Dariusz Makowski

Department of Microelectronics and Computer Science

tel. 631 2720

dmakow@dmcs.pl

http://neo.dmcs.pl/psw

Lecture Agenda

- Microprocessor systems, embedded systems
- ARM processors family
- Peripheral devices
- ARM processor as platform for embedded programs
- Methodology of Designing Embedded Systems

Operating System

Operating System (OS) is a dedicated software for microprocessor-based devices that manages hardware resources and provides environment for applications.

OS is like an interface between hardware, applications and user:

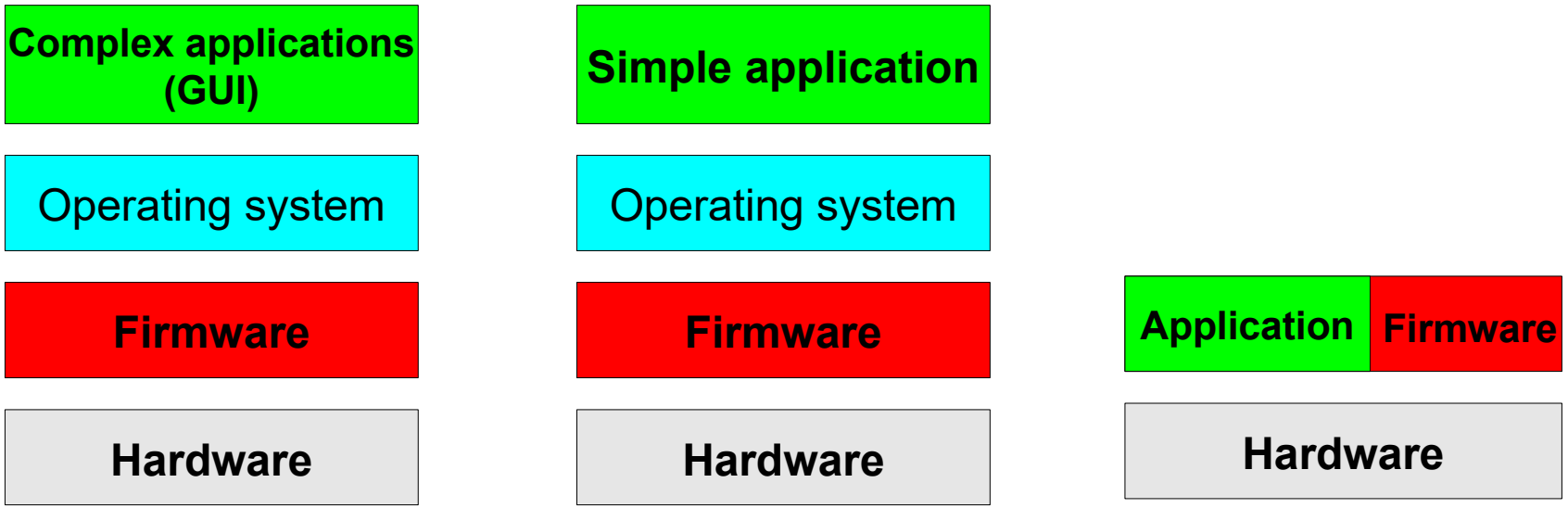
- Piece of software that sits between applications and hardware.
- Hides hardware details from applications.
- Provides standard interfaces to hardware and software devices.
- Provides protection mechanisms.
- Typical services:
  - Memory management (main memory, secondary memory, virtual memory, paging, file system),
  - Process management (scheduling, task management, synchronization, interrupt and exception handling, inter-task communication,
  - Input-Output management (device driver),
  - Support for distributed applications and multiprocessors.

Operating System (2)

We can distinguish three main layers of OS:

- **Kernel –** manages system resources, hardware/software,
  - Kernel provides functions for tasks management:
    - Scheduling,
    - Dispatching,
    - Intercommunication and synchronisation,
- **Shell**
  - provides user interface,
  - access to services of kernel,
- **File System** – mechanisms for storing, organization, manipulation and retrieval of data.

Computers and software



Personal Computer

Complex Embedded computer

Simple Embedded computer

Personal Computers, Universal Computers:

- High level languages (Assembler, C/C++, Pascal, Java, Basic...)

Embedded systems, controllers:

- Low level programming Assembler,
- High level languages (C/C++, Basic, Ada).

Design goal of Embedded OS

- **Small:** minimal memory footprint,
- **Open:** many interfaces and protocols, open system standards,
- **Modular:** easy to integrate custom components,
- **Portable:** run on different devices,
- **Real-time:** support of hard deadlines, bounded interrupts, scheduling, synchronization,
- **Power consumption:** integrated power management, mobile devices,
- **Robustness:** fault tolerant, halts, guards, exceptions, CRC, …
- **Configurable:** adaptable to required functionality.

Do we really need Operating System for Embedded Devices ?

- Monolithic kernel is too feature reach, requires significant processing power, memory, etc...
- Simple Embedded Systems can operate without OS – simpler design, cost-effective solution.
- Embedded Systems without OS allows to meet requirements of Real-Time Operating System (RTOS).
- Analysis and debugging is much simpler than usage of operating system.
- Significant amount of Embedded Systems requires only multitasking functionality.

Multitasking

- Multitasking is a method by which multiple processes (tasks) share common processing resources (e.g. CPU).
- Single CPU core can only execute single task.
- CPU can be assigned to different tasks that are performed in different time.
- Multitasking allows to select one task that will obtain CPU while other tasks are waiting.
- The act of reassigning a CPU from one task to another one is called a context switch.
- When context switches occur frequently enough the illusion of parallelism is achieved.

Pseudo-kernel

Simple embedded system can work without operating system – we need only executive with multitasking functionality.

- Multitasking can be achieved with OS and even without operating system.
- We can use simple infinite loop (pseudo-kernel).
- Program can be divided into non-blocking functions tasks.
- Tasks can be executed in turn.

Pseudo-kernels are usually implemented in Embedded Systems build with simple microcontrollers with small resources (RAM and ROM).

Pseudo-kernels can be divided into two groups:

- Systems based on infinite loop.
- Systems triggered with interrupts

Pooled Loop (1)

- Used for fast response to single devices,
- Single and repetitive instruction is used to test events (flags),
- If the event has not occurred the next command (task) is executed,
- Works well on single processor systems,
- Overlapping of events is not allowed or minimized,
- Usually implemented in systems as a background task, when interrupts are available.

Pooled Loop (2) – example

```
...
_branch_main:
        ldr     r0, =main
        mov     lr, pc
…
void main (void) {
                for (;;)                        /* execute program in infinite loop */
                {
                /*      Task 1          */
                        if (data_available){    /* check if data received */
                                process_data();         /* make some calculations */
                                data_available = 0;     /* clear flag and give processor time to other task */
                        }
                /*      Task 2          */
                        if (event_2){
                                ...do some work here...
                        }
                /*      Task 3          */
                        if (event_3){
                                ...do some work here...
                        }
                }
}
```

Exercise 1

Write a simple program for terminal device using pooled loop. The terminal device is equipped with LED matrix display, matrix keyboard, communication interface (e.g. EIA RS 232) and sound generator.

The system should meets the following criteria:

- Device should read data from keyboard,
- Display information on the LED display,
- Send data via serial interface,
- Device should read command from serial interface (e.g. clear display, display text, etc...),
- Generate warning (sound) when error will be detected (error in transmission or reading keyboard),
- Interrupts not available.

Cyclic Executives (1)

Cyclic Executives:

- Run on non-interrupt driven systems,
- Provides illusion of simultaneity,
- Requires simple and short processes,
- Tasks are executed according to Round-Robin algorithm,
- In the main loop one can use pointers to functions as a pointers to processes (stored in the task table),
- Intertask synchronisation and communication can be achieved through global variables.

Cyclic Executives (2)

```
void main (void) {
        for (;;)                                    /* execute program in infinite loop */
        {
                Task_number_1();
                Task_Number_2();
                Task_Number_3();    /* different rate structures can be achieved by repeating the
                                       task in the list, e.g. task number there require 3 times more
                                       processing power */
                Task_Number_3();
                Task_Number_2();
                Task_Number_3();
                Task_Number_2();    /* task number two must be executed more often than other,
                                       e.g. keyboard reading */
                Task_Number_4();
                ...
        }
}
```

Tasks cannot be executed in blocking mode. Single task can easily lock the whole “operating system”.

Cyclic Executives (3)

- Pointer to function can be used to call available tasks.
- We can easily add new task – we only need to modify table with function pointers.
- We can easily change order and frequency of called functions/tasks by manipulating function pointers.
- Tasks can be dispatched using interrupts.
- Interrupt handler can save and restore context for available tasks.

Exercise 2:

- Write a simple multitasking system able to execute max. 10 tasks.
- Use table for storing function pointers.

Cyclic Executives (3) – function pointer

```
#define AVAILABLE_TASKS    10
void Task1(void){
  …
  return;
}
  ...
void Task4(void){
  …
  return;
}
void main (void) {
    void (*fp[AVAILABLE_TASKS])(void) = {Task1, Task3, Task2, Task3, Task3, Task4, NULL};
    int TaskNumber=0;
            for (;;)                                    /* execute program in infinite loop */
            {
                    (*fp[TaskNumber++])();

                    if (fp[TaskNumber] == NULL) TaskNumber=0;

                    ...
            }
}
```

Pros and Cons of Polled Loop

Pros:

- Simple implementation, easy debugging,
- Requires much less resources comparing to OS,
- Well defined behaviour of the program (assuming that INT are not used),
- No context switching required,
- The order of executing task cannot be easily changed.

Cons:

- Main loop operates in locking mode, waste processor time for checking events
- No support for asynchronous events,
- Difficulty in adding new tasks,
- Code must “tuned” by programmer,
- Suitable for single processor systems with small amount of independent processes,
- Inadequate for complex systems, especially with Ethernet support.

Cooperative Multitasking

- Multiple tasks share processing resources (CPU).
- Tasks cooperates in such a way that voluntarily cede time to each other, e.g. when one task is waiting for resources (data from RS232 port) processing power is ceded to another task.
- Tasks can be executed in main infinite loop.
- Task dispatch can performed with break command or yield() function.

Example 1:

Three processes (process_A...process_C) are executed according to Round Robin algorithms. No context switching is available, processes use common stack. Program starts with tasks initialisation. Task switching realised in the main loop using break command.

Example 2:

Four tasks (task0 - task3) are executed according to R-R algorithm. Task switching is realised with yield function. Each task has its own stack in common memory. After initialisation program run outside main function (Program Counter is modified according to restored context).

