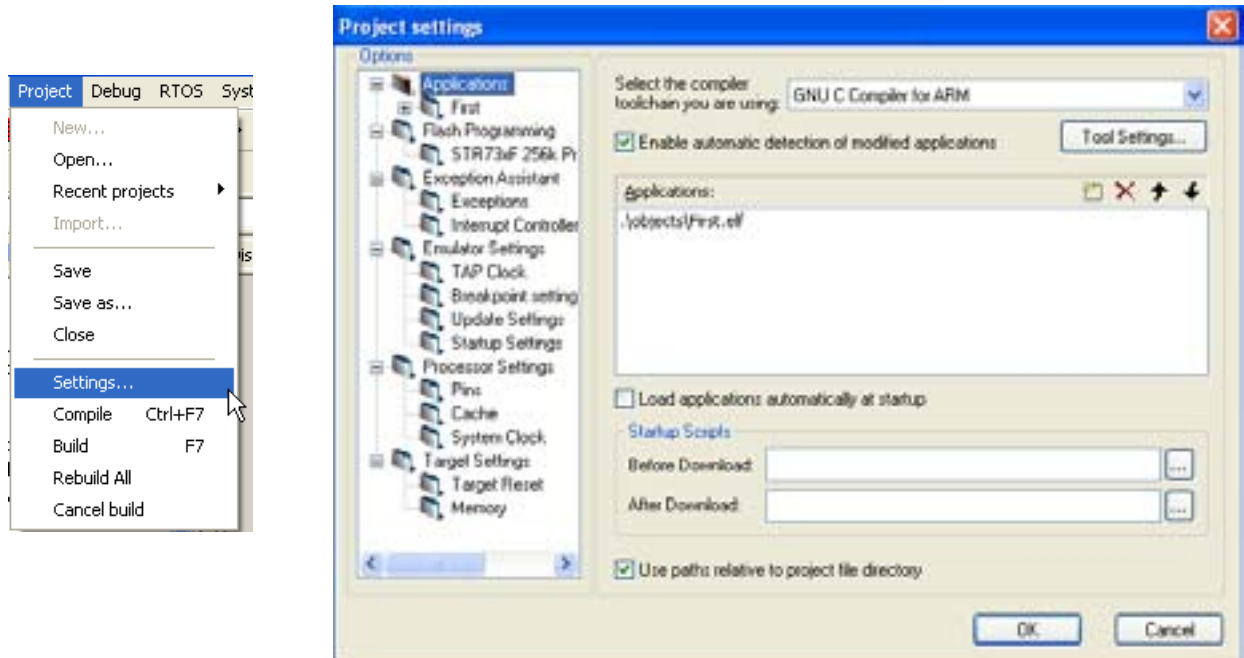
There are also dedicated windows for each of the STR7 peripherals. These can be accessed with the view\SFR window on the main toolbar

The SFR windows show you the configuration of all the STR7 special function registers in the data book format. This allows you to quickly confirm that a given peripheral is correctly setup. In addition, these windows allow you to manually control an on-chip peripheral



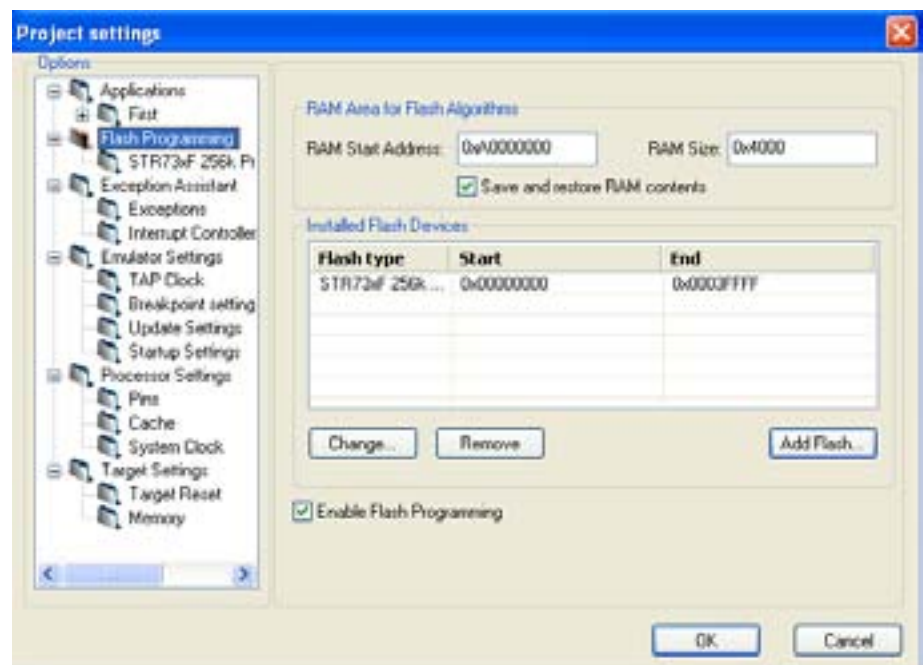
5.6.3.4 HiTOP Project Settings

Once you are familiar with the basic features of StartEasy and HiTOP, you may want to modify the project settings to begin work on your own project. This can be done by re-running StartEasy or directly within HiTOP. You can change the project settings within HiTOP via the Project\settings menu.



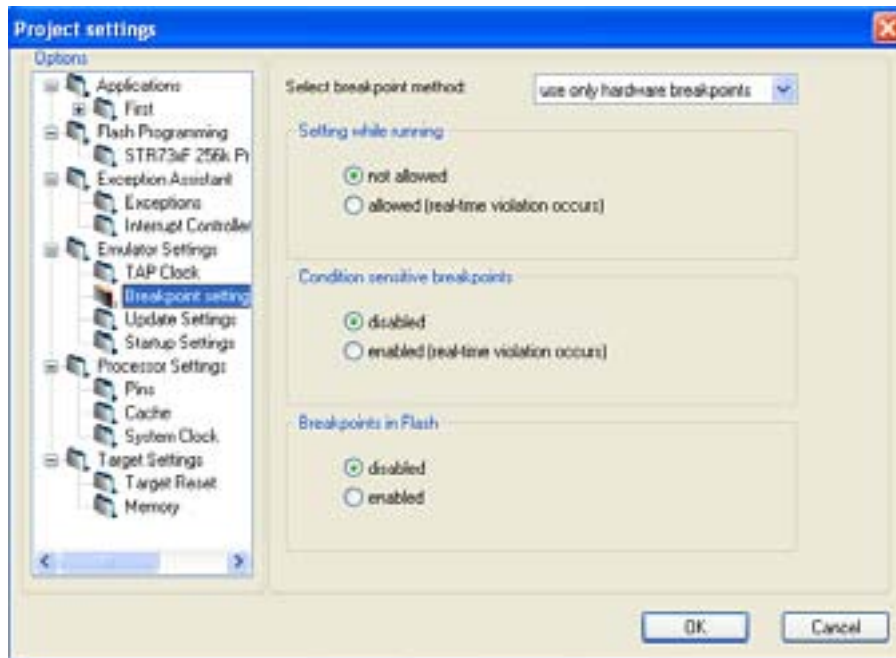
The applications menu allows you to select the compiler toolchain you are using. The default is the GCC compiler but a wide range of commercial compilers are also supported. Then you can select the application you want to debug. Here you select the .ELF file that is output from the compiler linker that you are using.

Once you have selected the project file and compiler tool, the FLASH programming section allows you to select the programming algorithm for the FLASH device you are using. This supports the STR7 on-chip FLASH memory and a wide range of FLASH memory chips, if you are using external FLASH memory. In this menu you must specify an area of RAM that the debugger can use for the programming algorithm during download. If you check the "save and restore RAM contents" box, the contents of this region will be preserved. If you uncheck it you must perform a target reset after download to restart the application. However the download process will be faster.



5.6.3.5 Advanced Breakpoints

In the emulator settings menu, the “TAP clock” is configured for the ARM7 microcontroller you are using and does not need to be changed. However the breakpoint settings menu does have several important options. First it is possible to force the JTAG to use software or hardware breakpoints. If you are debugging from RAM a software breakpoint will replace the application opcode with a breakpoint instruction, although this is hidden from the user.

