Dariusz Makowski

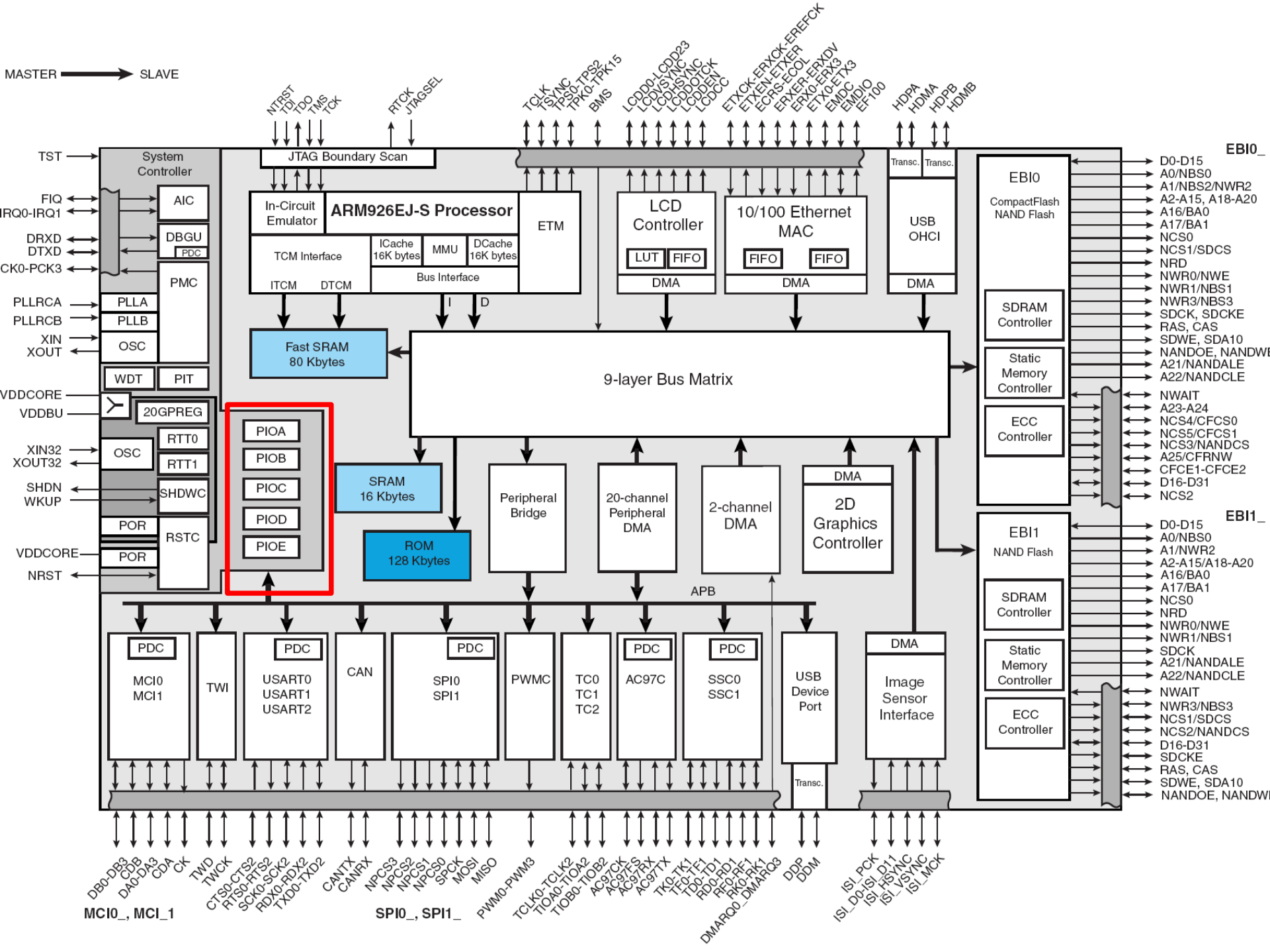
Department of Microelectronics and Computer Science

tel. 631 2720

dmakow@dmcs.pl

http://neo.dmcs.pl/sw

Input-Output ports of AMR processor based on ATMEL ARM AT91SAM9263


MASTER
SLAVE
System Controller
JTAG Boundary Scan
In-Circuit Emulator
ARM926EJ-S Processor
ETM
TCM Interface
ICache 16K bytes
MMU
DCache 16K bytes
Bus Interface
ITCM
DTCM
LCD Controller
LUT
FIFO
DMA
10/100 Ethernet MAC
FIFO
FIFO
DMA
Transc.
Transc.
USB OHCI
DMA
EBI0
CompactFlash NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
EBI1
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
AIC
DBGU
PDC
PMC
PLLA
PLLB
OSC
WDT
PIT
20GPREG
RTT0
RTT1
OSC
SHDWC
POR
RSTC
POR
PIOA
PIOB
PIOC
PIOD
PIOE
Fast SRAM 80 Kbytes
9-layer Bus Matrix
SRAM 16 Kbytes
ROM 128 Kbytes
Peripheral Bridge
20-channel Peripheral DMA
2-channel DMA
DMA
2D Graphics Controller
APB
PDC
MCI0 MCI1
TWI
PDC
USART0 USART1 USART2
CAN
PDC
SPI0 SPI1
PWMC
TC0 TC1 TC2
PDC
AC97C
PDC
SSC0 SSC1
USB Device Port
Transc.
DMA
Image Sensor Interface
TST
FIQ
IRQ0-IRQ1
DRXD
DTXD
PCK0-PCK3
PLLRCA
PLLRCB
XIN
XOUT
VDDCORE
VDDBU
XIN32
XOUT32
SHDN
WKUP
VDDCORE
NRST
NTRST TDI TDO TMS TCK
RTCK JTAGSEL
TCLK TSYNC TPS0-TPS2 TPK0-TPK15 BMS
LCDD0-LCDD23 LCDVSYNC LCDHSYNC LCDDOTCK LCDDEN LCDCC
ETXCK-ERXCK-EREFCK ETXEN-ETXER ECRS-ECOL ERXER-ERXDV ERX0-ERX3 ETX0-ETX3 EMDC EMDIO EF100
HDPA HDMA HDPB HDMB
EBI0_
D0-D15
A0/NBS0
A1/NBS2/NWR2
A2-A15, A18-A20
A16/BA0
A17/BA1
NCS0
NCS1/SDCS
NRD
NWR0/NWE
NWR1/NBS1
NWR3/NBS3
SDCK, SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
A21/NANDALE
A22/NANDCLE
NWAIT
A23-A24
NCS4/CFCS0
NCS5/CFCS1
NCS3/NANDCS
A25/CFRNW
CFCE1-CFCE2
D16-D31
NCS2
EBI1_
D0-D15
A0/NBS0
A1/NWR2
A2-A15/A18-A20
A16/BA0
A17/BA1
NCS0
NRD
NWR0/NWE
NWR1/NBS1
SDCK
A21/NANDALE
A22/NANDCLE
NWAIT
NWR3/NBS3
NCS1/SDCS
NCS2/NANDCS
D16-D31
SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
DB0-DB3 CDB DA0-DA3 CDA CK
MCI0_, MCI_1
TWD TWCK
CTS0-CTS2 RTS0-RTS2 SCK0-SCK2 RDX0-RDX2 TXD0-TXD2
CANTX CANRX
NPCS3 NPCS2 NPCS1 NPCS0 SPCK MOSI MISO
SPI0_, SPI1_
PWM0-PWM3
TCLK0-TCLK2 TIOA0-TIOA2 TIOB0-TIOB2
AC97CK AC97FS AC97RX AC97TX
TK0-TK1 TF0-TF1 TD0-TD1 RD0-RD1 RF0-RF1 RK0-RK1 DMARQ0_DMARQ3
DDP DDM
ISI_PCK ISI_D0-ISI_D11 ISI_HSYNC ISI_VSYNC ISI_MCK

Documentation for AT91SAM9263 Microcontroller

Features

- Incorporates the ARM926EJ-S™ ARM® Thumb® Processor
  - DSP Instruction Extensions, Jazelle® Technology for Java® Acceleration
  - 16 Kbyte Data Cache, 16 Kbyte Instruction Cache, Write Buffer
  - 220 MIPS at 200 MHz
  - Memory Management Unit
  - EmbeddedICE™, Debug Communication Channel Support
  - Mid-level Implementation Embedded Trace Macrocell™
- Bus Matrix
  - Nine 32-bit-layer Matrix, Allowing a Total of 28.8 Gbps of On-chip Bus Bandwidth
  - Boot Mode Select Option, Remap Command
- Embedded Memories
  - One 128 Kbyte Internal ROM, Single-cycle Access at Maximum Bus Matrix Speed
  - One 80 Kbyte Internal SRAM, Single-cycle Access at Maximum Processor or Bus Matrix Speed
  - One 16 Kbyte Internal SRAM, Single-cycle Access at Maximum Bus Matrix Speed
- Dual External Bus Interface (EBI0 and EBI1)
  - EBI0 Supports SDRAM, Static Memory, ECC-enabled NAND Flash and CompactFlash®
  - EBI1 Supports SDRAM, Static Memory and ECC-enabled NAND Flash
- DMA Controller (DMAC)
  - Acts as one Bus Matrix Master
  - Embeds 2 Unidirectional Channels with Programmable Priority, Address Generation, Channel Buffering and Control
- Twenty Peripheral DMA Controller Channels (PDC)
- LCD Controller
  - Supports Passive or Active Displays
  - Up to 24 bits per Pixel in TFT Mode, Up to 16 bits per Pixel in STN Color Mode
  - Up to 16M Colors in TFT Mode, Resolution Up to 2048x2048, Supports Virtual Screen Buffers

ATMEL®

AT91 ARM Thumb Microcontrollers

AT91SAM9263

Preliminary

Documentation for AT91SAM9263 – I/O Ports

AT91SAM9263 Preliminary

31. Parallel Input/Output Controller (PIO)

31.1 Overview

The Parallel Input/Output Controller (PIO) manages up to 32 fully programmable input/output lines. Each I/O line may be dedicated as a general-purpose I/O or be assigned to a function of an embedded peripheral. This assures effective optimization of the pins of a product.

Each I/O line is associated with a bit number in all of the 32-bit registers of the 32-bit wide User Interface.

Each I/O line of the PIO Controller features:

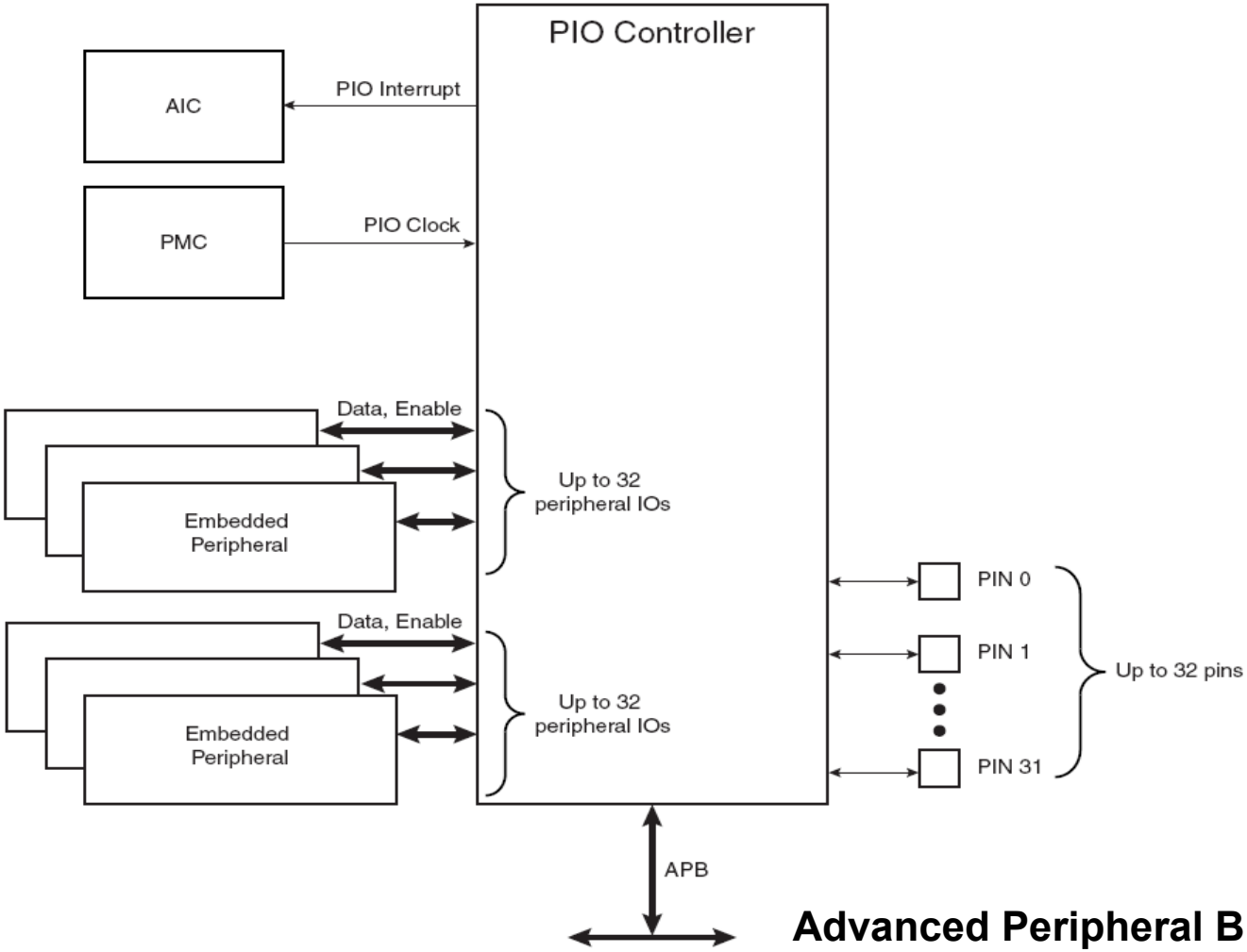
- An input change interrupt enabling level change detection on any I/O line.
- A glitch filter providing rejection of pulses lower than one-half of clock cycle.
- Multi-drive capability similar to an open drain I/O line.
- Control of the the pull-up of the I/O line.
- Input visibility and output control.

The PIO Controller also features a synchronous output providing up to 32 bits of data output in a single write operation.

Źródło: ATMEL, doc6249.pdf, strona 425

Block Diagram of 32-bits I/O Port

Figure 31-1. Block Diagram



Advanced Peripheral Bus

Power Consumption vs Clock Signal

31.3.3 Power Management

The Power Management Controller controls the PIO Controller clock in order to save power. Writing any of the registers of the user interface does not require the PIO Controller clock to be enabled. This means that the configuration of the I/O lines does not require the PIO Controller clock to be enabled.

However, when the clock is disabled, not all of the features of the PIO Controller are available. Note that the Input Change Interrupt and the read of the pin level require the clock to be validated.

After a hardware reset, the PIO clock is disabled by default.

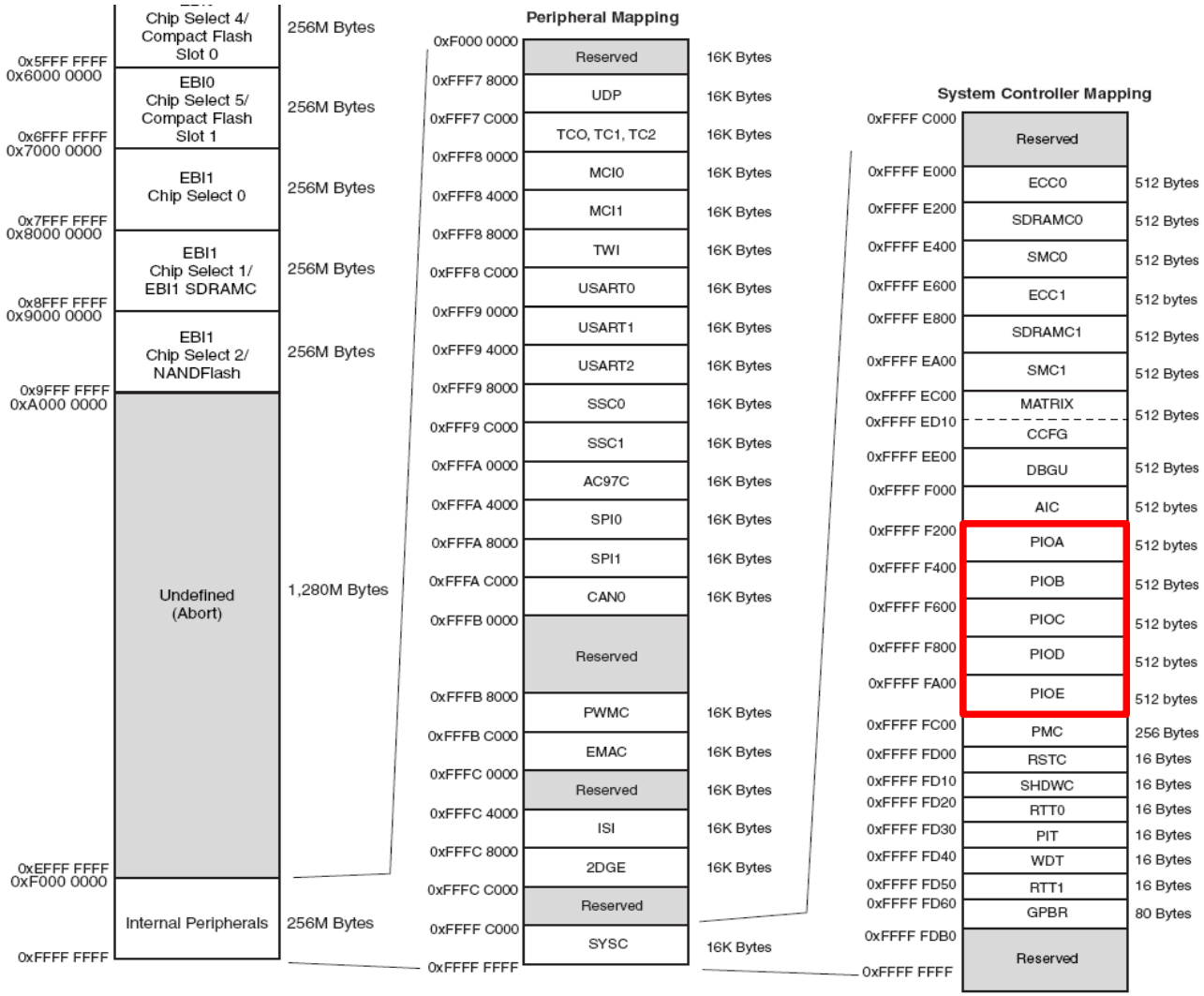
The user must configure the Power Management Controller before any access to the input line information.

Control Registers for I/O ports

Table 31-2. Register Mapping

Offset	Register	Name	Access	Reset
0x0000	PIO Enable Register	PIO_PER	Write-only	–
0x0004	PIO Disable Register	PIO_PDR	Write-only	–
0x0008	PIO Status Register	PIO_PSR	Read-only	(1)
0x000C	Reserved			
0x0010	Output Enable Register	PIO_OER	Write-only	–
0x0014	Output Disable Register	PIO_ODR	Write-only	–
0x0018	Output Status Register	PIO_OSR	Read-only	0x0000 0000
0x001C	Reserved			
0x0020	Glitch Input Filter Enable Register	PIO_IFER	Write-only	–
0x0024	Glitch Input Filter Disable Register	PIO_IFDR	Write-only	–
0x0028	Glitch Input Filter Status Register	PIO_IFSR	Read-only	0x0000 0000
0x002C	Reserved			
0x0030	Set Output Data Register	PIO_SODR	Write-only	–
0x0034	Clear Output Data Register	PIO_CODR	Write-only	
0x0038	Output Data Status Register	PIO_ODSR	Read-only or(2) Read-write	–
0x003C	Pin Data Status Register	PIO_PDSR	Read-only	(3)
0x0040	Interrupt Enable Register	PIO_IER	Write-only	–
0x0044	Interrupt Disable Register	PIO_IDR	Write-only	–
0x0048	Interrupt Mask Register	PIO_IMR	Read-only	0x00000000
0x004C	Interrupt Status Register(4)	PIO_ISR	Read-only	0x00000000
0x0050	Multi-driver Enable Register	PIO_MDER	Write-only	–
0x0054	Multi-driver Disable Register	PIO_MDDR	Write-only	–
0x0058	Multi-driver Status Register	PIO_MDSR	Read-only	0x00000000
0x005C	Reserved			
0x0060	Pull-up Disable Register	PIO_PUDR	Write-only	–
0x0064	Pull-up Enable Register	PIO_PUER	Write-only	–
0x0068	Pad Pull-up Status Register	PIO_PUSR	Read-only	0x00000000
0x006C	Reserved			

Memory Map


Chip Select 4/ Compact Flash Slot 0
256M Bytes
0x5FFF FFFF
0x6000 0000
EBI0 Chip Select 5/ Compact Flash Slot 1
256M Bytes
0x6FFF FFFF
0x7000 0000
EBI1 Chip Select 0
256M Bytes
0x7FFF FFFF
0x8000 0000
EBI1 Chip Select 1/ EBI1 SDRAMC
256M Bytes
0x8FFF FFFF
0x9000 0000
EBI1 Chip Select 2/ NANDFlash
256M Bytes
0x9FFF FFFF
0xA000 0000
Undefined (Abort)
1,280M Bytes
0xEFFF FFFF
0xF000 0000
Internal Peripherals
256M Bytes
0xFFFF FFFF
Peripheral Mapping
0xF000 0000
Reserved
16K Bytes
0xFFF7 8000
UDP
16K Bytes
0xFFF7 C000
TCO, TC1, TC2
16K Bytes
0xFFF8 0000
MCI0
16K Bytes
0xFFF8 4000
MCI1
16K Bytes
0xFFF8 8000
TWI
16K Bytes
0xFFF8 C000
USART0
16K Bytes
0xFFF9 0000
USART1
16K Bytes
0xFFF9 4000
USART2
16K Bytes
0xFFF9 8000
SSC0
16K Bytes
0xFFF9 C000
SSC1
16K Bytes
0xFFFA 0000
AC97C
16K Bytes
0xFFFA 4000
SPI0
16K Bytes
0xFFFA 8000
SPI1
16K Bytes
0xFFFA C000
CAN0
16K Bytes
0xFFFB 0000
Reserved
0xFFFB 8000
PWMC
16K Bytes
0xFFFB C000
EMAC
16K Bytes
0xFFFC 0000
Reserved
16K Bytes
0xFFFC 4000
ISI
16K Bytes
0xFFFC 8000
2DGE
16K Bytes
0xFFFC C000
Reserved
0xFFFF C000
SYSC
16K Bytes
0xFFFF FFFF
System Controller Mapping
0xFFFF C000
Reserved
0xFFFF E000
ECC0
512 Bytes
0xFFFF E200
SDRAMC0
512 Bytes
0xFFFF E400
SMC0
512 Bytes
0xFFFF E600
ECC1
512 bytes
0xFFFF E800
SDRAMC1
512 Bytes
0xFFFF EA00
SMC1
512 Bytes
0xFFFF EC00
MATRIX
512 Bytes
0xFFFF ED10
CCFG
0xFFFF EE00
DBGU
512 Bytes
0xFFFF F000
AIC
512 bytes
0xFFFF F200
PIOA
512 bytes
0xFFFF F400
PIOB
512 Bytes
0xFFFF F600
PIOC
512 bytes
0xFFFF F800
PIOD
512 bytes
0xFFFF FA00
PIOE
512 bytes
0xFFFF FC00
PMC
256 Bytes
0xFFFF FD00
RSTC
16 Bytes
0xFFFF FD10
SHDWC
16 Bytes
0xFFFF FD20
RTT0
16 Bytes
0xFFFF FD30
PIT
16 Bytes
0xFFFF FD40
WDT
16 Bytes
0xFFFF FD50
RTT1
16 Bytes
0xFFFF FD60
GPBR
80 Bytes
0xFFFF FDB0
Reserved
0xFFFF FFFF

Documentation as Source of Registers' Information

31.6.12 PIO Controller Output Data Status Register

Name: PIO_ODSR

Addresses: 0xFFFFF238 (PIOA), 0xFFFFF438 (PIOB), 0xFFFFF638 (PIOC), 0xFFFFF838 (PIOD), 0xFFFFFA38 (PIOE)

Access Type: Read-only or Read-write

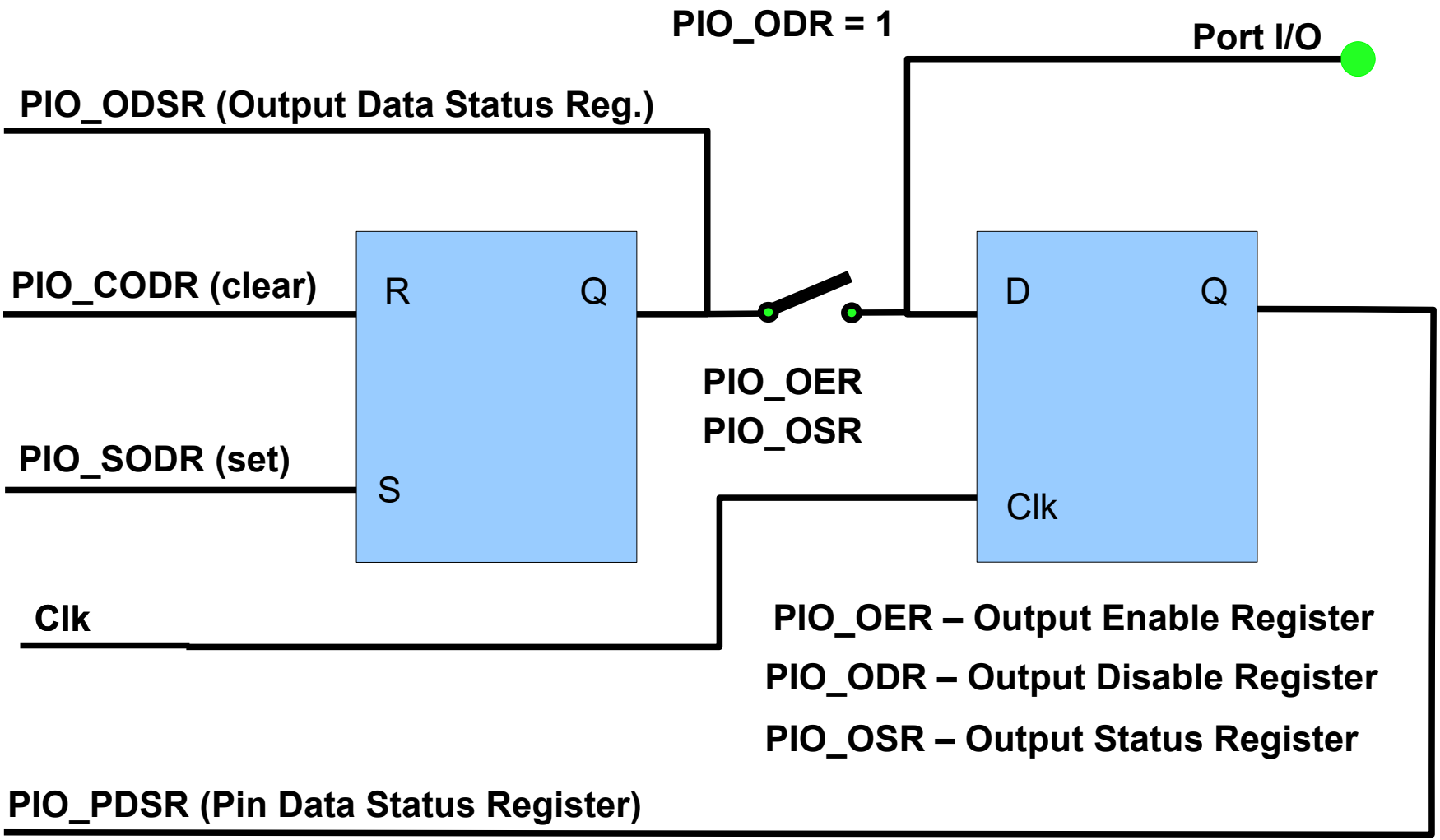
31	30	29	28	27	26	25	24
P31	P30	P29	P28	P27	P26	P25	P24
23	22	21	20	19	18	17	16
P23	P22	P21	P20	P19	P18	P17	P16
15	14	13	12	11	10	9	8
P15	P14	P13	P12	P11	P10	P9	P8
7	6	5	4	3	2	1	0
P7	P6	P5	P4	P3	P2	P1	P0

- **P0-P31: Output Data Status**

0 = The data to be driven on the I/O line is 0.

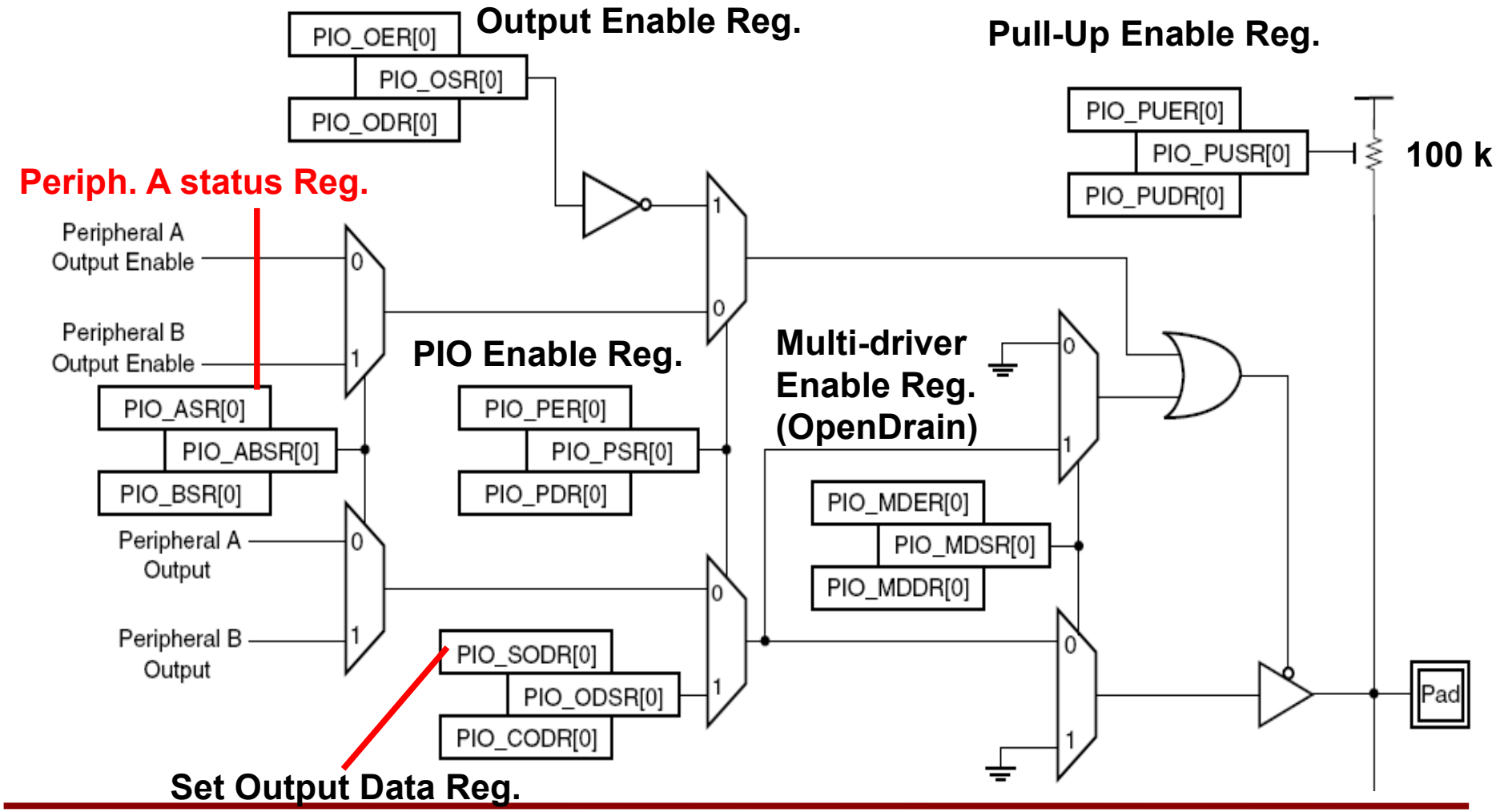
1 = The data to be driven on the I/O line is 1.

Simplified Block Diagram of I/O Port

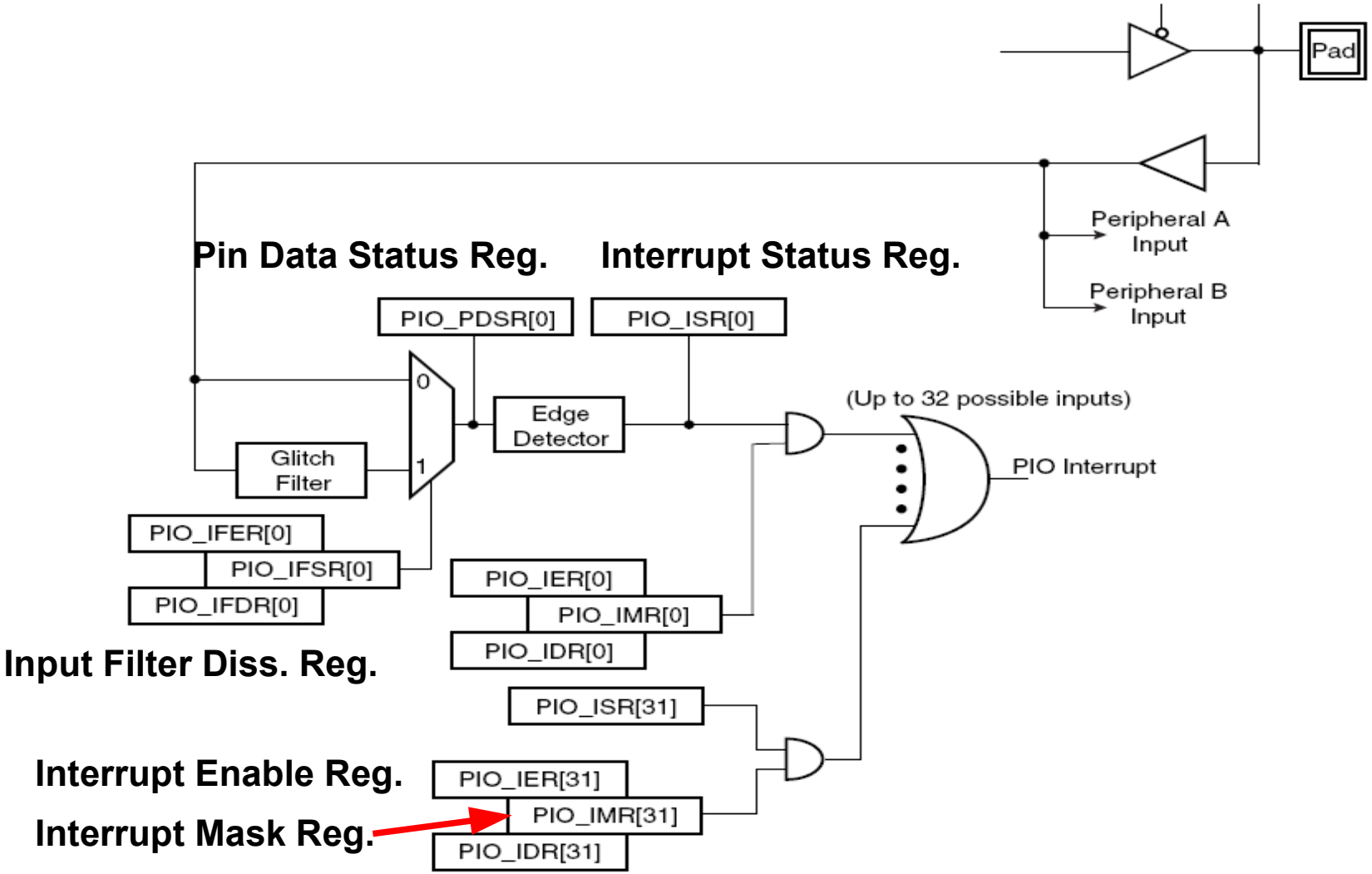


I/O Port – How to Control Output ?