Cooperative Multitasking - Example 1

```
void main (void) {
        process_inita();
        process_initb();
        process_initb();

 for (;;) /* execute program in the infinite loop */
  {
    process_a(); /* switch to next function          */
    process_b(); /* using  break                     */
    process_c(); /*                                  */
  }
}
```

```
void process_a (void){
        for (;;){
                switch (state_a){
                  case 1: phase_a1();
                          break;
                  case 2: phase_a2();
                          break;
                }
        }
}
```

```
void process_b (void){
        for (;;){
                switch (state_b){
                        case 1: phase_b1();
                                break;
                        case 2: phase_b2();
                                break;
                }
        }
}
```

Context switching

- **Context Switch is the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource.**
- Context is usually stored on stack,
- Context contains copy of:
  - main registers (r0-r12),
  - Program Counter (r15),
  - Stack pointer (r14),
  - Coprocessor registers,
  - Task description, e.g. task priority, task executing time, task mode etc...
- Interrupts should be disabled during context switch.

Cooperative Multitasking with Context Switching

```
void main (void) {
/*initialise system, configure handlers of functions, init stack, etc...*/
        init_task1();
        init_task2();
        TaskNumber = NUMBEROFTASKS;
        yield ();
        while(1);                   /* IDLE task */
}
void Task_1 (void){
        task1();                    /* execute code of task 1 */
        yield();                    /* resource busy, switch to the next task */
}
void Task_2 (void){

        task2();                    /* execute code of task 2 */
        yield();                    /* resource busy, switch to the next task */
}
/* still as a function */
void yield (void){
        save(context[TaskNumber]);              /* store context of the current task */
        TaskNumber = (++TaskNumber) % NUMBEROFTASKS;
        restore(context[TaskNumber]);           /* restore context of the new task,
                                                   recover program counter */
}
```

Cooperative Multitasking - Summary

Pros:

- Cost-effective alternative for ED with Operating System,
- Processor does not waste time polling event, can execute next task,
- Number of processes is limited by CPU processing power
- Cooperating processes are responsible for switching tasks,
- Context is changed by calling yield() function.

Cons:

- Tasks cannot be preempted,
- Programmer needs to remember to “switch” task,
- Incorrect function can lock the whole system (task cannot be switched)
- Difficulty in estimating reaction time of the system for asynchronous events.

Multitasking with Interrupts

- Context switching can be realized using interrupts from hardware timer (e.g. PIT).
- Timer provides time for task executing - quantum or timeslice. Quantum should be slightly longer that typical reaction of the system, to have enough time to finish most of the tasks and switch context (e.g. 10-100 ms).
- Task are scheduled using interrupt controller. If only single interrupt is available, interrupt handler is responsible for task scheduling and switching.
- Tasks synchronisation can be realized with interrupt masking or application of global semaphores.

Cooperative Multitasking with Interrupts

```
void main (void) {
/* initialise system, configure handlers of functions, init stack, etc...*/
        init();
        while(1);               /* IDLE task */
}

/* Timer generates interrupts  */
void int1 (void){
        save(context);          /* store context of the previous task */
        restore(++context;)     /* restore context of the current task */
        task1();                /* execute task 1 */
}

void int2 (void){
        save(context);          /* store context of the previous task */
        restore(++context;)         /* restore context of the current task */
        task2();                /* execute task 2 */
}

void int3 (void){
        save(context);          /* store context of the previous task */
        restore(++context;)         /* restore context of the current task */
        task3();                /* execute task  3 */
}
```

Cooperative Multitasking with Timer Interrupt

```
void main (void) {
/*initialise system, configure handlers of functions, init stack, etc...*/
        init_task1();
        init_task2();
        init_IRQ();                          /* configure timer interrupt */
        init_Timer();                        /* configure and start timer, timer period = timeslice */
        TaskNumber = NUMBEROFTASKS;          /* wait for first timer interrupt */
        while(1);                            /* IDLE task */
}
void Task_1 (void){
        task1();                /* execute code of task 1 */
        yield();                /* resource busy, switch to the next task */
}
void Task_2 (void){

        task2();                /* execute code of task 2 */
        yield();                /* resource busy, switch to the next task */
}
void yield (void){
                                /* Trigger software interrupt for timer */
}
Void TimerHandler (void){
        save(context[TaskNumber]);                      /* store context of the current task */
        TaskNumber = (++TaskNumber) % NUMBEROFTASKS;
        restore(context[TaskNumber]);                   /* restore context of the new task, recover PC*/
}
```

Cooperative Multitasking with Interrupts - Summary

Pros:

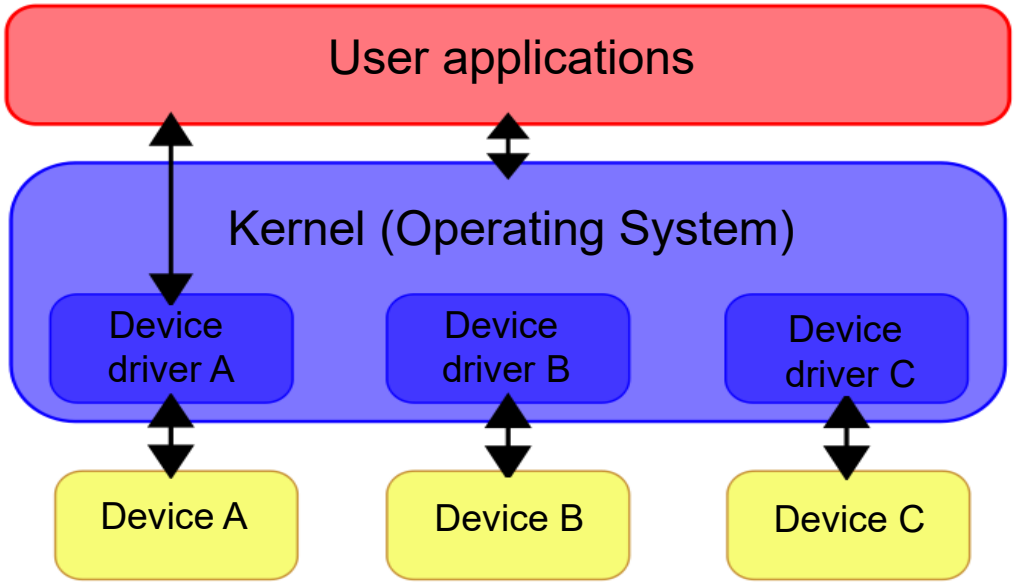
- Priority of tasks are assigned to interrupts priorities,
- Timeslicing – processes are allowed to run in preemptive mode. Time slice or quantum can be defined for cooperating tasks.
- New tasks can be easily added.

Cons:

- Priorities depends on interrupt controller or interrupt handler.

Drivers of Peripheral Devices

Drivers of Peripheral Devices (1)



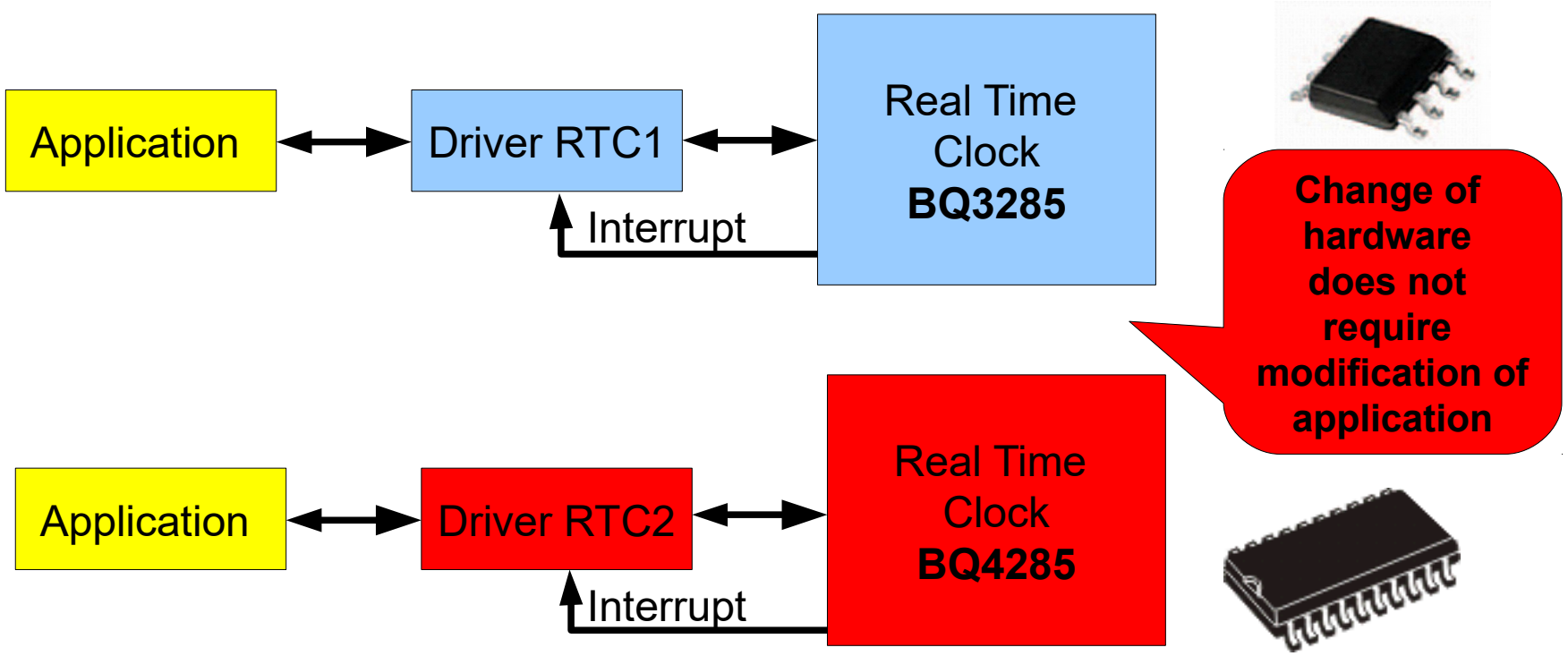
Device Driver – program responsible for communication between hardware device and software programs or applications. Drivers are provided as collection of functions. Drivers are hardware-dependent and operating-system-specific. Driver imitate some of hardware device features. Driver functions names and returned values depends in the operating system (OS Application Programming Interface). Direct access to hardware is restricted in case of OSs – device drivers should be always used.

In case of Embedded Systems applications can communicate directly with hardware, sometimes it is difficult to distinguish between drivers and applications. However, it is recommended to separate application layer functions from hardware using simple device drivers.

Drivers of Peripheral Devices (2)

Peripheral Device Driver (internal or external device) – provides basic functions that allow for easy usage of the device.

Device driver allows to hide hardware device – it provides collection of functions for initialisation and communication with device. Driver require also interrupt handler and functions for configuration of interrupts.



Drivers – Device Initialisation

Peripheral device driver – functions usually implemented:

Device_Open () - function used for device initialisation

Function can take parameters, e.g. driver controls more that one device, two USART transceivers. Parameter can be used to calculate offset for device registers.