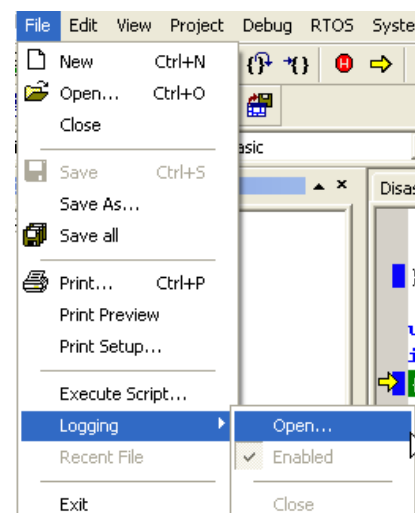


This does not use the hardware breakpoint registers and enables you to set more than two breakpoints. Setting the “while running” option allows you to set and clear a breakpoint without halting the code. This can be extremely useful when you are debugging a complex real-time application. The condition-sensitive breakpoint option allows you to set a breakpoint on an ARM instruction that is conditionally executed. If this option is enabled, the code will only halt when the instruction’s condition codes match the CPSR. Again this is extremely useful when debugging real ARM code. The JTAG hardware is limited to two hardware breakpoints. If you are debugging from RAM you can set multiple breakpoints and the HiTOP will use software breakpoints. This technique can be extended to code which is being debugged out of FLASH by enabling the “Breakpoints in FLASH” option. This will reprogram the FLASH with software breakpoints prior to starting the code running. This is slower but allows you to set as many breakpoints as you want. The remaining processor and target options are specific to the STR7 and will not generally need to be changed.

5.6.3.6 Script Language

The HiTOP debugger also contains a script language called HiSCRIPT. This is a C-like language that can be used to build software test harnesses or simply automate common sequences within HiTOP.

From the file menu select “file\logging\open” and create a file called ResetGoMain.scr in the HiSCRIPT directory of your project. Make sure you have selected the .scr extension. Select the Log mode as “Commands Only” and overwrite the existing file.

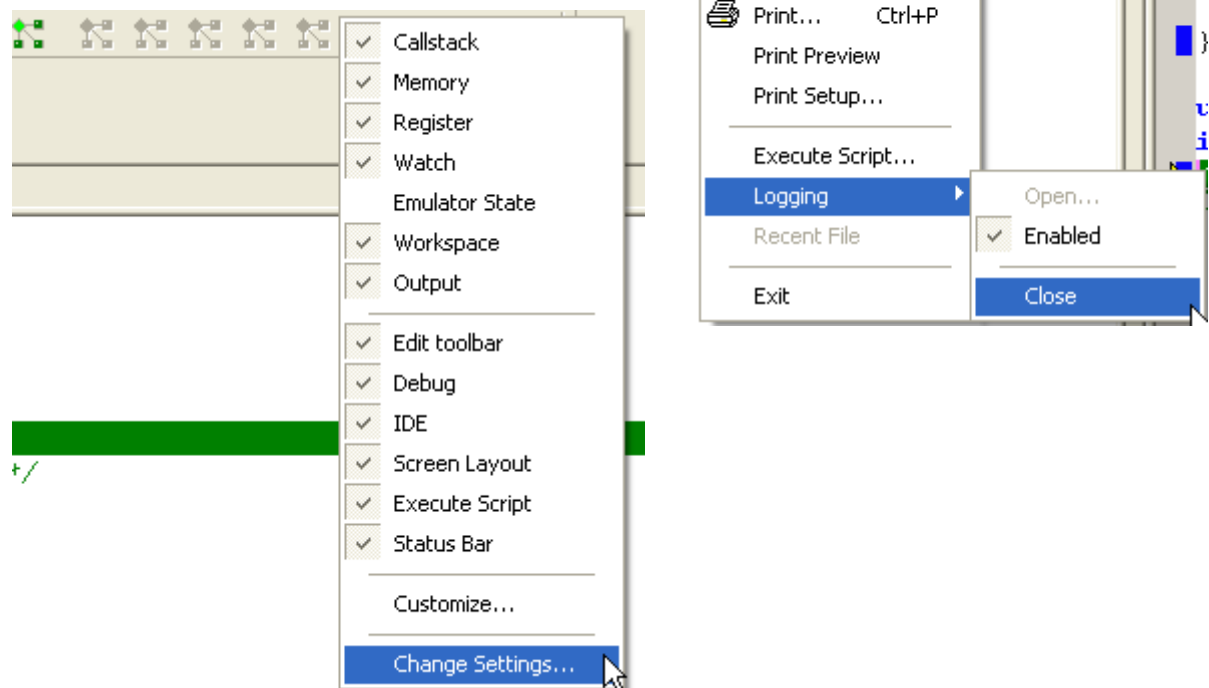


Now select OK and any HiTOP instructions you do within the debugger will be saved as HiSCRIPT commands. Once you have enabled logging, perform a target reset and a go until main(). Again, select file logging and close option to stop the command logging.

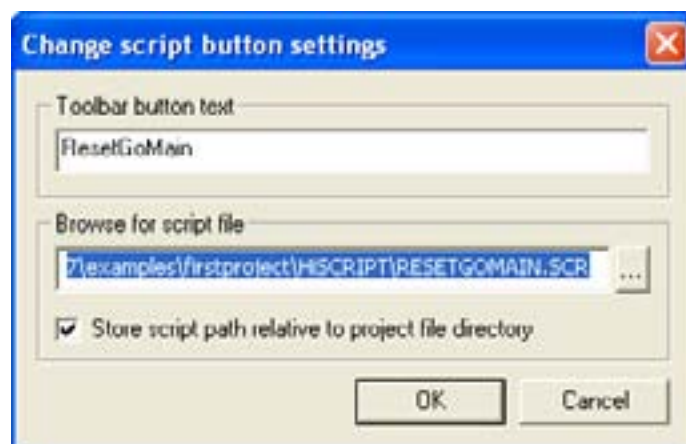
This will generate a HiSCRIPT file that will replay your instructions. The actual HiSCRIPT file contains some extra instructions but it can be edited down to the following minimal instructions:

```
RESET TARGET
GO UNTIL main
```

The HiSCRIPT file can be added to the toolbar by highlighting the script toolbar, right-clicking and selecting change settings.



Then in the change settings dialog, enter a symbolic name for the script and the filename of the HiSCRIPT file.



The script can then be executed from the toolbar:

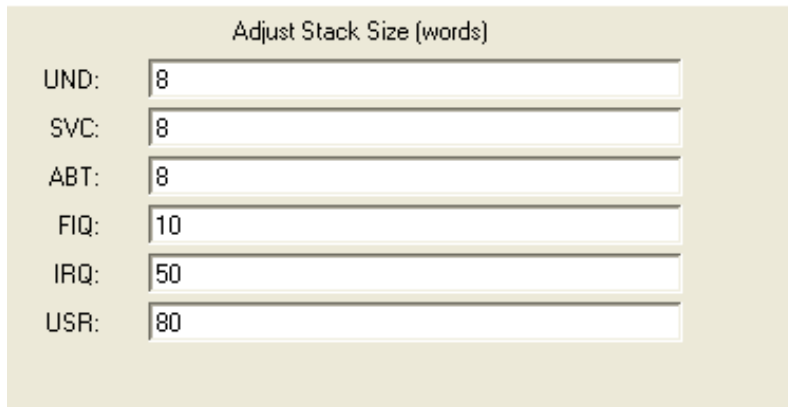


This is an important tutorial. You should explore all the features of HiTOP and StartEasy so that you can easily define, edit and debug an application program before proceeding to the next sections!

5.7 Exercise 2: Startup Code

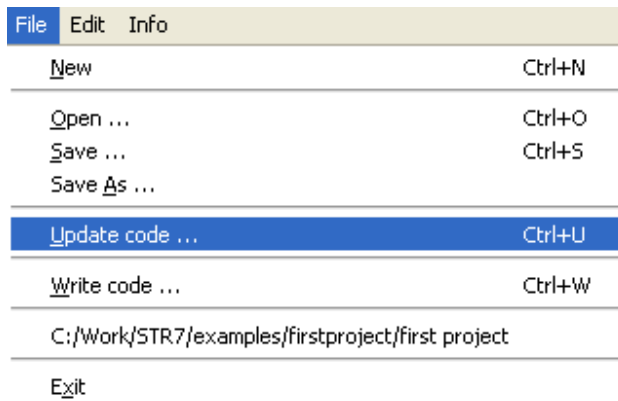
In this exercise we will configure the compiler startup code to configure the stack for each operating mode of the ARM7. We will also ensure that the interrupts are switched on and that our program is correctly located on the interrupt vector.

