# Industrial Embedded Systems
# - Design for Harsh Environment -

Dr. Alexander Walsch
alexander.walsch@ge.com

Part VI

WS 2012/13

Technical University Munich (TUM)

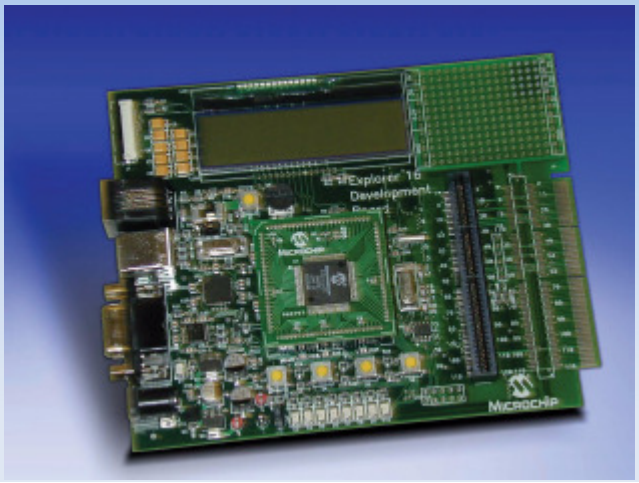
# SW Design Approach

- Identify standards (coding, best practice, etc.) which should be used (overlap with requirements analysis)
  - Company internal design practice: coding and style guidelines
  - Standards: architectural and design constraints imposed by certification, coding standards (e.g. MISRA C/C++)
- Specify major software building blocks and do a rough footprint calculation (e.g. OS, libraries, peripheral support)
- Specify interfaces between software building blocks
- Identify software design patterns if applicable

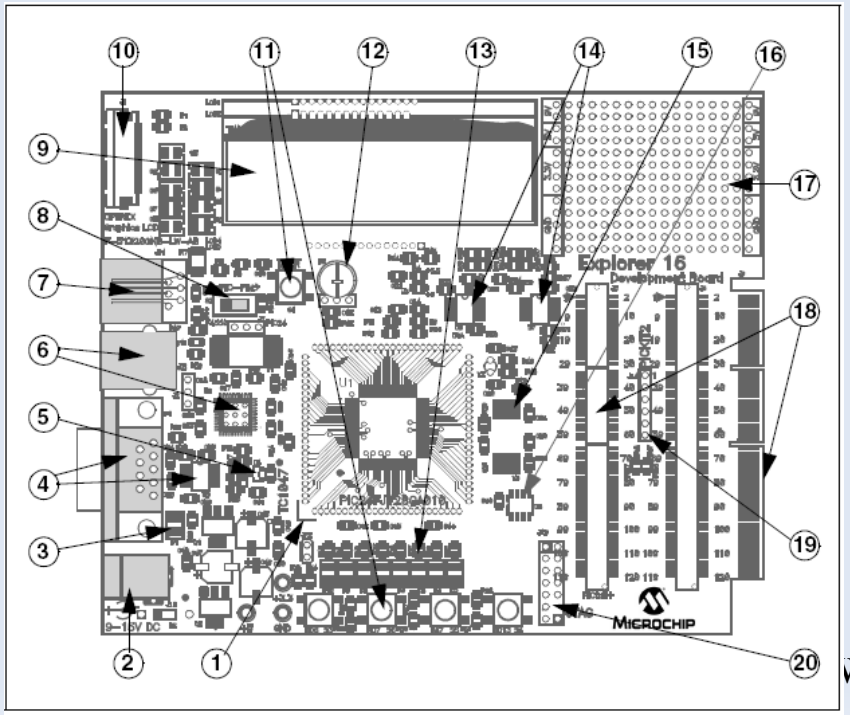
# First Steps

- Common practice:
  - Choose an embedded processor (often based on peripherals, past usage)
  - Get an evaluation board and the software support (IDE – editor, compiler, linker, debugger)
  - Implement the most demanding functionality (usually something which uses up a lot of hardware resources) – download the examples and change them
  - Outcome: have a hands-on feeling for feasability (compare lecture II)
- All IC vendors provide evaluation boards and evaluation versions of software (sometimes completely free of charge)
  - E.g. : Microchip MPLAB, TI Code Composer, Freescale Code Warrior
  - Usually come with a large collection of examples

