

The Insider's Guide To The ARM[®] STR71X

An Engineer's Introduction To The STR71X Series
Trevor Martin BSc. (hons) CEng. MIEE

www.hitex.co.uk



Published by Hitex (UK) Ltd.

ISBN: 0-9549988 3

First Published (preliminary edition) February 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: Alison Wenlock & 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., 09/02/2006

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	8
1.1	Outline	8
1.2	The Pipeline	8
1.3	Registers	9
1.4	Current Program Status Register.....	10
1.5	Exception Modes.....	11
1.6	ARM 7 Instruction Set	13
1.6.1	Branching	15
1.6.2	Data Processing Instructions	16
1.6.2.1	Copying Registers.....	17
1.6.2.2	Copying Multiple Registers	17
1.7	Swap Instruction.....	18
1.8	Modifying The Status Registers	18
1.9	Software Interrupt.....	19
1.10	MAC Unit.....	20
1.11	THUMB Instruction Set	21
1.12	Summary	22
2	Chapter 2: Software Development	24
2.1	Outline	24
2.2	The Development tools	25
2.2.1	StartEasy.....	25
2.2.2	HiTOP IDE	25
2.2.3	Which Compiler?.....	25
2.2.4	DA-C.....	25
2.2.5	TESSY.....	26
2.3	Tutorial	26
2.4	Startup Code	26
2.5	ARM Procedure Call Standard.....	29
2.6	Interworking ARM and THUMB.....	30
2.7	STDIO libraries.....	30
2.8	Accessing Peripherals.....	30
2.9	Interrupt Service Routines.....	31
2.10	Software Interrupt.....	32
2.11	In-Line Functions.....	33
2.11.1	Inline Assembler.....	33
2.12	Linker Script Files.....	33
2.13	C++ support	34
2.14	Hardware Debugging Tools	35
2.15	Important	36
2.16	Summary	36
3	Chapter 3: System Peripherals	38
3.1	Outline	38
3.2	Bus Structure	39
3.2.1	Memory Map	40
3.3	Boot Configuration	41
3.3.1	In-Application Programming.....	42
3.3.2	FLASH Memory Protection	44

3.3.3	Code Enable and Disable Debug Protection	44
3.4	Power, Reset and Clock Control	45
3.4.1	Boot Configuration II	45
3.4.2	STR7 Power supplies.....	46
3.4.3	Low Power Configuration	46
3.4.4	Active Power Management	47
3.4.5	Clock Module	48
3.4.6	Low Power Modes.....	50
3.4.6.1	Slow Mode	50
3.4.6.2	Wait for Interrupt	50
3.4.6.3	STOP mode	51
3.4.6.4	Standby.....	51
3.4.7	Important	51
3.5	Advanced Peripheral Bus Bridges	53
3.5.1	Software Reset.....	53
3.5.2	Error Registers	53
3.6	Interrupt Structure	53
3.6.1	General Purpose Interrupt.....	54
3.6.2	Leaving an IRQ Interrupt.....	55
3.6.3	Nested Interrupt Handling	56
3.6.4	FIQ Interrupt.....	57
3.6.5	Leaving An FIQ Interrupt.....	57
3.6.6	External Interrupts.....	58
3.6.6.1	STOP Mode	59
3.6.6.2	XTI Software Interrupt.....	59
3.7	External Memory Interface	60
3.8	Bootling From External Memory	62
3.9	Summary	62
4	Chapter 4: User Peripherals	64
4.1	Outline	64
4.2	General Purpose Peripherals.....	65
4.3	GPIO	65
4.4	Timer Counters	66
4.4.1	Input Capture	68
4.4.2	PWM Input Capture.....	68
4.4.3	Output Compare.....	69
4.4.4	PWM Output.....	69
4.4.5	One Pulse Mode	70
4.5	Real Time Clock.....	71
4.6	Analogue to Digital Converter	73
4.7	Watchdog	75
4.8	Communications Peripherals	76
4.8.1	UART.....	76
4.8.1.1	Smartcard Support.....	79
4.8.2	Buffered SPI.....	80
4.8.2.1	SPI Master Mode	82
4.8.2.2	SPI Slave Mode	83
4.8.2.3	SPI Error Interrupts	83
4.8.3	The I2C Module.....	84
4.8.3.1	I2C Addressing	86
4.8.3.2	Slave Mode	86
4.8.3.3	I2C Master Mode	87
4.8.4	The CAN Controller	89

4.8.4.1	ISO 7 Layer Model.....	89
4.8.4.2	CAN Node Design.....	90
4.8.4.3	CAN Message Objects.....	91
4.8.4.4	CAN Bus Arbitration.....	92
4.8.4.5	STR7 CAN Module	93
4.8.4.6	Deterministic CAN Protocols.....	104
4.8.5	USB 2.0 Full Speed Slave Peripheral	105
4.8.5.1	Introduction to USB.....	105
4.8.5.2	The STR71x USB Peripheral	115
4.9	Summary	121
5	Chapter 5: Tutorial Exercises	123
5.1	Introduction	123
5.2	Exercise 0: Installing the software	124
5.3	Setting up the Hardware	125
5.4	Overview	125
5.5	Exercise 1 Setting up your first project	126
5.5.1	StartEasy.....	126
5.5.2	Project structure	131
5.5.3	HiTOP Debugger.....	132
5.5.3.1	Editing your project	135
5.5.3.2	Run Control.....	136
5.5.3.3	Viewing data	140
5.5.3.4	HiTOP Project Settings	142
5.5.3.5	Advanced Breakpoints	143
5.5.3.6	Script Language.....	143
5.6	Exercise 2: Startup Code	145
5.6.1	Exercise 3: Interworking ARM & THUMB Instruction Sets	147
5.7	Exercise 4: Software Interrupt.....	150
5.8	Exercise 5: C++.....	151
5.9	Exercise 6: FLASH Programming	152
5.10	Exercise 7: Clock Configuration.....	153
5.11	Exercise 8: Low Power Modes.....	155
5.12	Exercise 9 : IRQ Interrupts.....	156
5.13	Exercise 10: FIQ Interrupt.....	159
5.14	Exercise 11: External Interrupt.....	160
5.15	Exercise 12 : GPIO	162
5.16	Exercise 13: Timer	163
5.17	Exercise 14: Real Time Clock (RTC)	164
5.18	Exercise 15: Analogue to Digital Converter	165
5.19	Exercise 16: Watchdog	166
5.20	Exercise 17: UART.....	167
5.21	Exercise 18: BSPI	168
5.22	Exercise 19: I2C.....	169
5.23	Exercise 20 CAN.....	170
5.24	Exercise 21: USB	173
6	Bibliography	175
6.1	Publications.....	175
6.2	Web URL.....	175

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 STR71x 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 STR71x family of microcontrollers.

1 Chapter 1: The ARM7 CPU Core

1.1 Outline

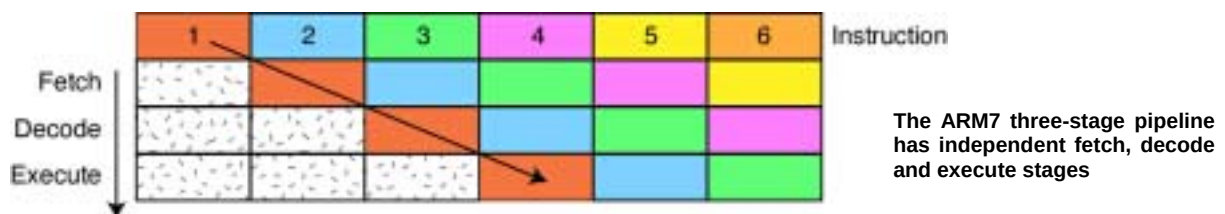
The CPU at the heart of the STR7 family is an ARM7. You do not need to be an expert in ARM7 programming to use the STR7, as many of the complexities are taken care of by the C compiler. You do need to have a basic understanding of how the CPU is working and its unique features in order to produce a reliable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

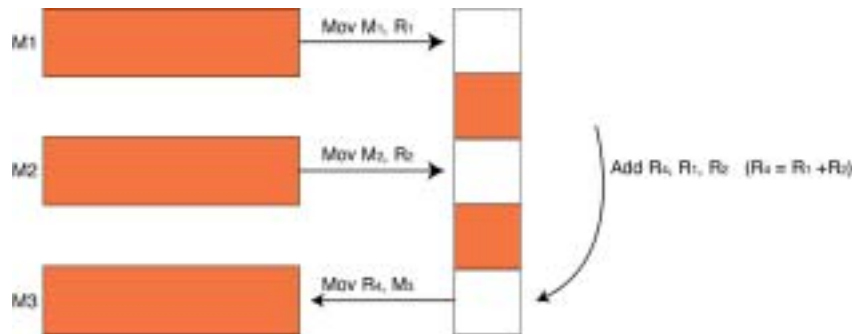
For example, the instruction:

```
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.

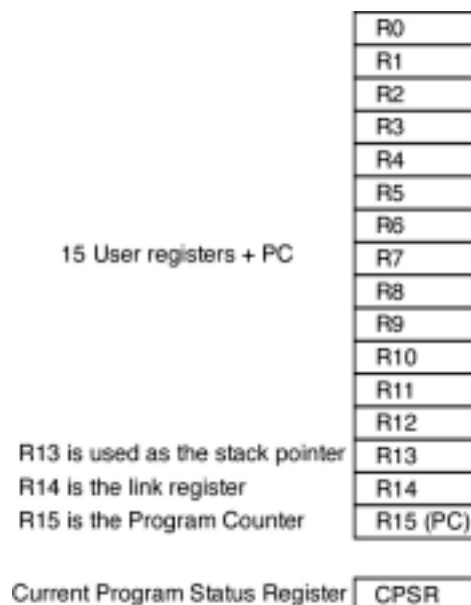
1.3 Registers

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 ARM7 CPU is a load-and-store architecture. All data processing instructions may only be carried out on a central register file

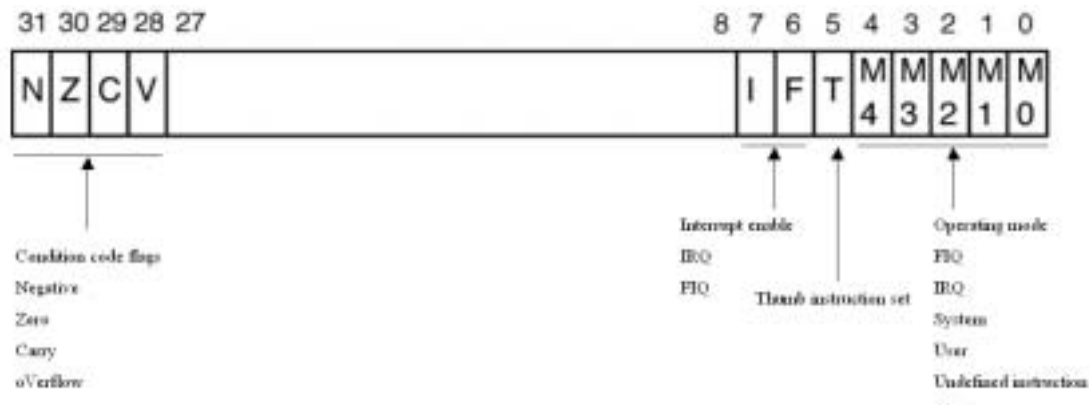
The central set of registers are a bank of 16 user registers R0 – R15. Each of these registers is 32 bits wide and R0 – R12 are user registers (in that they do not have any specific other function.) The Registers R13 – R15 do have special functions in the CPU. R13 is used as the stack pointer (SP). R14 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

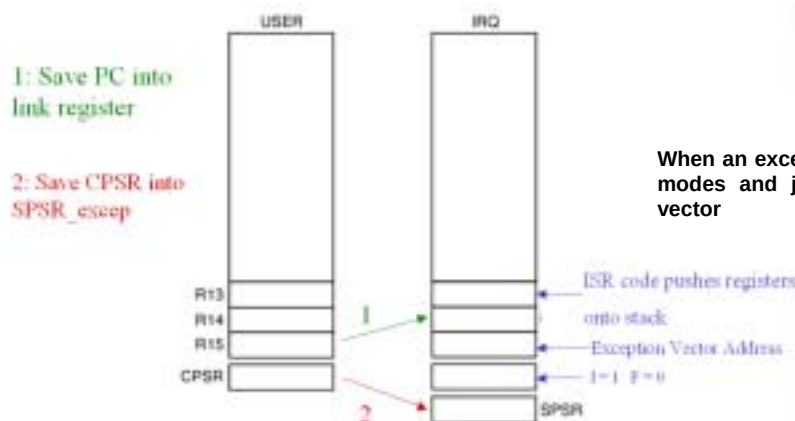
NB: There is a gap in the vector table because there is a missing vector at 0x00000014. This location was used on an earlier ARM architecture and has been preserved on ARM7 to ensure software compatibility between different ARM architectures. However in the STR7 family these four bytes are used for a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



When an exception occurs the CPU will change modes and jump to the associated interrupt vector

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.

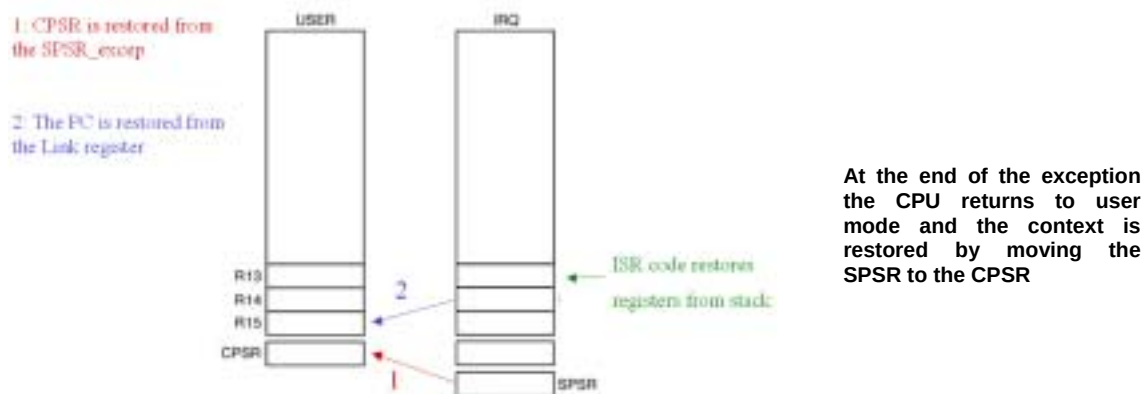
However, in the case of the FIQ and IRQ instructions, when an exception occurs the current instruction being executed is discarded and the exception is entered. When the code returns from the exception the link register contains the address of the discarded instruction plus four. In order to resume processing 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
```

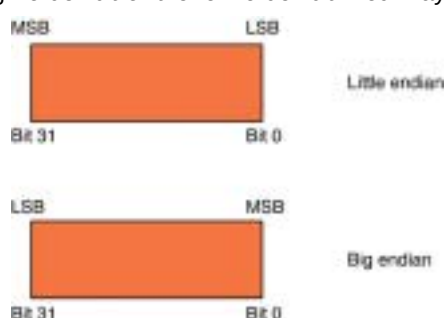
Once the return instruction has been executed, the modified contents of the link register are moved into the PC, the user mode is restored and the SPSR is restored to the CPSR. Also, in the case of the FIQ or IRQ exceptions, the relevant interrupt is enabled. This exits the privileged mode and returns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

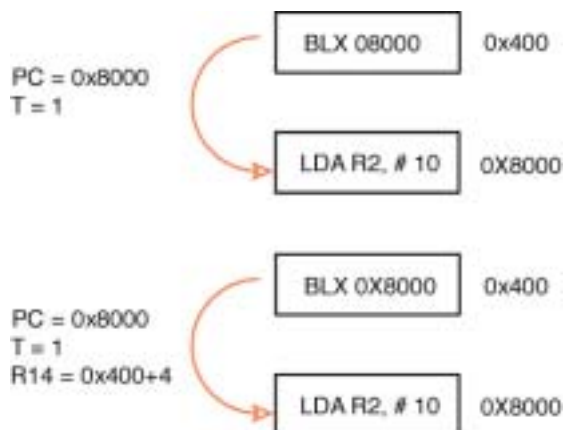
1.6.1 Branching

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.

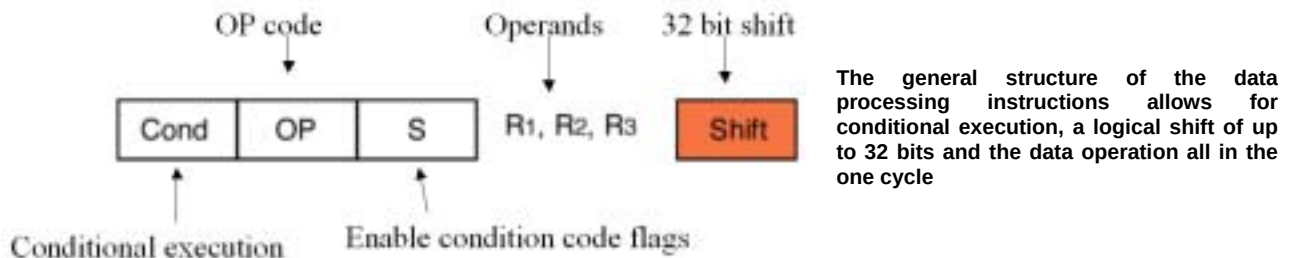


The branch exchange and branch link exchange instructions perform the same jumps as branch and branch link but also swap instruction sets from ARM to THUMB and vice versa.

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

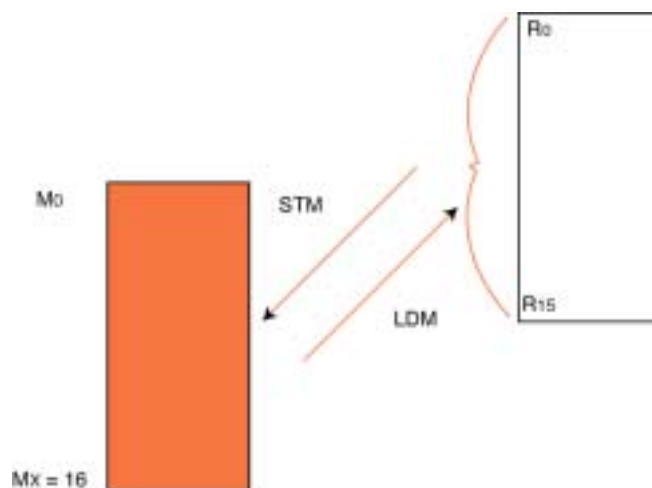
The next group of instructions are the data transfer instructions. The ARM7 CPU has load-and-store register instructions that can move signed and unsigned Word, Half Word and Byte quantities to and from a selected register.

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.

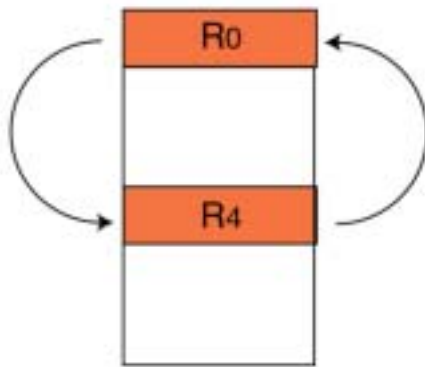


The load and store multiple instructions allow you to save or restore the entire register file or any subset of registers in the one instruction

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.

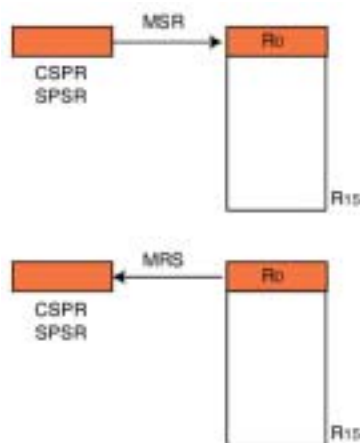
This instruction is not reachable from the C language and is supported by intrinsic functions within the compiler library.



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.

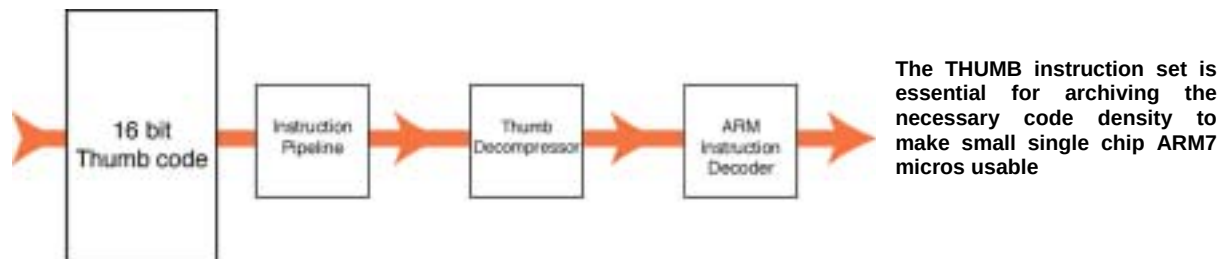
1.10 MAC Unit

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.



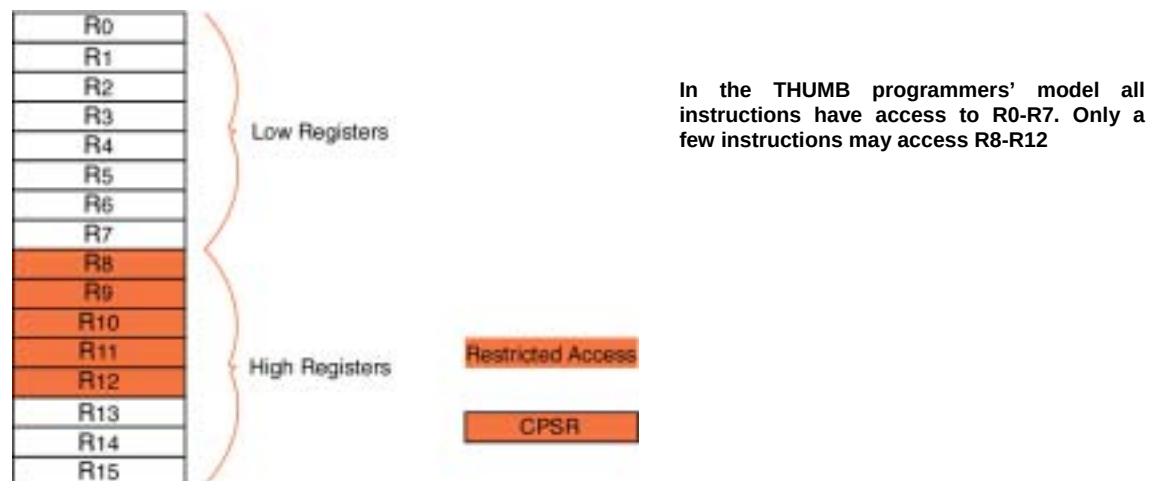
This allows instructions to be stored in a 16-bit format, expanded into ARM instructions and then executed. Although the THUMB instructions will result in lower code performance compared to ARM instructions, they will achieve a much higher code density. So, in order to build a reasonably-sized application 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”).

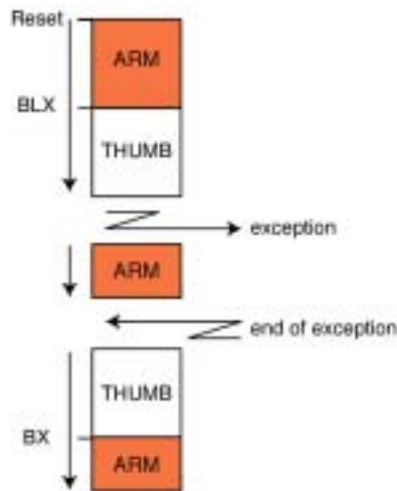


However access to R8-R12 (the “high registers”) is restricted to a few instructions:

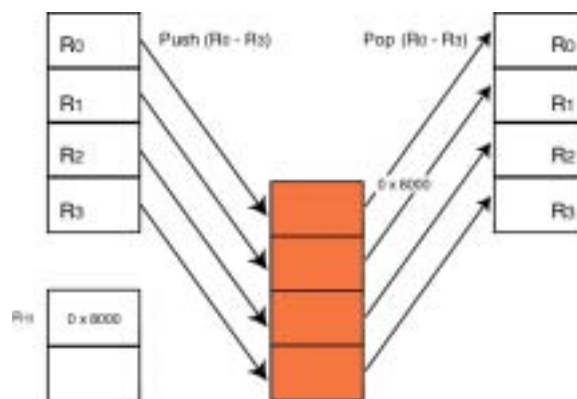
MOV, ADD, CMP

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 STR71x 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 Hiscrypt 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
                LDR      PC, [PC, #-0x808]      /* Reserved Vector */
                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, #-0x0808]
               LDR    PC,FIQ_Addr

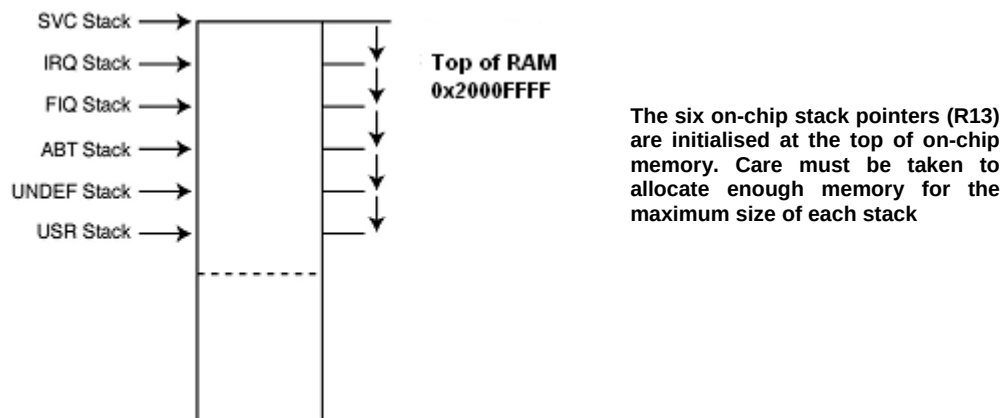
Reset_Addr:   DD      Reset_Handler
Undef_Addr:   DD      Undef_Handler
SWI_Addr:     DD      SWI_Handler
PAbt_Addr:    DD      PAbt_Handler

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.



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.

```

// Setup Stack for each mode
LDR R0, =Top_Stack
// Enter Undefined Instruction Mode and set its Stack
// Pointer
Switch mode and disable interrupts → MSR CPSR_c, #Mode_UND|I_Bit|F_Bit
Load address into the stack pointer → MOV SP, R0
Calculate start address of next stack → SUB R0, R0, #UND_Stack_Size

// Enter Abort Mode and set its Stack Pointer
MSR CPSR_c, #Mode_ABT|I_Bit|F_Bit
MOV SP, R0
SUB R0, R0, #ABT_Stack_Size

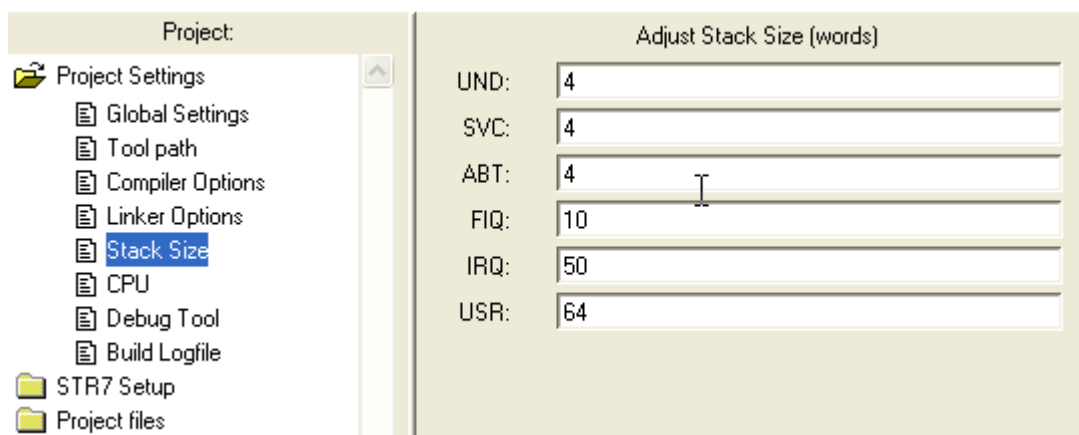
.....FIQ_IRQ and supervisor stacks

// Enter User Mode and set its Stack Pointer
Finally switch to USER mode and → MSR CPSR_c, #Mode_USR
enable interrupts → MOV SP, R0

// Enter the C code
Check for ARM or Thumb mode → LDR R0, =?C?INIT
TST R0, #1 ; Bit-0 set: INIT is Thumb
Load an Exit address into the link register → LDREQ LR, =exit?A ; ARM Mode
LDRENE LR, =exit?T ; Thumb Mode
Jump to the C init routine → BX R0

```

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.



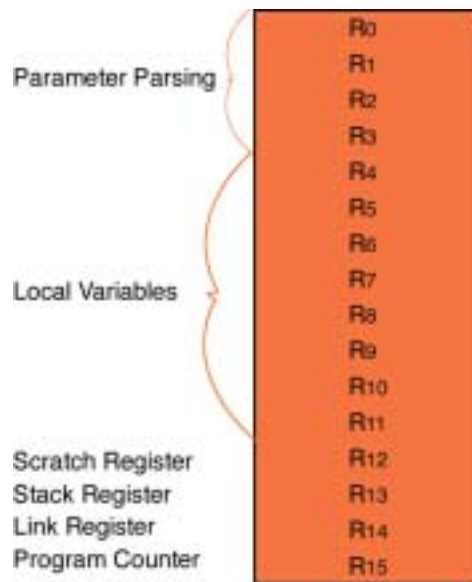
In addition, the graphical editor allows you to configure some of the STR710 system peripherals. We will see these in more detail later, but remember that they can be configured directly in the startup code.

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: 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
    EXINT      = 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, #-0x0FF0] /* 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

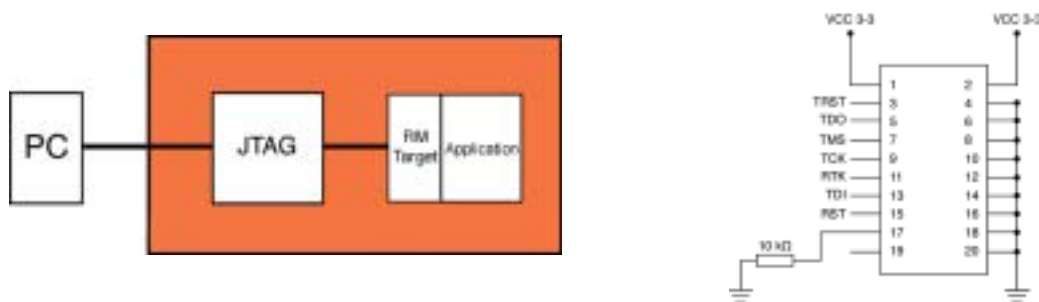
constructs. This is really outside the scope of this book, but an example is included here to get you started if you want to use the C++ language.

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

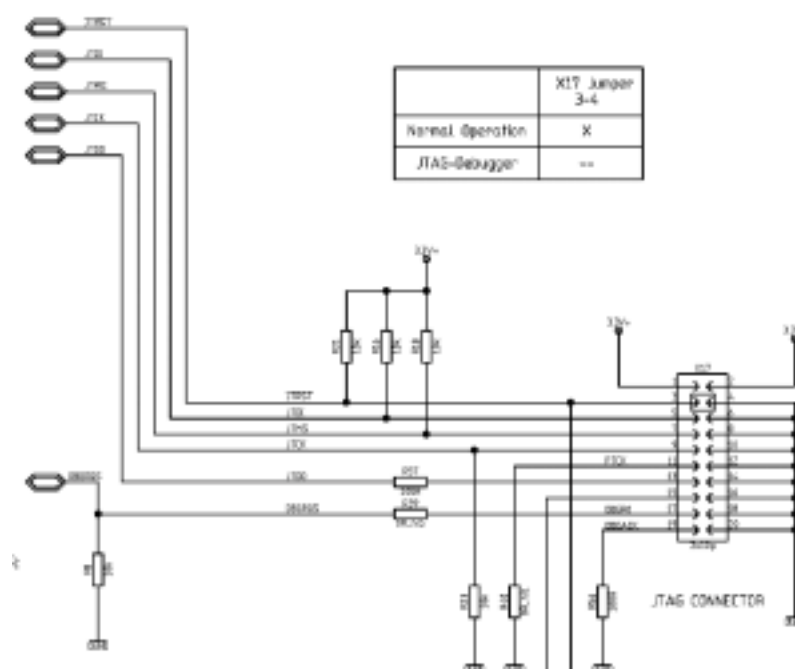
2.14 Hardware Debugging Tools

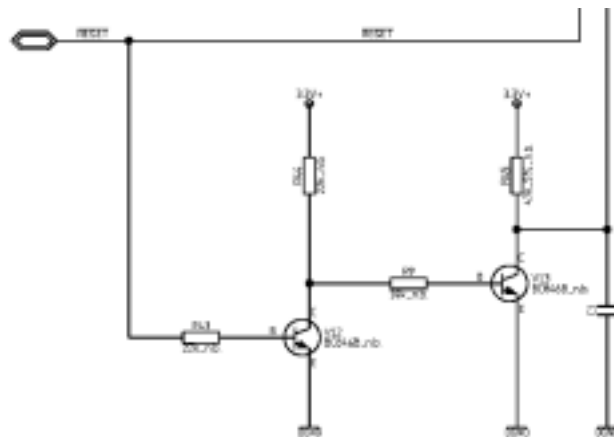
ST Microelectronics have designed the STR7 to have the maximum on-chip debug support. There are several levels of support. The simplest is a JTAG debug port. This port allows you to connect to the STR710 from the PC for a debug session. The JTAG interface allows you to have basic run control of the chip. 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 six GPIO pins as secondary functions. This means that when you are using the JTAG these pins will be unavailable for your application.

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.

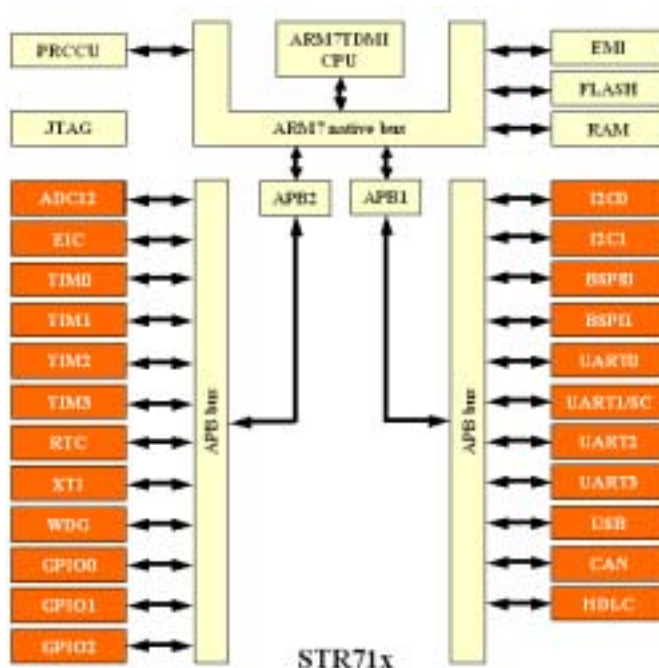
3 Chapter 3: System Peripherals

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 STR7 device itself. In this section we will concentrate on the system peripherals, that is to say the features which are used to control the performance and functional features of the device. This includes the on-chip FLASH and SRAM memory, the external bus interface which is present on the STR720 devices, the Clock and power control subsystem. Finally we will take a look at the STR7 interrupt structure and the advanced support provided by STR7 to supplement the two CPU interrupt lines IRQ and FIQ.

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. All the remaining peripherals are located on two buses called the Advanced Peripheral buses (APB). These buses are again standard bus structures defined by ARM. The APB buses are connected to the AHB bus by a bridge. This bridge contains a clock divider which allows the APB buses 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. The STR7 also has the flexibility of having two completely separate APB buses.



The STR7 has three internal buses. A high speed bus to connect the CPU to the FLASH and RAM and two peripheral buses.

All of the communications peripherals are connected to the APB1 bus and all of the remaining general purpose peripherals are connected to the APB2 bus. So it is possible to run each of the different buses at different clock speeds depending on your requirements for CPU processing power, peripheral performance and power consumption.

3.2.1 Memory Map

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

0xFFFF FFFF	
0xFFFF FC00	EIC
0xE000 0000	APB2
0xC000 0000	APB1
0xA000 0000	RCCU
0x6000 0000	EXTMEM
0x4000 0000	FLASH
0x2000 0000	RAM
0x0000 0000	Boot MEM

Despite several separate internal buses the STR7 memory map is completely linear and is defined by the 4Gbyte address space of the STR7.