Function can return operation code or descriptor (pointer to structure describing device status, device buffers, registers, etc...) to peripheral device.

A few different devices can open driver (dedicated for multioperations). Driver is equipped with deviceCounter that tells how many devices are connected to driver. In such a case device initialisation is performed only once.

Device_Close () - function called when application stop using device. Close function should deactivate device in safe way, e.g. I/O - configure ports as inputs. USART – turn of f transmitter/receiver, deactivate interrupts.

If function Open was called a few times, the device is used by a few applications, the Close function decrements only deviceCounter and release semaphore. Device is disabled and resources are freed when deviceCounter is equal to 0. Function Close can take parameters and returns code of operation.

Functions Open and Close should configure interrupts of the peripheral device..

Drivers – Communication with Devices

ReadData Device_Read () - function used for reading data from device, e.g. serial port. Read function can be locked when waiting for data or can return information that data are not available. Some functions use timeout timers – function lock for the defined period of time. If data are still not available within this time, function finishes and returns suitable code. Processor timer can be used for timeout measurement.

Read function can use interrupts or Direct Memory Access (DMA). I such case data are written to data buffer and suitable flag is set or DMA transfer is triggered when enough data is in buffer.

Device_Write () - function used for writing data to hardware device, e.g. sending data using serial port. Function can be also locked when transfer of data is not fished, e.g. transfer of 1 byte using RS 232 interface with 9600 baudrate requires 1 ms. We can use interrupts, timeout timer or DMA transfer to enhance data transfer.

Read and write functions can return results of operations, e.g. information that data was sent correctly. In such a case after transmission, receiver can be activated to receive confirmation code (depends on the used protocol). If data was not send successfully, transfer can be repeated

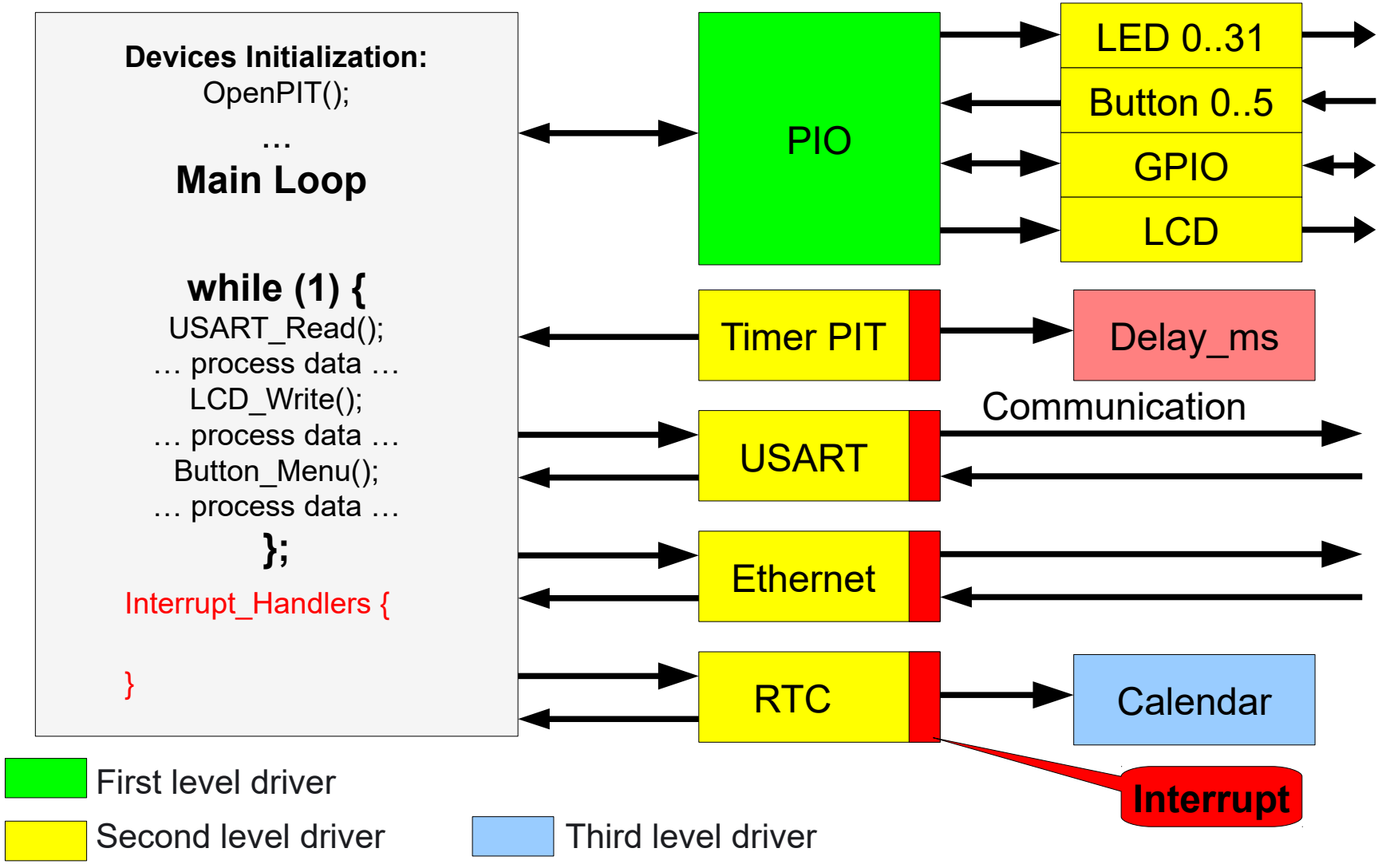
Drivers – Auxiliary Functions

DeviceStatus CheckStatus () - function used for checking device status, e.g. check Timer or UART flag, check if device is active, etc... Status functions can be used by others driver functions or applications, e.g. after writing data to transmitter, write function read status register (polling) to find out when transfer will be finished.

Device_INT_Handler() - function or handler for interrupt(s) of peripheral device, e.g. handler for PIT timer. Supplementary functions that allows to register, activate and deactivate interrupt(s) should be also provided.

Device_WriteString () - function based on the basic write function can be created to send string of characters (instead of single character). Function can use **Device_Write()** that sends or stores a single character. Function can inherit locking properties from lower level function.

Drivers for exemplary Embedded System – layers (1)



Drivers for exemplary Embedded System - layers (2)

- First level drivers:
  - PIO port driver (I/O),
- Second level drivers (use first level drivers):
  - LED driver,
  - Keyboard driver,
  - LCD display driver,
  - GPIO ports driver,
  - PIT Timer driver,
  - USART interface driver,
  - Ethernet interface driver,
  - RTC clock driver.
- Third level drivers (use second level drivers):
  - Calendar driver.

Driver for Parallel I/O Port

PIO_Struct* **PIO_Open** (unsigned int *RegistersPointer, unsigned int PortMask);

void **PIO_Close** (unsigned int *RegistersPointer, unsigned int PortMask);

unsigned int **PIO_Read** (PIO_Struct* PoiterToPIO);

void **PIO_Write** (PIO_Struct* PoiterToPIO, unsigned int Data);

unsigned int **PIO_Status** (PIO_Struct* PoiterToPIO);

Functions that return code of operation:

unsigned int **PIO_Read** (PIO_Struct* PoiterToPIO, unsigned int *ReadData);

unsigned int **PIO_Write** (PIO_Struct* PoiterToPIO, unsigred int *DataToSend);

unsigned int **PIO_Status** (PIO_Struct* PoiterToPIO, unsigned int *DeviceStatus);

Auxiliary functions:

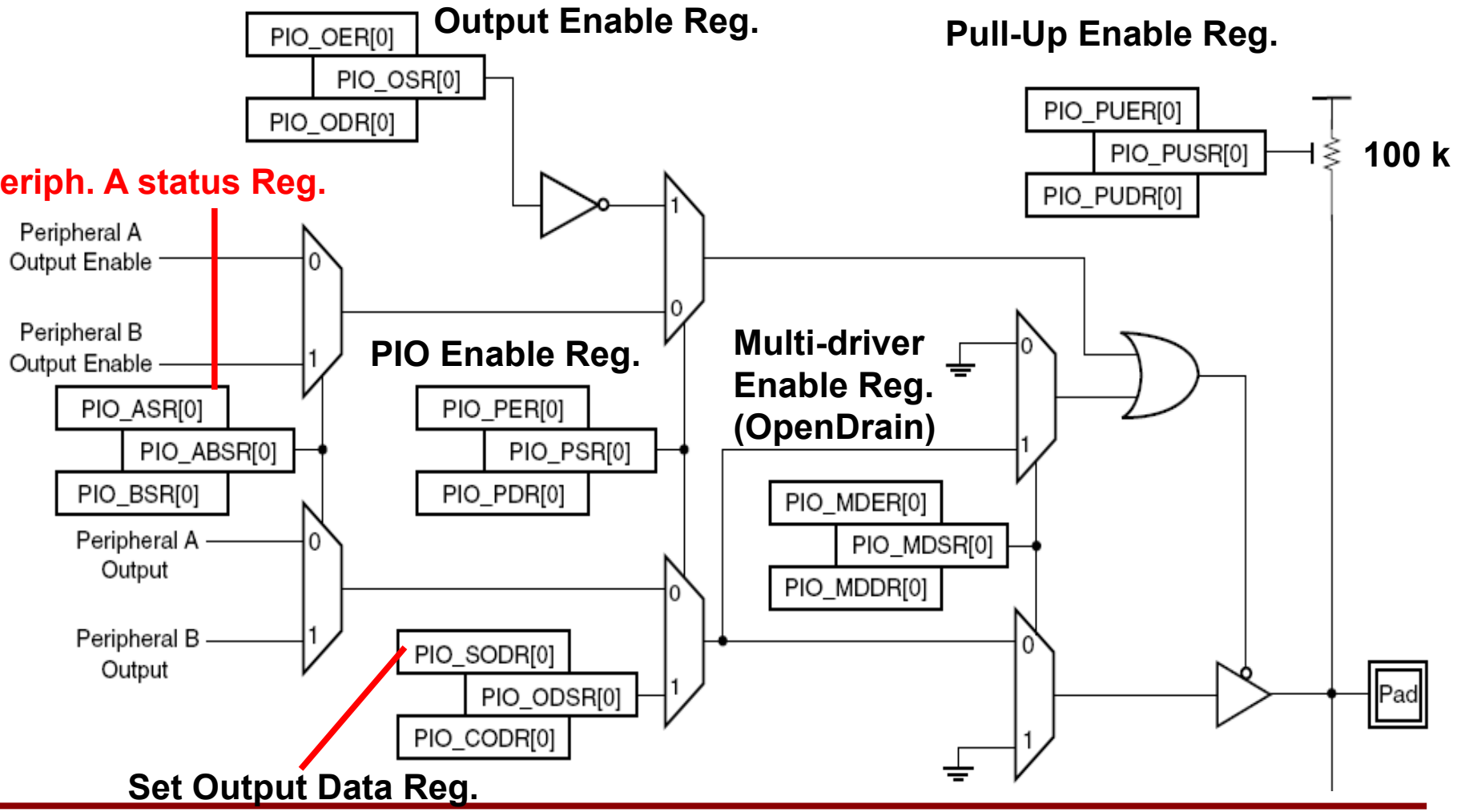
void **PIO_EnablePullUp** (unsigned int *RegistersPointer, unsigned int PortMask);

void **PIO_DisablePullUp** (unsigned int *RegistersPointer, unsigned int PortMask);

