CEG2400 - Microcomputer Systems

Lecture 2: Memory

Philip Leong

Memory

- This lecture: focus on memory and addressing memory
  - In particular RAM (which is primary storage)
- Consists of storage cells
  - Arranged as bits (0 or 1)
  - We can deal with them in n-bit groups called words (typically 8, 16, 32 or 64 bits)
  - Sometimes use bytes which are 8-bits in length e.g. for characters
  - 1024=1K
  - 1024*1024=1M
  - 1024*1024*1024=1G
- Usually refer to memory size in bytes e.g. we say we have 128MB memory and rarely use words as the unit

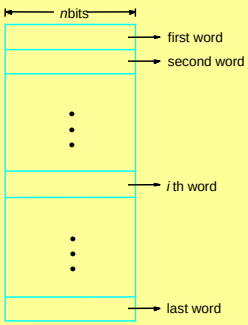
Addresses

- Use addresses to store or retrieve a single item of information
- For some k, memory consists of 2^k unique addresses which range from 0-2^k-1
  - The possible addresses are the address space of the computer
  - E.g. 28-bit address has 2^{28} (268435456) locations (normally we use words for addresses c.f. memory size on previous slide)

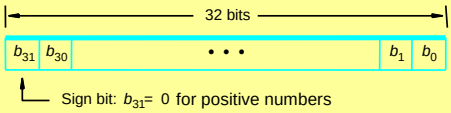
Byte addresses

- Information quantities: bit, byte, word
- Byte=8 bits
- Word typically varies 16-64 bits (the ARM architecture has 32-bit words which we will assume from now on)
- Most machines address memory in units of bytes
  - Implies for a 32-bit machine, successive words are at address 0, 4, 8, 12 …

Organization of memory

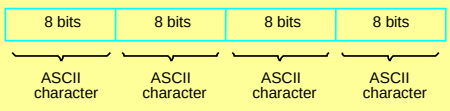


Integers and Characters



Sign bit: b_{31}= 0 for positive numbers
b_{31}= 1 for negative numbers

(a) A signed integer



(b) Four characters

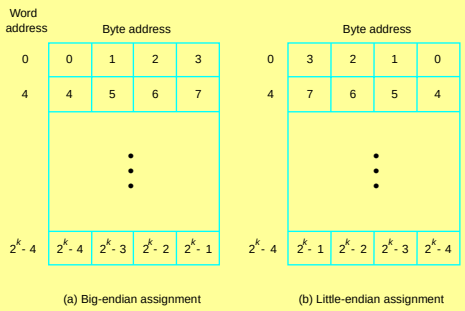
More/less significant bytes

- Consider the hexadecimal (base 16) 32-bit number 12342A3F=$1x16^7+2x16^6+3x16^5+4x16^4+2x16^3+10x16^2+3x16^1+15x16^0$
- This number has four bytes 12, 34, 2A, 4F (4x8=32-bits)
- Bytes/bits with higher weighting are “more significant” e.g. the byte 34 is more significant than 2A
- Bytes/bits with lower weighting are “less significant”
- We also use terms “most significant byte/bit” and “least significant byte/bit”

Big/little endian

- Two ways byte addresses can be assigned across words
  - more significant bytes first (big endian)
  - less significant bytes first (little endian)
- ARM allows both big and little endian addresses
  - LPC2100 fixes endianess to little endian

Big/little endian



Word alignment

- 32-bit words align naturally at addresses 0, 4, 8 …
  - These are aligned addresses
- Unaligned accesses are either not allowed or slower e.g. read a 32-bit word from address 1 (why?)
- In ARM
  - A word = 32-bits, half-word = 16 bits
  - Words aligned on 4-byte boundaries i.e. word addresses must be multiples of 4
  - Half words aligned on even byte boundaries