The on-chip RAM is located at 0x20000000 with the on-chip FLASH at 0x40000000 upwards. For STR7 devices the external bus interface is fixed from 0x60000000. Above these is the Power and Clock control unit at 0xA0000000. The two APB buses appear at 0xC0000000 and 0xE0000000. Each peripheral on the APB bus is allocated a 4k address space, this gives plenty of room to lay out the Special Function Registers

0xFFFF FFFF	EIC	0XC001 0000	reserved
0xFFFF F800	WDG	0XC000 F000	HDLC + RAM
0xE000 E000	RTC	0XC000 E000	reserved
0xE000 D000	TIMER 3	0XC000 D000	reserved
0xE000 C000	TIMER 2	0XC000 C000	BSPi 1
0xE000 B000	TIMER 1	0XC000 B000	BSPi 0
0xE000 A000	TIMER 0	0XC000 A000	CAN
0xE000 9000	CLKOUT	0XC000 9000	USB + RAM
0xE000 8000	ADC	0XC000 8000	UART 3
0xE000 7000	reserved	0XC000 7000	UART 2
0xE000 6000	IOPORT 2	0XC000 6000	UART 1
0xE000 5000	IOPORT 1	0XC000 5000	UART 0
0xE000 4000	IOPORT 0	0XC000 4000	reserved
0xE000 3000	reserved	0XC000 3000	I2C 1
0xE000 2000	XTI	0XC000 2000	I2C 0
0xE000 1000	APB BRIDGE 2 REGS	0XC000 1000	APB BRIDGE 1 REGS
0xE000 0000		0XC000 0000	

The two peripheral buses provide 4K address spaces for the on-chip peripherals. The comms. peripherals are on APB 1 and the general purpose peripherals are on APB 2.

(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 APB1 bus, its registers are fixed at the top of memory and we will see the reason for this when we examine the STR7 interrupt structure.

3.3 Boot Configuration

In the first chapter we saw that the ARM7 CPU has its exception table fixed at 0x00000000 and 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 external memory) may be re-mapped to appear at 0x00000000. The type of memory mapped at zero is selected by three external processor pins, BOOTEN, BOOT0 and BOOT1. This allows us to boot from on-chip FLASH, external memory or during development we can debug out of RAM, saving the need to continually burn the FLASH memory. 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 state 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 STR7 development board has three 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.

On-Chip FLASH

Boot EN	Boot 0	Boot 1	Memory Mapped at 0
1	0	0	On Chip Flash
1	1	0	On Chip Ram
1	1	1	External Memory CS0

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 On-chip FLASH memory is divided into two banks. The first bank is up to 256K and is intended to hold the program code. The second bank is 16k in size and is intended to hold application data. However, there is nothing special about either bank and the data bank can hold program code and vice versa.

0x4010 0F8F 0x4010 0000	FLASH Registers	20b
	reserved	
0x400C 4000	B1F1	8K
0x400C 2000	B1F0	8K
0x400C 0000	reserved	
0x4004 0000	B0F7	64k
0x4003 0000	B0F6	64k
0x4002 4000	B0F5	64k
0x4001 0000	B0F4	32k
0x4000 8000	B0F3	8k
0x4000 6000	B0F2	8k
0x4000 4000	B0F1	8k
0x4000 2000	B0F0	8k
0x4000 0000		

The STR7 has two flash banks, The program bank is divided into 8 sectors, the small 8 K sectors are designed to hold a permanent bootloader or BIOS. The second bank is intended as a data flash to store program constants.

As we saw in Chapter 2, the STR7 can be programmed in circuit using the JTAG debug port and this method must initially be used to load the application code into the on-chip FLASH. However it is also possible to reprogram the FLASH from within the application code. This in-application programming allows you to reprogram sectors of the FLASH with new code, or to store program constants into the on-chip FLASH.

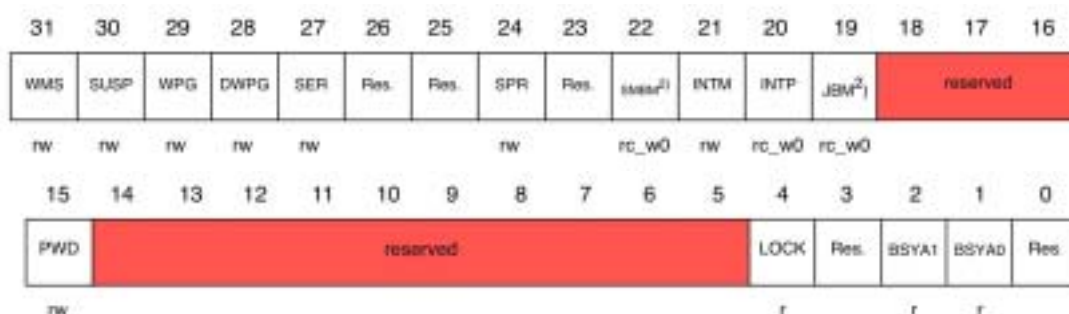
3.3.1 In-Application Programming

The on-chip FLASH is arranged as word-wide (32-bit) memory. Each sector can be erased independently and the FLASH can be written word or double-word wide. Once written, each sector can be protected against an erase and it is also possible to disable the JTAG port to prevent the internal FLASH from being read. Because the FLASH memory is arranged in two separate banks, it is possible to execute code from the program bank and write to the data bank. Programming of the FLASH memory is achieved by the FLASH Program Erase Controller (FPEC).



The FLASH program 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 an entire bank.



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.



FPEC Control register 1. This register allows you to easily perform a FLASH sector erase and either a 32 or 64 bit FLASH programming operation.



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.

To erase an entire bank you must set all the sector bits within that bank:

```
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
    {
        ;
    }
}
```

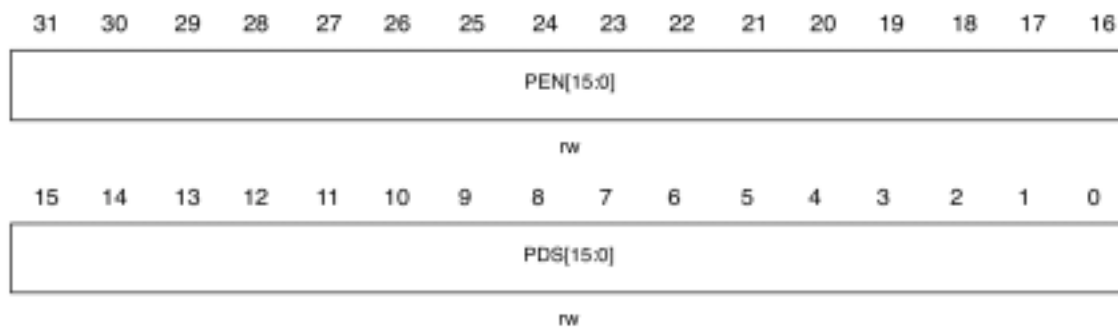
Programming a FLASH cell is a similar simple operation. First you must select whether you are going to program a word or double-word by setting the appropriate bit in the FLASH control register 0. Next write the word address of the location you want to program into the FLASH address register and the 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;
    FLASH_AR = address;
    FLASH_DR0 = data;
    FLASH_CR0 |= 0x80000000;
    while (FLASH_CR0 & 0x00000010) //wait for the lock 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 ongoing, it is not possible to access the FLASH bank which is being updated. 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 and then set the WMS bit.

3.3.2 FLASH Memory Protection

The STR7 has two memory protection methods for the FLASH banks. 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 one, you must write and erase it as if it were a standard FLASH cell with code running from the on-chip RAM.

3.3.3 Code Enable and Disable Debug Protection

Setting the DBGP bit directly will only cause a temporary un-protect. 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. 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 un-protection method is only a temporary unprotect, the sector will revert to being a protected sector as soon as the processor is reset.

Exercise 6: Flash Programming

This exercise demonstrates erasing and programming the data FLASH with constants

The on-chip FLASH memory has two operating modes which can be set in the Power and Clock control unit. The FLASH memory can be accessed in burst mode. In this mode all sequential accesses to the FLASH memory are made at a maximum of 50MHz and zero wait state. Any non-sequential access will incur an additional wait state. This mode is particularly suitable for ARM (32-bit) instructions. Since this code uses conditional execution of all instructions, the program flow is much more linear than a more traditional microcontroller instruction set and we will get a higher sustained access rate to the FLASH memory. The FLASH memory can also be placed into a low power mode. In this mode access to the FLASH memory is limited to accesses at 33 MHz, reducing the overall power consumption. In addition, if the CPU is placed in a power saving mode, the FLASH memory will place itself in a reduced power mode. To get maximum power saving, the FLASH memory can be fully powered down, but

this introduces a start-up delay when the CPU wakes up and resumes operation. We will look at these modes in more detail in the next section.

3.4 Power, Reset and Clock Control

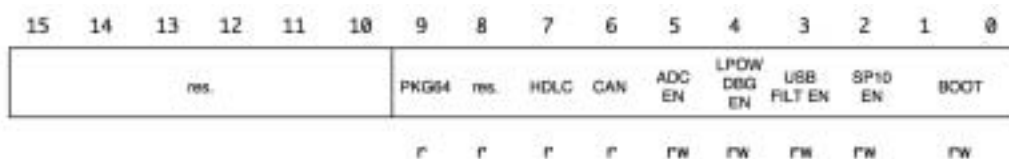
The STR7 has a sophisticated Power, Reset and Clock control unit (PRCCU). The PRCCU, in combination with other features on the STR7, allows the microcontroller to be placed in a low power consumption configuration. Power consumption can be further reduced by active power control management while the application is running.



The Power Reset and Clock control unit is initialised by the start-up code to provide the system clocks. During operation it is also used to manage the power saving modes of the STR7

3.4.1 Boot Configuration II

The Boot configuration register in the Power control unit is used to set and determine the general configuration of the STR7.



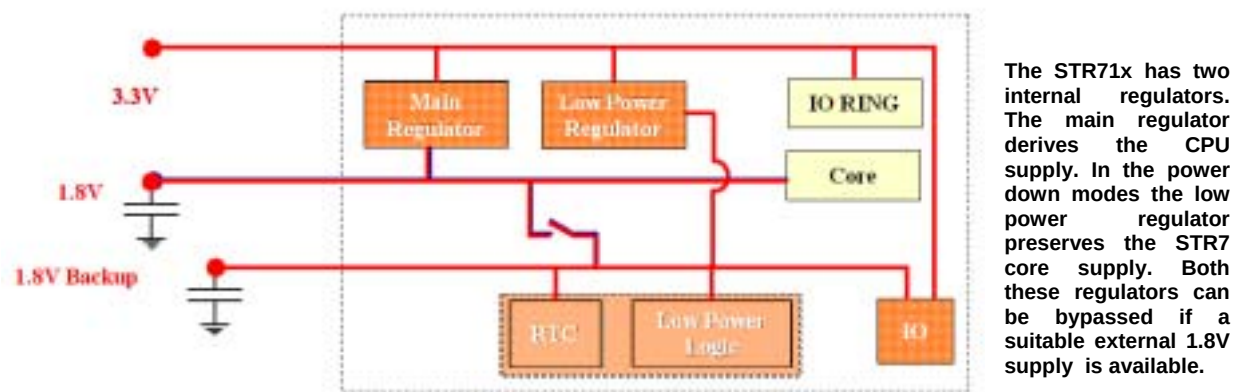
Power control unit boot configuration register. The boot configuration register allows you to determine and configure some of the basic operating features of the STR7

There are read-only bits that allow us to determine the package type, if it is 64 pin or 144 pin and if the STR7 variant has the Can and HDLC peripherals fitted. These bits allow application software to check the features of the device it is running on and can help make application code portable over several hardware platforms. Also in this register are two bits which must be set to enable the analogue to digital converter and the SPI peripheral. On reset these devices are disabled to conserve power. These control bits are easy to overlook when first starting with the STR7. If you are using the USB peripheral this register also allows you to enable a filter in the USB transceiver. (For power saving again this filter is disabled after reset.) In the boot configuration register you can also enable low power debug support. A common debug problem with ARM7 is that as soon as the CPU is placed in a power down mode, it stops responding to requests from the JTAG port, causing the debugger to lose control of the chip. In the configuration register you can enable low power debug support. This will cause the ARM7 to

come out of a power down state in response to a request from the JTAG unit. If your application is making a lot of use of the STR7 low power modes, this is an important feature that allows you to test your code without having to comment out the power saving functions. Finally, the Boot configuration register allows you to read the setting of the external boot pins and also to modify the boot configuration on the fly.

3.4.2 STR7 Power supplies

Internally, the STR7 uses two supply voltages, 3.3V for the peripherals and 1.8V for the ARM7 CPU. The STR7 runs from a single 3.3V supply and has an internal voltage regulator which is used to generate the 1.8V supply for the CPU. The diagram below shows the STR7 power supply used on the development board. The 1.8V pins must only be connected to the two external capacitors.



In Standby mode the voltage regulator is switched off and a second low power regulator is used to provide backup power for the STR7. For maximum power saving it is possible to provide an external 1.8V backup voltage and bypass the low power regulator. However, it is still necessary to keep the 3.3V pins powered as this supply is used to power the I/O block that contains the nSTBY pin which is used to bring the STR7 out of standby mode. Both of the voltage regulators contain a low voltage detector which will hold the CPU in reset if the V18 voltage falls below 1.35V.

3.4.3 Low Power Configuration



The power control register allows you to place the STR71x into a low power configuration.

In the power control register there are a number of bits that determine the overall power consumption of the STR7, plus a group of bits that help manage the running power consumption. Firstly we can configure the FLASH memory to run in low power mode as discussed in the on-chip FLASH section above. When set, the FLASH LP bit limits the CPU clock speed to 33 MHz and places the on-chip FLASH in its low power mode. You can dynamically change the FLASH power mode, but you must also manage the CPU clock frequency to ensure the low power operational frequency is not exceeded. If you require low power consumption, various features of the STR7 may be disabled in order to conserve power. If you are not using the real time clock, the Oscillator Bypass bit can be set which disables the RTC clock circuit and further reduces power consumption.

Within the Power control register it is also possible to configure the on-chip voltage regulators. If you are designing for low power equipment and you already have a 1.8V supply on your board, the on-chip voltage regulators can be disabled so you do not incur further power loss in the on-chip regulators. The Low Voltage Regulator Bypass bit (LPVRBYP) and the Main Regulator Bypass (VRBYP) bit are used to disable either or both regulators. However, if either regulator is disabled, power must be present on the relevant 1.8V supply pin or the CPU may be damaged. In order to prevent accidental writes to this register, bit 15 is a write-enable bit which must be set in the same write cycle that any of the configuration bits is modified. In the PRCCU the Peripheral enable register can be used to stop the clock to individual peripherals. This allows you to switch off any peripheral you are not using in your application. The USB and HDLC peripherals have a dedicated Phase Locked Loop (PLL2) by default this unit is active. If you are not using either the HDLC or USB peripheral or you have a dedicated 48MHz oscillator filled for the USB you should disable PLL2 in order to get a minimum power consumption configuration.

```
PCU_PWRCR      |= 0x00008000;           //Set the write enable bit
PCU_PWRCR      |= 0x00008200;           //Place the FLASH into low power mode MCLK < 33MHz
PCU_PWRCR      |= 0x00008000;           //Set the write enable bit
PCU_PWRCR      |= 0x00008080;           //Switch off the RTC clock
PCU_BOOTCR     &= 0x00000003;           //switch off ADC, SPI0 if they are not being used
RCCU_PER       = 0x00000000;           //switch off the Clocks to unused peripherals
PCU_PLL2CR     |= 0x00000007;           //switch off the HDLC PLL
```

In addition the on-chip regulators may be disabled and the 3.3V and 1.8V can be supplied from off chip regulators. **However if the regulators are disabled the power supplies must be present or the chip can be destroyed.**

```
PCU_PWRCR      |= 0x00000028;           // Bypass the on-chip regulators, external
                                           // regulated power must be present
```

3.4.4 Active Power Management

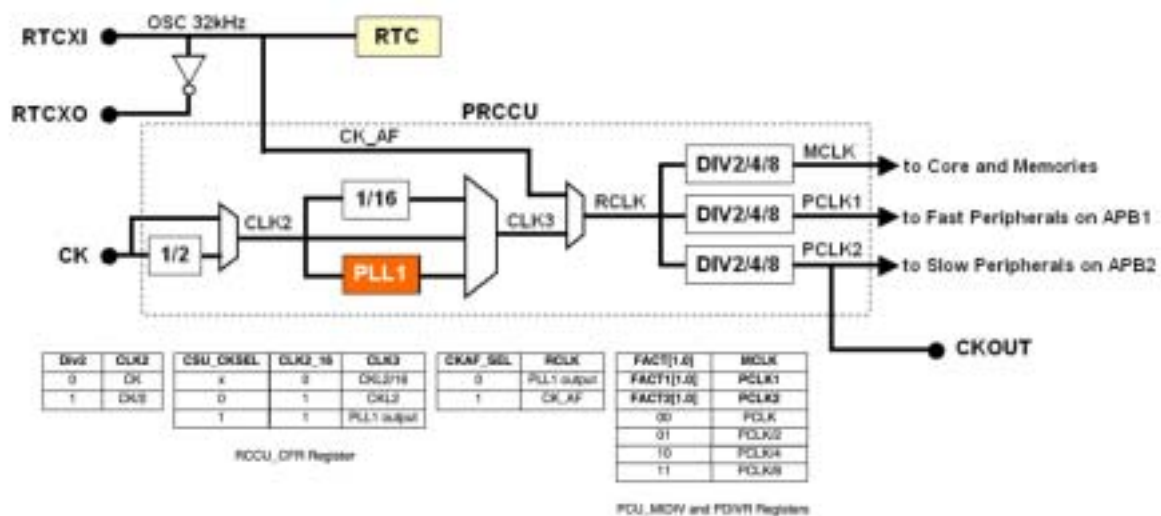
Although power savings may be made by placing the STR7 in a low power configuration real power savings are made through active power management of the STR7 low power modes and clock circuits. The STR7 has five main modes including a very low power standby mode as shown below

RUN 48 MHz	60 mA
SLOW 1 MHz	5 mA
WFI 32 KHz	500 μ A
STOP	200 μ A
STANDBY	20 μ A

The STR7 builds on the low power consumption of the ARM7 CPU to provide additional power down modes and a flexible clocking scheme

3.4.5 Clock Module

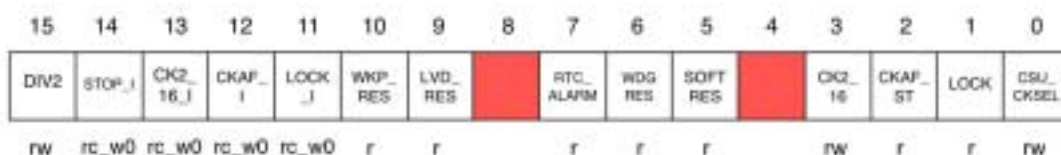
Before we look at these operating modes it is necessary to understand the clock structure of the STR7.



The Clock module allows a wide range of clock frequencies to be generated for the CPU (MCLK) and the peripheral buses APB1 and APB2. The clock signals are derived from an external oscillator or the real time clock watch crystal.

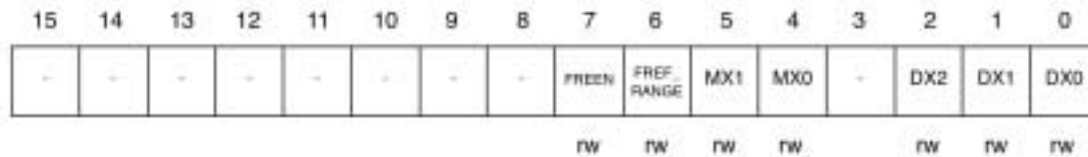
The STR7 can have three external clock crystals, a main crystal to derive the CPU and a peripheral clock. This crystal has a maximum frequency of 16MHz. A second watch crystal of 32.768KHz is used for the real time clock. The USB and HDLC peripherals also require their own external clock crystals. In the case of the USB peripheral, the crystal needs to be 48MHz. If you are not using the HDLC peripheral, the HDLC can be switched through to the USB peripheral. This allows you to generate the 48MHz clock frequency in the dedicated PLL and use a lower external clock crystal, thus reducing potential EMC problems.

The main component of the clock module is used to derive the CPU and AHB bus clock (MCLK) and the two clocks for the separate APB buses PCLK1 and PCLK2. These clocks can be run over a wide range of frequencies which are easily changed to match the running requirements of your application. During normal operation you will want to derive a MCLK frequency of 48MHz. The evaluation board is fitted with a crystal to give a CK of 16MHz and we can use the PLL1 to multiply up this external clock to the maximum operating speed of the STR7. The input to PLL1 may be CK the external clock frequency or CK/2. 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 a 50% duty cycle, so if you are using a low-cost oscillator the divider may make the clock operation more stable.



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 up to 16MHz. 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. Configuration of the PLL is made through the PLL configuration register



The PLL Configuration register sets the multiply and divide ratio 'n' 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 CLK3 stage of the STR7 clock unit, we have a choice of running at the CK frequency or CK divided by 2, 16 or 32, as well as a high frequency and low frequency output from the PLL. These clock sources are selected in the Clock Flag register. After a power on, the CK frequency is passed through to CK3. The DIV2 bit enables the divide by 2 and the CK2_16 bit enables the divide by 16. If the PLL is enabled and stable, it may be selected as the CK3 Clock source by setting the CSU_SEL bit.

The main clock for the CPU and buses is called RCLK, this clock may either be the frequency we have derived for CLK3, or for very low power operation we can select the real time clock watch crystal frequency and run the STR7 at 32.768KHz. This selection is made in the Clock control register through the Alternate Function Clock Select bit (CKAF_SEL).

The RCLK frequency is finally supplied to the CPU and peripheral buses by a final set of divider registers, which allow the clock frequency for the CPU and the two peripheral buses to be divided by 1, 2, 4 or 8. So the CPU could be running at full speed, while we clock the peripherals at a lower frequency in order to save power. However, the various clock sources are constrained by the frequency limits shown below:

Clock Signal	Description	Min	Max
CK	Clock Input	-	16 MHz
PLLREF	Clock at the PLL input	1.5 MHz	8 MHz
PLLCK	PLL output Clock	-	200 MHz
RCLK	Clock Control Unit output	-	50 MHz
MCLK	Main System Clock	-	50 MHz
PCLK	Clock for Fast Peripherals on APB1	-	33 MHz
FCLK	Clock for Fast Peripherals on APB2	-	33 MHz

So if MCLK = 48MHz,

then RCLK = MCLK x 1 = 48MHz

This allows us to run PCLKK1 and FCLK2 at 24MHz and 12MHz respectively.

The output of PLL1 = 48 MHz and on our target board the oscillator is 16 MHz, since the maximum input to the PLL is 8 MHz CLK2 = CK/2.

```
RCCU_PLL1CR    =    0x00000071;           // Enable the PLL to 40MHz
while (!(RCCU_CFR & 0x02));               // wait for it to lock
RCCU_CFR       |=    0x00000009;           // Select the PLL output as RCLK
PCU_MDIVR      =    0x00000000;           // set MCLK as RCLK/1
PCU_PDIVR      =    0x00000101           // set PCLK and FCLK to RCLK/2
```

Exercise 7: Clock Module

This exercise configures the PRCCU clock module to give a 48 MHz CPU clock and a 24 MHz clock for the peripheral buses

3.4.6 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.

3.4.6.1 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. Remember that whenever the CPU clock runs below 33MHz, we can enable the FLASH low power mode to conserve power.

```
RCCU_CFR       &= ~0x00000008;           // Select the CKCL output as RCLK
PCU_PWRCR      |= 0x00008200;           //select low power FLASH mode
```

3.4.6.2 Wait for Interrupt

In Wait For Interrupt (WFI) mode the CPU is halted, but the peripherals continue to run. WFI mode is entered by setting the WFI bit in the system mode register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	-	-	-	-	-	-	HALT	WFI

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

The clock source in this mode can be selected to be CK/16 or the RTC watch crystal. The clock selection is set by two bits in the Clock control register, the WFI_CKSEL selects either the CK/16 clock source, or the RTC crystal and the LPOWFI bit enables the use of this low speed clock during wait for interrupt mode, or continues the use of the current MCLK frequency.

When an interrupt is generated by a peripheral, the CPU resumes processing and can switch back to the PLL clock if you need maximum processing speed. In this mode it is possible to reduce power consumption to as low as 500 microAmps.

```
RCCU_CCR       |=    0x00000002;           //set WFI clock asCLK2/16
RCCU_CCR       |=    0x00000001;           //Enter low power mode during WFI
.....
RCCU_SMR       =    0x00000000;           //enter WFI mode
```

3.4.6.3 STOP mode

In Stop mode the RCLK is disabled, which halts both the CPU and the peripherals. Stop mode is entered through a specific setting of the external interrupt structure which will be discussed later (section xxx). Since all the peripheral clocks are stopped, the STR7 cannot leave STOP mode through an on-chip interrupt. The only mechanism for leaving Stop mode is a reset or an external interrupt wake up signal. Once the wake up event has been generated, the ARM7 will resume processing after 2048 clock cycles. For maximum power saving the on-chip FLASH can be placed into power down mode which further minimises the STR7 power dissipation during Stop mode.

```
XTI_MRL      = 0x00000080;          // enable the external interrupt on pin 1.13
XTI_TRL      = 0x00000000;          // trigger on falling edge
XTI_CTRL     = 0x00000001;          // enable wakeup
.....

XTI_CTRL     = 0x00000003;          // Write 0 0 to ensure the stop sequence initialises
XTI_CTRL     = 0x00000003;
XTI_CTRL     = 0x00000007;          // Enter STOP mode
XTI_CTRL     = 0x00000003;          // write 1 0 1 to stop bit
XTI_CTRL     = 0x00000007;          // in External interrupt control register
```

3.4.6.4 Standby

In Standby mode the power consumption of the STR7 is around 20 microAmps. In this mode all the system clocks are stopped, all I/O pins are forced to high impedance, the main voltage regulator is switched off and the low power regulator is used to maintain the contents of the core registers. In addition, the dedicated STDBY pin will go low while the processor is in Standby mode. You can enter Standby mode by setting the Power down bit in the power control register (remember that to change this bit you must also write to the WREN bit in the same register. Exiting the ultra low power standby mode can only be achieved by asserting a reset, by an alarm interrupt from the real time clock, or by pulling the Wake Up pin high.

```
PCU_PWRCR    |= 0x00008000;
PCU_PWRCR    |= 0x00008040;          // Enter standby mode
```

Mode		SLOW	WFI	LPWFI	STOP	STANDBY
Clocks	MCLK	Slowed down	Stopped	Stopped	Stopped	Stopped
	RCLK	Slowed down	ON	Slowed down	Stopped	Stopped
Core		Runs	Stops	Stops	Stops	OFF
IOs		ON	ON	ON	ON	Selectable
Embedded Flash		ON	ON	ON	Selectable	ON
Exit Event		-	Interrupt	Interrupt	Wake-up	RTC Alarm Wake-up Reset LVD
Power Supply	Main 3.3V	ON	ON	ON	ON	OFF
	Internal 1.8V	-	-	-	Selectable	-
	Backup 1.8V	-	-	-	-	Selectable

3.4.7 Important

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 (see section). 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.

```
PCU_BOOTCR    |= 0x00000010; // Enable low power debug
```

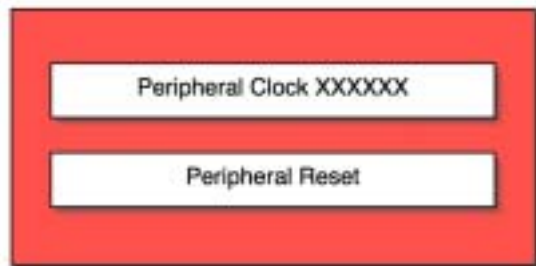
Exercise 8: Power Modes

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 .

3.5 Advanced Peripheral Bus Bridges

Both of the Advanced peripheral buses have bridges connected to the main Advanced High speed bus. Within these bridges are a group of user-controlled registers that enable error detection on the peripheral buses 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.



3.5.1 Software Reset

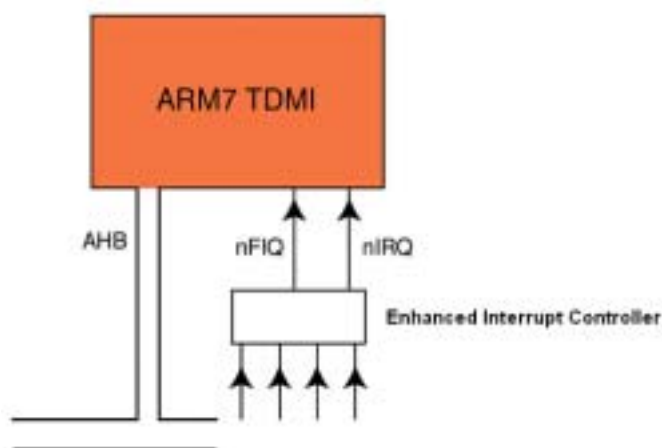
In the APB bridge the Software Reset register allows a given peripheral to be reset by a system wide reset, or by setting an appropriate bit. An individual peripheral can manually be held in reset until it is released by your application software. A complete software reset can be forced by setting the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

3.5.2 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 buses. 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.

3.6 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 that 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.



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

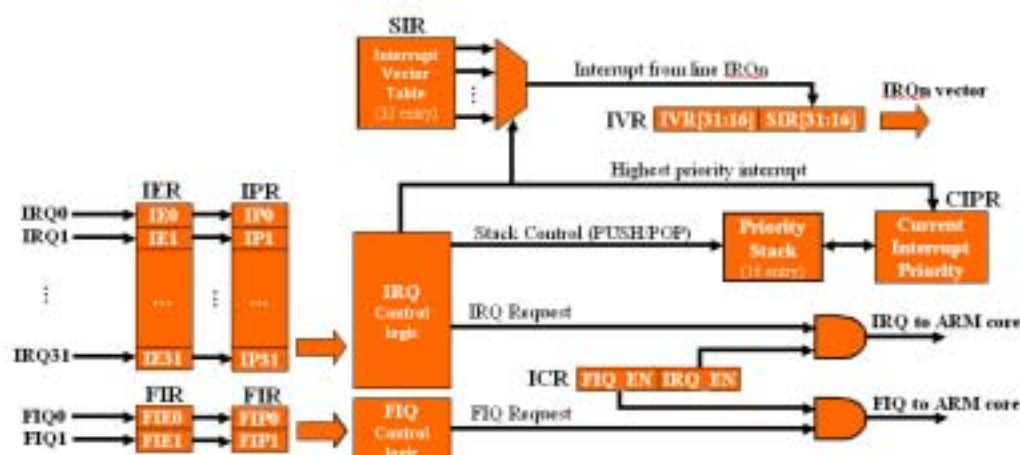
In a simple system they could be connected through a large OR gate. This would mean that when an interrupt was asserted, the CPU would have to check each peripheral in order to determine the source of the interrupt. This could take many cycles. Clearly, a more sophisticated approach is required. In order 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 can be described in three parts. Firstly 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 (XTI) that allows up to 14 GPIO lines to be configured as external interrupt lines or wake up sources. In the following sections we will have a look at each of these functions in turn.

3.6.1 General Purpose Interrupt

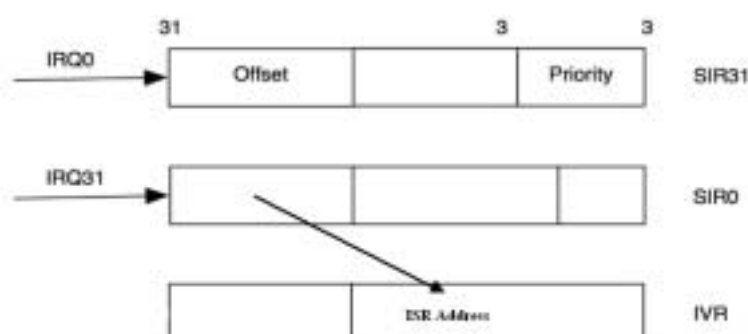
An individual interrupt source must be enabled both within the EIC by setting the appropriate bit in the Interrupt enable register and globally, enabling interrupt generation from the EIC by setting the IRQ output enable bit in the Interrupt Control register and also within the peripheral itself by setting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

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. As 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 IRQn line. 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.

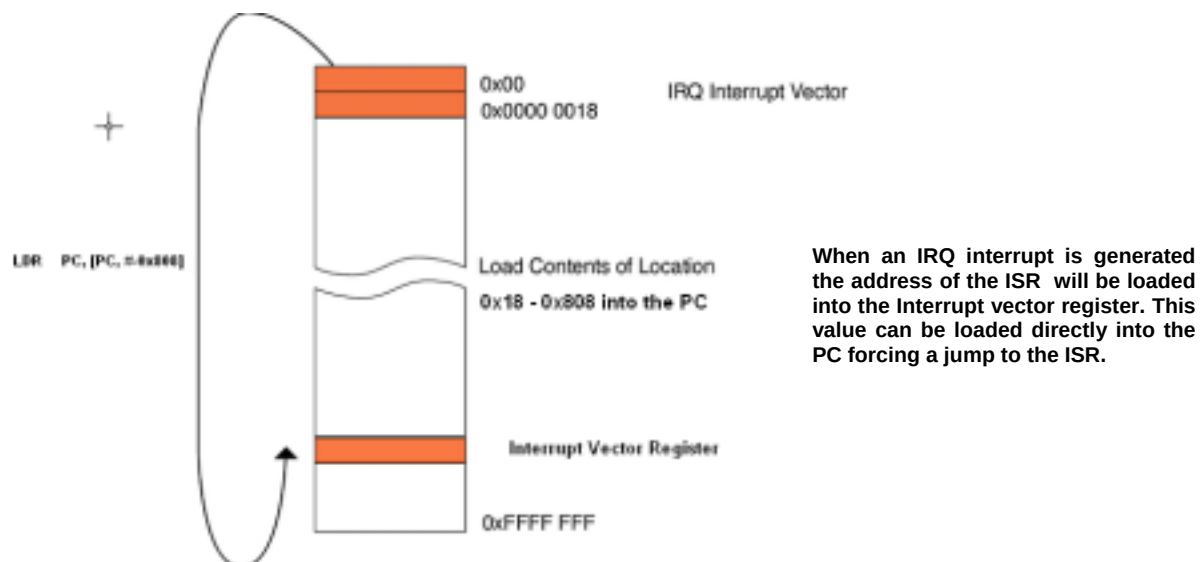


When a peripheral generates an IRQ interrupt the offset value stored in the matching Source Interrupt Registers (SIR) is transferred to the lower half word of the Interrupt vector register.

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 reference to the required interrupt service routine. This mechanism may be used in one of two ways. To allow low latency interrupt handling, ie serve the interrupt as fast as possible or to support nested interrupt ie 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. 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, #-0x808]
```

What this instruction does is to take the current value of the program counter (0x00000018) deduct 0x808 and load the contents of this memory location into the program counter.



By deducting 0x00000808 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
```

3.6.2 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 the interrupt to the EIC which then begins its end of interrupt sequence. This sequence POPs the priority stack and serves the lower priority pending interrupt.

3.6.3 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 the 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 XTI_IRQ_isr ( void ) __attribute__ ((interrupt ("IRQ")));

void XTI_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(5);      // 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
```

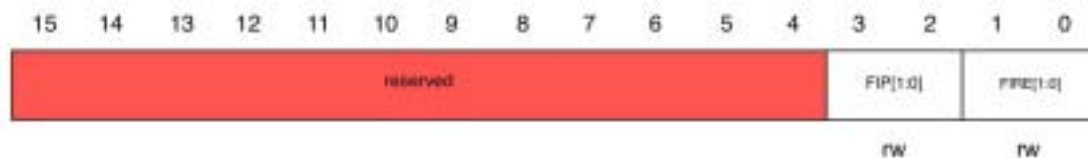
Once the user code has finished serving the interrupt, the macro restores the system link register then programs the CPSR to force the CPU back into IRQ mode with general purpose interrupts enabled. Finally the IRQ SPSR is restored. After the macro has finished, the interrupt pending bit for the current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

Exercise 9: Nested interrupts

In this exercise the EIC is configured to handle nested interrupts

3.6.4 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.