1. Open the StartEasy project in EX2 Startup directory
2. Open the project setting stack size tab and set each of the stacks to the values shown below



Adjust Stack Size (words)	
UND:	8
SVC:	8
ABT:	8
FIQ:	10
IRQ:	50
USR:	80

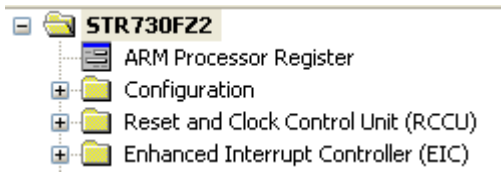
3. Open the project settings debug tool dialog and enter your Tantino serial number
4. Select file\update code and regenerate the project.



Open the HiTOP project and load the new project into the FLASH memory

5. Reset the target and then select the Go Until option and run the code to main

6. Open the view\SFR window and then select ARM processor registers. In this window examine the state of the CPU and the configuration of each operating mode.



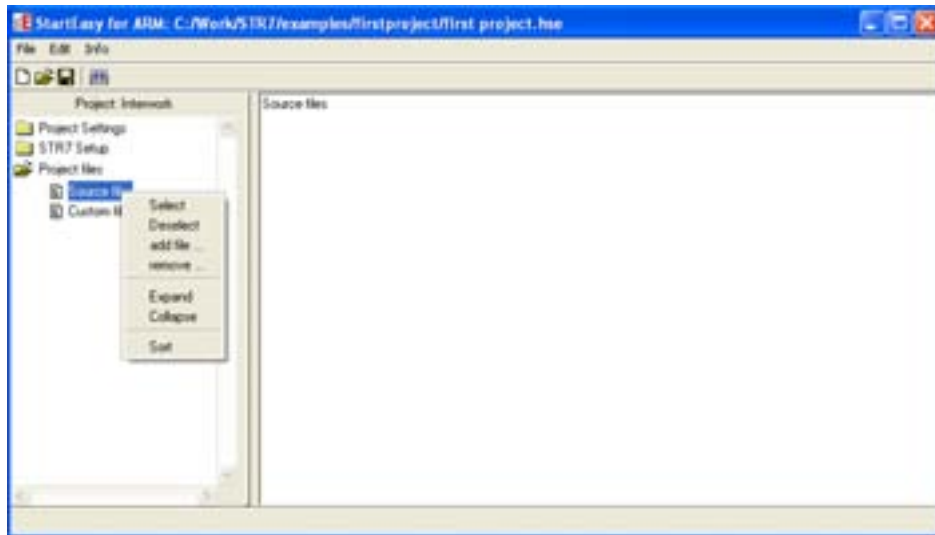
7. In the source code window, select the startup tab and view the startup source code. Scroll through until you find the vector table and familiarise yourself with this Assembler file.

5.7.1 Exercise 3: Interworking ARM & THUMB Instruction Sets

In this example we will build a very simple program to run in the ARM 32-bit instruction set and call a 16-bit THUMB function and then return to the 32 bit ARM mode.

Open the StartEasy project in EX3 Interwork

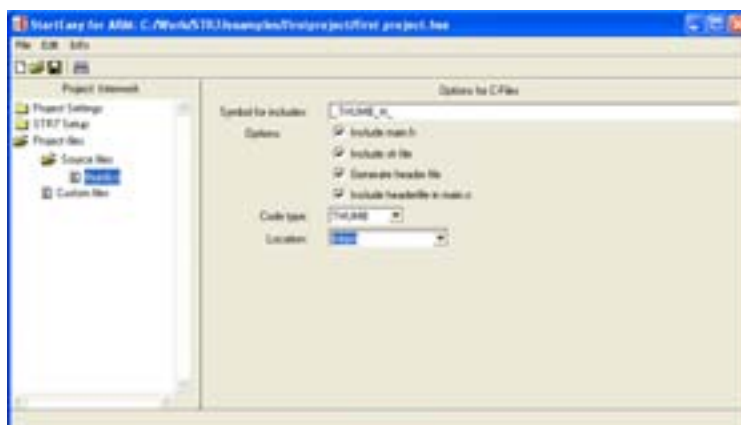
1. Select Project files\source files then right-click and select add file:



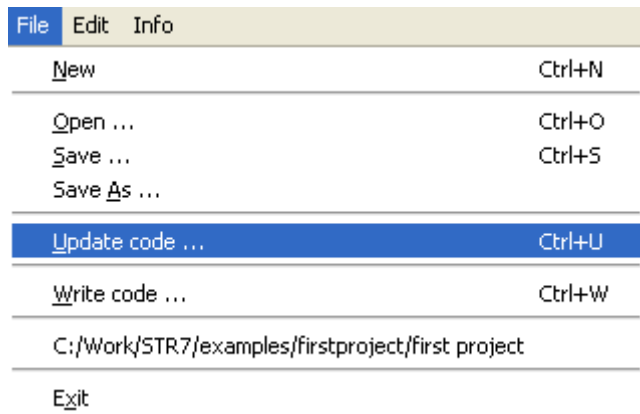
2. Add the file THUMB.c to your project.



3. In the dialogue for THUMB.c select THUMB as the code type. This will cause all code within this module to be compiled in the THUMB instruction set. Also tick all the boxes to generate and include header files. Finally select the code location as "intern" to locate the code in the on-chip FLASH.



4. Select file\update code and regenerate the project



5. Open the HiTOP project file and download the Interwork application into the STR7 FLASH memory
6. Reset the target and run to main()
7. In the source window select the disassembly window and check that the instructions are compiler as ARM 32 bit instructions. You can also check that the T bit in the CPSR is set to zero for ARM execution.

Each ARM (32 bit) instruction is four bytes long




0x00001E8	1EFF2FE1	bx lr
#144		
0x00001EC	04E02DE5	str lr, [sp, #-4h]
#156		int();
0x00001F0	D1FFFFEB	bl int
#167		thumb();
0x00001F4	CDFFFFEB	bl __code_end__

N	0
Z	1
C	1
I	0
F	0
T	0

The T bit is clear

8. Run the code up to the call to the THUMB function, open the disassembly window and single step into this function to observe the switch from ARM to THUMB code.

In Thumb mode each instruction is two bytes long



#35		IOPORT2_PD = 0x0000FF00;
T:0x00000210	024A	ldr r2, [pc, #2h] ; 21ch
T:0x00000212	FF23	mov r3, #ffh
T:0x00000214	1B02	lsl r3, r3, #8h
T:0x00000216	1360	str r3, [r2, #0h]
#36		}
T:0x00000218	7047	bx lr

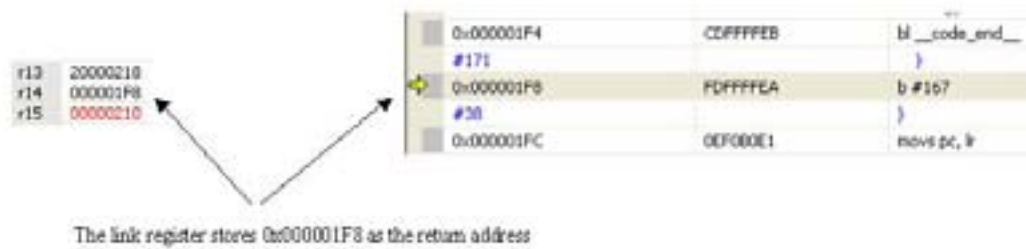
Return to the ARM calling function with a branch exchange on the contents of the link register

9. Observe the switch from 32-bit to 16-bit code and the THUMB flag in the CPSR.

N	0
Z	1
C	1
I	0
F	0
T	1

The T flag is set when you are in Thumb mode

10. Note the contents of the link register, single step (F11) until you return to the ARM code. Check the return address matched the value stored in the link register. **Note: the actual return address in your program might not be identical to that shown here!**



5.8 Exercise 4: Software Interrupt

In this exercise we will define an inline Assembler function to call a software interrupt and place the value 0x02 in the calling instruction. In the Software Interrupt SWI, we will decode the instruction to see which SWI function has been called and then use a case statement to run the appropriate code.