# Evaluation Board Example
# - Microchip Explorer 16 -

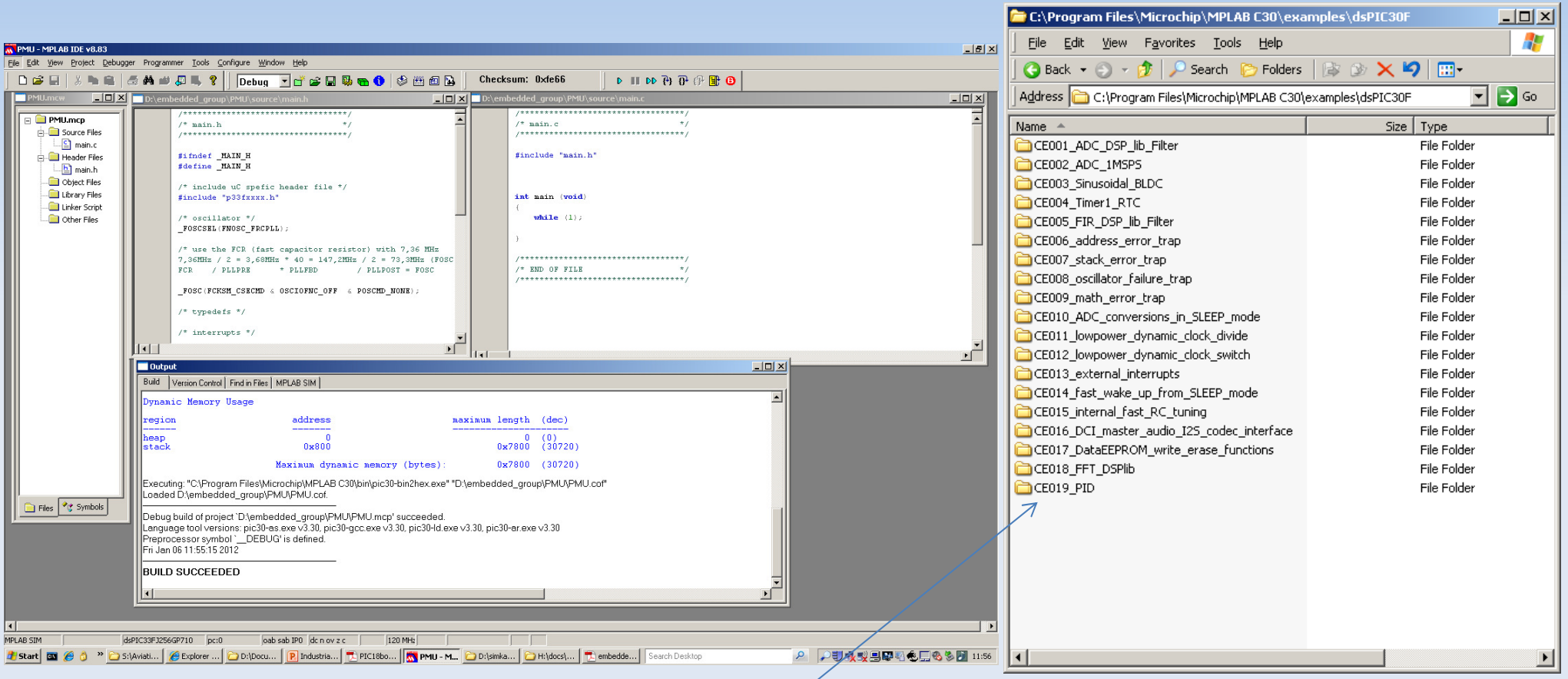


Source: microchip.com



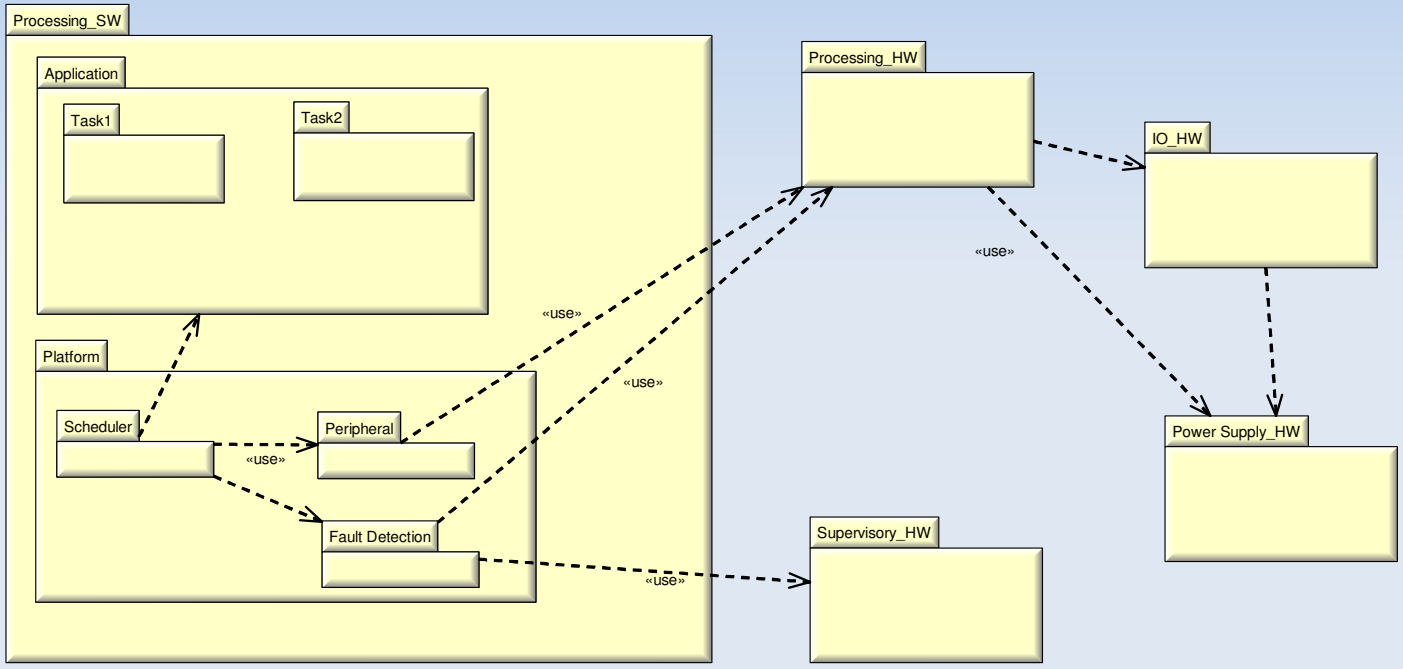

1. 100-pin PIM riser, compatible with the PIM versions of all Microchip PIC24F/24H/dsPIC33F devices
2. Direct 9 VDC power input that provides +3.3V and +5V (regulated) to the entire board
3. Power indicator LED
4. RS-232 serial port and associated hardware
5. On-board analog thermal sensor
6. USB connectivity for communications and device programming/debugging
7. Standard 6-wire In-Circuit Debugger (ICD) connector for connections to an MPLAB ICD 2 programmer/debugger module
8. Hardware selection of PIM or soldered on-board microcontroller (in future versions)
9. 2-line by 16-character LCD
10. Provisioning on PCB for add on graphic LCD
11. Push button switches for device Reset and user-defined inputs
12. Potentiometer for analog input
13. Eight indicator LEDs
14. 74HCT4053 multiplexers for selectable crossover configuration on serial communication lines
15. Serial EEPROM
16. Independent crystals for precision microcontroller clocking (8 MHz) and RTCC operation (32.768 kHz)
17. Prototype area for developing custom applications
18. Socket and edge connector for PICtail™ Plus card compatibility
19. Six-pin interface for PICkit 2 Programmer
20. JTAG connector pad for optional boundary scan functionality

# Evaluation Board Example Ctd.
# - Microchip Explorer 16 -




Example C code projects

# Software Architecture



# Scheduling

- The need for scheduling (as taken from PMU system requirements specification):

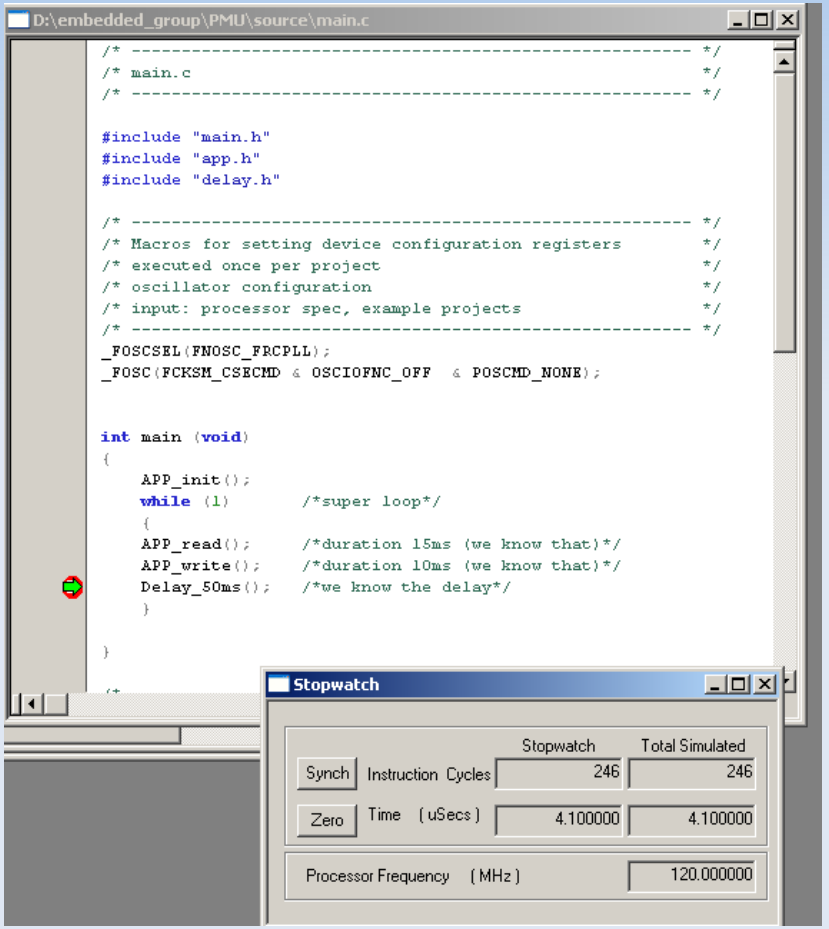
PMUSysRQ 8: Pressure readings communicated via CAN shall not be older than 100ms.

PMUSafetyRQ4: The process safety time shall not exceed 3s.

- Task response time:
  also known as execution time is the total time required for the computer to complete a task (IO, memory access, overhead, CPU execution time) – a task in general is an instance of a program that consumes time
- Task cycle time:
  time between periodic task calls (start of execution)