```
EIC_ICR = EIC_FIQ_ENABLE | EIC_IRQ_DISABLE;    // enable FIQ interrupts
EIC_FIR = 0x00000002;                          // enable the watchdog FIQ request
WDG_MR  = 0x00000001;                          // enable the FIQ interrupt
WDG_CR  = 0x00000002;                          // Start the watchdog as a periodic timer

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

void FIQ_Handler (void)
{
    EIC_FIR |= 0x00000008;                      // Clear the Fast Interrupt pending bit
    WDG_SR  = 0x00000000;                      // Clear the Watchdog End of count flag
}
```

3.6.5 Leaving An FIQ Interrupt

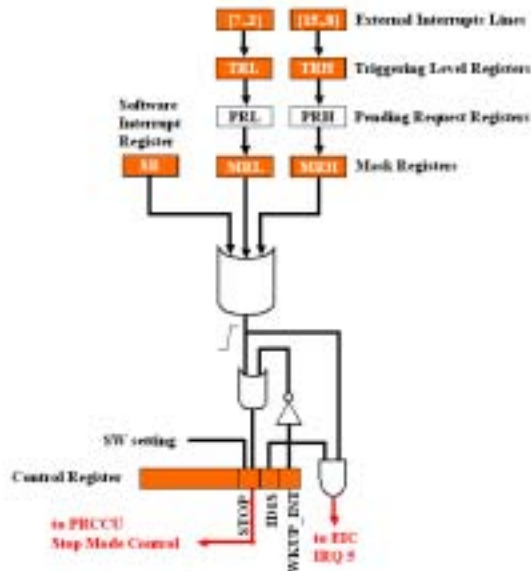
It is important to remember to clear any interrupt status flags in the timer or watchdog before quitting the interrupt. Failure to do this will cause continual interrupts to be generated.

Exercise 10: FIQ interrupt

In this exercise Watchdog is enabled as a periodic timer and generates a fast interrupt.

3.6.6 External Interrupts

The external interrupt controller is an addition to the Enhanced Interrupt controller. The XTI is connected to the main STR7 interrupt structure through the IRQ interrupt mechanism and is connected to channel five and supported by source interrupt register 5.



The External Interrupt controller provides 16 external interrupt lines. The same pins may also be used as GPIO pins or as alternate function (peripheral) pins

The XTI supports sixteen interrupt sources. Fourteen of these are external interrupt lines. The external interrupt lines may be configured to generate an interrupt or to wake the processor up from stop mode, or both. The external interrupt pins may be configured as GPIO lines or may have their secondary functions enabled. This allows the STR7 to enter into the low power STOP mode (all clocks to CPU and peripherals stopped) to wait for some activity on a peripheral such as a UART. The external interrupt line on the UART RX pin can be enabled, so when data is sent to the UART the processor will be woken up by the external interrupt in time to start receiving the serial data. This technique can be used for each of the UARTS, HDLC, SPI, I2C, and CAN peripherals. In addition, the USB peripheral has a dedicated XTI interrupt which is triggered by the end of a USB suspend condition. The XTI also has a software interrupt register that will generate an XTI interrupt when software writes to it. This is a separate mechanism to the dedicated software interrupt instruction in the ARM and THUMB instruction sets.



The External interrupt interface allows each external interrupt line to be triggered by a rising or falling edge so it can generate an interrupt and or wakeup event

To configure the XTI interrupt lines, first set the mask bits to 1 for the lines you want to use in the upper and lower mask registers (XTI_MRH and XTI_MRL). Then set the trigger polarity registers (XTI_TRH and XTI_TRL) for the

enabled interrupt lines. (This is either rising edge or falling edge.) Finally, the enabled XTI interrupt lines can be enabled as interrupt and / or wake up sources.



The wake up control register globally enables the external interrupts to generate a wake up and interrupt event. This register is also used to place the STR7 into stop mode.

The Wake Up control register contains two bits which are used to control the behaviour of the external interrupt lines. The table below shows how the enabled lines may be used to generate an interrupt or wake up condition, or both an interrupt and wake up.

ID1S	WKUP-IN	WakeUP	Interrupt
0	0	0	0
0	1	x	0
1	0	0	x
1	1	x	x

The STR7 can wake up from a stop mode and continue processing or wake up and generate an interrupt on external interrupt channel 5

```
// Load the XTI ISR address into the EIC vector register
EIC_SIR5 = ((unsigned int)XTI_IRQ_isr + 4)<<16 | 1;
EIC_IER0 = CHANNEL(5); //Enable the XTI interrupt channel
EIC_ICR = EIC_IRQ_ENABLE; //Enable the General purpose interrupts
XTI_MRL = 0x00000080; //enable the external interrupt on pin 1.13
XTI_TRL = 0x00000000; //trigger on falling edge
XTI_CTRL = 0x00000002; //enable external interrupts
```

3.6.6.1 STOP Mode

The XTI control register is also used to place the STR7 into STOP mode. In this mode the CPU and peripheral clocks are halted. This gives a power consumption figure of around 200 micro amps and, importantly, allows the STR7 to leave this low power mode and resume processing quickly. In order to enter stop mode you must write a bit sequence to the stop bit in the XTI control register. The application code must write the sequence 1,0,1 to the stop bit. The STR7 will then enter stop mode, provided that the wake up bit in the XTI control register is set. The XTI also checks that no XTI interrupts are pending and at least one external interrupt line is enabled.

3.6.6.2 XTI Software Interrupt

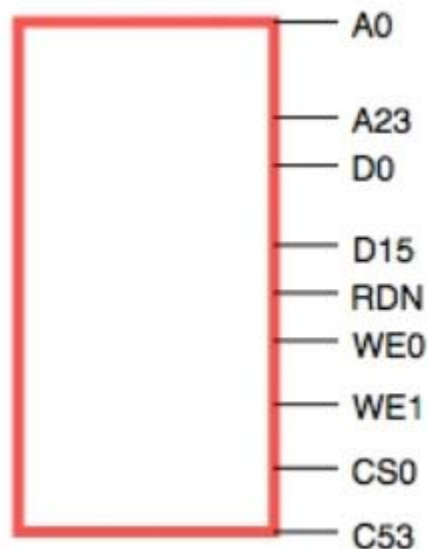
The XTI software interrupt register provides eight software interrupt flags. If the application software sets any of these flags an XTI interrupt will be generated. Again, it is important to remember to clear these flags when you enter an XTI interrupt routine generated by the software interrupt register. If you end the interrupt routine without clearing the software interrupt flags, a fresh XTI interrupt will be generated.

Exercise 11: XTI

In this exercise the External interrupt controller is configured and is used to trigger a general purpose interrupt routine.

3.7 External Memory Interface

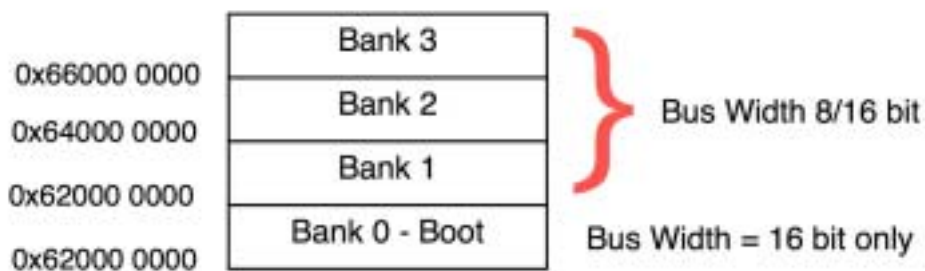
The STR710Z1T6 and STR710Z2T6 variants are bonded out in a 144 pin TQFP package. The additional pins on the package bring out the external memory interface (EMI) which allows you to interface to eight or sixteen bit peripherals or memory devices with minimal external glue logic. The external bus consists of A0 to A23 address lines and D0 to D15 for the data bus. The two write lines support addressing of the upper and lower byte depending on whether you have an eight or 16 bit address bus. The read signal RDn is an active low which should be connected to the output enable of the external components. Finally, there are four chipselects which are used to address four pages of memory which are each 16 Mbytes in size.



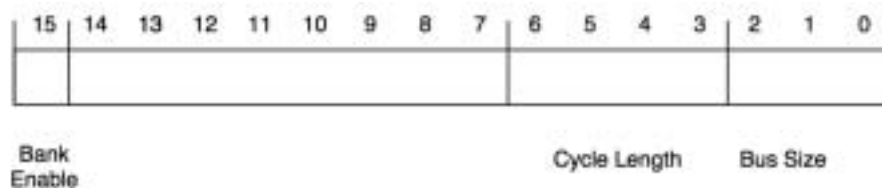
The external bus interface supports an eight or 16 bit external bus with four chipselect lines

The EBI is controlled by four chipselect registers which are used to enable the external address space and configure the chipselect. The chipselects have a fixed start address as shown below.

The characteristics of each chipselect are configured by the individual chipselect register.

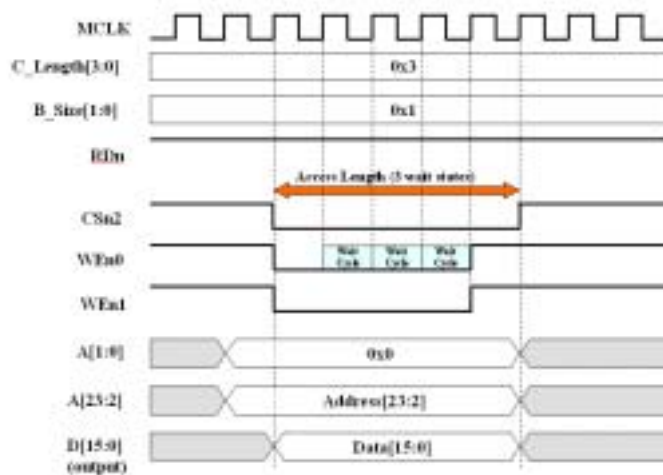


Each chipselect has an address range of 16Mbyte and with the exception of CS0 may be configured for an 8 or 16 bit external bus



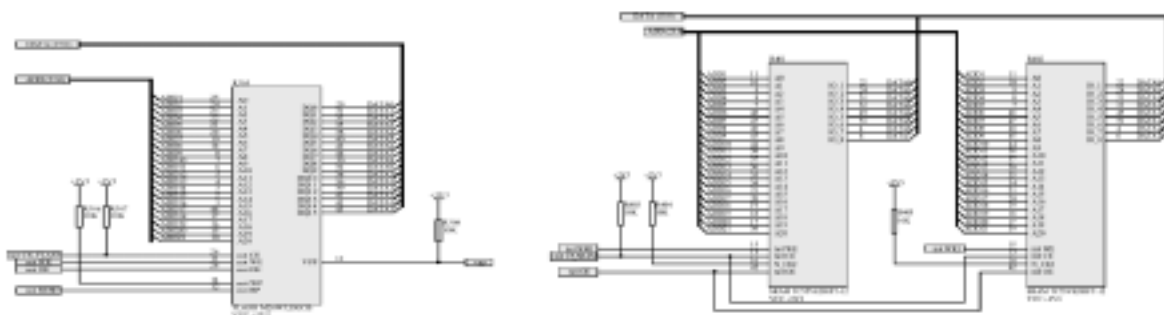
Each chipselect register configures the local bus size and waitstates

There are three fields which need to be configured within the chipselect register. You must select the external bus size to be either 8 or 16 bits and enable the chipselect to make it active. The remaining cycle length field (C_Length) controls the number of wait states which will be added to each read and write cycle. The total period of a read or write cycle will be equal to $(C_Length + 1) \times MCLK$.



The external bus waitstates are added as multiples of MCLK to the read and write cycles

Once enabled, the external buses need minimal glue logic in order to interface with external memory components and external peripherals.



These examples illustrate interfacing 8 and 16 bit devices to the STR7 bus. The full schematic is given on the accompanying CD

3.8 Booting From External Memory

On reset if the BOOTEN, BOOT1 and BOOT0 pins are pulled up to Vcc, the STR7 will activate Chipselect 0 as a default 16-bit bus and the FLASH memory located on this chipselect at address 0x80000000 will be mirrored at 0x00000000. We can build our project as normal with the code located at zero and program the external FLASH memory by the JTAG.

Since the external bus on chipselect zero is 16 bits, it really only makes sense to store THUMB code in the external memory. In fact, if you place ARM code in the external FLASH it will take two cycles to fetch one instruction and the ARM instructions will run slower than THUMB. As the ARM7 CPU will switch to executing ARM 32-bit instructions when it enters an interrupt, it is best to locate the interrupt routines in the on-chip FLASH.

3.9 Summary

This is an important chapter because you must be familiar with the system architecture of the STR7 in order to use it successfully. To get the best performance from the STR71x you need to be aware of how the ARM7 has been implemented as part of a general-purpose microcontroller. This means understanding the bus and clock system and also the interrupt system.

4 Chapter 4: User Peripherals

4.1 Outline

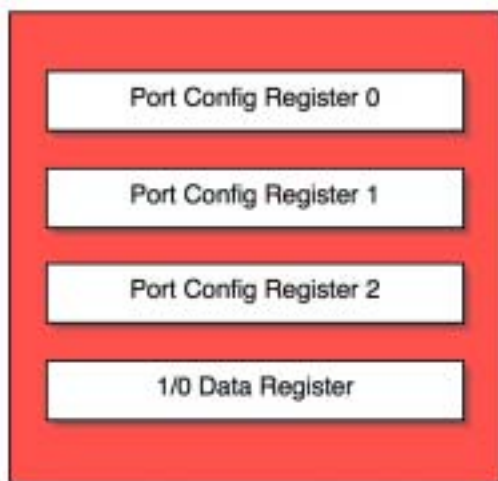
This chapter presents each of the user peripherals in turn. The examples show how to configure and operate each peripheral. By working through each peripheral in turn you will gain a good insight into the capabilities of the STR7. The example programs can be used as the basis for more complex programs 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 two separate advanced peripheral busses. The general purpose peripherals are located on APB1 and the communications-orientated peripherals are located on APB2. We will first look at the general purpose peripherals located on APB1, which are clocked by PCLK.

4.3 GPIO

The GPIO ports that are provided by ARM are fairly simple TTL-style IO ports. However ST Microelectronics has designed their own far more sophisticated port structure for the STR7. Depending on the variant of STR7 you are using, there are up to 48 GPIO lines, including high current IO lines, which can sink up to 8mA (P2.0 to P2.7). 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 14 pins to be independently configured as external interrupts or wake-up pins. This is in addition to the pin acting as a 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 GPIO interface contains three configuration registers and a single read/write IO register

The GPIO pins are controlled by three configuration registers and a data register for every 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	Input	Input / Output	Output	Output	Alternate Function	Alternate Function
Output Mode	Tristate	Tristate	Tristate	Weak Push-Pull	Open Drain	Push-Pull	Open Drain	Push-Pull
Input Mode	Analog Input	TTL Levels	CMOS Levels	CMOS Levels	-	-	CMOS Levels	CMOS 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;
    while((count= count<<1) <= 0x00010000)           //Set the LED tracer start value
    next LED                                           //Shift the set bit to enable the
    {
        for(delay = 0; delay<=0x0000f000; delay++)
        {
            ;
        }

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

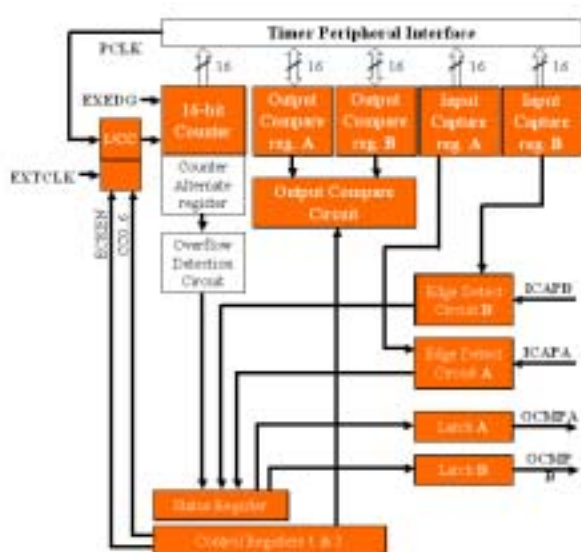
```

Exercise 12: GPIO

This exercise demonstrates programming the GPIO pins

4.4 Timer Counters

The STR7 has four timer counter blocks. These timers consist of a free-running 16-bit counter with prescaler and a series of capture and compare registers.



The STR710 has three independent 16 bit timer counters with dedicated capture and compare registers

The timer clock may be selected from either the PCLK bus clock or from an external clock source. By linking this pin to the output of a second timer block, it is possible to cascade the timers to build more complex timer arrays. Each timer counter is controlled by eight registers, which allows 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, that is managed by two control registers:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	PWMI	reserved	POLVB	POLVA	OLVLB	OLVLA	OCBE	OCAE	OPM	PWM	IEDGB	IEDGA	EXEDG	ECKEN	
r/w	r/w	-	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

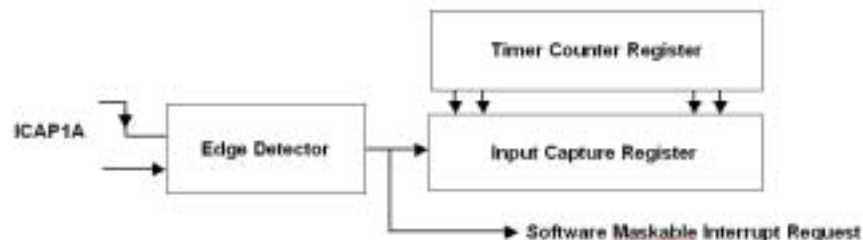
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ICAIE	OCAIE	TOE	ICBIE	OCBIE	reserved		CC7	CC6	CC5	CC4	CC3	CC2	CC1	CC0	
r/w	r/w	r/w	r/w	r/w	-		r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

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 APB2 clock. The timer prescaler value is held in the lower eight bits of control register 2. This prescaler is only applied to the APB2 clock, an external clock source is fed directly to the timer. If the external clock is used an additional bit EXEDG in Control register 1 allows you to determine if the rising or falling edge will increment the counter.

4.4.1 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.



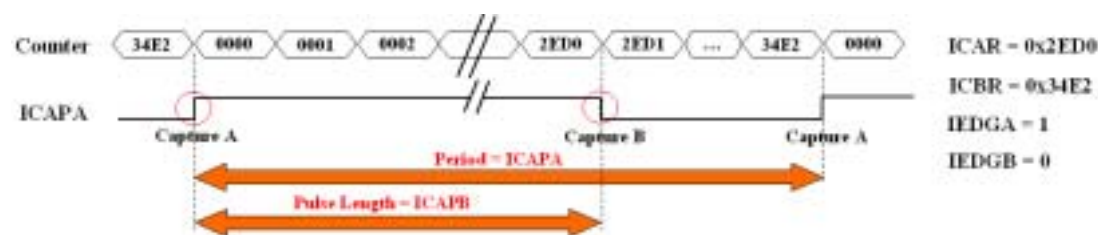
Each timer counter has two capture registers which may be used for custom edge measurement

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
```

4.4.2 PWM Input Capture

Additionally there is a PWD capture mode that allows the measurement of the pulse width and 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.



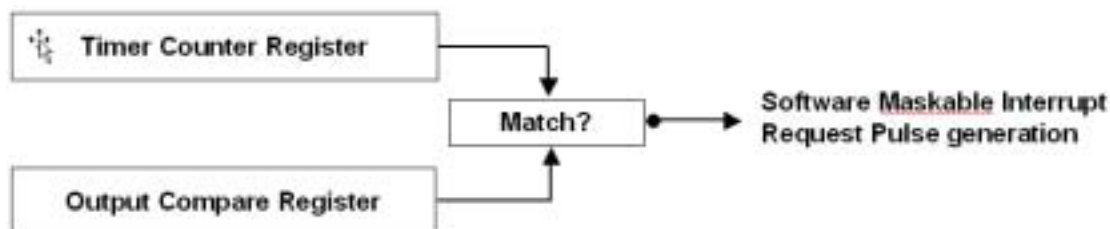
The PWM capture mode is synchronised on a pulse edge to reset the timer (Capture A). The next pulse edge (Capture A) will be captured into capture register B, this is the duty cycle. The next capture A event restarts the cycle and the capture A register contains the PWM period

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 the capture register A contains the signal period. By enabling the capture A interrupt, we can read both these values 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
```

4.4.3 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

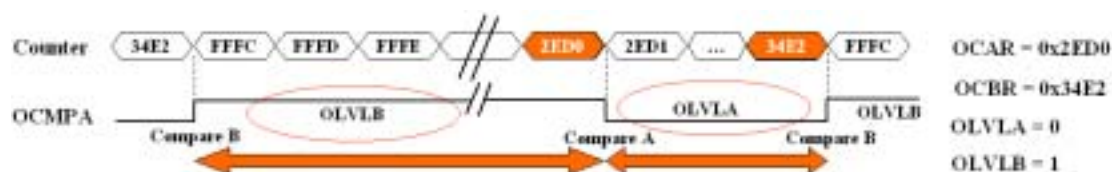
Now whenever the free running timer matches the contents of either compare register, an interrupt will be generated. If you want to generate a waveform in addition to switching the GPIO pins to their secondary function, you must enable the output compare pins by setting the OCAE and OCBE bits in control 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 oputput level on OCPMA pin
  
```

4.4.4 PWM Output

Like the PWM input mode, the STR7 timers have a special mode to allow the 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 0xFFFC. 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 0xFFFC to restart the

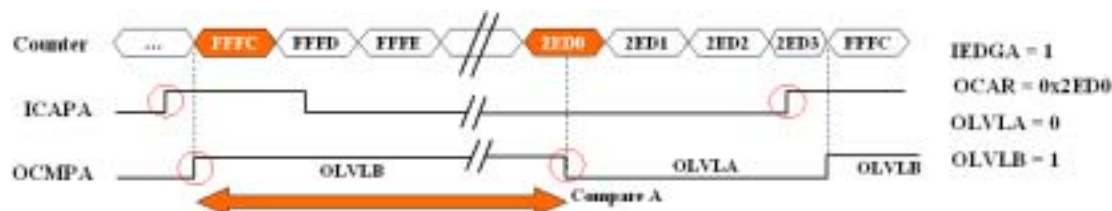
At the start of a cycle the timer is reset to 0xFFFFC and the OCA pin is set high. The value loaded into the compare A register is the period of the pulse. When this match is made the compare event loads a logic zero stored in OLVA onto the output compare pin. The PWM period is stored in the compare 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 = 0x0000008FF; // enable the capture B interrupt
// and set the prescaler
TIM1_CR1 = 0x000008150; // Timer enable,

// set output levels on compare A,
// enable compare A and PWM mode
```

4.4.5 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 0xFFFFC 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 OLVLA bit is loaded onto output compare pin A and the timer then waits for the next capture event.

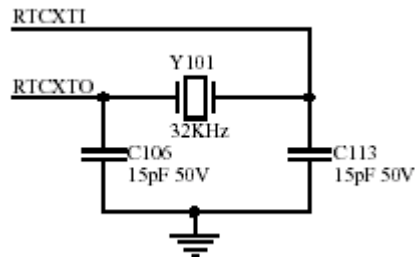
```
TIM1_OCAR = 0x0000FF00 //Set the pulse length
TIM1_CR2 = 0x0000FF; //set the prescaler
TIM1_CR1 = 0x0008265; //Timer enable, set the pulse output level on OCPMA pin
//enable one pulse mode, set capture edge
```

Exercise 13: 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.

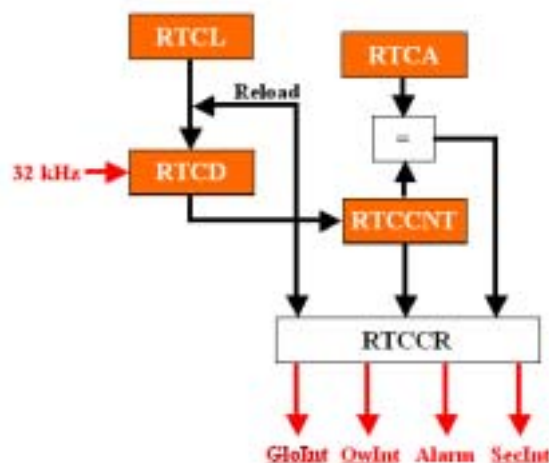
4.5 Real Time Clock

The Real Time Clock (RTC) peripheral in the STR7 provides a 32-bit counter that is incremented at one second intervals. A matching 32-bit alarm register acts as a comparator to the main counter register and can generate an alarm interrupt when a match is made. The Real Time Clock is located on APB2 but derives its clock source from an external crystal, which is generally a 32.768KHz watch crystal.



The real time clock derives its clock source from an external 32KHz watch crystal. The watch crystal can also be used to clock the CPU in low power modes.

The Real Time Clock peripheral consists of two main blocks. 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 is accessed as a peripheral on the APB2 bus. The programmer's interface consists of 10 registers, as shown below.



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 via 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 this 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 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. This gives a nominal one second interrupt. An interrupt can also 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));

```

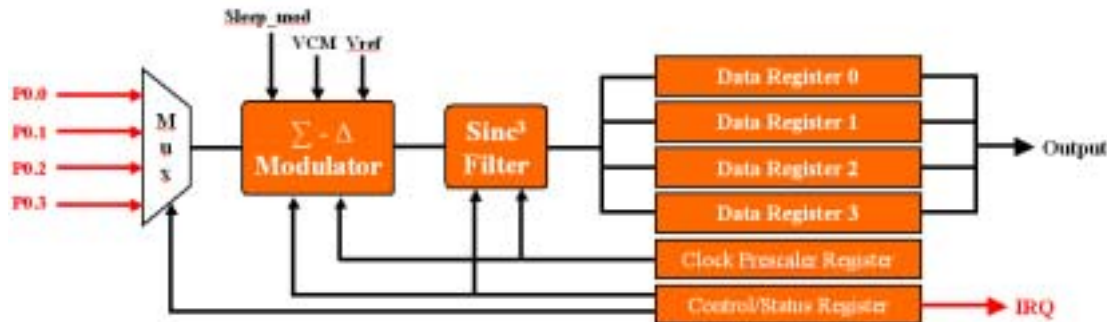
The Real Time Clock will remain active during the power down modes but care should be taken when reading its registers immediately after coming out of a power down mode. Because the real time clock has its own oscillator, the registers on the APB2 bus will not be updated for one RTC clock tick after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

Exercise 14: Real Time Clock

This exercise configures the Real time clock to work with an external 32 KHz oscillator. The second and alarm interrupts are enabled and served by a simple interrupt

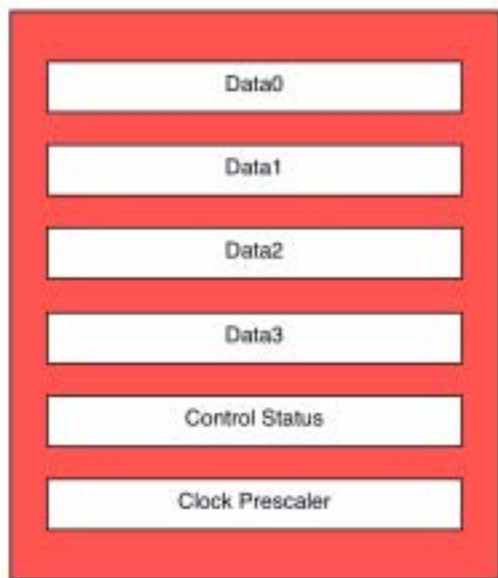
4.6 Analogue to Digital Converter

The STR7 has an on board analogue to digital converter with four multiplexed inputs. Although the GPIO pin are 5 volt tolerant the input range for the A/D converted is 0 – 2.5 V and care should be taken not to exceed this maximum value or you may damage the converter.



The STR7 has a four channel Analogue to Digital Converter with a 12-bit resolution

The A/D conversion technique used is a sigma-delta method that oversamples the analogue signal to produce a digital value with a 12-bit resolution. The sigma-delta technique gives an accurate reading with a good signal to noise ratio. However the downside is that the conversion period is 2mS and 500Hz, so the converter is really only useful for slowly changing signals. If your application can live with this, it will give you good results otherwise you will need to use an external converter.



The ADC has a results register for each channel and may be used in single channel conversion mode or round robin conversion of all four channels

After a hardware reset the A/D converter is always disabled, so before you start to use the converter, it must be enabled by writing to the ADC_EN bit in the BOOTCR register, otherwise nothing will ever happen. Like the other general-purpose peripherals, the A/D converter derives its clock from the APB2 bus. The internal clock used in the converter must be a maximum of 1.0 MHz. The APB2 clock is divided down by the value written to the clock prescaler register, defined by the formula:

$$\text{Prescaler} = \text{APB2_freq} / (\text{sample_freq} * 512 * 4)$$

The A/D converter may be used in two different modes. In the first mode the A/D converter can be used in a round robin fashion where each channel is sampled one after the other. At the end of each conversion, an interrupt can be generated. The second mode allows for a conversion on a single channel selected by the

application software. Again on completion of conversion, an interrupt may be generated. The operating mode is selected by setting or clearing the AXT bit in the status register.



The Control/Status register sets the conversion mode and enables the interrupt channel and contains conversion complete and overrun flags.

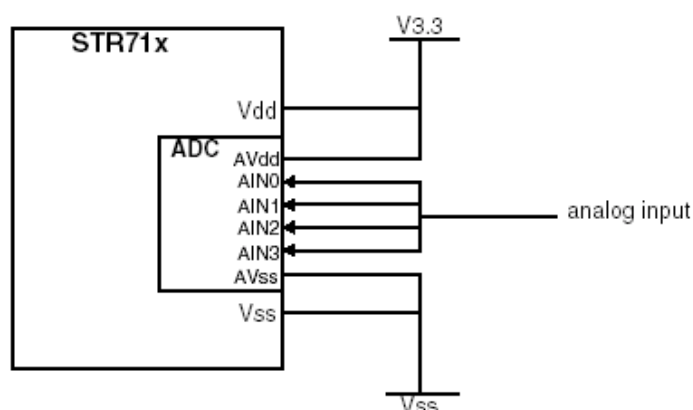
On completion of conversion, the data available flags (DA0-3) will be set and data can be read from the matching data results register. When the data is read the channel data available flag is automatically cleared. Each channel can be independently enabled to generate an interrupt when a new conversion value is available.

```
PCU_BOOTCR = 0x00000020;           // Switch on the A/D Convertor!
ADC_CPR     = 0x00000005;           // Set the A/D prescaler to divide by 10
                                           // for 10MHz Mclk to give 1MHz ADC clock
ADC_CSR     = 0x00000F00;           // Enter round robin mode and interrupt on each
// conversion
```

In single mode operation the AXT bit is set to one and the channel to be sampled is selected via the two channel address bits, (A0:1).

```
ADC_CSR     = 0x00000140;           // Enter single channel mode and enable channel
// zero and its interrupt
```

However the A/D converter will make four conversions before a valid result is available so the rate of conversion for a given channel is the same in round robin or single channel mode. To get the maximum rate of conversion for a single analogue source, it should be connected to all four channels and the A/D converter should be run in round robin mode. This way, a sample of the analogue voltage will be produced for every conversion made by the A/D converter.



For faster conversion a single analogue source may be connected to each ADC channel with round robin conversion enabled

Exercise 15 : Analogue to Digital Converter

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

4.7 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 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:

$$\text{Timeout in micro secs} = (\text{Prescaler} + 1) \times (\text{reloadvalue} + 1) \times T_{\text{pclk2}}/1000$$

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 values 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 in watchdog status register contains an end-of-count flag, which 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;        //Pat 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 if 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.

```

EIC_FIR= 0x00000002; // enable the watchdog FIQ request
WDG_MR = 0x00000001; // enable the FIQ 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 16: Watchdog

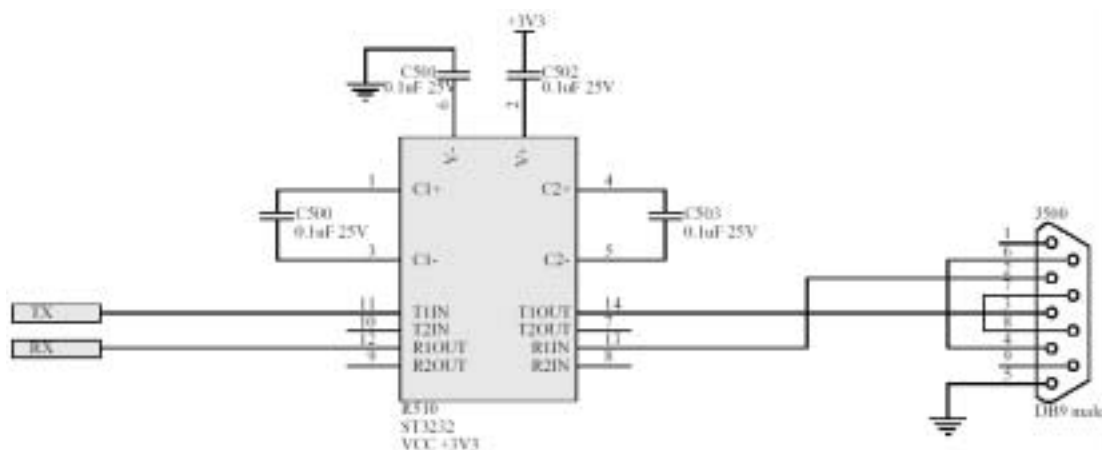
This exercise configures the watchdog as a periodic timer and enables the end of count interrupt. Each interrupt is used to toggle the bank of LED's on the evaluation board

4.8 Communications Peripherals

All the remaining peripherals are the serial communications peripherals. These are all located on the APB1 bus and with the exception of the USB and HDLC controllers, they are all driven by the APB1 bus clock, called Fclk.

4.8.1 UART

Most small microcontrollers have at least one UART and some have two. However the STR7 is almost unique in having three 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 or USB peripheral to make the STR7 the ideal hardware for a protocol converter or RS232 concentrator.



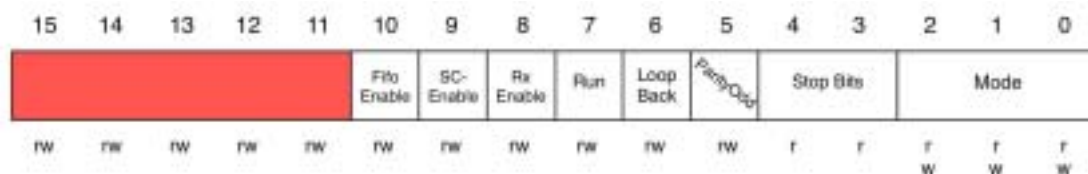
On the evaluation board, each UART is brought out to a 9 way D type, with RS232 level driver

The three 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 USRT FIFO and receive functions as well as selecting the loopback test mode and smartcard mode

After reset, the UART is disabled because with the RUN bit 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 FIFO's. 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.

$$\text{Reload value} = \text{PCLK1} / (16 \times \text{BAUDRATE})$$

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:

$$\text{Timeout} = \text{PCLK1}/\text{BAUDRATE}$$

4.8.1.1 Smartcard Support

The UART also has a special mode designed to allow the STR7 to interface with smartcard equipment. This mode can be enabled by setting the SC-Enable bit in the control register. Unless you are developing smartcard interface equipment, it is recommended that you leave this bit set to zero. If smartcard mode is enabled the standard behavior of the UART will be modified as follows:

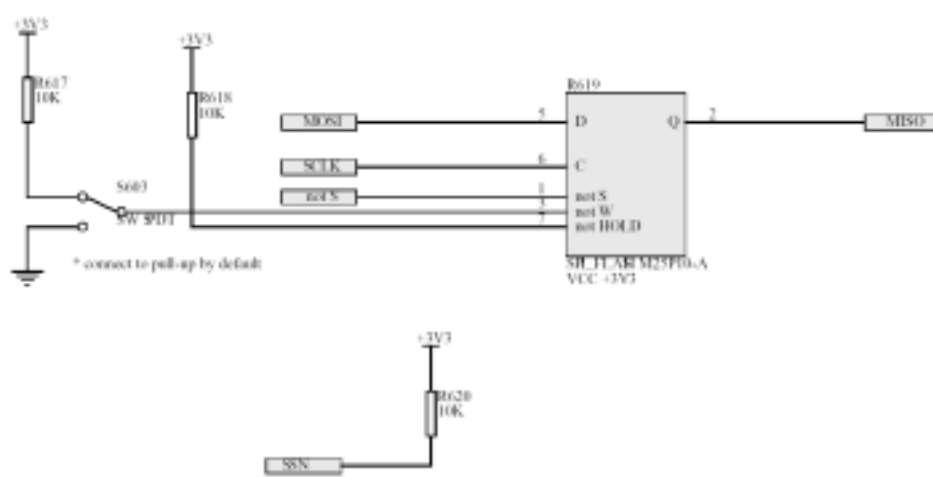
In smartcard mode, the UART will pull the TX line low for a Baud clock period, if a parity error is detected when a character is received. This signals to the smartcard that an error has occurred and it should resend the character. An additional time guard register is enabled. This is a programmable transmission delay between each character. This allows line pacing so that you can send data to a remote device with limited processing and memory resources and ensure it has enough time to process each character before the next one arrives.

Exercise 17: 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

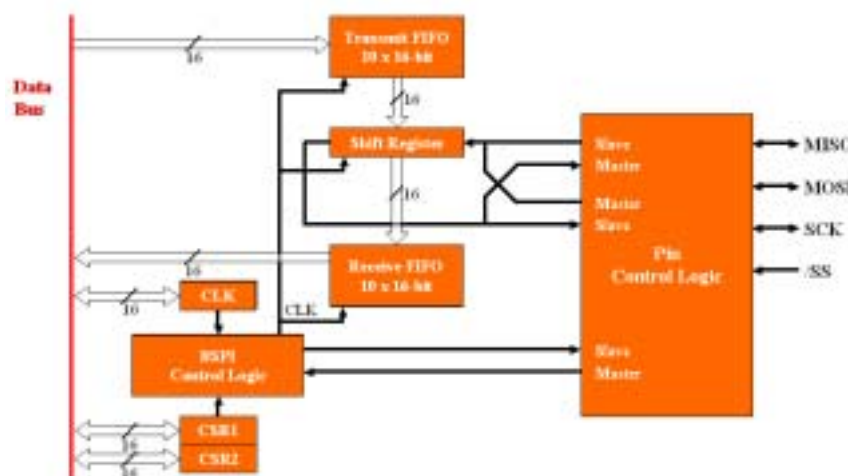
4.8.2 Buffered SPI

The STR7 has two fully independent SPI peripherals that support full duplex communication at speeds up to 5.5 Mbit/sec in master mode and 4 in slave Mbit/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. The SPI channel is a full duplex channel as the data from the master is shifted out as 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 and easy-to-use serial connection. Like the UARTS, both SPI peripherals are identical so code developed to run on one SPI peripheral will also work on the second peripheral.



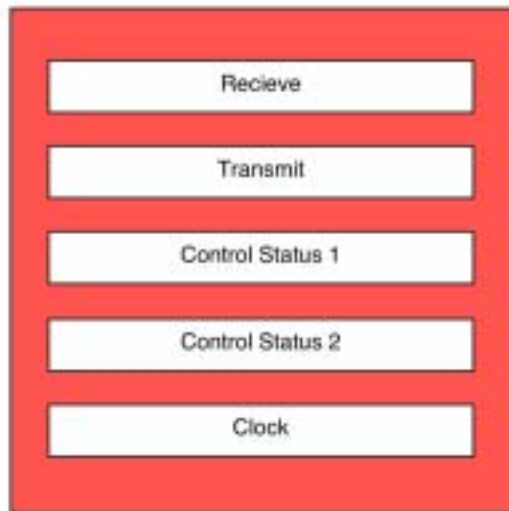
External SPI devices may be easily interfaced to either BSPI peripheral. The master slave select pin should be pulled high during operation.