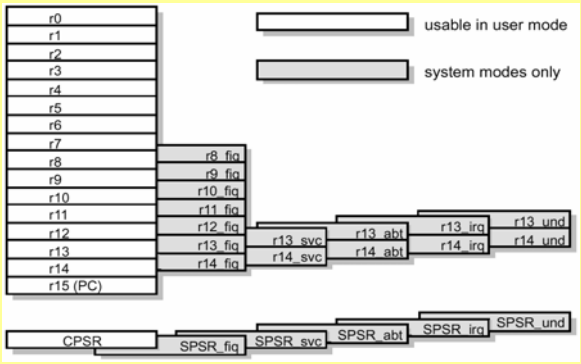
ARM Features

- Load-Store architecture
- Fixed-length (32-bit) instructions
- 3-address instruction formats (2 source operand regs, 1 result operand reg)
- Conditional execution of ALL instructions
- Multiple Load-Store register instructions
- A single-cycle n-bit shift with ALU operation
- Coprocessor instruction interfacing
- Thumb architecture (dense 16-bit compressed instruction set)

Registers

- Most higher end machines have a number of registers to store temporary data in the processor
  - Transfers to/from memory (i.e. Load/Store) are relatively slow
  - Operations only involving registers are fast
  - High speed makes time to execute an instruction shorter

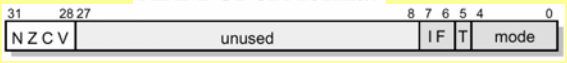
ARM Registers



ARM Programmer's Model (con't)

- R0 to R12 are general purpose registers (32-bits)
  - R13 stack pointer, R14 link register, R16 CPSR
  - Used by programmer for (almost) any purpose without restriction
- R15 is the Program Counter (PC)
- The remaining shaped ones are system mode registers - used during interrupts, exceptions or system programming (to be considered in later lectures)
- Current Program Status Register (CPSR) contains conditional flags and other status bits

ARM CPSR format



Condition codes

- In order to do conditional branches and other instructions, some operations implicitly set flags
- Four commonly used (1-bit) flags
  - N (negative) 1 if result –ve else 0
  - Z (zero) 1 if result 0 else 0
  - V (overflow) 1 if arithmetic overflow occurs else 0
  - C (carry) 1 if carry out occurs –ve else 0
- Quiz
  - Give examples of arithmetic operations which will cause N,Z,V,C to be set to 1

Examples

- N (negative)
  - 2-5
- Z (zero)
  - 10-10
- V (overflow)
  - 7FFFFFFF+1
- C (carry)
  - FFFFFFFF+1

Data Processing Instructions

- Three types of instructions:
  - Data Processing
  - Data Transfer
  - Control Flow
- Rules apply to ARM data processing instructions:
  - All operands are 32 bits, come either from registers or are specified as constants (called literals) in the instruction itself
  - The result is also 32 bits and is placed in a register
  - 3 operands - 2 for inputs and 1 for result
- Example:

```
ADD     r0, r1, r2              ; r0 := r1 + r2
```

- Works for both unsigned and 2's complement signed
- This may produce carry out signal and overflow bits, but ignored by default
- Result register can be the same with input operand register

Data Processing Instructions - Arithmetic operations

- Here are ARM's arithmetic operations:

```
ADD     r0, r1, r2          ; r0 := r1 + r2
ADC     r0, r1, r2          ; r0 := r1 + r2 + C
SUB     r0, r1, r2          ; r0 := r1 - r2
SBC     r0, r1, r2          ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2          ; r0 := r2 - r1
RSC     r0, r1, r2          ; r0 := r2 - r1 + C - 1
```

- RSB stands for reverse subtraction
- Operands may be unsigned or 2's complement signed integers
- 'C' is the carry (C) bit in the CPSR - Current Program Status Reg

Data Processing Instructions - Logical operations

- Here are ARM's bit-wise logical operations:

```
AND   r0, r1, r2      ; r0 := r1 and r2 (bit-by-bit for 32 bits)
ORR   r0, r1, r2      ; r0 := r1 or r2
EOR   r0, r1, r2      ; r0 := r1 xor r2
BIC   r0, r1, r2      ; r0 := r1 and not r2
```