1. Open the StartEasy project in the SWI directory
2. In the project settings\debug tool menu, enter the serial number on the base of your Tantino
3. Update the project and open the new HiTOP project
4. Execute the program up to the first software interrupt call

```

while(1)
{
/* BEGIN USER CODE MAIN LOOP */
SoftwareInterrupt1; //Generate software interrupt

```

5. Switch to disassembler mode and examine the SWI opcode and note the address of the instruction.

#176		SoftwareInterrupt1;
0x000001F8	010000EF	swi 1h

6. Step the software interrupt instruction (F11) and see the jump to the SWI interrupt vector.

#81		B SWI_Handler
0x00000008	A30000EA	b SWI_Handler

7. Continue single stepping to enter the SWI interrupt handler.
8. Run the code to the switch statement.
9. Observe the contents of the link_ptr This should be the SWI instruction address + 4

#40		temp = *(link_ptr);
0x000002A0	0E30A0E1	mov r3, lr
0x000002A4	043043E2	sub r3, r3, #4h
0x000002A8	003093E5	ldr r3, [r3]

10. Observe the contents of the temp variable. This should be the value of the ordinal encoded into the SWI instruction.

#42		switch (temp)
0x000002B8	98309FE5	ldr r3, [pc, #4]
0x000002BC	003093E5	ldr r3, [r3]

11. Run the code to the closing brace of the SWI interrupt handler and observe the ISR exit code

0x00000350	0C40BDE8	ldmia sp!, {r2,r3,r14}
0x00000354	0EF0B0E1	movs pc, lr
0x00000358	20020020	dw 20000220h

12. Finally run the program at full speed to see the LEDs FLASH in a new and interesting way

Note: The C source for the SWI interrupt handler is in the module Interrupt.c

5.9 Exercise 6: STR730 System Memory Mode Bootloader

This example demonstrates using the on-chip bootloader to load a simple LED-flashing program into the on-chip SRAM at 0xA0000000. It is the basis of the full Hitex STR73x serial-bootstrap FLASH programming system. There are some complex programming challenges writing the small programs that can run solely in the on-chip SRAM. The GNU-ARM compiler used here has no specific extensions of this type of programming so some « brute-force » techniques are required. The usual STARTUP.S CPU configuration file is not required for example and the linker control file needs to be modified to place all code and data in the SRAM. An added complication is that debugging via JTAG is not supported in SystemMemoryBoot mode.

The SystemMemoryBoot program leaves the CPU in THUMB mode with the supervisor stack at 0xA0002000 (i.e. the top of the SRAM). The UART0 is fully configured and the baudrate automatically determined from measuring the duration of the bootloader trigger byte of '0'.

The example consists of three parts :

BOOTLOADER : A C-coded THUMB program of under 128 bytes in length that is loaded by the SystemMemoryBoot ROM to 0xA0000000. It receives the GPIO application and places at 0xA0000100, switches to ARM mode and jumps to it.

GPIO : A simple C application that prints digits to the LED arrays on the board. It has been built to run at 0xA0000100 and uses ARM instructions. It is called from the BOOTLOADER application automatically.

INLINE730.EXE : A PC application that runs from a MS-DOS command line that performs the following actions :

- (i) Sends the '0' bootloader trigger byte
- (ii) Received the '0x95' bootloader response byte
- (iii) Sends the BOOTLOADER application via the PC COM1 port
- (iv) Sends the GPIO LED-flashing example application via the COM1 port

You can change the COM port used by the loader by editing the BOOTLOAD.BAT batch file:

Use COM1 (default)

```
inlinestr730.exe dummy.hex com1 0 19200
```

Use COM2

```
inlinestr730.exe dummy.hex com2 0 19200
```

5.9.1 Running The Bootloader Example

Connect the RS232 channel 1 (RS232_1 on the board) to your PC's COM port. With the STR730 board powered up, move the M1 jumper to the closed position. Press the RST reset button and then move the M1 jumper back to the default open position. The STR730 will now be in SystemMemoryBoot mode. Now run the BOOTLOAD.BAT batchfile in the BOOT\PC_LOADER directory. The GPIO program will be loaded and run so that the LEDs will count from 0-0xF.

```
Hitex <UK> STR730 Bootloader Example v1.05
*** Make Sure STR730 Is In Bootstrap Mode!
COM Port = com1
Parameter 3 = 0 RS232
Baudrate = 19200

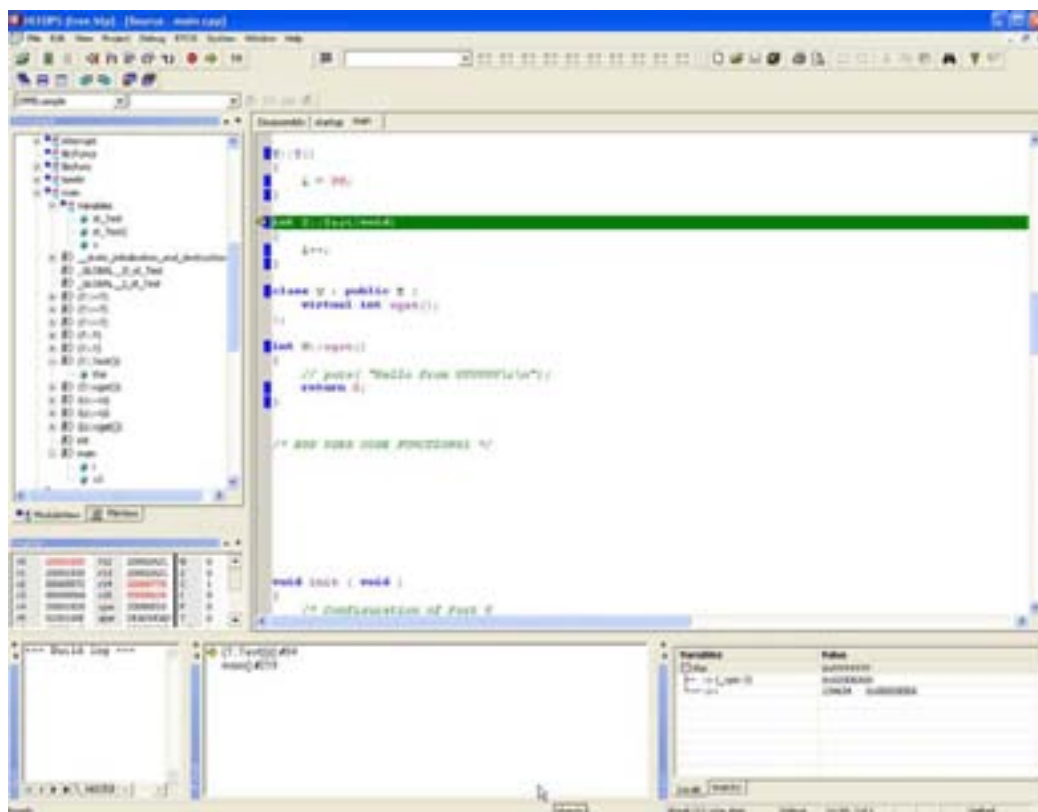
Bootloader Process A Resounding Success
C:\STR730Examples\Boot\PC_loader>_
```

Simple C application
loading via bootloader.

5.10 Exercise 5: C++

This example demonstrates building a C++ project with the GCC compiler and starting the debug session in HiTOP.

1. Open the StartEasy project in the CPP directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. Update the project and open the new HiTOP project
4. Run the application to main()



C++ support in HiTOP5 includes class awareness

5.11 Exercise 7: FLASH Programming

This program demonstrates the basic erase and write routines required to store program code or constants into the STR7 FLASH memory.

1. Open the StartEasy project in the FLASH directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. Update the project and open the new HiTOP project
4. Run the code to main and set the memory window to 0x0008000

Memory - Mem0				
Address	Data			
0x0008000	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008010	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008020	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008030	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008040	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008050	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008060	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008070	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008080	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
0x0008090	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF

5. Set a breakpoint on the second call to the SRAM_erase_FLASH_func () function.

```
// ERASE sector 4 at 0x8000 again
// PUT YOUR BREAKPOINT ON THE NEXT LINE!
error_status = SRAM_erase_FLASH_func(0x10) ;
if(error_status != FLASH_OK)
{
```

6. Run the code until it hits the breakpoint.
7. Check in the memory window that the FLASH has been updated with the new pattern.