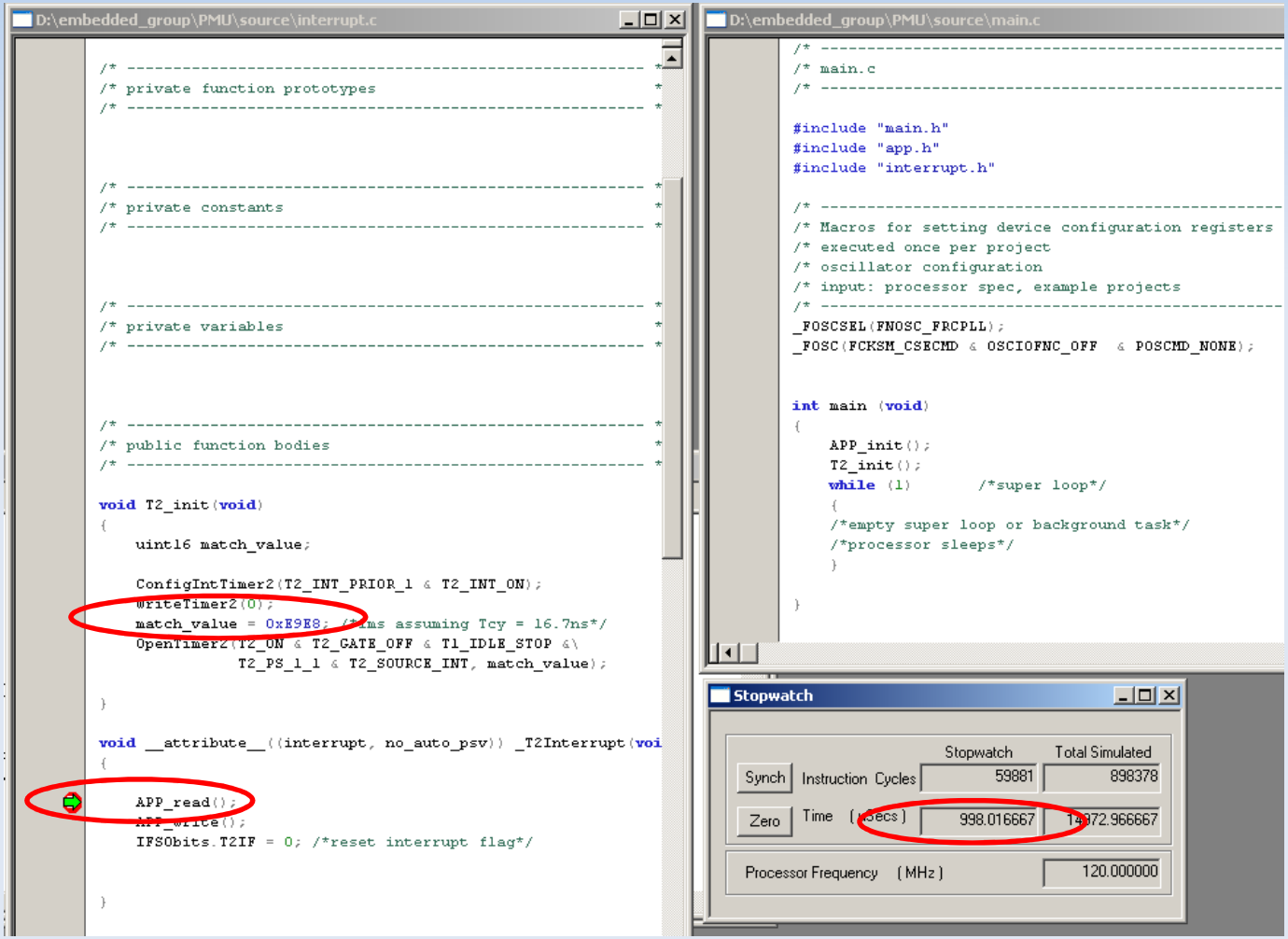
# Scheduling II

- The main loop:
  - Super loop
  - Functions (tasks) to be executed in sequence
  - Functions run-to-completion
  - Single stack
- But:
  - Relies on timeliness of executed functions
  - Variation of function response time will affect timing of all others



# Scheduling III

- Timer based interrupts:

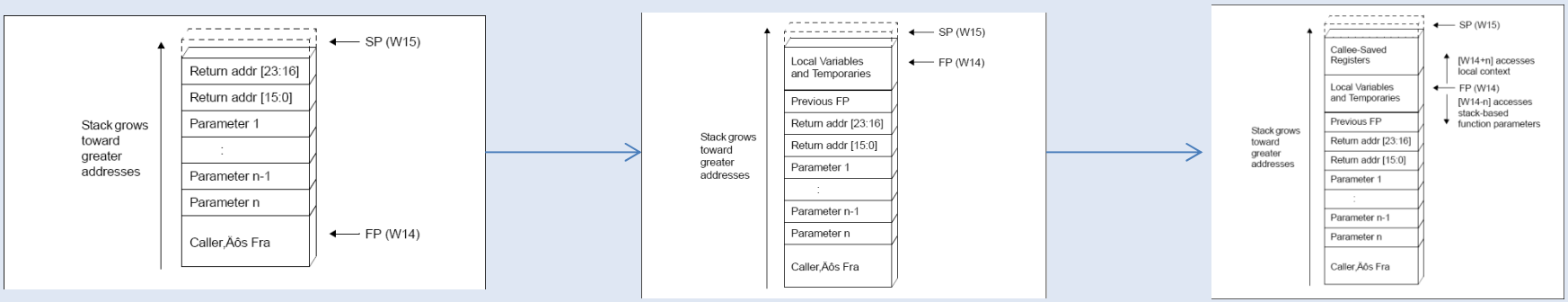



fosc = 2 * fcy

- Task (C function) executed within the timer-driven interrupt service routine (ISR)
- Timing accurate
- Single stack
- Two priorities: high priority foreground vs. background

# Scheduling IV

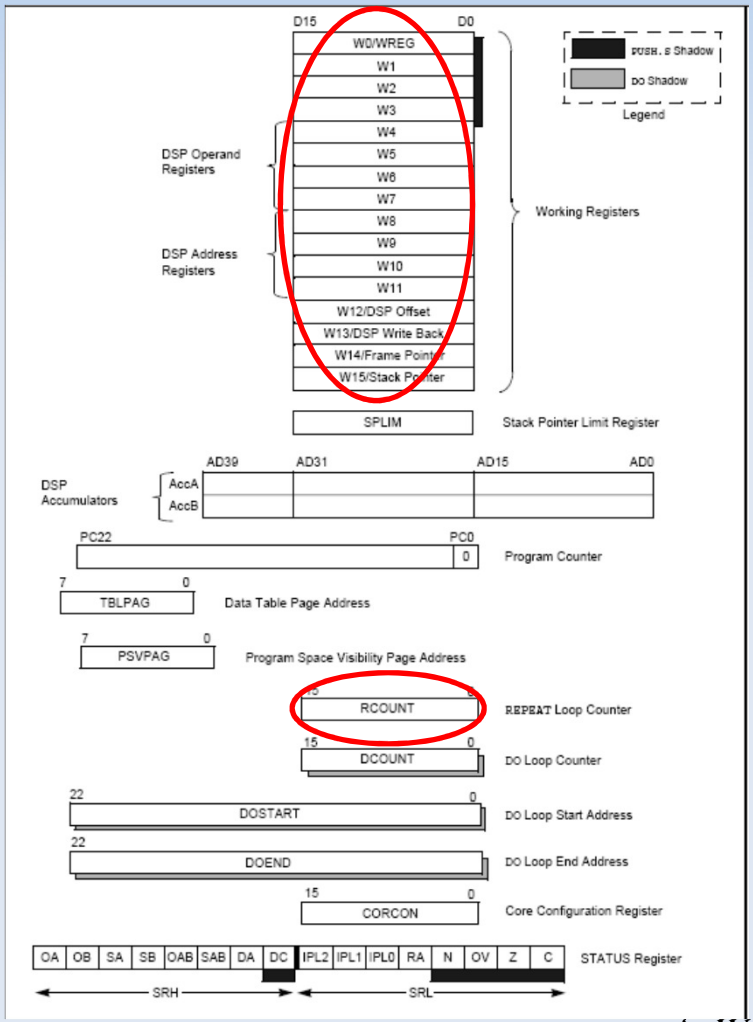
- Context switch
  - Switch from one task to another (P1 to P2)
  - Store P1 context (stack pointer if it is a multi-stack implementation, program counter, registers) – if we switch stacks we need assembly language
  - Restore P2 context
- Is there a „natural“ context switch?
  - If we work on one stack there is: function and interrupt calls save context automatically (the compiler does that for us):




call instruction (taken from microchip.com)

# Scheduling V

- Function call conventions for context:



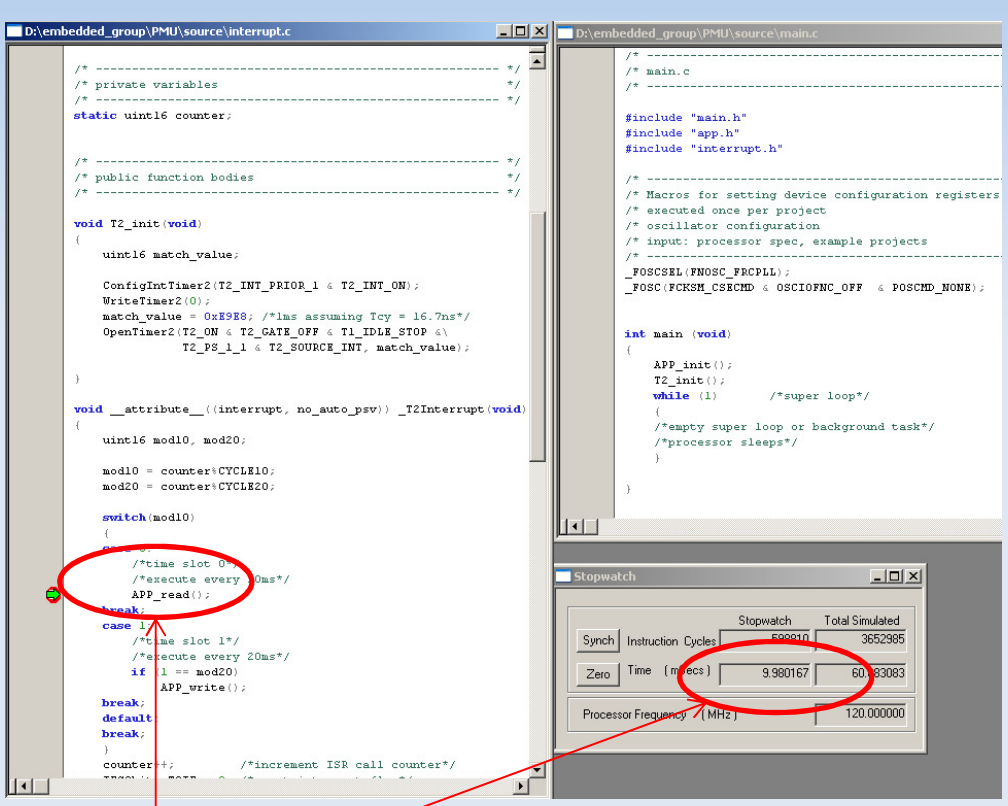

- Interrupts save context in their stack frame
  - dsPIC default: W0-W15
  - RCOUNT
  - More on demand – *save* parameter in case of dsPIC C30 compiler
  - In case of the C30 compiler this also applies for functions called within an ISR
- We conclude: a timer-driven interrupt gives us timing accuracy and saves our context