- BIC stands for 'bit clear', where every '1' in the second operand clears the corresponding bit in the first:

```
r1:   0101 0011 1010 1111 1101 1010 0110 1011
r2:   1111 1111 1111 1111 0000 0000 0000 0000
r0:   0000 0000 0000 0000 1101 1010 0110 1011
```


Data Processing Instructions - Register Moves

- Here are ARM's register move operations:

```
MOV   r0, r2      ; r0 := r2
MVN   r0, r2      ; r0 := not r2
```

- MVN stands for 'move negated'

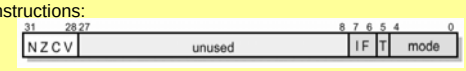
```
r2:   0101 0011 1010 1111 1101 1010 0110 1011
r0:   1010 1100 0101 0000 0010 0101 1001 0100
```


Data Processing Instructions - Comparison Operations

- Here are ARM's register comparison operations:

```
CMP   r1, r2      ; set cc on r1 - r2
CMN   r1, r2      ; set cc on r1 + r2
TST   r1, r2      ; set cc on r1 and r2
TEQ   r1, r2      ; set cc on r1 xor r2
```

- Results of subtract, add, and, xor are NOT stored in any registers
- Only the condition code bits (cc) in the CPSR are set or cleared by these instructions:



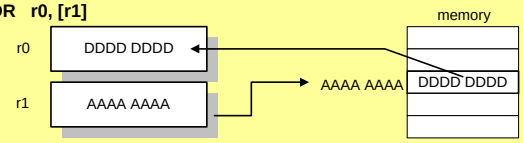
- Take CMP r1,r2 instruction:
  - N = 1 if MSB of (r1 - r2) is '1'
  - Z = 1 if (r1 - r2) = 0
  - C = 1 if (r1, r2) are both unsigned integers AND (r1 < r2)
  - V = 1 if (r1, r2) are signed integers AND (r1 < r2)

Data Transfer Instructions - single register load/store instructions

- Three basic forms of data transfer instructions:
  - Single register load/store instructions
  - Multiple register load/store instructions
  - Single register swap instructions
- Use a value in one register (called the **base** register) as a memory **address** and either loads the **data** value from that address into a destination register or stores the register value to memory:

```
LDR   r0, [r1]      ; r0 := mem32[r1]
STR   r0, [r1]      ; mem32[r1] := r0
```

- This is called **register-indirect addressing**
- **LDR r0, [r1]**

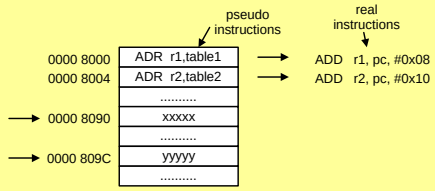


Data Transfer Instructions - Set up the address pointer

- Need to initialize address in r1 in the first place. How?
- Use ADR pseudo instruction - looks like normal instruction, but it does not really exists. Instead the assembler translates it to one or more real instructions.
- The following example copies data from TABLE 1 to TABLE2

```
copy      ADR   r1, TABLE1     ; r1 points to TABLE1
          ADR   r2, TABLE2     ; r2 points to TABLE2
          LDR   r0, [r1]       ; load first value ....
          STR   r0, [r2]       ;    and store it in TABLE2
          .......
TABLE1    ......               ; <source of data>
          ......
TABLE2    ......               ; <destination of data>
```


Data Transfer Instructions - ADR instruction



- How does the **ADR** instruction work? Address is 32-bit, difficult to put a 32-bit address value in a register in the first place
- Solution: Program Counter PC (**r15**) is often close to the desired data address value
- **ADR r1, TABLE1** is translated into an instruction that add or subtract a constant to PC (**r15**), and put the results in r1
- This constant is known as **PC-relative offset**, and it is calculated as: addr_of_table1 - (PC_value + 8)