Figure 31-3. I/O Line Control Logic



I/O – How to Read Input ?



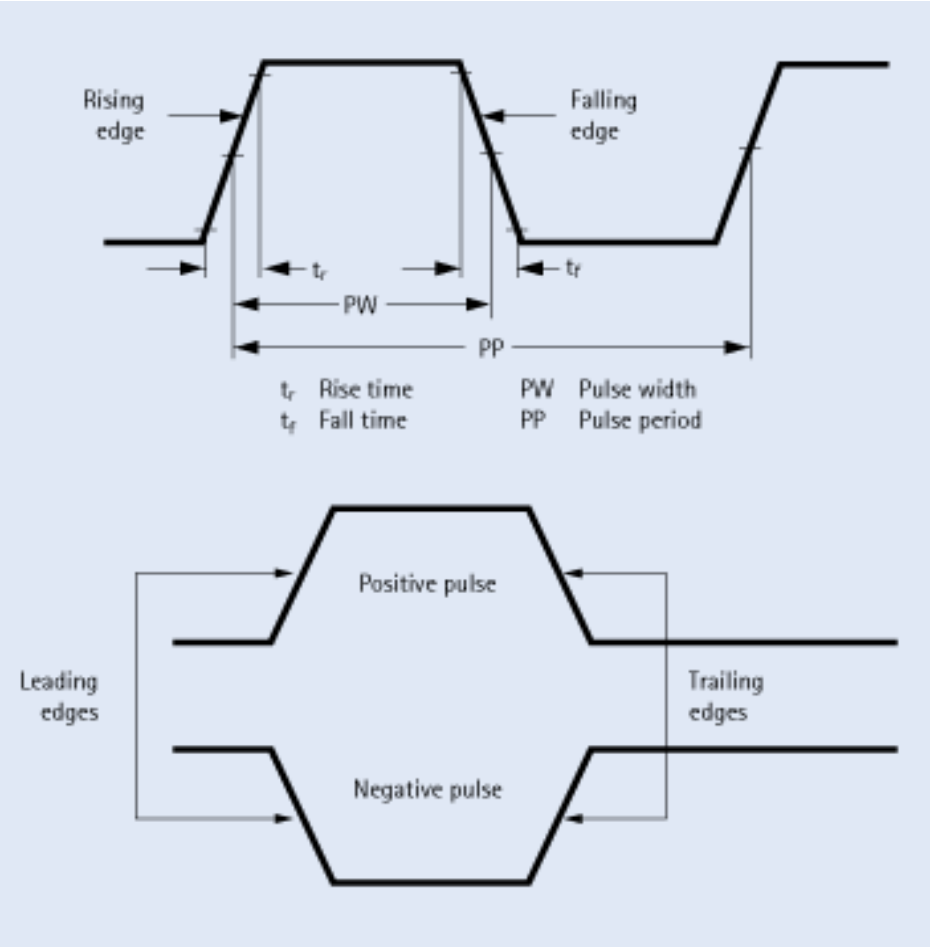
Digital Signal

Digital Signal can be characterised with:

- f – frequency (period),
- A – amplitude.

Digital circuits can be triggered with:

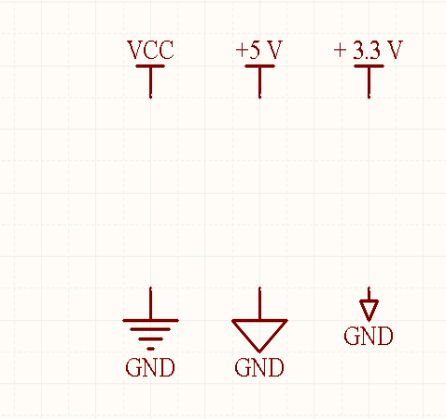
- Change of signal level (lower or higher than signal threshold level),
- Change of signal slope (transaction of digital signal from '0' to '1' or from '1' to '0').



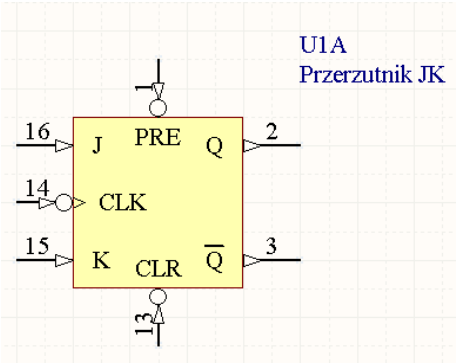
Schematic diagrams

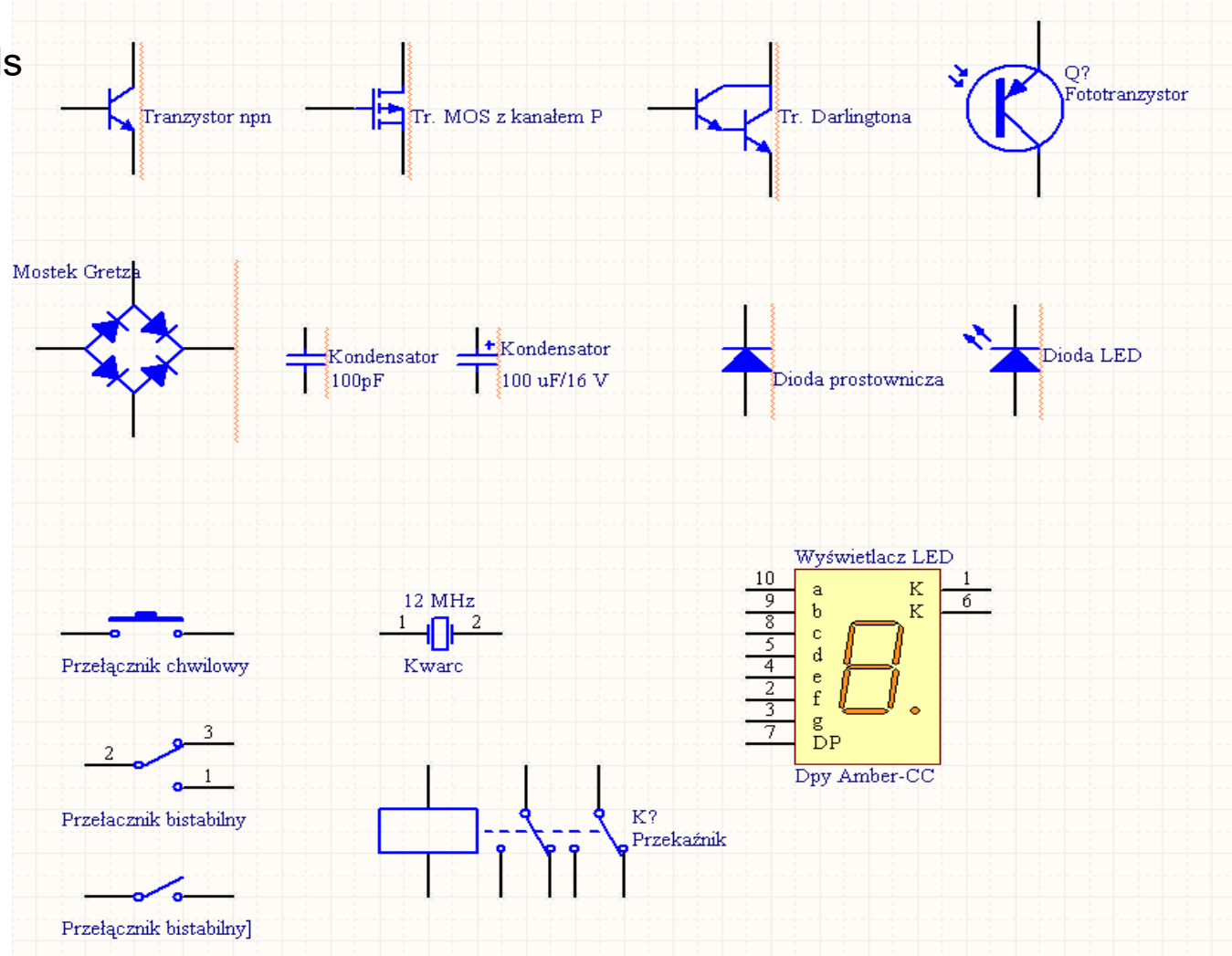
Schematic Diagrams (1)

Power Supply Bus Symbols



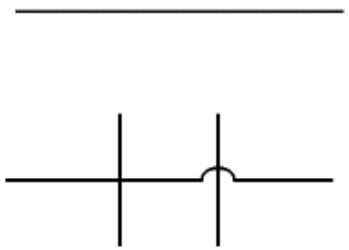
Ground Symbols

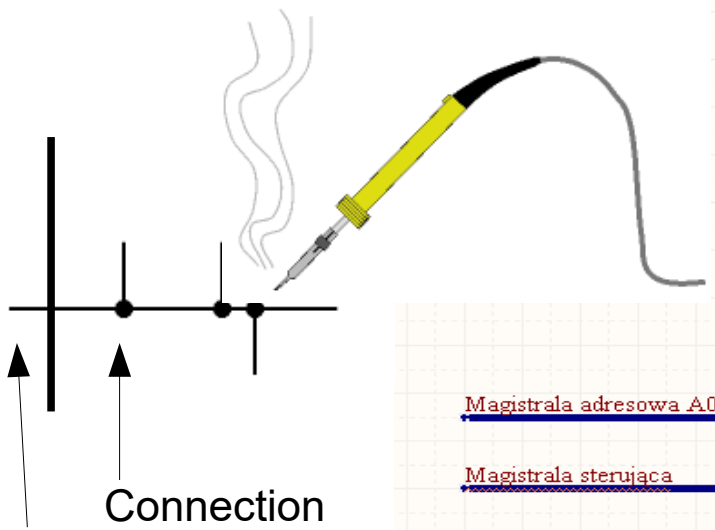




Schematic Diagrams (2)

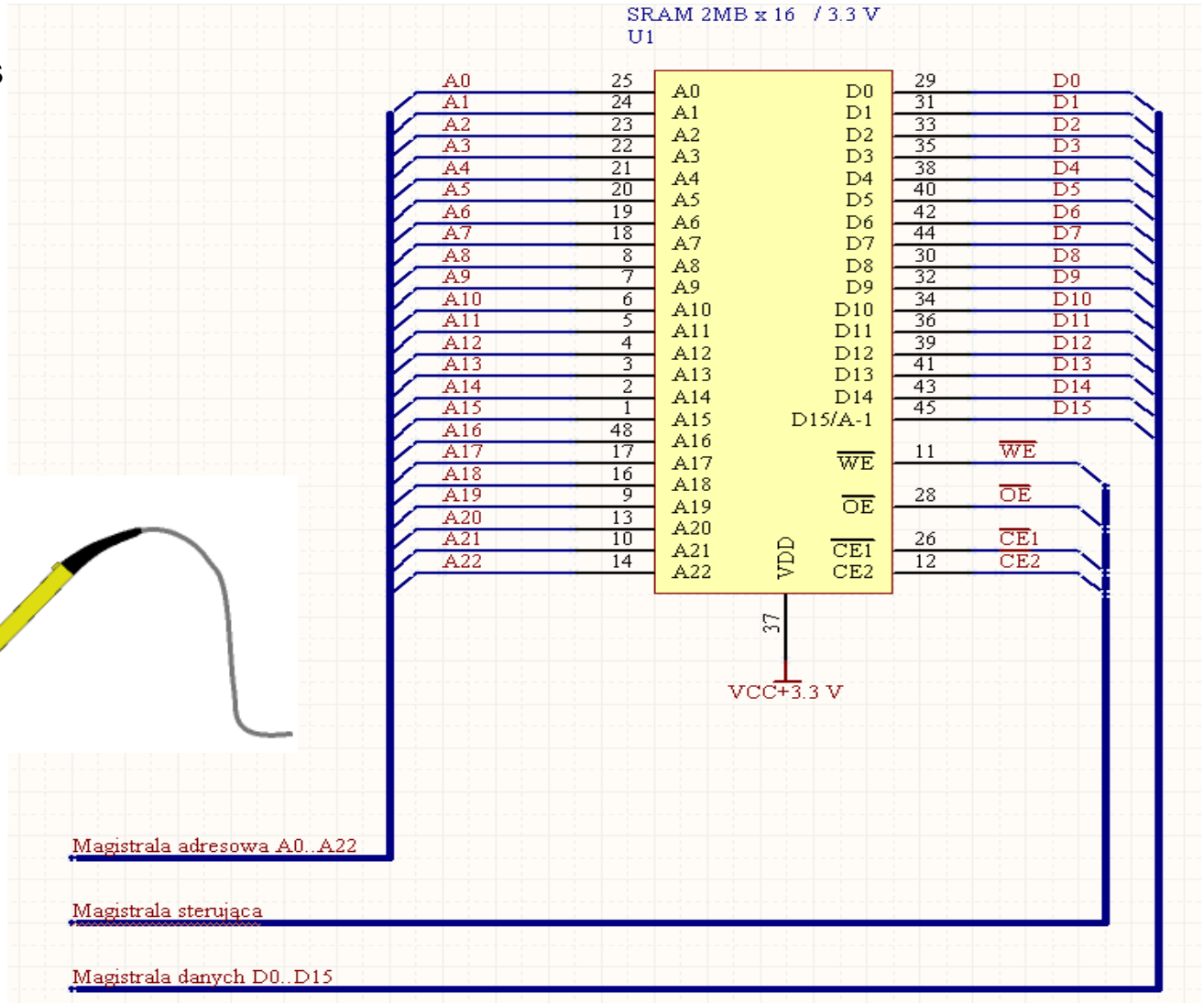
Electrical connections



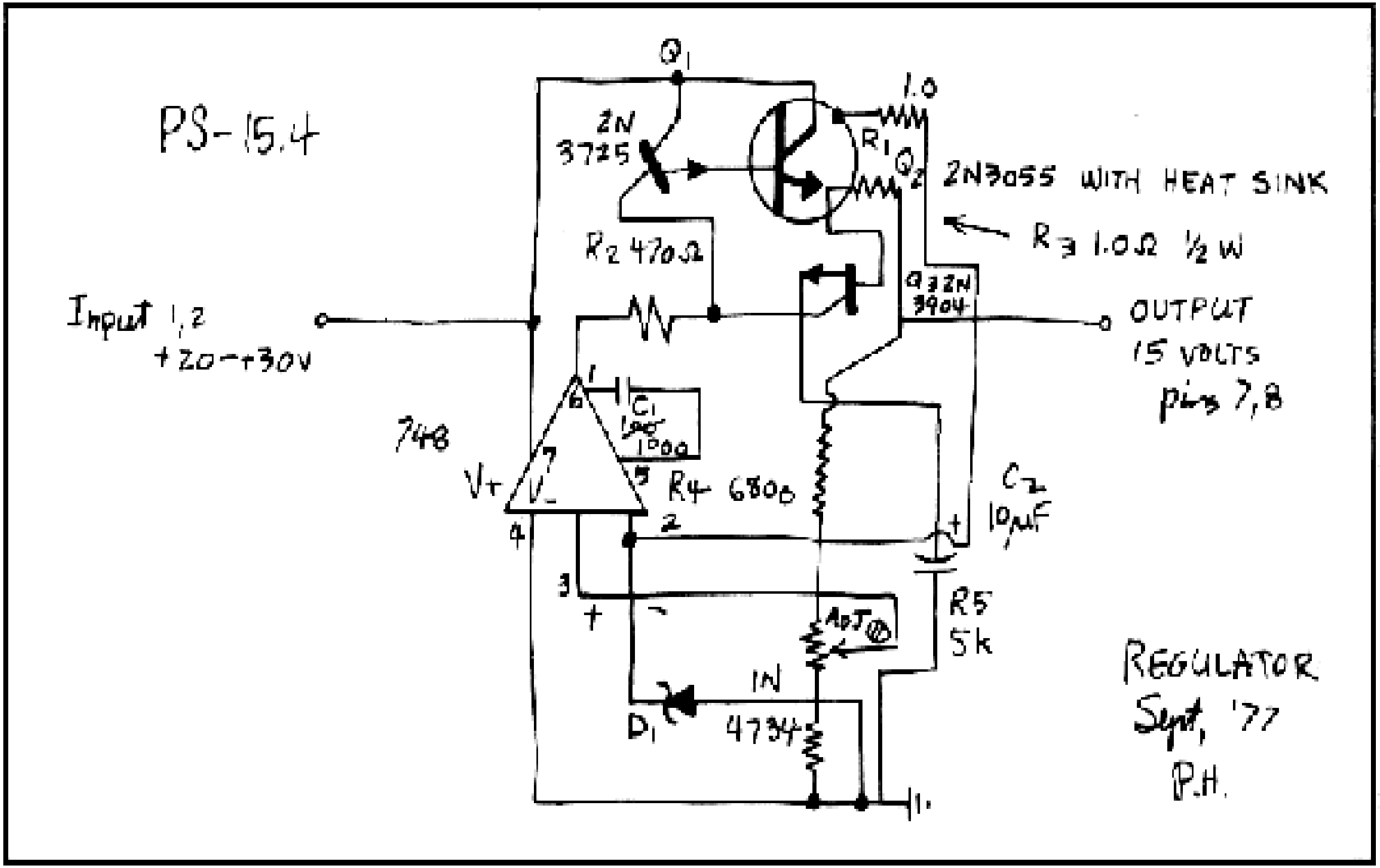


Connection

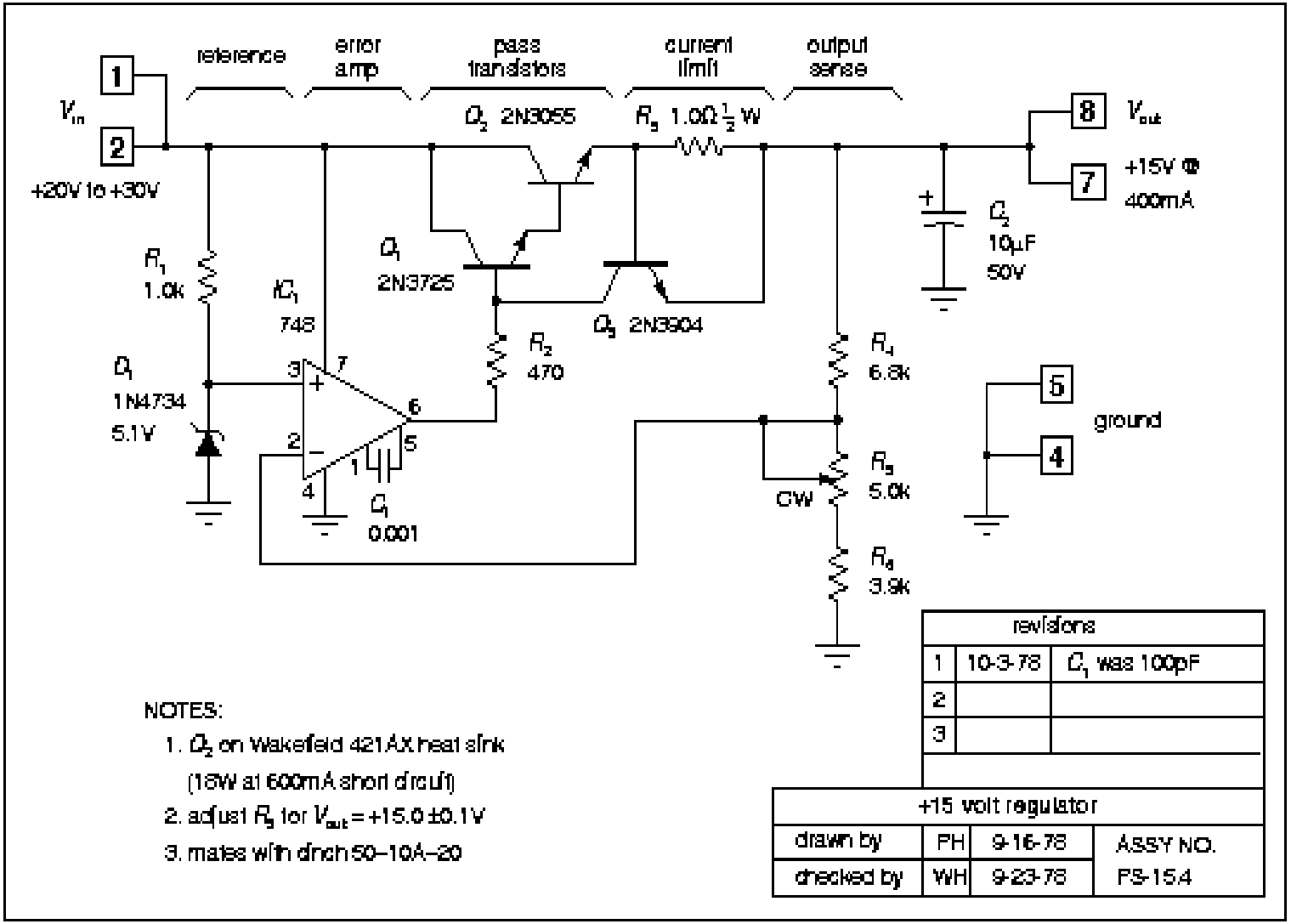
No connection



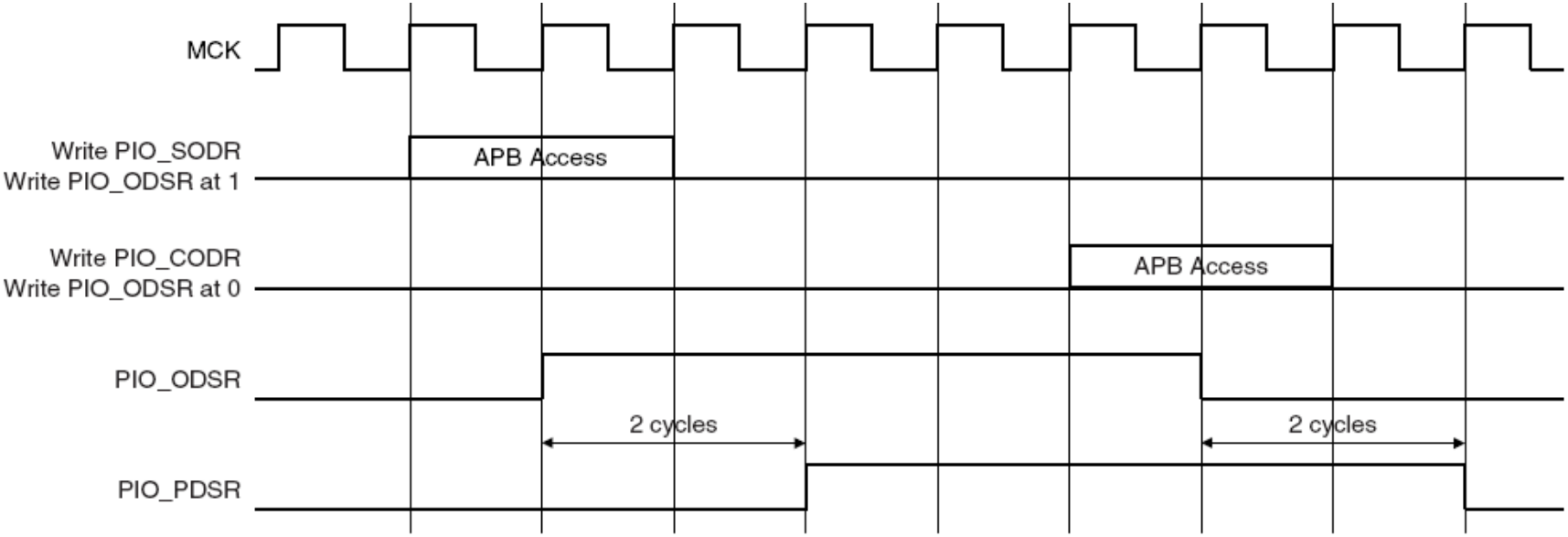
Schematic Diagram – How to Draw ?



Schematic Diagrams – Better Way



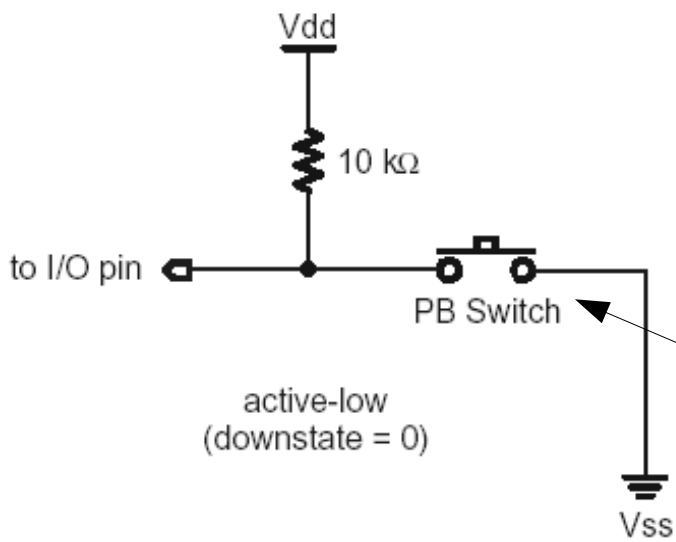
Timing charts during I/O operations



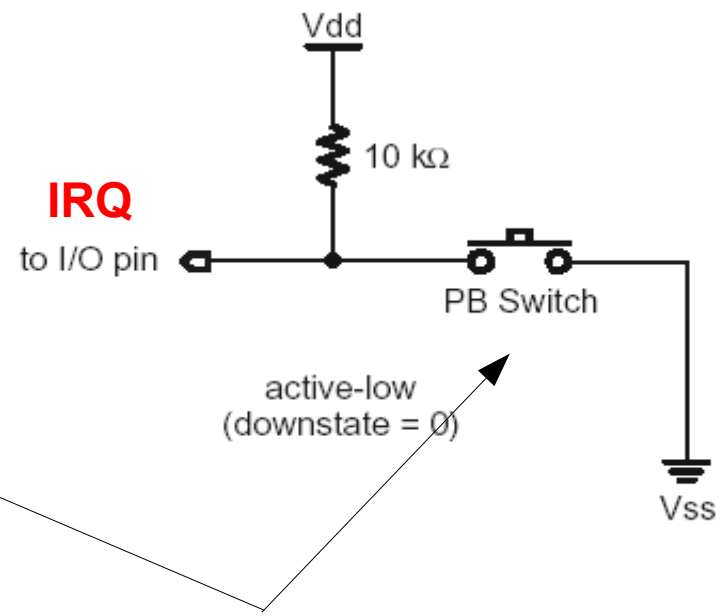
- 1 clock delay, when output driven from registers SODR/CODR,
- 2 clocks delay during access to the whole port (32 bits, set bits of PIO_OWSR register).

Reading state of button

Polling loop



Interrupt



Asynchronous signal

How to Control Clock Signal for Peripheral Devices

PMC Peripheral Clock Enable Register
Register Name:PMC_PCER
Address: 0xFFFFFC10

Table 10-1. AT91SAM9263 Peripheral Identifiers

Peripheral ID	Peripheral Mnemonic	Peripheral Name	External Interrupt
0	AIC	Advanced Interrupt Controller	FIQ
1	SYSC	System Controller Interrupt	
2	PIOA	Parallel I/O Controller A	
3	PIOB	Parallel I/O Controller B	
4	PIOC to PIOE	Parallel I/O Controller C, D and E	
5	reserved		
6	reserved		
7	US0	USART 0	
8	US1	USART 1	
9	US2	USART 2	

```
write_register(PMC_PCER,0x00000110);      // Peripheral clocks 2 and 4 are enabled.
write_register(PMC_PCDR,0x00000010);      // Peripheral clock 2 is disabled.
```

I/O Registers for I/O Ports

```
typedef volatile unsigned int *AT91_REG;          // Hardware register definition
        AT91_REG  PIO_PER = 0xFFFFF200;     // PIO Enable Register, 32-bit register
        AT91_REG  PIO_PDR = 0xFFFFF204;     // PIO Disable Register
        AT91_REG  PIO_PSR = 0xFFFFF208;     // PIO Status Register
        AT91_REG  Reserved0[1]= 0xFFFFF20C; // Filler
        AT91_REG  PIO_OER;                  // Output Enable Register
        AT91_REG  PIO_ODR;                  // Output Disable Registerr
        AT91_REG  PIO_OSR;                  // Output Status Register
        AT91_REG  Reserved1[1];             //
        AT91_REG  PIO_IFER;                 // Input Filter Enable Register
        AT91_REG  PIO_IFDR;                 // Input Filter Disable Register
        AT91_REG  PIO_IFSR;                 // Input Filter Status Register
        AT91_REG  Reserved2[1];             //
        AT91_REG  PIO_SODR;                 // Set Output Data Register
        AT91_REG  PIO_CODR;                 // Clear Output Data Register
        AT91_REG  PIO_ODSR;                 // Output Data Status Register
```

I/O Registers Mapped into Structure (1)

```
typedef volatile unsigned int AT91_REG;      // Hardware register definition
typedef struct _AT91S_PIO {
        AT91_REG  PIO_PER;          // PIO Enable Register, 32-bit register
        AT91_REG  PIO_PDR;          // PIO Disable Register
        AT91_REG  PIO_PSR;          // PIO Status Register
        AT91_REG  Reserved0[1];             //
        AT91_REG  PIO_OER;          // Output Enable Register
        AT91_REG  PIO_ODR;          // Output Disable Registerr
        AT91_REG  PIO_OSR;          // Output Status Register
        AT91_REG  Reserved1[1];             //
        AT91_REG  PIO_IFER;         // Input Filter Enable Register
        AT91_REG  PIO_IFDR;         // Input Filter Disable Register
        AT91_REG  PIO_IFSR;         // Input Filter Status Register
        AT91_REG  Reserved2[1];             //
        AT91_REG  PIO_SODR;         // Set Output Data Register
        AT91_REG  PIO_CODR;         // Clear Output Data Register
        AT91_REG  PIO_ODSR;         // Output Data Status Register
} AT91S_PIO, *AT91PS_PIO;
```

I/O Registers Mapped into Structure (2)

Declartion of a new structure type creates a template for registers mapped on the memory of the processor. A Symbolic name is assigned to each register. The created structure is called according to used processor and functionality defined by registers, e.g. **AT91S_PIO** and ***AT91PS_PIO.**

Lack of information describing access to registers, e.g. access mode R/W, value after reset, offset.

The information can be supplied as a comments in header file.

```
typedef struct _AT91S_PIO {          /*   Register name          R/W    Reset value    Offset
    AT91_REG  PIO_PER;               // PIO Enable Register       W        -          0x00
    AT91_REG  PIO_PDR;               // PIO Disable Register      W        -          0x04
    AT91_REG  PIO_PSR;               // PIO Status Register       R        -          0x08
    AT91_REG  Reserved0[1];                 // memory filler
    AT91_REG  PIO_OER;               // Output Enable Register    W        -          0x10
    AT91_REG  PIO_ODR;               // Output Disable Register   W        -          0x14
    AT91_REG  PIO_OSR;               // Output Status Register    W        -          0x18
    } AT91S_PIO, *AT91PS_PIO
    /* structure describing registers file (block of registers) for I/O ports PIOA...PIOE */
#define AT91C_BASE_PIOA     (AT91PS_PIO)    0xFFFFF200    // (PIOA) Base Address
    /* definition of bit mask for zero bit in port PA */
#define AT91C_PIO_PA0       (1 <<  0)       // Pin Controlled by PA0
```

How can we set 0 and 19 bits of OER register ?