The SPI peripherals have a standard four-wire interface to the SPI bus, which 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 STR710 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 APB1 peripheral block.

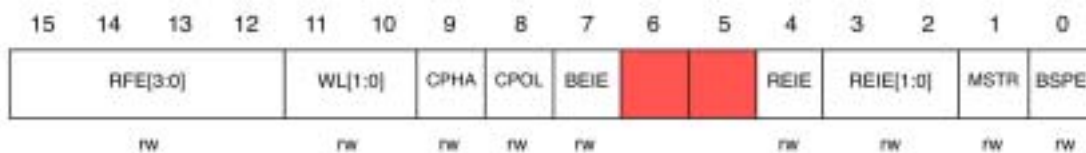


The SPI interface consists of only five registers

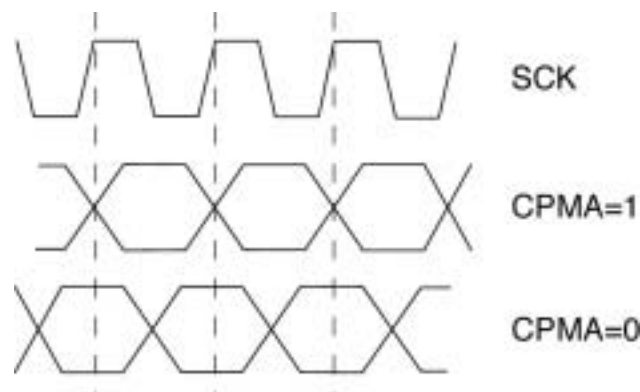
After a hardware reset, the SPI peripheral will be configured as a slave device. If you intend to use it as a master device you must configure the Master clock divide register prior to switching from a slave to a master device. This divides down the APB1 clock to give the SPI clock. The value programmed into the clock divide register must be even and a minimum of six in master mode and a minimum of eight in slave mode.

$$\text{Spi bit rate} = \text{APB1/BSPI_CLK}$$

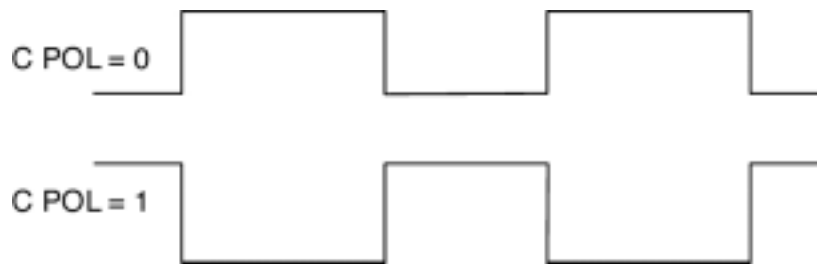
The remaining configuration bits are in the control/status register_1 register. Here the SPI peripheral can be configured to match the transmission characteristics of the other devices on the bus.



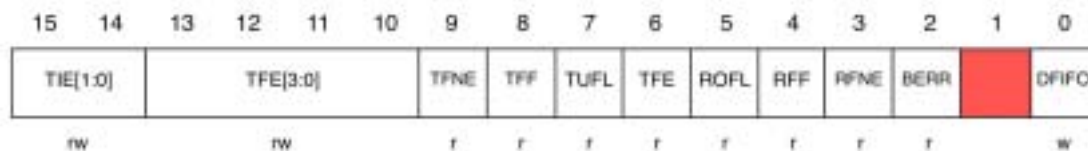
Control status register 1 configures the operating mode of the BSPI peripheral and sets the clock parameters to match those of the external SPI device. Here you can also set the word length and enable the FIFO's



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. 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

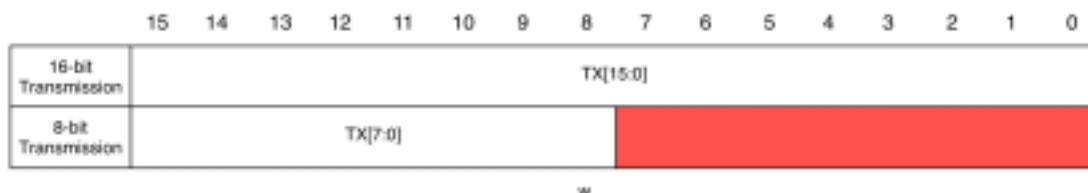
Firstly, most microcontrollers only support transmission and reception of eight bit words. The STR7 SPI peripheral supports both 8 or 16-bit word lengths. Secondly, both the receive and transmit shift registers have FIFO buffers of up to ten words deep. Both of these features are configured in status/control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

4.8.2.1 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 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

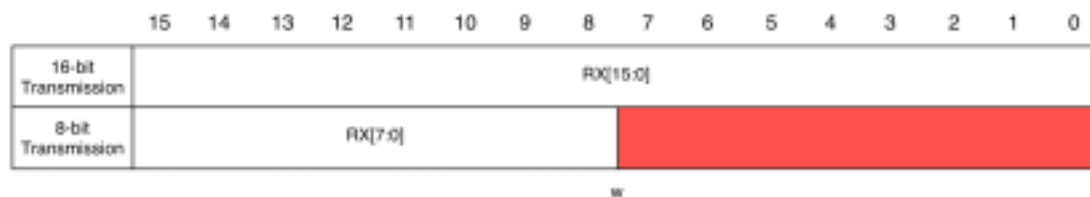
when the transmit FIFO is empty and the transmit shift register has finished sending data. A transmit interrupt can be generated when either the FIFO full, FIFO empty or underflow flag is set.

```
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.

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.



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

4.8.2.2 SPI Slave Mode

In slave mode the slave select and serial clock pins act as 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
```

4.8.2.3 SPI Error Interrupts

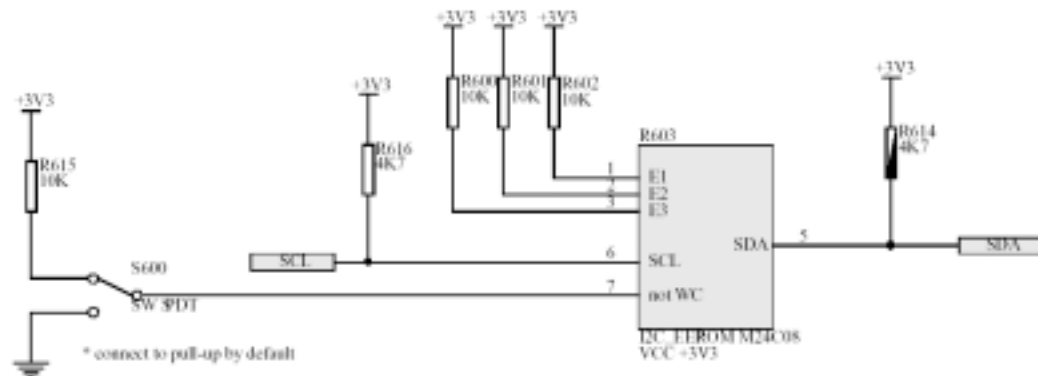
In addition to the transmit and receive interrupts, the SPI peripheral has a bus error interrupt. In master mode this interrupt will be generated if the slave select pin is pulled low by another device. This indicates that there is a second master on the bus and that there is a bus contention.

Exercise 18: BSPI Loop Back

This exercise configures both SPI modules as master and slave and then transfers data between them

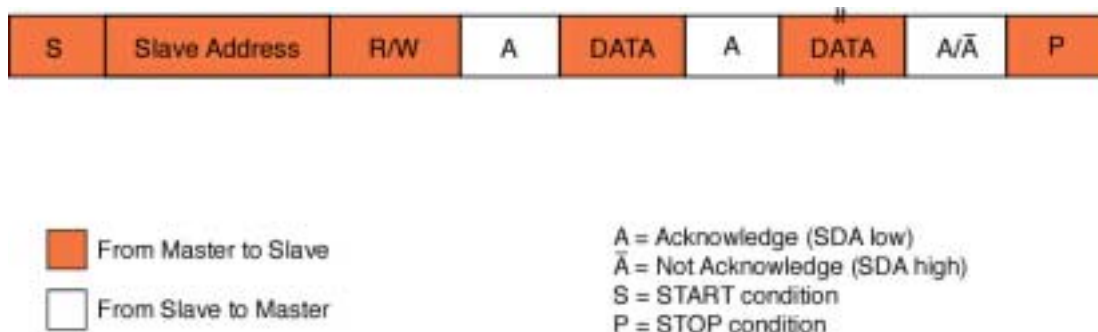
4.8.3 The I2C Module

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 STR71x 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
I2C_CR   = 0x00000020; // enable the I2C module
I2C_CR   = 0x00000020; // write twice to enable
I2C_CR |= 0x00000001; // enable the ITE interrupt
I2C_CR |= 0x00000004; // enable Ack
I2C_CCR  = 0x0000002C; // set the bit rate at 50K
I2C_ECCR = 0x00000003; // set the upper clock register bits
I2C_OAR1 = 0x00000002; // Set own address

```

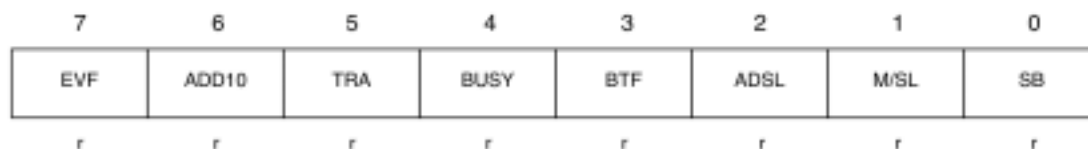
4.8.3.1 I2C Addressing

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.

4.8.3.2 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.



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);
}
```

4.8.3.3 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 19: I2C loop back

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

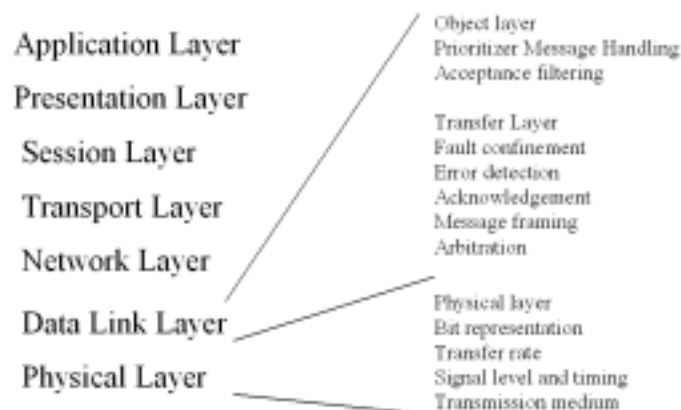
4.8.4 The CAN Controller

Variants of the STR7 are available with an integrated CAN controller on-chip. 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 peripheral.

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.

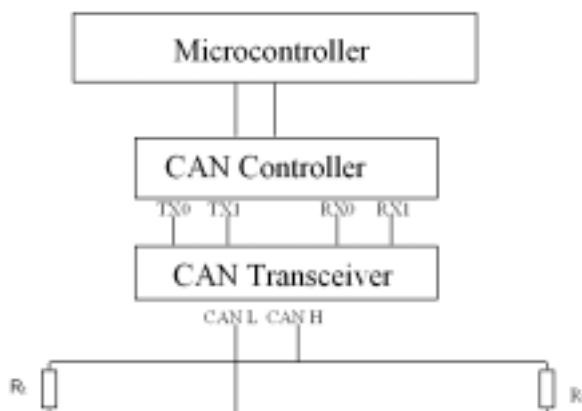
4.8.4.1 ISO 7 Layer Model

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.



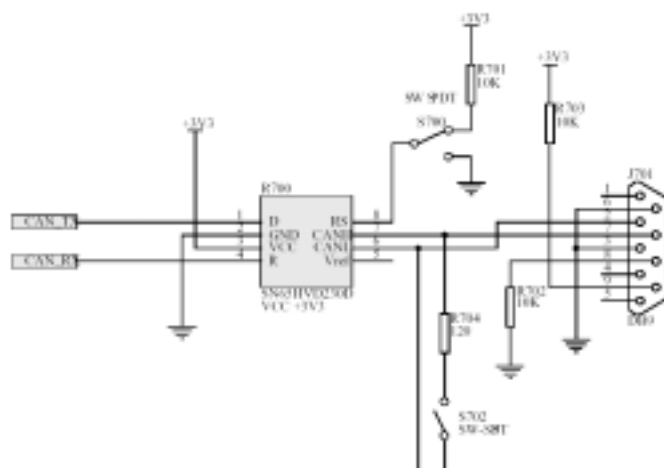
4.8.4.2 CAN Node Design

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!



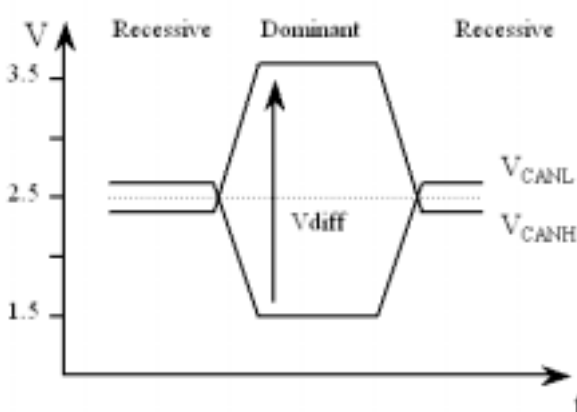
CAN node hardware: A typical CAN node has a microcontroller, CAN controller, physical layer and is connected to a twisted pair terminated by 120 Ohm resistors.

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.

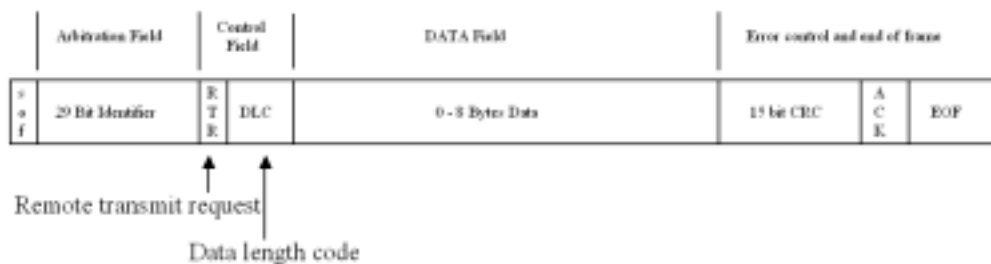


CAN Physical layer signals: On the CAN bus logic zero is represented by a maximum voltage difference called dominant, logic 1 by bus idle called recessive. A dominant bit will overwrite a recessive bit.

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.

4.8.4.3 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:

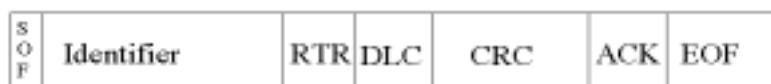


CAN message packet : The message packet if formed by the CAN controller, the application software provides the data bytes the message identifier and the RTR bit.

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.



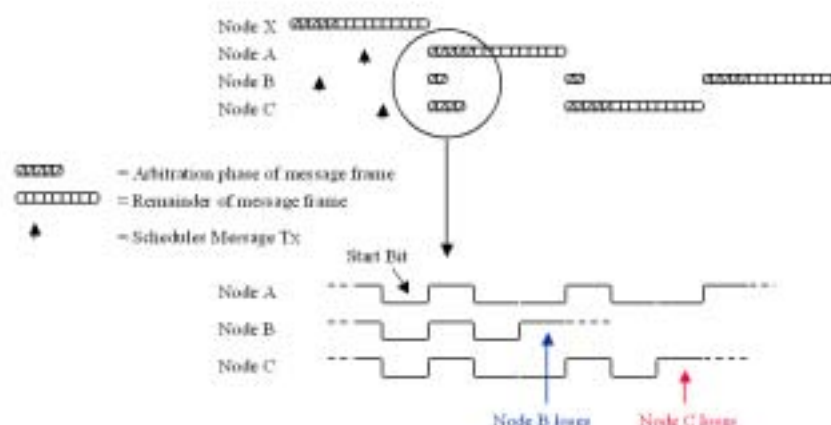
Remote Transmit Request: The RTR frame is used to request message packets from the network as a master slave transaction

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

4.8.4.4 CAN Bus Arbitration

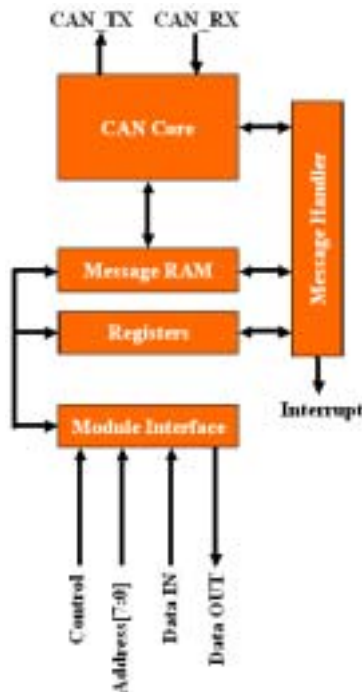
If a message is scheduled to be transmitted onto the bus and the bus is idle it will be transmitted and may be picked up by any interested node. If a message is scheduled and the bus is active it will have to wait until the bus is idle before it can before transmission. If several messages are scheduled 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.

4.8.4.5 STR7 CAN Module



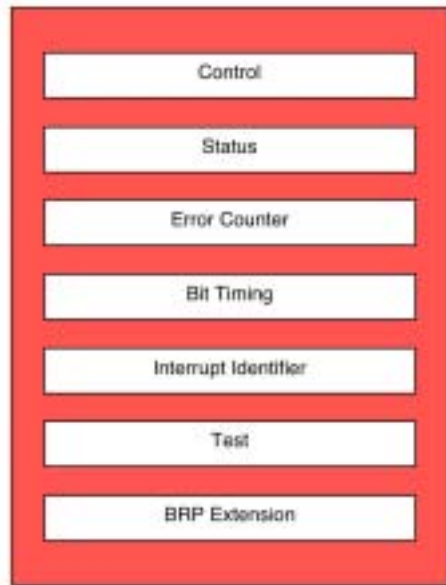
The STR7 CAN module is a full CAN module with an integral message RAM that supports 32 user - configurable receive/transmit buffers.

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.



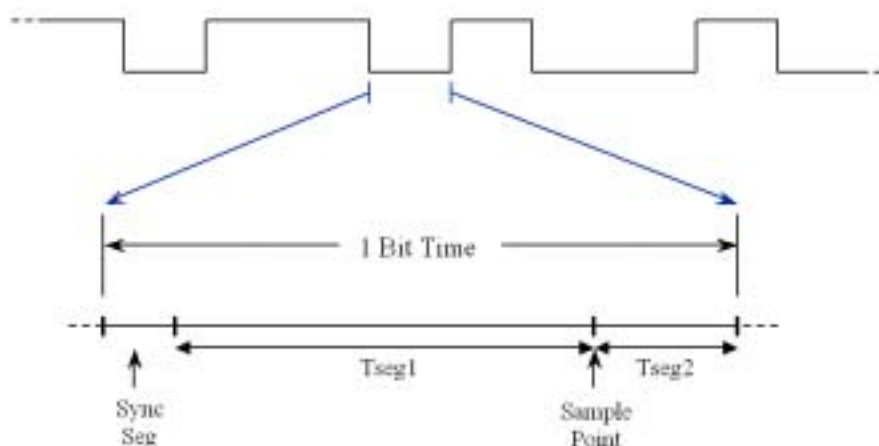
The CAN protocol registers configure the CAN bus parameters and manage the interrupt handling and error containment

4.8.4.5.1 Initialising The CAN Controller

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

4.8.4.5.1.1 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 $\text{Tseg1} = 11 - 1 = 10$ 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.

$\text{Tseg2} \geq 5 \times \text{Tq}$, then program SJW to 4

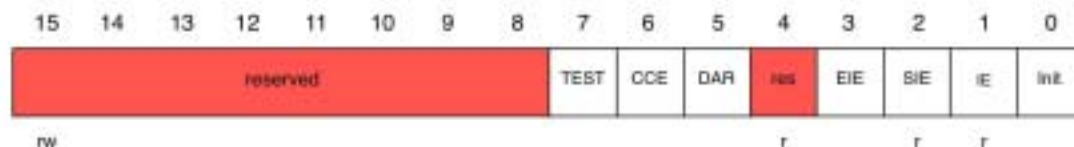
$\text{Tseg2} < 5 \times \text{Tq}$, then program SJW to $(\text{Tseg2} - 1) \times \text{Tq}$

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.

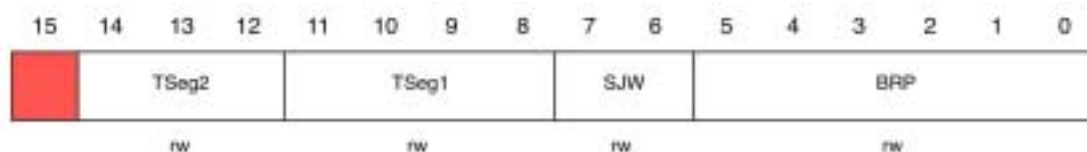
4.8.4.5.2 Configuring The CAN Module

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.

```

CAN_CR      = 0x000000C1;    // init config change and test enable
CAN_BRPR    = 0x00000000;    // set extended Baud rate prescaler to 0
CAN_BTR     = 0x000045C3;    // set bit rate to 500K for FCLK = 24 MHz
CAN_TESTR   = 0x00000004;    // set into basic mode
CAN_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.

4.8.4.5.3 Using The CAN Module

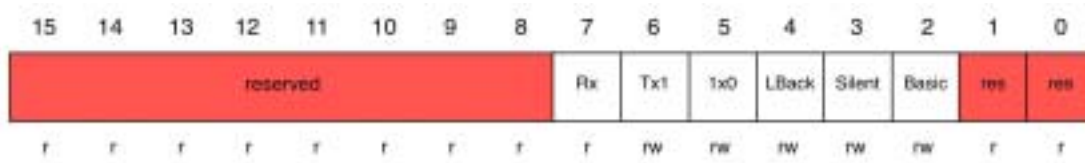
Once the CAN module is fully configured, it may be used to send and receive packets of data. The STR7 CAN module has two operating modes. For simple CAN networks with low data rates, there is an easy-to-use basic CAN mode which has a single transmit and receive buffer. For more demanding networks with 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

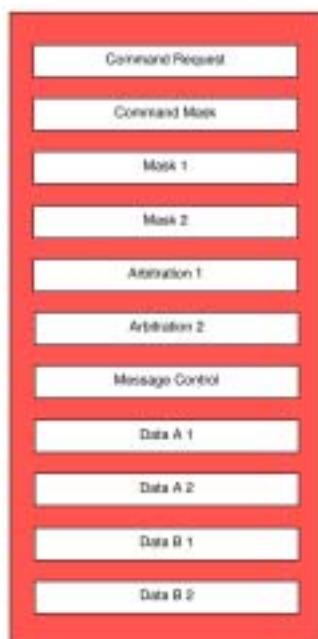
4.8.4.5.3.1 Basic Mode

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.

```
CAN_IF1_A1R = 0x00000000;    // Clear the arbitration register
CAN_IF1_A2R = 0x0000A001;    // Set the ID for a standard message
CAN_IF1_MCR |= 0x00000001;    // Set the data length code to send just one byte
CAN_IF1_DA1R = 0x00000055;    // Write the data in data0 register
CAN_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.

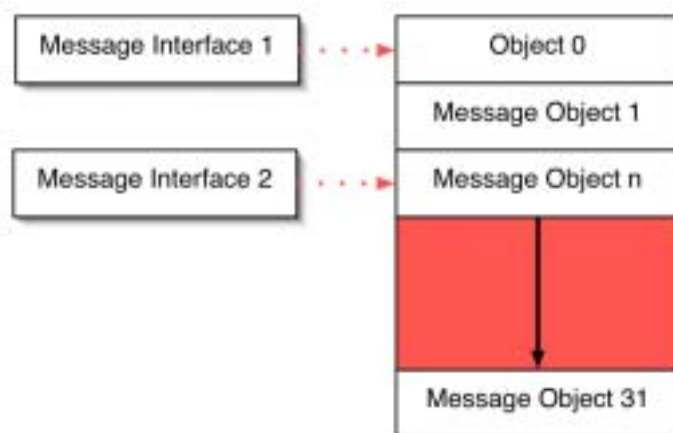
```
CAN_IF2_A2R = 0x8000;
```

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(CAN_IF2_MCR & 0x8000)
{
    id = (CAN_IF2_A2R >>2) &0x000007FF;    // read the ID 11 bit standard
    dlc = CAN_IF2_MCR & 0x0x0000000F    // read the data length code
    test = CAN_IF2_DA1R;    // read the message data
    CAN_IF2_MCR &= ~0x8000;    // clear the new data flag
}
```

4.8.4.5.3.2 Full CAN Mode

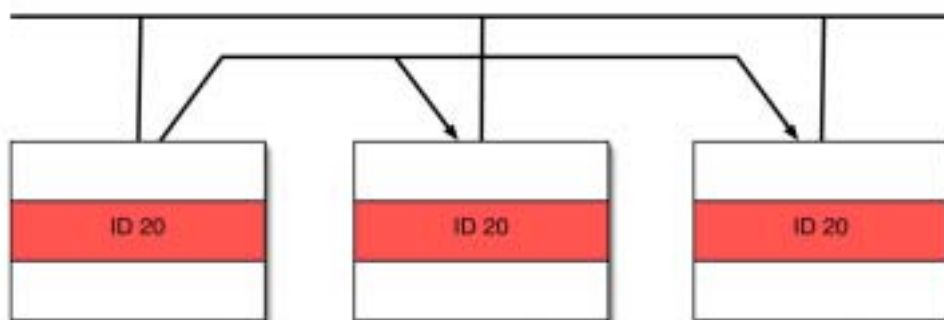
While basic CAN mode is easy to use, it does mean that an interrupt will be generated for every message received and even with the use of the message masks in a medium to heavily loaded network, this will put a significant loading on the CPU. The STR7 CAN module has a full CAN mode that enables a block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

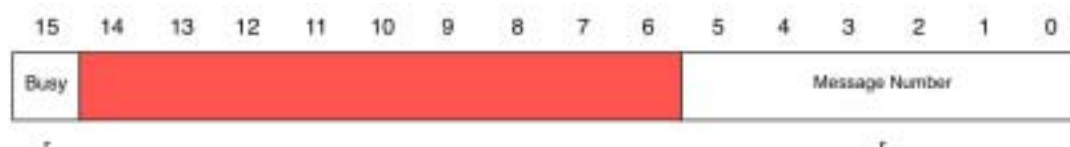
sets both act as interface “windows” and unlike Basic CAN mode, do not have specific transmit or receive functions. They can be used to perform either task. Each of the 32 message objects may be configured as a transmit or receive buffer when the CAN controller is initialised. The transmit objects can 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 busv 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.

```

CAN_CR          = 0x00000041;          //init and config change enable
CAN_BRPR        = 0x00000000;          //Clear the extended BRP
CAN_BTR         = 0x000045C3;          //set bit rate to 500K

//Configure a transmit object ( could use the IF2 registers)
CAN_IF1_CMR     = 0x000000F0;          // enable write of mask,arb and control registers
CAN_IF1_M1R     = 0x00000000;          // Clear the lower mask registers
CAN_IF1_M2R     = 0x00000000;          // Clear the upper
CAN_IF1_A1R     = 0x00000000;          // Clear lower arbitration register
CAN_IF1_A2R     = 0x0000A004;          // Message valed, std frame,transmit, ID1
CAN_IF1_MCR     = 0x00000001;          // DLC = 1
CAN_IF1_CRR     = 0x00000001;          // write to message object one
while(CAN_IF1_CRR & 0x8000);          // wait for the message object to be written too

//configure a receive object ( could use the IF2 registers)
CAN_IF1_CMR     = 0x000000F0;          // enable write of mask,arb and control registers
CAN_IF1_M1R     = 0x00000000;          // Clear the lower mask registers
CAN_IF1_M2R     = 0x00000000;          // Clear the upper
CAN_IF1_A1R     = 0x00000000;          // Clear lower arbitration register
CAN_IF1_A2R     = 0x00008004;          // Message valed, std frame,receive, ID1
CAN_IF1_MCR     = 0x00000000;          // clear the message control register
CAN_IF1_CRR     = 0x00000002;          // write to message object two

while(CAN_IF1_CRR & 0x8000);          // Wait for the message object to be written too

CAN_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.

```

CAN_IF1_CMR     = 0x00000086;          //Write data and control register set TXrequest bit
CAN_IF1_DA1R    = 0x00000055;          //load some data
CAN_IF1_CRR     = 0x00000001;          //write to message object one

while(CAN_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(CAN_ND1R & 0x02)                    // if message object 2 has received new data
{
    CAN_IF1_CMR = 0x00000003;          // Mark all data registers for read
    CAN_IF1_CRR = 0x00000002;          // read message object two

    while(CAN_IF1_CRR & 0x00008000);    // Wait for the busy bit to clear

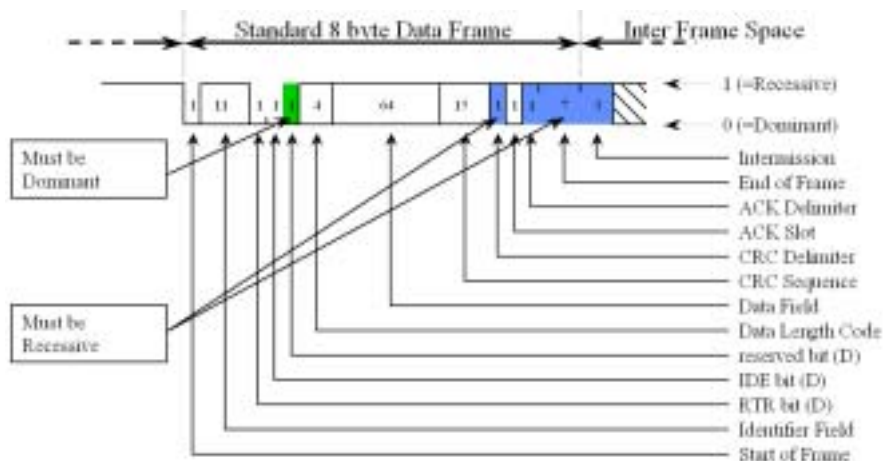
    CAN_data_0_1 = CAN_IF1_DA1R;        // read the data from the message registers
    CAN_data_2_3 = CAN_IF1_DA2R;
    CAN_data_4_5 = CAN_IF1_DB1R;
    CAN_data_6_7 = CAN_IF1_DB2R;
}

```

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.

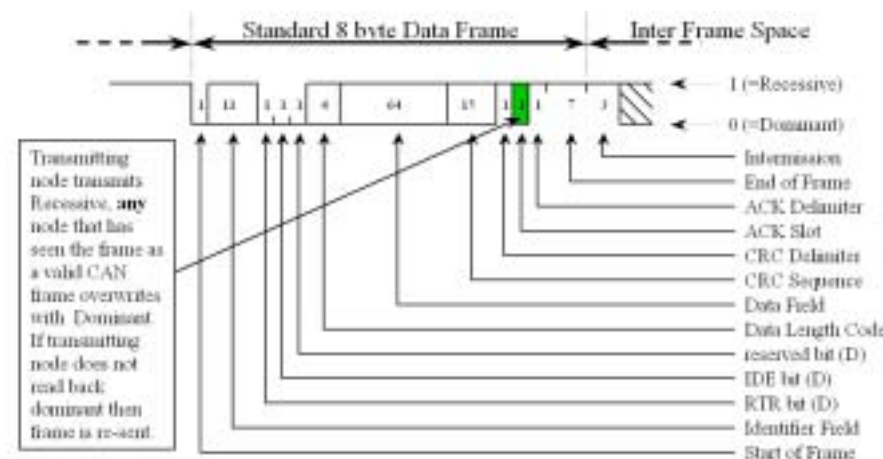
4.8.4.5.4 CAN Error Containment

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.

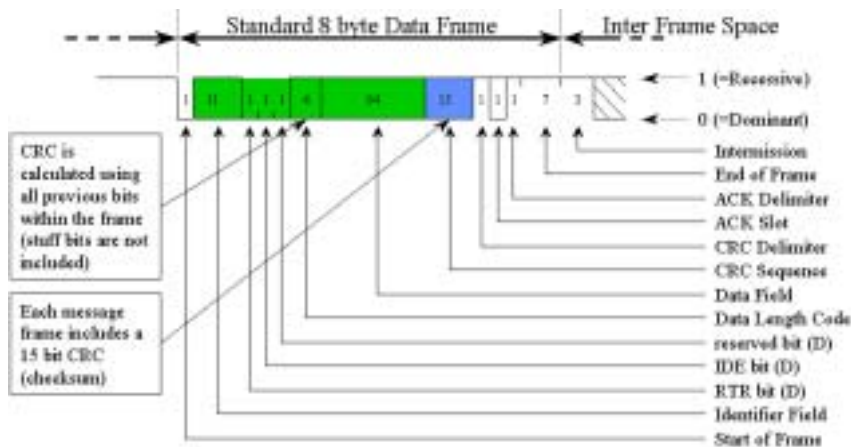


Frame Check: The frame check tests that a correctly formatted Can message has been received.

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.

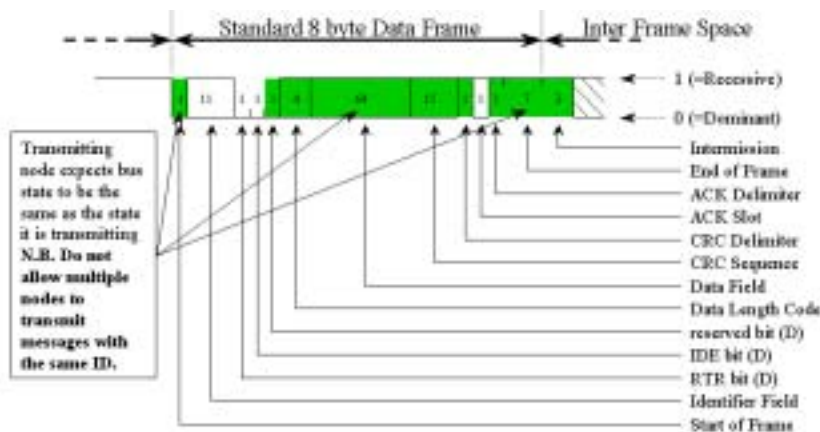


The CAN message packet also contains a 15 bit CRC which is automatically generated by the transmitter and checked by the receiver. This CRC can detect and correct 4 bits of error in the region from the start of frame to the beginning of the CRC field. If the CRC fails and the message is rejected an error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

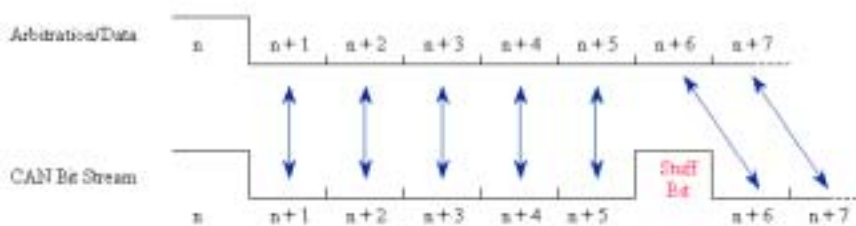
Once a node has won arbitration it will start to write its message onto the bus. As during arbitration as each bit is written onto the bus the CAN controller is reading back the level written onto the bus. As the node has won arbitration nothing else should be transmitting so each bit level written onto 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 one logic 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.

4.8.4.5.4.1 STR7 Error Handling

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

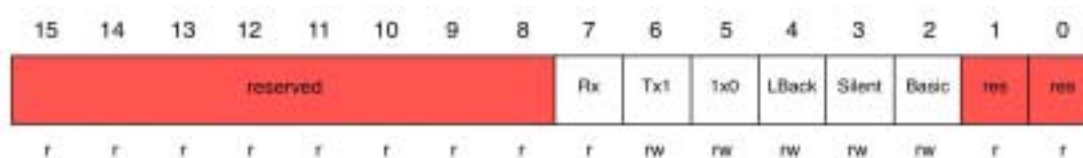


The status register contains the error warning flags and a Last error code that details the last fault that occurred on the CAN bus.

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.

4.8.4.5.5 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.

4.8.4.6 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 20: CAN loopback

This exercise configures the CAN controller in Full CAN mode and uses the loopback test mode to simulate sending and receiving CAN message packets

4.8.5 USB 2.0 Full Speed Slave Peripheral

One of the most complex peripherals available in the STR7 family is the STR710 USB peripheral. In order to fully understand this peripheral you will also need a clear understanding of how the USB network operates and to build a USB based system, you will also need to know how to build a PC application that can access the USB network. This requires a lot of background knowledge and some skill with PC programming and the windows operating system, which is normally beyond the remit of embedded firmware development.

4.8.5.1 Introduction to USB

Before we look at the USB peripheral we will give an overview of the complete USB system. This can be split into three parts; USB theory of operation, overview of the STR7 peripheral and introduction to the PC application requirements.

The USB network was first supported in the Windows operating system by adding additional drivers to Windows 98. The driver support was added as a standard part of the Windows operating system in Windows 2000. The goals of USB was primarily to allow easy expansion of a PC's peripherals with a "foolproof" plug-and-play network. The USB 1.0 standard was released in 1996 and was soon superseded by version 1.1. The current revision of the standard is 2.0 and is maintained by the USB implementers forum, who host a website at www.usb.org. You can download the full specification from this website, along with a number of useful utilities which we shall look at later. The USB peripheral on the STR7 supports USB2.0 and throughout this is the revision of the specification that we will be working to.