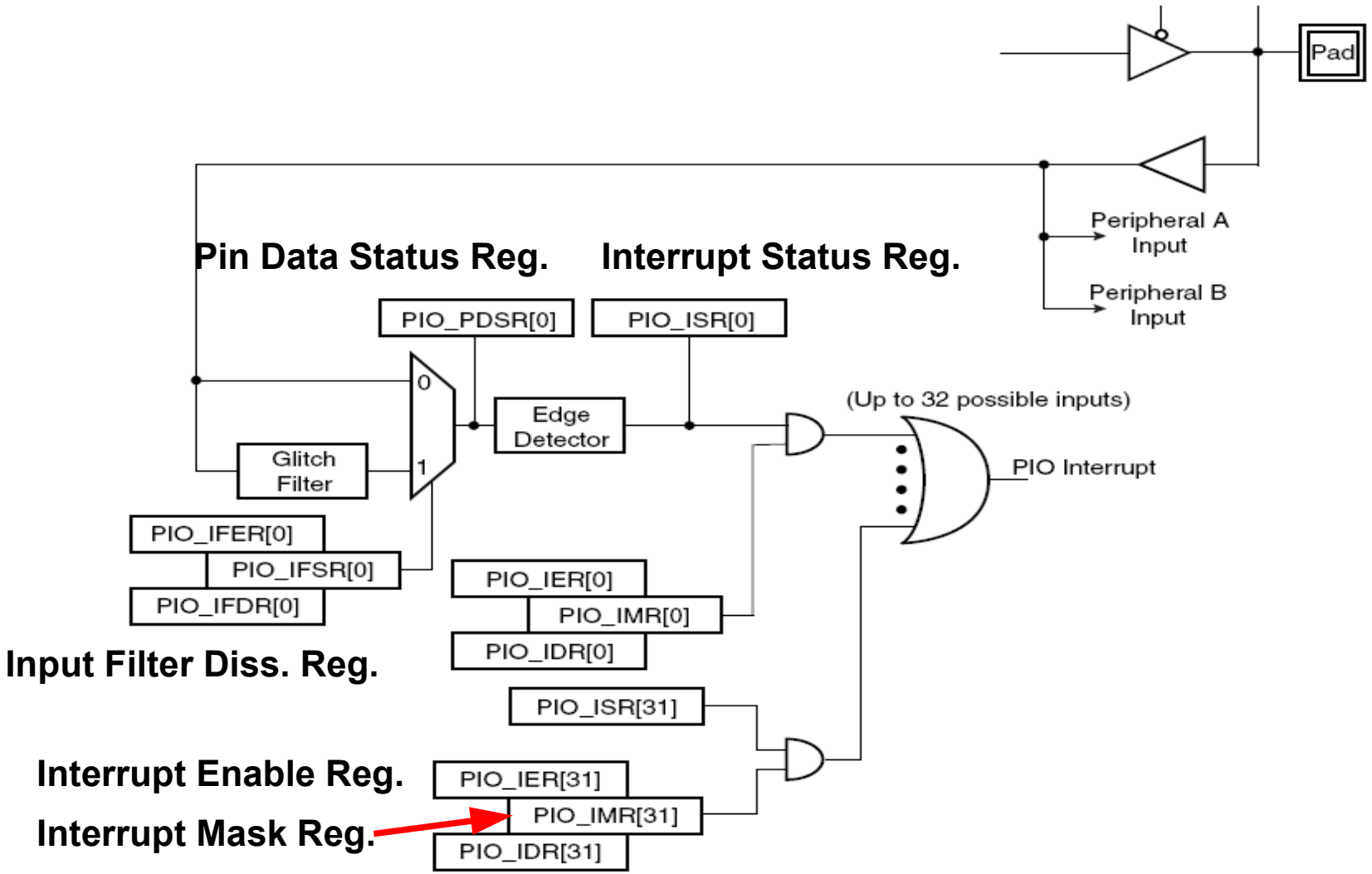
unsigned int **PIO_StatusPullUp** (unsigned int *RegistersPointer, unsigned int PortMask);

Driver for Parallel I/O Port – Block Diagram of Output Path

Figure 31-3. I/O Line Control Logic



Driver for Parallel I/O Port – Block Diagram of Input Path



How to write good driver ?

1. Prepare structure that images peripheral device registers. Define constants for masks and configuration parameters,

2. Reserve global flags (variables) that can be used for checking of device status, e.g. check if device was already initialised, how many devices use driver,

3. Develop functions for communication with driver and API, e.g. (Open, Close, Read, Write),

4. Develop handler for interrupt(s). The previous functions should definitely work before interrupts are activated. In other case debugging will be much more difficult or impossible.

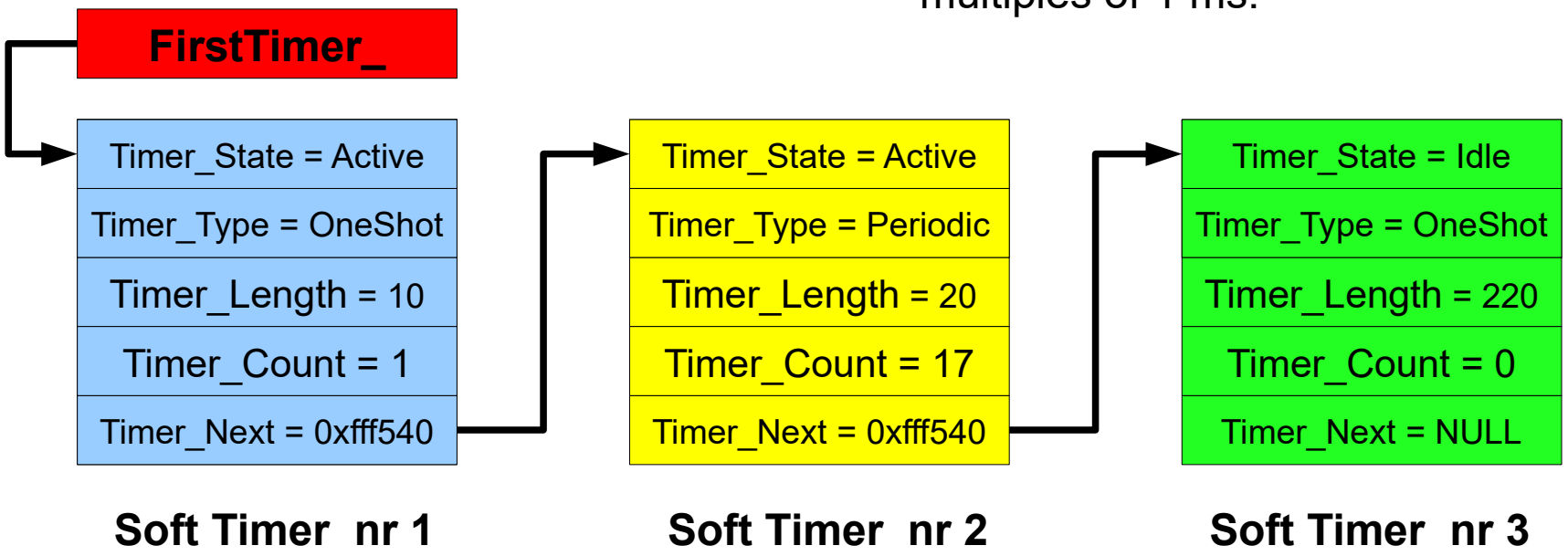
Multi-timer driver

```
struct {
        TimerState      Timer_State;    /* current timer state */
        TimerType       Timer_Type;     /* current timer mode */
        unsigned int    Timer_Length;   /* length of delay - number of hardware timer ticks */
        unsigned int    Timer_Count;    /* number of ticks to expiere for each software timer */
        Timer *         Timer_next;     /* pointer to the next software timer */
}       FirstTimer, *FirstTimer_;
```

Hardware Timer generates Interrupts every 1 ms.

Software timers can be used for interrupts that are multiples of 1 ms.

Sorted list of timers



Example of Driver for Timer

```
enum TimerState {Idle, Active, Done};
enum TimerType {OneShot, Periodic};
typedef struct {
        TimerState      Timer_State;        /* current timer state */
        TimerType       Timer_Type;         /* current timer mode */
        unsigned int    Timer_Length;       /* length of delay - number of hardware timer ticks */
        unsigned int    Timer_Count;        /* number of ticks to expiere for each software timer */
        Timer *         Timer_next;         /* pointer to the next software timer */
}       Timer, *Timer_;

int Timer_Open(Timer_ * TPoin)              /* configure hardware and soft timer */
int Timer_Close(Timer_ * TPoin)             /* release hardware or soft timer */
int Timer_Start(unsigned int miliseconds, TimerType Type, Timer_ * TPoin) /* start timer */
int Timer_Wait_For (Timer_ * TPoin)/* wait until timer fired */
void Timer_Cancel (Timer_ * TPoin)          /* turn off software timer */

static void Timer_INT (void);               /* hardware timer interrupt, e.g. 1 ms */
```

Functions of Software Timer (1)

```
int Timer_Start (unsigned int miliseconds, TimerType Type, Timer_ * TPoin){
  if (Tpoin->Timer_State != Idle)
        return -1;
  Tpoin->Timer_State = Active;
  Tpoin->Timer_Type = Type;
  Tpoin->Timer_Length = miliseconds / MSPERTICK;     /* delay in ms */
  AddTimerToList (Tpoin);                            `/* add pointer to the previous timer structure */
  return 0;
}

void Timer_Cancel (Timer_ * TPoin){
        Tpoin->Timer_State = Idle;
        RemoveTimerFromList (Tpoin);

}
```

Functions of Software Timer (2)

```
int Timer_Wait_For (Timer_ * TPoin){
        if (Tpoin->Timer_State != Active)
                return -1;
        while (Tpoin->Timer_State != Done);
        if (Tpoin->Timer_Type = Periodic){
                Tpoin->Timer_State = Active;
                Tpoin->Timer_Count=Tpoin->TimerLength;
        }
        else
        {
                Tpoin->Timer_State = Idle;
        }
        return 0;
}
```