Manipulation on Registers Bits

Save value to register:

```
AT91PS_PIO->PIO_OER = 0x5;
```

Read value from register:

```
volatile unsigned int ReadData;
ReadData = AT91PS_PIO->PIO_OSR;
```

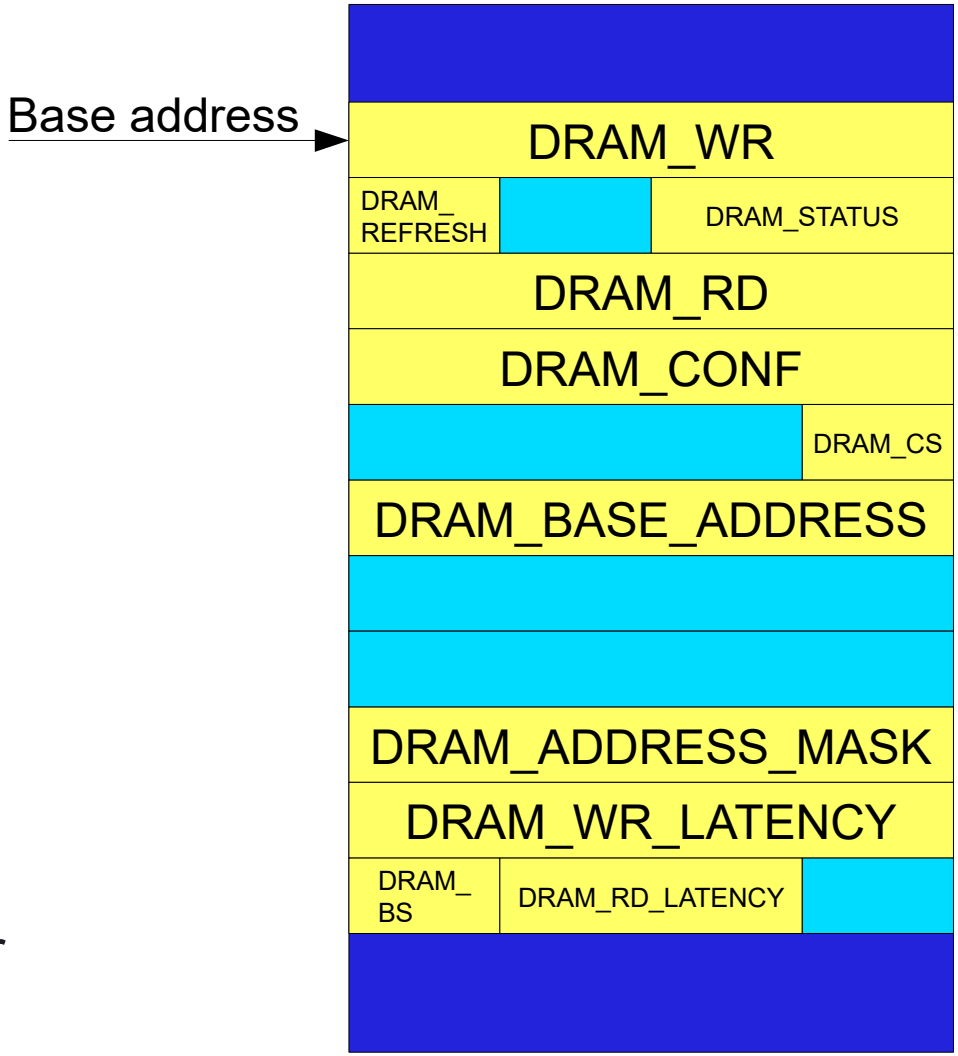
Bit operations:

```
AT91C_BASE_PIOA->ENABLE_REGISTER  = (AT91C_PIO_PA0 | AT91C_PIO_PA19);
AT91C_BASE_PIOA->DISABLE_REGISTER = (AT91C_PIO_PA0 | AT91C_PIO_PA19);
```

How to negate bit ?

Registers mapped into structure - exercise

- Registers of DRAM memory are mapped into memory space,
- Base address: 0xFFFE.2000,
- Registers type: 8, 16, 32 bit,



Task to do:

- Create new struct type for DRAM registers,
- Declare pointer,
- Read, write data from memory,
- Set and clear configuration registers (bit 5, bit 29),
- Check busy flag in status register (bit 9)

Bit-fields – Register Mapped as Structure

```
Struct Port_4bit {
unsigned Bit_0          :    1;
unsigned Bit_1          :    1;
unsigned Bit_2          :    1;
unsigned Bit_3          :    1;
unsigned Bit_Filler     :    4;
};
#define PORTC (*(Port_4bit*)0x4010.0002U)
int i = PORTC.Bit_0;        /* read data */
PORTC.Bit_2 = 1;            /* write data */

Port_4bit* PortTC = (Port_4bit*) 0x4010.000FU;
int i = PortTC->Bit_0;
PortTC->Bit_0 = 1;
```

- Bit-fields allows to 'pack' data – usage of single bits, e.g. bit flags
- Increase of code complexity required for operations on registers
- Bit-fields can be mapped in different ways in memory according different compilers and processors architectures
- Cannot use ***offsetof*** macro to calculate data offset in structure
- Cannot use ***sizeof*** macro to calculate size of data
- Tables cannot use bit-fields

Union – Registers With Different Functionalities

```
extern volatile union {
        struct {
                unsigned EID16      :1;
                unsigned EID17      :1;
                unsigned            :1;
                unsigned EXIDE      :1;
                unsigned            :1;
                unsigned SID0       :1;
                unsigned SID1       :1;
                unsigned SID2       :1;
        };
        struct {
                unsigned            :3;
                unsigned EXIDEN     :1;
        };
} RXF3SIDLbits_;
```

Structures have the same address:

```
#define RXF3SIDLbits
    (*(Port_RXF3SIDLbits_*)0x4010.0000)
```

Access to data mapped into structure:

```
/* data in first structure */
        RXF3SIDLbits.EID16 = 1;

/* data in second structure */
        RXF3SIDLbits.EXIDEN = 0;
```

Example of Control Register – Real-time Timer

15.4.1 Real-time Timer Mode Register

Register Name: RTT_MR

Addresses: 0xFFFFFD20 (0), 0xFFFFFD50 (1)

Access Type: Read/Write

31	30	29	28	27	26	25	24
–	–	–	–	–	–	–	–

23	22	21	20	19	18	17	16
–	–	–	–	–	RTTRST	RTTINCIEN	ALMIEN

15	14	13	12	11	10	9	8
RTPRES							

7	6	5	4	3	2	1	0
RTPRES							

- **ALMIEN: Alarm Interrupt Enable**

0 = The bit ALMS in RTT_SR has no effect on interrupt.

1 = The bit ALMS in RTT_SR asserts interrupt.

- **RTTINCIEN: Real-time Timer Increment Interrupt Enable**

0 = The bit RTTINC in RTT_SR has no effect on interrupt.

1 = The bit RTTINC in RTT_SR asserts interrupt.

- **RTTRST: Real-time Timer Restart**

1 = Reloads and restarts the clock divider with the new programmed value. This also resets the 32-bit counter.

// -------- RTTC_RTMR : (RTTC Offset: 0x0) Real-time Mode Register --------

```
#define AT91C_RTTC_RTPRES       (0xFFFF <<  0)    // (RTTC) Real-time Timer Prescaler Value
#define AT91C_RTTC_ALMIEN       (0x1 << 16)       // (RTTC) Alarm Interrupt Enable
#define AT91C_RTTC_RTTINCIEN    (0x1 << 17)       // (RTTC) Real Time Timer Increment Interrupt Enable
#define AT91C_RTTC_RTTRST       (0x1 << 18)       // (RTTC) Real Time Timer Restart
```

Registers Definition – Header Files (1)

```
#ifndef _PROJECT_H
#define _PROJECT_H
/*
 * Include your AT91 Library files and specific
 * compiler definitions
 */
#include "AT91SAM9263-EK.h"
#include "AT91SAM9263.h"
#endif  // _PROJECT_H
```

```
/*-------------------------*/
/*  LEDs Definition  */
/*-------------------------*/
#define AT91B_LED1                  AT91C_PIO_PB8   /* DS1 */
#define AT91B_LED2                  AT91C_PIO_PC29  /* DS2 */
#define AT91B_NB_LEB        2
#define AT91D_BASE_PIO_LED1         (AT91C_BASE_PIOB)
#define AT91D_BASE_PIO_LED2         (AT91C_BASE_PIOC)
#define AT91D_ID_PIO_LED1           (AT91C_ID_PIOB)
#define AT91D_ID_PIO_LED2           (AT91C_ID_PIOC)
```

```
/*--------------------------------*/
/* Push Button Definition  */
/*--------------------------------*/
#define AT91B_BP1       AT91C_PIO_PC5  // Left click
#define AT91B_BP2       AT91C_PIO_PC4  // Right click
#define AT91D_BASE_PIO_BP       AT91C_BASE_PIOC
#define AT91D_ID_PIO_BP         AT91C_ID_PIOCDE
```

Registers Definition – Header Files (2)

```
/*-----------------------*/
/*  LEDs Definition  */
/*-----------------------*/

#define AT91B_LED1                AT91C_PIO_PB8   /* DS1 */
#define AT91B_LED2                AT91C_PIO_PC29  /* DS2 */
#define AT91B_NB_LEB        2
#define AT91D_BASE_PIO_LED1       (AT91C_BASE_PIOB)
#define AT91D_BASE_PIO_LED2       (AT91C_BASE_PIOC)
#define AT91D_ID_PIO_LED1         (AT91C_ID_PIOB)
#define AT91D_ID_PIO_LED2         (AT91C_ID_PIOC)

/*-------------------------------*/
/* Push Button Definition  */
/*-------------------------------*/

#define AT91B_BP1       AT91C_PIO_PC5  // Left click
#define AT91B_BP2       AT91C_PIO_PC4  // Right click
#define AT91D_BASE_PIO_BP       AT91C_BASE_PIOC
#define AT91D_ID_PIO_BP         AT91C_ID_PIOCDE
```

#define AT91C_PIO_PB8 (1 << 8) // Pin Controlled by PB8

#define AT91C_PIO_PC29 (1 << 29) // Pin Controlled by PC29

#define AT91C_BASE_PIOB (AT91_CAST(AT91PS_PIO) 0xFFFFF400) // (PIOB) Base Address

#define AT91C_BASE_PIOC (AT91_CAST(AT91PS_PIO) 0xFFFFF600) // (PIOC) Base Address

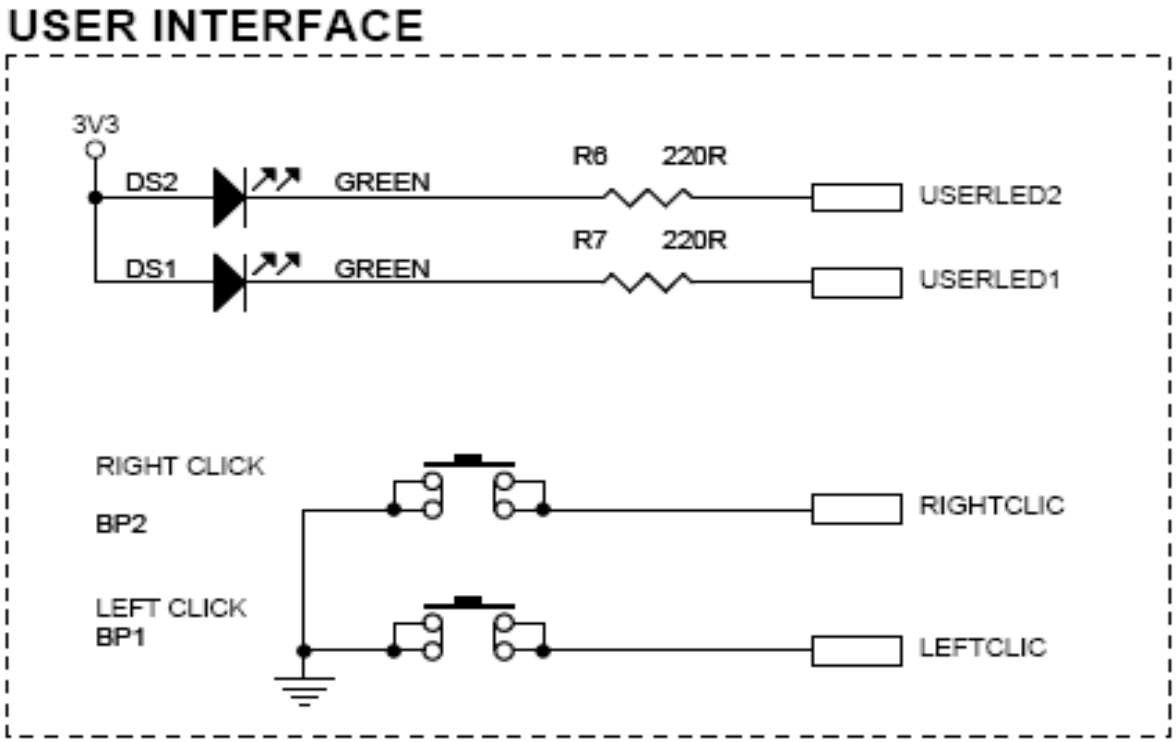
#define AT91C_ID_PIOB (3) // Parallel IO Controller B

#define AT91C_PIO_PC4 (1 << 4) // Pin Controlled by PC4

#define AT91C_PIO_PC5 (1 << 5) // Pin Controlled by PC5

#define AT91C_ID_PIOCDE (4) // Parallel IO Controller C, Parallel IO Controller D, Parallel IO Controller E

ATMEL Development Board – LEDs, Buttons




```
#define AT91B_LED1      AT91C_PIO_PB8   /* DS1 */
#define AT91B_LED2      AT91C_PIO_PC29  /* DS2 */
#define AT91B_BP1       AT91C_PIO_PC5   // Left click
#define AT91B_BP2       AT91C_PIO_PC4   // Right clic
```

Configuration of I/O ports

```
#define AT91C_PIO_PB8        (1U <<  8)                    // Pin Controlled by PB8
#define AT91C_BASE_PIOB   (AT91PS_PIO)  0xFFFF.F400U       // (PIOB) Base Address
```

Input mode:

```
/* Enable the peripheral clock for the PIO controller, This is mandatory when PIO are configured as input */
AT91C_BASE_PMC->PMC_PCER = (1 << AT91C_ID_PIOCDE ); // peripheral clock enable register (port C, D, E)
/* Set the PIO line in input */
AT91C_BASE_PIOD->PIO_ODR = 0x0000.000FU;            // 1 – Set direction of the pin to input
/* Set the PIO controller in PIO mode instead of peripheral mode */
AT91C_BASE_PIOD->PIO_PER = AT91C_PIO_PB8;           // 1 – Enable PIO to control the pin
```

Output mode:

```
/* Configure the pin in output */
AT91C_BASE_PIOB->PIO_OER = AT91C_PIO_PB8 ;
/* Set the PIO controller in PIO mode instead of peripheral mode */
AT91C_BASE_PIOD->PIO_PER = 0xFFFF.FFFFU;            // 1 – Enable PIO to control the pin
AT91C_BASE_PIOE->PIO_PER = AT91C_PIO_PB31;
/* Disable pull-up */
AT91C_BASE_PIOA->PIO_PPUDR = 0xFFFF.0000U;          // 1 – Disable the PIO pull-up resistor
```

Time in processor systems

How can We Measure Time ?

- Generate defined delay ?
- Generate date and time ?
- Measure length of pulses ?
- Delay in Real-Time systems ?

Crystal Clock...

- Quartz from chemical point of view is a compound called silicon dioxide. Properly cut and mounted crystal of quartz can be made to vibrate, or oscillate, using an alternating electric current. The frequency at which the crystal oscillates is dependent on its shape and size, and the positions at which electrodes are placed on it. If the crystal is accurately shaped and positioned, it will oscillate at a desired frequency; in clocks and watches, the frequency is usually 32,768 Hz, as a crystal for this frequency is conveniently small. Such a crystals are usually used in digital systems.

Timers

Timer – peripheral device of processor dedicated for time measurement (counting single processor cycles). Flag is marked or interrupt is triggered when timer counter reaches threshold level. Timers are used as a system time source. They can be used to generate delays, switch threads, generate events, etc...

Example of different Timers:

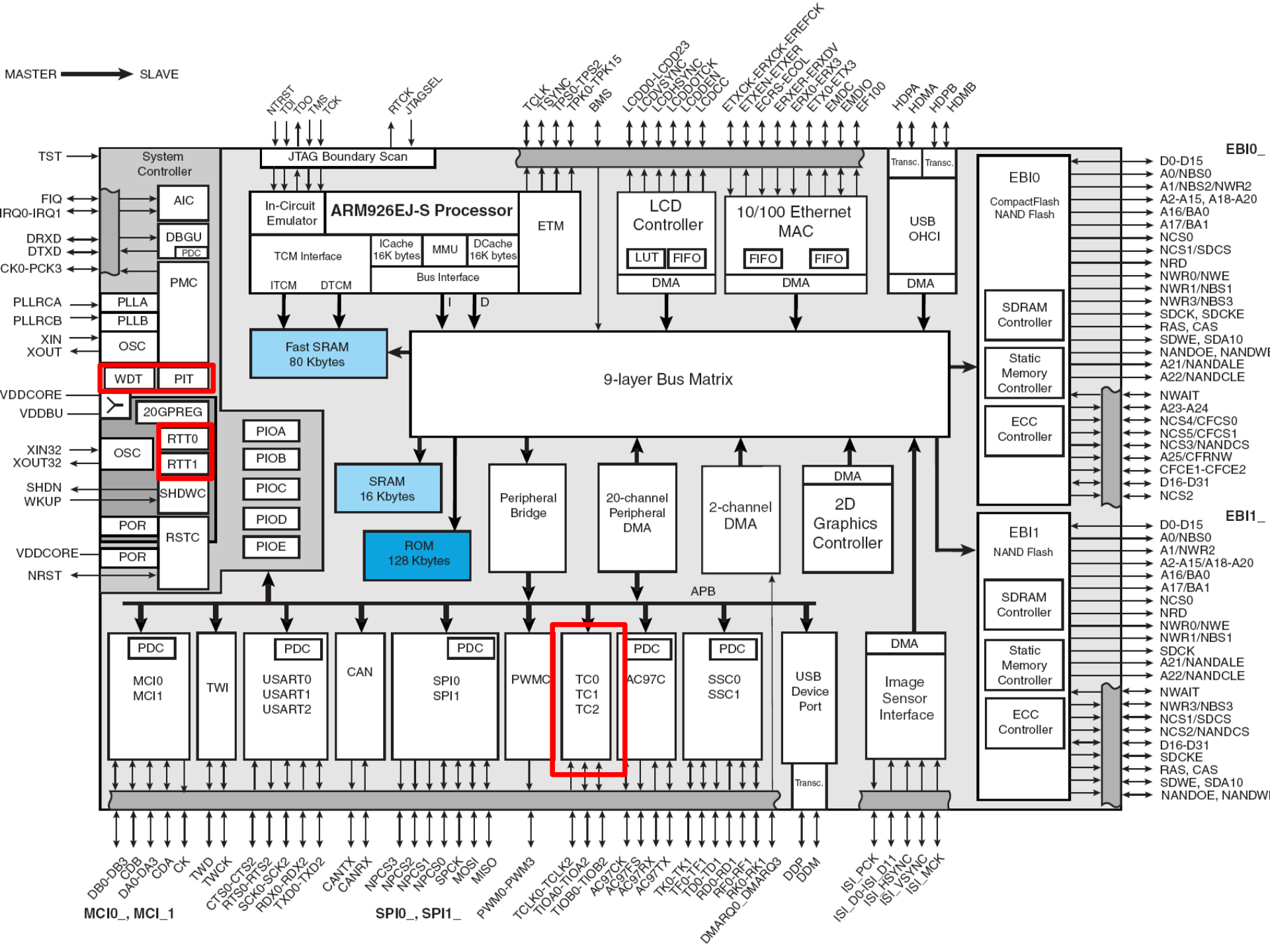
PIT Timer (Periodic Interval Timer, Programmable Interrupt Timer),

RTT Timer (Real-Time Timer),

PWM Timer (Pulse Width Modulation),

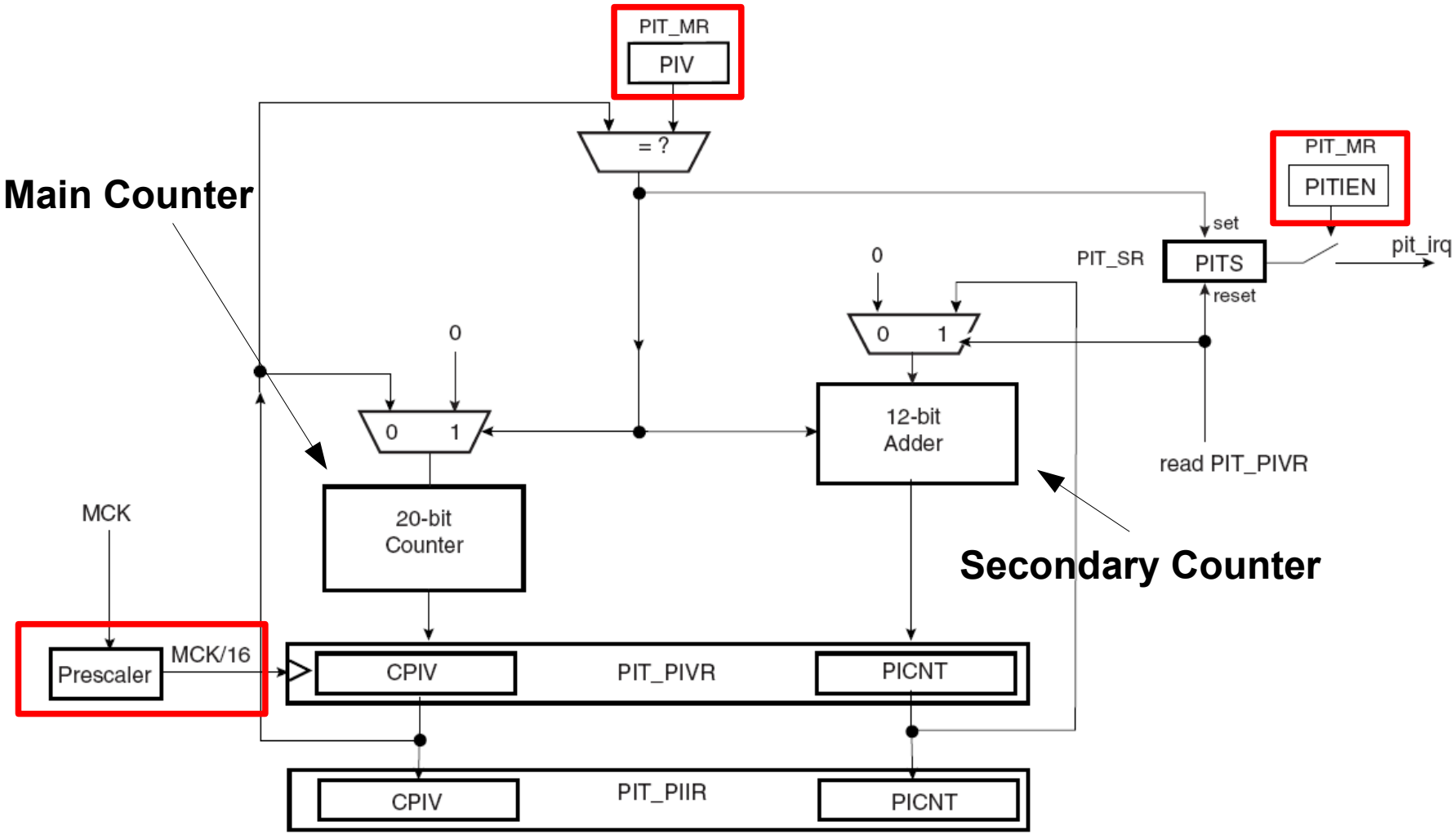
TC Timer (Timer Counter),

WDT Timer (Watch-dog).

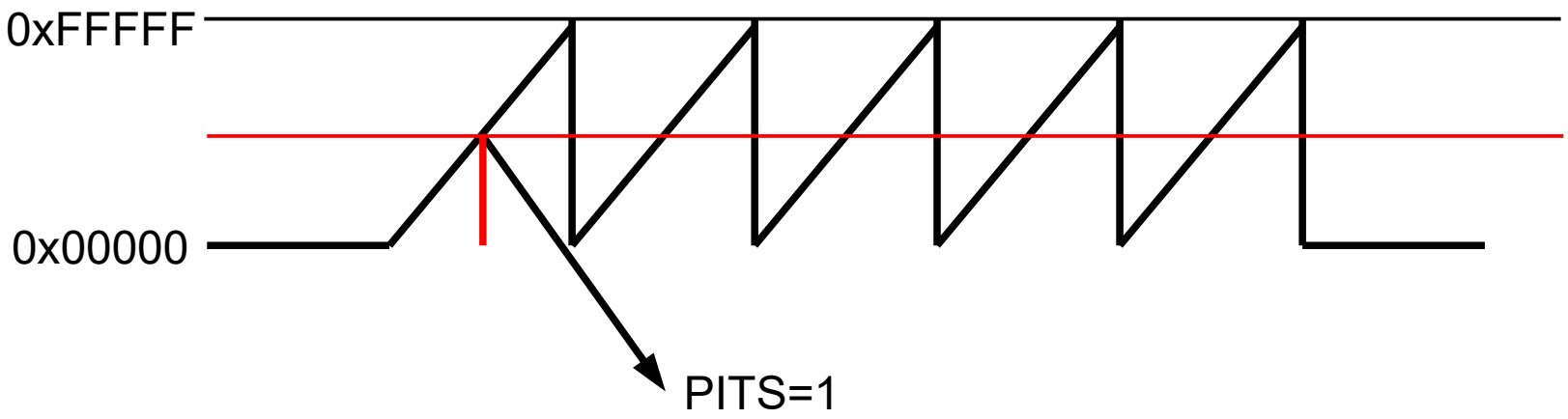

MASTER
SLAVE
System Controller
JTAG Boundary Scan
In-Circuit Emulator
ARM926EJ-S Processor
ETM
TCM Interface
ICache 16K bytes
MMU
DCache 16K bytes
Bus Interface
ITCM
DTCM
LCD Controller
LUT
FIFO
DMA
10/100 Ethernet MAC
FIFO
FIFO
DMA
Transc.
Transc.
USB OHCI
DMA
EBI0
CompactFlash NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
EBI1
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
AIC
DBGU
PDC
PMC
PLLA
PLLB
OSC
WDT
PIT
20GPREG
RTT0
RTT1
OSC
SHDWC
POR
RSTC
POR
PIOA
PIOB
PIOC
PIOD
PIOE
Fast SRAM 80 Kbytes
9-layer Bus Matrix
SRAM 16 Kbytes
ROM 128 Kbytes
Peripheral Bridge
20-channel Peripheral DMA
2-channel DMA
DMA
2D Graphics Controller
APB
PDC
MCI0 MCI1
TWI
PDC
USART0 USART1 USART2
CAN
PDC
SPI0 SPI1
PWMC
TC0 TC1 TC2
PDC
AC97C
PDC
SSC0 SSC1
USB Device Port
Transc.
DMA
Image Sensor Interface
TST
FIQ
IRQ0-IRQ1
DRXD
DTXD
PCK0-PCK3
PLLRCA
PLLRCB
XIN
XOUT
VDDCORE
VDDBU
XIN32
XOUT32
SHDN
WKUP
VDDCORE
NRST
NTRST TDI TDO TMS TCK
RTCK JTAGSEL
TCLK TSYNC TPS0-TPS2 TPK0-TPK15 BMS
LCDD0-LCDD23 LCDVSYNC LCDHSYNC LCDDOTCK LCDDEN LCDCC
ETXCK-ERXCK-EREFCK ETXEN-ETXER ECRS-ECOL ERXER-ERXDV ERX0-ERX3 ETX0-ETX3 EMDC EMDIO EF100
HDPA HDMA HDPB HDMB
EBI0_
D0-D15
A0/NBS0
A1/NBS2/NWR2
A2-A15, A18-A20
A16/BA0
A17/BA1
NCS0
NCS1/SDCS
NRD
NWR0/NWE
NWR1/NBS1
NWR3/NBS3
SDCK, SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
A21/NANDALE
A22/NANDCLE
NWAIT
A23-A24
NCS4/CFCS0
NCS5/CFCS1
NCS3/NANDCS
A25/CFRNW
CFCE1-CFCE2
D16-D31
NCS2
EBI1_
D0-D15
A0/NBS0
A1/NWR2
A2-A15/A18-A20
A16/BA0
A17/BA1
NCS0
NRD
NWR0/NWE
NWR1/NBS1
SDCK
A21/NANDALE
A22/NANDCLE
NWAIT
NWR3/NBS3
NCS1/SDCS
NCS2/NANDCS
D16-D31
SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
DB0-DB3 CDB DA0-DA3 CDA CK TWD TWCK CTS0-CTS2 RTS0-RTS2 SCK0-SCK2 RDX0-RDX2 TXD0-TXD2 CANTX CANRX NPCS3 NPCS2 NPCS1 NPCS0 SPCK MOSI MISO PWM0-PWM3 TCLK0-TCLK2 TIOA0-TIOA2 TIOB0-TIOB2 AC97CK AC97FS AC97RX AC97TX TK0-TK1 TF0-TF1 TD0-TD1 RD0-RD1 RF0-RF1 RK0-RK1 DMARQ0_DMARQ3 DDP DDM ISI_PCK ISI_D0-ISI_D11 ISI_HSYNC ISI_VSYNC ISI_MCK
MCI0_, MCI_1
SPI0_, SPI1_

Periodic Interval Timer

Block Diagram of PIT



Automatic Reload of Timer

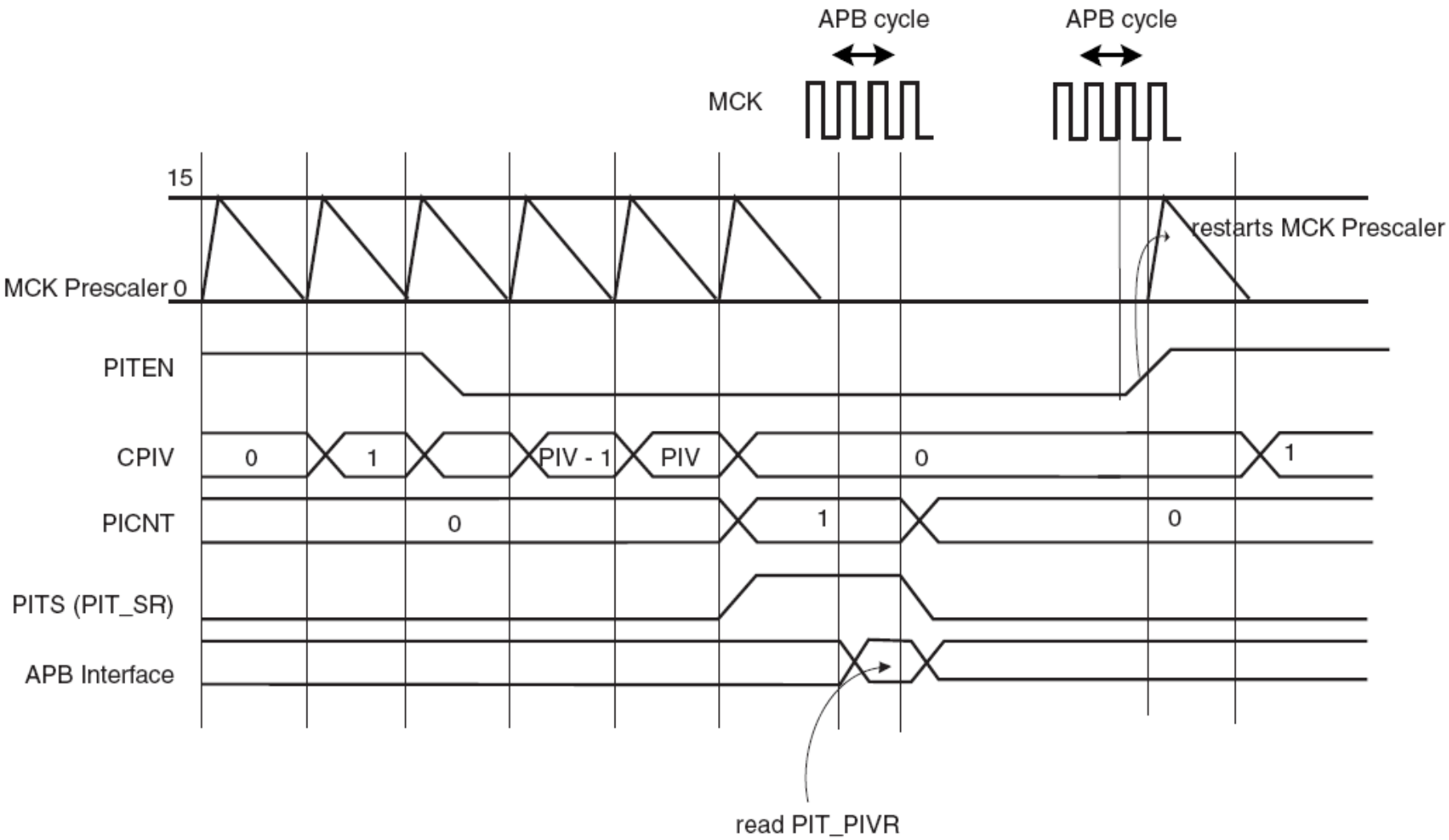


Period of generated interrupts:

(PIV_VALUE+1)*16 / Clk

Clk = 100 MHz, PIV = 62500 => t_{PIT} = 10 ms

PIT in operation



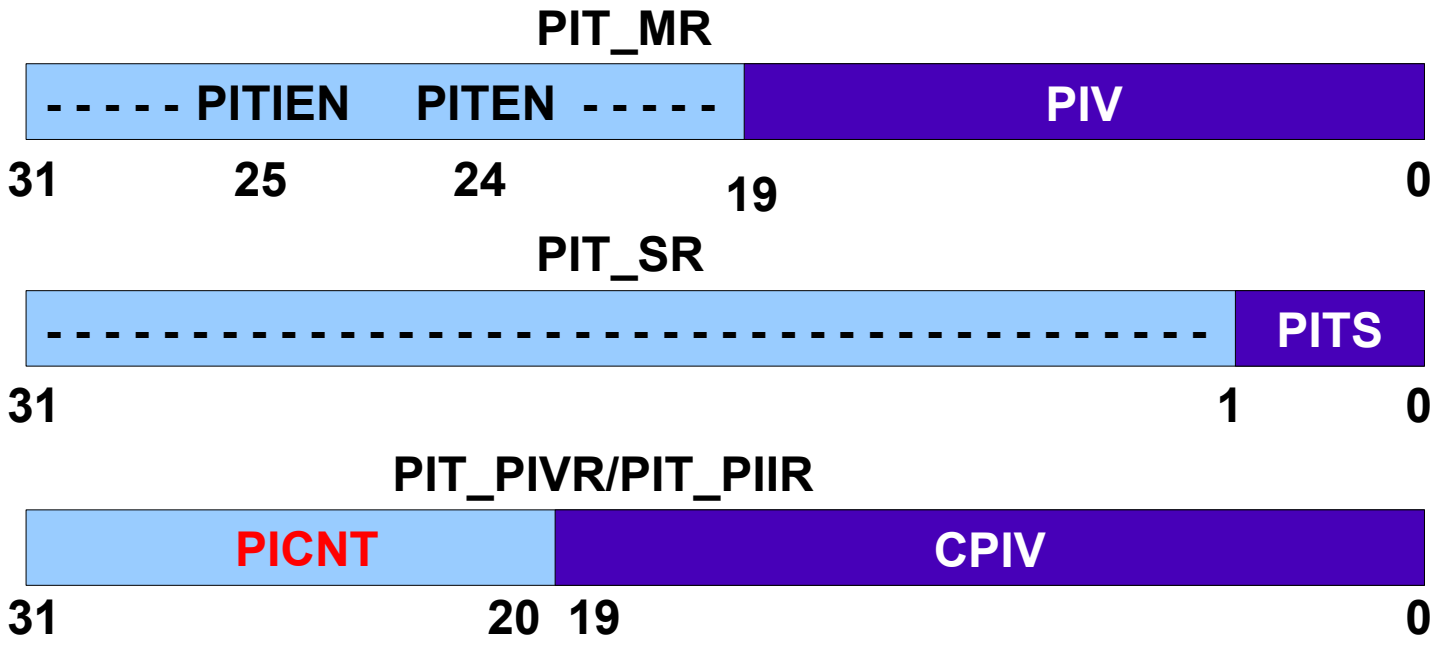
Registers of PIT

Table 16-1. Register Mapping

Offset	Register	Name	Access	Reset
0x00	Mode Register **Address:** 0xFFFFFD30	PIT_MR	Read-write	0x000F_FFFF
0x04	Status Register	PIT_SR	Read-only	0x0000_0000
0x08	Periodic Interval Value Register	PIT_PIVR	Read-only	0x0000_0000
0x0C	Periodic Interval Image Register	PIT_PIIR	Read-only	0x0000_0000

```
typedef struct S_PIT {          /*  Register name          R/W   Reset val.    Offset
AT91_REG  PIT_MR;               // PIT Mode Register       R/W   0x000F.FFFF   0x00
AT91_REG  PIT_SR;               // PIT Status Register     R     0x0000.0000   0x04
AT91_REG  PIT_PIVR;             // PIT Per. Int. Val. Reg. R     0x0000.0000   0x08
AT91_REG  PIT_PIIR;             // PIT Per. Int. Image Reg. R    0x0000.0000   0x0C
}  S_PIT, *PS_PIT;
/* Block of PIT registers */
#define PIT     ((PS_PIT)    0xFFFFFD30)    // (PIT) Base Address
```