4.8.5.1.1 USB Physical Network

The USB network supports three communication speeds; low speed which runs at 1.5 Mbits/sec and is primarily used for simple devices like keyboards and mice; full speed which runs at 12 Mbits/sec and is suitable for most other PC peripheral and finally High speed, which runs at 480 Mbits/sec and is aimed at video devices that require a high bandwidth.

The USB specification also defines the physical cabling and connectors. This ensures that any user will put the right plug in the right socket.. The two connectors are shown below.

PERFORMANCE	APPLICATION	ATTRIBUTES
LOW SPEED Interactive Devices 10 - 100kb/s	Keyboard, Mouse Game peripherals Monitor Configuration	Lower cost Hot plugging Ease of use Multiple peripherals
FULL SPEED Phone, Audio Compressed Video 500Kb/s - 10Mb/s	Printers Scanners Telephony Audio	Low cost Hot plugging Ease of use Guaranteed latency Guaranteed Bandwidth Multiple devices
HIGH SPEED Video, Disk 25 - 500 Mb/s	Video Mass Storage	High Bandwidth Guaranteed latency Ease of use

The USB protocol tightly defines the USB physical layer including the network connectors

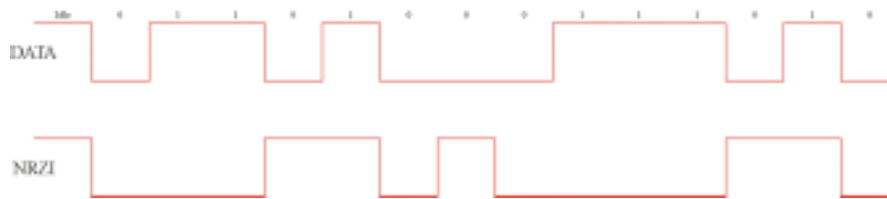
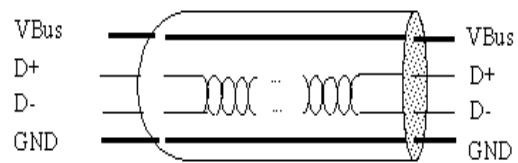
Internally the cable has four shielded wires. Two are used to carry power and two are for data. The power wires carry 5V which can deliver 500mA of current. The data wires are called D+ and D- . The maximum length of a shielded full speed cable is 5 meters. Since the physical layer transceiver is incorporated into the STR71x, the hardware design consists of connecting the D+ wire to the D+ pin and the D- wire to the D- pin as we shall see later only one other external component is required to complete the design.



A standard USB cable contains two data wires D+ and D- and two power wires carrying a 5V supply

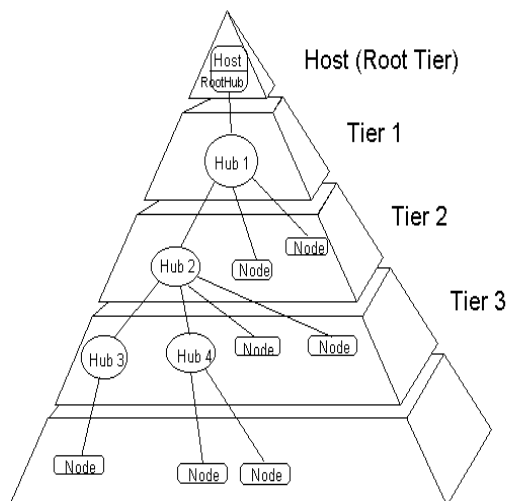
The physical layer signalling uses a Non-Return to Zero (NRZ) inverted encoding scheme. This means that the logic level on the USB network will change state every time it sees a logic zero in the datastream. This allows remote nodes to extract the data and a clock signal with which to synchronise themselves. Because the physical layer of the

USB network is heavily defined, you will not generally need to examine the physical layer signals. USB debugging is generally concerned with observing transactions at the data packet level. However there is some necessary jargon to be aware of when discussing the bit level signalling on the USB network. On a full speed USB cable, a logic one is 5V and is called a K-state and a Logic zero is 0 V and is called a J-state. To keep you on your toes, the signalling voltages are inverted for low speed communication!



The physical data signalling is encoded as a non return to zero inverted data stream.

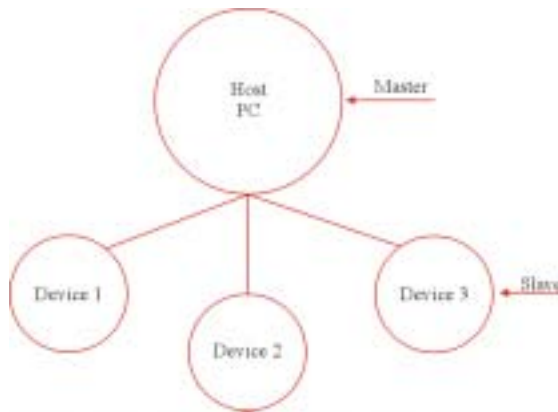
The physical USB network is implemented as a tiered star network. The USB root node must be a PC (or other bus master) and this provides one attachment port for an external USB peripheral. If more than one peripheral is required, you can connect a hub to the root port and the hub will provide additional connection ports. For large numbers of USB peripherals, further hubs may be added in order to provide additional ports for peripherals. The USB network can support up to 127 external nodes (hubs and devices) and six tiers of hubs and requires one bus master.



The physical USB network is a tiered star with one master node and up to 127 slave nodes and a maximum of five tiers

4.8.5.1.2 Logical Network

However to the developer the logical USB network appears as a star network. The hub components do not introduce any programming complexity are essentially invisible as far as the programmer is concerned. So if you develop a USB device by connecting it to a root port on the PC the same device will work when connected to the PC via several intermediate hubs. So to the programmer the USB network appears as a star network with the PC at the center and all the USB devices are available as addressable nodes.



To the programmer the USB network appears as a master slave star network

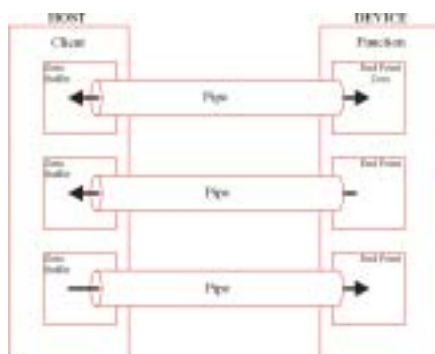
The other key feature of the USB network is that it is a master-slave network. The PC is in control and is the only device on the network that can initiate a data transfer. With USB 2.0 peer-to-peer communication is not possible and the STR71x USB peripheral is a slave device only and cannot act as a master. A version of USB called "USB on the go" is a new addition to the specification and directly supports peer-to-peer communication allowing for example, pictures stored on a camera to be transferred directly onto a USB memory stick, without the need for a PC or other USB master.

4.8.5.1.3 Signalling Speed

Since the USB network is designed to be plug-and-play, the PC will have no knowledge of a new device when it is first plugged onto the network. The first thing that the PC needs to determine when a new device is added is the bit rate required to communicate to the new device. This is done by adding a pull-up resistor to either the D+ or D- line. If the D+ line is pulled up, the PC will assume that a full speed device has been added. If it is D- , it means low speed. High speed devices will first appear as full speed and then negotiate up to high speed, once the connection has been established.

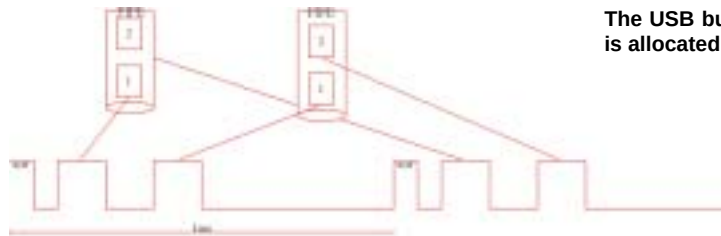
4.8.5.1.4 USB Pipes

Once a device has been connected to the PC and the signalling speed has been determined, the PC can start to transfer data to and from the new device. These data packets are transferred over a set of logical connections called pipes. A pipe originates from a buffer in the PC and is connected to a remote device with a specific device address. The pipe is terminated inside the device at an Endpoint. In microcontroller terms the Endpoint may be viewed as a buffer where the data is stored and an interrupt that signals the CPU that a new data packet has arrived.



Each USB slave is characterised by a local address and a set of logical endpoint buffers. The Host creates logical connections called "pipes" to each endpoint which are used to transfer packets of information

These logical pipes are implemented on the serial bus as time division multiplexing. Every 1msec the PC sends a Start of Frame (SOF) token to delineate the 12 Mbit/sec bus into a series of frames. Each pipe is allocated a slot in each frame so it can transfer data as required.



The USB bus is split into 1msec frames and each pipe is allocated a number of packets per frame

USB supports several different types of pipes with different transfer characteristics in order to support different types of application. It is possible to design a USB device capable of supporting several different configurations which can be dynamically changed to match the running PC application. The types of pipes available are control, interrupt, bulk and isochronous. All of these pipes are unidirectional, except the control pipe, which is bidirectional. The control pipe is reserved for the PC to send and request configuration information to the device and is generally not used by the application software. Every device has a control pipe and it is always connected to Endpoint Zero. So when a new device is plugged onto the network, it will always appear as device zero and the PC can communicate to it by sending control information to Endpoint Zero. The remaining types of pipe are used solely for the user application and in the STR7210 there are up to 15-user endpoints, which are allocated as one of the three remaining pipe types.

4.8.5.1.4.1 Interrupt Pipe

The first of the user pipe types is an interrupt pipe. The key thing about an interrupt pipe is that it isn't one. Since only the PC can initiate a data transfer, no network device can asynchronously communicate to the PC. With an interrupt pipe the developer can define how often a data transfer is requested between the PC and the remote device. This can be between 1msec and 255msec. So really in USB an interrupt pipe has a defined polling rate. In the case of a mouse we can guarantee a data transfer every 10 msec for example. Defining the polling rate does not guarantee that data will be transferred every 10 msec, but rather that the transaction will occur somewhere within the tenth frame. So a certain amount of timing jitter is inherent in a USB transaction.

4.8.5.1.4.2 Isochronous Pipe

The second type of user pipe is called an isochronous pipe. Isochronous pipes are used for transferring real time data such as audio data. Isochronous pipes have no error detection and an isochronous pipe sends a new packet of data every frame regardless of the success of the last packet. So in an audio application a lost or corrupt packet will sound like noise on the line until the next successful packet arrives. An important feature of isochronous data is that it must be transferred at a constant rate. Like an interrupt pipe, an isochronous pipe is also subject to the kind of jitter described above. So in the case of isochronous data no interrupt is generated when the data arrives in the endpoint buffer. Instead the interrupt is raised on the start of frame token. This guarantees a regular 1 msec interrupt on the isochronous endpoint allowing data to be read at a regular rate.

4.8.5.1.4.3 Bulk Pipe

The bulk pipe is for all data that is not control, interrupt or isochronous. Data is transferred in the same manner and packet sizes as with an interrupt pipe, but bulk pipes have no defined polling rate. A bulk pipe will take up any bandwidth that is left over after the other pipes have finished their transfers. If the bus is very busy, then a bulk transfer may be delayed. Conversely, if the bus is idle then multiple bulk transfers can take place in a single 1 msec frame, where interrupt and isochronous are limited to a maximum of one packet per frame. An example of bulk transfers would be sending data to a printer. As long as the data is printed in a reasonable time frame the exact transfer rate is not important.

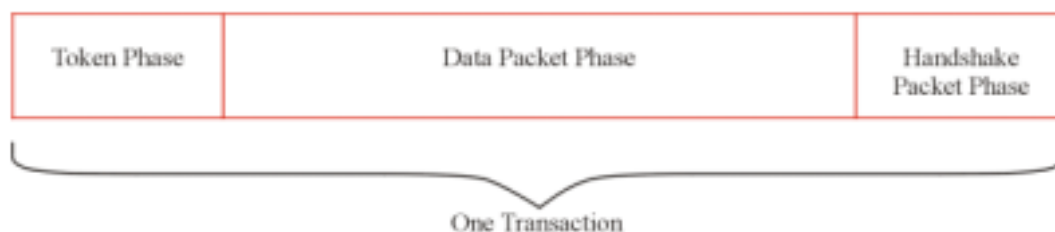
4.8.5.1.5 Bandwidth Allocation

So, in terms of bandwidth allocation, the control pipes are allocated 10% Interrupt, Isochronous is given 90% and bulk makes do with any idle periods on the network. These are maximum allocations so on most networks there will be plenty of unused bandwidth. The operating system on the PC is responsible for bandwidth management and should not let a new device on to the network if the resources are not available to service it.

	Control	Isochronous (FS and HS)	Interrupt	Bulk (FS and HS)
Data format	Pre- or vendor defined	Stream	No structure	Stream
Transfer direction	Bi-direction	Uni-direction	Either input or output	Either input or output
Packet size (bytes)	8, 16, 32, or 64	1 ~ 1023	LS: < 8 FS: < 64	8, 16, 32, or 64
Bus access	Best effort guaranteed	< 90% Periodic	< 90% 1ms~255ms	Good effort no guaranteed
Data Sequence	Setup, data and status	No retry, data toggling	Retry, data toggling	Retry, data toggling

4.8.5.1.6 USB Network Transactions

As we have seen, the data transfer over the USB network is time division multiplexing. The bus is delineated into frames by sending a start of frame taken every millisecond and the pipes transfer packets of data within these frames. Each data transfer is constructed from a three packet transaction.



Each USB transaction consists of three packets which are exchanged between the master and slave nodes

This consists of a token phase that defines what type of transaction is about to take place. Next comes a data phase that transfers the necessary data and finally a handshake phase that establishes that the transfer has been successful. Each of these packets is made out of the same basic structure that contains fields for packet type, destination address and endpoint, data field if necessary and error checking.



Each of the USB packets, token data or handshake has the same physical structure.

4.8.5.1.6.1 Token Phase

There are five tokens that are available in USB 2.0. We have already seen the Start of Frame (SOF) token which is used to mark the start of a 1 msec frame. This token does not have an associated data or acknowledge packet. The IN token starts a transfer of data into the PC and the OUT starts a transfer of data from the PC to the network device. All references to data direction are considered relative to the PC (i.e. IN to the PC or OUT of the PC). The setup packets are exclusive to the control channel and are used to send commands to the USB device. This may be a request for configuration information, or a command to set a particular configuration. Finally the PREAMBLE packet is used to signal the start of a low speed transaction. The PREAMBLE packet causes the full speed and high speed ports on HUBS to close and the low speed ports to open. Then a transaction follows and at the end of the low speed transaction the ports revert to their original state.

4.8.5.1.6.2 Data Phase

Once the token has been sent by the PC it will be followed by a data packet. There are two types of data packet called DATA0 and DATA1. These data packets perform an identical function and the only difference is one bit in the packet header called the data toggle bit which is used for error detection. I will explain this when we look at the error containment methods used in the USB protocol.

4.8.5.1.6.3 Handshake Phase

The final phase of a bus transaction is the handshake phase. In this phase there are three handshake packets that are used to signal whether a successful transaction has taken place.

The ACK packet is used to acknowledge a successful transfer of data and will end the bus transaction. The PC will then start the next token phase. The NAK packet is a not acknowledge. This may signal that the transfer has failed because an error checking rule has been violated. A NAK may also be generated if the Endpoint buffer is not ready for the transaction. So, if data from a previous OUT transfer is still in the Endpoint buffer, the USB peripheral will generate NAK handshakes until the CPU has read the data. If a NAK is generated the PC will attempt to resend the same transaction in the next frame. Remember that for isochronous transfers the handshake is ignored. If the Endpoint buffer is full and the CPU never removes the data we will get continuous NAK exchanges on the pipe every frame. This will start to waste bandwidth. For this reason there is a third form of handshake called STALL. A STALL handshake is used to tell the PC that USB device can no longer communicate on this pipe. For example, if a printer ran out of paper and its memory became full, sending further documents would cause lots of NAK packets on the bus. In this case, the printer can tell the PC via a stall packet that it is not ready to receive any more data and communication will stop over this pipe. The PC could then try to find out why the data pipe is stalled by requesting data packets on an IN pipe which could transfer diagnostic information.

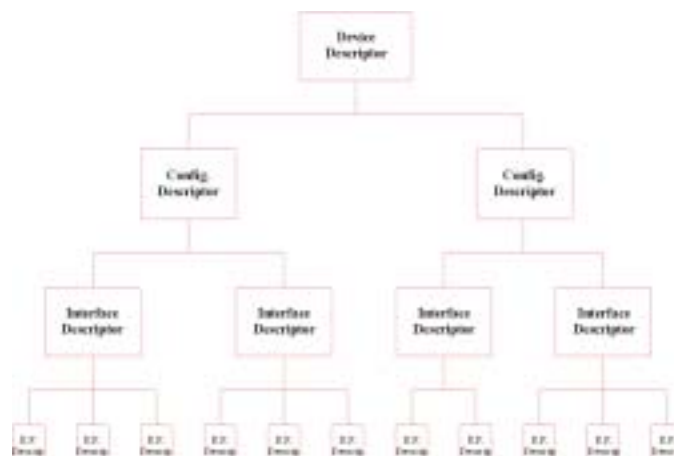
4.8.5.1.7 Error Containment

Within the USB protocol there are six different methods of error containment. There are packet error checks which include a CRC, packet ID check and bit stuffing rules. When a packet is transferred the protocol can detect a false end of packet, bus time, loss of activity and babble where a node continues to transmit beyond its end of packet. The two data packets also support a data toggle error check where a data one packet must always follow a data zero packet and vice versa. So if, for example, a data one packet gets lost, the node receiving the data will get a data zero packet followed by another data zero packet and an error will be raised. All of these error handling methods are handled within the USB peripheral and are essentially transparent to the programmer.

4.8.5.1.8 Device Configuration

When a device is first connected to the PC, its signalling speed is discovered and it will have Endpoint 0 configured to accept a control pipe. In addition, every new device that is plugged onto the network will be assigned address zero. This way the PC knows what bit rate to use and will have one control channel available at address zero endpoint zero. This control pipe is then used by the PC to discover the capabilities of the new device and to add it to the network. The process the PC uses to gather this information is called “Enumeration”. So in addition to configuring the USB peripheral on the STR71x you need to provide some firmware that responds to the PC enumeration requests.

The data that the PC requests is held in a hierarchy of descriptors. The descriptors are simply arrays of data that must be transferred to the PC in response to enumeration requests. As you can see from the picture below it is possible to build complex device configurations because the USB network has been designed to be as flexible and as future proof as possible. However the minimum number of descriptors required is a device descriptor, configuration descriptor, interface descriptor and three endpoint descriptors (one control one IN and one OUT pipe).



The configuration information for every USB node is described as a hierarchy of descriptors which are transferred to the PC during the device enumeration procedure

4.8.5.1.9 Device Descriptor

At the top of the descriptor tree is the device descriptor. This descriptor contains the basic information about the device. Included in this descriptor is a vendor ID and product ID field. These are two unique numbers that identify what device has been connected. The windows operating system will use these numbers to determine what device driver to load. The vendor ID number is the number assigned to each company producing USB-based devices. The USB implementers' forum is in charge of administering the assignment of vendor IDs. You can purchase a vendor ID from their website (www.usb.org) for \$1500 administration charge (they must be very heavy) or \$2500 if you want to use the USB logo on your product. Either way you must have a vendor ID if you want to sell a USB product on the open market. The product ID is a second 16-bit field which contains a number assigned by the manufacturer to identify a specific product. The device descriptor also contains a maximum packet size field.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant DEVICE (01h)
2	bcdUSB	2	USB specification release number (BCD)
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize(0)	1	Maximum packet size for Endpoint 0
8	idVendor	1	Vendor ID
10	idProduct	1	Product ID
12	bcdDevice	1	Device release number (BCD)
14	iManufacturer	1	Index of string descriptor for the manufacturer
15	iProduct	1	Index of string descriptor containing serial number
16	iSerialNumber	1	Number of possible configurations
17	bNumConfigurations	1	Number of possible configurations

4.8.5.1.10 Configuration Descriptor

The configuration descriptor contains information about the device's power requirements and the number of interfaces it supports. A device can have multiple configurations and the PC can select the configuration that best matches the requirements of the application software it is running

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Configuration (02h)
2	bTotalLength	2	Size of all data returned for this configuration in bytes
4	bNumInterface	1	Number of interfaces the configuration supports
5	bConfigurationValue	1	Identifier for Set_Configuration and Get_Configuration requests
6	iConfiguration	1	Index of string descriptor for the configuration
7	bmAttributes	1	Self power/bus power and remote wakeup settings
8	MaxPower	1	Bus power required, expressed as (maximum milliamperes/2)

4.8.5.1.11 Interface Descriptor

The interface descriptor describes a collection of endpoints. This interface will support a group of pipes which are suitable for a particular task. Each configuration can have multiple interfaces and these interfaces may be active at the same time or can be dynamically selected by the PC.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant interface (04h)
2	bInterfaceNumber	1	Number identifying this interface
3	bAlternateSetting	1	Value used to select an alternate setting
4	bNumEndPoints	1	Number of endpoints supported, except Endpoint 0
5	bInterfaceClass	1	Class code
6	bInterfaceSubclass	1	Subclass code
7	bInterfaceProtocol	1	Protocol code
8	bInterface	1	Index of string descriptor for the interface

4.8.5.1.12 Endpoint Descriptor

The endpoint descriptor transfers the configuration details of each endpoint supported in a given interface. The descriptor carries details of the transfer type supported, the maximum packet size, the endpoint number and the polling rate if it is an interrupt pipe.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	The constant Endpoint (05h)
2	bEndpointAddress	1	Endpoint number and Direction
3	bmAttributes	1	Transfer type supported
4	wMaxPacketSize	2	Maximum packet size supported
5	bInterval	1	Maximum latency/polling interval/NAK rate

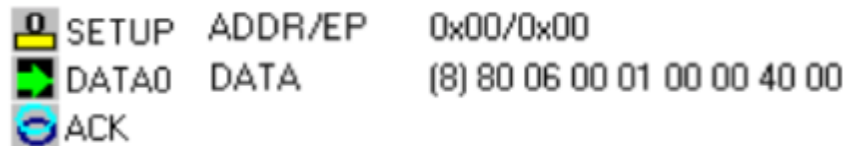
This is not an exhaustive list of all the possible descriptors that can be requested by the PC, but as a minimum the USB device must provide the PC with device, configuration interface and endpoint descriptors.

4.8.5.1.13 Enumeration

The enumeration process takes place over the control channel attached to endpoint zero when the device is first attached. The PC will send a series control transfers that request the USB device to transfer its descriptors to the PC.

Offset	Value	Interpretation
0	80	Device to Host Standard Device
1	06	Get descriptor
2	00	Descriptor index
3	01	Device descriptor
4	0000	Index
6	0040	Descriptor length

A typical control transaction recorded and decoded by a USB bus analyser



The PC sends a setup packet to initiate the control transfer. This is followed by a data packet which contains the command codes for the control action that the PC wants to carry out. After a successful handshake packet, the PC will send an IN packet and the USB device will return a data packet containing the requested information (typically a descriptor.) So as a minimum the enumeration process will make the following requests:

Get the first eight bytes of the device descriptor.

When the device is first connected, the PC will use the smallest packet size possible to communicate with endpoint zero. This will load the vendor and product ID along with the maximum packet size for endpoint zero. For all further transfers the PC can now adjust its data packet size to the maximum supported by endpoint zero, thus reducing the number of transfers required to complete the enumeration process.

Reset the node.

Once the connection to endpoint zero has been optimised, the PC will issue a reset command to the new node to ensure it is in a known state.

Assign a network address.

Since all new devices appear on the network as address zero, this address must be kept free. So, after the reset command the PC will assign the new device a unique network address for all further communication.

Request the device descriptors.

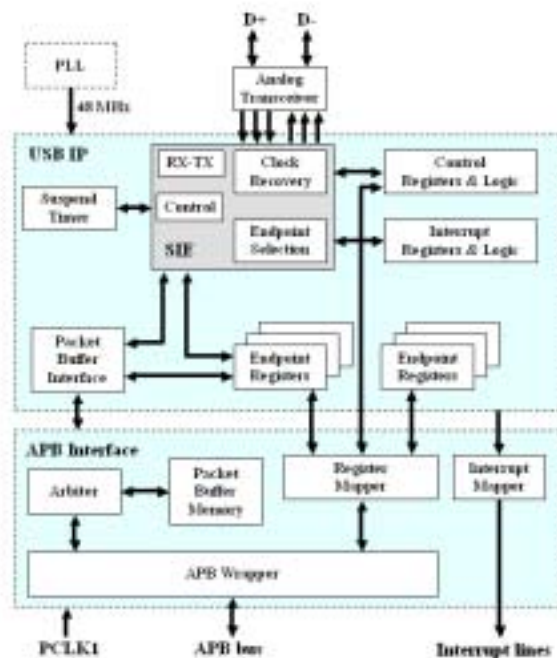
Once the new address has been assigned, the PC will request the full device descriptor followed by the configuration, interface and endpoint descriptors.

Once the PC has finished the enumeration process it will use the vendor and product IDs to assign a matching device driver. It can use further control transfers to select which configuration and interface it wants to use to communicate with the new device. Then the device is ready for use. The USB protocol supports eleven different control commands as summarised below. If a device does not support a particular command, it should return a STALL handshake to cancel the command and the PC will move on to its next control transfer.

Operation	Purpose
Get Status	Requests the status of an interface or endpoint
Clear Feature	Disables a feature on an interface or endpoint
Set Feature	Enables a feature on an interface or endpoint
Set Address	Sets the network address of a new device
Get Descriptor	Requests a specific descriptor
Set Descriptor	Adds or updates a specific device descriptor
Get Configuration	Requests the current device configuration
Set Configuration	Sets the desired device configuration
Get Interface	Requests the current device interface
Set Interface	Sets the desired device interface
Synch Frame	returns the frame number from a given endpoint

4.8.5.2 The STR71x USB Peripheral

Now that we have some understanding of the USB network and its protocols, we can have a look at the USB peripheral within the STR7. Like all the other STR7 communications peripherals, the USB peripheral is located on APB1.



The STR71x USB module is a USB 2.0 full speed module with integral transceiver and packet memory

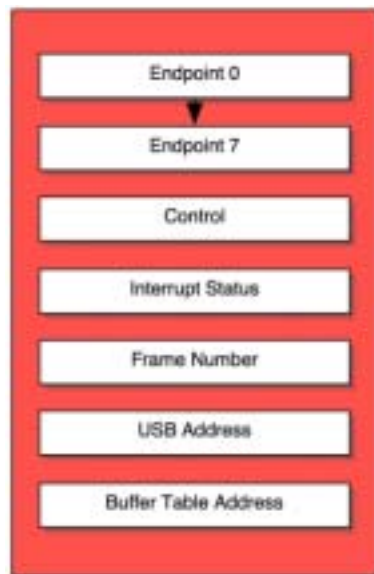
4.8.5.2.1 USB Clock

The USB peripheral contains the USB physical layer and the USB protocol engine, which supports up to eight endpoints which may be double buffered. The USB peripheral requires a 48MHz clock frequency. Since the APB1 clock is limited to a maximum of 33MHz, the USB peripheral clock is provided by a dedicated external crystal. If you are not using the HDLC peripheral, it is possible to derive the clock using the HDLC PLL, which allows you to have a low frequency external crystal in, rather than an EMC generating 48 MHz.



The USB module requires a dedicated 48MHz clock. If the HDLC peripheral is not being used, PLL2 can be used to generate this frequency from a lower value external crystal

The programmers' interface for the USB peripheral consists of a register for each endpoint, a control and interrupt status register, two registers that give the current frame number and the network address and finally a buffer table access register that is used to configure the endpoint buffers.



The USB module has a surprisingly simple interface which is primarily concerned with handling the endpoint buffers

Once the clock source has been configured, the USB peripheral must be enabled by switching on the analog physical layer by setting the PDWN bit in the control register. There is a delay while the transceiver is initialised and then the USB peripheral may be taken out of reset by clearing the FRES reset bit in the control register. Finally, any spurious interrupt flags must be cleared before the interrupt structure is enabled. At this point the USB peripheral is active, but any requests from the USB network will only receive a NAK handshake. Before the device is ready to communicate on the network, the endpoints must be configured.

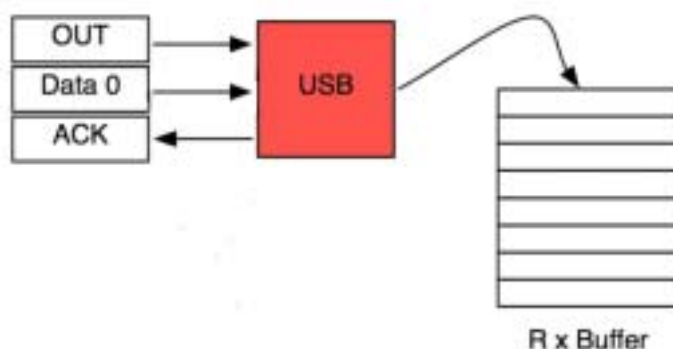
4.8.5.2.2 USB RAM

The USB peripheral contains 512 bytes of RAM that may be allocated to the enabled endpoints, to act as transmit and receive buffers. The buffer table access register acts as a pointer to the start address of a buffer description block which is held in part of the USB RAM. This table can be located on any 8 byte boundary within this memory.

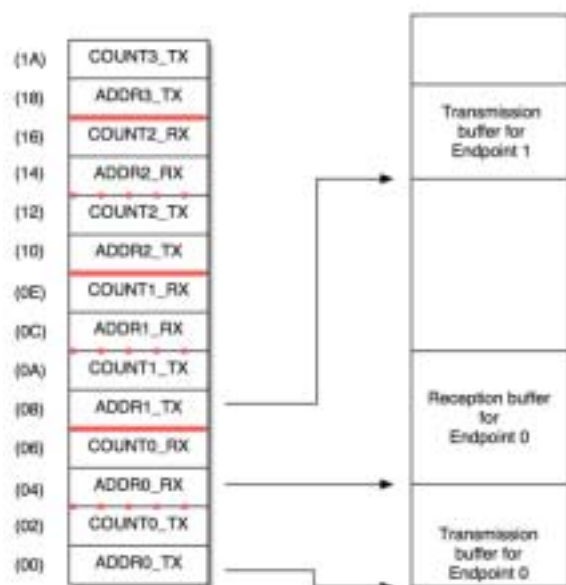


The buffer table address register holds the start address of the buffer description block which describes how the USB message RAM is partitioned between the enabled endpoint registers

The buffer description block is used to partition the USB RAM into a series of endpoint buffers for each of the enabled endpoints. The buffer description block contains the start address and the size of a transmit and receive buffer for each endpoint. Since only the control endpoint is bidirectional, all the other endpoints need only specify a single buffer for transmit or receive, depending on whether they are an IN or OUT endpoint.



Each endpoint register contains both configuration information (endpoint address, endpoint type) and data transfer flags used during communication

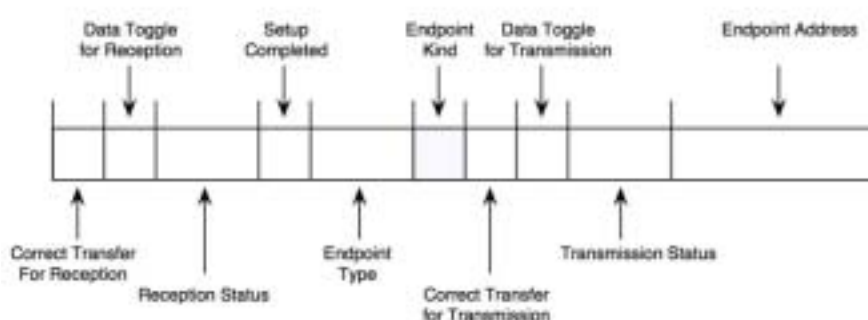


The buffer description block defines the start address and size of individual buffers allocated to each enabled endpoint within the message RAM

Once the endpoint buffers are configured, the USB interrupts must then be configured. The USB peripheral has two interrupt lines. The first is for high priority endpoints. Any endpoint enabled as a bulk or isochronous endpoint will be treated as a high priority endpoint. All other endpoints and other interrupt sources within the USB peripheral, are connected to the second low priority interrupt line. When an interrupt is generated, the interrupt status register contains flags, each interrupt source and an address field for the active endpoint if a USB transaction caused the interrupt.

4.8.5.2.3 Endpoint Registers

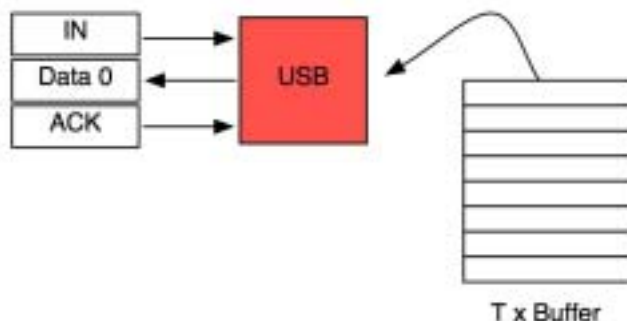
Finally, the endpoint register for each active register must be configured. In this register as a minimum we must set the address of the endpoint. The endpoint type must be set to match the transfer type that will communicate with this endpoint. Finally to enable the endpoint, the TX or RX status field must be set to 0x03.



An out packet will transfer data to the STR71x USB endpoint were it will be stored in the endpoint buffer. The endpoint is disabled (generates only NAK's) until the data is read and the endpoint is reactivated

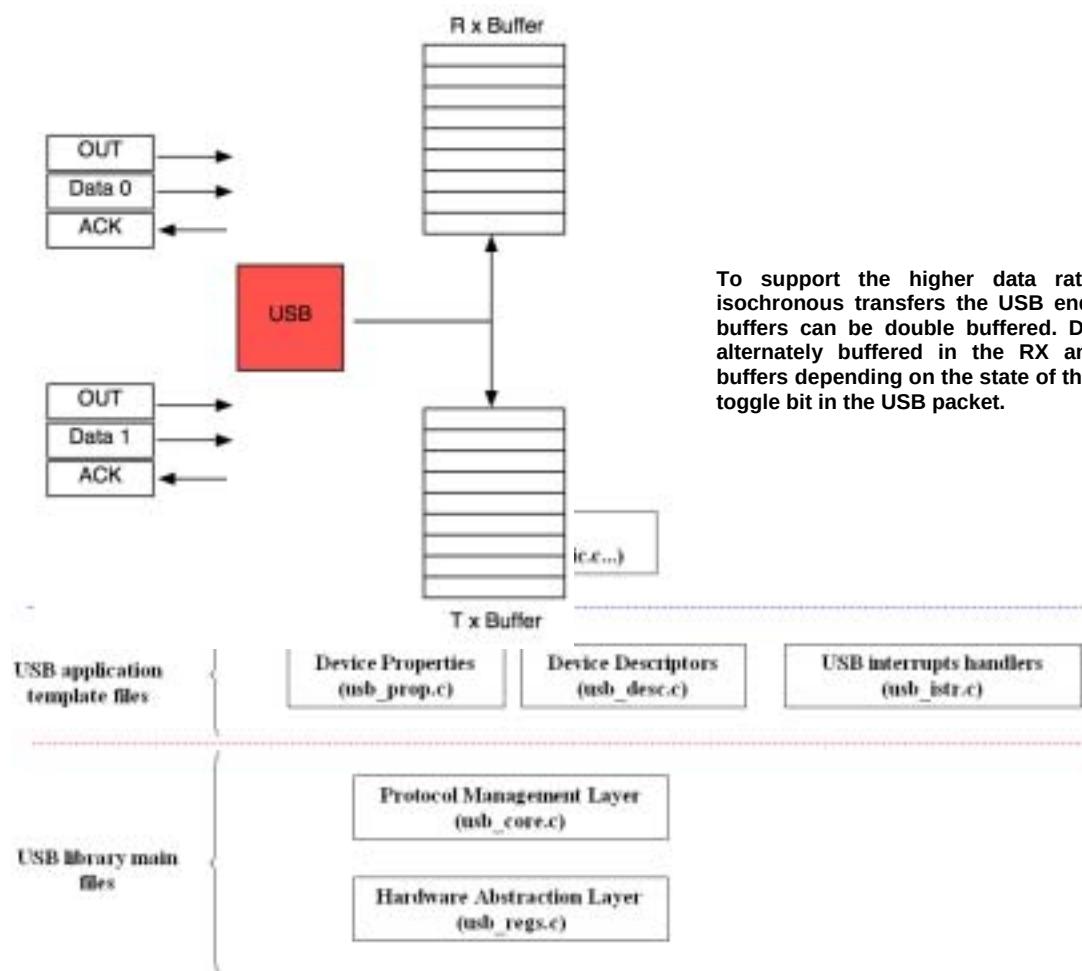
Now when a USB transaction occurs on an endpoint, the data packet will be copied into the endpoint receive buffer and an interrupt generated. When a fresh packet of data is received, the endpoint RX status field is set to 0x02, which means any further communication on this endpoint will receive a NAK handshake. When the device firmware responds to the interrupt, it must copy the data from the receive buffer and re-enable the endpoint by setting the status field back to 0x03.

Similarly for an IN transfer, the application firmware must fill the endpoint TX buffer with a data packet and then enable the endpoint via the TX status field. When the USB host requests the data, the packet will be transferred, and the endpoint disabled. The firmware must then refill and re enable the endpoint, ready for the next request.



When an IN request is made to an STR71x endpoint the contents of the endpoint buffer will be sent to the USB master. Further communication from the endpoint is disabled (generates only NAKs) until the buffer is refilled and the endpoint is reactivated.