Source: microchip.com

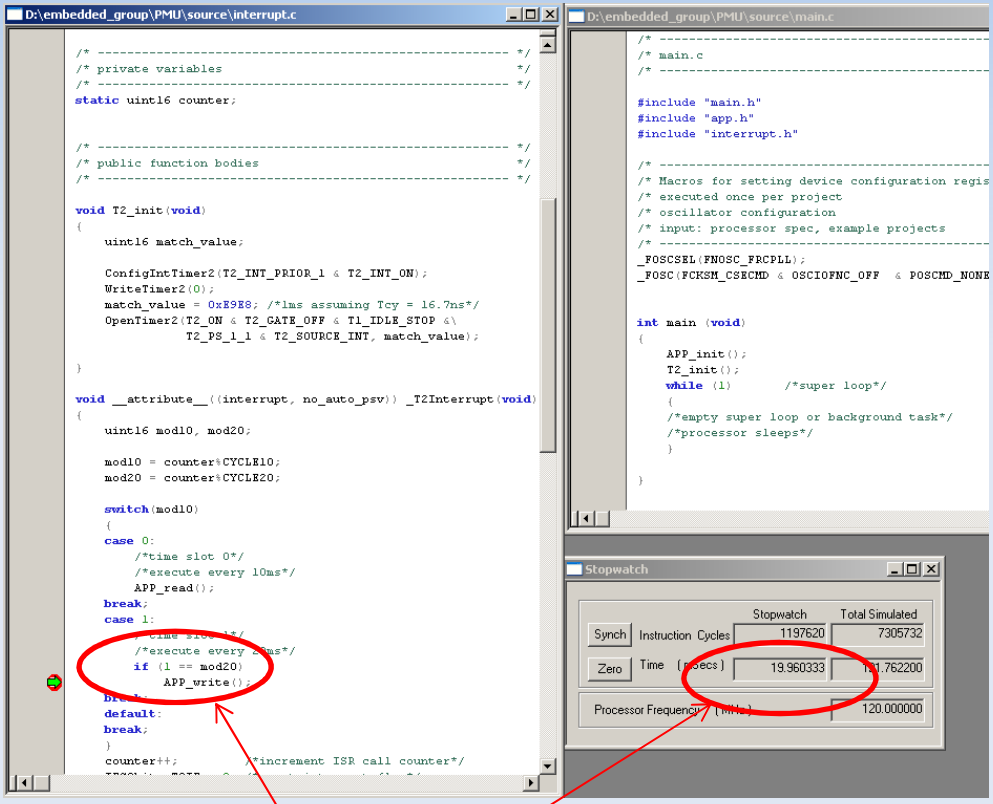
# Scheduling VI

- We can use an ISR to realize a light-weight scheduler:
  - We can call different functions at different times (round-robin based on elapsed time to realize different cycle times)
  - All tasks are C functions that run to completion
  - We can put a background task into the while(1){...} loop in main. E.g. serial communication
  - BUT: does not really work well if we do have different asynchronous sources of interrupt (e.g. timer and ADC)
- Why do we use our own scheduler at all?
  - Cost of commercial OS
  - Lack of certificate (if we need to certify we need to show that the OS meets the criteria of the certification)
  - Therefore, a very simple scheduler might be a good alternative

# Scheduling VII



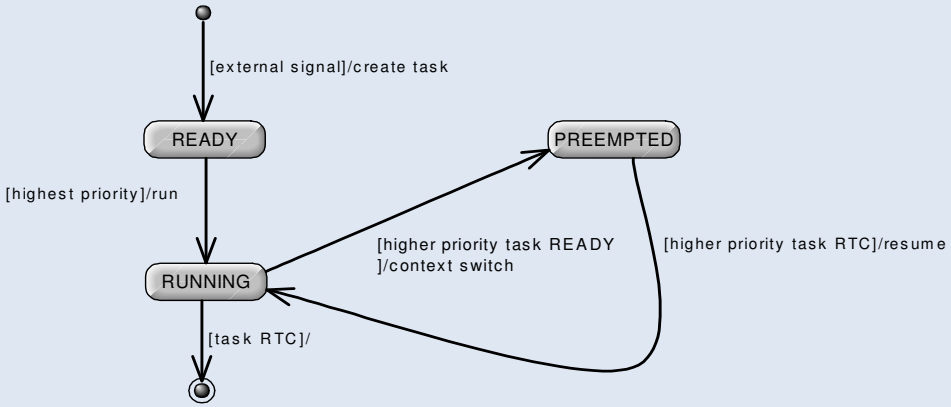

10ms cycle

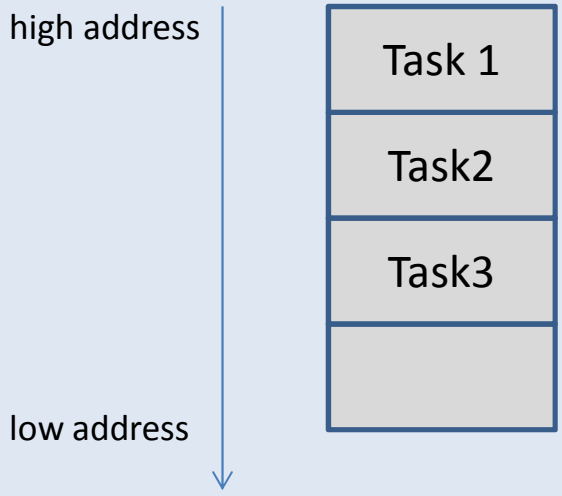



20ms cycle

# Scheduling VIII

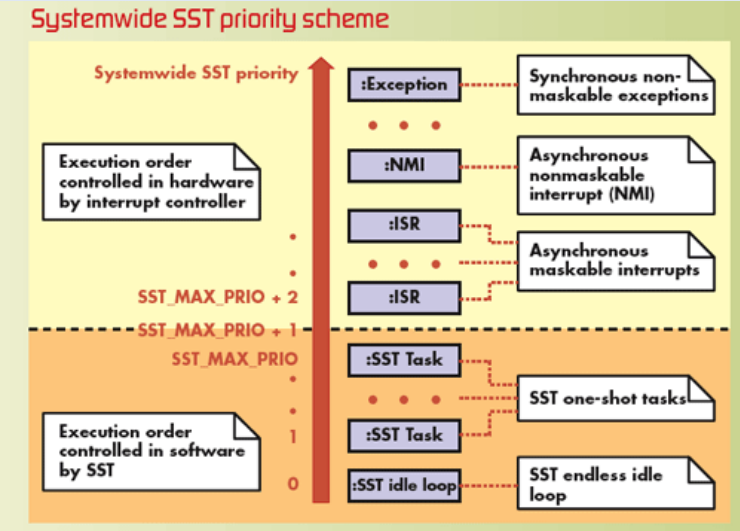
- For multiple sources of interrupt we can realize a fixed-priority single-stack scheduler using plain C (compiler takes care of context)
  - Every task is realized by a non-blocking (does not wait for external signal) thread of execution
  - Once an IRQ is fired it is marked for execution (READY) and is run if no task of higher priority is currently running

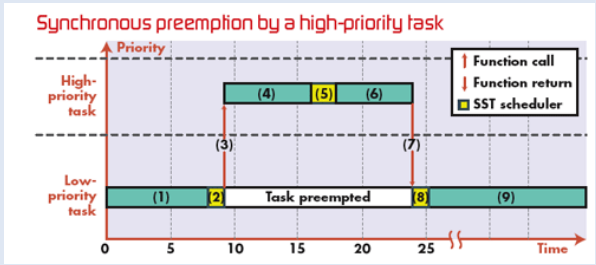


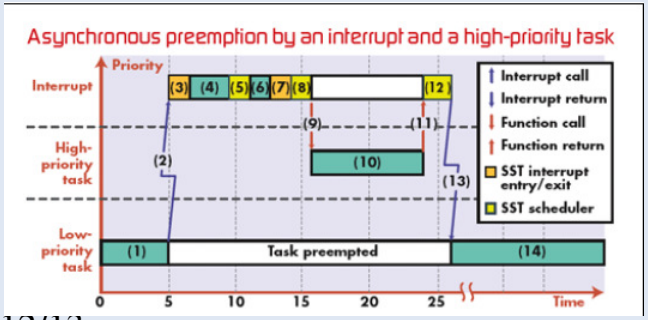



# Scheduling IX

- State-machine based task execution (no state for resource waiting)
- C compiler ISR handling takes over task switching
- Different interrupt sources (timer, ADC, etc) can trigger task creation (post event and mark task for execution)
- Refer to „Build a super simple tasker“ http://www.state-machine.com/resources/articles.php