PIT registers



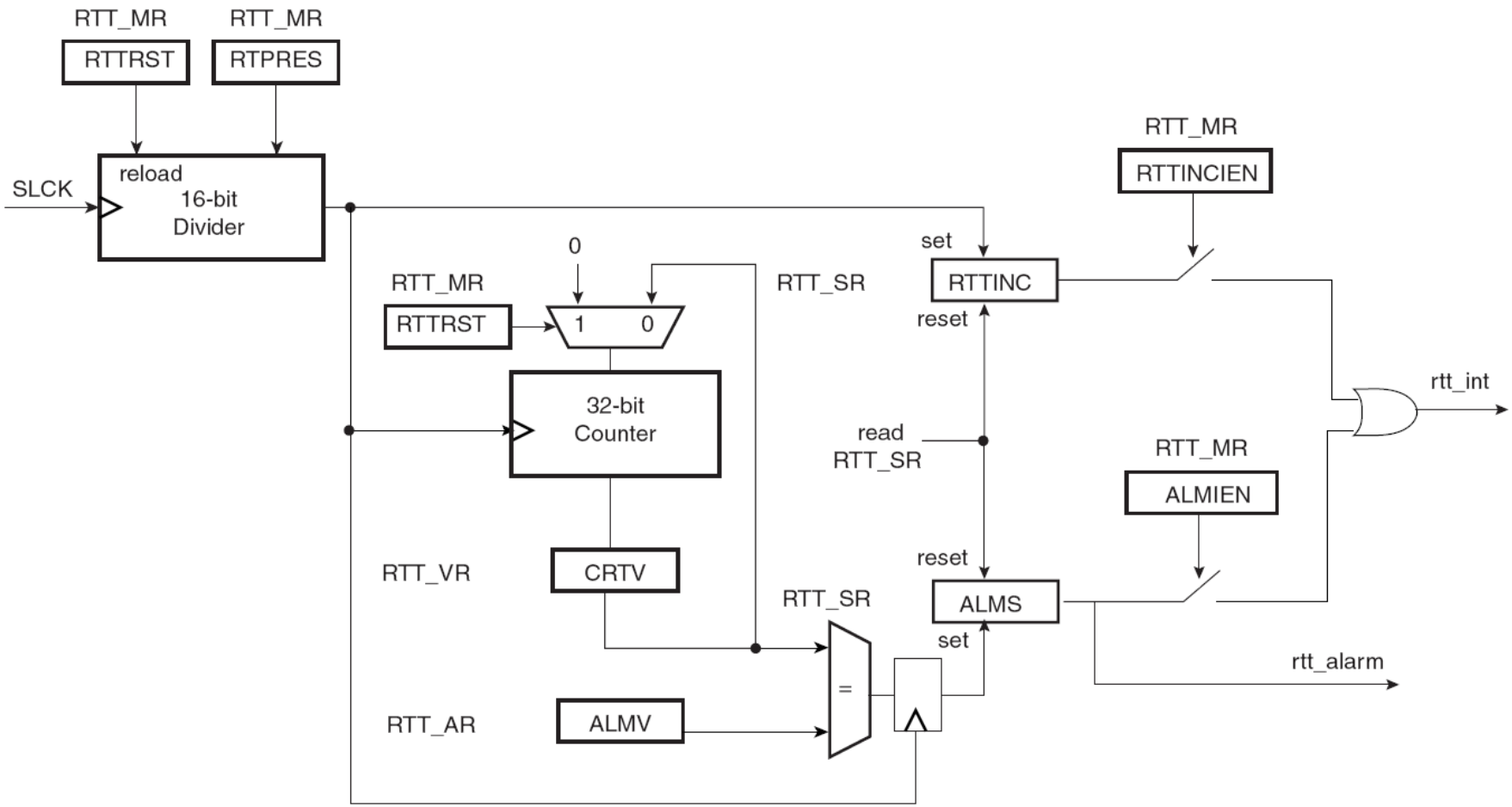
Real Time Timer

Real Time Timer (RTT) is used to measure longer periods of time than PIT timer.

Features of RTT:

- 32-bit down counter and programmable 16 bit divider,
- Can be used to measure elapsed seconds,
  - triggered with slow clock (32.768 kHz),
  - 1s increment with a typical slow clock of 32.768kHz,
  - count up to maximum 136 years (for 1 Hz clock signal),
- Alarm can generate an interrupt,
- Additional interrupt when main timer is increased by one.

Real Time Timer – block diagram



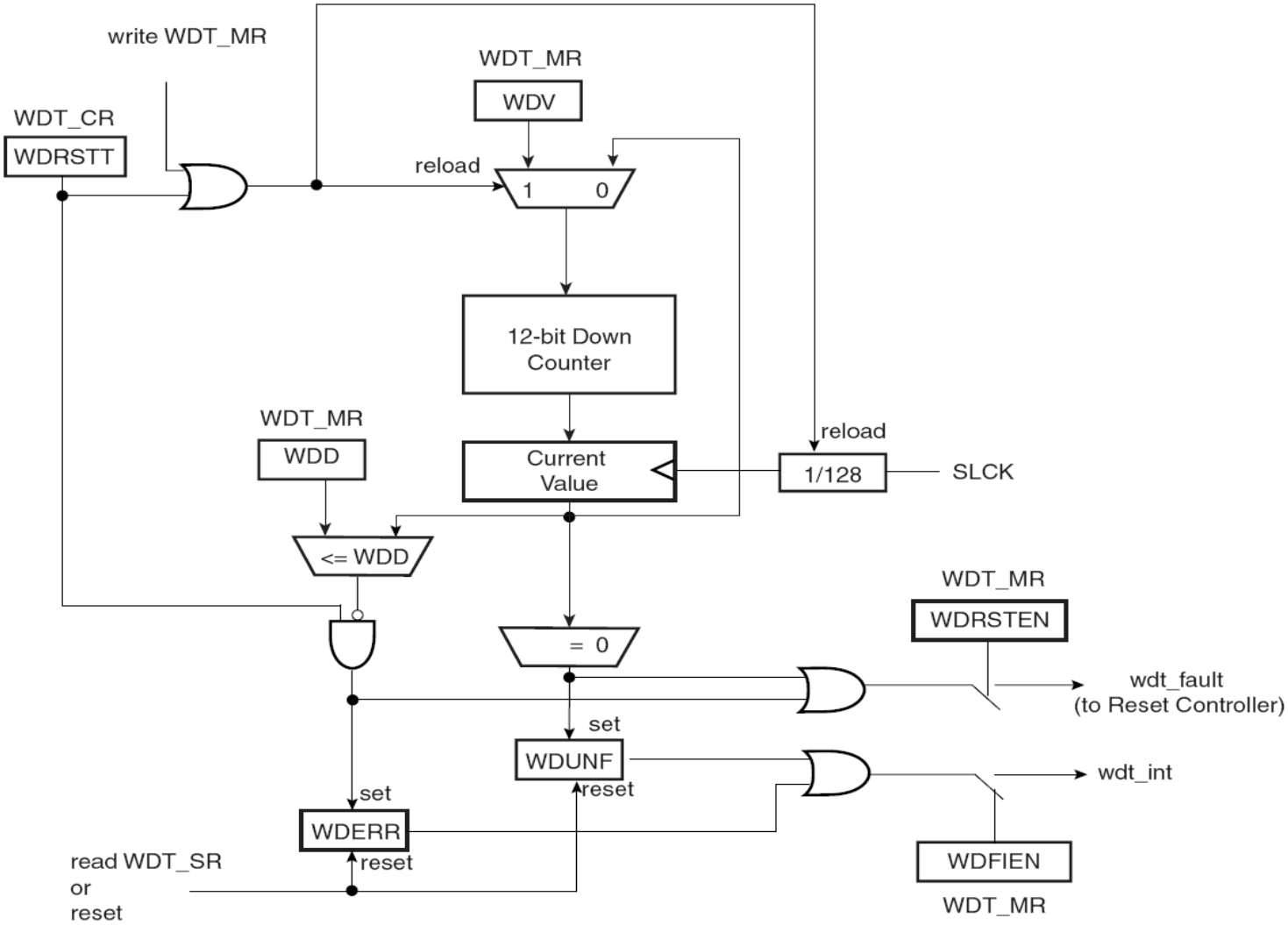
Watchdog Timer

Watchdog Timer (WDT) is used to prevent microprocessor system lock-up if the software becomes trapped in a deadlock.

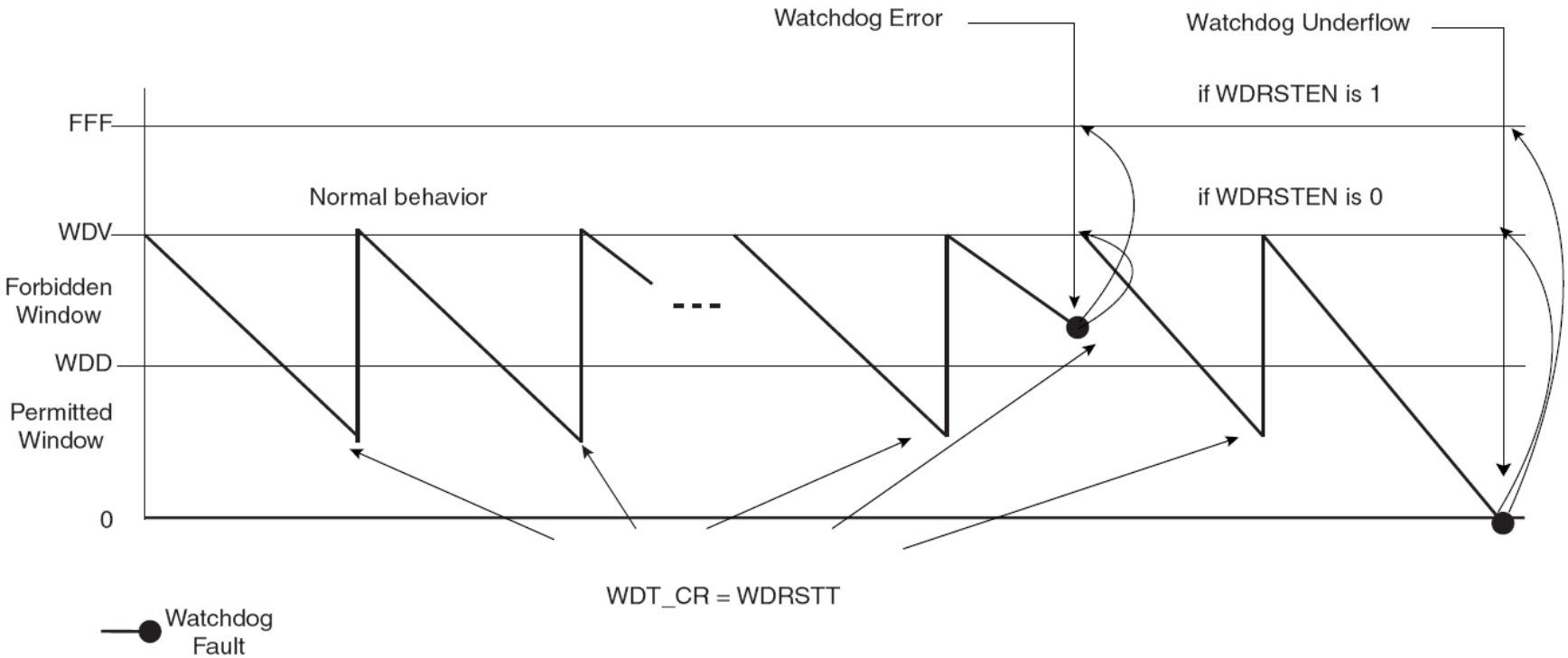
Features of WDT:

- 12-bit down counter,
- Triggered with slow clock (32.768 kHz),
- Maximum watchdog period of up to 16 seconds,
- Can generate a general reset or a processor reset only,
- WDT can be stopped while the processor is in debug mode or idle mode,
- Write protected WDT_CR (control register).

Watchdog Timer – block diagram



WDT – timing charts



WDT – registers (1)

17.4.1 Watchdog Timer Control Register

Register Name: WDT_CR

Access Type: Write-only

31	30	29	28	27	26	25	24
KEY							

23	22	21	20	19	18	17	16
–	–	–	–	–	–	–	–

15	14	13	12	11	10	9	8
–	–	–	–	–	–	–	–

7	6	5	4	3	2	1	0
–	–	–	–	–	–	–	WDRSTT

- **WDRSTT: Watchdog Restart**

0: No effect.

1: Restarts the Watchdog.

- **KEY: Password**

Should be written at value 0xA5. Writing any other value in this field aborts the write operation.

WDT – registers (2)

17.4.2 Watchdog Timer Mode Register

Register Name: WDT_MR

Access Type: Read/Write Once

31	30	29	28	27	26	25	24
		WDIDLEHLT	WDDBGHLT	WDD			

23	22	21	20	19	18	17	16
WDD							

15	14	13	12	11	10	9	8
WDDIS	WDRPROC	WDRSTEN	WDFIEN	WDV			

7	6	5	4	3	2	1	0
WDV							

Timer Counter

- **Features**
  - Three 16-bit Timer/Counter channels
  - Wide range of functions:
    - Frequency measurement
    - Event counting
    - Interval measurement
    - Pulse generation
    - Delay timing
    - Pulse Width Modulation
  - Clock inputs
    - 3 External and 5 Internal clock inputs
  - Two configurable Input/Ouput signals
  - Internal interrupt signal

Interfaces in Embedded Systems

Fundamental Definitions

- **Computer Memory**

Electronic or mechanic device used for storing digital data or computer programs (operating system and applications).

- **Peripheral Device**

Electronic device connected to processor via system bus or computer interface. External devices are used to realise dedicated functionality of the computer system. Internal devices are mainly used by processor and operating system.

- **Computer Bus**

Electrical connection or subsystem that transfers data between computer components: processors, memories and peripheral devices. System bus is composed of dozens of multiple connections (Parallel Bus) or a few single serial channels (Serial Bus).

- **Interface**

Electronic or optical device that allows to connect two or more devices. Interface can be parallel or serial.

Connectivity of Processor and Peripheral Devices

Interfaces used in Embedded Systems:

- Parallel Interface PIO (usually 8, 16 or 32 bits),
- Serial interfaces:
  - Universal Serial Asynchronous Receiver-Transmitter (USART),
  - Serial Peripheral Interface (SPI),
  - Synchronous Serial Controller (SSC)
  - I2C, Two-wire Interface (TWI),
  - Controlled Area Network (CAN),
  - Universal Serial Bus (USB),
  - Ethernet 10/100 Mbits (1 Gbit),
  - Debug/programming interface (EIA RS232, JTAG, SPI, DBGU).

Interfaces available in AT91SAM9263

- Parallel Interface PIO (configurable 32 bits),
- Serial interfaces:
  - Debug interface (DBGU),
  - Universal Serial Asynchronous Receiver-Transmitter (USART),
  - Serial Peripheral Interface (SPI),
  - Synchronous Serial Controller (SSC),
  - I2C, Two-wire Interface (TWI),
  - Controlled Area Network (CAN),
  - Universal Serial Bus (USB, host, endpoint),
  - Ethernet 10/100 Mbits,
  - Programming interface (JTAG).

No Lecture

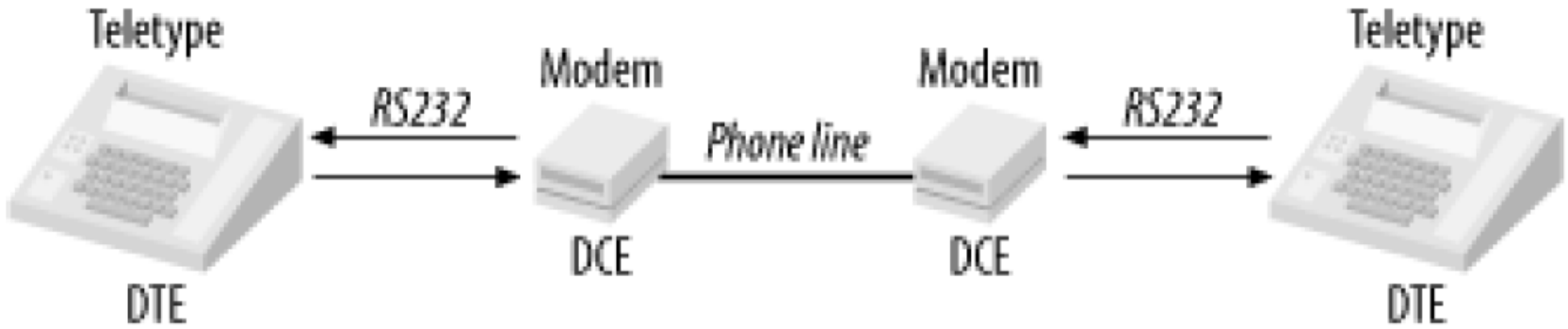
- 12.11.2018 Independence Day - after Day
- 26.11.2018 No Lecture

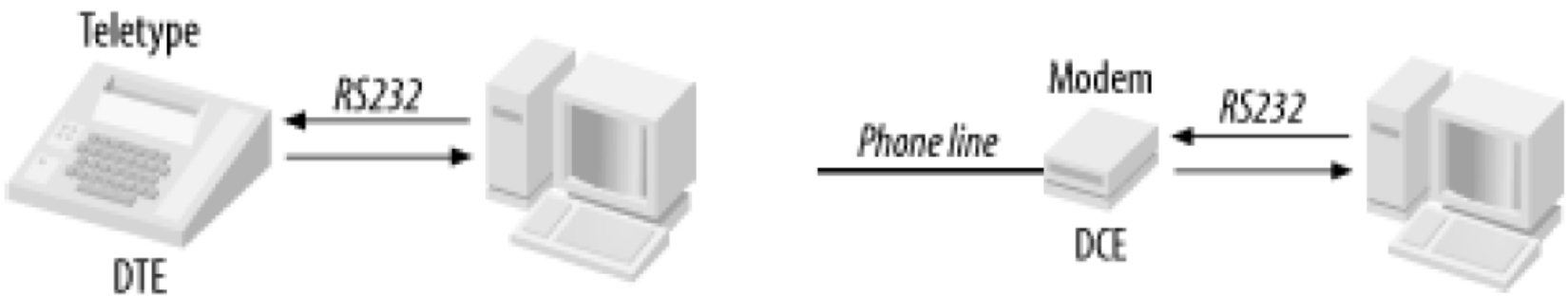
- 03.12.2018 No Lecture, No Lab
- 10.12.2018 No Lecture

- Erasmus Practice #1: ~04.12.2018

Universal Asynchronous Receiver/Transmitter Module

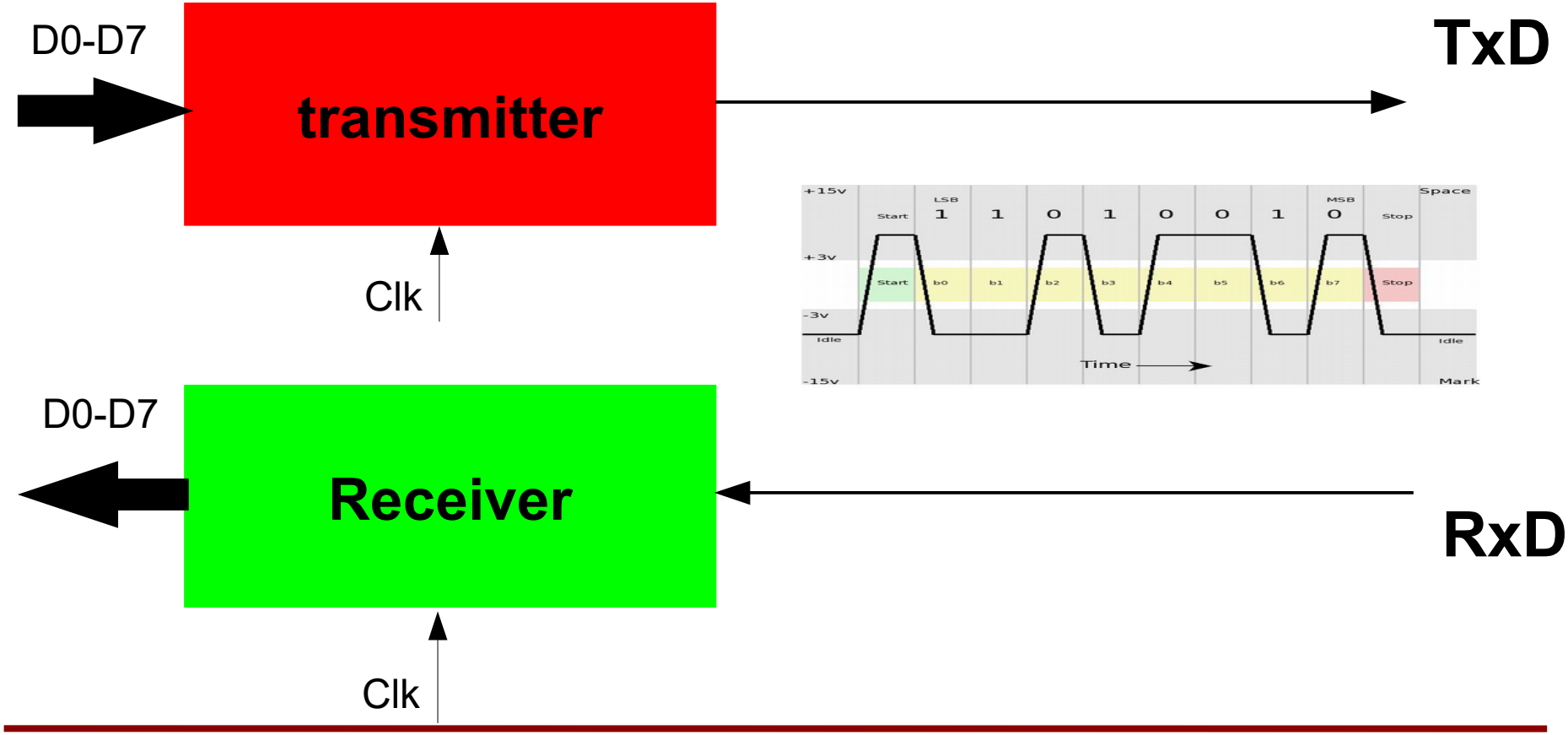
EIA RS232 Serial Interface





UART Transceiver

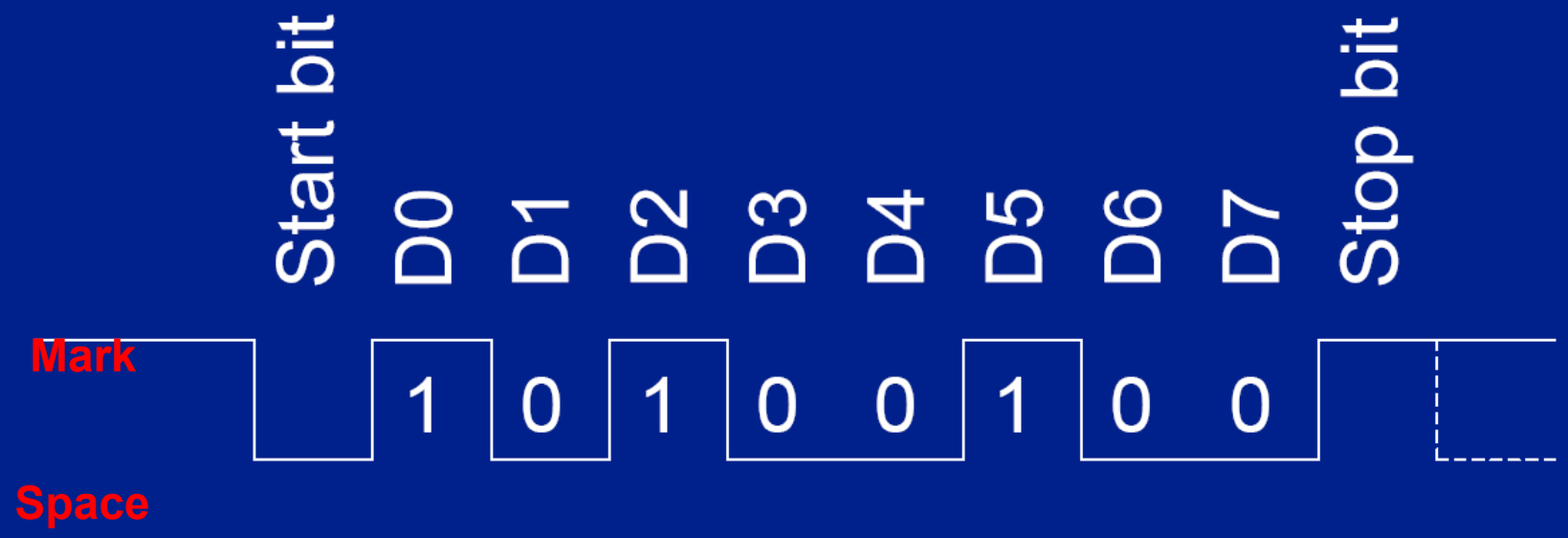
Shift register



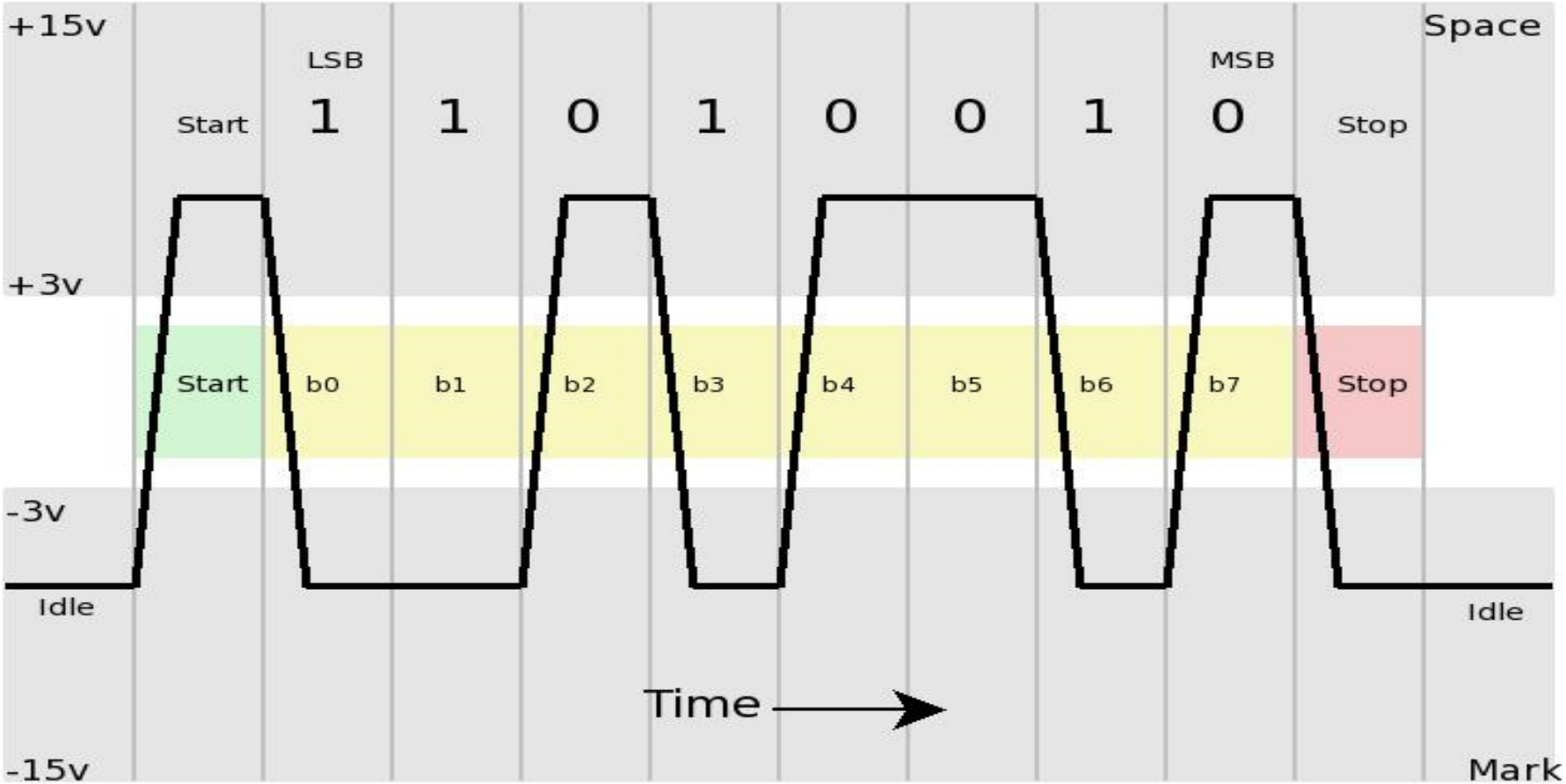
Data Frame of UART (1)

Asynchronous 8 bit waveform example

- Data is H'25' = B'00100101'

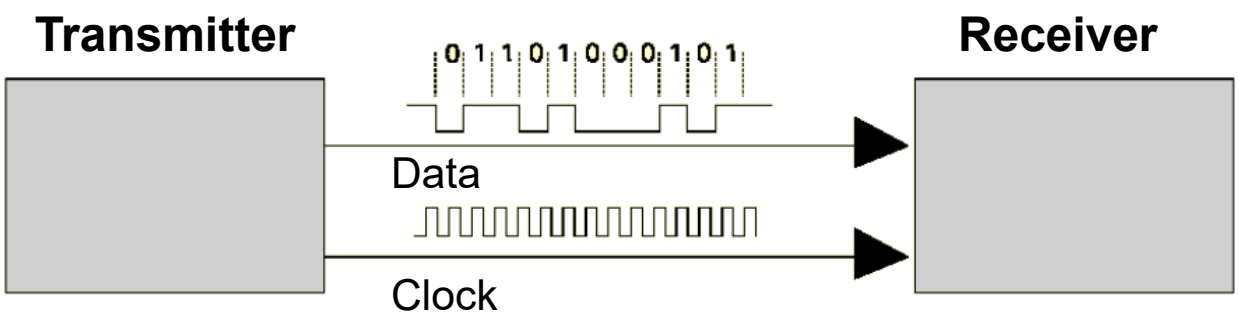


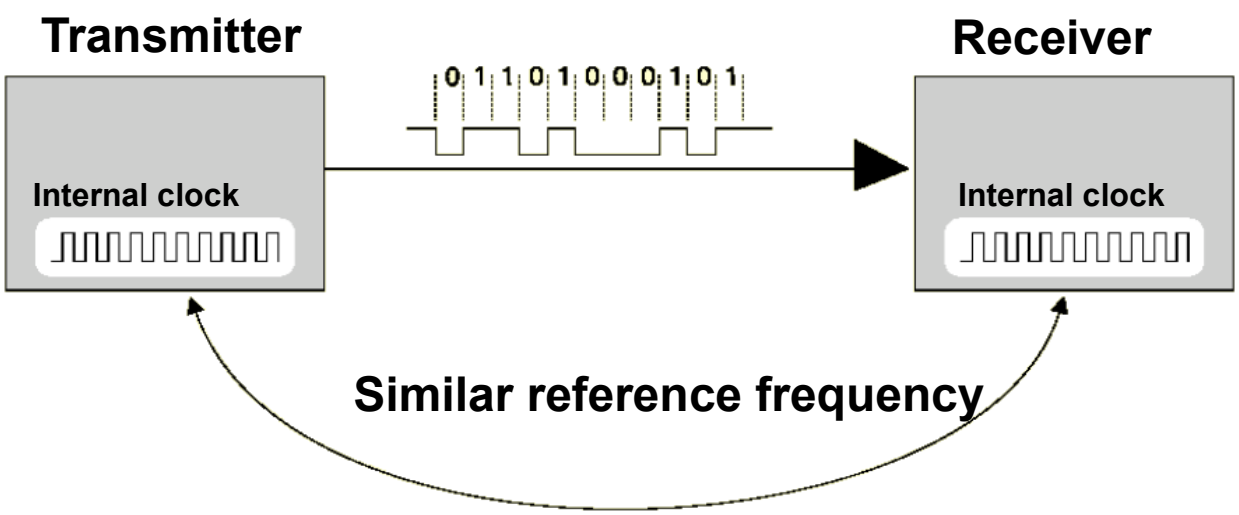
Data Frame of UART (2)