This level of buffering is fine for control and interrupt transfers but isochronous transfers potentially have large data packets and bulk endpoints may have several transfers in the one frame. In order to support these higher bandwidth endpoints, the STR7 USB peripheral has a “double buffer” option to use both the transmit and receive buffers to store data for a unidirectional endpoint. If an endpoint is configured as bulk or isochronous the “endpoint kind” bit can be set to enable double buffering.



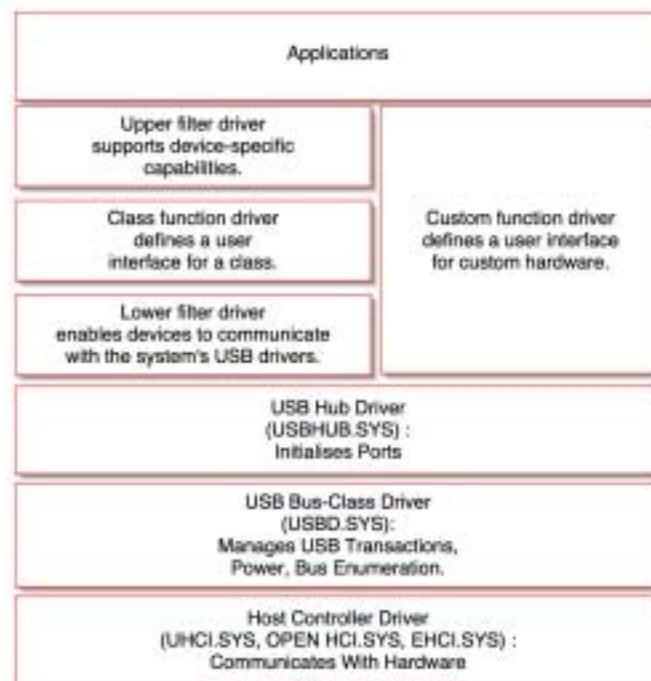
To support the higher data rates of isochronous transfers the USB endpoint buffers can be double buffered. Data is alternately buffered in the RX and TX buffers depending on the state of the data toggle bit in the USB packet.

ts will be
rdpoint is
ering can
l.

ements a

4.8.5.2.4 PC Device Drivers and Client Software

In addition to understanding the USB network and the STR7 peripheral, we also need to know how to access the USB peripheral from some application software on the PC. The Windows operating system (98, 2000, Millenium and XP) supports USB with a built-in USB stack and some generic driver support.



The Windows driver model allows your application software to use a class driver. This is a USB device driver written to support a generic class of devices such as audio devices, printers, mass storage devices, or you may provide your own. If you want to use your own driver this means writing a kernel mode device driver. If you know how to do this then fine! If not, it would involve learning a lot about the Windows operating system. There are a number of ready-made general purpose device drivers that require no development work other than filling in your Vendor and Product ID's. You can download an evaluation version of such a driver from www.thesycon.com.

If your USB peripheral can take advantage of one of the Windows class drivers then you can save yourself the trouble and expense of developing your own driver. The two most interesting class drivers in Windows are the "Human interface device" (HID) class and the "Mass Storage" class. As its name implies, the HID class is the driver used by USB keyboards, mice, joysticks etc.. However you can also make use of the HID driver to send and receive basic IO data such as front panel data (reading switches, illuminating LEDs and so on), or communicating with remote sensors. The second class is the Mass Storage class which allows the USB device to appear as a removable drive. Thus the HID driver gives a basic bidirectional connection between the PC and the USB device and the Mass storage driver allows the transfer of large amounts of data. An understanding of both of these drivers will cover a large range of applications.

You can make a USB device into a HID class device by setting the class code in the interface descriptor to three as shown below

Interface	Descriptor
09h	Descriptor length
04h	Descriptor type
00	Number of interface
00	Alternate setting
01	Number of endpoints
03	Class code
00	Subclass code
00	protocol code
00	Index of string

When the PC discovers a HID device it will start to request a further set of descriptors called report descriptors. The report descriptors define the structure of the data that is to be transferred for this HID device. The USB Implementers Forum maintain a set of HID usage tables which define data structures from a number of common devices. However it is possible to define a vendor-specific structure that matches your requirements.

The report structure shown below defines a vendor-specific report descriptor that configures the HID driver to transfer two 8-bit signed bytes IN to the PC and two 8-bit signed bytes OUT of the PC. The IN transfer must be done over an interrupt pipe and the OUT transfer will be made over the control channel by default, or it can use an OUT interrupt pipe if one is available. The maximum transfer rate for a HID based device would be that of an interrupt pipe or a 64-byte packet every frame, or 64Kbytes/sec - roughly five times the speed of a serial port.

HID Report Descriptor Table			
06h A0h FFh	Usage page	(Vendor Defined)	
09h A5h	Usage	(Vendor Defined)	
A1h 01h	Collection	(Application)	
09h 06h	Usage	(Vendor defined)	
Input Report			
09h A7h	Usage	(Vendor defined)	
15h 80h	Logical Minimum (-128)		
25 7F	Logical Maximum(127)		
75 08	Report size	(8 bits)	
95 02	Report Count	(2 fields)	
81 02	Input	(Data variable absolute)	
Output Report			
09h A7h	Usage	(Vendor defined)	
15h 80h	Logical Minimum (-128)		
25 7F	Logical Maximum(127)		
75 08	Report size	(8 bits)	
95 02	Report Count	(2 fields)	
91 02	Output	(data variable absolute)	
C0h	End Collection		

Once the USB device has enumerated as a HID device we need to be able to communicate to the USB network from your application software. This is done by the WIN32 API that allows you to access many of the functions within the Windows operating system. It is possible to use any development tool that can access the API such as Visual C++ or Visual Basic. If you are using visual C++ you need to order the Windows Driver Development Kit DDK from the Microsoft website which is free but costs \$25 for shipping. The DDK includes .lib and .h files that are necessary for accessing the API functions needed to control the HID driver.

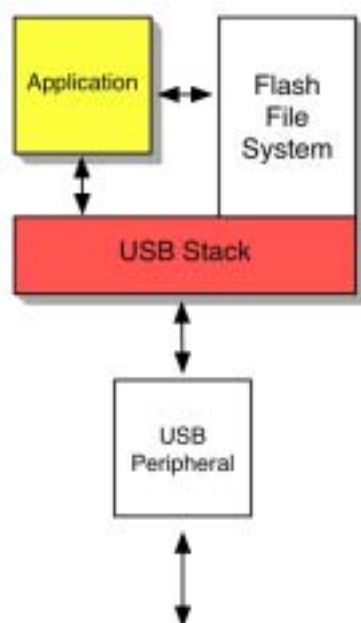
API Function	DLL	Purpose
HidD_GetHidGuid	hid.dll	Obtain the GUID for the HID class
SetupDiGetClassDevs	setupapi.dll	Return a device information set containing all of the devices in a specified class
SetupDiEnumDeviceInterfaces	setupapi.dll	Return information about a device in the device information set
SetupDiGetClassDevs	setupapi.dll	The constant Configuration (02h)
SetupDiDestroyInfoList	setupapi.dll	Free resources used by SetupDiGetClassDevs
CreateFile	kernel32.dll	Open communications with a device
HidD_GetAttributes	hid.dll	Return A Vendor ID, Product ID, and Version Number
HidD_GetPreparedData	hid.dll	Return a handle to a buffer with information about the device's capabilities
HidP_GetCaps	hid.dll	Return a structure describing the device's capabilities
Hid_FreePreparedData	hid.dll	Free resources used by HidD_GetPreparedData
WriteFile	kernel32.dll	Send an Output report to the device
ReadFile	kernel32.dll	Read an Input report from the device
HidD_SetFeature	hid.dll	Send a Feature report to the device
HidD_GetFeature	hid.dll	Read a Feature report from the device
CloseHandle	kernel32.dll	Free resources used by CreateFile

A full HID programming tutorial is given in the book “USB Complete” by Jan Axelson. This is recommended reading if you are just starting out with USB. As a brief outline, the application software has to find out how many HID drivers are active within Windows then interrogate each one until it discovers the driver associated with your USB peripheral. This is done by locating the driver with your vendor and product ID. Once the driver is found we can read its capabilities and this will give the report structure to the application. Finally we can use read file and write file to transfer our application data. A complete client example is included with the MCBSTR7 evaluation software and all the necessary API function calls are encapsulated into six C functions that you may easily use in your own application.

Exercise 21: USB HID

This exercise configures the USB peripheral as a HID device and communicates with a dedicated PC client via the windows HID device driver.

The second driver of interest is the Mass Storage Class driver. This allows our USB device to appear as a remote drive so that we can easily upload and download files to it. The Keil ARTX operating system comes with Mass storage support that integrates with the FLASH file system, also included in the ARTX operating system. You do not need to modify the Mass storage firmware. This can be treated as a black box. You just need to manage file system from your application firmware. The file system can make use of the 16K of data flash provided by the STR7 or you can add a multimedia card to the SPI port for mega bytes of storage.



The Keil ARTX real time executive developed for small ARM based microcontrollers contains “middleware” stacks that accelerate the development of complex applications.

4.9 Summary

The STR71x 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 lists additional tools and resources that assist development of an STR71x-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 in this chapter 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 Exercise 0: Installing the software

Once you have inserted the starterkit 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.

5.3 Setting up the Hardware

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



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. Finally the USB cable should be plugged into the Tantino 7-9 to connect the JTAG debugger to the PC. Finally an RS232 serial cable should be added to connect the socket labelled "RS232 -A" to a PC serial port.

5.4 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 interrupt structure. The GCC compiler is an open source C/C++ compiler, which will generate 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 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". These will be discussed in chapter XXX.

5.5 Exercise 1 Setting up your first project

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

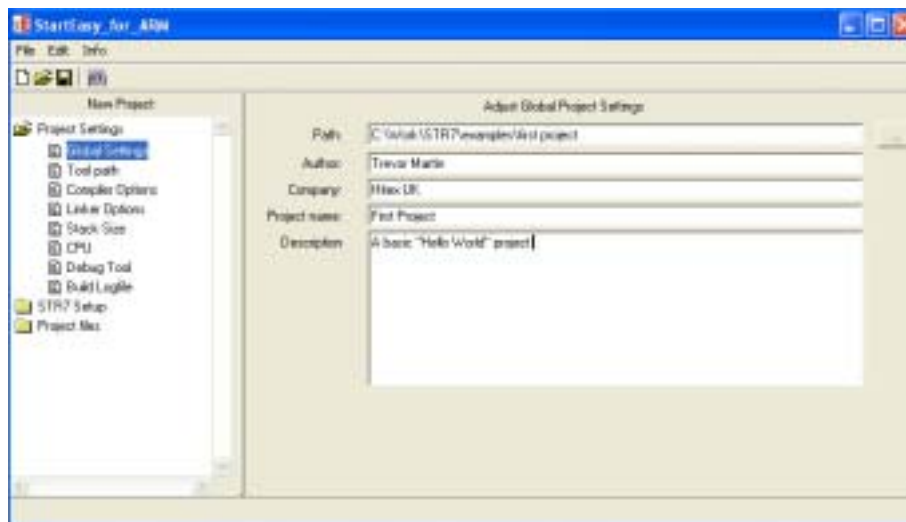
5.5.1 StartEasy

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.

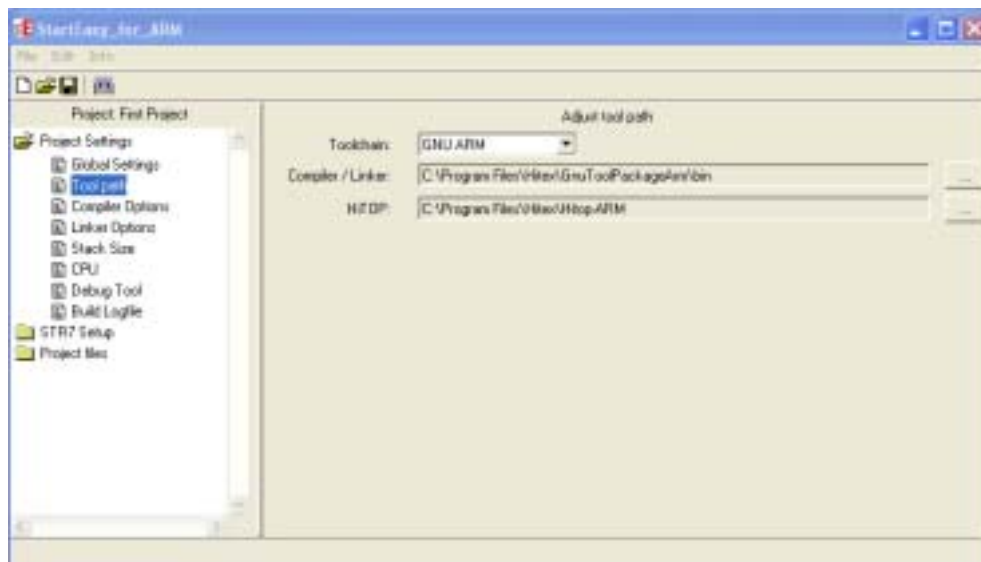
To start StartEasy, select and double-click the StartEasy for ARM icon:



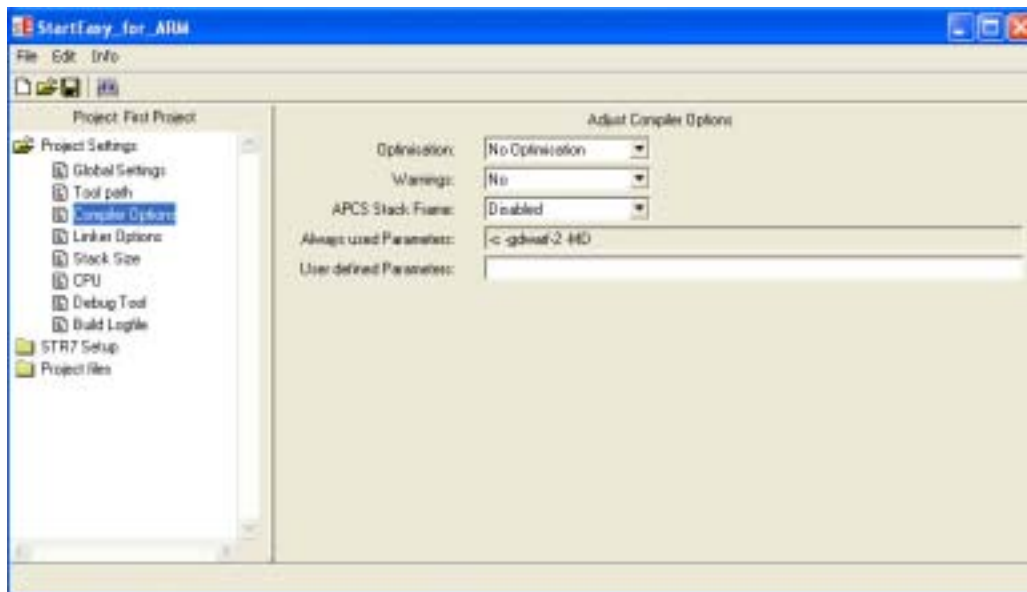
This will open the project settings window, which should be filled-in with the basic project information, as shown below. The project name will also be the name of the debug file produced by the GCC linker.



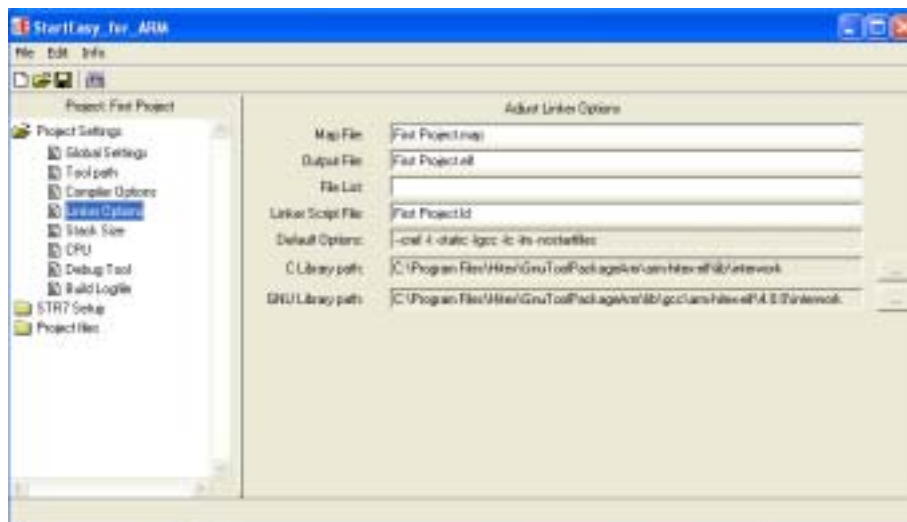
Next, click on the project settings folder to expand the project settings options and select tool path. There are a number of different installation packages for the GCC compiler but the package installed here requires the tool path to be setup as shown below:



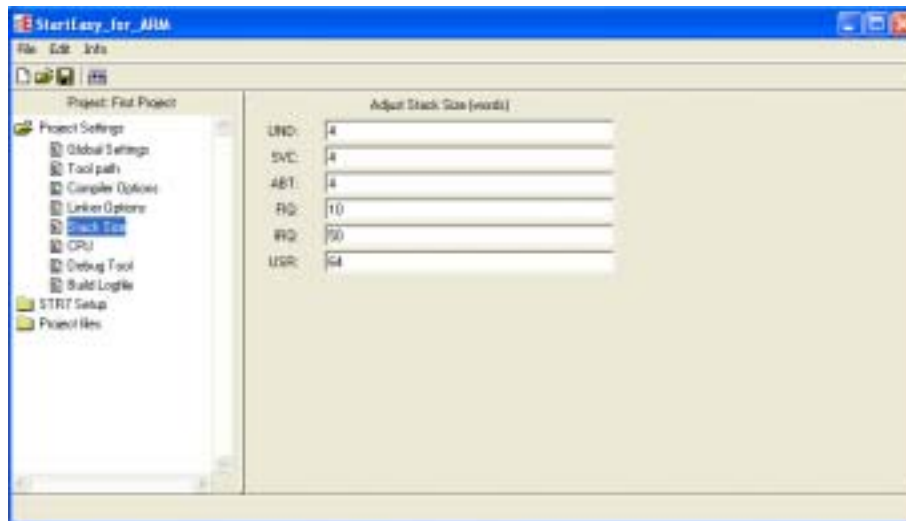
For the current example the compiler options can be left on their default setting but these will be discussed in more detail later on in this chapter.



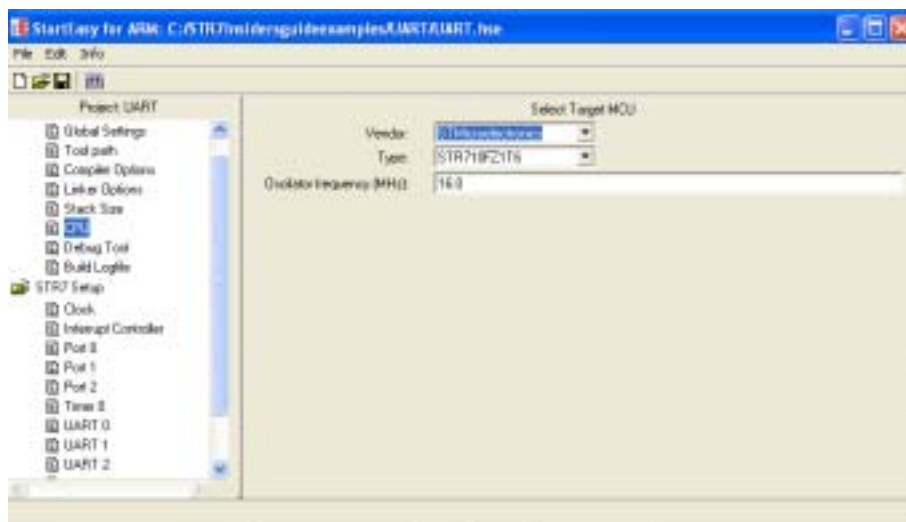
Next open the linker options dialogue. The project output files and the linker script file default to the name of your project. Again, because the GCC compiler can be installed to various locations, check that the linker library paths are configured to match the current installation.



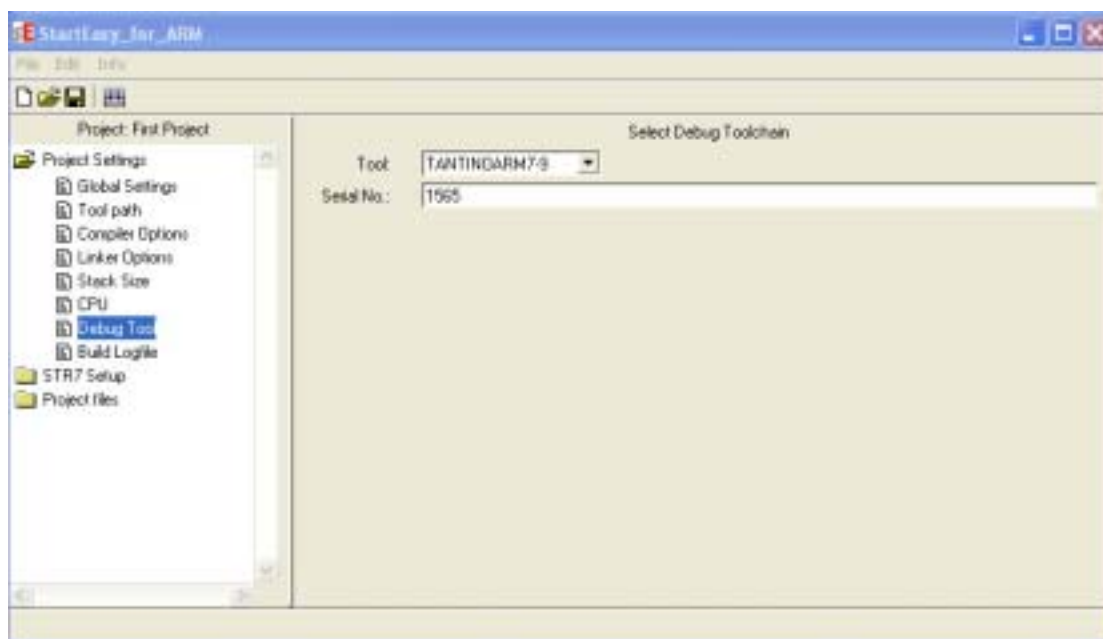
Once you have checked the linker options, open the "Stack Size" dialogue. The stack size window allows you to define how many bytes are allocated to each of the stacks available in the different operating modes of the ARM7. The stack configuration is done in the startup code. This is held in an Assembler 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.



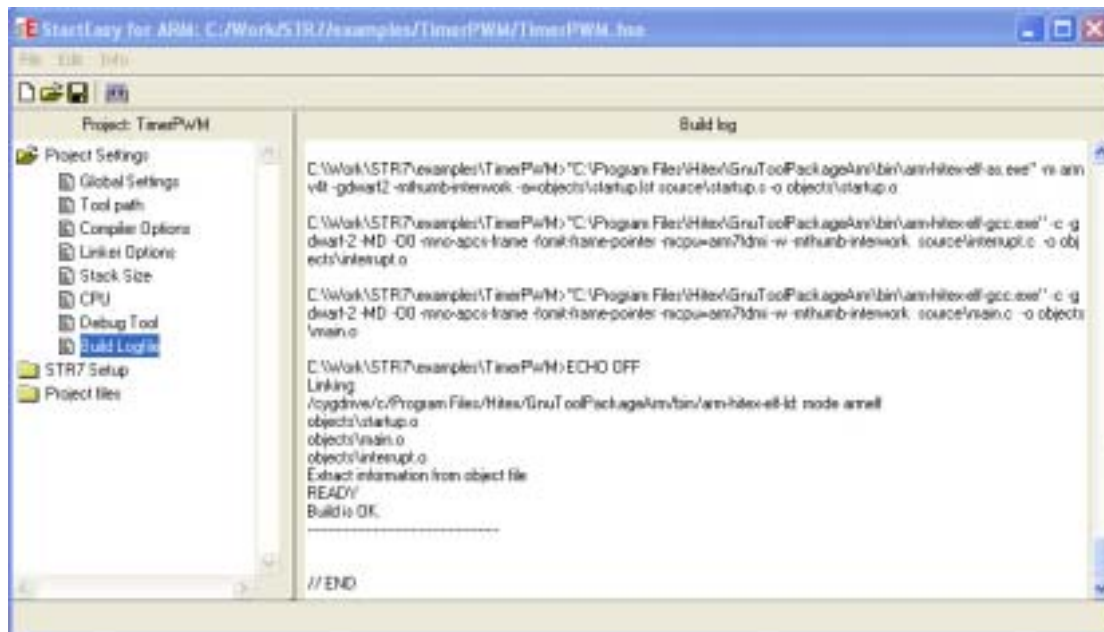
Once you have examined the stack window, open the CPU dialogue. In this window you must select the vendor to be “ST Microelectronics” and the Type to be the “STR710FZ1T6”. You must also set the Oscillator frequency to be 16.0MHz



Now open the Debug dialogue. 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.



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 "Hello World" project, we are going to send a simple text message to a terminal monitor on the PC, via UART0. To do this, we must configure the General Purpose IO and the UART 0 itself. First open the PORT 0 dialogue and select tab for pins 8 to 15. In the dialogue box for pin 8 and pin 9, select "Alternate function OD". This will generate initialising code which will switch the pins from being GPIO pins to their alternate functions as UART 0 Rx and TX pins.



Next select the UART0 dialogue and switch the state from “Disable” to “Normal”. This will generate code to initialise the UART to run at 9600 Baud, 8 bits, no parity, one stop bit.



Next we also want to be able to control the LEDs on the target board. These are connected to PORT2 pins 8 to 15, so select the PORT2 menu and switch these pins to output PP (Push pull)

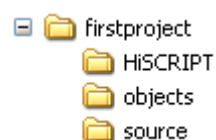
This finishes configuring the project and the STR7 for our simple “Hello World” application. To generate the project, select edit and example applications from the menu toolbar.



From the example drop down box select the “Hello World” example and enter a simple message in the text box. 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.

5.5.2 Project structure

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

Startup.s	These files contain the Assembler startup code which configures the processor before you enter your C code
Startup_stmicelectronics.s	
Startup_generic.s	
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
STR710.h	Include file to define the STR710 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 */

        printf(HELLO_STRING);
        for(i=0; i!=100000; 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.

```
#ifndef _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 */

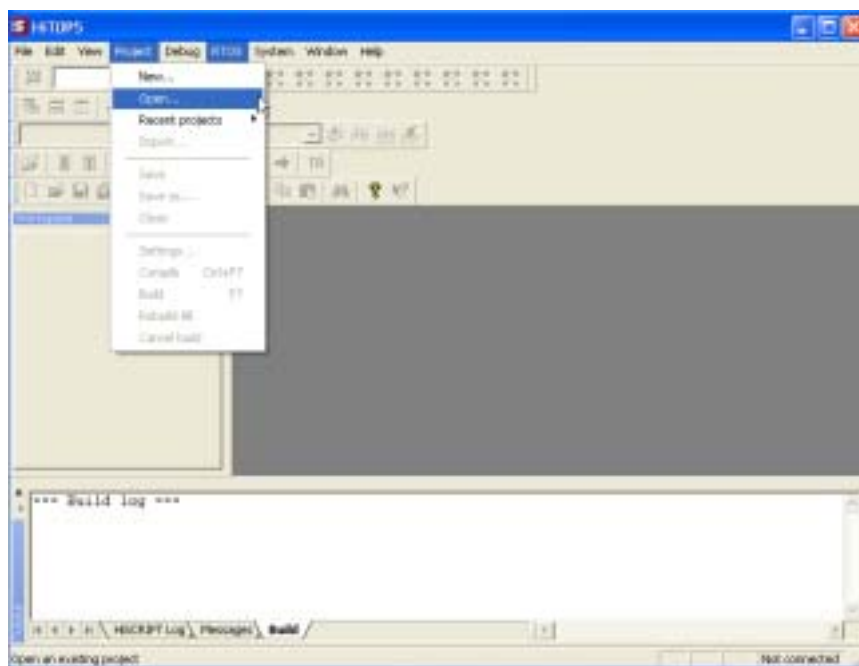
/* BEGIN USER CODE GLOBAL FUNCTIONS */
extern void UserFunction(void);
/* END USER CODE GLOBAL FUNCTIONS */

#endif
```

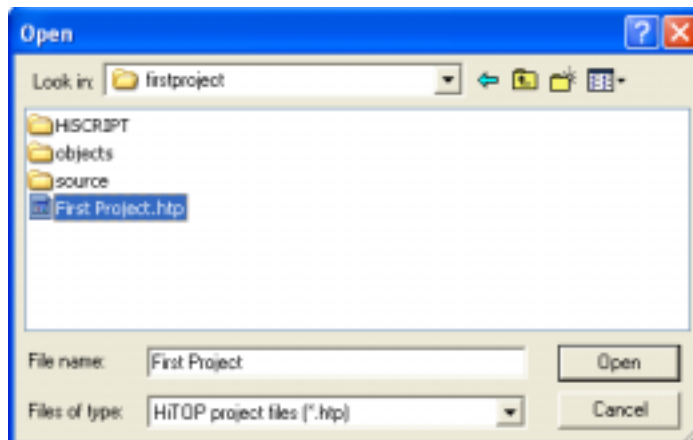
The `#ifdef` symbol is declared once in the home module. When the project is built the declarations are made when the home module is compiled and will be declared as externals in any other module where the include file is used.

5.5.3 HiTOP Debugger

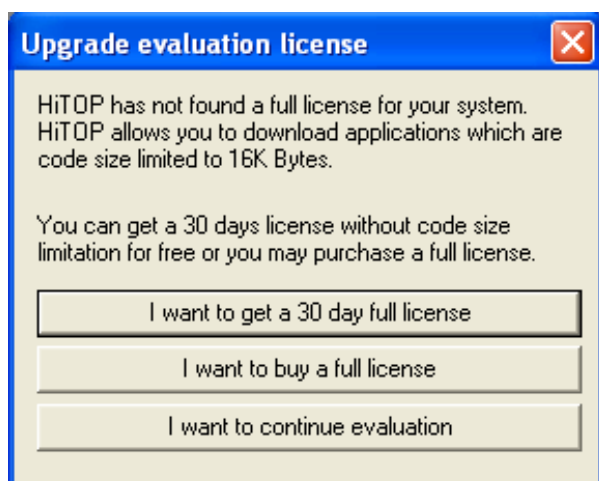
Once you have generated the example project, start HiTOP by selecting and double-clicking on the HiTOP icon:



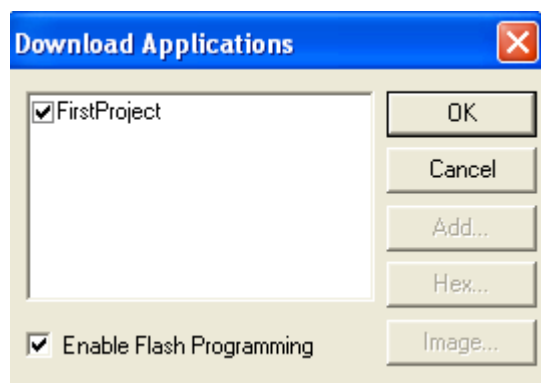
When the HiTOP program has launched, select “project\open project” and select the HiTOP project in the Hello World project directory. The project extension is “.htp”. When you do this, make sure the evaluation board and the Tantino7-9 are connected.



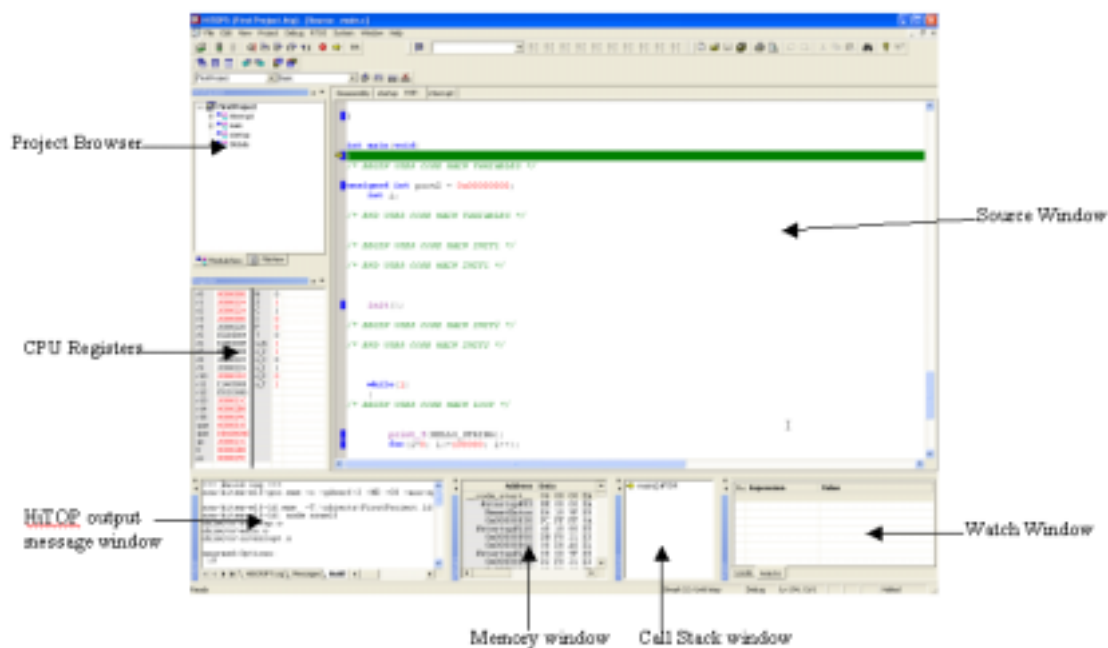
When you open the project, a reminder screen will be displayed. Click the “I want to continue evaluation” button. You also have the option to purchase a full licence or get a 30 day trail licence that has no code limitations.



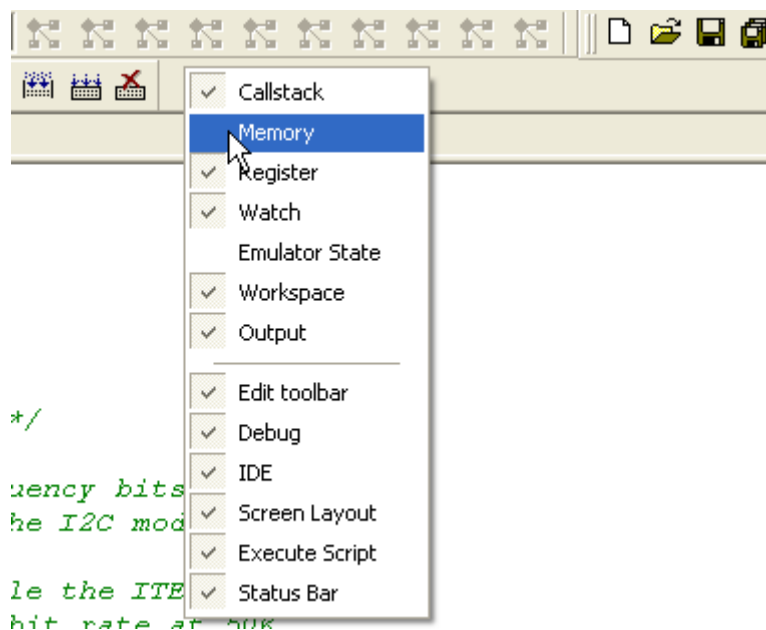
Once you have passed the reminder screen, HiTOP will start and ask if you want to download your application to the STR7 FLASH memory. Click OK to begin the FLASH download. If an error occurs at this point it will most likely be that the boot jumpers are incorrectly set on the evaluation board.