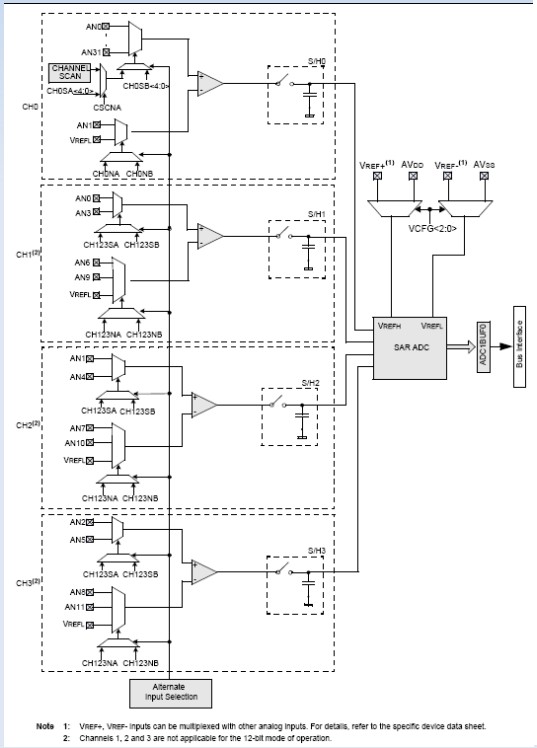


# Peripherals

- Embedded processors do come with many peripherals. Usually we need two things:
  - The peripheral specification (tells what bit to set to achieve a certain functionality)
  - The device specification
  - An example program or library (tells us the sequence of necessary steps and gives a quick feedback on functionality)




**dsPIC33F/PIC24H/dsPIC33E/PIC24E ADC Peripheral Module Library Help**

**Table of Contents**



Source: microchip.com

# Peripherals II

- It might also be a good idea to define wrapper function to
  - Abstract the hardware – useful when working with different embedded processors
  - Allow unified error handling
- Common functions: OpenPeripheral(), ClosePeripheral(), ReadPeripheral(), WritePeripheral(),...
- Macros: EnableInterrupt, DisableInterrupt,...
- In case of the dsPIC Microchip provides ready-to-use functions for peripherals, watchdog, and reset (other IC vendors do as well).

# Fault Detection

- Specific to safety-related systems in industrial domain (IEC61508-2) to achieve a higher DC -> influences PFD/PFH and architectural constraints
- What is a DC (diagnostic coverage)?
  - Hardware failures can lead to hazardous system states (not good!) which can result in harm (very bad!) – but they do not have to necessarily
  - DC is the percentage of faults that are detected by checks; $\lambda_{dd} = \lambda_d \times DC/100$
  - A system (in the safety world of 61508) is allowed to fail safe (but it is not available anymore – fail-operational).
  - If we can avoid a dangerous system failure by detecting dangerous component faults ($\lambda_d$) in advance we can transfer $\lambda_d$ into $\lambda_s$ (actually a $\lambda_{du}$ is transferred into a $\lambda_{dd}$).
  - DC comes in four categories: no (<60%), low (60% < DC < 90%), medium (90% < DC < 99%), high (DC > 99%)

# Fault Detection II

| Component | See table(s) | Requirements for diagnostic coverage or safe failure fraction claimed | | |
|---|---|---|---|---|
| | | Low (60 %) | Medium (90 %) | High (99 %) |
| **Electromechanical devices** | A.2 | Does not energize or de-energize<br>Welded contacts | Does not energize or de-energize<br>Individual contacts welded | Does not energize or de-energize<br>Individual contacts welded<br>No positive guidance of contacts (for relays this failure is not assumed if they are built and tested according to EN 50205 or equivalent)<br>No positive opening (for position switches this failure is not assumed if they are built and tested according to EN 60947-5-1, or equivalent) |
| **Discrete hardware** | A.3, A.7, A.9, A.11 | | | |
| Digital I/O | | Stuck-at | DC fault model | DC fault model<br>drift and oscillation |
| Analogue I/O | | Stuck-at | DC fault model<br>drift and oscillation | DC fault model<br>drift and oscillation |
| Power supply | | Stuck-at | DC fault model<br>drift and oscillation | DC fault model<br>drift and oscillation |
| **Bus** | A.3 | | | |
| General | A.7 | Stuck-at of the addresses | Time out | Time out |
| Memory management unit | A.8 | Stuck-at of data or addresses | Wrong address decoding | Wrong address decoding |
| Direct memory access | | No or continuous access | DC fault model for data and addresses | All faults which affect data in the memory |
| | | | Wrong access time | Wrong data or addresses<br>Wrong access time |
| Bus-arbitration (see note 1) | | Stuck-at of arbitration signals | No or continuous arbitration | No or continuous or wrong arbitration |
| **CPU** | A.4, A.10 | | | |
| Register, internal RAM | | Stuck-at for data and addresses | DC fault model for data and addresses | DC fault model for data and addresses<br>Dynamic cross-over for memory cells<br>No, wrong or multiple addressing |
| Coding and execution including flag register | | Wrong coding or no execution | Wrong coding or wrong execution | No definite failure assumption |
| Address calculation | | Stuck-at | DC fault model | No definite failure assumption |
| Program counter, stack pointer | | Stuck-at | DC fault model | DC fault model |
| **Interrupt handling** | A.4 | No or continuous interrupts | No or continuous interrupts<br>Cross-over of interrupts | No or continuous interrupts<br>Cross-over of interrupts |

| Component | See table(s) | Requirements for diagnostic coverage or safe failure fraction claimed | | |
|---|---|---|---|---|
| | | Low (60 %) | Medium (90 %) | High (99 %) |
| **Invariable memory** | A.5 | Stuck-at for data and addresses | DC fault model for data and addresses | All faults which affect data in the memory |
| **Variable memory** | A.6 | Stuck-at for data and addresses | DC fault model for data and addresses<br>Change of information caused by soft-errors for DRAM with integration 1 Mbits and higher | DC fault model for data and addresses<br>Dynamic cross-over for memory cells<br>No, wrong or multiple addressing<br>Change of information caused by soft-errors for DRAM with integration 1 Mbits and higher |
| **Clock (quartz)** | A.12 | Sub- or super-harmonic | Sub- or super-harmonic | Sub- or super-harmonic |
| **Communication and mass storage** | A.13 | Wrong data or addresses<br>No transmission | All faults which affect data in the memory<br>Wrong data or addresses<br>Wrong transmission time<br>Wrong transmission sequence | All faults which affect data in the memory<br>Wrong data or addresses<br>Wrong transmission time<br>Wrong transmission sequence |
| **Sensors** | A.14 | Stuck-at | DC fault model<br>Drift and oscillation | DC fault model<br>Drift and oscillation |
| **Final elements** | A.15 | Stuck-at | DC fault model<br>Drift and oscillation | DC fault model<br>Drift and oscillation |

NOTE 1 Bus-arbitration is the mechanism for deciding which device has control of the bus.

NOTE 2 "Stuck-at" is a fault category which can be described with continuous "0" or "1" or "on" at the pins of a component.

NOTE 3 "DC fault model" (DC = direct current) includes the following failure modes: stuck-at faults, stuck-open, open or high impedance outputs as well as short circuits between signal lines.

Source: IEC61508-2, general faults to be detected or analyzed

# Fault Detection III

- Invariable memory and variable memory

| Diagnostic technique/measure | See IEC 61508-7 | Maximum diagnostic coverage considered achievable | Notes |
|---|---|---|---|
| Word-saving multi-bit redundancy | A.4.1 | Medium | |
| Modified checksum | A.4.2 | Low | |
| Signature of one word (8-bit) | A.4.3 | Medium | The effectiveness of the signature depends on the width of the signature in relation to the block length of the information to be protected |
| Signature of a double word (16-bit) | A.4.4 | High | The effectiveness of the signature depends on the width of the signature in relation to the block length of the information to be protected |
| Block replication | A.4.5 | High | |

| Diagnostic technique/measure | See IEC 61508-7 | Maximum diagnostic coverage considered achievable | Notes |
|---|---|---|---|
| RAM test "checkerboard" or "march" | A.5.1 | Low | |
| RAM test "walk-path" | A.5.2 | Medium | |
| RAM test "galpat" or "transparent galpat" | A.5.3 | High | |
| RAM test "Abraham" | A.5.4 | High | |
| Parity-bit for RAM | A.5.5 | Low | |
| RAM monitoring with a modified Hamming code, or detection of data failures with error-detection-correction codes (EDC) | A.5.6 | High | |
| Double RAM with hardware or software comparison and read/write test | A.5.7 | High | |

NOTE 1 This table does not replace any of the requirements of annex C.
NOTE 2 The requirements of annex C are relevant for the determination of diagnostic coverage.
NOTE 3 For general notes concerning this table, see the text preceding table A.1.
NOTE 4 For RAM which is read/written only infrequently (for example during configuration) the measures A.4.1 to A.4.4 are effective if they are executed after each read/write access.