Send data: 0100.1011b = 0x4B

Synchronous vs asynchronous transmission

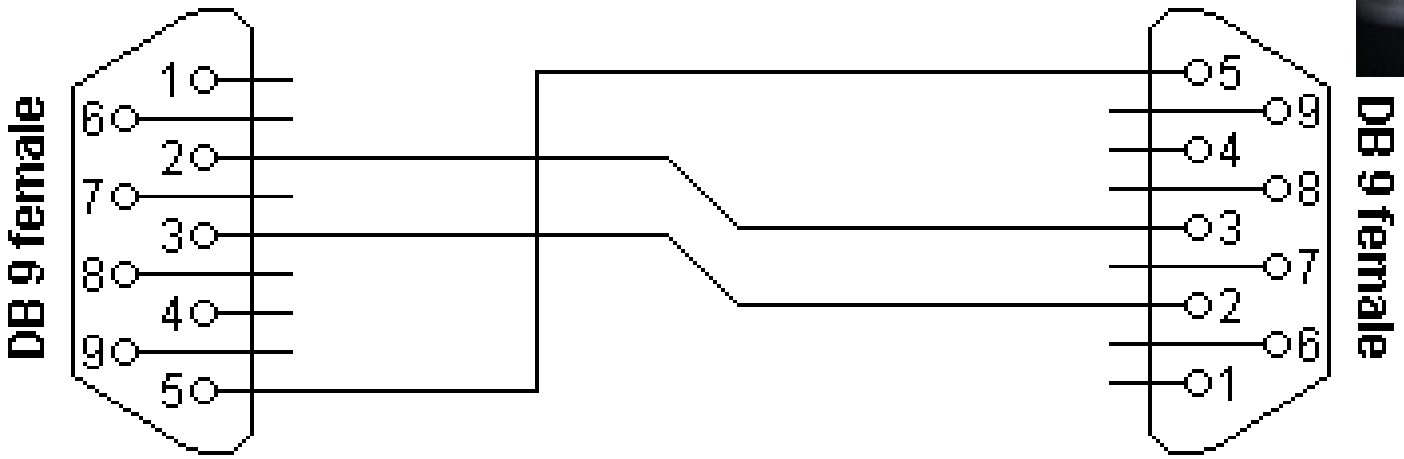




Electrical specification of EIA RS232c

SPECIFICATIONS		RS232
Mode of Operation		SINGLE -ENDED
Total Number of Drivers and Receivers on One Line		1 DRIVER 1 RECVR
Maximum Cable Length		50 FT.
Maximum Data Rate		20kb/s
Maximum Driver Output Voltage		+/-25V
Driver Output Signal Level (Loaded Min.)	Loaded	+/-5V to +/-15V
Driver Output Signal Level (Unloaded Max)	Unloaded	+/-25V
Driver Load Impedance (Ohms)		3k to 7k
Max. Driver Current in High Z State	Power On	N/A
Max. Driver Current in High Z State	Power Off	+/-6mA @ +/-2v
Slew Rate (Max.)		30V/uS
Receiver Input Voltage Range		+/-15V
Receiver Input Sensitivity		+/-3V
Receiver Input Resistance (Ohms)		3k to 7k

Null-Modem Cabel EIA 232



Connector 1	Connector 2	Function
2	3	Rx ← Tx
3	2	Tx → Rx
5	5	Signal ground

Hardware Flow Control

Symbol	Circuit	Line state	Remarks
DTR	108/2		Computer ready
DSR	107		Modem ready
RTS	105		Request to send
CTS	106		Ready to send
TxD	103		Start transmission

DTE Data Terminal Equipment – terminal, PC

DCE - Data Circuit-terminating Equipment – Modem

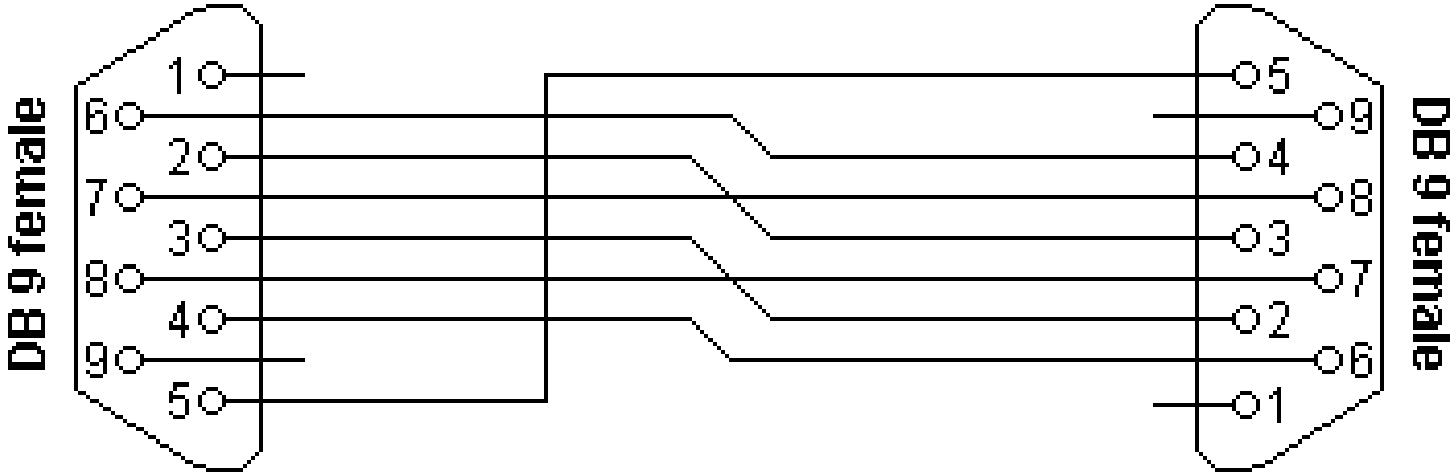
DSR - Data Set Ready - modem

DTR - Data Terminal Ready – terminal

RTS - Request to Send Data

CTS - Clear to Send - ready to send data

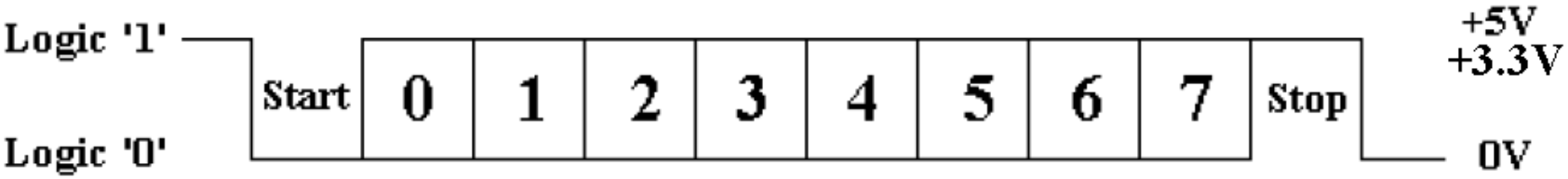
Null-Modem Cabel EIA 232 with Hardware flow Control



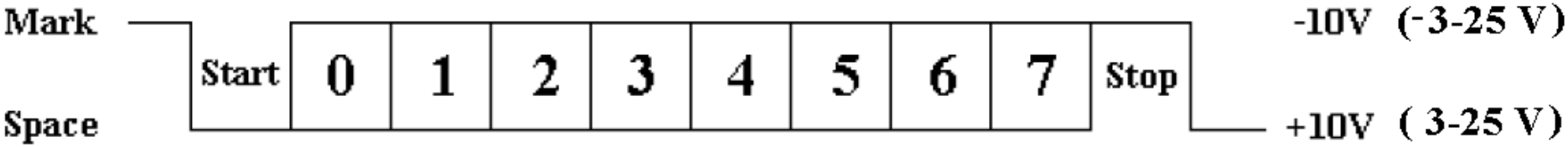
Connector 1	Connector 2	Function
2	3	Rx ← Tx
3	2	Tx → Rx
4	6	DTR → DSR
5	5	Signal ground
6	4	DSR ← DTR
7	8	RTS → CTS
8	7	CTS ← RTS

Voltage Levels of EIA RS232

Processor output

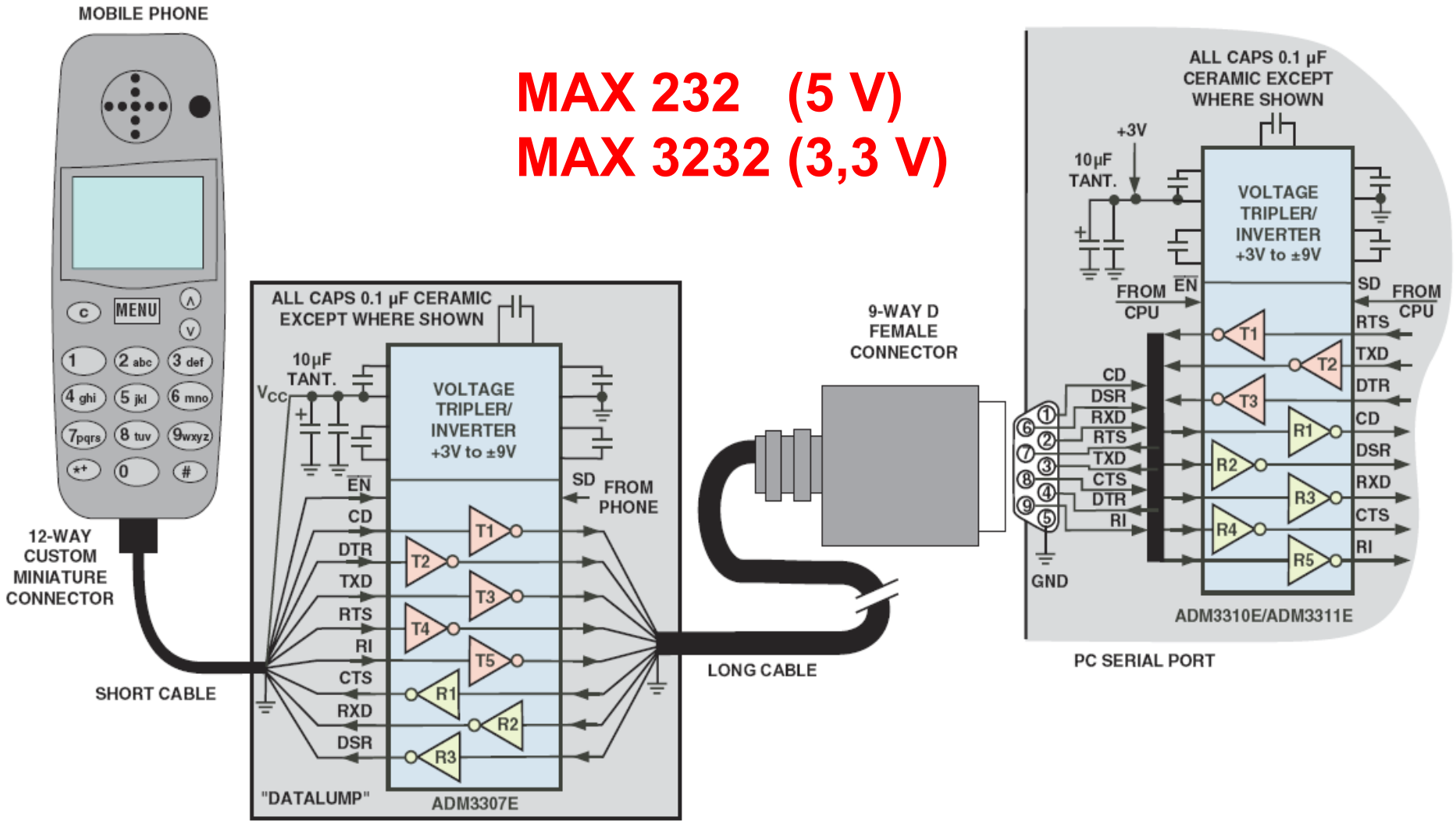


EIA RS 232



RS-232 Logic Waveform

Voltage Levels Translator



Software for EIA RS232 communication

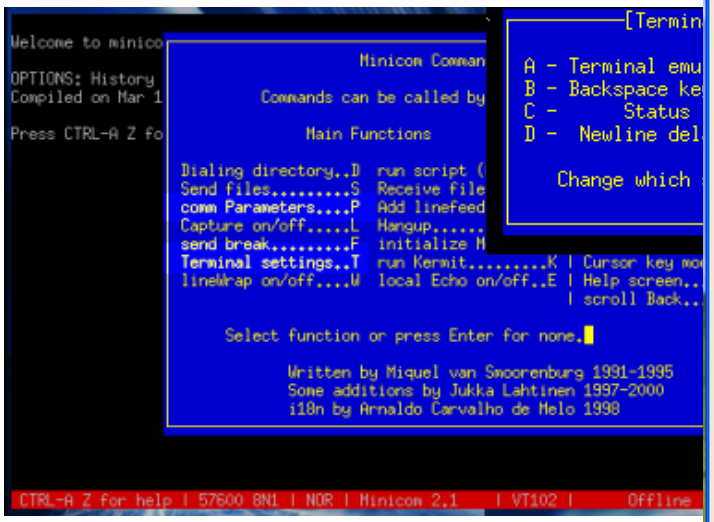
Hyper terminal

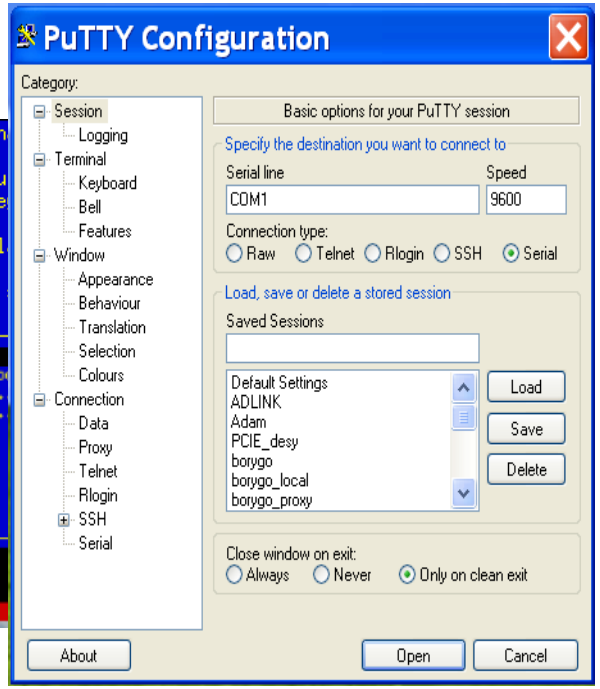
Minicom

ssh

Terminal

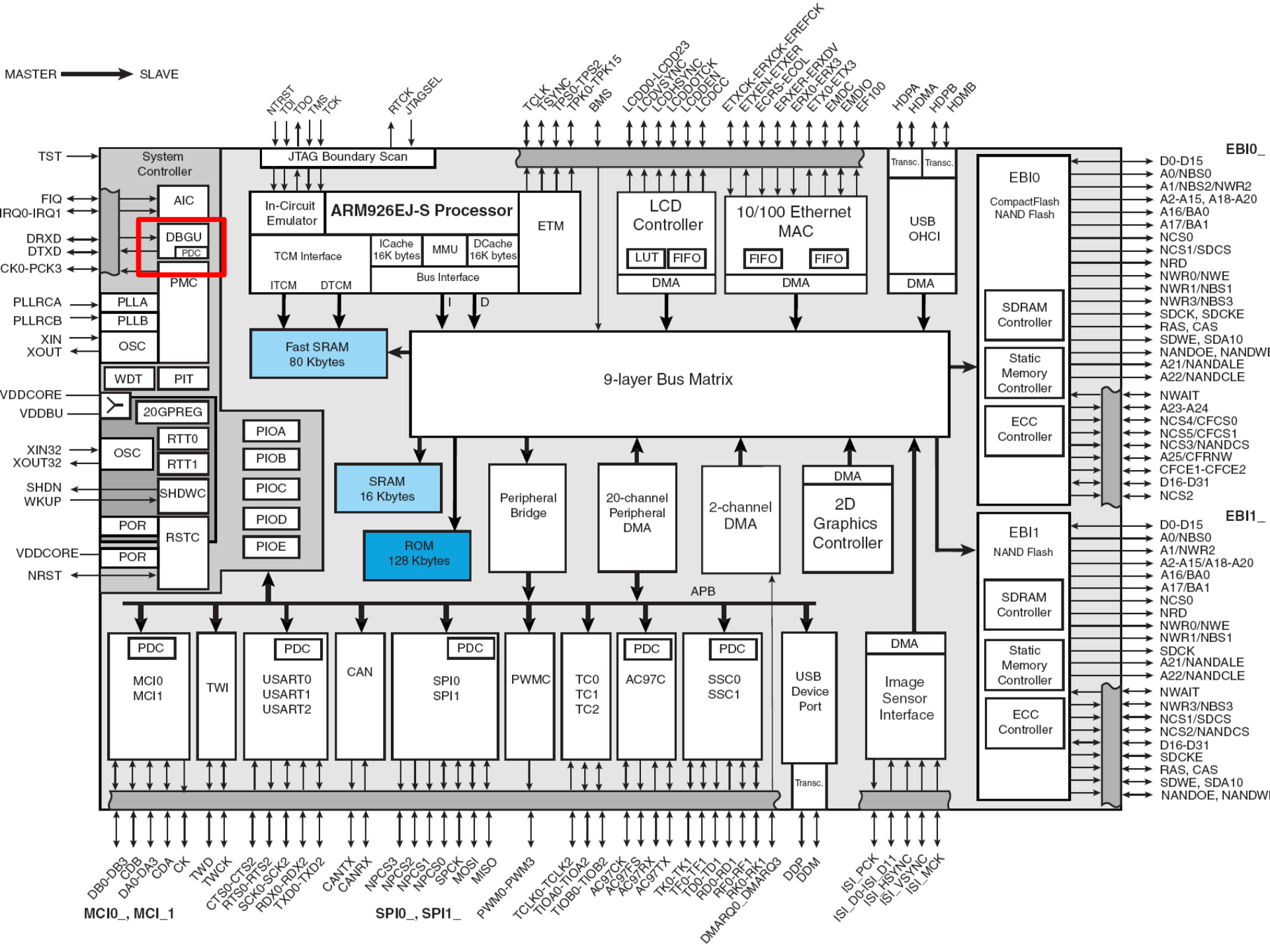
(http://www.elester-pkp.com.pl/index.php?id=92&lang=pl&zoom=0)





AT91SAM9263 – debug module DBGU

(chapter 30)

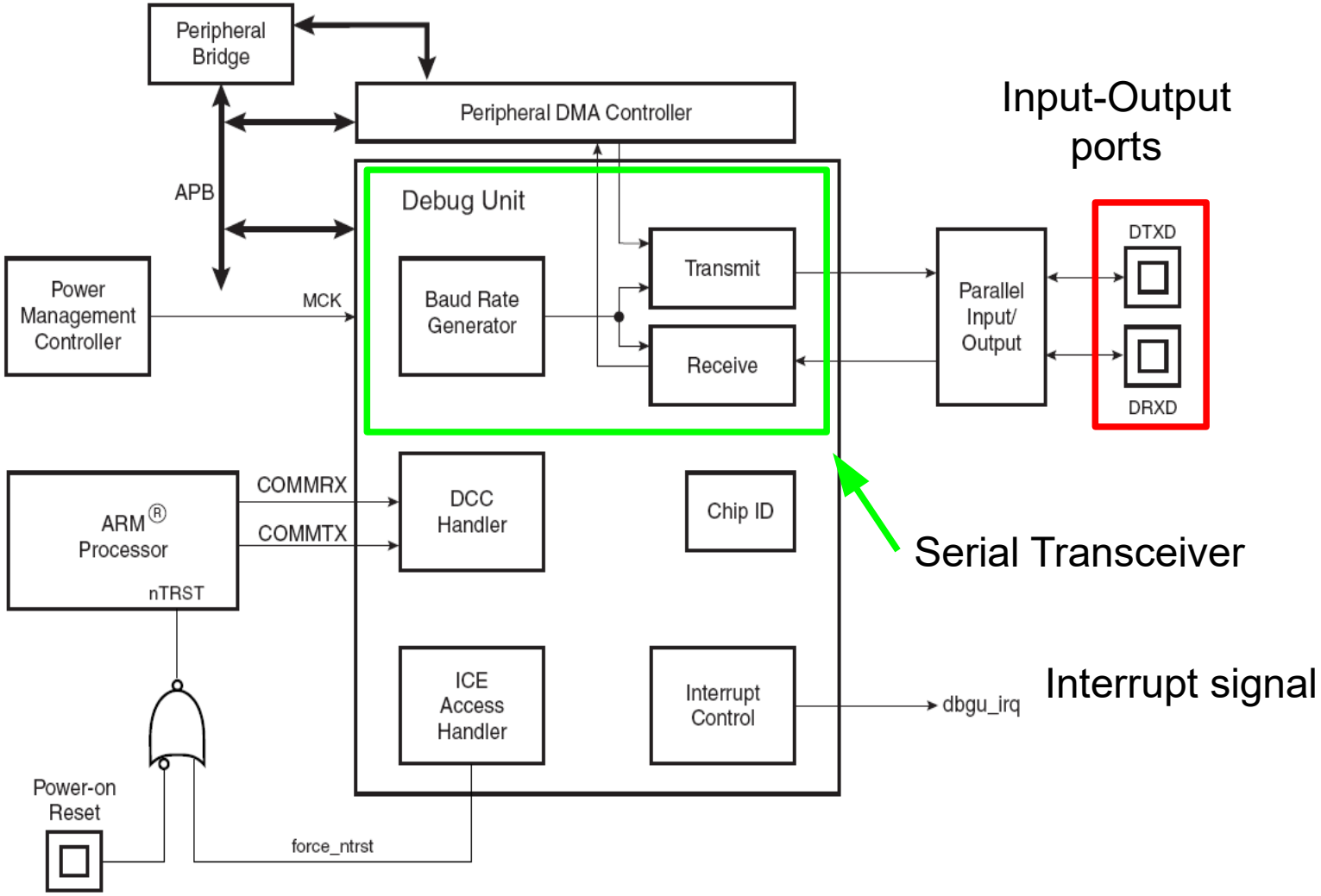

MASTER
SLAVE
System Controller
AIC
DBGU
PDC
PMC
PLLA
PLLB
OSC
WDT
PIT
20GPREG
RTT0
RTT1
SHDWC
POR
RSTC
PIOA
PIOB
PIOC
PIOD
PIOE
JTAG Boundary Scan
In-Circuit Emulator
ARM926EJ-S Processor
TCM Interface
ICache 16K bytes
MMU
DCache 16K bytes
Bus Interface
ITCM
DTCM
ETM
LCD Controller
LUT
FIFO
DMA
10/100 Ethernet MAC
FIFO
FIFO
DMA
Transc.
Transc.
USB OHCI
DMA
EBI0
CompactFlash NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
Fast SRAM 80 Kbytes
9-layer Bus Matrix
SRAM 16 Kbytes
ROM 128 Kbytes
Peripheral Bridge
20-channel Peripheral DMA
2-channel DMA
DMA
2D Graphics Controller
EBI1
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
APB
PDC
MCI0 MCI1
TWI
PDC
USART0 USART1 USART2
CAN
PDC
SPI0 SPI1
PWMC
TC0 TC1 TC2
PDC
AC97C
PDC
SSC0 SSC1
USB Device Port
Transc.
DMA
Image Sensor Interface
TST
FIQ
IRQ0-IRQ1
DRXD
DTXD
PCK0-PCK3
PLLRCA
PLLRCB
XIN
XOUT
VDDCORE
VDDBU
XIN32
XOUT32
SHDN
WKUP
VDDCORE
NRST
NTRST TDI TDO TMS TCK
RTCK JTAGSEL
TCLK TSYNC TPS0-TPS2 TPK0-TPK15 BMS
LCDD0-LCDD23 LCDVSYNC LCDHSYNC LCDDOTCK LCDDEN LCDCC
ETXCK-ERXCK-EREFCK ETXEN-ETXER ECRS-ECOL ERXER-ERXDV ERX0-ERX3 ETX0-ETX3 EMDC EMDIO EF100
HDPA HDMA HDPB HDMB
EBI0_
D0-D15
A0/NBS0
A1/NBS2/NWR2
A2-A15, A18-A20
A16/BA0
A17/BA1
NCS0
NCS1/SDCS
NRD
NWR0/NWE
NWR1/NBS1
NWR3/NBS3
SDCK, SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
A21/NANDALE
A22/NANDCLE
NWAIT
A23-A24
NCS4/CFCS0
NCS5/CFCS1
NCS3/NANDCS
A25/CFRNW
CFCE1-CFCE2
D16-D31
NCS2
EBI1_
D0-D15
A0/NBS0
A1/NWR2
A2-A15/A18-A20
A16/BA0
A17/BA1
NCS0
NRD
NWR0/NWE
NWR1/NBS1
SDCK
A21/NANDALE
A22/NANDCLE
NWAIT
NWR3/NBS3
NCS1/SDCS
NCS2/NANDCS
D16-D31
SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
DB0-DB3 CDB DA0-DA3 CDA CK
MCI0_, MCI_1
TWD TWCK
CTS0-CTS2 RTS0-RTS2 SCK0-SCK2 RDX0-RDX2 TXD0-TXD2
CANTX CANRX
NPCS3 NPCS2 NPCS1 NPCS0 SPCK MOSI MISO
SPI0_, SPI1_
PWM0-PWM3
TCLK0-TCLK2 TIOA0-TIOA2 TIOB0-TIOB2
AC97CK AC97FS AC97RX AC97TX
TK0-TK1 TF0-TF1 TD0-TD1 RD0-RD1 RF0-RF1 RK0-RK1 DMARQ0_DMARQ3
DDP DDM
ISI_PCK ISI_D0-ISI_D11 ISI_HSYNC ISI_VSYNC ISI_MCK

Serial interface as Diagnostic Tool

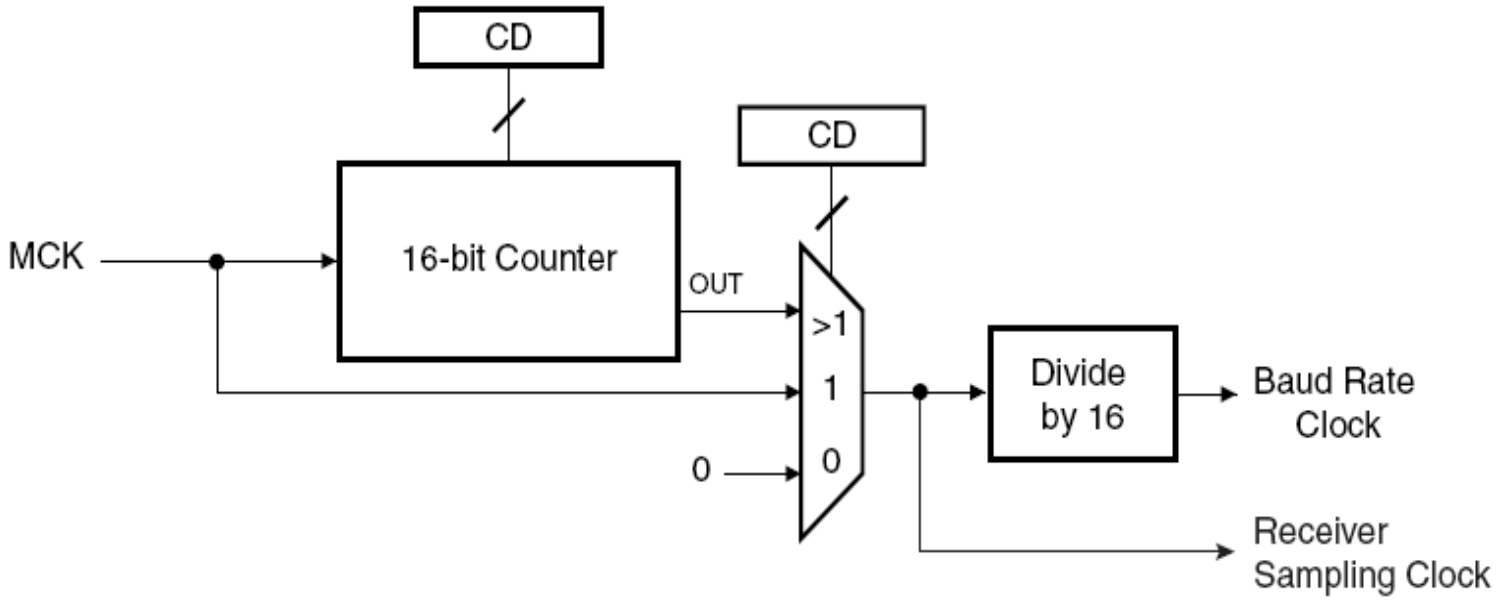
Features of DBGU port (DeBuG Unit):

- Asynchronous data transmission compatible with RS232 standard (8 bits, single parity bit – can be switched off),
- Single system interrupt, shared with PIT, RTT, WDT, DMA, PMC, RSTC, MC,
- Frame correctness analysis,
- RxD buffer overflow signal,
- Diagnostic modes: external loopback, local loopback and echo,
- Maximum transmission baudrate 1 Mbit/s,
- Direct connectivity to debug module build in ARM core (COMMRx/COMMTx).

Block diagram of DBGU transmission module



Transmission speed



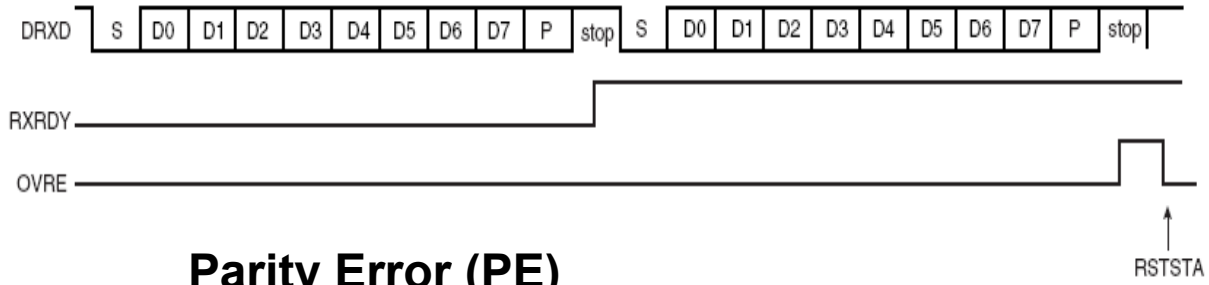
Reference clock generator is responsible for Baud Rate .