LDR rn,=val pseudo-instruction

- "LDR rn,=value" load arbitrary value into a register in most efficient way possible:
  - a simple value like 0xff assembled with a move instruction (like MOV rn,#0xff which takes 1 cycle on a standard ARM7)
  - a more complex instruction converted into a load from a program counter-relative address (such as LDR rn,[pc,#OFFSET] which takes 3 cycles on an ARM7)
- Seems complicated, but you don't need to worry about it: just use LDR rn,=CONST.
  - There is a little confusion created because LDR is not only a pseudo-instruction, but depending on the context an actual ARM assembler opcode mnemonic (LoaD Register).

Data Transfer Instructions – Base plus offset addressing

- Extend the copy program further to copy NEXT word:

```
copy    ADR   r1, TABLE1    ; r1 points to TABLE1
        ADR   r2, TABLE2    ; r2 points to TABLE2
        LDR   r0, [r1]      ; load first value ....
        STR   r0, [r2]      ;    and store it in TABLE2
        ADD   r1, r1, #4    ; step r1 onto next word
        ADD   r2, r2, #4    ; step r2 onto next word
        LDR   r0, [r1]      ; load second value ...
        STR   r0, [r2]      ;    and store it
        ......
```

- Simplify with **pre-indexed addressing mode**

LDR r0, [r1, #4] ; r0 := mem_{32} [r1 + 4]

base address — r1 — offset — #4 — effective address

Data Transfer Instructions – pre-indexed with auto-indexing

- A simplified version is:

```
copy    ADR   r1, TABLE1    ; r1 points to TABLE1
        ADR   r2, TABLE2    ; r2 points to TABLE2
        LDR   r0, [r1]      ; load first value ....
        STR   r0, [r2]      ;    and store it in TABLE2
        LDR   r0, [r1, #4]  ; load second value ...
        STR   r0, [r2, #4]  ;    and store it
        ......
```

- Pre-indexed addressing does not change r1. Sometimes, it is useful to modify the base register to point to the new address. This is achieve by adding a '!', and is pre-indexed addressing with **auto-indexing**:

```
LDR   r0, [r1, #4]!   ;  r0 : = mem32 [r1 + 4]
                      ;  r1 := r1 + 4
```

- The '**!**' indicates that the instruction should update the base register after the data transfer

Data Transfer Instructions - post-indexed addressing

- Another useful form of the instruction is:

```
LDR   r0, [r1], #4    ;  r0 : = mem32 [r1]
                      ;  r1 := r1 + 4
```

- This is called: post-indexed addressing - the base address is used without an offset as the transfer address, after which it is auto-indexed.
- Using this, we can improve the copy program:

```
copy    ADR   r1, TABLE1    ; r1 points to TABLE1
        ADR   r2, TABLE2    ; r2 points to TABLE2
loop    LDR   r0, [r1], #4  ; get TABLE1 1st word ....
        STR   r0, [r2], #4  ;    copy it to TABLE2
        ???                 ; if more, go back to loop
        ......
TABLE1  ......              ;  < source of data >
```


Data Transfer Instructions Summary

- Size of data can be reduced to 8-bit byte with:

```
LDRB   r0, [r1]      ;  r0 : = mem8 [r1]
```

- Summary of addressing modes:

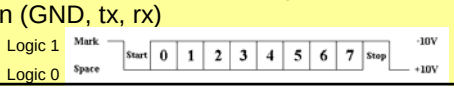
```
LDR   r0, [r1]                ; register-indirect addressing
LDR   r0, [r1 , # offset]     ; pre-indexed addressing
LDR   r0, [r1 , # offset]!    ; pre-indexed, auto-indexing
LDR   r0, [r1], # offset      ; post-indexed, auto-indexing
ADR   r0, address_label       ; PC relative addressing
```


Interfacing

- Many peripherals were designed to be connected directly to microcontrollers (uC) or are internal to the uC
- Interfacing involves hardware and software to allow them to be used within a software program