Source: IEC61508-2

# Fault Detection IV

- IO

| Diagnostic technique/measure | See IEC 61508-7 | Maximum diagnostic coverage considered achievable | Notes |
|---|---|---|---|
| Failure detection by on-line monitoring | A.1.1 | Low (low demand mode)<br>Medium (high demand or continuous mode) | Depends on diagnostic coverage of failure detection |
| Test pattern | A.6.1 | High | |
| Code protection | A.6.2 | High | |
| Multi-channel parallel output | A.6.3 | High | Only if dataflow changes within diagnostic test interval |
| Monitored outputs | A.6.4 | High | Only if dataflow changes within diagnostic test interval |
| Input comparison/voting (1oo2, 2oo3 or better redundancy) | A.6.5 | High | Only if dataflow changes within diagnostic test interval |
| NOTE 1 This table does not replace any of the requirements of annex C.<br>NOTE 2 The requirements of annex C are relevant for the determination of diagnostic coverage.<br>NOTE 3 For general notes concerning this table, see the text preceding table A.1. | | | |

- Program sequence

| Diagnostic technique/measure | See IEC 61508-7 | Maximum diagnostic coverage considered achievable | Notes |
|---|---|---|---|
| Watch-dog with separate time base without time-window | A.9.1 | Low | |
| Watch-dog with separate time base and time-window | A.9.2 | Medium | |
| Logical monitoring of program sequence | A.9.3 | Medium | Depends on the quality of the monitoring |
| Combination of temporal and logical monitoring of programme sequences | A.9.4 | High | |
| Temporal monitoring with on-line check | A.9.5 | Medium | |
| NOTE 1 This table does not replace any of the requirements of annex C.<br>NOTE 2 The requirements of annex C are relevant for the determination of diagnostic coverage.<br>NOTE 3 For general notes concerning this table, see the text preceding table A.1. | | | |

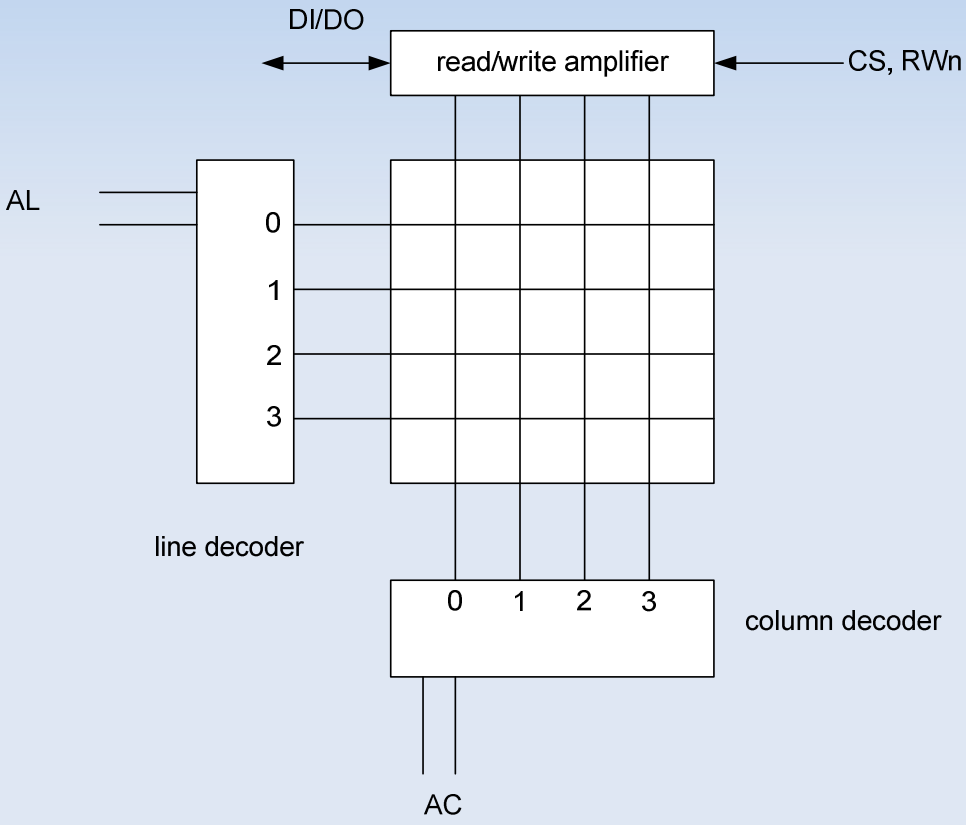
Source: IEC61508-2

# Fault Detection V

- Fault detection is a series of activities that happen at startup, background (cyclic tests) and specific maintenance cycles
  - CPU
  - Memory (used one)
  - IO
  - Program sequence
- Basic tool for fault detection evaluation is FMEA/FMEDA
- Time-critical test is cyclic background test since it checks physical resource during operation (must align to the process safety time specified in the systems requirements).

# Fault Detection
# - Memory -

- Memory matrix organization (1-bit … n-bit) – in reality one data word stored at a specific address
- address decoder, read and write amplifiers, control signals, data in and out
- low diagnostic coverage: stuck-at for data and/or address (constantly ‘0’ or ‘1’)
- medium diagnostic coverage: DC fault model for data and address (stuck-at, high-Z, X-talk)

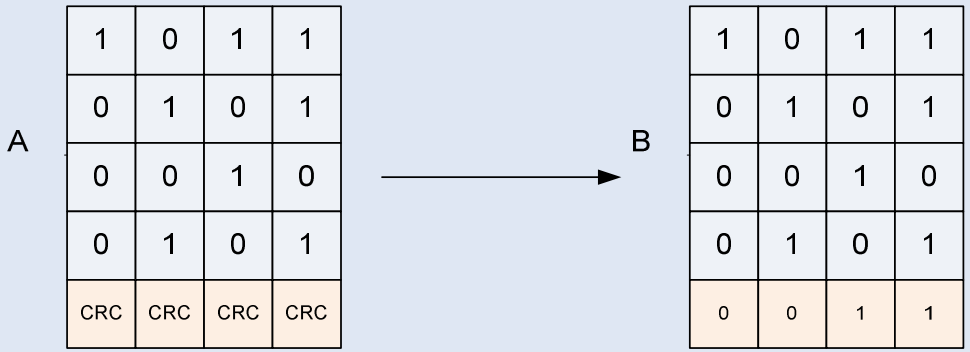


# Fault Detection
# - Non-variable Memory (program memory) -

- Modified checksum test, based on XOR and circular shift operations
- Defined checksum is compared to the checksum calculated during operation
- Odd-numbered bit errors within a column are detected
- Low diagnostic coverage test

# Fault Detection
# - Non-variable Memory (program memory) II -

- Signature of one word test (CRC), based on Modulo-2 (XOR) arithmetic
- Memory content is interpreted as a bit stream
- Division by a defined polynomial yields zero,  P(X) = 11001 in the example
- All one bit and multi-bit failures within one word and 99.6% of all possible bit failures are detected
- Medium diagnostic coverage test

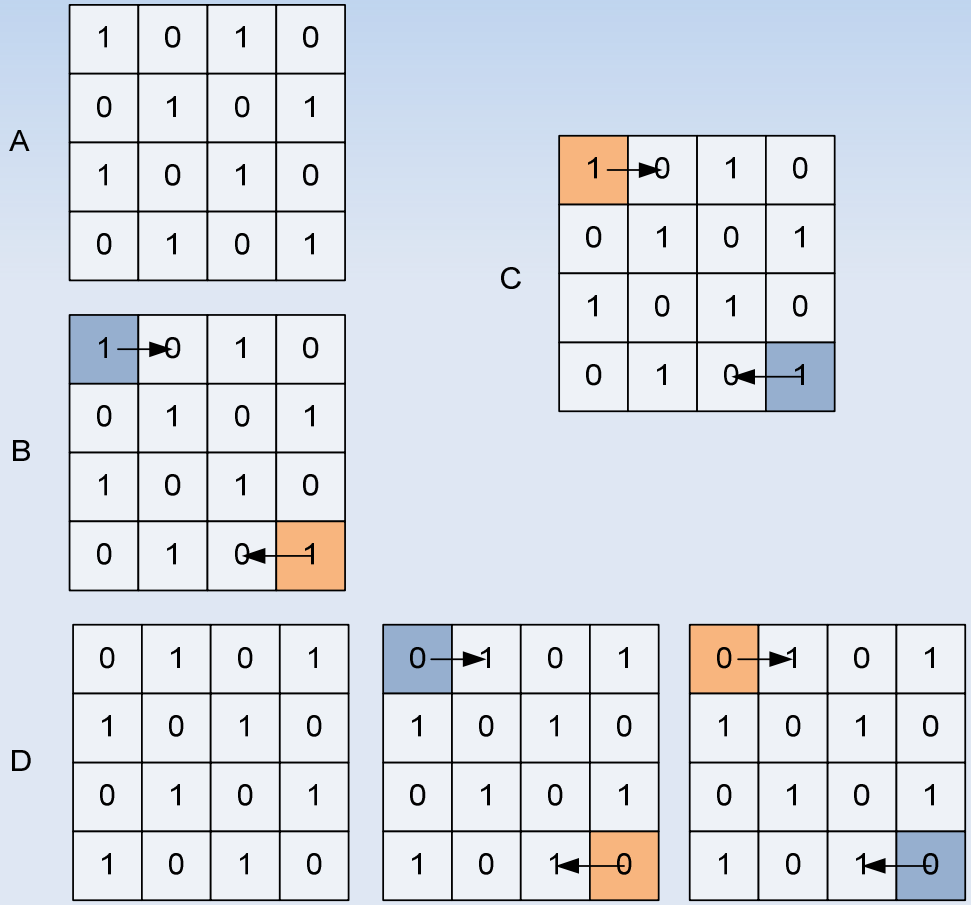


# Fault Detection
# - Non-variable memory (EEPROM) -

- EEPROM content is copied to SRAM and verified during system initialization -> working copy
- All changes are made to working copy
- Working copy is written to EEPROM before power-down or at defined slow cycles (wear-out effect!)
- EEPROM test is reduced to a RAM test – we work from RAM data