Memory - Mem0				
Address	Data			
0x0008000	55555555	AAAAAAAA	FF	
0x0008010	FFFFFFFF	FFFFFFFF	FF	

8. Run the erase_FLASH function a second time and check that the contents of the FLASH sector have been reset to 0xFF.

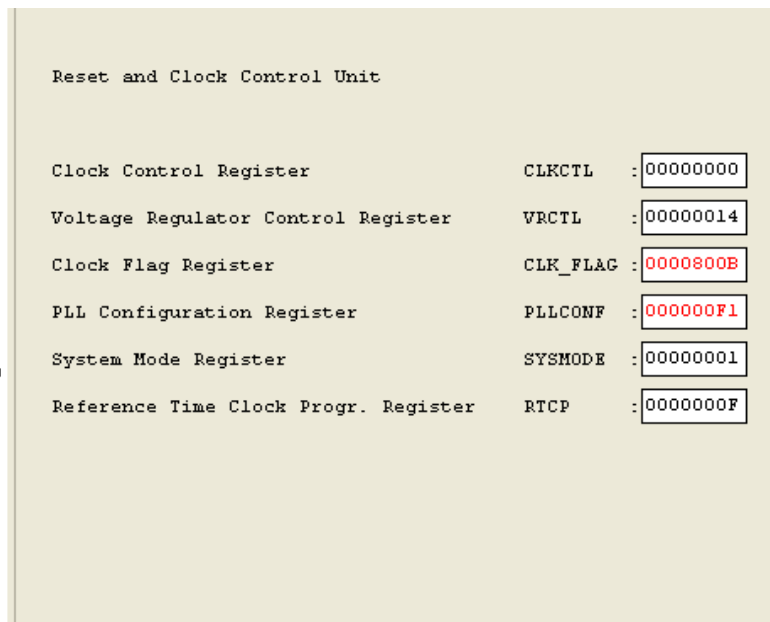
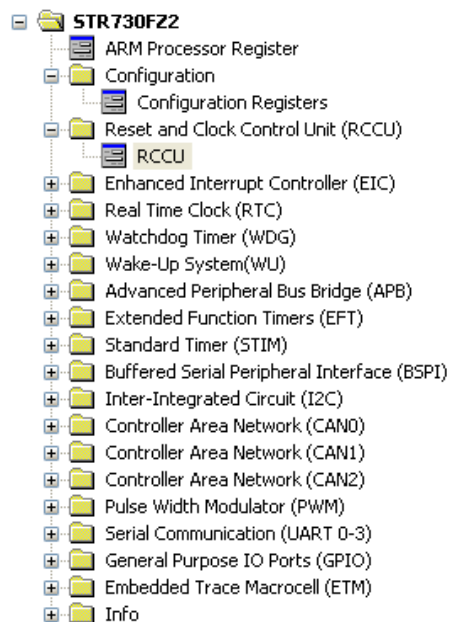
Memory - Mem0				
Address	Data			
0x0008000	FFFFFFFF	FFFFFFFF	FF	
0x0008010	FFFFFFFF	FFFFFFFF	FF	

Note that the functions that erase and write the FLASH have been copied into SRAM as it is not possible to execute from the FLASH bank which is currently being programmed, even if it is in another sector.

5.12 Exercise 8: Clock Configuration

In this example we will configure the Clock module to generate an MCLK of 32 MHz. The code to configure the clock is contained in the .S assembler code that runs before main.

1. Open the StartEasy project in the Clock directory.
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. Update the project and open the new HiTOP project
4. Run the code to main()
5. Open the View SFR\reset,clock control unit window and check the clock settings are configured correctly.



5.13 Exercise 9: Low Power Modes

In this example we will place the STR7 in a low power configuration and then cycle it through its low power modes. If you connect an ammeter inline with the PSU you will be able to see the power reduction at each stage.

1. Open the StartEasy project in the PowerDown directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. Update the project and open the new HiTOP project
4. Start the code running and press the INT interrupt button. Each time you press the button the interrupt routine will place the STR7 in a different power mode as follows:
 - i. Low power configuration
 - ii. Slow mode
 - iii. Wait for Interrupt
 - iv. Halt
 - v. Stop
5. Run through each of the powerdown modes and observe the power consumption on the meter.
6. When you reach standby mode the port pins will become tristated. This will cause the attached LEDs to slowly switch off as the remaining power is drained.
7. Finally, if while experimenting with the low power mode you lock out the Tantino debugger, change the bootmode pins so that the STR73x boots from RAM. This will allow you to regain control of the STR73x and erase the FLASH memory.

5.14 Exercise 10: IRQ Interrupts

This example uses the watchdog to generate an IRQ interrupt. This example examines the action of the EIC.

1. Open the StartEasy project in the IRQ directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. Update the project and open the new HiTOP project.
4. Run the code until it reaches the while(1) loop.

```

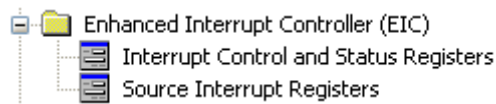
/* BEGIN USER CODE MAIN INIT2 */
EIC_SIR41 = ((unsigned int)Watchdog_IRQ_isr + 4) << 16 | 1; //Load the vector register
EIC_IER1 = 0x00000200; // Enable irq interrupt channels */
EIC_ICR = EIC_IRQ_ENABLE; // EIC interrupt enable register */

WDG_MR = 0x00000001; //enable the interrupt
WDG_CR = 0x00000002; //start the count

while(1)
{
/* BEGIN USER CODE MAIN LOOP */
/* END USER CODE MAIN LOOP */
}

```

5. Examine the interrupt configuration code for the EIC
6. Open the view\SFR\EIC\ General registers. Check that the IRQ_EN bit is enabled.



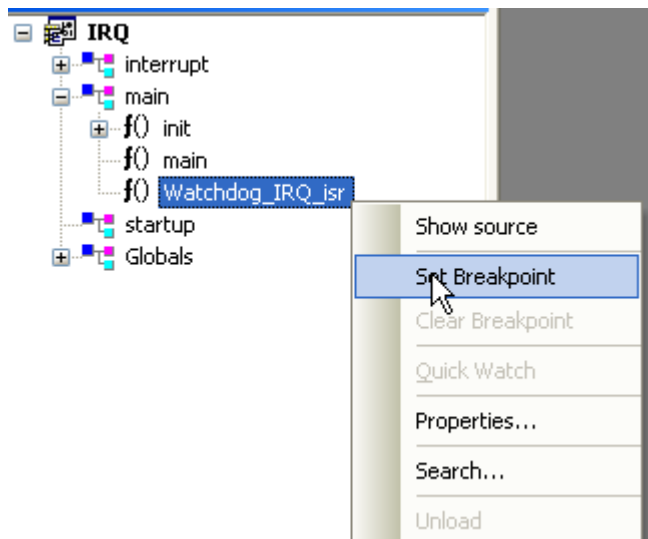
Interrupt Control Register

EIC_ICR : 00000001

7. Next open the Enable and pending window and check that channel 5 is open but not pending. Check that the Watchdog interrupt routine is at address 0x00000114

EIC_SIR41 : 01140001

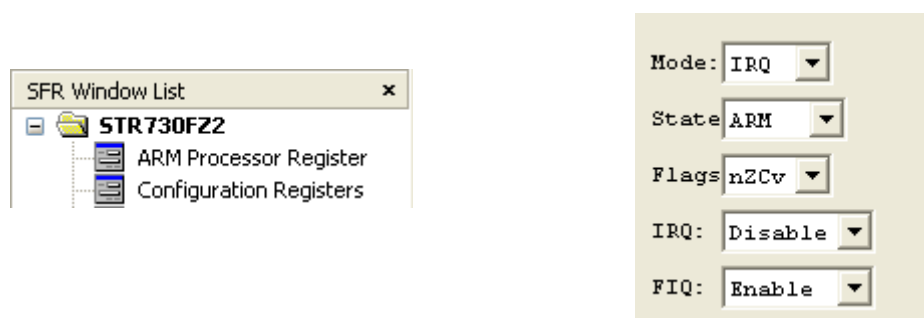
8. Switch back to the source window and locate the Watchdog_IRQ_ISR function in the code browser and set a breakpoint



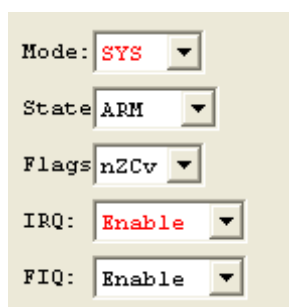
9. Run the code and the watchdog will generate an interrupt when it times out.
10. When the code halts at the entry to the ISR, switch to the disassembly window and check that the entry address matches the value stored in the vector register.

0x00000114	04C02DE5	str r12, [sp, #-4h]!
0x00000118	0DC0A0E1	mov r12, sp
0x0000011C	0CD82DE9	stmfd sp!, {r2,r3,r11,r12,r14,pc}
0x00000120	04B04CE2	sub r11, r12, #4h
#60		(void)

11. Open the View\SFR\ARM Processor registers and check that the processor has entered IRQ mode and that the IRQ interrupts are disabled.



12. Step the code so the Macro is run and check that the processor has entered system mode and that the IRQ interrupts are enabled.



13. Run the code to the exit macro and observe the switch back to IRQ mode.

Mode:	IRQ
State:	ARM
Flags:	nzCv
IRQ:	Disable
FIQ:	Enable

14. Locate the return command and read the value in the link register. Calculate the return address, then step the code to confirm that you are correct.