Interrupt Handler for Hardware Timer

```
static void Timer_INT (void){          /* hardware timer interrupt, e.g. 1 ms */

    0. Hardware Timer (reinitialise each t = 1 ms, confirm interrupt, etc...)
    1. Check list of active Timers,
    2. Decrement Timer_Count,
    3. If Timer_Count equal to 0 and TimerType = OneShot remove timer from list,
    4. If Timer_Count equal to 0 and TimerType = Periodic reinitialise timer:
       Timer_Count = Timer_Length.
    5. Modify flag for given timer (Timer_Fired) or generate software interrupt from
    Timer.

}
```

Drivers in C++

```
enum TimerState { Idle, Active, Done };
enum TimerType { OneShot, Periodic };
class Timer {
        public:
                Timer ();
               ~Timer ();

                int Start (unsigned int miliseconds, TimerType = OneShot);
                int Wait_For ();
                void Cancel ();

                TimerState      State;
                TimerType       Type;
                unsigned int    Length;


                unsigned int    Count;
                Timer *         pNext;
        private:
                static void INT ();
};
```

Startup File

Structure of Startup File

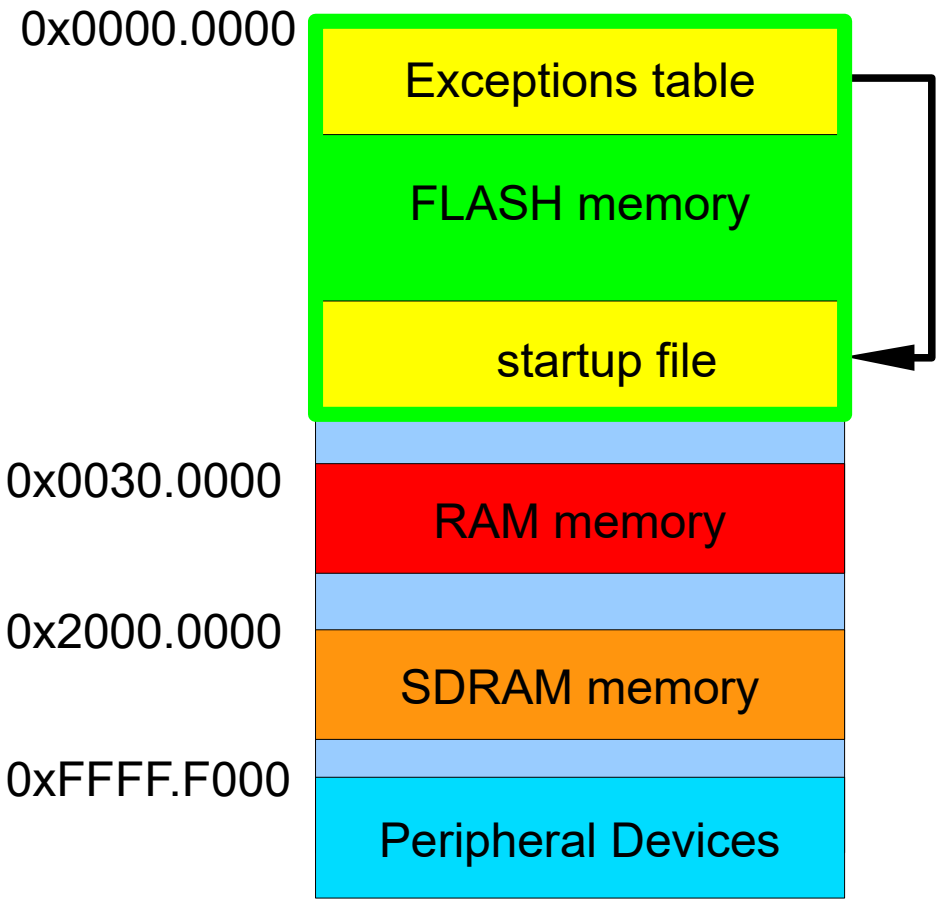
Startup file code is executed by processor just after reset. The code is responsible for configuration of basic peripheral devices and modules of processor.

Startup file is usually written in assembler because the code requires access to all processor resources that cannot be accessed from higher level languages. The code is executed before high level code (e.g. C/C++). Startup file is responsible for:

- Allocation of memory and configuration of stack pointers for different modes of operation (user, supervisor, IRQ, FIQ, etc...),
- Configuration of memory (remap FLASH memory, activate SRAM/DRAM, clean memory),
- Initialise exception vectors,
- Copy of Operating System or application code to memory,
- Initialise global variables in RAM (copy data from ROM, init variables with 0s),
- Configure peripheral devices,
- Initialise interrupt controller,
- Change processor mode if required,
- Call int main (void) function.

Structure of Startup File (2)

```
RESET vector (address 0)
.section .text
reset_handler:
ldr     pc, =_low_level_init
/*       Initialization...          */
_low_level_init:


_stack_init:
_init_data:
_init_bss:

_branch_main:
```



Structure of Startup File (3)

Program in the main loop cannot be finished. Processor cannot execute return or exit functions.

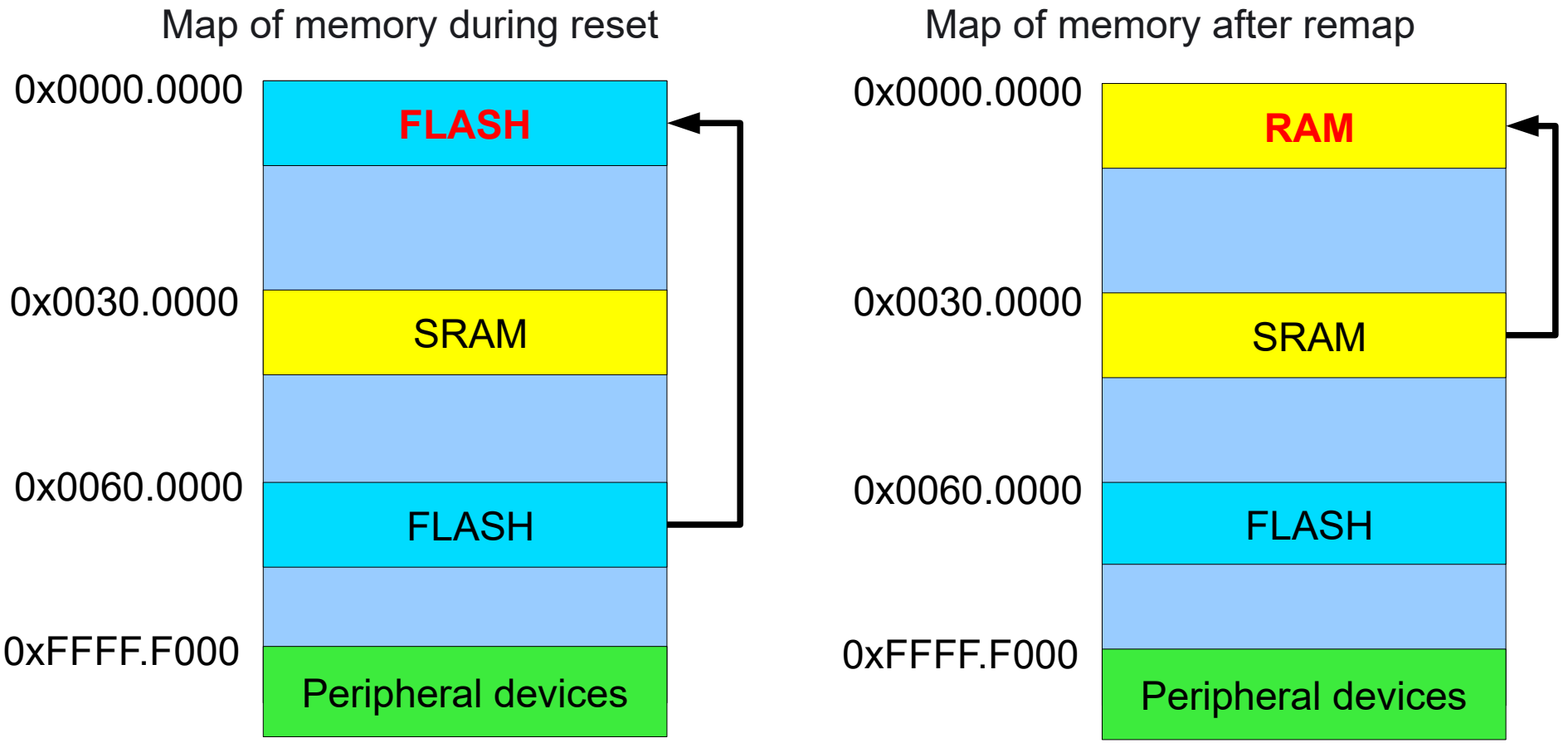
```
...
...
_branch_main:
        ldr      r0, =main
        mov      lr, pc
        bx       r0
...
...
void main (int) {
                 While (1)
                 {

                        main program

                 }
        return 0;
}
```

Structure of Startup File (4)

Configuration of essential devices, required for processor operation:

- Configuration of reference clock module (PLL). After reset processor operates with “slow clock” (internal RC generator),
- Configuration of memory controller (FLASH, RAM) – configure number of WaitStates, base address,
- Remap memory FLASH<->SRAM,
- Configure Watch-Dog timer (after reset Watch-Dog is active),
- Configure AIC module (assign default interrupt handlers),
- Initialise stack pointers for different modes of operation (user, IRQ, FIQ,...),
- Unblock NRST input (external reset).