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

5.5.3.1 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. Now add the following lines of code to the existing source - first declare a new variable

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

unsigned int port2 = 0x00000000;
    int i;

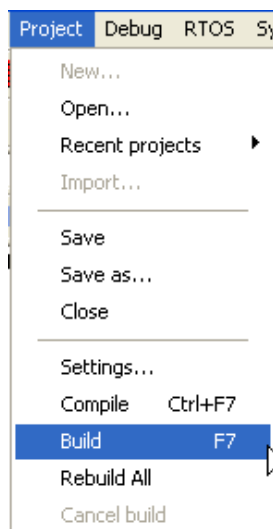
/* END USER CODE MAIN VARIABLES */
```

and add code to toggle the LED's

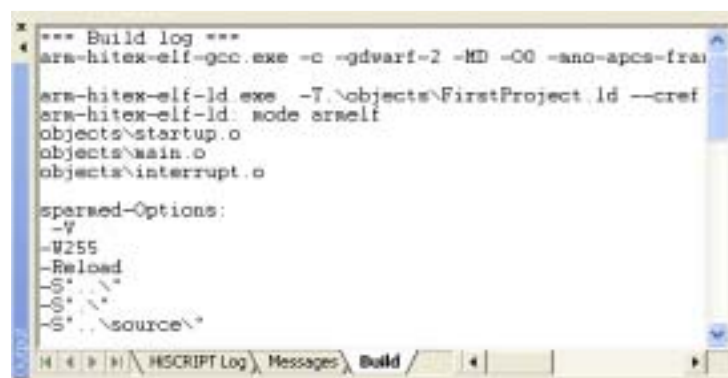
```
printf(HELLO_STRING);
for(i=0; i!=100000; i++);

port2 = port2^0x0000FF00;
IOPORT2_PD = port2;
```

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

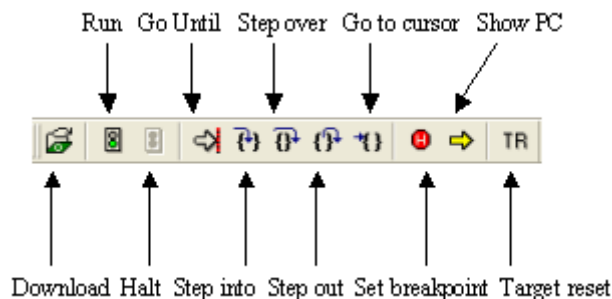


After the build has finished, the new code will be downloaded into the evaluation board

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.

5.5.3.2 Run Control

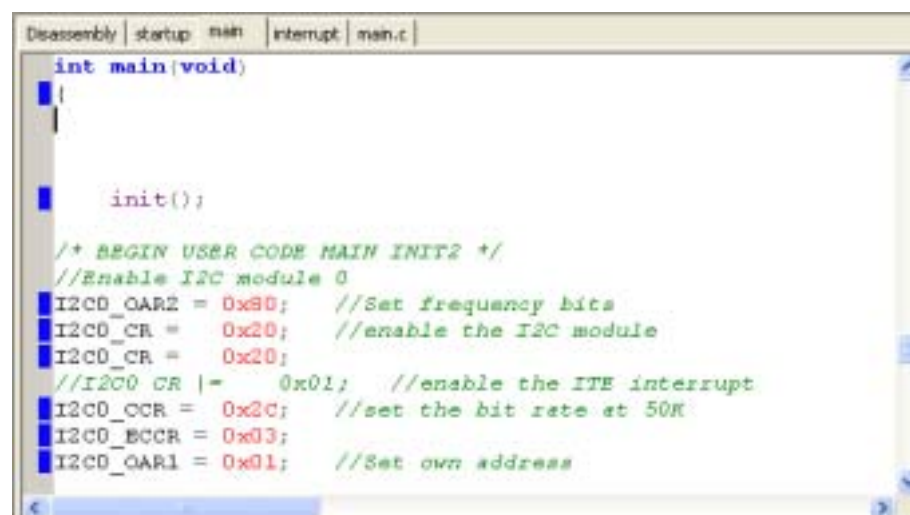
Once the project is loaded, the STR7 is reset and the program counter is forced to the reset vector. From here it is possible to execute code at full speed or single step line-by-line. The debug toolbar has specific buttons to control execution of code on the ARM7 CPU



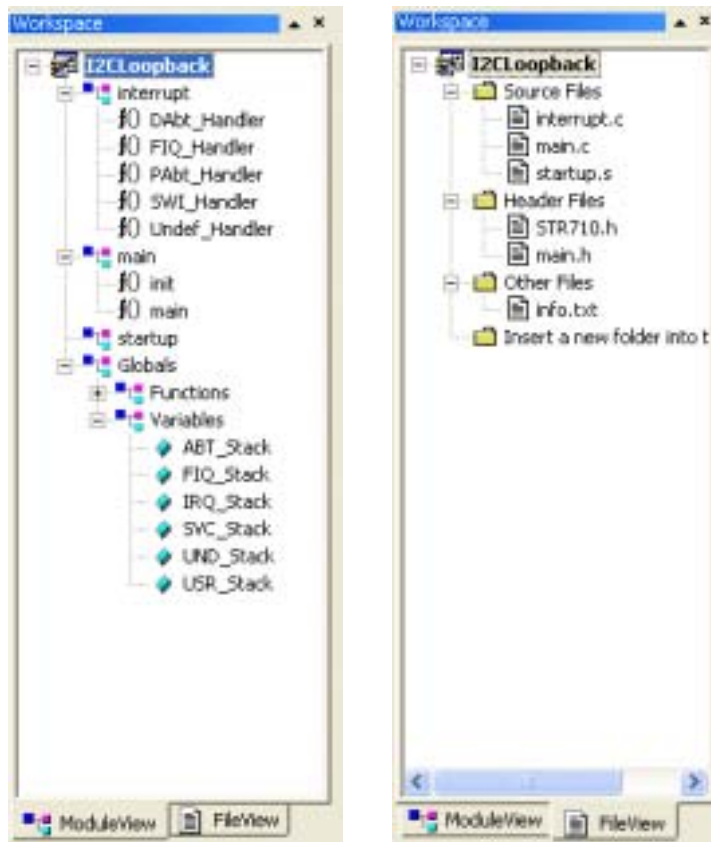
These functions can also be accessed via the debug menu, which also displays the keyboard shortcuts. It is worth learning the keyboard shortcuts as these are the fastest way to control the execution of your code.



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



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:

```

init();

/* BEGIN USER CODE MAIN INIT2 */

/* END USER CODE MAIN INIT2 */

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

print f(HELLO STRING);
for(i=0; i!=100000; i++);

/* END USER CODE MAIN LOOP */

```

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.

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.

```
void put_char ( char character )
{
    unsigned char TransmitState;

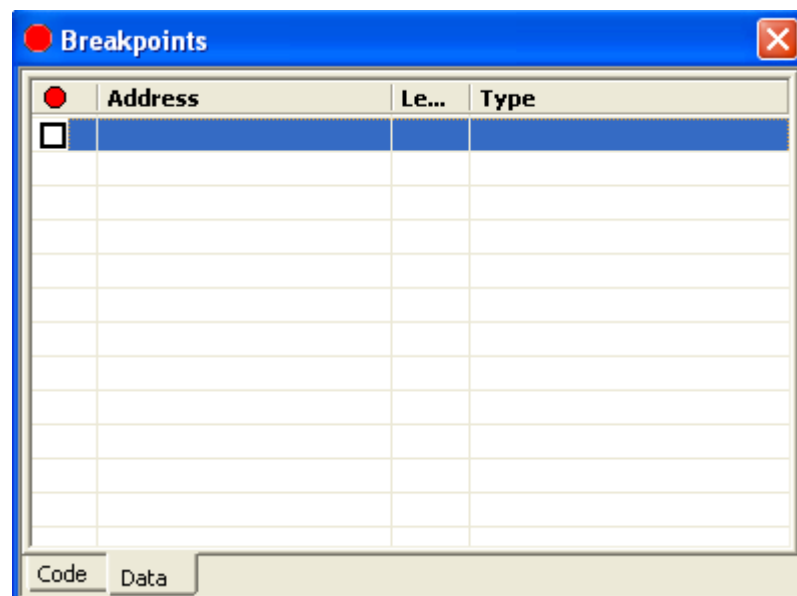
    /* check if transmit buffer is free */
    while ( UART0_SR & UART_TXFULL )
    { ; }
    /* write character */
    UART0_TxBUFR = (unsigned char)character;
}
```

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.

```
void put_char ( char character )
{
    unsigned char TransmitState;

    /* check if transmit buffer is free */
    while ( UART0_SR & UART_TXFULL )
    { ; }
    /* write character */
    UART0_TxBUFR = (unsigned char)character;
}
```

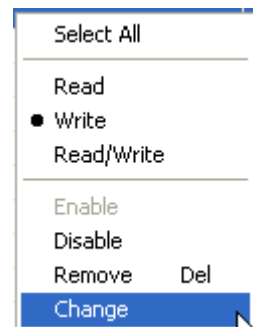
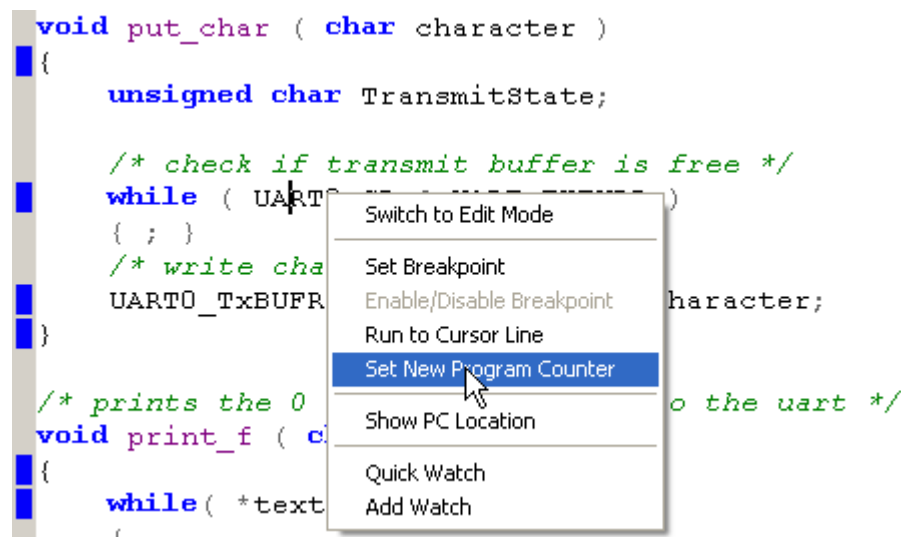
It is also possible to set breakpoints on data variables. This allows you to halt code when a variable is read or written too. Open the breakpoint menu under `debug\breakpoint` and select the data tab:



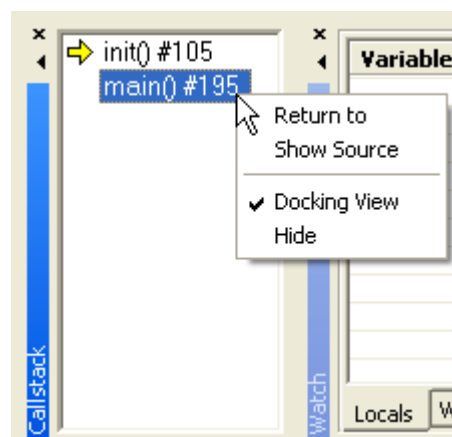
Now locate the variable you want to break on, either in the source window or the project browser, then drag-and-drop it to the breakpoint window. By default, the break condition is on a write to the variable. If you select the local options this can be changed to read or read/write. The change option 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, 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 effected so you must use this option with care.



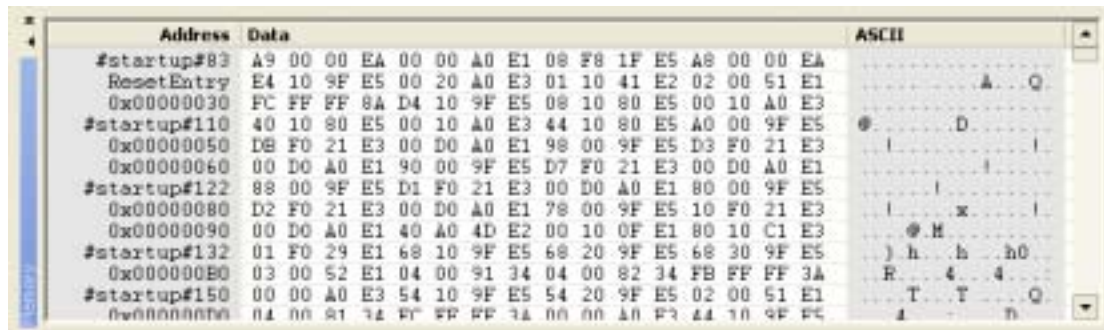
The callstack window displays the calling hierarchy of the functions pushed onto the stack. If you double-click on a function name, the source will be displayed at the point that the program will return to. The local options displayed by a right-click also allow you to run the program up to a selected 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

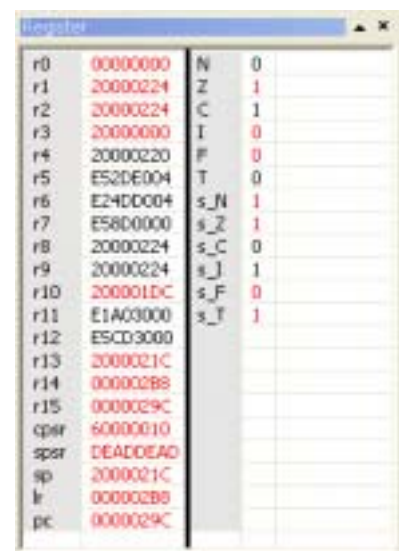
5.5.3.3 Viewing data

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

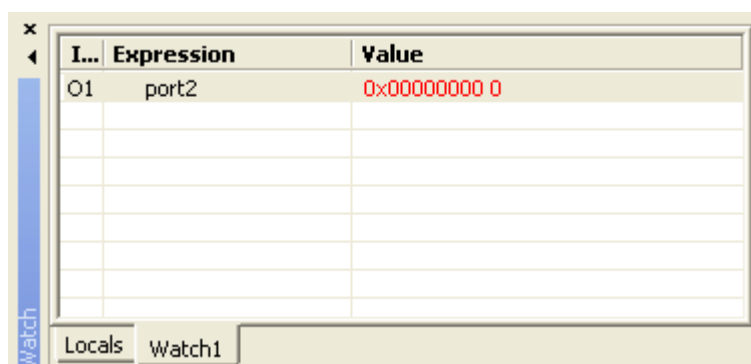
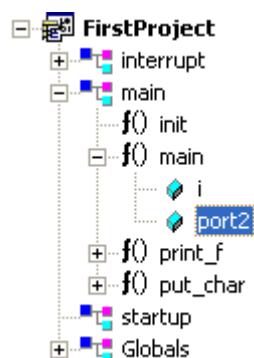


You can set the address range of the window by double-clicking on an entry in the address column and entering an absolute address or a symbolic name. The contents of memory locations can be changed by overtyping the values in the data or ASCII window. The more advanced options are available by right-clicking 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”.

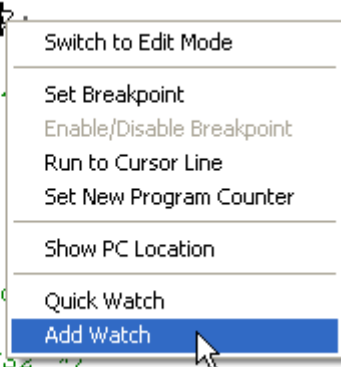
The Watch window supports all C and C++ data types including complex objects such as classes, arrays, structures and unions.

```
IOPORT2_PD = port2;
```

```
USER CODE MAIN LOOP
```

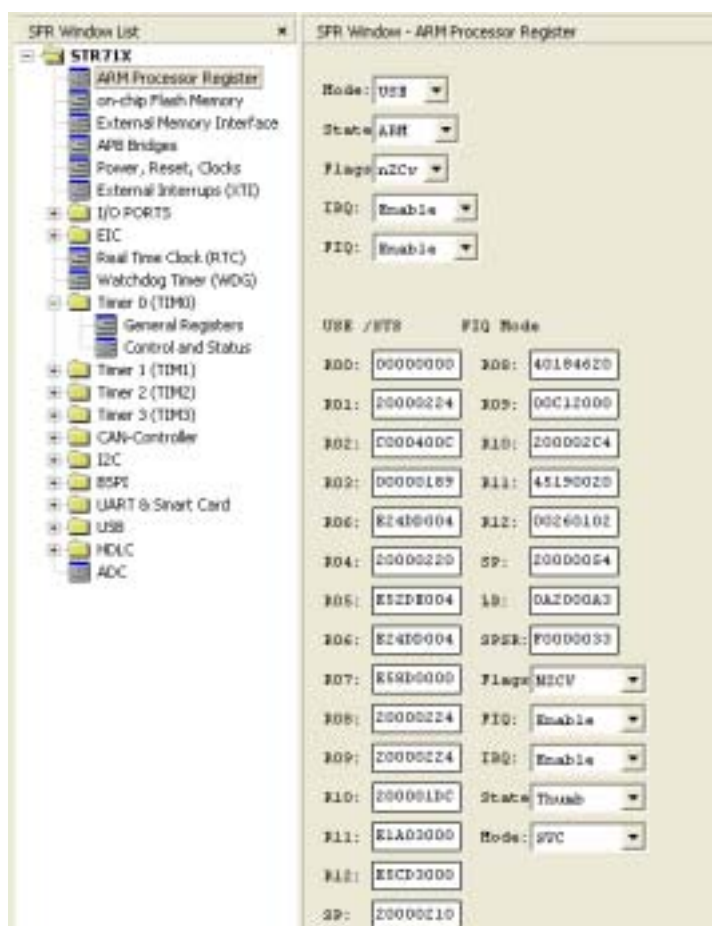
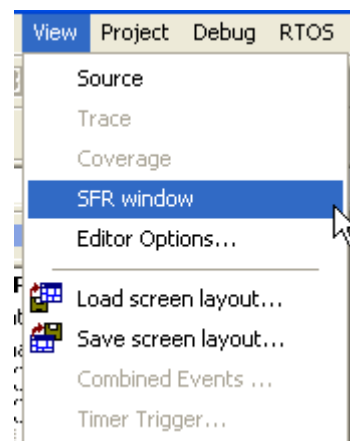
```
USER CODE FUNCTION
```

```
USER CODE FUNCTION2
```



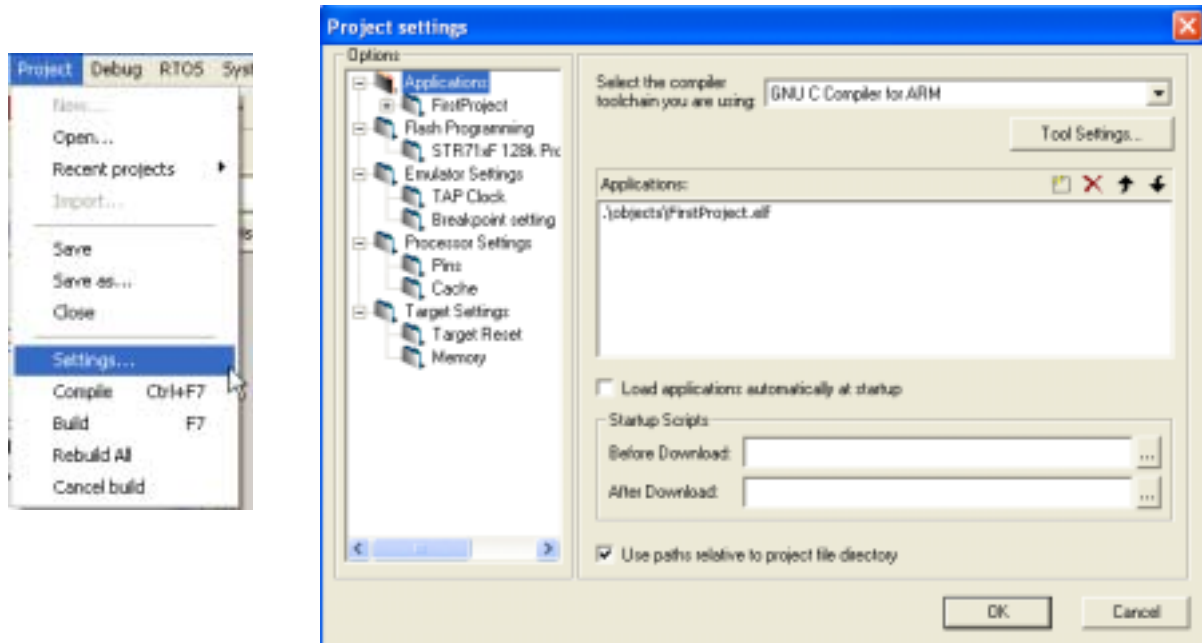
There are also dedicated windows for each of the STR7 peripherals. These can be accessed with the view\SFR window on the main toolbar

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



5.5.3.4 HiTOP Project Settings

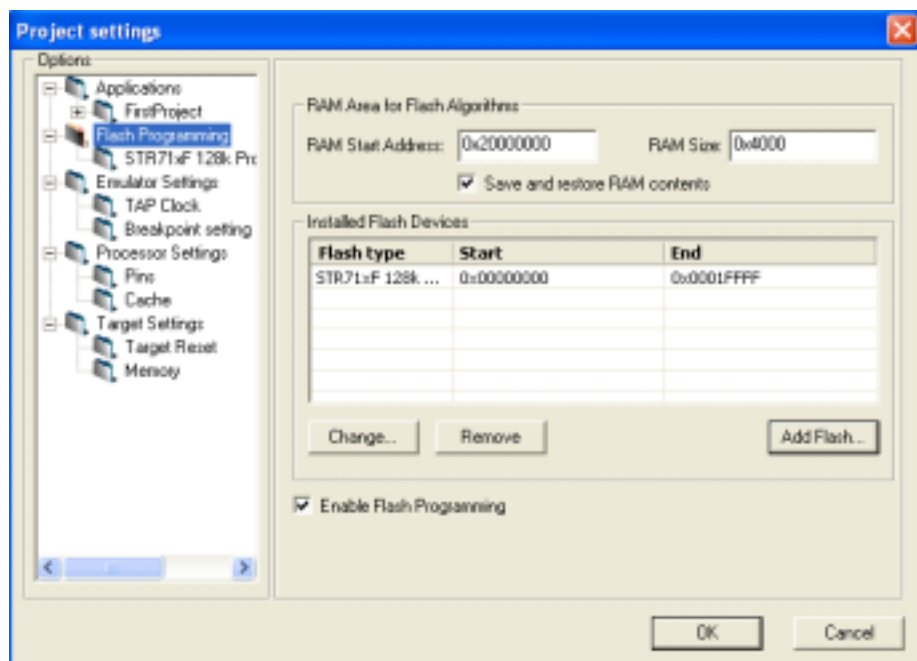
Once you are familiar with the basic features of StartEasy and HiTOP, you may want to modify the project settings to begin work on your own project. This can be done by re-running StartEasy or directly within HiTOP. You can change the project settings within HiTOP via the Project\Settings menu.



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.

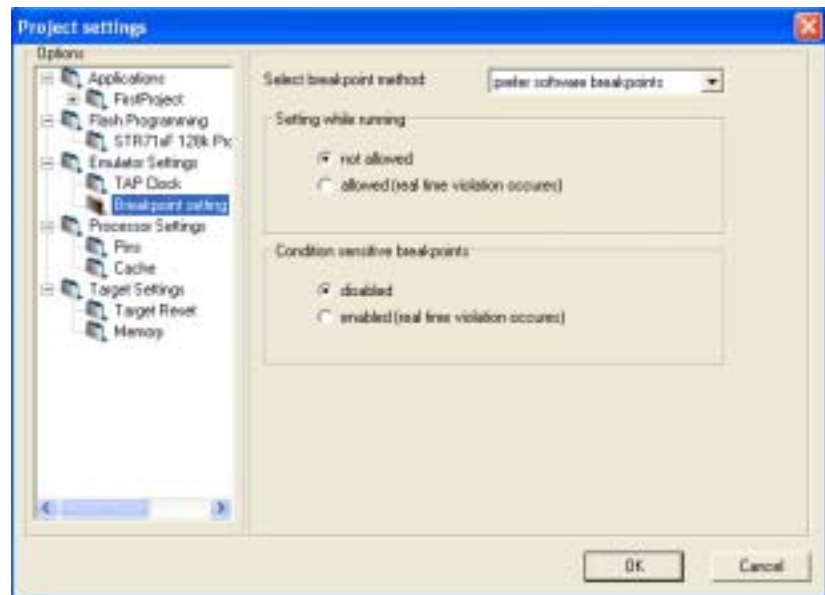
Once you have selected the project file and compiler tool, the FLASH programming section allows you to select the programming algorithm for the FLASH device you are using. This supports the STR7 on-chip FLASH memory and a wide range of FLASH memory chips, if you are using external FLASH memory. In this 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.



5.5.3.5 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. 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

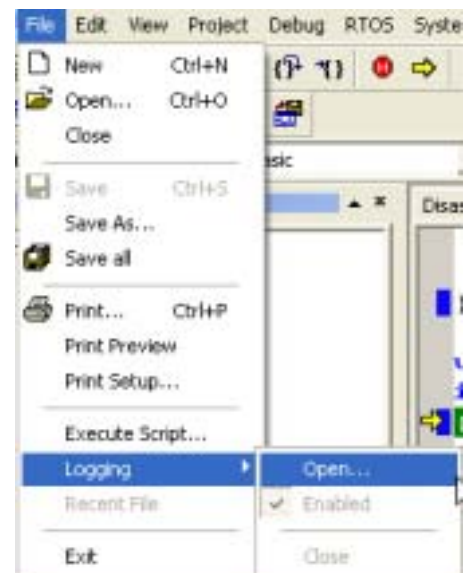


real-time application. The condition-sensitive breakpoint option allows you to set a breakpoint on an ARM instruction that is conditionally executed. If this option is enabled, the code will only halt when the instruction's condition codes match the CPSR. Again this is extremely useful when debugging ARM code. The remaining processor and target options are specific to the STR7 and will not generally need to be changed.

5.5.3.6 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 over write the existing file.

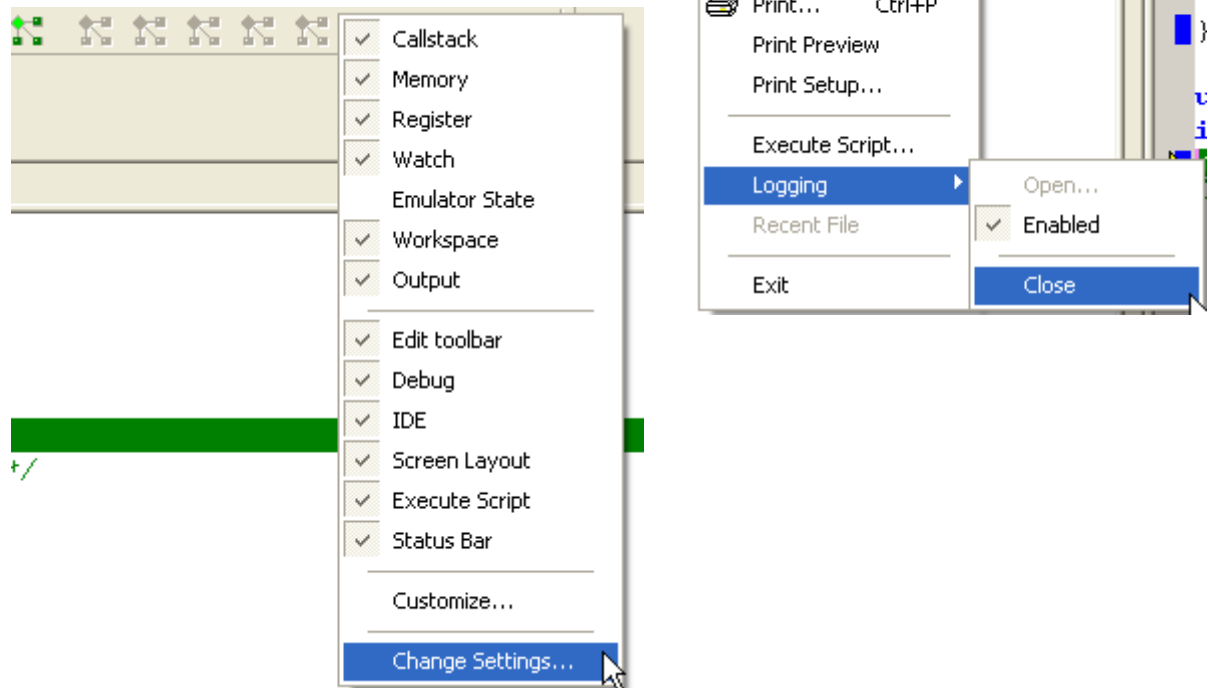


Now select OK and any HiTOP instructions you do within the debugger will be saved as HiSCRIPT commands. Once you have enabled logging, perform a target reset and a go until main(). Again, select file\logging and close option to stop the command logging.

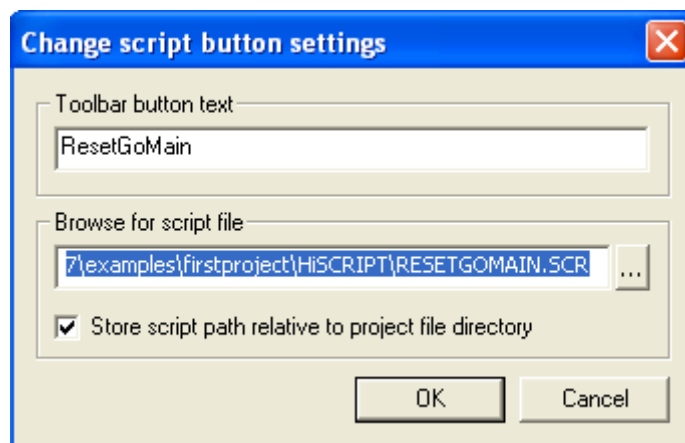
This will generate a HiSCRIPT file that will replay your instructions. The actual HiSCRIPT file contains some extra instructions but it can be edited down to the following minimal instructions:

```
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:

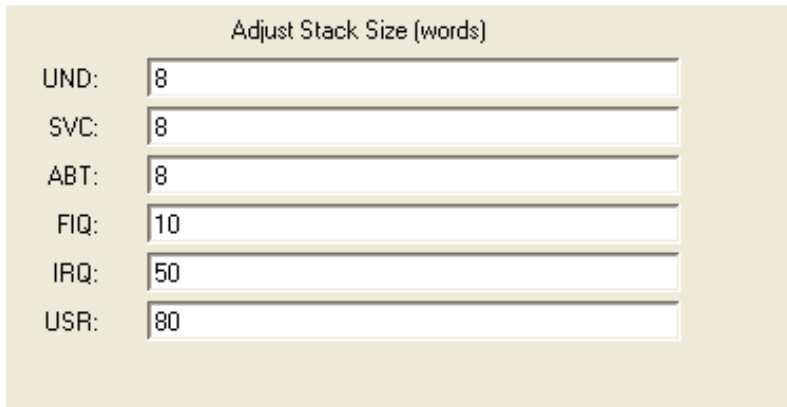


This is an important tutorial. You should explore all the features of HiTOP and StartEasy so that you can easily define, edit and debug an application program

5.6 Exercise 2: Startup Code

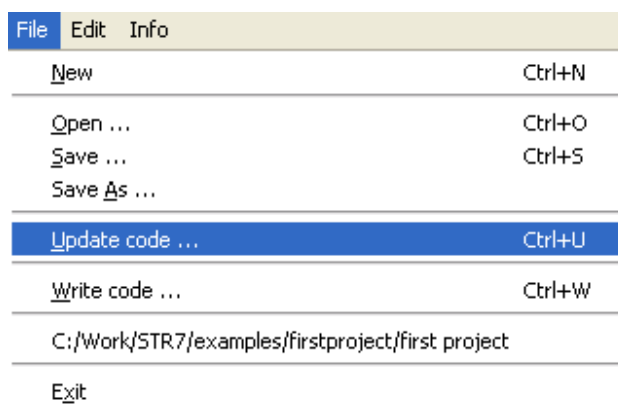
In this exercise we will configure the compiler startup code to configure the stack for each operating mode of the ARM7. We will also ensure that the interrupts are switched on and that our program is correctly located on the interrupt vector.

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

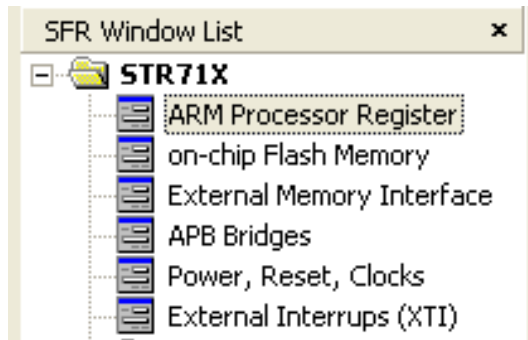
3. Open the project settings debug tool dialog and enter your Tantino serial number
4. Select file\update code and regenerate the project.



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.



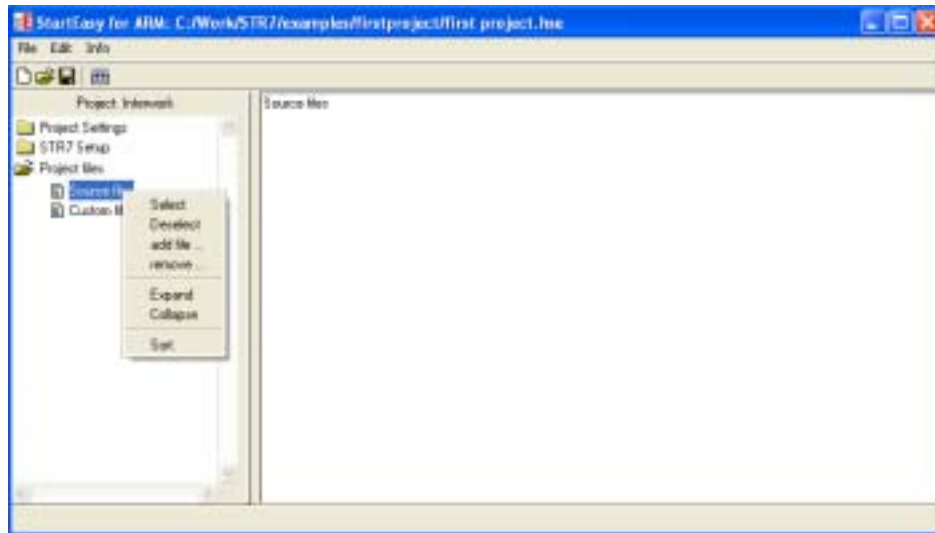
7. In the source code window, select the startup tab and view the startup source code. Scroll through until you find the vector table and familiarise yourself with this Assembler file.

5.6.1 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

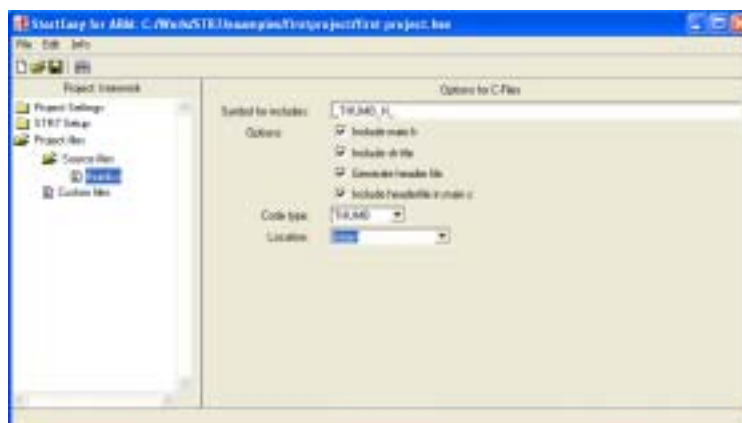
1. Select Project files\source files then right-click and select add file:



2. Add the file THUMB.c to your project.

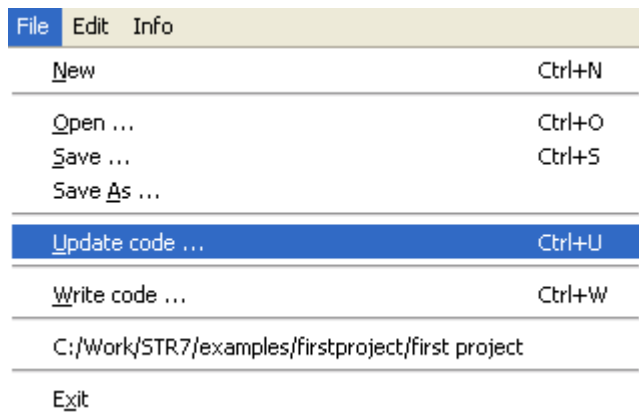


3. In the dialogue for THUMB.c select THUMB as the code type. This will cause all code within this module to be compiled in the THUMB instruction set. Also tick all the boxes to generate and include header files. Finally select the code location as "intern" to locate the code in the on-chip FLASH.