The return instruction from the watchdog interrupt:

0x00000150	04F05EE2	subs pc, lr, #4h
------------	----------	------------------

The current contents of the link register in this case are:

r14 00000448

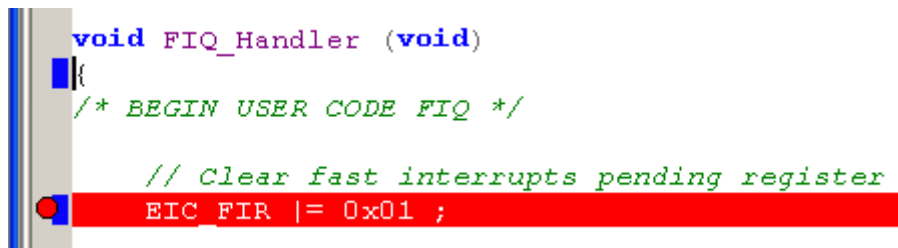
Return address is 0x00000444:

0x00000440	002083E5	str r2, [r3]
→ #288	FEFFFFFFEA	b #288
0x00000448	14010000	dw 114h

5.15 Exercise 11: FIQ Interrupt

This project configures external interrupt line 0 as a Fast interrupt. The interrupt will be generated when the button on the evaluation board is pressed.

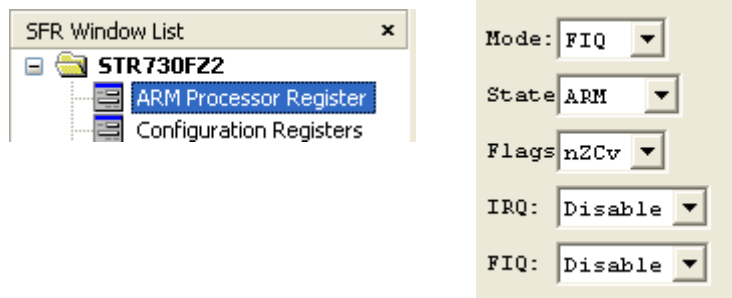
1. Open the StartEasy project in the Interrupt directory.
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. Update the project and open the new HiTOP project.
4. Set a breakpoint on the second line of the FIQ service routine in the Interrupt.c module, as shown:



```
void FIQ_Handler (void)
{
    /* BEGIN USER CODE FIQ */

    // Clear fast interrupts pending register
    EIC_FIR |= 0x01 ;
```

5. Run the code and press the “INT” button on the evaluation board. This will generate a FIQ interrupt and the debugger will stop at the breakpoint.
6. Open the view\SFR\ARM CPU registers window and confirm the CPU is in FIQ mode.



7. Remove the breakpoint, run the program at full speed and observe the value on the LED array incrementing every time you press the INT button.

5.16 Exercise 12: Memory to Memory DMA transfer

This exercise demonstrates configuration of the DMA unit to perform a memory to memory DMA transfer. The PwM example also demonstrates how to do a peripheral to peripheral transfer.

5. Open the StartEasy project in the DMA directory.
6. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
7. Update the project and open the new HiTOP project.
8. Run the code to main and view the memory at 0xA0000800 and at 0xA0000C00

Memory - Mem0		
Address	Data	ASCII
0xA0000800	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000810	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000820	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000830	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000840	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000850	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000860	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000870	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000880	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000890	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Locals	Watch1	Mem0 Watch2 Mem1

Memory - Mem1		
Address	Data	ASCII
0xA0000C00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C30	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C40	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C50	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C60	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C70	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xA0000C90	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Locals	Watch1	Mem0 Watch2 Mem1

9. Select memory window 0 (0xA0000800) right click with the mouse and select the fill option
10. Fill the memory with the string "Hello World"

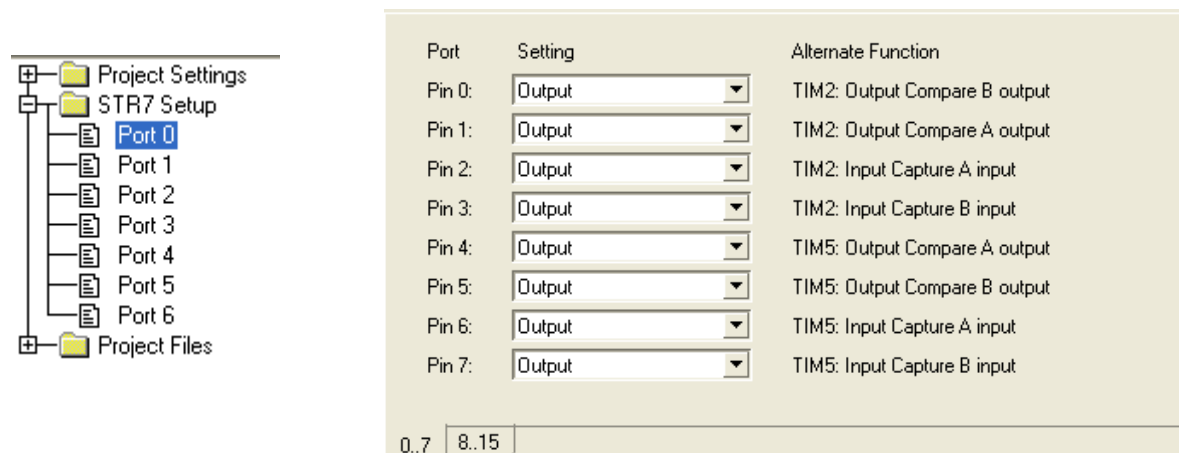


11. Run the code and the string will be copied to the 0xA0000C00 address. Halt the code and switch to the 0xA0000C00 window to see the copied data.

5.17 Exercise 13 : General Purpose IO (GPIO)

On the evaluation board the IO lines from port 0 are connected to a pair of seven segment LED displays. This example demonstrates configuring these lines to be outputs and then drives the LEDs in a count from zero to ninety-nine.

1. Open the StartEasy project in the GPIO directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. In the STR7 Setup\Port 0 menu, switch pins 0 – 9 to Output



4. Select File\Update code to rewrite the project
5. Start HiTOP and load the GPIO project
6. Examine the code that configures the GPIO registers
7. Run the code to see the displays update

5.18 Exercise 14 : Timebase timer

This example configures timebase timer 0 to generate an IRQ interrupt at its maximum period. The interrupt again updates the LED display in a counting sequence.

1. Open the StartEasy project in the GPIO directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. Open the project with HiTOP
4. Set a breakpoint on the timebase interrupt at the top of main.c
5. Run the code, when the timer reaches its end of count an interrupt will be generated
6. Examine the configuration of the timer in the sfr windows

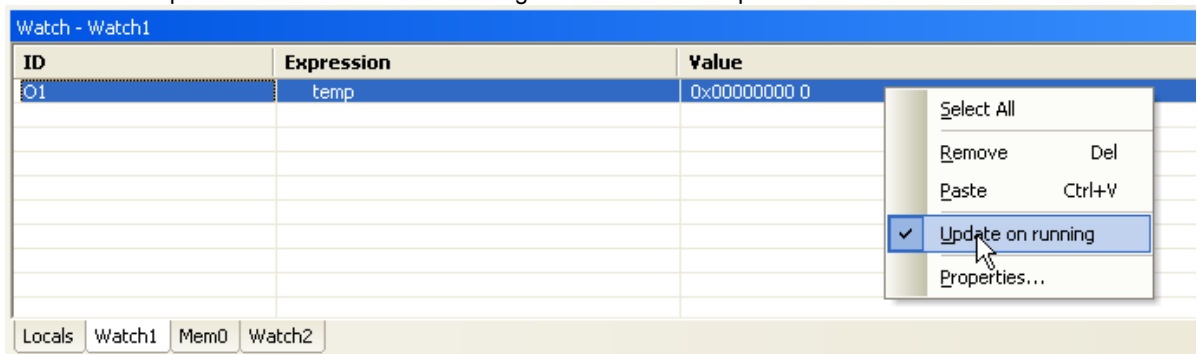
Standart Timer 0

STIM Control Register	ST0_CR	:	00000002
STIM Prescaler Register	ST0_PR	:	000000FF
STIM Preload Value Register	ST0_VR	:	0000FFFF
STIM Counter Register	ST0_CNT	:	0000B7B8
STIM Status Register	ST0_SR	:	00000001
STIM Mask Register	ST0_MR	:	00000001