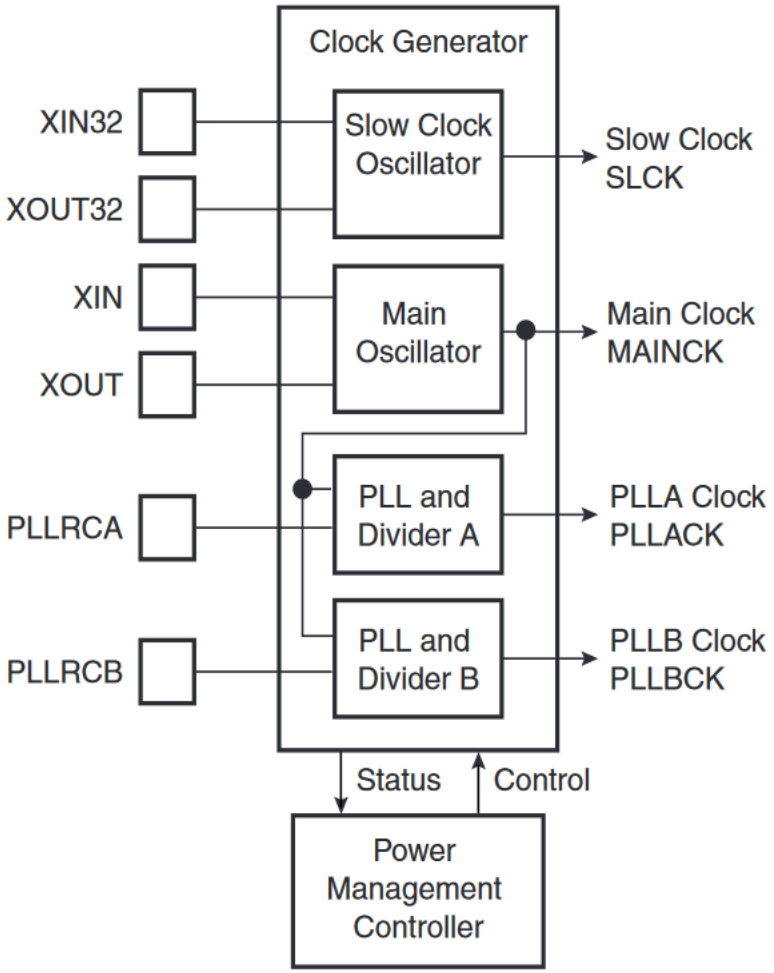
Remapping of Memory



Remapping of FLASH memory is performed after execution of startup code (REMAP register)

Clock Generator

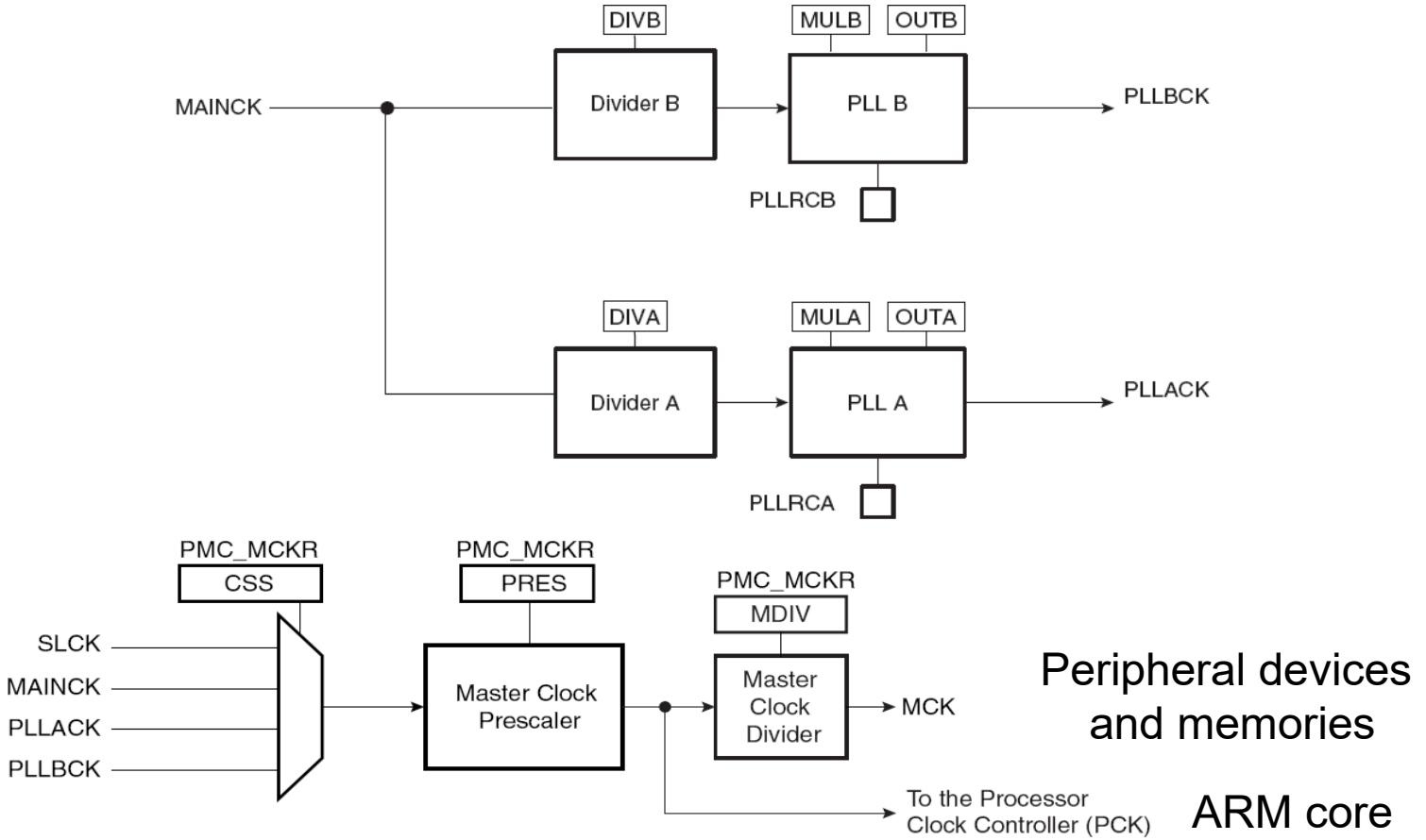
- Embeds the low-power 32768 Hz Slow Clock Oscillator
  - Provides the permanent Slow Clock SLCK to the system
- Embeds the Main Oscillator
  - Oscillator bypass feature
  - Supports 3 to 20 MHz crystals
- Embeds 2 PLLs
  - Output 80 to 240 MHz clocks
  - Integrates an input divider to increase output accuracy
  - 1 MHz Minimum input frequency



Configuration of Reference Clock – Chapter 28 (1)

After reset processor operates with slow clock with frequency f = 32768 Hz. Slow clock is always active (generated by build-in RC generator).

After reset Phase Locked Loops (PLLs) are disabled.



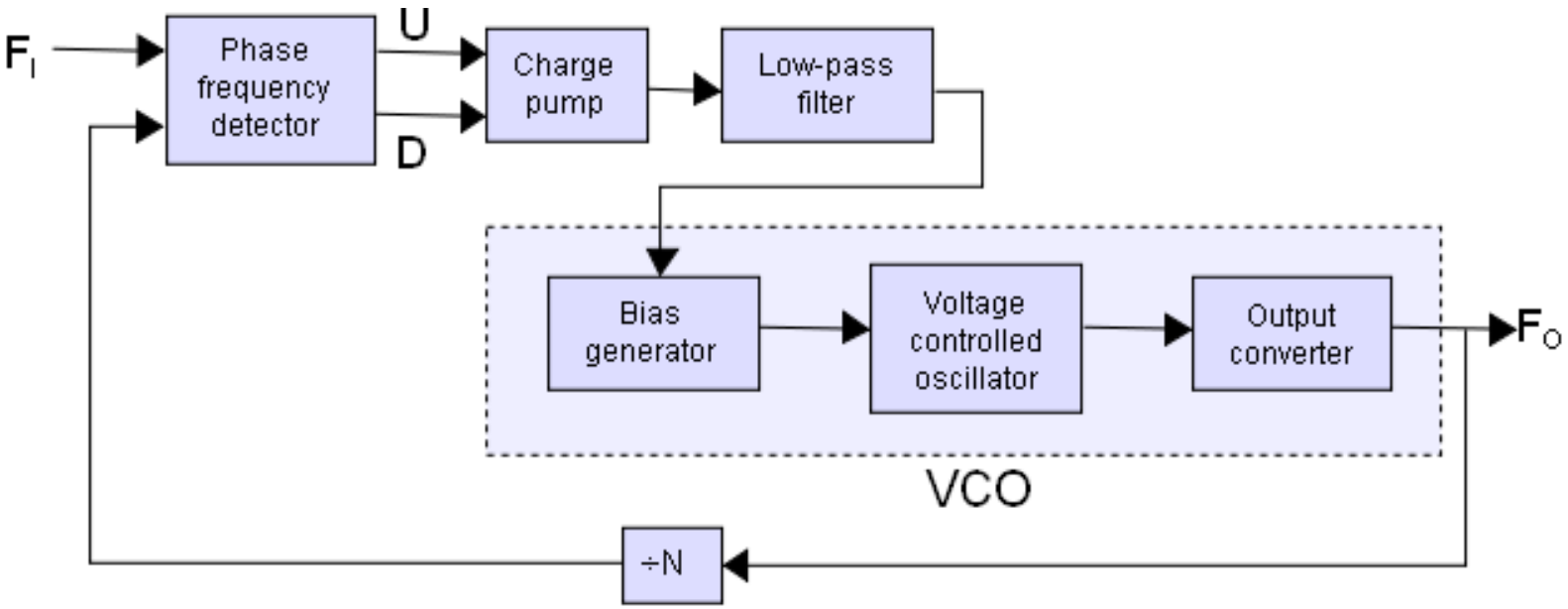
Generator with PLL (1)

Phase Locked Loop (PLL) – electronic circuit based on feedback used for automatic regulation of generator frequency. PLL is used in frequency synthesisers, heterodyne RF receivers, frequency sources and frequency multipliers.

PLL generator is composed of:

- Reference generator (quartz generator),
- Phase detector,
- Low pass filter,
- Voltage Controlled Oscillator (VCO),
- Feedback loop with suitable frequency divider.

Generator with PLL (2)



High frequency signal generated by VCO is connected to output (Fo). The same signal is connected to divider (/N). The output of divider is connected to phase detector (PD) and compared with stable reference signal. Phase difference of reference signal and output signal divided by N is used to control VCO. Feedback loop strive to obtain synchronous signals (phase error equal to 0). Low pass filter is required to make the loop stable.

Configuration of Reference Clock (2)

Procedure to turn on PLL generator:

1. Turn on quartz generator. Wait until frequency will be stable (bit PMC_MOSCS).
2. Configure PLLA, f = 16 367 660*110/9 = ~200 MHz. After turning on PLLA wait until PPLA will be stable (bit PMC_LOCKA) and frequency will be also stable (bit PMC_MCKRDY).
3. Switch processor to use PLLA (in example additional divider by 2), bit AT91C_PMC_CSS_PLLA_CLK. Wait until frequency will be stable.