Baud rate can be calculated using formula:

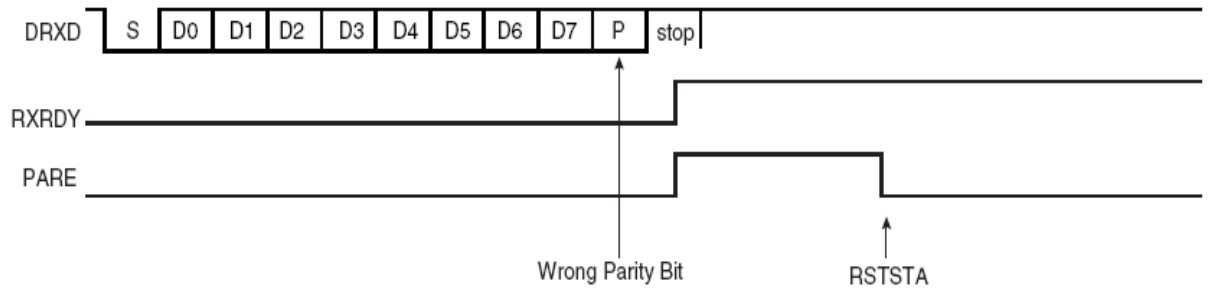
Baud Rate = MCK / (16 x CD), where CD
Clock Divisor can be found in DBGU_BRGR register

Transmission errors

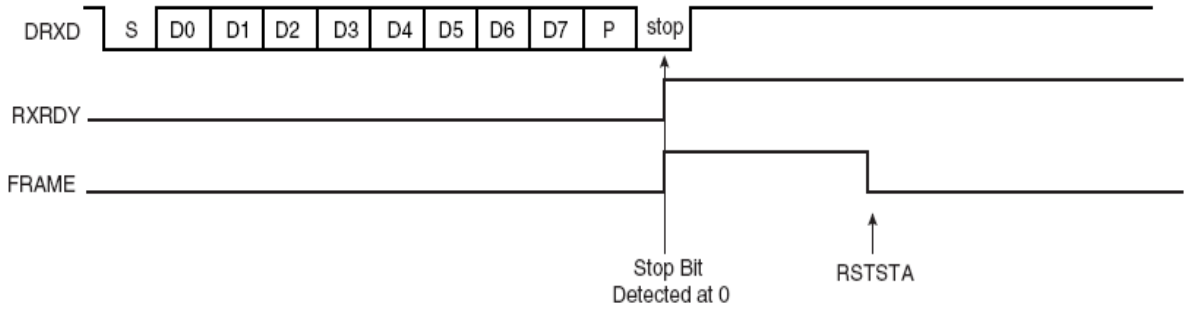
Receiver Buffer Overflow (BGU_RHR)



Parity Error (PE)



Frame Error (FE)



Configuration of DBGU transceiver

static void Open_DBGU (void){

1. Deactivate DBGU interrupts (register AT91C_BASE_DBGU->DBGU_IDR)
2. Reset and turn off receiver (register AT91C_BASE_DBGU->DBGU_CR)
3. Reset and turn off transmitter (register AT91C_BASE_DBGU->DBGU_CR)
4. Configure RxD i TxD DBGU as input peripheral ports (registers AT91C_BASE_PIOC->PIO_ASR and AT91C_BASE_PIOC->PIO_PDR)
5. Configure throughput (e.g. 115200 bps, register AT91C_BASE_DBGU->DBGU_BRGR)
6. Configure operation mode (e.g. 8N1, register AT91C_BASE_DBGU->DBGU_MR, flags AT91C_US_CHMODE_NORMAL, AT91C_US_PAR_NONE)
7. Configure interrupts if used, e.g. Open_DBGU_INT()
8. Turn on receiver (register AT91C_BASE_DBGU->DBGU_CR),
9. Turn on transmitter if required (register AT91C_BASE_DBGU->DBGU_CR),

}

Read and write via DBGU port

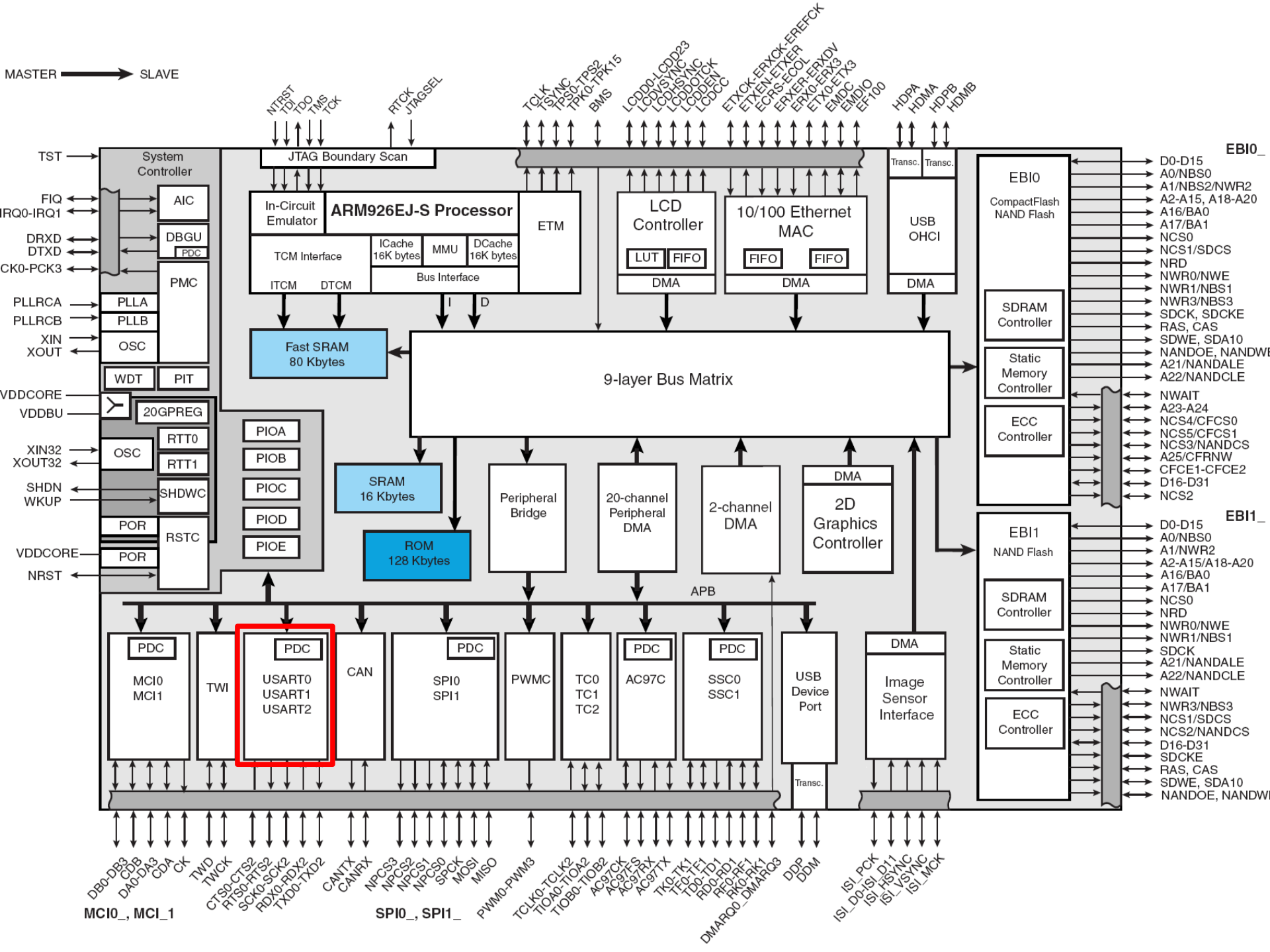
Interrupts are disabled.

```
void dbgu_print_ascii (const char Buffer)
{
   while ( data_are_in_buffer ) {
        while ( …TXRDY... ){};          /* wait intil Tx buffer busy – check TXRDY flag */
        DBGU_THR = ...                  /* write a single char to Transmitter Holding Register  */
         }
}

void dbgu_read_ascii (char *Buffer, unsigned int Size){
        do {
        While ( ...RXRDY... ){};        /* wait until data available */
        Buffer[...] = DBGU_RHR;         /* read data from Receiver Holding Register */
        } while ( …read_enough_data... )
}
```

AT91SAM9263 – USART

(chapter 34)

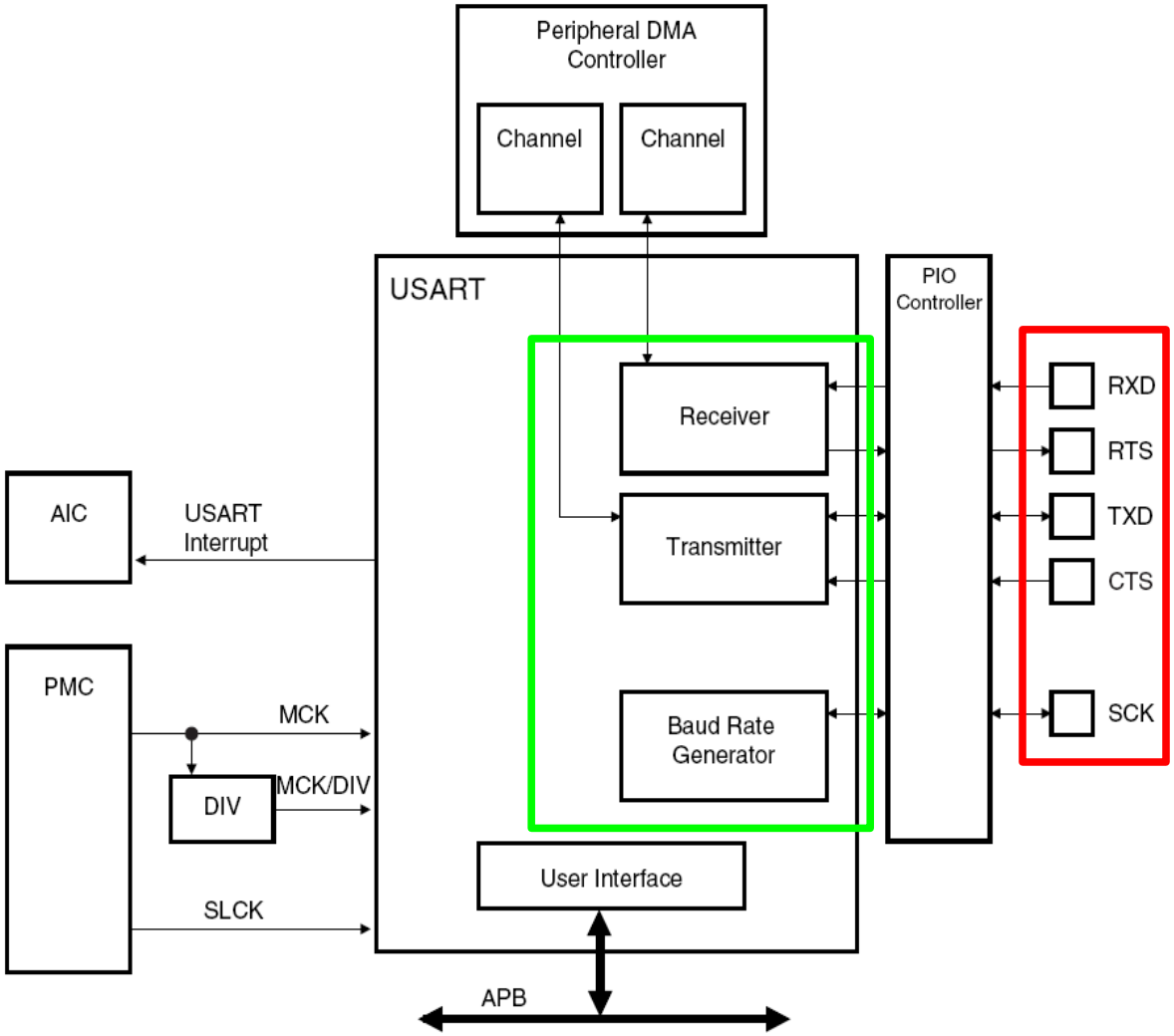

MASTER
SLAVE
System Controller
JTAG Boundary Scan
In-Circuit Emulator
ARM926EJ-S Processor
ETM
TCM Interface
ICache 16K bytes
MMU
DCache 16K bytes
Bus Interface
ITCM
DTCM
LCD Controller
LUT
FIFO
DMA
10/100 Ethernet MAC
FIFO
FIFO
DMA
Transc.
Transc.
USB OHCI
DMA
EBI0
CompactFlash NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
EBI1
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
AIC
DBGU
PDC
PMC
PLLA
PLLB
OSC
WDT
PIT
20GPREG
RTT0
RTT1
OSC
SHDWC
POR
RSTC
POR
PIOA
PIOB
PIOC
PIOD
PIOE
Fast SRAM 80 Kbytes
9-layer Bus Matrix
SRAM 16 Kbytes
ROM 128 Kbytes
Peripheral Bridge
20-channel Peripheral DMA
2-channel DMA
DMA
2D Graphics Controller
APB
PDC
MCI0 MCI1
TWI
PDC
USART0 USART1 USART2
CAN
PDC
SPI0 SPI1
PWMC
TC0 TC1 TC2
PDC
AC97C
PDC
SSC0 SSC1
USB Device Port
Transc.
DMA
Image Sensor Interface
TST
FIQ
IRQ0-IRQ1
DRXD
DTXD
PCK0-PCK3
PLLRCA
PLLRCB
XIN
XOUT
VDDCORE
VDDBU
XIN32
XOUT32
SHDN
WKUP
VDDCORE
NRST
NTRST TDI TDO TMS TCK
RTCK JTAGSEL
TCLK TSYNC TPS0-TPS2 TPK0-TPK15 BMS
LCDD0-LCDD23 LCDVSYNC LCDHSYNC LCDDOTCK LCDDEN LCDCC
ETXCK-ERXCK-EREFCK ETXEN-ETXER ECRS-ECOL ERXER-ERXDV ERX0-ERX3 ETX0-ETX3 EMDC EMDIO EF100
HDPA HDMA HDPB HDMB
EBI0_
D0-D15
A0/NBS0
A1/NBS2/NWR2
A2-A15, A18-A20
A16/BA0
A17/BA1
NCS0
NCS1/SDCS
NRD
NWR0/NWE
NWR1/NBS1
NWR3/NBS3
SDCK, SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
A21/NANDALE
A22/NANDCLE
NWAIT
A23-A24
NCS4/CFCS0
NCS5/CFCS1
NCS3/NANDCS
A25/CFRNW
CFCE1-CFCE2
D16-D31
NCS2
EBI1_
D0-D15
A0/NBS0
A1/NWR2
A2-A15/A18-A20
A16/BA0
A17/BA1
NCS0
NRD
NWR0/NWE
NWR1/NBS1
SDCK
A21/NANDALE
A22/NANDCLE
NWAIT
NWR3/NBS3
NCS1/SDCS
NCS2/NANDCS
D16-D31
SDCKE
RAS, CAS
SDWE, SDA10
NANDOE, NANDWE
DB0-DB3 CDB DA0-DA3 CDA CK
TWD TWCK
CTS0-CTS2 RTS0-RTS2 SCK0-SCK2 RDX0-RDX2 TXD0-TXD2
CANTX CANRX
NPCS3 NPCS2 NPCS1 NPCS0 SPCK MOSI MISO
PWM0-PWM3
TCLK0-TCLK2 TIOA0-TIOA2 TIOB0-TIOB2
AC97CK AC97FS AC97RX AC97TX
TK0-TK1 TF0-TF1 TD0-TD1 RD0-RD1 RF0-RF1 RK0-RK1 DMARQ0_DMARQ3
DDP DDM
ISI_PCK ISI_D0-ISI_D11 ISI_HSYNC ISI_VSYNC ISI_MCK
MCI0_, MCI_1
SPI0_, SPI1_

Serial port USART

Features of Universal Synch. Asynch. Receiver-Transmitter:

- Asynchronous or synchronous data transfer,
- Programmable frame length, parity, stop bits,
- Single system interrupt (shared with: PIT, RTT, WDT,DMA, PMC, RSTC, MC),
- Analysis of correctness of received frames,
- Buffer overflow error TxD or RxD,
- Elastic buffer – possibility of receiving frames with different length (uses additional counter),
- Diagnostic modes: external loopback, local loopback and echo,
- Maximum transmission speed 1 Mbit/s,
- Hardware flow control,
- Support for Multidrop transmission – data and address,
- Available Direct Memory Access channel,
- Support for RS485 differential transmission mode and infrared systems (build-in IrDA modulator-demodulator).

Block diagram of USART transceiver



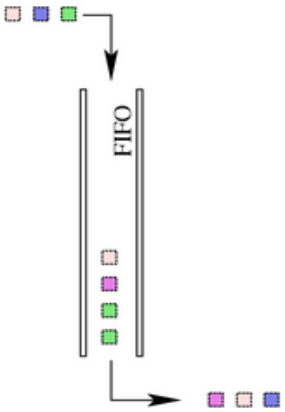
Data structures

Stack (1)

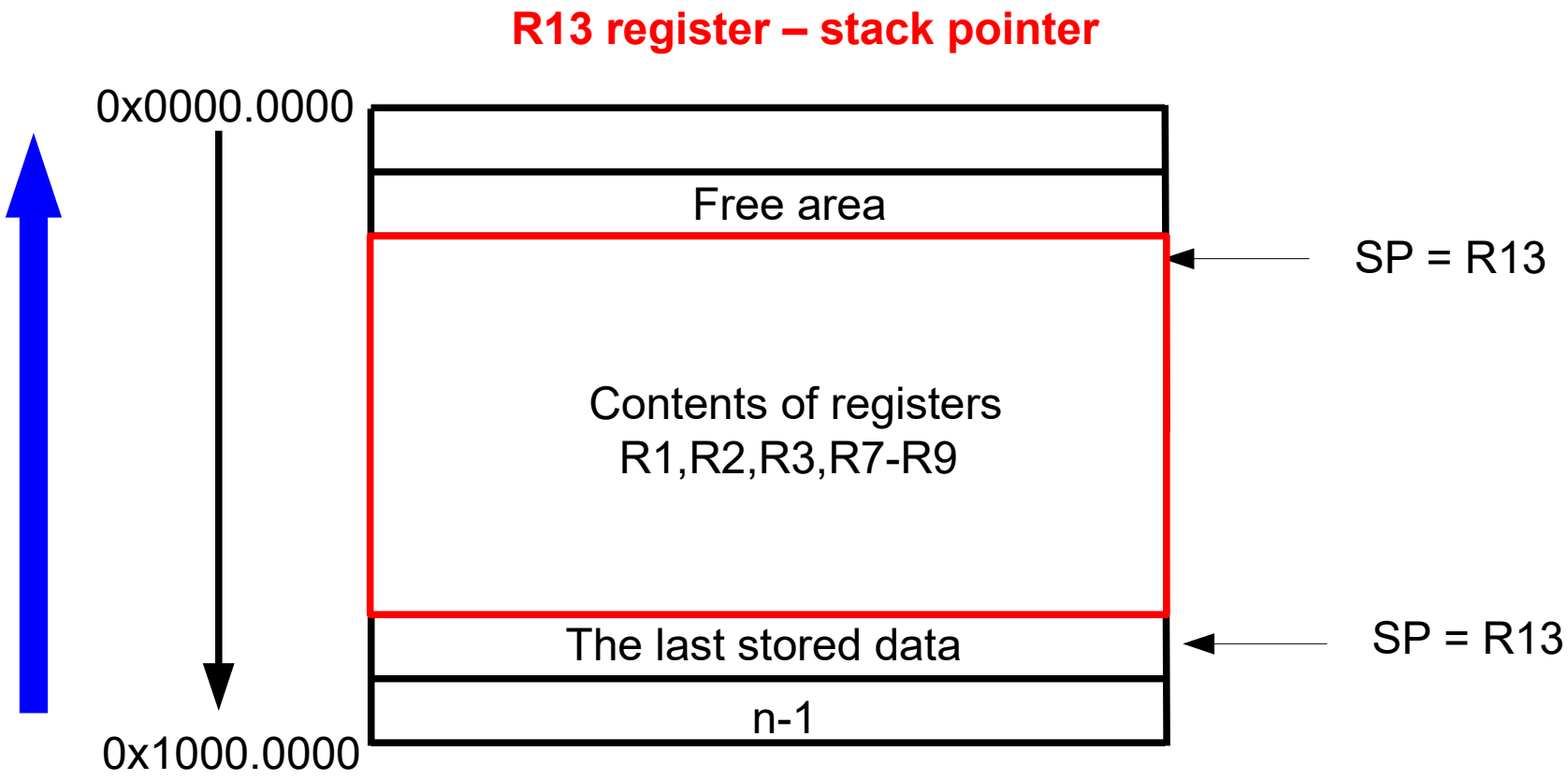
Stack or LIFO (Last-In, First-Out) – abstract data type and data structure. A stack can have any abstract data type as an element, but it is characterized by only two fundamental operations: push and pop. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.

FIFO (First In, First Out) – a linear buffer, the opposite structure to stack. The first element placed into FIFO is immediately transferred to the end of the queue. Therefore the first element stored in FIFO is supposed to be processed first.

First–in First–out (FIFO)

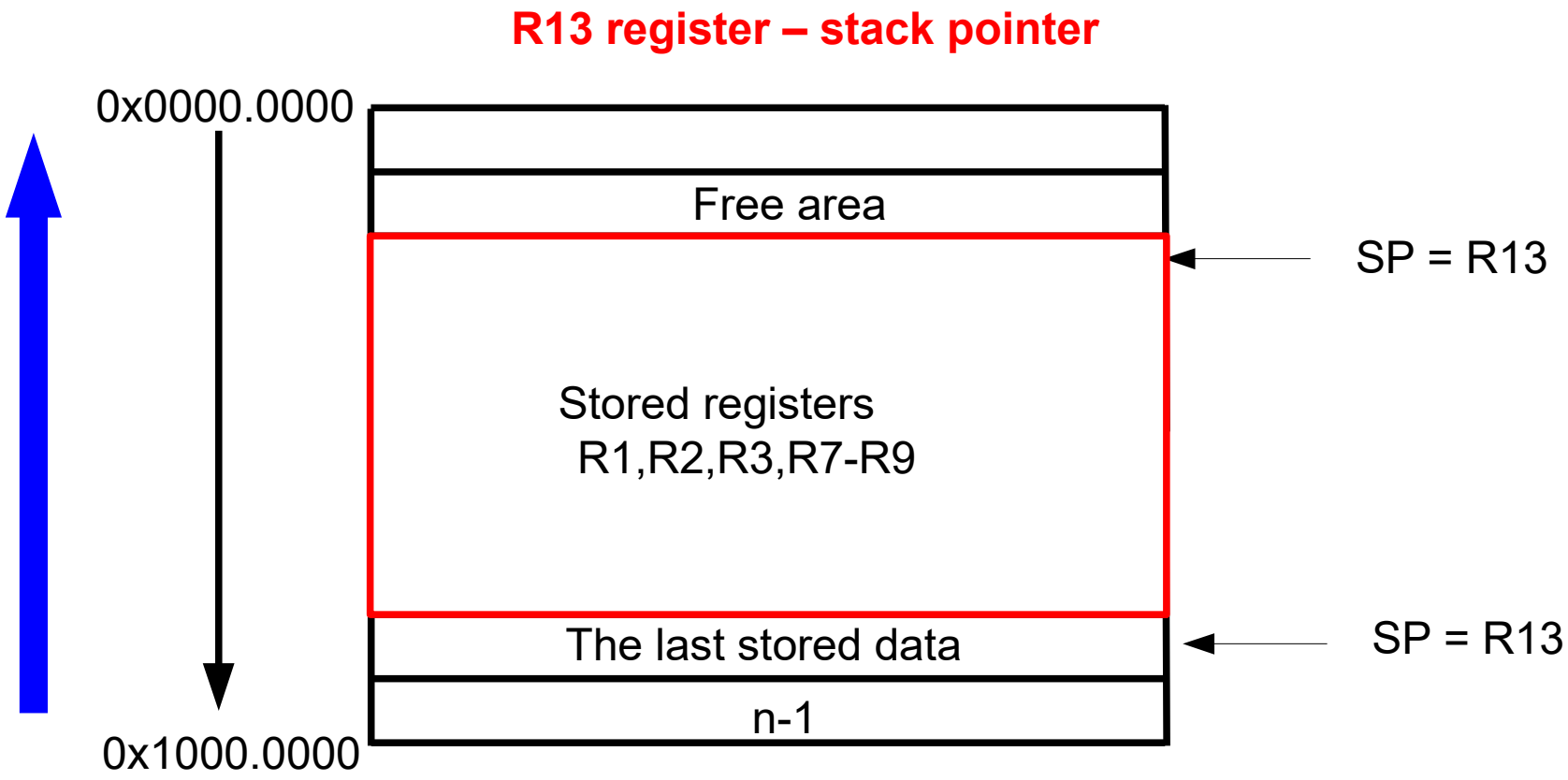


Stack – push data



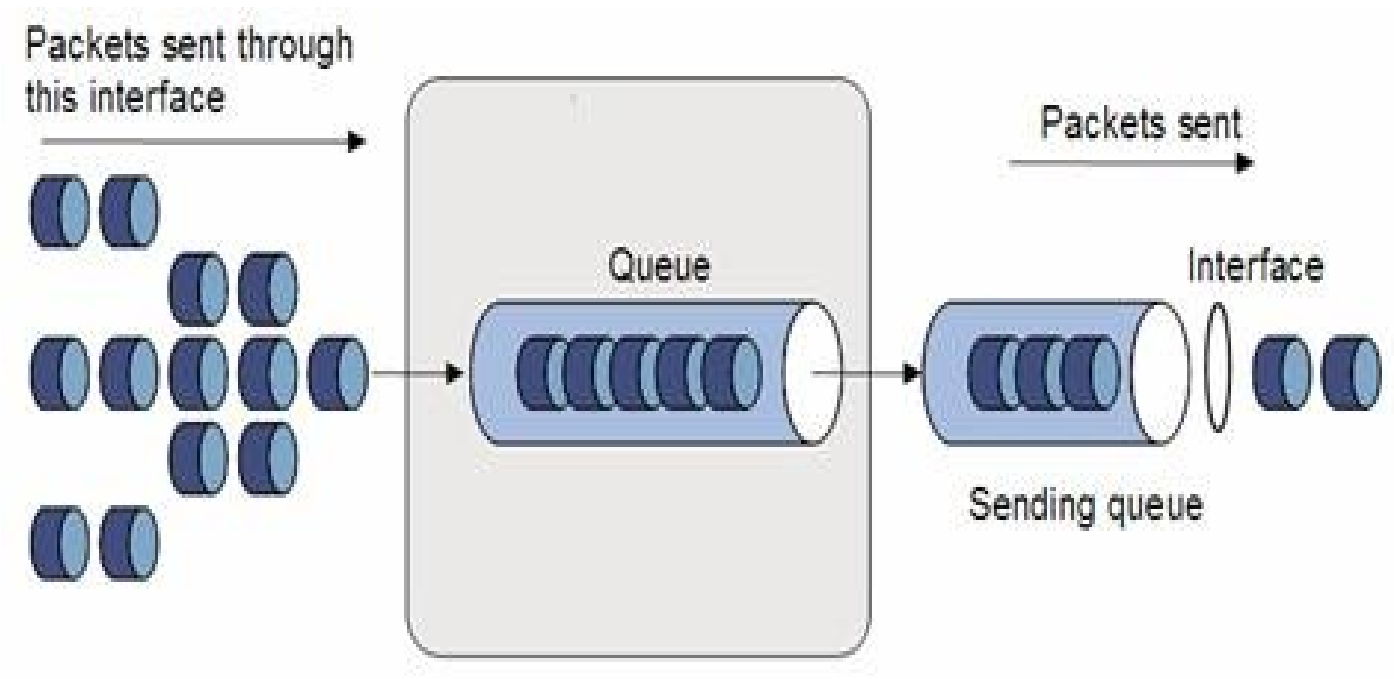

```
STMDB   SP!, {registers list}
STMDB   SP!, {R1,R2,R3,R7-R9}     | decrease SP by 24, stores 8 registers on stack
```

Stack - pop



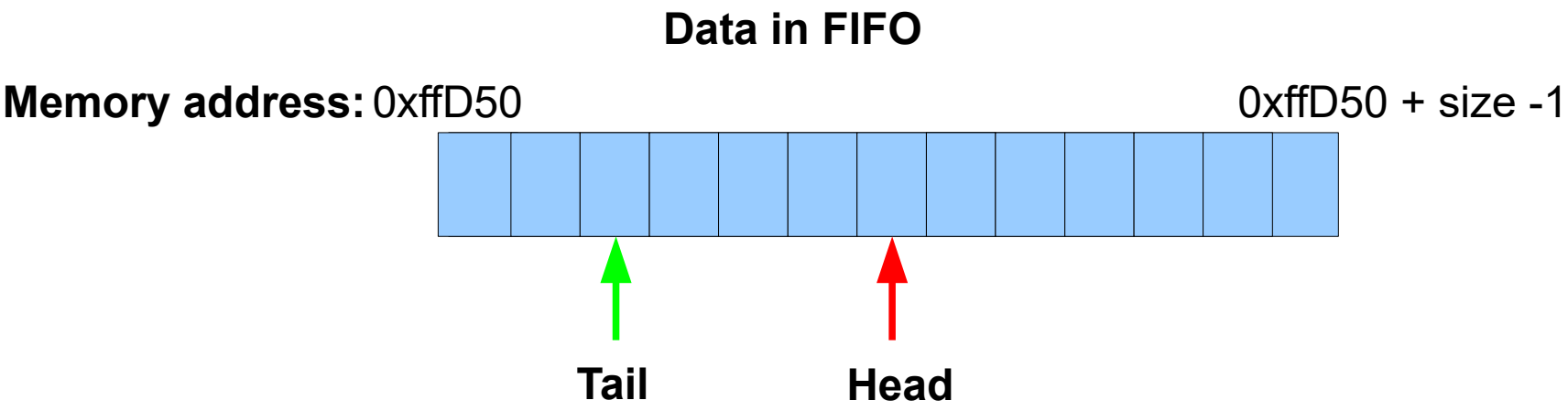

```
LDMIA    SP!, {list of registers}
LDMIA    SP!, {R1,R2,R3,R7-R9}    | increase SP by 24, recover 8 registers from stack
```

FIFO (1)



- A few different applications can try to write data into FIFO queue. In such a case a semaphore can be used to control access during writing data to queue.
- Data are read from queue in the same order as was written

FIFO (2)



Write data to FIFO:

- Increase Head by one, write data.

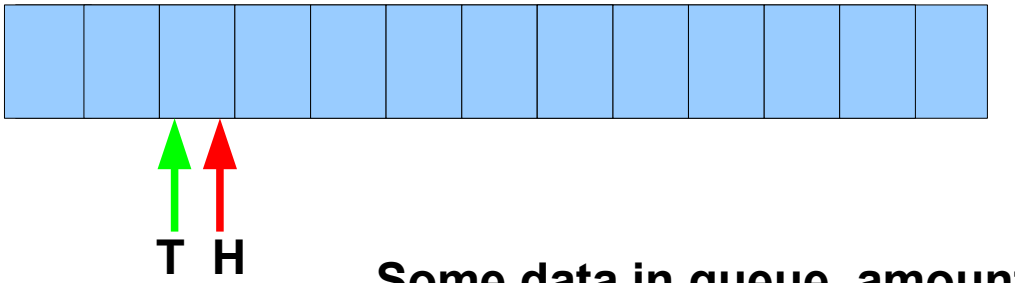
Read data from FIFO:

- Read data, increase Tail by one.

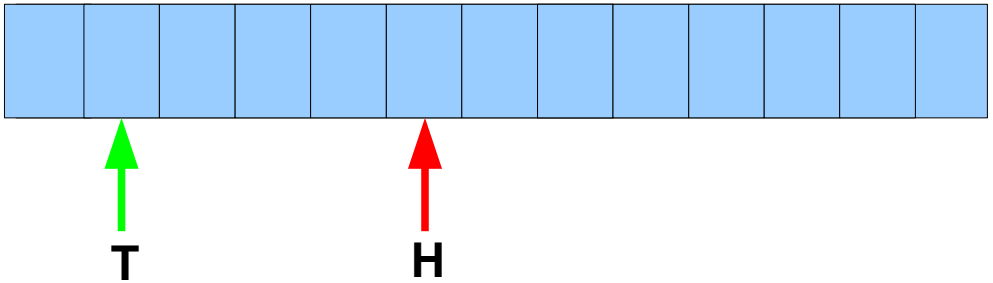
When the Tail or Head points the last element in queue the pointer is not increased (zero is written to the pointer) - circular buffer.

FIFO (3)

Empty FIFO T = H

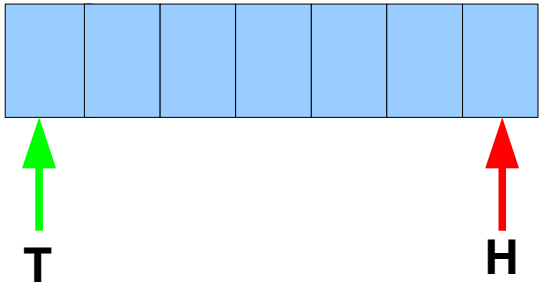


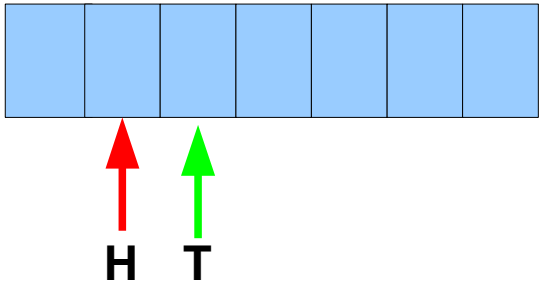
Some data in queue, amount of data = H – T



Full FIFO

(T = 0) & (H = Size) or T – H = 1





FIFO – implementation in C (1)

```
#define BUFFERSIZE   0xFF                    /* FIFO buffer size and mask */
typedef struct FIFO {
   char buffer [BUFFERSIZE+1];
   unsigned int head;
   unsigned int tail;
};
void FIFO_Init (struct FIFO *Fifo);
void FIFO_Empty (struct FIFO *Fifo);
int FIFO_Put (struct FIFO *Fifo, char Data);
int FIFO_Get (struct FIFO *Fifo, char *Data)

void FIFO_Init (struct FIFO *Fifo){
        Fifo->head=0;
        Fifo->tail=0;

        /* optional: initialize data in buffer with 0 */
}
```

FIFO – implementation in C (2)

```
void FIFO_Empty (struct FIFO *Fifo){
        Fifo->head = Fifo->tail;                                        /* now FIFO is empty*/
}
int FIFO_Put (struct FIFO *Fifo, char Data){
        if ((Fifo->tail-Fifo->head)==1 || (Fifo->tail-Fifo->head)==BUFFERSIZE)){
        return -1; };                                                   /* FIFO overflow */
        Fifo->buffer[Fifo->head] = Data;
        Fifo->head = (Fifo->head + 1) & BUFFERSIZE;
        return 1;                                                       /* Put 1 byte successfully */
}
int FIFO_Get (struct FIFO *Fifo, char *Data){
        If ((TxFifo.head!=TxFifo.tail)){
        *Data = Fifo->buffer[Fifo->tail];
        Fifo->tail = (Fifo->tail + 1) & BUFFERSIZE;
        return 1;                                                       /* Get 1 byte successfully */
        } else return -1;                                               /* No data in FIFO */
}
```

FIFO – traps

```
void FIFO_Empty (struct FIFO *Fifo){
        Fifo->head = Fifo->tail;                                        /* now FIFO is empty*/
}
int FIFO_Put (struct FIFO *Fifo, char Data){
        if ((Fifo->tail-Fifo->head)==1 || (Fifo->tail-Fifo->head)==BUFFERSIZE)){
        return -1; };                                                   /* FIFO overflow */
        Fifo->buffer[Fifo->head++] = Data;
        Fifo->head = Fifo->head & BUFFERSIZE;                           /* be carefull with interrupts */
        return 1;                                                       /* Put 1 byte successfully */
}
int FIFO_Get (struct FIFO *Fifo, char *Data){
        If ((TxFifo.head!=TxFifo.tail)){
        *Data = Fifo->buffer[Fifo->tail++];
        Fifo->tail &= BUFFERSIZE;                                       /* be carefull with interrupts */
        return 1;                                                       /* Get 1 byte successfully */
        } else return -1;                                               /* No data in FIFO */
}
```

Lecture Agenda

- Microprocessor systems, embedded systems
- ARM processors family
- Peripheral devices
- Memories and address decoders
- ARM processor as platform for embedded programs
- Methodology of designing embedded systems
- Interfaces in embedded systems
- Real-time microprocessor systems

From Acorn Computers Ltd. ARM to ARM Ltd.

Acorn

- Small company founded in November 1990,
  - Spun out of Acorn Computers (BBC Micro computer),
- Design the ARM range of RISC processor cores,
- ***ARM company does not fabricate silicon itself***,
- Licenses ARM cores to partners: **I**ntellectual **P**roperty **C**ores of ARM processors and peripheral devices,
- Develop tools (compilers, debuggers), starter-kits for embedded system development and creates standards, etc...

List of ARM silicon partners

Agilent, AKM, Alcatel, Altera, Atmel, Broadcom, Chip Express, Cirrus Logic, Digital Semiconductor, eSilicon, Fujitsu, GEC Plessey, Global UniChip, HP, Hyundai, IBM, Intel, ITRI, LG Semicon, LSI Logic, Lucent, Matsushita, Micrel, Micronas, Mitsubishi, Freescale, NEC, OKI, Philips, Qualcomm, Rockwell, Rohm, Samsung, Samsung, Sanyo, Seagate, Seiko Epson, Sharp, Sony, STMicroelectronics, Symbios Logic, Texas Instruments, Xilinx, Yamaha, Zeevo, ZTEIC, ...