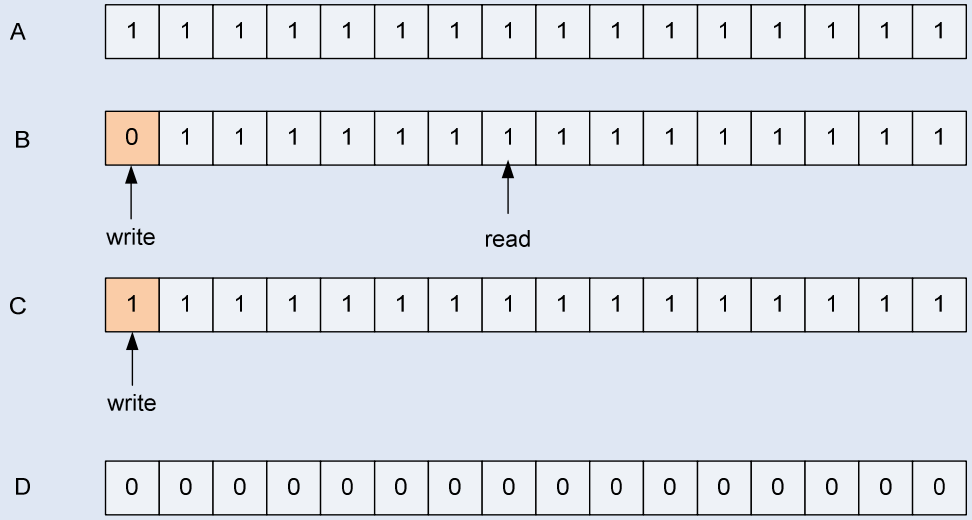
# Fault Detection
# - Variable memory (SRAM) -

- Checkerboard test – low diagnostic coverage
- Cells are checked for correct content in pairs
- Initialization, upward test, downward test, inverse initialization, upward test, downward test -> 10 * n complexity (number of load store operations)
- Pairs are address inverse



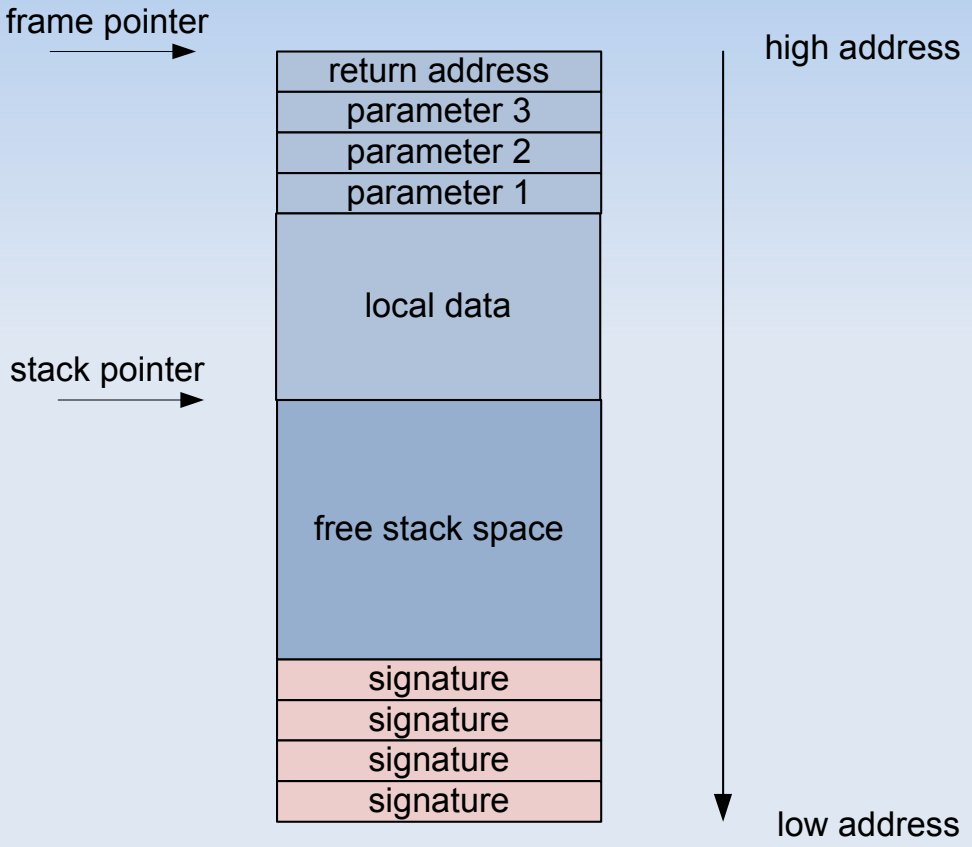
# Fault Detection
# - Variable memory (SRAM) II -

- Walking pattern - medium diagnostic coverage
- Initialization (A), the first cell is inverted and all remaining cells are checked for correct content (B), the first cell is inverted again (C), the test is conducted again with inverse background (D) -> 2*n*n + 6*n complexity (number of load store operations)



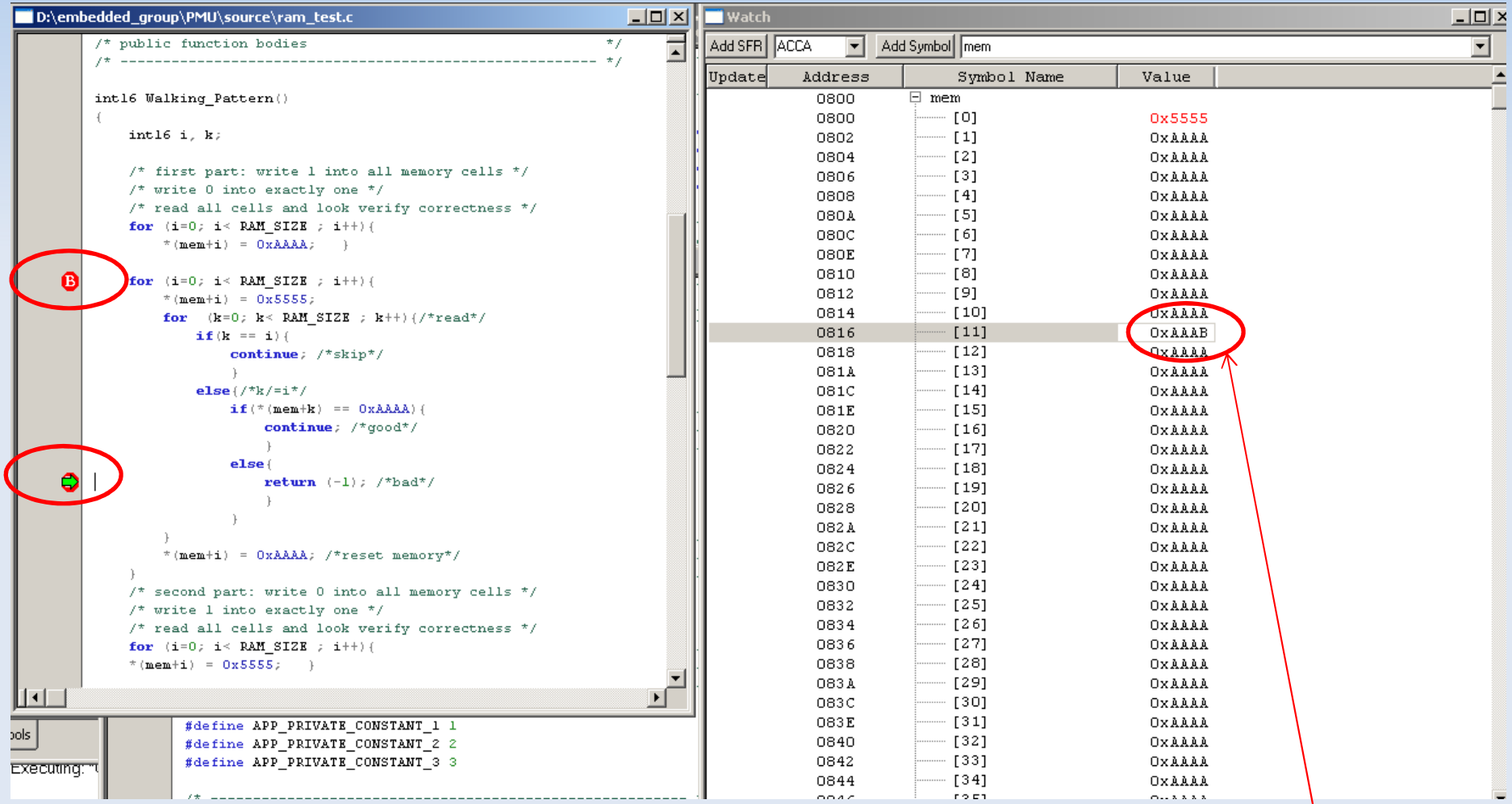
# Fault Detection
# - Variable memory (Stack) -