4. Select

fileupdate 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 compiled 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

0x000001E8	1EFF2FE1	bx lr
#144		{
0x000001EC	04E02DE5	str lr, [sp, #-4h]!
#156		init();
0x000001F0	D1FFFFEB	bl init
#167		thumb();
0x000001F4	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

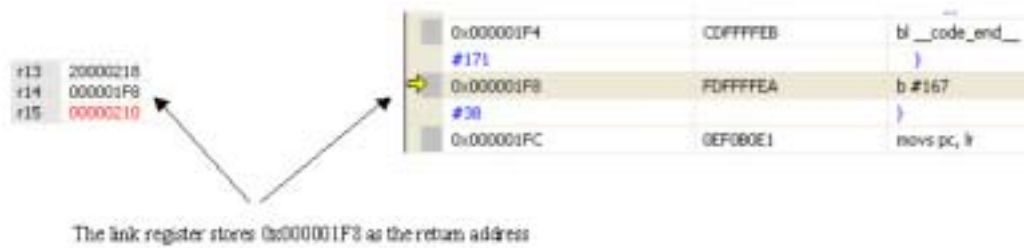
Return to the ARM calling function with a branch exchange on the contents of the link register

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



5.7 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

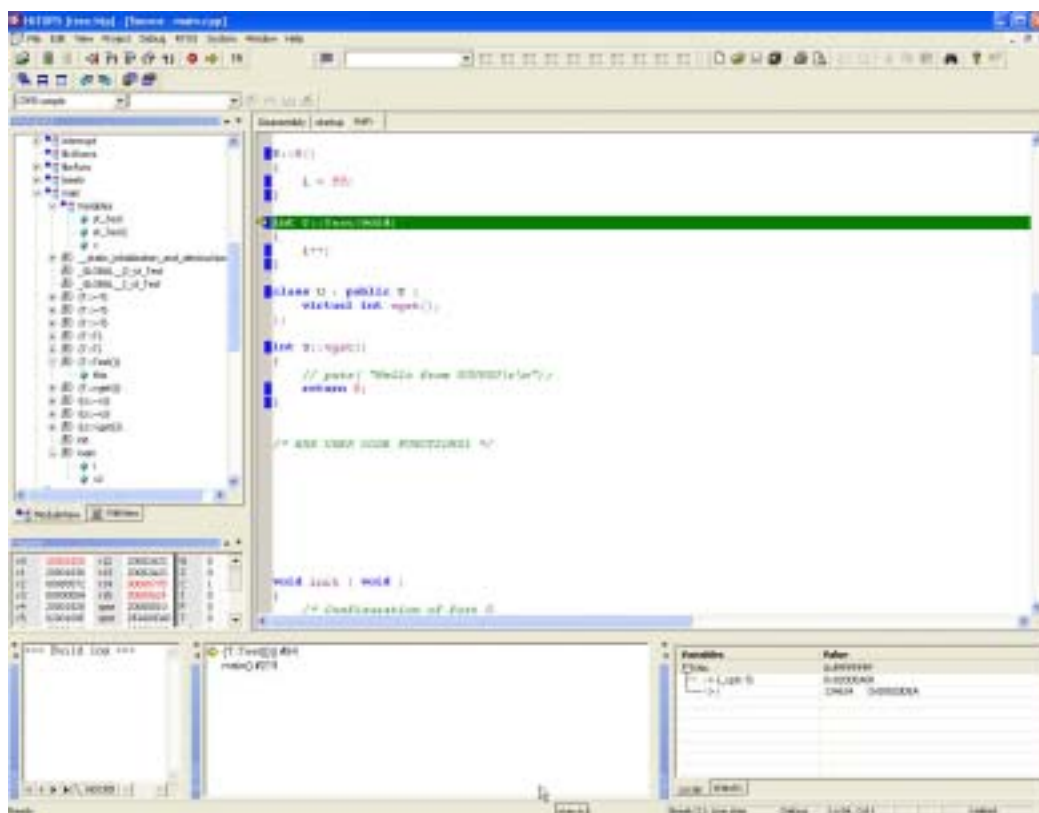
0x00000350	0C40BDE8	ldmia sp!, {r2,r3,r14}
0x00000354	0EF0B0E1	movs pc, lr
0x00000358	20020020	dw 20000220h

12. Finally run the program at full speed to see the LEDs FLASH in a new and interesting way
N.B. The C source for the SWI interrupt handler is in the module Interrupt.c

5.8 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()

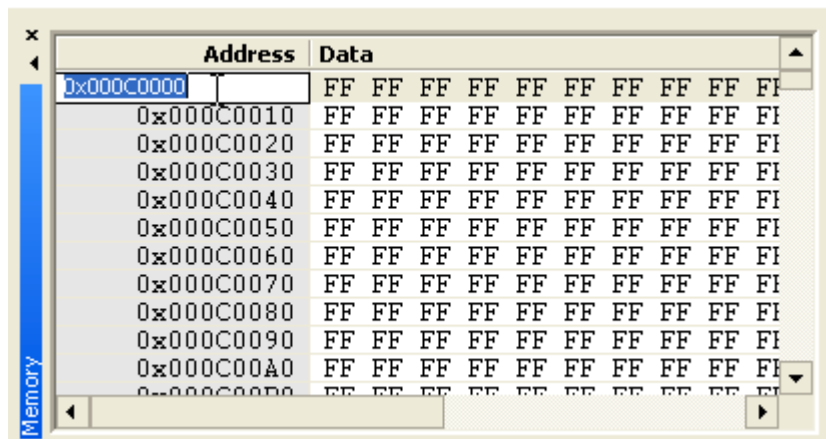


C++ support in HiTOP5 includes class awareness

5.9 Exercise 6: 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 0x000C0000



5. Set a breakpoint on the second call to the erase_FLASH() function.

```

erase_flash(0x00010000);
write_flash(0xC0000, 0x55555555);
write_flash(0xC0004, 0xAAAAAAAA);
erase_flash(0x00010000);

```

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.

```

0x000C0000  55 55 55 55 AA AA AA AA ]

```

8. Run the erase_FLASH function a second time and check that the contents of the FLASH sector have been reset to 0xFF.

```

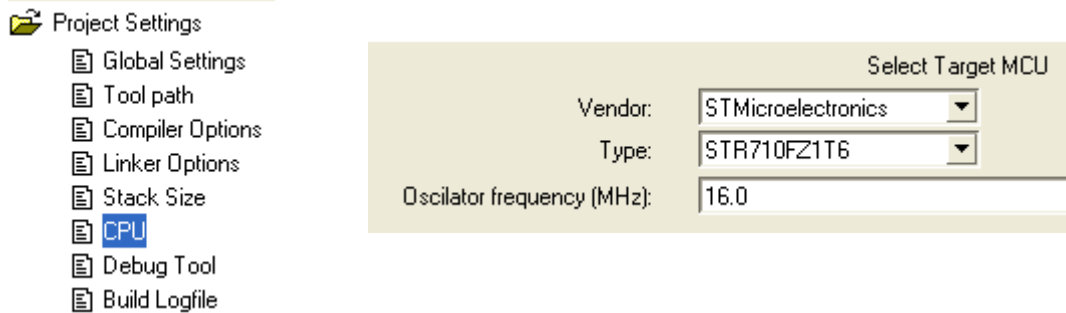
0x000C0000  FF FF FF FF FF FF FF FF ]

```

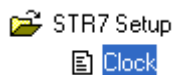
5.10 Exercise 7: Clock Configuration

In this example we will configure the Clock module to generate an MCLK of 48 Mhz and PCLK1 and PCLK2 of 24 MHz.

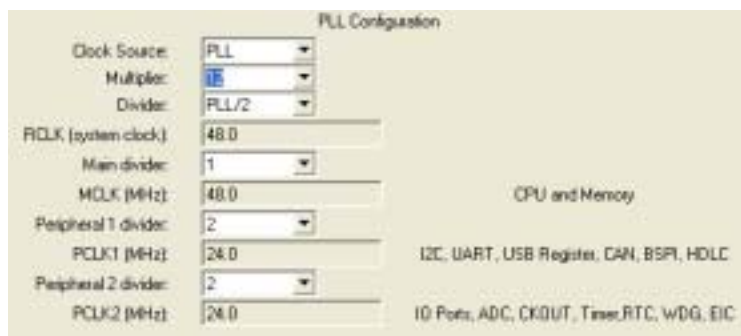
1. Open the StartEasy project in the Clock directory.
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. Open the Project settings CPU window and set the oscillator frequency to 16.0MHz.



4. Open the STR7\clock window



5. Configure the clock module for MCLK = 48 MHz PCLKx = 24MHz



6. Update the project and open the new HiTOP project
7. Run the code to main()

8. Open the View SFR\Power, reset,clock window and check the clock settings are configured correctly.

```
=====
```

Clock Flag Register PRCCU_CFR: 880B

DIV2 : OSCIN Divided by 2: Yes ▾

STOP_I : Stop Int.: not Pending ▾

CK2_16_I: CK2_16 Int.: not Pending ▾

CKAF_I : CKAF Int.: not Pending ▾

LOCK_I : Lock Int.: Pending ▾

RESET FLAGS: WKP_RES: No ▾

LVD_RES: No ▾ RTC_ALARM: No ▾

WDG_RES: No ▾ SOFTRES: No ▾

CK2_16 : CK2/16 is RCLK Source: No ▾

CKAF_ST: CK_AF is RCLK Source: No ▾

LOCK : PLL1 Locked in: Yes ▾

CSU_CKSEL: CSU Clock Select: PLL1 ▾

```
=====
```

PLL Config. Register PLL1CR: 0071

PLL Free Running Mode: disabled ▾

Reference Frequency Range: >3MHz ▾

PLL Multiplikation Factor: 16 ▾

PLL Division Factor: PLLCK/2 ▾

```
=====
```

Peripheral Enable Register PER: FFFF

PH_CK.4: USB Kernel Clock: enabled ▾

PH_CK.2: EMI Clock: enabled ▾

```
=====
```

5.11 Exercise 8: 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 a 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 S2 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. Stop
 - v. Standby
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 slowly switch off as the remaining power is drained.
7. To exit Standby mode, press the S3 wake-up button and you can cycle through the power down modes again.
8. Entering the Standby mode takes a few cycles, consequently the code immediately after the powerdown instruction will be executed (Port pin 2.15 is set). This overrun should be borne in mind when developing a real application
9. Finally, if while experimenting with the low power mode you lock out the Tantino debugger, change the bootmode pins so that the STR71x boots from RAM. This will allow you to regain control of the STR71x and erase the FLASH memory.

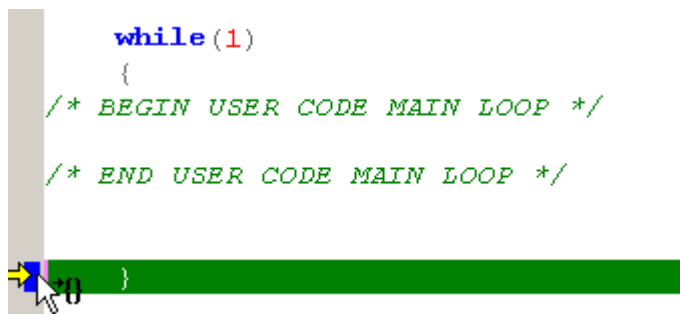
5.12 Exercise 9 : IRQ Interrupts

This example uses the External Interrupt Module (XTI) to generate an IRQ interrupt. This example examines the action of the EIC. Later we will look at the XTI module

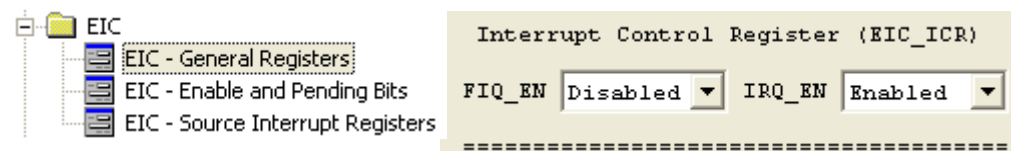
1. Open the StartEasy project in the IRQ directory
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino
3. In the STR7 Interrupt controller menu ensure that channel 5 (XTI) is enabled



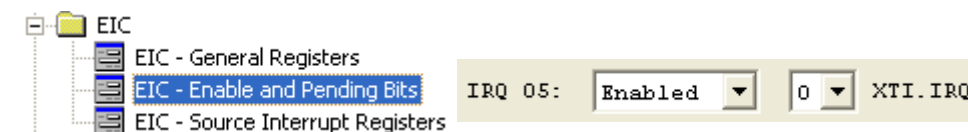
4. Update the project and open the new HiTOP project.
5. Run the code until it reaches the while(1) loop.



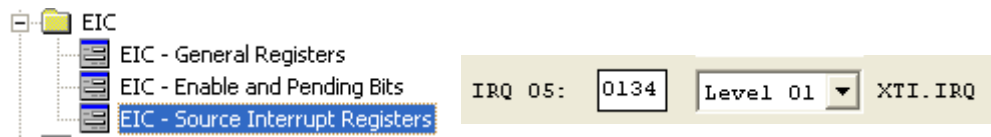
6. Open the view\SFR\EIC\ General registers. Check that the IRQ_EN bit is enabled.



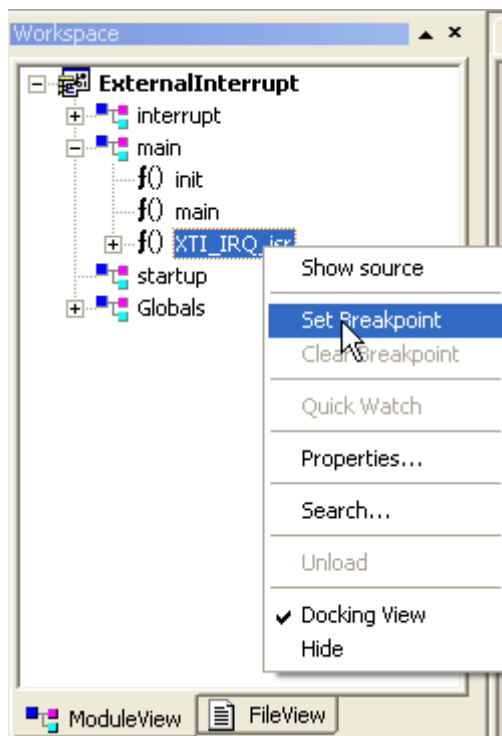
7. Next open the Enable and pending window and check that channel 5 is open but not pending.



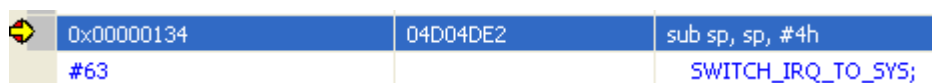
8. Finally open the source interrupt window and make a note of the value stored in the vector register



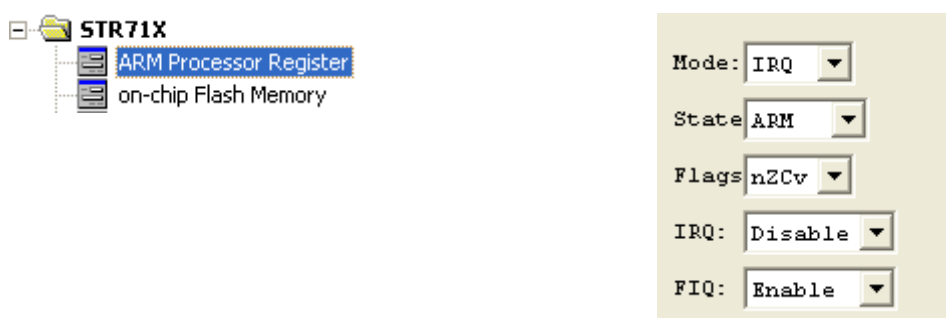
9. Switch back to the source window and locate the XTI_IRQ_ISR function in the code browser and set a breakpoint



10. Run the code and press the S2 interrupt button on the evaluation board
11. 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.



12. Open the View\SFRI\ARM Processor registers and check that the processor has entered IRQ mode and that the IRQ interrupts are disabled.



13. Step the code so the Macro is run and check that the processor has entered system mode and that the IRQ interrupts are enabled.

Mode: **SYS** ▼

State: ARM ▼

Flags: nZCv ▼

IRQ: **Enable** ▼

FIQ: **Enable** ▼

14. 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** ▼

15. 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 XTI interrupt:

➡	0x000001C8	04F05EE2	subs pc, lr, #4h
---	------------	----------	------------------

The current contents of the link register are:

r14 **000002F0**

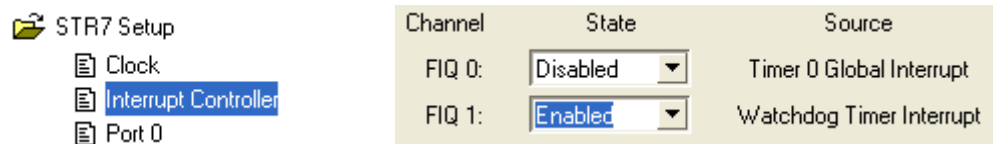
Return address is 0x000002EC:

	#199		}
➡	0x000002EC	FEFFFFEA	b #199

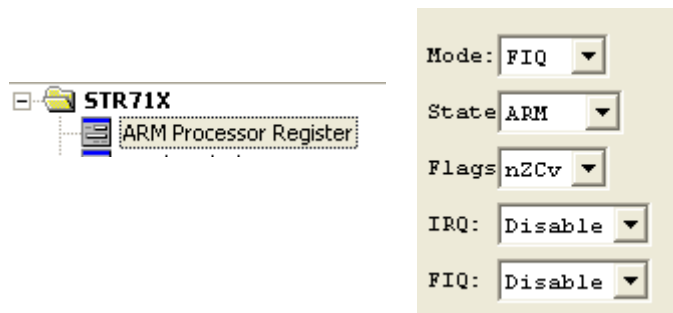
5.13 Exercise 10: FIQ Interrupt

This project configures the watchdog timer as a periodic timer and enables its interrupt. At the end of the count a fast interrupt is generated and is serviced by the default handler.

1. Open the StartEasy project in the SimpleInterrupt directory.
2. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
3. In the Interrupt controller window ensure that the FIQ1 watchdog timer interrupt is enabled.



4. Update the project and open the new HiTOP project.
5. Set a breakpoint on the FIQ service routine in the Interrupt.c module.
6. Run the code and it will hit the breakpoint.
7. Open the view\SFR\ARM CPU registers window and confirm the CPU is in FIQ mode.



8. Run the program at full speed and observe the LEDs flashing again

5.14 Exercise 11: External Interrupt

This example is basically the same code as the IRQ example but here we will take a closer look at the External interrupt configuration. The External interrupt switch is used to switch on the LEDs and the XTI software interrupt register is used to generate an interrupt that switches them off.

1. Open the StartEasy project in the XTI 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 the While loop so the XTI configuration code has run

```

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

/* END USER CODE MAIN LOOP */

}

```

5. Check that the XTI mask register is configured to enable pin 1.3 (the mask register bits do not represent a contiguous series of pins)

XTI Mask Register High (XTI_MRH)								00
XTI Mask Register LOW (XTI_MRL)								80
15	14	13	12	11	10	09	08	
0	0	0	0	0	0	0	0	
07	06	05	04	03	02	01	00	
1	0	0	0	0	0	0	0	

6. Check that the trigger register configures this pin to generate an interrupt on the falling edge

XTI Trig. Pol. Reg. High (XTI_TRH)								00
XTI Trig. Pol. Reg. Low (XTI_TRL)								00
15	14	13	12	11	10	09	08	
0	0	0	0	0	0	0	0	
07	06	05	04	03	02	01	00	
0	0	0	0	0	0	0	0	

7. Finally check that the XTI module is enabled correctly

Wake-up Control Register (XTI_CTRL)

STOP	0	IDIS	1	WKUP	0
------	--------------------------------	------	--------------------------------	------	--------------------------------

8. Set a breakpoint to the XTI ISR and run the code
9. Observe interrupts generated by the Software interrupt register and by pressing the interrupt button

XTI Pending Register High (XTI_PRH)

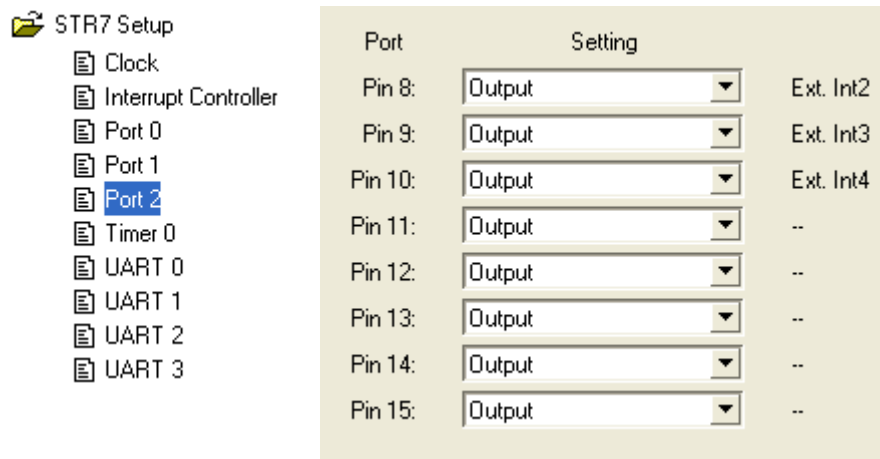
XTI Pending Register LOW (XTI_PRL)

15	14	13	12	11	10	09	08
0	0	0	0	1	1	1	1
07	06	05	04	03	02	01	00
1	0	0	0	0	0	0	0

5.15 Exercise 12 : GPIO

On the evaluation board the upper eight lines of port 2 are connected to a set of LEDs. This example demonstrates configuring these lines to be outputs and then drives the LED's in a "Tracer" pattern.

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 2 menu, switch pins 8 – 15 to Output



4. Select File\Update code to rewrite the project
5. Start HiTOP and load the GPIO project
6. Run the code to see the LED's update

5.16 Exercise 13: 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

1. On the evaluation board connect the port pins as follows:

Timer 1 Output compareA		Timer 2 Input Capture A
GPIO Pin 1.7	-----	GPIO Pin 0.14

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

Control Register 1 (TIM1_CR1):	Control Register 1 (TIM2_CR1):
EN: Timer counter is: Enabled	EN: Timer counter is: Enabled
PWMI: PWM Input is: not active	PWMI: PWM Input is: active
OLVLB: 0 FOLVB: Copy to Pin no	OLVLB: 0 FOLVB: Copy to Pin no
OLVLA: 1 FOLVA: Copy to Pin no	OLVLA: 0 FOLVA: Copy to Pin no
OCBE: Pin is used for: General I/O	OCBE: Pin is used for: General I/O
OCAE: Pin is used for: Compare A	OCAE: Pin is used for: General I/O
OPM: One Pulse Mode is: not active	OPM: One Pulse Mode is: not active
PWM: PWM Mode is: active	PWM: PWM Mode is: not active
IEDGB: Input Edge B is: falling	IEDGB: Input Edge B is: falling
IEDGA: Input Edge A is: falling	IEDGA: Input Edge A is: rising
EXEDG: Ext. Clock Edge is: falling	EXEDG: Ext. Clock Edge is: falling
ECKEN: External Clock is: Disabled	ECKEN: External Clock is: Disabled

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. The measured PWM pulse value is written to the LED bank.

5.17 Exercise 14: Real Time Clock (RTC)

This exercise demonstrates configuring the Real Time Clock to run from an external 32KHz clock source and generate a second interrupt, plus also create an alarm at 30 seconds.

The evaluation board is not fitted with a watch crystal for the real time clock and it is not very easy to retro-fit one. However the code in this example has been tested to ensure that it works as a basic example for the RTC.

5.18 Exercise 15: Analogue to Digital Converter

This example configures the Analog to Digital Converter (ADC) to make an eight bit conversion on channel zero and write the result to the LED pins. On the evaluation board the potentiometer is connected to channel zero and can be used to provide a variable analogue voltage.

1. Check that the ADC X 27 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.

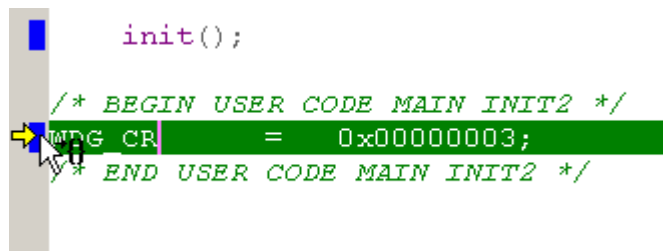


8. Run the code at full speed and check that the interrupt modulates the evaluation board LEDs by adjusting the potentiometer.

5.19 Exercise 16: 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 upto the watchdog enable line:



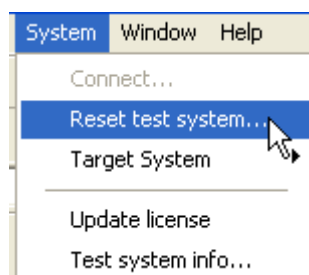
```

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 S2 interrupt button, this will force the CPU to execute a tight loop without serving the watchdog.
7. Try to halt the program execution - an error will be reported.
8. When the watchdog times out, it resets the processor and the on-chip JTAG module. This causes the debugger to loose control of the STR7.
9. To recover from such a failure select system\reset target system.

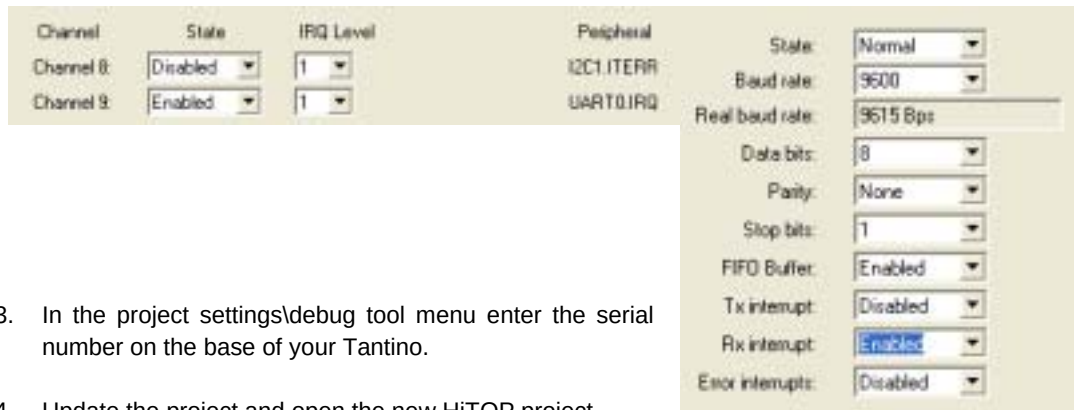


10. When debugging a real application, it is best not to enable the watchdog until the final stages of the project.

5.20 Exercise 17: 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. This project has the same configuration as example one, except that the Rx interrupt is enabled.



Channel	State	IRQ Level	Peripheral
Channel 8:	Disabled	1	I2C1.ITERR
Channel 9:	Enabled	1	UART0.IRQ

State:	Normal
Baud rate:	9600
Real baud rate:	9615 Bps
Data bits:	8
Parity:	None
Stop bits:	1
FIFO Buffer:	Enabled
Tx interrupt:	Disabled
Rx interrupt:	Enabled
Error interrupts:	Disabled

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. 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.
6. Start the code running and switch to Hyperterminal.
7. Type in some characters and they will be echoed back to you when the receive FIFO is full.

5.21 Exercise 18: 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.

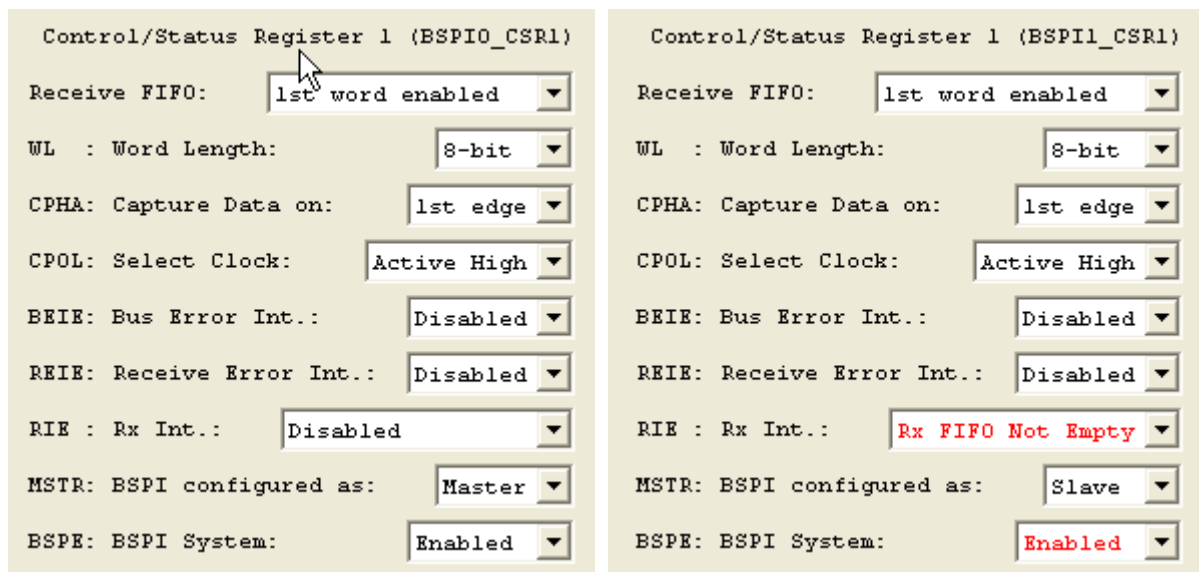
1. On the evaluation board connect the port header pins as follows:

```
MISO Port pin P0.0    > Port pin 0.4
MOSI Port pin P0.1    > Port pin 0.5
SCLK Port pin P0.2    > Port pin 0.6
```

2. The master slave select pin must be pulled high and a GPIO pin is used to select the slave peripheral:

```
SSL      Port pin P0.3    > 10K pull up to 3.3 V
SSL      Port pin P0.7    > 10K pull up to P2.00
```

3. Open the StartEasy project in the UART directory.
4. In the project settings\debug tool menu enter the serial number on the base of your Tantino.
5. Update the project and open the new HiTOP project .
6. Run the code so that it initialises both BSPI peripherals and examine their configuration in the SFR windows.



Control/Status Register 1 (BSPI0_CSR1)	Control/Status Register 1 (BSPI1_CSR1)
Receive FIFO: 1st word enabled	Receive FIFO: 1st word enabled
WL : Word Length: 8-bit	WL : Word Length: 8-bit
CPHA: Capture Data on: 1st edge	CPHA: Capture Data on: 1st edge
CPOL: Select Clock: Active High	CPOL: Select Clock: Active High
BEIE: Bus Error Int.: Disabled	BEIE: Bus Error Int.: Disabled
REIE: Receive Error Int.: Disabled	REIE: Receive Error Int.: Disabled
RIE : Rx Int.: Disabled	RIE : Rx Int.: Rx FIFO Not Empty
MSTR: BSPI configured as: Master	MSTR: BSPI configured as: Slave
BSPE: BSPI System: Enabled	BSPE: BSPI System: Enabled

7. Run the code and observe the ADC data being sent over the SPI bus and then being copied to the LED bank.

5.22 Exercise 19: I2C

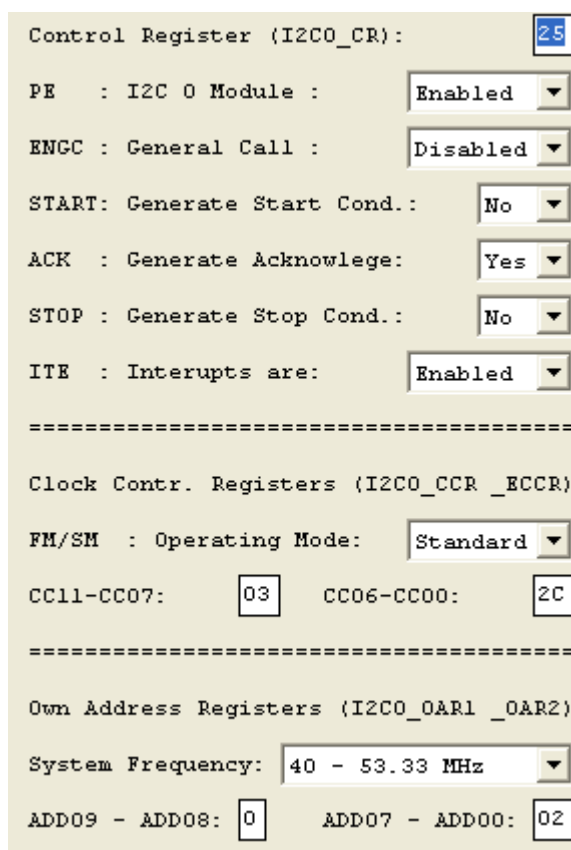
This example connects the two I2C peripherals together and configures them as master and slave. The master is then used to initiate the transfer of data between the two peripherals. As in the SPI example, ADC conversion data is sent between the two peripherals and written to the bank of LEDs.

1. On the evaluation board connect the port header pins as follows:

SCL	Port pin P0.2	---	Port pin 1.13
SDA	Port pin P0.3	---	Port pin 1.14

Both lines must also be pulled up to 3.3 volts by a 4K7 resistor.

2. Open the StartEasy project in the I2C 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 so that it initialises both I2C peripherals and examine their settings:



Control Register (I2C0_CR): 25

PE : I2C 0 Module : Enabled

ENGC : General Call : Disabled

START: Generate Start Cond.: No

ACK : Generate Acknowledge: Yes

STOP : Generate Stop Cond.: No

ITE : Interrupts are: Enabled

=====

Clock Contr. Registers (I2C0_CCR_ECCR)

FM/SM : Operating Mode: Standard

CC11-CC07: 03 CC06-CC00: 2C

=====

Own Address Registers (I2C0_OAR1_OAR2)

System Frequency: 40 - 53.33 MHz

ADD09 - ADD08: 0 ADD07 - ADD00: 02

6. Now run the code at full speed and check that the ADC data is sent over the I2C bus.

5.23 Exercise 20 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. As in the I2C and SPI examples, the ADC is used to generate data which is written to the bank of LEDs.

1. Ensure that the following jumpers are fitted.

TX	x36
RX	x32
SHDN	x37
TERM	x30

2. Open the StartEasy project in the CANbasic directory and check the configuration of the CAN TX and RX port pins.

Port	Setting	Alternate Functions
Pin 8:	Input	--
Pin 9:	Input	--
Pin 10:	Input	USB clock in (48MHz)
Pin 11:	Input	CAN RxD
Pin 12:	Alternate Function PP	CAN TxD
Pin 13:	Input	I2C clock, HDLC ref. clk. in
Pin 14:	Input	I2C serial data, HDLC RxD
Pin 15:	Input	HDLC TxD

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 .

- Run the code so that it initialises the CAN peripheral and examine its settings.

Bit Timing Register (CAN_BTR):

TSeg1: Segment before sample Point:

TSeg2: Segment after Sample Point:

SJW : Synchronisation Jump Width:

BRP : Baud Rate Prescaler:

=====

Test Register (CAN_TESTR):

Rx: : Rx Pin is:

Tx: : Tx Pin:

LBack : Module in L. Back Mode:

Silent: Module in Silent Mode:

Basic : Module in Basic Mode:

=====

BRP Extension Register (CAN_BRPR)

BRPE: Baud Rate Presc. Extension:

- 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.

Arb. Registers 1 and 2 (CAN_IF2_A1/2R)

ID 28-16: ID 15-0:

MsgVal: Message is Valid:

Std : Identifier:

Dir : Message Direction:

=====

Message Control Register (CAN_IF2MCR)

NewDat: New Data stored:

MsgLst: Message lost:

IntPnd: Interrupt is Pending:

UHash : Use acceptance Mask:

TxInt : Transmit Int.:

RxInt : Receive Int. :

ExtEn : Remote Enable:

TxBqst: Transmission Requested:

LoB : Single Obj. in Buffer:

DLC : Objekt Data Length Code:

=====

Data A/R Registers (CAN_IF2_DA/R1R)

Data A1: Data A2:

Data B1: Data B2:

Arb. Registers 1 and 2 (CAN_IF1_A1/2R)

ID 28-16: ID 15-0:

MsgVal: Message is Valid:

Std : Identifier:

Dir : Message Direction:

=====

Message Control Register (CAN_IF1MCR)

NewDat: New Data stored:

MsgLst: Message lost:

IntPnd: Interrupt is Pending:

UHash : Use acceptance Mask:

TxInt : Transmit Int.:

RxInt : Receive Int. :

ExtEn : Remote Enable:

TxBqst: Transmission Requested:

LoB : Single Obj. in Buffer:

DLC : Objekt Data Length Code:

=====

Data A/R Registers (CAN_IF1_DA/R1R)

Data A1: Data A2:

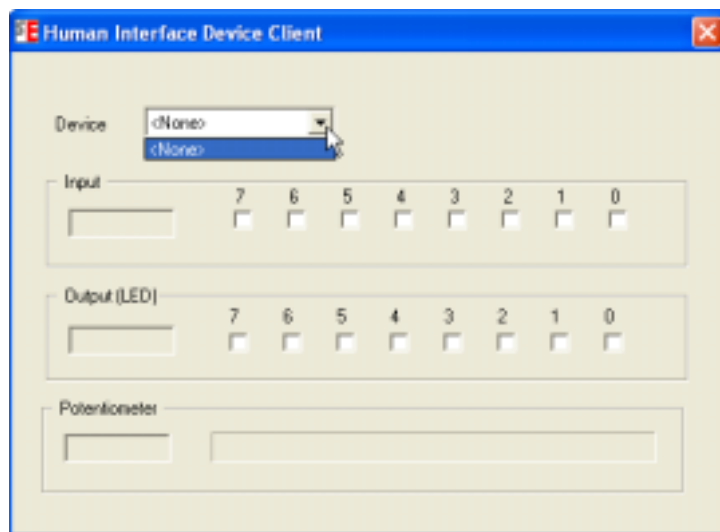
Data B1: Data B2:

7. Next load the full CAN example and examine how it uses the IF message registers to access the CAN message RAM.

5.24 Exercise 21: USB

The USB example demonstrates using the ST USB library to configure the peripheral to enumerate as a HID (Human Interface Device) device. A dedicated HID client for the XP operating system allows you to read/write data to the IO ports and display ADC data.

1. On the evaluation board check that the x31 USB Clk is fitted.
2. Open the StartEasy project in the USB 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. As the project size exceeds the 16K evaluation limit of HiTOP, you must get a 30 day full licence to be able to debug the project. The evaluation version will let you download the project but it does not provide debugging symbols.
6. Start the code running and plug the evaluation board into a spare USB port on the PC.
7. The STR71x will enumerate as a HID device. You can check it is correctly installed in the Control Panel\System\Hardware\device Manager\USB controllers window.
8. Once the STR71x board is installed successfully, it may be accessed by starting HID_Client.exe in the project root directory.



This client allows you to access the ADC and read/write to the IO Port pins.

This example is based on the ST USB library. The full documentation for the programming API can be found with the ST documentation on the CD accompanying this book.

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.keil.com

USB implementers forum
Website for USB complete with HID source code
commercial device driver for USB
USB and STR7 tools
STR7 tools

A silver Volvo station wagon is shown driving on a road. The car is positioned in the center of the frame, moving towards the viewer. Several orange traffic cones are visible on the road, some in the foreground and others further back, suggesting a controlled driving environment or a test track. The car's headlights are on, and the overall scene is captured in a slightly blurred, dynamic shot.

A screenshot of a Windows XP desktop. The Start menu is open, showing the 'All Programs' list. The 'Run' dialog box is open, showing the text 'Run' and the text 'C:\Program Files\Internet Explorer\iexplore.exe'. The desktop background is a light blue gradient.

A young man with dark hair is looking through the eyepiece of a microscope. The microscope is a light-colored, boxy model. The background is dark and out of focus.



VTMVROVVUUPS

eZ2E11E r hfE2Cklp8=t 3aa8<=i . - r t o ~ ~ Cl p j ; § E r j E m ~ ° © l = ' - ' E r 2 ° . l r h l = ' s Q = F b w k
qE2H00=MF0OTS5VOMSS