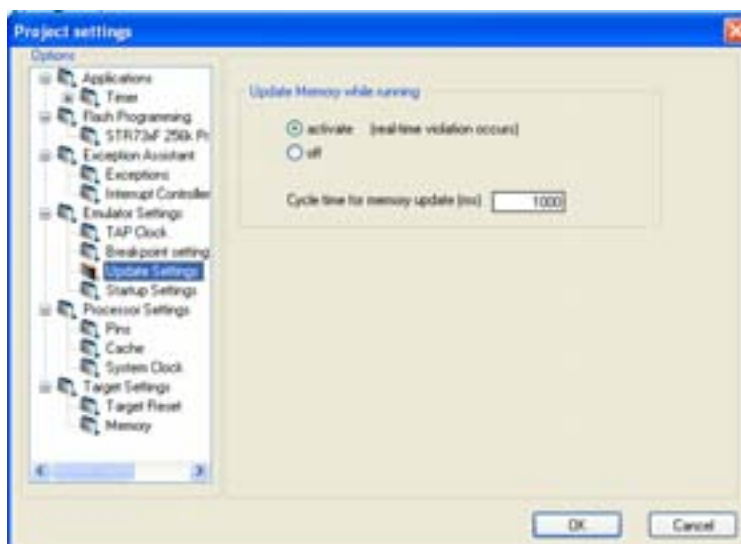
5.19 Exercise 15: Timer

In this exercise we will configure Timer 1 to generate a PWM signal that is output on GPIO 1.7. This output is fed back into the capture A pin of timer 2, which is configured in PWMI mode to measure the period and pulse width of the PWM signal. The ADC is used to read the potentiometer and DMA the result directly into the Timer 1 register. The captured pulse width value is written to the PWM1 module to sound the buzzer.

1. On the evaluation board connect the port pins as follows:
 - i. Timer 1 Output compareA
1. GPIO Pin 1.7
 - Timer 0 Input Capture A
GPIO Pin 1.11
2. Open the StartEasy project in the timer directory
3. In the project settings\debug tool menu enter the serial number on the base of your Tantino
4. Update the project and open the HiTOP project
5. Run the code to the While(1) loop and check the configuration in the view\sfr\timer1\control status and view\sfr\timer2\control status windows.
6. Run the code at full speed and observe the PWM output on Pin 1.7 with an oscilloscope. The PWM can be modulated by varying the potentiometer.
7. To see the ADC update add the "temp variable to the watchwindow
8. Select the temp variable in the watch window right click and select update



9. Make sure the project/settings/update option is active



5.20 Exercise 16: PWM Module

This exercise demonstrates setting up the PWM module and used the DMA to transfer a result from the ADC channel 0 directly to the PWM duty cycle register

1. Open the StartEasy project in the PWM directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. Make sure the x27 Jumper is fitted on the STR730 evaluation board
4. Open the PWM project with HiTOP
5. Start the project running
6. Examine the code in MAIN.C for the PWM setup and the peripheral to peripheral DMA

```

ADC_CLR4 = 0x0000000;    // Clear the powerdown bit
ADC_DMAE = 0x00008000;   // Enable the ADC DMA support
ADC_DMAR = 0x00000001;   // Enable channel 0 DMA transfer

// Setup DMA to transfer ADC result to PWM duty ratio register
ARB_PRIOR = 0x00000003;  // Enable DMA arbitration

// Set DMA source registers to point at ADC results register 0
DMA3_SOURCE0 = ((unsigned short)&ADC_D0 & 0xFFFF) ; // 0xF850; // Set the
DMA3_SOURCEH0 = (unsigned short)((((unsigned int)&ADC_D0)>>16) & 0xFFFF) ; /

DMA3_DESTL0   = ((unsigned short)&PWM1_DUT & 0xFFFF);    // Set the destinat
DMA3_DESTH0   = (unsigned short)((((unsigned int)&PWM1_DUT)>>16) & 0xFFFF);

// Setup DMA transfer
DMA3_MAX0      = 0x1;    // Set the transfer size to 1 data unit
// Start the transfer
DMA3_CTRL0     = 0x28F;  // normal mode, fixed address, peripheral is source

// Setup the ADC
ADC_CLR4 = 0x0000000;    // Clear the powerdown bit
ADC_CLR1 = 0x00000020;   // Set ADC clock 5.3MHz
ADC_CLR0 = 0x00000002;   // Start calibration

// Wait till end of ADC calibration
while (ADC_CLR0) { ; }

// Set ADC Mode
ADC_CLR2 = 0x00008000;   // Scan mode conversion ch:0 one channel
ADC_CLR0 = 0x00000001;   // Start conversion

// Setup PWM
PWM1_PER = 0x00004000;   // Set the PWM period
PWM1_DUT = 0x00000200;   // Set the PWM Duty
PWM1_PEN = 0x00000001;   // Start the count

```

5.21 Exercise 17: Analog to Digital Converter

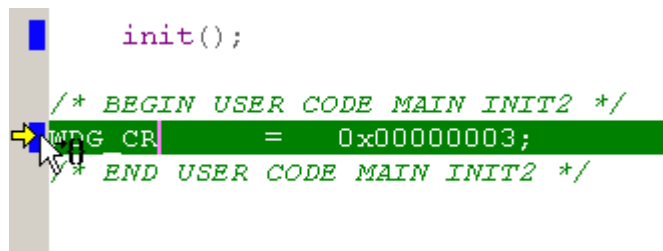
This example configures the Analog to Digital Converter (ADC) to make a scan conversion on ADC channels 0 – 2. The result from channel 0 is used to control the LEDs on the evaluation board. The analog watchdog is also enabled on channel 0 and an interrupt will be generated if the high or low threshold is crossed.

1. Check that the ADC X27 jumper is connected on the evaluation board.
2. Open the StartEasy project in the IRQ directory.
3. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
4. Update the project and open the new HiTOP project .
5. Run the code to the start of the ADC configuration code in main().
6. Open the view\sfr\ADC window .
7. Single step the code and check the configuration of the ADC.
8. Run the code at full speed and check that the ADC value is written to the LED arrays as an 8-bit hex value (10-bit ADC reading divided by 4). Notice that the value only runs between 0x0002 and 0xFC.

5.22 Exercise 18: Watchdog

This exercise configures and serves the watchdog and demonstrates the problems of trying to debug an application with the watchdog enabled.

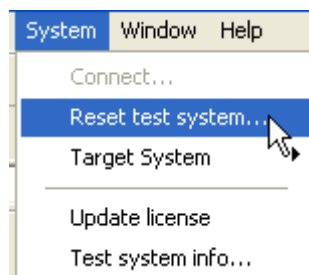
1. Open the StartEasy project in the Watchdog directory.
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. Update the project and open the HiTOP project.
4. Run the code up to the watchdog enable line:



```
init();

/* BEGIN USER CODE MAIN INIT2 */
WDG_CR = 0x00000003;
/* END USER CODE MAIN INIT2 */
```

5. Start the code running at full speed.
6. Press the INT interrupt button on the STR730 board. This will force the CPU to execute a tight loop without serving the watchdog.
7. Try to halt the program execution. The execution will seem to have stopped but any attempt to restart the program will result in an error.
8. When the watchdog times out, it resets the processor and the on-chip JTAG module. This causes the debugger to lose control of the STR7.
9. To recover from such a failure, select system\reset target system.



10. In the light of the above behaviour, when debugging a real application, it is best not to enable the watchdog until the final stages of the project.

5.23 Exercise 19: UART

This example demonstrates the use of the UARTs in interrupt mode with the FIFOs enabled. The evaluation board is connected to the PC and will loopback data sent to it but the interrupt will only trigger when the receive FIFO has received 8 characters (half full).

1. Open the StartEasy project in the UART directory.
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. Update the project and open the new HiTOP project .
4. Connect UART zero (connector X43 RS232 A) to a comm. Port on your PC and start Hyperterminal, configured for 9600 baud 8 bits, no parity, one stop bit.
5. Start the code running and switch to Hyperterminal.
6. Type in some characters and they will be echoed back to you when the receive FIFO is full.
7. Examine the UART configuration code and the UART interrupt routine

5.24 Exercise 20: BSPI

This example connects the two BSPI peripherals together and configures them as master and slave. The master is then used to initiate the transfer of data between the two peripherals.

1. On the evaluation board connect the port header pins as follows:
 2. MISO Port pin P6.11 > Port pin 5.1
 3. MOSI Port pin P6.12 > Port pin 5.0
 4. SCLK Port pin P6.12 > Port pin 4.15
5. The master slave select pin must be pulled high and a GPIO pin is used to select the slave peripheral:
 6. SSL Port pin P6.14 > 10K pull up to 3.3 V
 7. SSL Port pin P4.14 > 10K pull up to P2.00
8. Open the StartEasy project in the UART directory.
9. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
10. Update the project and open the new HiTOP project .
11. Run the code so that it initialises both BSPI peripherals and examine their configuration in the SFR windows.