History of ARM Processors

1983 – Sophie Wilson and Steve Furber fabricate the first RISC processor in Acorn Computers Limited, Cambridge, ARM = **A**corn (**A**dvanced) **R**ISC **M**achine

1985 – The first processor ARM 1 (architecture version v1)

1986 – First ARM 2 processors left company (32-bits, 26-bits address, 16 registers 16-bits, 30.000 transistors, architecture version v2/v2a, 8 MHz)

1990 – Apple Computer and VLSI Technology start work on the next version of ARM core,

1990 – New company is created Advanced RISC Machines Ltd. Responsible for the development of ARM cores,

1991 – The cooperation of Apple and VLSI Tech. provides new ARM 6 processor (ARM 610 applied in Apple Newton PDA, architecture version v3, 33 MHz)

1995 – ARM company offers famous **ARM7TDMI core** (core architecture **ARMv4T**) and Intel offers StrongARM (233 MHz)

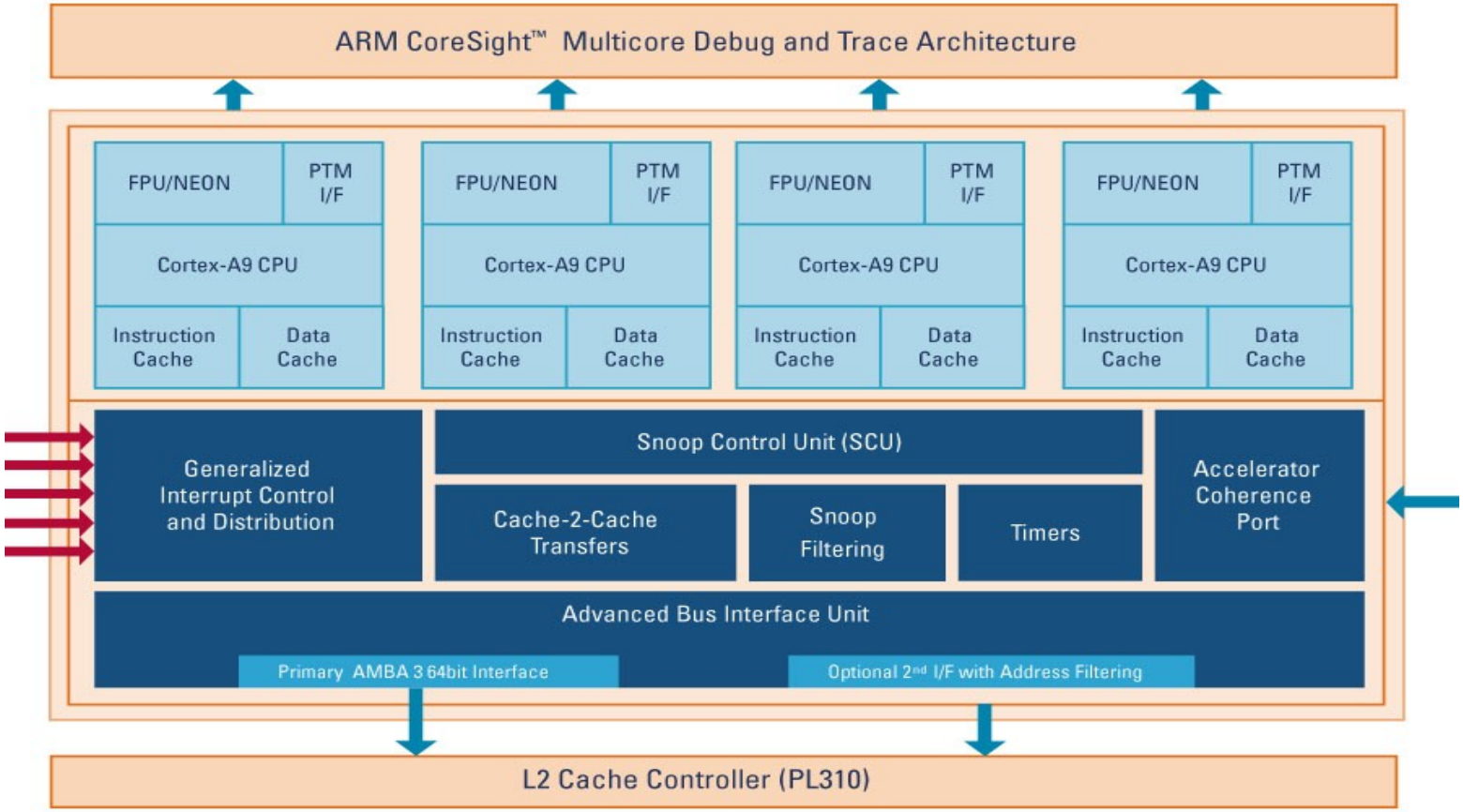
2001 – ARM company offers **ARM9TDMI core** (core architecture **ARMv5TEJ**, 220 MHz)

2004 – Cortex M3 processor (ARMv7-M, 100 MHz)

2008 – ARM Cortex A8 (core architecture ARMv7, 1 GHz)

now – ARM Cortex A9/A15 – MPCore architecture

ARM Cortex A9 in MPCore Configuration



New MPCore technology allows to design SoC – four A9 cores

Processors with ARM Core

- ARM processors are widely used in embedded systems and mobile devices that require low power devices
- The ARM processor is the most commonly used device in the World. You can find the processor in hard discs, mobile phones, routers, calculators and toys,
- Currently, more than 75% of 32-bits embedded CPUs market belongs to ARM processors,
- The most famous and successful processor is ARM7TDMI, very often used in mobile phones,
- Processing power of ARM devices allows to install multitasking operating systems with TCP/IP software stack and filesystem (e.g. FAT32).
- The known operating systems for ARM processors: embedded Linux (Embedded Debian, Embedded Ubuntu), Windows CE, Symbian, NUTOS (Ethernut), RTEMS,...

ARM Powered Products

Comparison of Selected ARMs

Family	Architecture Version	Core	Feature	Cache (I/D)/MMU	Typical MIPS @ MHz
ARM6	ARMv3	ARM610	Cache, no coprocessor	4K unified	17 MIPS @ 20 MHz
ARM7	ARMv3	ARM7500FE	Integrated SoC. "FE" Added FPA and EDO memory controller.	4 KB unified	55 MIPS @ 56 MHz
ARM7TDMI	ARMv5TEJ	ARM7EJ-S	Jazelle DBX, Enhanced DSP instructions, 5-stage pipeline	8 KB	120 MIPS @ 133 MHz
StrongARM	ARMv4	SA-110	5-stage pipeline, MMU	16 KB/16 KB, MMU	235 MIPS @ 206 MHz
ARM8	ARMv4	ARM810[7]	5-stage pipeline, static branch prediction, double-bandwidth memory	8 KB unified, MMU	1.0 DMIPS/MHz
ARM9TDMI	ARMv4T	ARM920T	5-stage pipeline	16 KB/16 KB, MMU	245 MIPS @ 250 MHz
ARM9E	ARMv5TEJ	ARM926EJ-S	Jazelle DBX, Enhanced DSP instructions	variable, TCMs, MMU	220 MIPS @ 200 MHz
ARM10E	ARMv5TE	ARM1020E	VFP, 6-stage pipeline, Enhanced DSP instructions	32 KB/32 KB, MMU	300 MIPS @ 325 MHz
XScale	ARMv5TE	PXA27x	MMX and SSE instruction set, four MACs,	32 Kb/32 Kb, MMU	800 MIPS @ 624 MHz
ARM11	ARMv6	ARM1136J(F)-S	SIMD, Jazelle DBX, VFP, 8-stage pipeline	variable, MMU	740 @ 532-665 MHz
Cortex	ARMv7-A	Cortex-A8	Application profile, VFP, NEON, Jazelle RCT, Thumb-2, 13-stage superscalar pipeline	variable (L1+L2), MMU+TrustZone	>1000 MIPS@ 600 M-1 GHz

ARM Processor Core

ARM architecture (1)

ARM processor core – processor designed according to ARM processor architecture described in high level description language (VHDL lub Verilog) provided as macro-cell or Intellectual Property (IP).

Features of ARM processor cores:

- Supposed to be used for further development – microcontroller, SoC
- 32-bits RISC architecture
- Optimised for low power consumption
- Support three different modes of operation:
  - ARM instructions, 32 bits,
  - Thumb instructions, 16 bits,
  - Jazelle DBX - Direct java instructions.
- Supported Big or Little Endian
- Fast Interrupt Response mode for Real-time applications
- Virtual memory
- List of efficient and powerful instructions selected from both RISC and CISC architectures
- Hardware support for higher level software (Ada, C, C++)

ARM architecture (2)

Nomenclature:

ARM {x} {y} {z} {T} {D} {M} {I} {E} {J} {F} {S}

- **x** – core family
- **y** – implemented Memory Management Unit
- **z** – cache memory
- **T** – Thumb mode (16 bit command)
- **D** – Build in debugger, (usually via JTAG interface)
- **M** – Build in multiplier, hardware multiplier (32x32 => 64 bits)
- **I** – In-Circuit Emulator, another ICE debugger
- **E –** Enhanced DSP instructions, Digital Signal Processing
- **J** – Jazelle mode
- **F** – Floating-point unit
- **S** – Synthesizable version, available source code for further synthesis and EDA tools

Example of ARM cores:

ARM7TDMI **ARM9TDMI-EJ-S**

ARM architecture (3)

- **Core in version 1, v1**
  - Base arithmetic and logic operations,
  - Hardware interrupts,
  - 8 and 32 bits operations,
  - 26 bits address
- **Core in version 2, v2**
  - Implemented Multiply ACcumulate unit,
  - Available coprocessor,
  - Additional commands for threads synchronisation ,
  - 26 bits address
- **Core in version 3, v3**
  - New registers CPSR, SPSR, MRS, MSR,
  - Additional modes Abort and Undef,
  - 32 bits address

ARM architecture (4)

- **Core in version 4, v4**
  - First standardised architecture
  - Available 16 bits operations
  - THUMB - new mode of operation, 16 bits commands
  - Added privileged mode
  - PC can be incremented by 64 bits
- **Core in version 5, v5**
  - Improved cooperation between ARM and THUMB modes, mode of operation can be changed during program execution,
  - Added instruction CLZ
  - Software breakpoints
  - Support for multiprocessor operation
- **Core in version 6, v6**
  - Improved MMU (Management Memory Unit)
  - Hardware support for video and sound processing (FFT, MPEG4, SIMD etc...)
  - Improved exception handing (new flag in PSR)

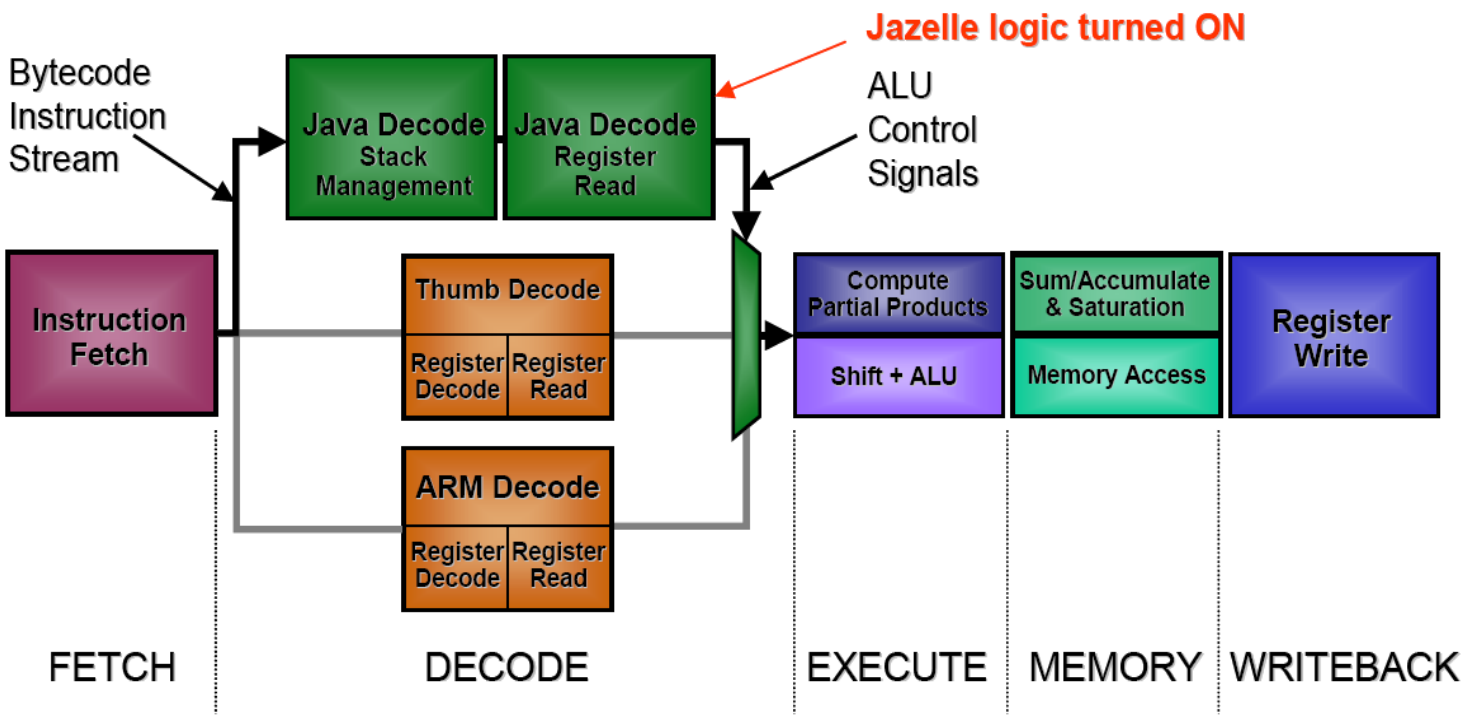
ARM instruction sets

Taking into consideration executed commands ARM processor can operate in one of the following modes:

- **ARM** – 32-bits instructions optimised for time execution (code must be aligned to 4 bytes),
- **Thumb**, **Thumb-2** – 16-bits instructions optimised for code size (code must be aligned to 2 bytes, processor registers are still 32 bits wide),
- **Jazelle v1** – mode used for direct execution of Java code (without virtual machine JVM) (1000 Caffeine Marks @ 200MHz)

Support for Java language

- ARM core marked with 'J'
- Dynamic exchange of registers and stack
- Hardware decoder of Java instructions

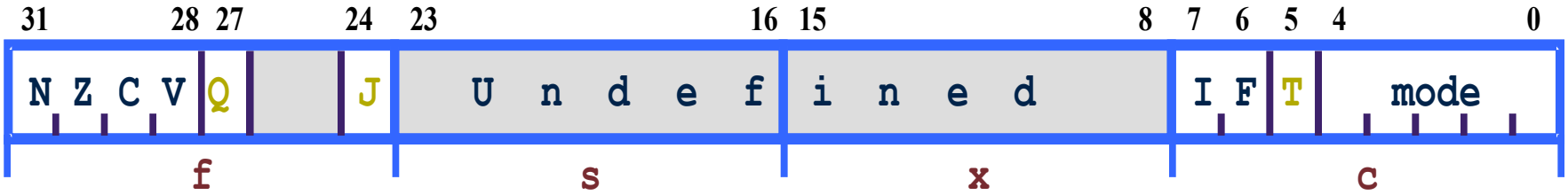


Programming Model – Registers

ARM Processor provides 37 registers (all are 32-bits wide). The registers are arranged into several banks (accessible bank being governed by the current processor mode):

- **PC (r15)** – Program Counter
- **CPSR** – Main status register, Current Program Status Register
- **SPSR** – Copy of status register, available in different modes of operation Saved Program Status Register
- **LR (r14)** – Link Register, used for stack frame during execution of subroutines or return address register
- **SP (r13)** – used as a Stack Pointer
- **r0 - r12** – General purpose registers (dependent of the mode of operation)

Program Status Register



Condition code flags

- V – ALU operation o**V**erflowed
- C – ALU operation **C**arried out
- Z – **Z**ero result from ALU operation
- N – **N**egative result from ALU operation

Flags for processor from family 5TE/J

- J – Processor in Jazelle mode
- Q – Sticky Overflow – saturation flag, set during ALU operations (QADD, QDADD, QSUB or QDSUB, or operation of SMLAxy, SMLAWx, result more than 32 bits)

Interrupt disable bits

- I=1 Disables the IRQ
- F=1 Disables the FIQ

Flags for xT architecture

- T=0 Processor in ARM mode
- T=1 Processor in Thumb mode

Mode bits

- Specify the processor operation mode (seven modes)

Read/Modify/Write strategy should be used to write data to PSR (to ensure further compatibility)

Programming Model – modes of processor operation

Operating mode – defined which resources of processor are available, e.g. registers, memory regions, peripheral devices, stack, etc...

ARM processor can operate in on of 7 modes:

- **User –** user mode (not privileged), dedicated for user programs execution
- **FIQ –** fast interrupts and high priority exceptions (used only when really necessary)
- **IRQ –** handling of low or normal priority interrupts
- **Supervisor** – supervisor mode gives access to all resource of the processor, used during debugging. Available after reset or during interrupt handling.
- **Abort –** used for handling of memory access exceptions (memory access violations)
- **Undef –** triggered when unknown or wrong commands is detected
- **System –** privileged mode, access to registers as in user mode, however various memory segments are available

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

Programming Model – registers available in User or System modes

Current Visible Registers

User Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr

Banked out Registers

FIQ	IRQ	SVC	Undef	Abort
r8				
r9				
r10				
r11				
r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr	spsr

Programming Model – registers available in FIQ mode

Current Visible Registers

FIQ Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

User

r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)

Banked out Registers

IRQ	SVC	Undef	Abort
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr

Programming Model – registers available in IRQ mode

Current Visible Registers

IRQ Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

Banked out Registers

User	FIQ	SVC	Undef	Abort
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Programming Model – registers available in Supervisor mode

Current Visible Registers

SVC Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

Banked out Registers

User	FIQ	IRQ	Undef	Abort
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Programming Model – registers available in Abort mode

Current Visible Registers

Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

Banked out Registers

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Programming Model – registers available in Undef mode

Current Visible Registers

Undef Mode

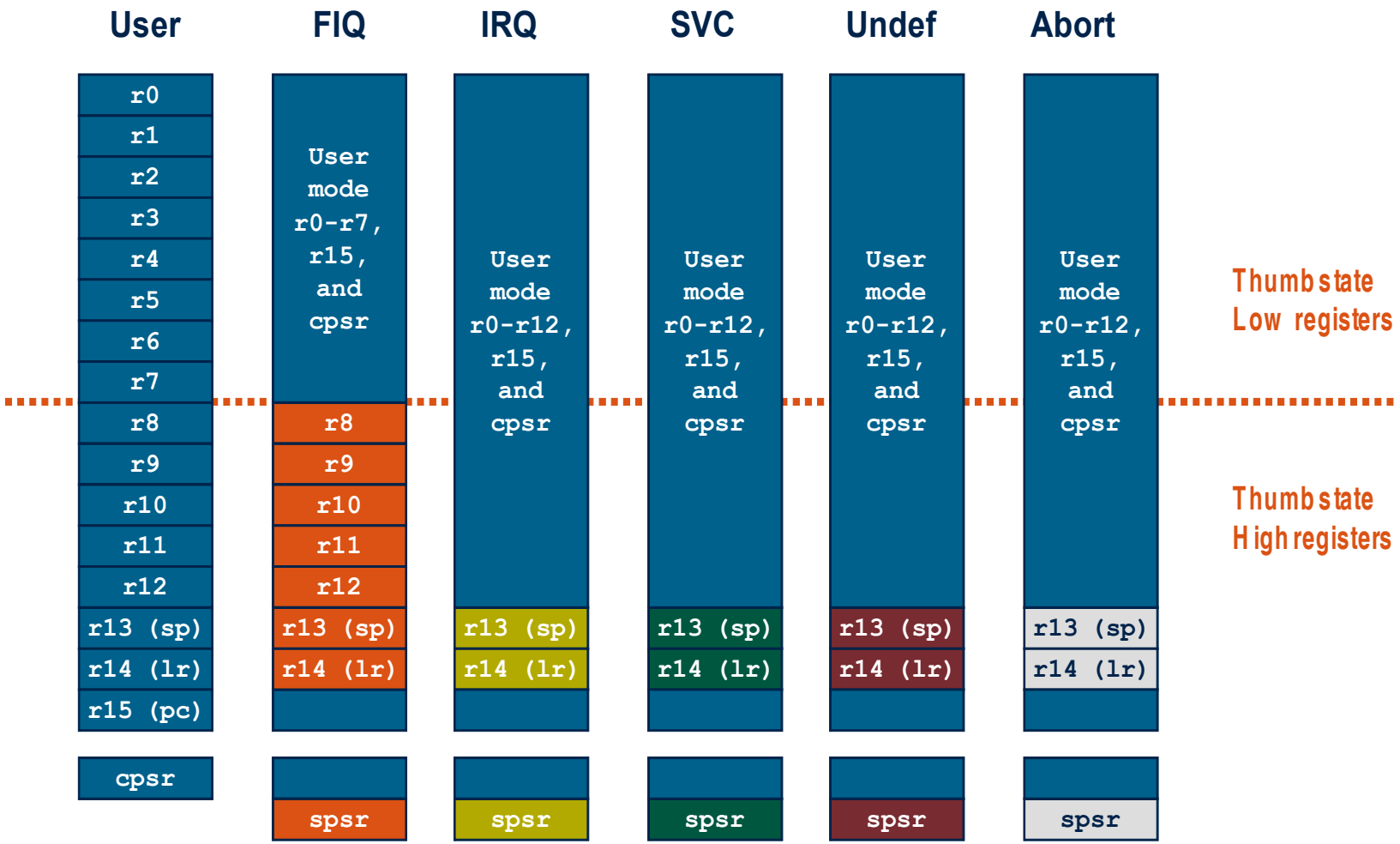
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

Banked out Registers

User	FIQ	IRQ	SVC	Abort
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

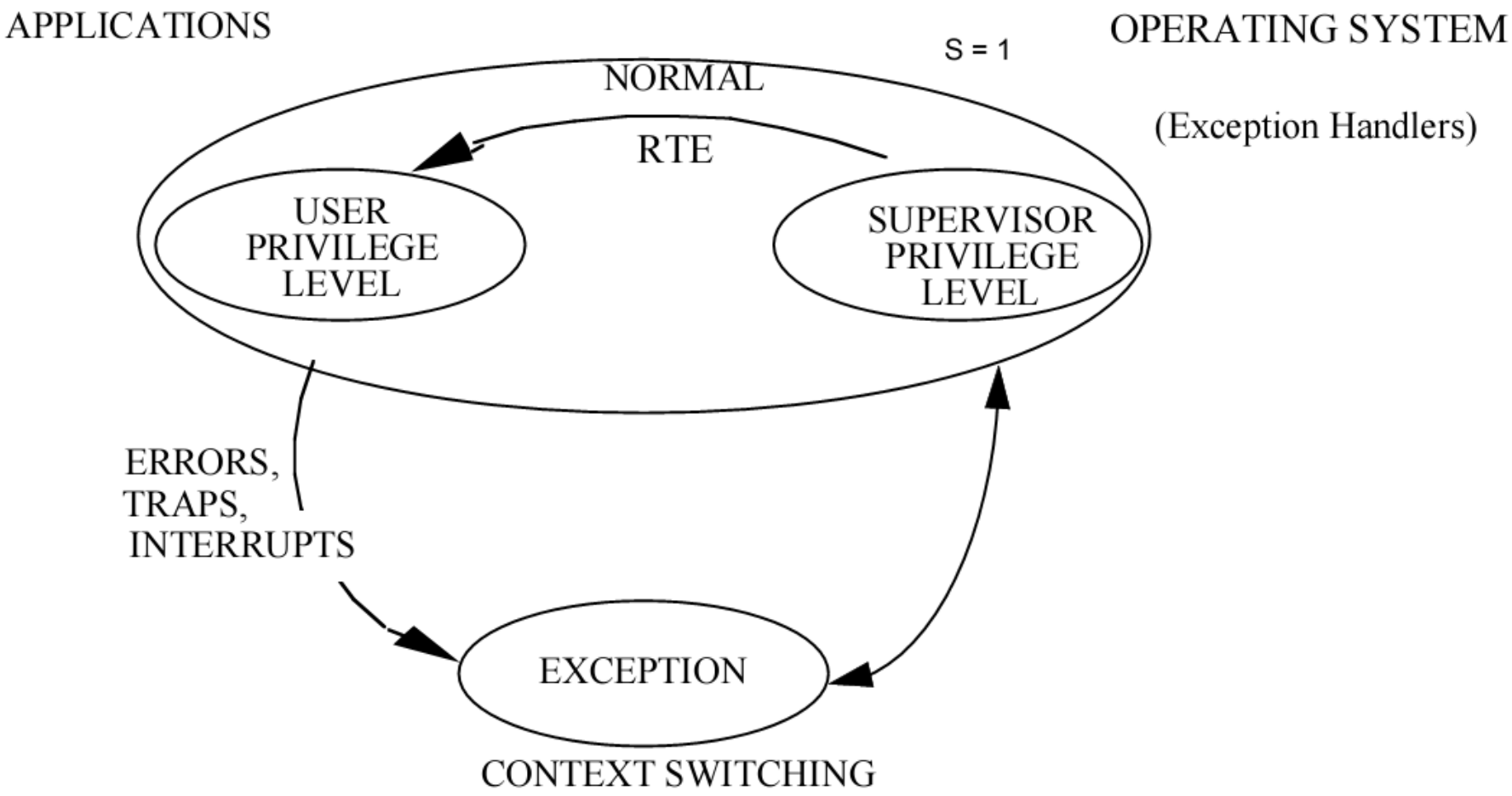
Programming Model – registers summary



Note: System mode uses the User mode register set

Interrupts and Exceptions

Handling of Exceptions



Exception

Exception – mechanism that control flow of data used in microprocessors-based systems and programming languages to handling asynchronous and unpredictable situations.

Exceptions can be divided into:

- Faults,
- Aborts,
- Traps.

In addition to exceptions processor supervises also interrupts.

ARM processors can handle two different modes of interrupts:

- **FIQ** - Fast interrupt (interrupt with low latency handling),
- **IRQ** - Normal Interrupt.

Interrupts

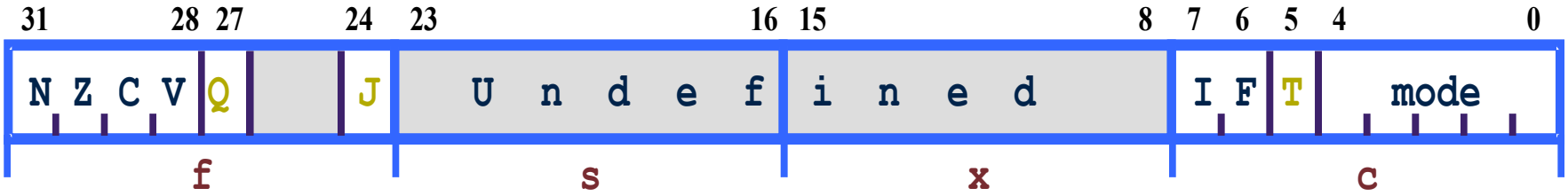
Interrupt or IRQ – Interrupt ReQuest – is an asynchronous signal indicating the need for attention or a synchronous event in software indicating the need for a change in execution. A hardware interrupt causes the processor to save its state of execution and begin execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

Examples of interrupts:

- Receive or transmission of data via serial interface (e.g. EIA RS232),
- Change of state or detected slope on processor's pin.

Status of device can be checked using software commands, however it requires continuous reading and checking of status register of the device. This operation is called polling. Even simple polling usually requires a significant amount of processing power and unnecessary loads processor, e.g. transmission of single symbol lasts ~100 us (processor can execute hundreds of thousands of instructions during this time).

Program Status Register



Condition code flags

- V – ALU operation o**V**erflowed
- C – ALU operation **C**arried out
- Z – **Z**ero result from ALU operation
- N – **N**egative result from ALU operation

Flags for processor from family 5TE/J

- J – Processor in Jazelle mode
- Q – Sticky Overflow – saturation flag, set during ALU operations (QADD, QDADD, QSUB or QDSUB, or operation of SMLAxy, SMLAWx, result more than 32 bits)

Interrupt disable bits

- I=1 Disables the IRQ
- F=1 Disables the FIQ

Flags for xT architecture

- T=0 Processor in ARM mode
- T=1 Processor in Thumb mode

Mode bits

- Specify the processor operation mode (seven modes)

Handling of exceptions

Execution of not allowed operation in given processor mode can cause exception, e.g. access to protected memory segment.

Handling of exception covers all operations when the exception was detected until the first command of exception handler.

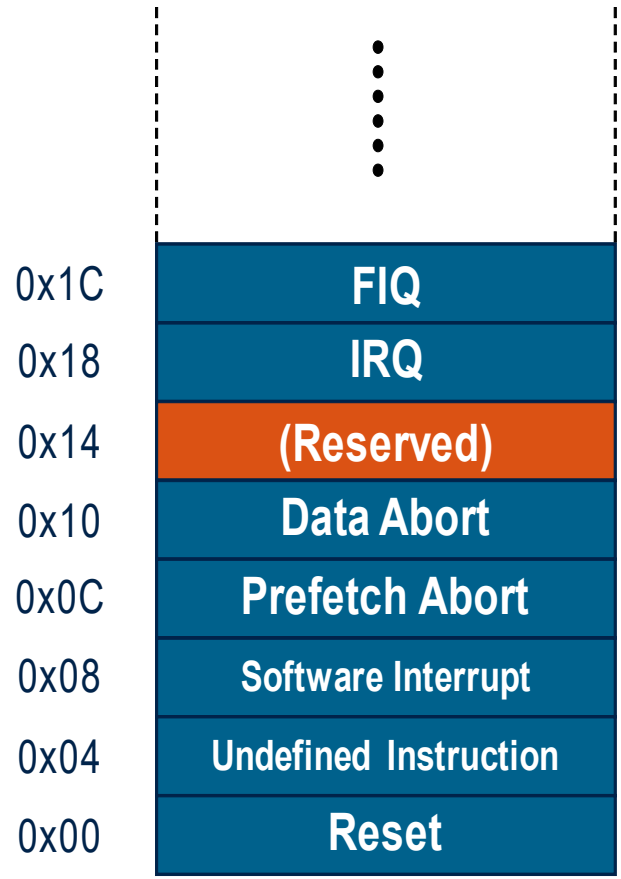
1. a) Change operating mode to ARM (from Thumb or Jazelle),
   b) Change to interrupt of exception mode (FIQ/IRQ),
   c) Set interrupt level mask on level equal to the handling interrupt (disable interrupts).
   d) Change registers bank:
      make a copy of CPSR → SPSR and PC (r15) → Link Register (r14),
   e) Make active SPSR register.
2. Calculate exception vector (interrupt).
3. Branch to the first instruction handling exception or interrupt.
4. Return from exception/interrupt:
   a) Recover CPSR (r15) register,
   b) Recover PC (Link Register r14),
   c) Return to the interrupted program.

Exceptions (1)

Exception handling by the ARM processor is controlled through the use of an area of memory called the vector table. This lives (normally) at the bottom of the memory map from 0x0 to 0x1c. Within this table one word is allocated to each of the various exception types. This word will contain some form of ARM instruction that should perform a branch. It does **not** contain an address.

When one of these exceptions is taken, the ARM goes through a low-overhead sequence of actions in order to invoke the appropriate exception handler. The current instruction is always allowed to complete (except in case of Reset).

IRQ is disabled on entry to all exceptions; FIQ is also disabled on entry to Reset and FIQ.



Memory image

Exceptions (2)

Reset - executed on power on

Undef - when an invalid instruction reaches the execute stage of the pipeline

SWI - when a software interrupt instruction is executed

Prefetch - when an instruction is fetched from memory that is invalid for some reason, if it reaches the execute stage then this exception is taken

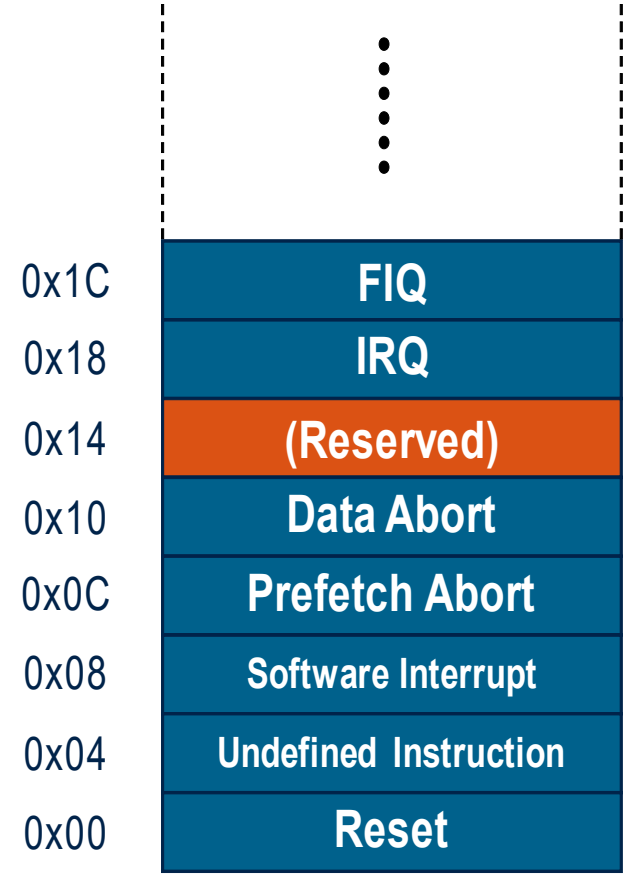
Data - if a load/store instruction tries to access an invalid memory location, then this exception is taken

IRQ - normal interrupt

FIQ - fast interrupt

Vector table is located in memory address 0x0.

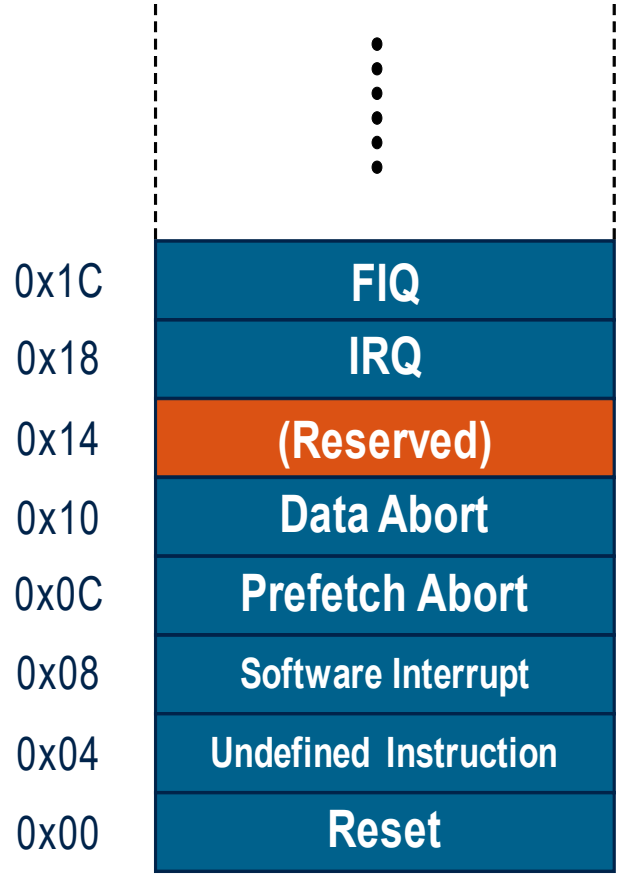
The base address of exception table can be modified: 0xFFFF.0000 (ARM 7/9/10).