UART

- RS232 is a serial communications standard, most computers are equipped with RS232 ports although this is slowly being replaced by USB which has higher throughput and better interoperability
- A universal asynchronous receiver transmitter (UART) is a peripheral that translates between parallel data and this serial format
- Since it is asynchronous, no clock is needed to only 3 wires are needed for the simplest RS232 connection (GND, tx, rx)

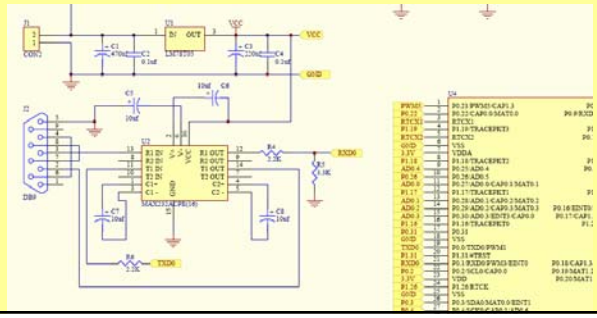


Hello world example

- Refer to User's manual (http://www.nxp.com/acrobat_download/usermanuals/UM_LPC21XX_LPC22XX_2.pdf) and uart programming applications note (http://www.nxp.com/acrobat_download/applicationnotes/AN10369_1.pdf)
- Course website which has the entire astartup.s and ahello.s files
  - astartup.s programs other peripherals on the chip that we will talk about later and calls __main
- Writing "Hello world\n" to the uart involves setting up all of the registers properly

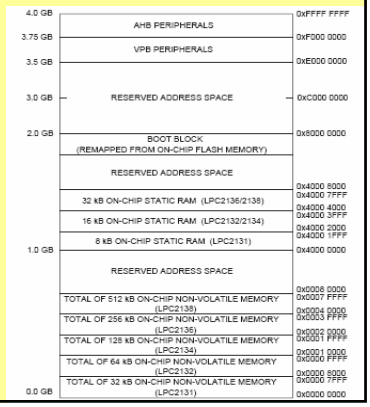
UART Hardware Interface

- All that is needed is a MAX232 chip which translates the 3.3V NXP levels to the +/-10V RS232 levels (VCC=5V)



Memory Map

- Software interface involves appropriately programming registers in the memory map to affect the hardware



UART Register Map

Table 73: UART0 register map

Name	Description	Bit functions and addresses MSB BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	LSB BIT0	Access	Reset value[1]	Address
U0RBR	Receiver Buffer Register	8-bit Read Data								RO	NA	0xE000 C000 (DLAB=0)
U0THR	Transmit Holding Register	8-bit Write Data								WO	NA	0xE000 C000 (DLAB=0)
U0DLL	Divisor Latch LSB	8-bit Data								R/W	0x01	0xE000 C000 (DLAB=1)
U0DLM	Divisor Latch MSB	8-bit Data								R/W	0x00	0xE000 C004 (DLAB=1)
U0IER	Interrupt Enable Register	Reserved	Reserved	Reserved	Reserved	Reserved	Enable RX Line Status Interrupt	Enable THRE Interrupt	Enable RX Data Available Interrupt	R/W	0x00	0xE000 C004 (DLAB=0)
U0IIR	Interrupt ID Register	FIFOs Enabled		Reserved	Reserved	IIR3	IIR2	IIR1	IIR0	RO	0x01	0xE000 C008
U0FCR	FIFO Control Register	RX Trigger			Reserved	Reserved	Reserved	TX FIFO Reset	RX FIFO Reset	FIFO Enable	WO	0x00	0xE000 C008
U0LCR	Line Control Register	DLAB	Set Break	Stick Parity	Even Parity Select	Parity Enable	Number of Stop Bits	Word Length Select			R/W	0x00	0xE000 C00C
U0LSR	Line Status Register	RX FIFO Error	TEMT	THRE	BI	FE	PE	OE	DR	RO	0x60	0xE000 C014
U0SCR	Scratch Pad Register	8-bit Data								R/W	0x00	0xE000 C01C
U0TER	Transmit Enable Register	TXEN	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	R/W	0x80	0xE000 C030

[1] Reset value reflects the data stored in used bits only. It does not include reserved bits content.