Konfiguracja PLLA:

```
AT91C_BASE_PMC->
                    PMC_PLLAR = AT91C_CKGR_SRCA     |    /* programming PLL */
                    AT91C_CKGR_OUTA_2               |    /* electrical parameters  */
                    (0x3F << 8)                     |    /* counter = 63 */
                    (AT91C_CKGR_MULA & (0x6D << 16))|    /* multiplier 109 */
                    (AT91C_CKGR_DIVA & 9);               /* divider 9 */
```

f ref = 16 367 660 Hz

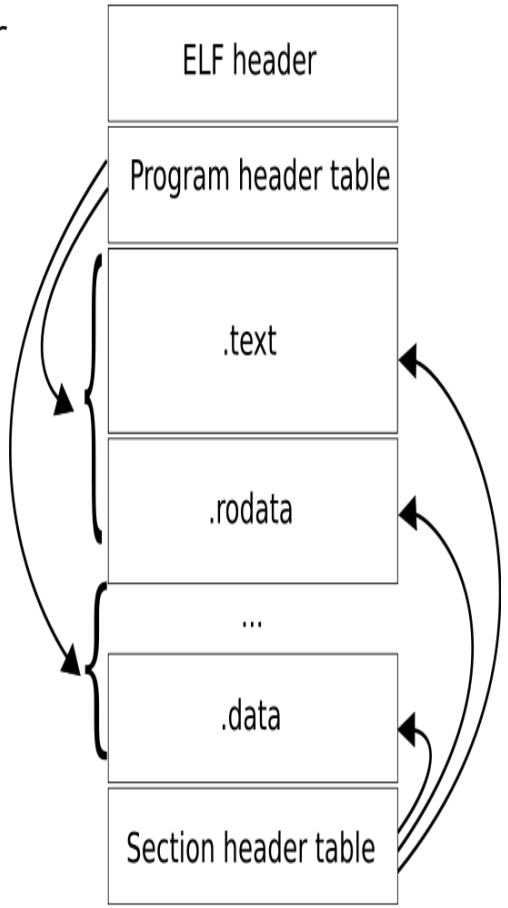
fout = fref * (MULA+1) / DIVA = 16 MHz * 110 / 9 => 200 MHz

f_{MCK} = fout / 2 => ~100 MHz

Analysis of lowlevel.c file

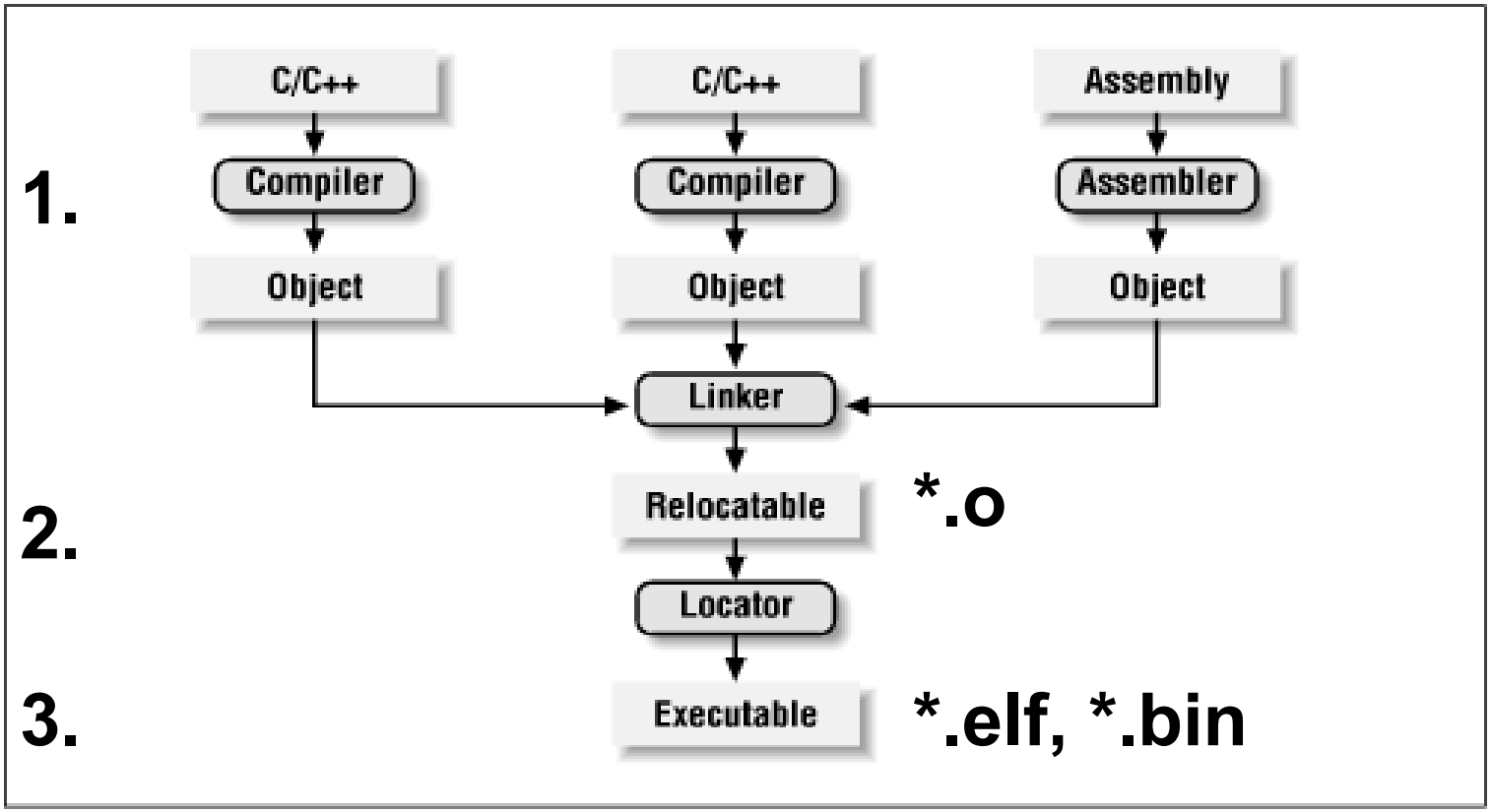
COFF vs ELF

- **COFF (Common Object File Format) –** specification of format for executable, object code, and shared library computer files used on Unix systems. COFF was introduced to substitute old **a.out format**. COFF is used also on other platforms, e.g. Windows. Currently, ELF standard is promoted instead of COFF.
- **ELF (Executable and Linkable Format) –** common standard file format for executables, object code, shared libraries, and core dumps used on different architectures, e.g.: x86 family, PowerPC, OpenVMS, BeOS, PlayStation Portable, PlayStation 2, PlayStation 3, Wii, Nintendo DS, GP2X, AmigaOS 4 and Symbian OS v9.
- Usufull tools:
  - readelf
  - elfdump
  - objdump



Źródłó: wikipedia

Compilation Process

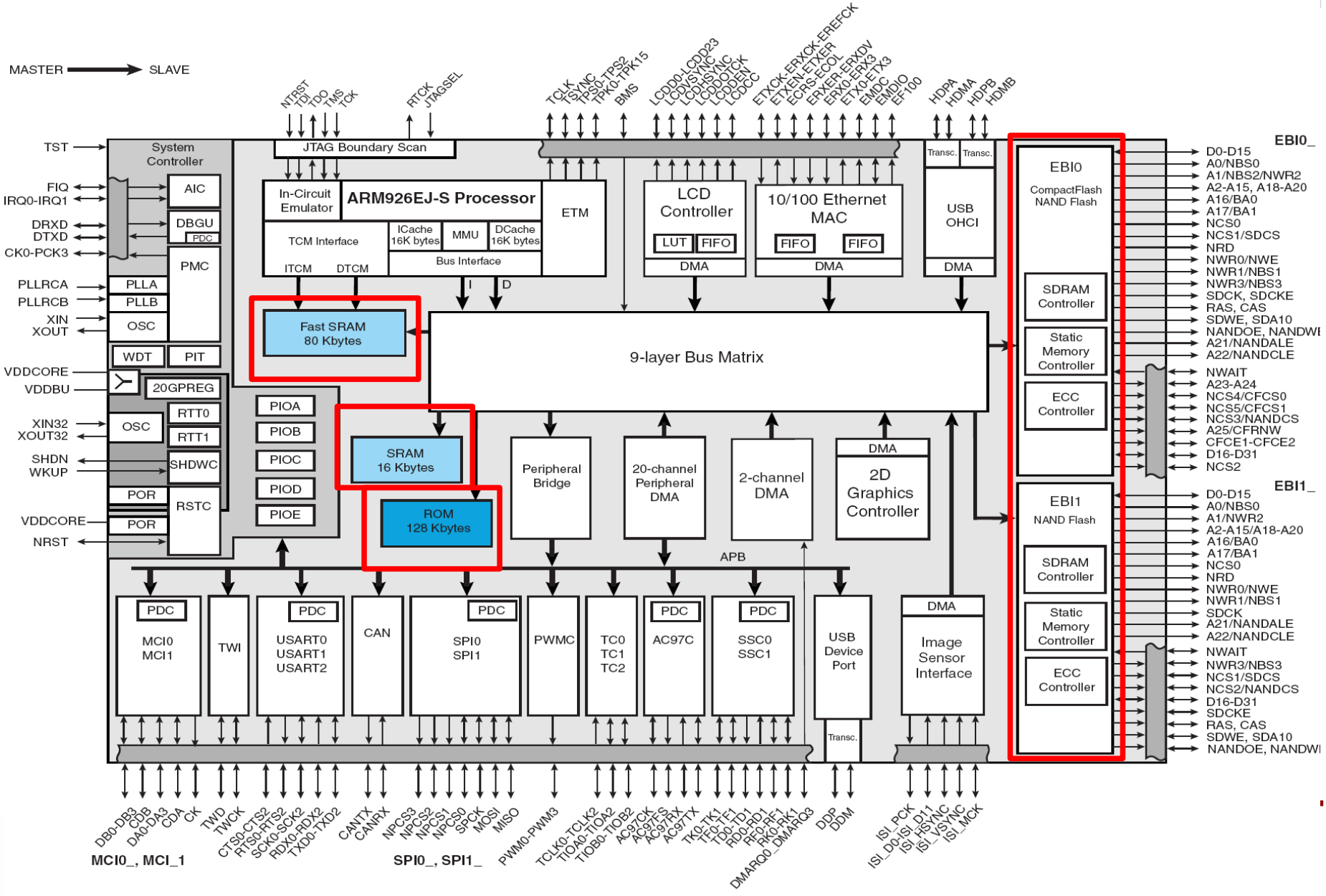


- 1 phase – compilation of source files → relocable binary files
- 2 phase – linking of relocable files → relocable binary file
- 3 phase – generation of executable file (with assigned addresses)

Linker Script

```
/* elf32-littlearm.lds for ARM At91SAM9263 */
OUTPUT_FORMAT ("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH (arm)
ENTRY (reset_handler)
/*#include "project.h"*/
SECTIONS
{
  . = 0x1.0000;                 /* base address */
        .text : { (.text) }     /* code section */
  . = 0x800.0000;               /* base address */
        .data : { (.data) }     /* initialized data */
        .bss : { *(.bss) }      /* uninitialized data */
}

LED_test: $(OBJS)
$(LD) $(LDFLAGS) -Ttext 0x20000000 -Tdata 0x300000 -n -o $(OUTFILE_SDRAM).elf $
(OBJS)
```