SFR Window - BSPI 0		
Buffered Serial Peripheral Interface 0		
BSPI Control/Status Register 1	BSP0_CSR1 :	00000003
BSPI Control/Status Register 2	BSP0_CSR2 :	00000044
BSPI Control/Status Register 3	BSP0_CSR3 :	00000000
BSPI Master Clock Divider Register	BSP0_CLK :	000000FF
BSPI Transmit Register	BSP0_TXR :	????????
BSPI Receive Register	BSP0_RXR :	0000FFFF

SFR Window - BSPI 1		
Buffered Serial Peripheral Interface 1		
BSPI Control/Status Register 1	BSP1_CSR1 :	00000005
BSPI Control/Status Register 2	BSP1_CSR2 :	00000040
BSPI Control/Status Register 3	BSP1_CSR3 :	00000000
BSPI Master Clock Divider Register	BSP1_CLK :	000000FF
BSPI Transmit Register	BSP1_TXR :	????????
BSPI Receive Register	BSP1_RXR :	0000FFFF

12. Run the code and observe the ADC data being sent over the SPI bus and then being copied to the LED array.

5.25 Exercise 21: I2C

The evaluation board is fitted with an I2C temperature sensor. This example demonstrates basic communication with this device and displays current temperature on the LED arrays.

1. Open the StartEasy project in the I2C directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. Update the project and open the new HiTOP project.
4. Run the code so that it initialises the I2C peripheral and examine its settings

SFR Window - I2C Inter-Integrated Circuit			
		I2C0	I2C1
Control Register	CR:	25	25
Status Register 1	SR1:	83	00
Status Register 2	SR2:	00	00
Own Address Register 1	OAR1:	2C	04
Own Address Register 2	OAR2:	02	80
Data Register	DR:	80	00
Clock Control Register	CCR:	00	2C
Extended Clock Control Register	ECCR:	03	03

5. Now run the code and read the temperature of the evaluation board in degrees C from the LED arrays! Please note that there is a self-heating effect in the sensor which makes it read a few degrees above ambient.

5.26 Exercise 22: CAN

This example configures the CAN peripheral in loopback mode and demonstrates the configuration of the CAN module and then sends and receives packets of CAN data. The code has been tested on a real CAN network so if you have an analyser, you may place the CAN peripheral in its operating mode and transmit data to a real network.

1. Ensure that the following jumpers are fitted to the evaluation board:

X11, X12, X13, X14, X2, X3

2. Connect a 9-way D-type cable between connectors X17 and X18
3. Run Start Easy and open the project in the basicCAN directory. Change the serial number on the base of your Tantino.
4. Update the project and open the new HiTOP project.
5. Run the code so that it initialises the CAN peripheral and examine its settings.

Bit Timing Register (CAN_BTR): <input type="text" value="49"/> TSeg1: Segment before sample Point: <input type="text" value="4"/> TSeg2: Segment after Sample Point: <input type="text" value="9"/> SJW : Synchronisation Jump Width: <input type="text" value="3"/> BRP : Baud Rate Prescaler: <input type="text" value="0B"/> ===== Test Register (CAN_TESTR): <input type="text" value="94"/> Rx: : Rx Pin is: <input type="text" value="Recessive (1)"/> Tx: : Tx Pin: <input type="text" value="contr. by CAN Core"/> LBack : Module in L. Back Mode: <input type="text" value="Yes"/> Silent: Module in Silent Mode: <input type="text" value="No"/> Basic : Module in Basic Mode: <input type="text" value="Yes"/> ===== BRP Extension Register (CAN_BRPR) BRPE: Baud Rate Presc. Extension: <input type="text" value="00"/>	Arb. Registers 1 and 2 (CAN_IF1_A1/2R) ID 28-16: <input type="text" value="0004"/> ID 15-0: <input type="text" value="0000"/> MsgVal: Message is Valid: <input type="text" value="Yes"/> Xtd : Identifier: <input type="text" value="Standard 11 Bit"/> Dir : Message Direction: <input type="text" value="Transmit"/> ===== Message Control Register (CAN_IF1MCR) NewDat: New Data stored: <input type="text" value="No"/> MsgLst: Message lost: <input type="text" value="No"/> IntPnd: Interrupt is Pending: <input type="text" value="No"/> UMask : Use acceptance Mask: <input type="text" value="No"/> TxInt : Transmit Int.: <input type="text" value="Disabled"/> RxInt : Receive Int. : <input type="text" value="Disabled"/> RmtEn : Remote Enable: <input type="text" value="No"/> TxRqst: Transmission Requested: <input type="text" value="No"/> EoB : Single Obj. in Buffer: <input type="text" value="No"/> DLC : Objekt Data Length Code: <input type="text" value="1"/> ===== Data A/B Registers (CAN_IF1_DA/BLR) Data A1: <input type="text" value="0055"/> Data A2: <input type="text" value="0000"/> Data B1: <input type="text" value="0000"/> Data B2: <input type="text" value="0000"/>
---	--

6. Send the first message and examine the IF1 (TX) and IF2 (RX) message buffers to check that the correct data has been sent and received.

7. Next load the full CAN example and examine how it uses the IF message registers to access the CAN message RAM.

Arb. Registers 1 and 2 (CAN_IF2_A1/2R)

ID 28-16: ID 15-0:

MsgVal: Message is Valid:

Xtd : Identifier:

Dir : Message Direction:

=====

Message Control Register (CAN_IF2MCR)

NewDat: New Data stored:

MsgLst: Message lost:

IntPnd: Interrupt is Pending:

UMask : Use acceptance Mask:

TxInt : Transmit Int.:

RxInt : Receive Int. :

RmtEn : Remote Enable:

TxRqst: Transmission Requested:

EoB : Single Obj. in Buffer:

DLC : Objekt Data Length Code:

=====

Data A/B Registers (CAN_IF2_DA/B1R)

Data A1: Data A2:

Data B1: Data B2:

6 Bibliography

6.1 Publications

ARM7 TDMI datasheet	ARM Ltd.	
ARM system on chip architecture	Steve Furber	
Architecture reference manual	David Seal	
ARM system developers guide	Andrew N. Sloss	
	Domonic Symes	
	Chris Wright	
 GCC the complete reference	 Arthur Griffith	
 STR7 User Manual	 ST Microelectronics	
STR7 USB Library manual	ST Microelectronics	
 Embedded Networking with CAN and CAN open	 Olaf Pfeiffer Christian Keydel Andrew Ayre	
 USB Complete	Jan Axelson	ISBN 096508195-8
Universal Serial Bus System Architecture	Don Anderson	ISBN 0-201-46137-4

6.2 Web URL

www.arm.com

www.st.com

www.hitex.co.uk

www.keil.co.uk

www.usb.org

www.lvr.com

www.thesycon.de

www.hitex.co.uk

www.hitex.co.uk/str730

www.keil.com

USB implementers forum
Website for USB complete with HID source code
commercial device driver for USB
USB and STR7 tools
Updates to this book and the example programs
STR7 tools

This Page Intentionally Blank

This Page Intentionally Blank

This book is intended as a hands-on guide for anyone planning to use the STR7 family of microcontrollers in a new design. It is laid out both as a reference book and as a tutorial. It is assumed that you have some experience in programming microcontrollers for embedded systems and are familiar with the C language. The bulk of technical information is spread over the first four chapters, which should be read in order if you are completely new to the STR7 and the ARM7 CPU.

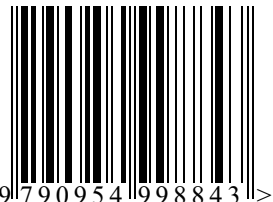
Throughout these chapters various exercises are listed. Each of these exercises is described in detail in Chapter Five, the Tutorial section. The Tutorial contains a worksheet for each exercise which steps you through an important aspect of the STR7.

All of the exercises are based on the Hitex STR73x evaluation kit which comes with an STR7 evaluation board and a JTAG debugger, as well as the GCC ARM compiler toolchain. It is hoped that by reading the book and doing the exercises you will quickly become familiar with the STR73x family of microcontrollers..

www.hitex.co.uk

hitex 
DEVELOPMENT TOOLS

ISBN 0-9549988-4-7



9 790954 998843 >

Hitex (UK) Ltd., Sir William Lyons Road, Science Park, Coventry, UK, CV4 7EZ.
Tel +44 (0) 2476 692066