Uart Registers

```
; UART0 registers

U0RBR EQU   0xE000C000
U0THR EQU   0xE000C000
U0IER EQU   0xE000C004
U0IIR EQU   0xE000C008
U0FCR EQU   0xE000C008
U0LCR EQU   0xE000C00C
U0LSR EQU   0xE000C014
U0SCR EQU   0xE000C01C
U0DLL EQU   0xE000C000
U0DLM EQU   0xE000C004
PINSEL0 EQU 0xE002C000
```

EQU Pseudo instruction

- **X EQU2**
- This is an assembler directive that is used to give a value to a label name. In this example it assigns num the value 2. Thus when num is used elsewhere in the code, the value 2 will be substituted (similar to using a #define to set up a constant in C).

PINSEL0

7.4.1 Pin Function Select Register 0 (PINSEL0 - 0xE002 C000)

The PINSEL0 register controls the functions of the pins as per the settings listed in Table 61. The direction control bit in the IO0DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

Table 58: Pin function Select register 0 (PINSEL0 - address 0xE002 C000) bit description

Bit	Symbol	Value	Function	Reset value
1:0	P0.0	00	GPIO Port 0.0	0
		01	TXD (UART0)	
		10	PWM1	
		11	Reserved	
3:2	P0.1	00	GPIO Port 0.1	0
		01	RxD (UART0)	
		10	PWM3	
		11	EINT0	
5:4	P0.2	00	GPIO Port 0.2	0
		01	SCL0 (I²C0)	
		10	Capture 0.0 (Timer 0)	
		11	Reserved	
7:6	P0.3	00	GPIO Port 0.3	0
		01	SDA0 (I²C0)	
		10	Match 0.0 (Timer 0)	
		11	EINT1	
9:8	P0.4	00	GPIO Port 0.4	0

U0FCR

Table 82: UART0 FIFO Control Register (U0FCR - address 0xE000 C008) bit description

Bit	Symbol	Value	Description	Reset value
0	FIFO Enable	0	UART0 FIFOs are disabled. Must not be used in the application.	0
		1	Active high enable for both UART0 Rx and TX FIFOs and U0FCR[7:1] access. This bit must be set for proper UART0 operation. Any transition on this bit will automatically clear the UART0 FIFOs.	
1	RX FIFO Reset	0	No impact on either of UART0 FIFOs.	0
		1	Writing a logic 1 to U0FCR[1] will clear all bytes in UART0 Rx FIFO and reset the pointer logic. This bit is self-clearing.	
2	TX FIFO Reset	0	No impact on either of UART0 FIFOs.	0
		1	Writing a logic 1 to U0FCR[2] will clear all bytes in UART0 TX FIFO and reset the pointer logic. This bit is self-clearing.	
5:3	-	0	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
7:6	RX Trigger Level		These two bits determine how many receiver UART0 FIFO characters must be written before an interrupt is activated.	0
		00	Trigger level 0 (1 character or 0x01)	
		01	Trigger level 1 (4 characters or 0x04)	
		10	Trigger level 2 (8 characters or 0x08)	
		11	Trigger level 3 (14 characters or 0x0E)	

U0LCR

Table 83: UART0 Line Control Register (U0LCR - address 0xE000 C00C) bit description