Memories


MASTER
SLAVE
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
TST
System Controller
JTAG Boundary Scan
FIQ
IRQ0-IRQ1
DRXD
DTXD
CK0-PCK3
AIC
DBGU
PDC
PMC
PLLRCA
PLLRCB
XIN
XOUT
PLLA
PLLB
OSC
WDT
PIT
VDDCORE
VDDBU
20GPREG
RTT0
RTT1
XIN32
XOUT32
OSC
SHDN
WKUP
SHDWC
POR
RSTC
VDDCORE
POR
NRST
In-Circuit Emulator
ARM926EJ-S Processor
ETM
TCM Interface
ICache 16K bytes
MMU
DCache 16K bytes
Bus Interface
ITCM
DTCM
I
D
LCD Controller
LUT
FIFO
DMA
10/100 Ethernet MAC
FIFO
FIFO
DMA
Transc.
Transc.
USB OHCI
DMA
Fast SRAM 80 Kbytes
9-layer Bus Matrix
SRAM 16 Kbytes
ROM 128 Kbytes
PIOA
PIOB
PIOC
PIOD
PIOE
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
EBI0
CompactFlash
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
EBI1
NAND Flash
SDRAM Controller
Static Memory Controller
ECC Controller
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
MCI0_, MCI_1
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

Embedded Memories

128 Kbyte ROM

- Single Cycle Access at full matrix speed

One 80 Kbyte Fast SRAM

- Single Cycle Access at full matrix speed
- Supports ARM926EJ-S TCM interface at full processor speed
- Allows internal Frame Buffer for up to 1/4 VGA 8 bpp screen

16 Kbyte Fast SRAM

- Single Cycle Access at full matrix speed

Internal Memory Mapping

Internal Memory Mapping

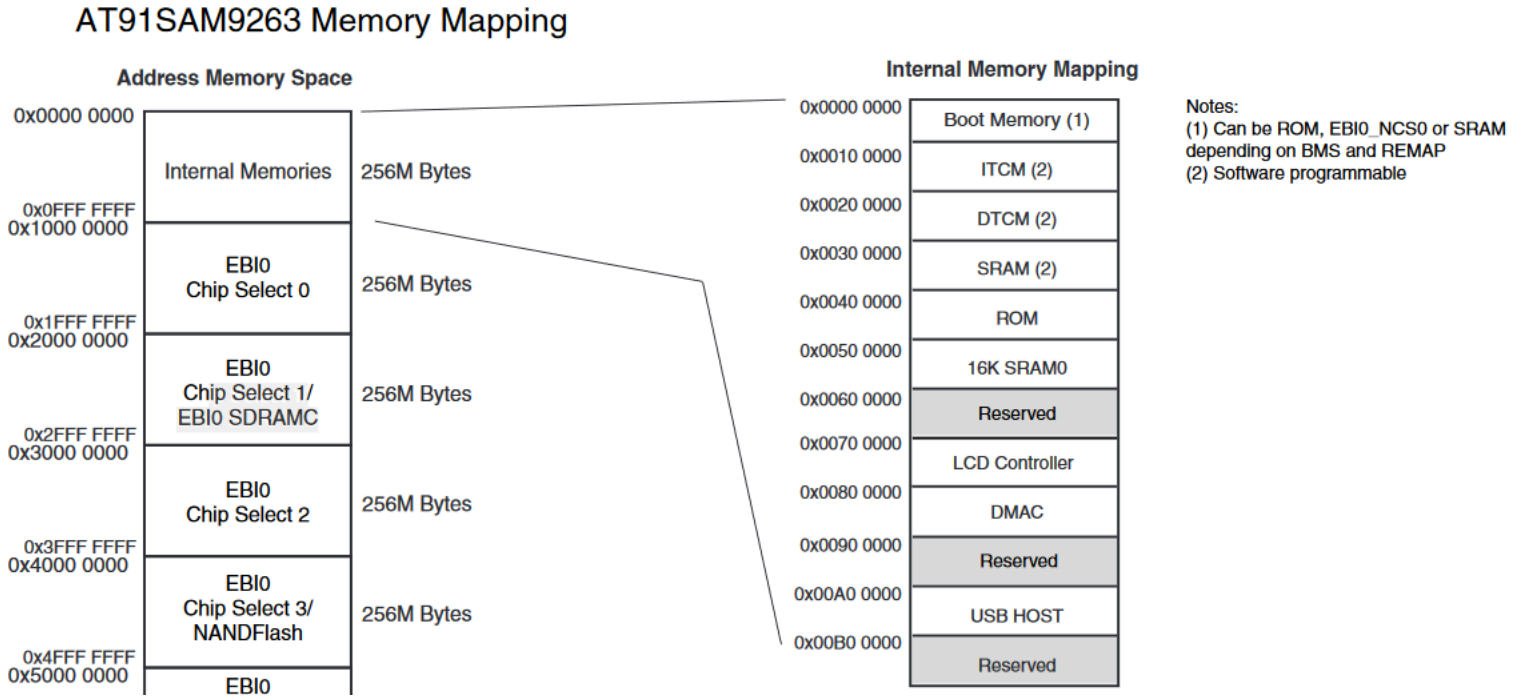
Address	REMAP = 0		REMAP = 1
	BMS = 1	BMS = 0	
0x0000 0000	ROM	EBI0_NCS0	SRAM C

Internal SRAM Block Size

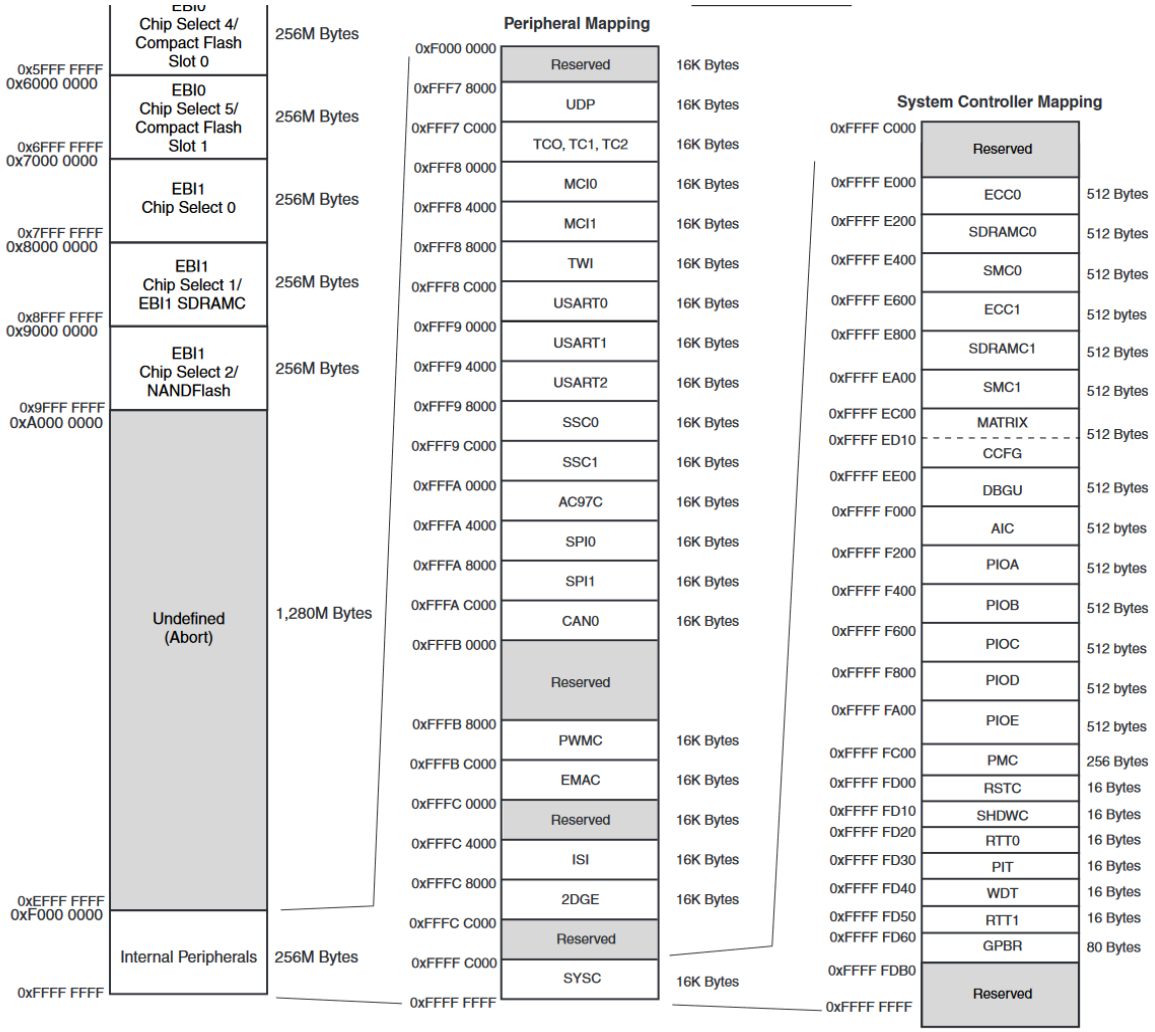
Internal SRAM C		Internal SRAM A (ITCM) Size		
		0	16 Kbytes	32 Kbytes
Internal SRAM B (DTCM) size	0	80 Kbytes	64 Kbytes	48 Kbytes
	16 Kbytes	64 Kbytes	48 Kbytes	32 Kbytes
	32 Kbytes	48 Kbytes	32 Kbytes	16 Kbytes

Memory mapping #1

- 0x0030.0000 – internal SRAM
- 0x2000.0000 – external SDRAM (Chip select 1)



Memory mapping #2



Linker script for ARM AT91SAM9263 processor

```
SECTIONS {
.text : {
  _stext = .;
        *(.text)       /* program code */
        *(.rodata)    /* read-only data (constants) */
        *(.rodata*)
        . = ALIGN(4);
  _etext = . ; }
/* all initialized .data  that go into FLASH  */
.data : AT ( ADDR (.text) + SIZEOF (.text) ) {
  _sdata = .;
        *(.vectors)  /* vectors table */
         (.data)      /* initialized data */
  _edata = .; }
/* all uninitialized .bss that go into FLASH  */
.bss (NOLOAD) : {
        . = ALIGN(4);
  _sbss = .;
        *(.bss)                    /* uninitialized data */
  _ebss = .; } }
end = .;
```

```
CROSS_COMPILE=arm-elf-
LD=$(CROSS_COMPILE)gcc
LDFLAGS+=-nostartfiles -Wl,--cref
LDFLAGS+=-lc -lgcc
LDFLAGS+=-T elf32-littlearm.lds
OBJS  = cstartup.o
OBJS+= lowlevel.o main.o

LED_test: $(OBJS)
$(LD) $(LDFLAGS) -Ttext 0x20000000
                 -Tdata 0x300000 -n -o
      $(OUTFILE_LED_test).elf $(OBJS)
```