Memory image

Exceptions Table

```
LDR     PC, =FIQ_Addr
LDR     PC, =IRQ_Addr
NOP ; Reserved vector
LDR     PC, =Abort_Addr
LDR     PC, =Prefetch_Addr
LDR     PC, =SWI_Addr
LDR     PC, =Undefined_Addr
LDR     PC, =Reset_Addr
```



Memory image

Exception Handlers (1)

```
IRQ_Addr:
/*- Manage Exception Entry */
/*- Adjust and save LR_irq in IRQ stack */
        sub     lr, lr, #4
        stmfd   sp!, {lr}
/*- Save r0 and SPSR in IRQ stack */
        mrs     r14, SPSR
        stmfd   sp!, {r0,r14}
/*- Write in the IVR to support Protect Mode */
/*- No effect in Normal Mode */
/*- De-assert the NIRQ and clear the source in Protect Mode */
        ldr     r14, =AT91C_BASE_AIC
        ldr     r0 , [r14, #AIC_IVR]
        str     r14, [r14, #AIC_IVR]
...
/*- Branch to the routine pointed by the AIC_IVR */
        mov     r14, pc
        bx      r0                      /* Branch to IRQ handler */
...
/*- Restore adjusted  LR_irq from IRQ stack directly in the PC */
        ldmia   sp!, {pc}^              /* ^ - Recover CSPR */
```

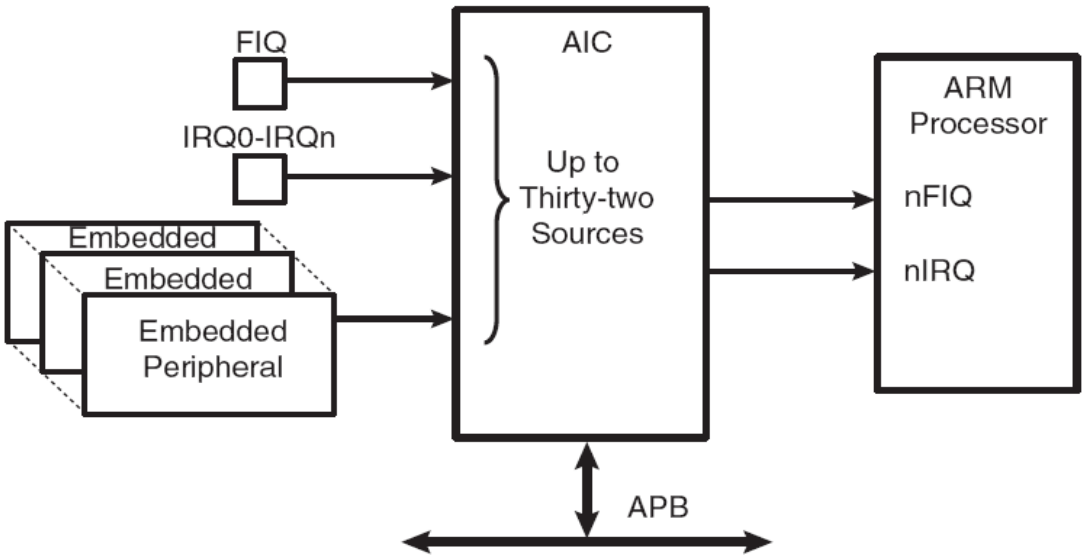
Exception Handlers (2)

```
/* lowlevel.c */
/*-----------------------------------------------------------------------------
 * Function Name       : default_spurious_handler
 * Object              : default handler for spurious interrupt
 *----------------------------------------------------------------------------*/
void default_spurious_handler(void)
{
    dbgu_print_ascii("-F- Spurious Interrupt\n\r ");
    while (1);
}

/*-----------------------------------------------------------------------------
 * Function Name       : default_fiq_handler
 * Object              : default handler for fast interrupt
 *----------------------------------------------------------------------------*/
void default_fiq_handler(void)
{
    dbgu_print_ascii("-F- Unexpected FIQ Interrupt\n\r ");
    while (1);
}
```

Advanced Interrupt Controller

Block diagram of AIC of ARM processor



- Manages vectorised interrupts,
- Can monitor up to 32 internal and external interrupts,
- Each interrupt can be disabled/enabled (masked),
- Handles normal nIRQ and fast nFIR interrupts,
- 8 priority levels (0 – the lowest, 7 – the highest),
- Handles interrupts triggered with level or edge.

Advanced Interrupt Controller of ARM processor

- AIC uses system clock, however the clock signal cannot be disabled to save power.
- Interrupts can be used to wake up processor from sleep or hibernation mode.
- Interrupt with number 0 (FIQ) is always FIQ type.
- Interrupt with number 1 (SYS) is logic sum of a few interrupts of internal peripheral devices of ARM core, programmer control priority and select interrupts
- Interrupts with numbers 2-31 (PID2-PID331) can be used for others internal and external devices and I/O ports.
- AIC is able to supervise interrupts triggered by selected level or edge.

Shared Interrupts

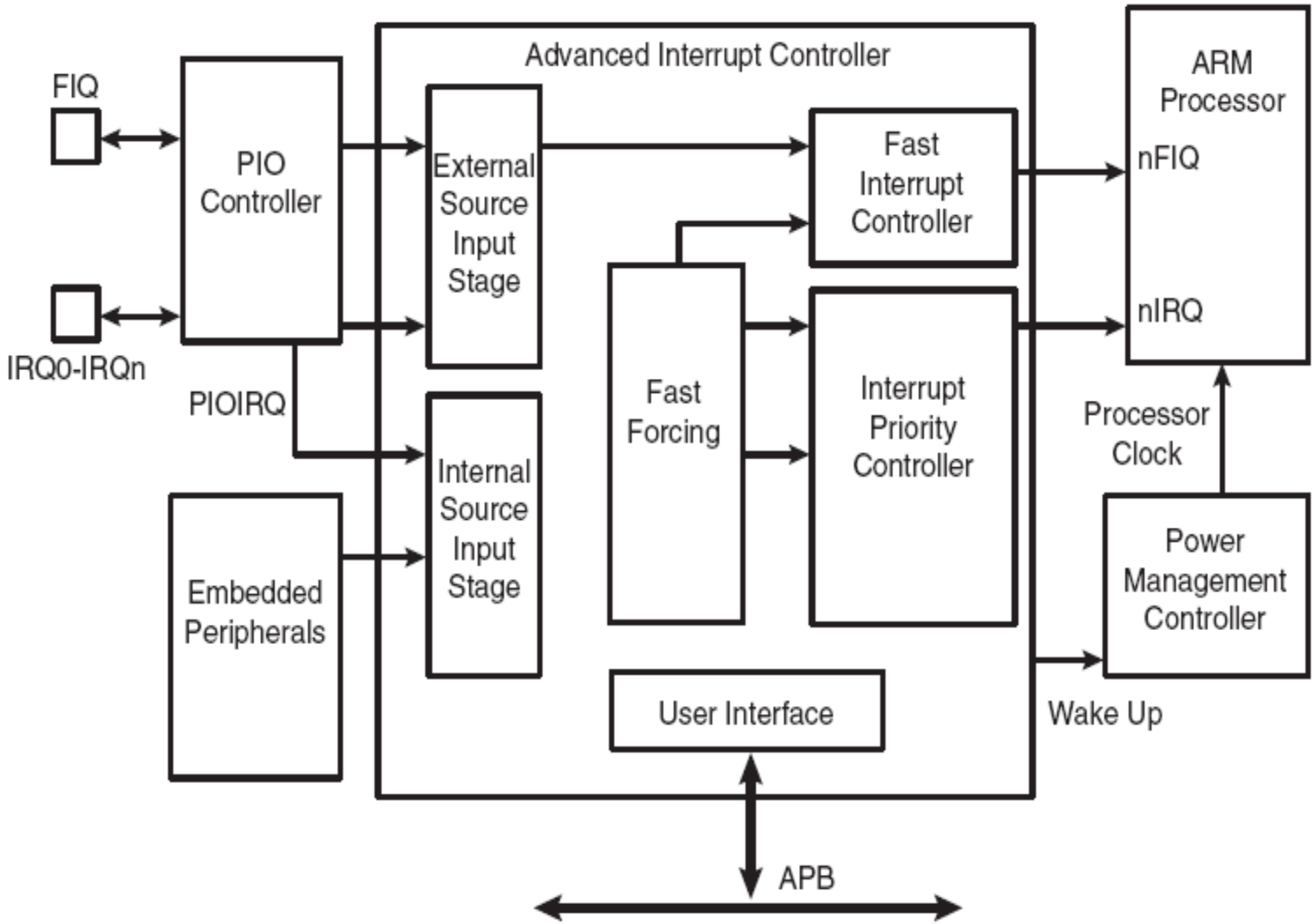
Internal peripheral devices use a single system shared interrupt SYS (number defined by constant AT91C_ID_SYS = 1).

Devices handled by system interrupt:

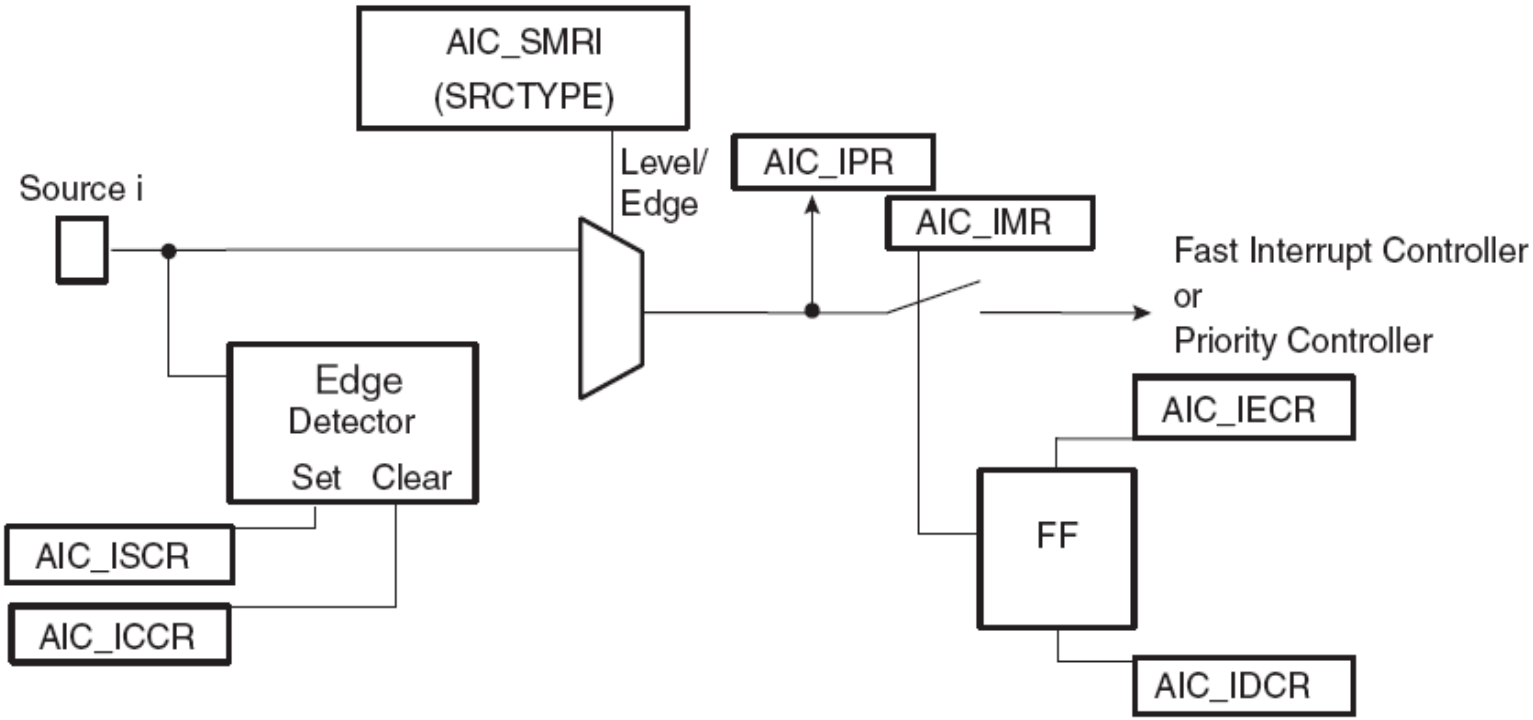
- Timers PIT, RTT, WDT,
- Diagnostic interface (DBGU),
- DMA controller (PMC),
- Reset circuit (RSTC),
- Memory Controller (MC).

Therefore, the SYS handler should check state of all interrupts and execute functions-handlers for the active interrupts (mask register AIC_MSK).

Block diagram of AIC

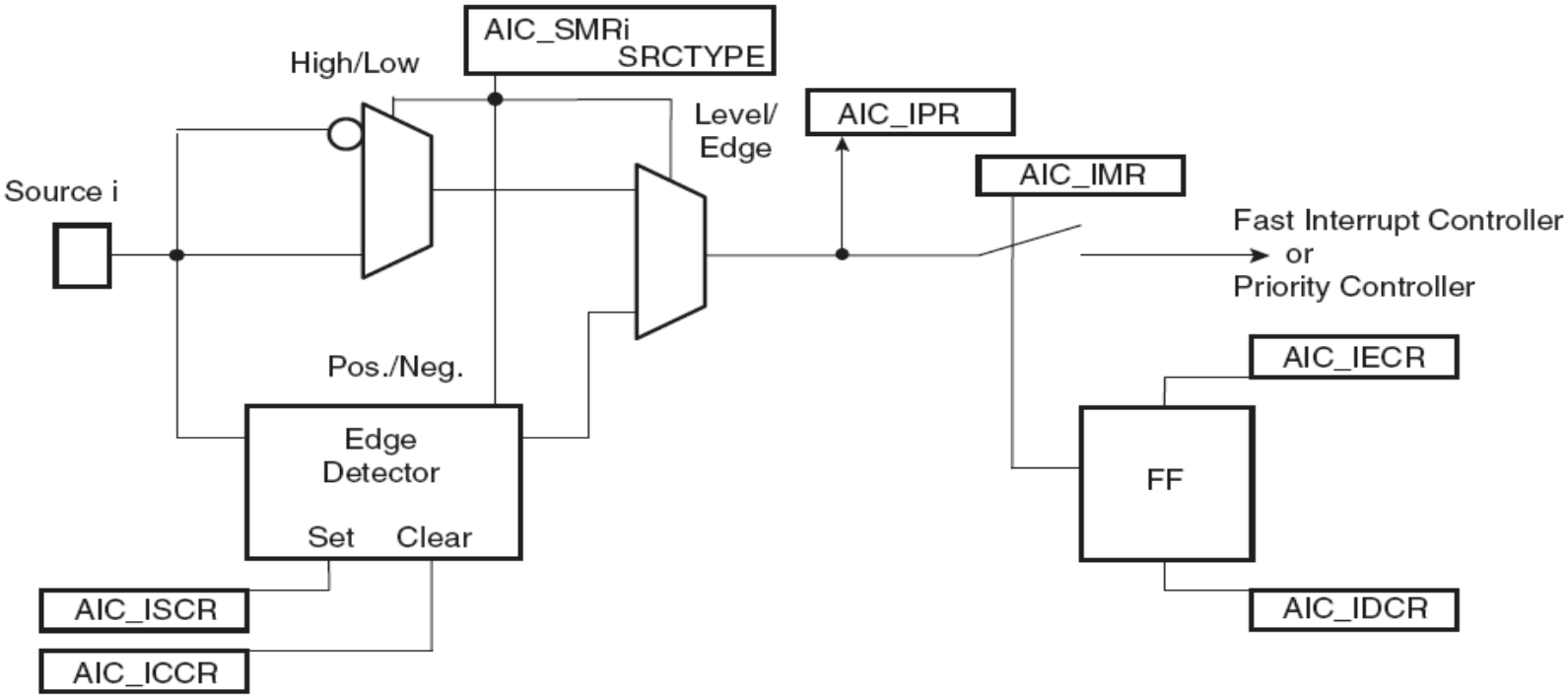


Internal Interrupts



- IRQ mask – AIC_IECR/IDCR (status → AIC_IMR),
- Clear interrupt flag when AIC_IVR register is read (for FIQ → AIC_FVR),
- Interrupt status available in AIC_IPR
- Interrupt can be triggered by high level or rising edge

External Interrupts



- User can select method of triggering: level (high, low) or edge (rising, falling)

ID Numbers for Peripheral Devices

```
// ****************************************************************************
//              PERIPHERAL ID DEFINITIONS FOR AT91SAM9263
// ****************************************************************************
#define AT91C_ID_FIQ    ( 0) // Advanced Interrupt Controller (FIQ)
#define AT91C_ID_SYS    ( 1) // System Controller
#define AT91C_ID_PIOA   ( 2) // Parallel IO Controller A
#define AT91C_ID_PIOB   ( 3) // Parallel IO Controller B
#define AT91C_ID_PIOCDE ( 4) // Parallel IO Controller C, Parallel IO Controller D, Parallel IO Controller E
#define AT91C_ID_US0    ( 7) // USART 0
#define AT91C_ID_US1    ( 8) // USART 1
#define AT91C_ID_US2    ( 9) // USART 2
#define AT91C_ID_MCI0   (10) // Multimedia Card Interface 0
#define AT91C_ID_MCI1   (11) // Multimedia Card Interface 1
#define AT91C_ID_CAN    (12) // CAN Controller
#define AT91C_ID_TWI    (13) // Two-Wire Interface
#define AT91C_ID_SPI0   (14) // Serial Peripheral Interface
```

ID=0, ID=30-31 external interrupts, others are internal

Registers of AIC (1)

Offset	Register	Name	Access	Reset
0x00	Source Mode Register 0	AIC_SMR0	Read-write	0x0
0x04	Source Mode Register 1	AIC_SMR1	Read-write	0x0
---	---	---	---	---
0x7C	Source Mode Register 31	AIC_SMR31	Read-write	0x0
0x80	Source Vector Register 0	AIC_SVR0	Read-write	0x0
0x84	Source Vector Register 1	AIC_SVR1	Read-write	0x0
---	---	---	---	---
0xFC	Source Vector Register 31	AIC_SVR31	Read-write	0x0
0x100	Interrupt Vector Register	AIC_IVR	Read-only	0x0
0x104	FIQ Interrupt Vector Register	AIC_FVR	Read-only	0x0
0x108	Interrupt Status Register	AIC_ISR	Read-only	0x0
0x10C	Interrupt Pending Register(3)	AIC_IPR	Read-only	0x0(1)
0x110	Interrupt Mask Register(3)	AIC_IMR	Read-only	0x0
0x114	Core Interrupt Status Register	AIC_CISR	Read-only	0x0
0x118 - 0x11C	Reserved	---	---	---
0x120	Interrupt Enable Command Register(3)	AIC_IECR	Write-only	---
0x124	Interrupt Disable Command Register(3)	AIC_IDCR	Write-only	---
0x128	Interrupt Clear Command Register(3)	AIC_ICCR	Write-only	---
0x12C	Interrupt Set Command Register(3)	AIC_ISCR	Write-only	---
0x130	End of Interrupt Command Register	AIC_EOICR	Write-only	---
0x134	Spurious Interrupt Vector Register	AIC_SPU	Read-write	0x0
0x138	Debug Control Register	AIC_DCR	Read-write	0x0
0x13C	Reserved	---	---	---
0x140	Fast Forcing Enable Register(3)	AIC_FFER	Write-only	---
0x144	Fast Forcing Disable Register(3)	AIC_FFDR	Write-only	---
0x148	Fast Forcing Status Register(3)	AIC_FFSR	Read-only	0x0
0x14C - 0x1E0	Reserved	---	---	---
0x1EC - 0x1FC	Reserved			

Registers of AIC – mapped as struct

```
typedef struct _AT91S_AIC {
    AT91_REG    AIC_SMR[32];    // Source Mode Register
    AT91_REG    AIC_SVR[32];    // Source Vector Register
    AT91_REG    AIC_IVR;        // IRQ Vector Register
    AT91_REG    AIC_FVR;        // FIQ Vector Register
    AT91_REG    AIC_ISR;        // Interrupt Status Register
    AT91_REG    AIC_IPR;        // Interrupt Pending Register
    AT91_REG    AIC_IMR;        // Interrupt Mask Register
    AT91_REG    AIC_CISR;       // Core Interrupt Status Register
    ...
} AT91S_AIC, *AT91PS_AIC;
#define AT91C_BASE_AIC      (AT91_CAST(AT91PS_AIC)    0xFFFFF000) // (AIC)
    Base Address
```

Registers of AIC (2)

AIC_SMR[32]; // Source Mode Register – configure method of int triggering, priority

AIC_SVR[32]; // Source Vector Register – 32-bit addresses for int handlers

AIC_IVR; // IRQ Vector Register – address of currently handled normal interrupt

AIC_FVR; // FIQ Vector Register – address of currently handled fast interrupt

AIC_ISR; // Interrupt Status Register – number of currently handled interrupt

AIC_IPR; // Interrupt Pending Register – register with pending interrupts, bits 0-31

AIC_IMR; // Interrupt Mask Register – register with masks for interrupts, bits 0-31

AIC_CISR; // Core Interrupt Status Register – status for IRQ/FIQ core interrupts

AIC_IECR; // Interrupt Enable Command Register – register for enabling interrupts

AIC_IDCR; // Interrupt Disable Command Register – register for disabling interrupts

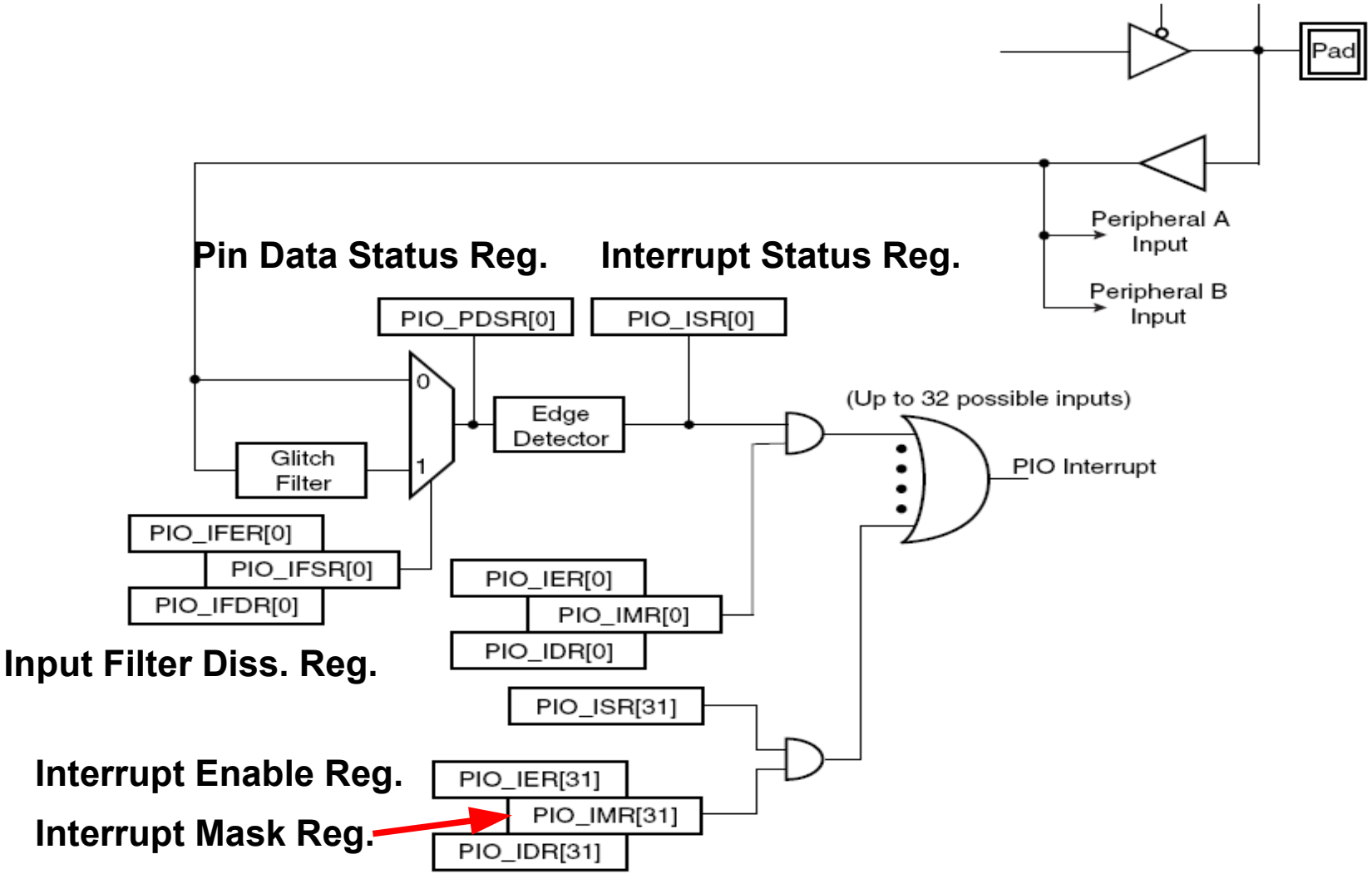
AIC_ICCR; // Interrupt Clear Command Register – register for deactivating interrupts

AIC_ISCR; // Interrupt Set Command Register – register for triggering interrupts

AIC_EOICR; // End of Interrupt Command Register – inform that INT treatment is finished

AIC_SPU; // Spurious Vector Register – handler for spurious interrupt

I/O – Interrupts



Keyboard interrupts configuration

Buttons are connected to Port C – interrupt generated by input signals of ports C/D/E (use mask AT91C_ID_PIOCDE)

Configuration of interrupts for C/D/E port(s):

1. Configure both ports as inputs (left and right hand buttons), activate clock signal
2. **Turn off interrupts for port C/D/E (register AIC_IDCR, mask AT91C_ID_PIOCDE)**
3. Configure pointer for C/D/E port interrupt handler – use AIC_SVR table
   AIC_SVR[AT91C_ID_PIOCDE] = ...
4. Configure method of interrupt triggering: high level, (AIC_SMR register, triggered by AT91C_AIC_SRCTYPE_EXT_HIGH_LEVEL and priority, e.g. AT91C_AIC_PRIOR_HIGHEST)
5. **Clear interrupt flag for port C/D/E (register AIC_ICCR)**
6. Turn on interrupts for both input ports (register PIO_IER)
7. Turn on interrupts for C/D/E port (register AIC_IECR)

INT Handler for Keyboard

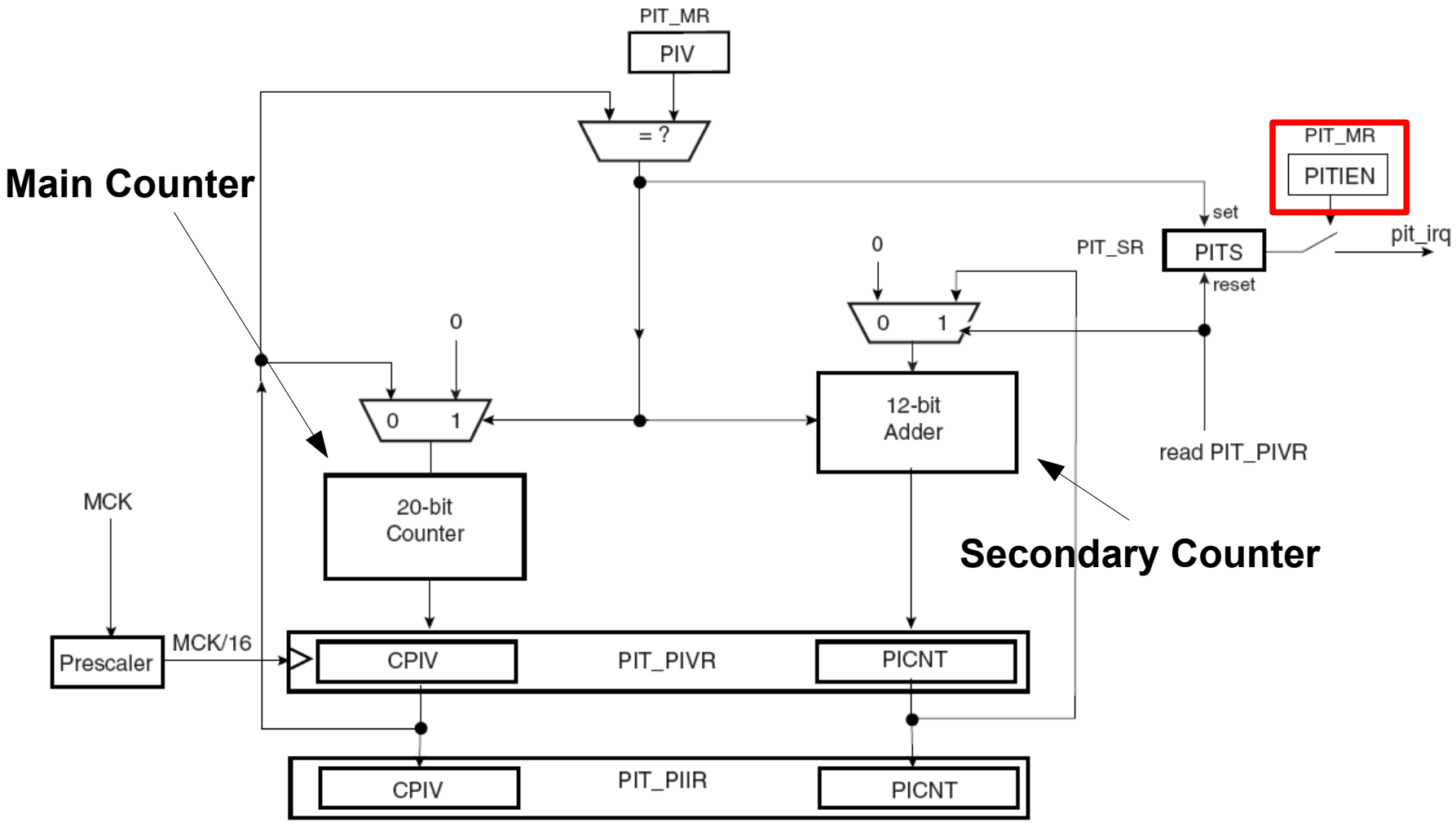
Set address for interrupt function (handler) for the interrupt (32-bits address)

AT91C_BASE_AIC->AIC_SVR[AT91C_ID_SYS] = (unsigned int) BUTTON_IRQ_handler;

Keyboard interrupt handler

```
void BUTTON_IRQ_handler (void) {
        If flag on the suitable bit-position is active the button is/was pressed (PIO_ISR)
                Read PIO_ISR status register to clear the flag
}
```

Interrupt from PIT



PIT Timer interrupts configuration

PIT Timer generates system interrupt (ID number 1) – interrupt from processor peripheral devices (System Controller, mask AT91C_ID_SYS)

Configuration of PIT Timer interrupts:

1. Calculate time counter value for defined period of time, e.g. 5 ms
2. Disable PIT Timer interrupts – only during configuration (AIC_IDCR, interrupt nr 1 – processor peripheral devices, used defined constant AT91C_ID_SYS)
3. Configure pointer for timer interrupt handler – handler for processor peripheral devices, see AIC_SVR table (AIC_SVR[AT91C_ID_SYS])
4. Configure method of interrupt triggering: level, edge, (AIC_SMR register, triggered by AT91C_AIC_SRCTYPE_INT_LEVEL_SENSITIVE, and priority, e.g. AT91C_AIC_PRIOR_LOWEST)
5. Clear interrupt flag of peripheral devices (AIC_ICCR register)
6. Turn on the interrupt AT91C_ID_SYS (AIC_IECR register)
7. Turn on PIT Timer interrupt (AT91C_PITC_PITIEN register)
8. Turn on PIT Timer (AT91C_PITC_PITEN)
9. Clear local counter (variable Local_Counter) to see if Timer triggers interrupts

INT Handler for Timer

Set address for interrupt function (handler) for the interrupt (32-bits address)

```
AT91C_BASE_AIC->AIC_SVR[AT91C_ID_SYS] = (unsigned int) TIMER_INT_handler;
```

Timer interrupt handler

```
void TIMER_INT_handler (void) {
        if flag PITIE for Timer interrupt is set (PIT_MR register)        /* interrupt enabled */
                if flag PITS in PIT_SR register is set                    /* timer requested int */
                        read the PITC_PIVR register to clear PITS flag in PIT_SR
                        /* delay ~100 ms */
                        TimerCounter++;                                   /* LedToggle... */
        else another device requested interrupts
                check which device requested INT,
                process INT, clear INT flag,
                if unknown device, just increase counter of unknown interrupts
}
```

Interrupts from DBGU transceiver

DGBU generates system interrupt (ID number 1) – interrupt from processor peripheral devices (System Controller, mask AT91C_ID_SYS). We have distinguish which device triggered interrupt. A few interrupts can be triggered.

DGBU can generate the following interrupts:

- RXRDY: Enable RXRDY Interrupt
- TXRDY: Enable TXRDY Interrupt
- ENDRX: Enable End of Receive Transfer Interrupt
- ENDTX: Enable End of Transmit Interrupt
- OVRE: Enable Overrun Error Interrupt
- FRAME: Enable Framing Error Interrupt
- PARE: Enable Parity Error Interrupt
- TXEMPTY: Enable TXEMPTY Interrupt
- TXBUFE: Enable Buffer Empty Interrupt
- RXBUFF: Enable Buffer Full Interrupt
- COMMTX: Enable COMMTX (from ARM) Interrupt
- COMMRX: Enable COMMRX (from ARM) Interrupt

Interrupts from DBGU transceiver

DGBU interrupt handler

```
void DGBU_INT_handler (void) {
        int IntStatus;
        SysIRQCounter++;                /* to have a feeling how many system INTs are triggered */
        IntStatus = DGBU->SR;
        if (IntStatus & DBGU->IMR )     /* interrupt from DGBU */
                if INT from TxD         /* transmitter interrupt */
                        WriteNewData ();    /* be careful INTcan be also generated in case of error */
                else if INT from RxD
                        ReadDataToBuffer();/* INT can be also generated when error occur */
                else
                        other device triggered INT;
}
```

Interrupt Handlers in C (1)

Functions used as handlers require usage of preprocessor directive __attribute__ ((interrupt("IRQ")))

```
void INTButton_handler()__attribute__ ((interrupt("IRQ")));
void INTPIT_handler()__attribute__ ((interrupt("IRQ")));
void Soft_Interrupt_handler()__attribute__ ((interrupt("SWI")));
void Abort_Exception_handler()__attribute__ ((interrupt("ABORT")));
void Undef_Exception_handler()__attribute__ ((interrupt("UNDEF")));
void __irq IRQ_Handler(void)
```

Functions used as a handler is similar to normal function in C language

```
void INTButton_handler() {
    // standard C function
}
```

During laboratory we do not use __attribute__ ((interrupt("IRQ"))), we use functions provided by ATMEL, defined in startup.S file.