Bit	Symbol	Value	Description	Reset value
1:0	Word Length Select	00	5 bit character length	0
		01	6 bit character length	
		10	7 bit character length	
		11	8 bit character length	
2	Stop Bit Select	0	1 stop bit.	0
		1	2 stop bits (1.5 if U0LCR[1:0]=00).	
3	Parity Enable	0	Disable parity generation and checking.	0
		1	Enable parity generation and checking.	
5:4	Parity Select	00	Odd parity. Number of 1s in the transmitted character and the attached parity bit will be odd.	0
		01	Even Parity. Number of 1s in the transmitted character and the attached parity bit will be even.	
		10	Forced "1" stick parity.	
		11	Forced "0" stick parity.	
6	Break Control	0	Disable break transmission.	0
		1	Enable break transmission. Output pin UART0 TXD is forced to logic 0 when U0LCR[6] is active high.	
7	Divisor Latch Access Bit (DLAB)	0	Disable access to Divisor Latches.	0
		1	Enable access to Divisor Latches.	

U0DLL

- U0DLL = Fpclk / (16 * Baudrate)
- If we want 57600 baud (Fpclk=13.824MHz) then U0DLL=15 (U0DLM:U0DLL)

Init uart

```
iuart0
  MOV R1, #0x5
  LDR R0, =PINSEL0
  STR R1, [R0]

  MOV R1, #0x7
  LDR R0, =U0FCR
  STRB      R1,[R0]

  MOV R1, #0x83
  LDR R0, =U0LCR
  STRB      R1,[R0]

  MOV R1, #0x0f
  LDR R0, =U0DLL
  STRB      R1,[R0]

  MOV R1, #0x00
  LDR R0, =U0DLM
  STRB      R1,[R0]

  MOV R1, #0x03
  LDR R0, =U0LCR
  STRB      R1,[R0]
  BX  R14
```

What do all of these values do?

U0LSR

			Note: A framing error is associated with the character at the top of the UART0 RBR FIFO.
		0	Framing error status is inactive.
		1	Framing error status is active.
4	Break Interrupt (BI)		When RXD0 is held in the spacing state (all 0's) for one full character transmission (start, data, parity, stop), a break interrupt occurs. Once the break condition has been detected, the receiver goes idle until RXD0 goes to marking state (all 1's). An U0LSR read clears this status bit. The time of break detection is dependent on U0FCR[0]. **Note:** The break interrupt is associated with the character at the top of the UART0 RBR FIFO.	0
		0	Break interrupt status is inactive.
		1	Break interrupt status is active.
5	Transmitter Holding Register Empty (THRE))		THRE is set immediately upon detection of an empty UART0 THR and is cleared on a U0THR write.	1
		0	U0THR contains valid data.
		1	U0THR is empty.
6	Transmitter Empty (TEMT)		TEMT is set when both U0THR and U0TSR are empty; TEMT is cleared when either the U0TSR or the U0THR contain valid data.	1
		0	U0THR and/or the U0TSR contains valid data.
		1	U0THR and the U0TSR are empty.
7	Error in RX FIFO (RXFE)		U0LSR[7] is set when a character with a Rx error such as framing error, parity error or break interrupt, is loaded into the U0RBR. This bit is cleared when the U0LSR register is read and there are no subsequent errors in the UART0 FIFO.	0
		0	U0RBR contains no UART0 RX errors or U0FCR[0]=0.
		1	UART0 RBR contains at least one UART0 RX error.

Write character in R0

```
writec
   LDR    R1, =U0LSR
  LDRB    R1, [R1]
  TST     R1, #0x40
  BEQ     writec
  LDR     R1, =U0THR
  STRB    R0, [R1]
  BX R14       ; return from subroutine
```


Hello World

```
; User Initial Stack & Heap
        AREA    |.text|, CODE, READONLY

        EXPORT  __main
__main BL      iuart0
loop   MOV     R0, #'H'
       BL      writec
       MOV     R0, #'e'
       BL      writec
       MOV     R0, #'l'
       BL      writec
       …
       MOV     R0, #'\r'
       BL      writec
       B       loop
```


Keil Simulator

- Can simulate peripherals, allow single stepping, view registers and device configuration etc
- Make sure your program runs in the simulator before downloading