- Stack data integrity is checked by correct program flow (the stack stores our task context)
- Stack limits are checked by signature or addresses (some controllers provide hardware support)
- Underlying hardware (SRAM) is checked by SRAM tests



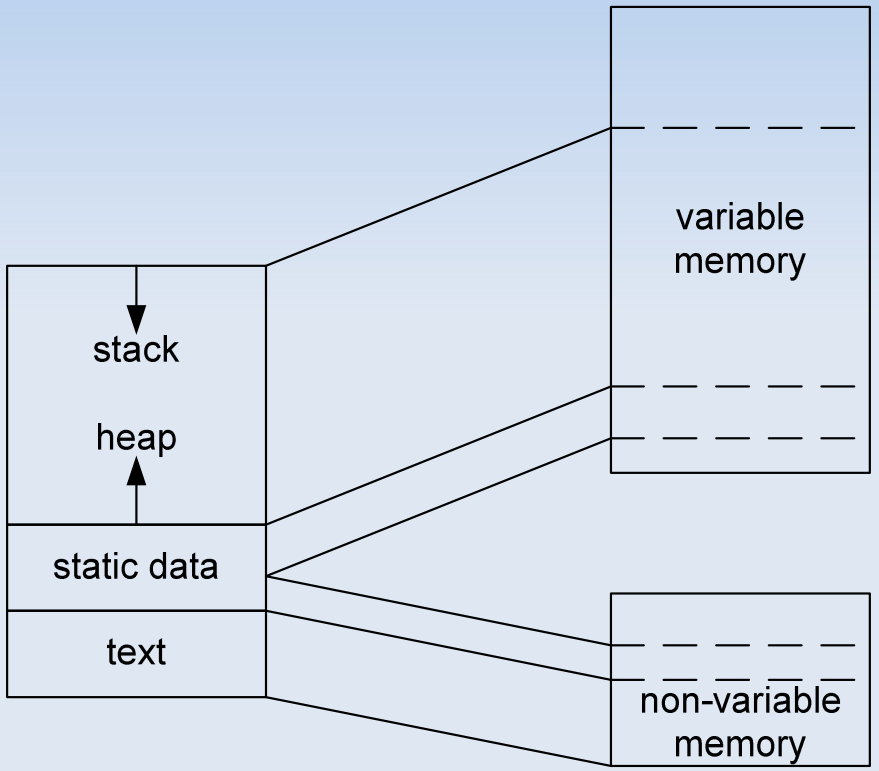
# Fault Detection
# - Example -

- RAM tests are destructive – therefore we need to safe the original data in advance



# Software Mapping

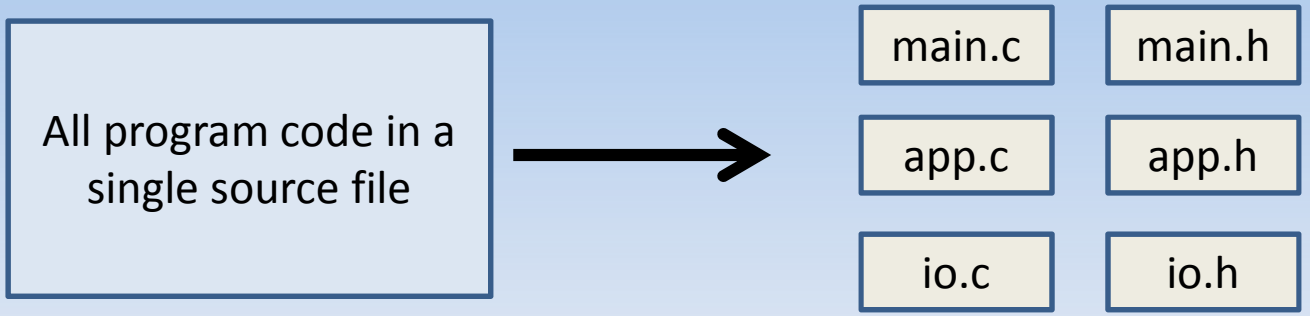
- stack – local data and spilling registers
- heap – dynamic data (do not use)
- static data
- text – program code
- logical layout mapped to physical location during linkage



# Code Structure

- Software stored in ASCII files – what is a good way to structure code?
- Object-orientation vs. precedural style
- Object-oriented programming languages (mostly C++) often not an option
  - Proven-in-use of tools challenge (safety-related software)
  - Prior experience of team
- Coding in C in an object-oriented way
  - Reusability
  - Data encapsulation
  - Function encapsulation

# Code Structure Ctd.




- Linkage: describes the accessibility of objects (something that uses memory) from one file to another, or even within the same file – internal, external, no linkage
  - Internal linkage, external object, accessible within one file
  - External linkage, external object, accessible throughout the whole program
  - No linkage, internal object, accessible from within the function
- Scope: region of a program in which an object is visible – block, file, function, function prototype

# Code Structure Ctd.

- External declarations (outside a function)

| Storage Class Specifier | Function or Data Object | Linkage | Duration |
|---|---|---|---|
| static | either | internal | static |
| extern | either | external | static |
| none | function | external | static |
| none | data object | external | static |

- Internal declarations

| Storage Class Specifier | Function or Data Object | Linkage | Duration |
|---|---|---|---|
| register | data object | none | automatic |
| auto | data object | none | automatic |
| static | data object | none | static |
| extern | either | external | static |
| none | data object | none | automatic |
| none | function | external | static |

# Code Structure Ctd.

D:\embedded_group\PMU\source\main.h

```
/* ------------------------------------------------------------ */
/* main.h                                                       */
/* ------------------------------------------------------------ */

#ifndef _MAIN_H
#define _MAIN_H

/* include uC spefic header file */
#include "p33fxxxx.h"

/* ------------------------------------------------------------ */
/* typedefs                                                     */
/* input: C compiler specification                              */
/* ------------------------------------------------------------ */
typedef int                 int16;
typedef long                int32;
typedef long long           int64;
typedef unsigned int        uint16;
typedef unsigned long       uint32;
typedef unsigned long long  uint64;
typedef float               float32;
typedef long double         float64;

/* ------------------------------------------------------------ */
/* interrupts                                                   */
/* ------------------------------------------------------------ */

#endif
/* ------------------------------------------------------------ */
/* END OF FILE                                                  */
/* ------------------------------------------------------------ */
```

D:\embedded_group\PMU\source\main.c

```
/* ------------------------------------------------------------ */
/* main.c                                                       */
/* ------------------------------------------------------------ */

#include "main.h"
#include "app.h"

/* ------------------------------------------------------------ */
/* Macros for setting device configuration registers            */
/* executed once per project                                    */
/* oscillator configuration                                     */
/* input: processor spec, example projects                      */
/* ------------------------------------------------------------ */
_FOSCSEL(FNOSC_FRCPLL);
_FOSC(FCKSM_CSECMD & OSCIOFNC_OFF  & POSCMD_NONE);


int main (void)
{
    APP_init();
    while (1)
    {
    APP_read();
    APP_write();
    }

}

/* ------------------------------------------------------------ */
/* END OF FILE                                                  */
/* ------------------------------------------------------------ */
```

| Type | Bits | Min | Max |
|---|---|---|---|
| char, signed char | 8 | -128 | 127 |
| unsigned char | 8 | 0 | 255 |
| short, signed short | 16 | -32768 | 32767 |
| unsigned short | 16 | 0 | 65535 |
| int, signed int | 16 | -32768 | 32767 |
| unsigned int | 16 | 0 | 65535 |
| long, signed long | 32 | $-2^{31}$ | $2^{31}-1$ |
| unsigned long | 32 | 0 | $2^{32}-1$ |
| long long**, signed long long** | 64 | $-2^{63}$ | $2^{63}-1$ |
| unsigned long long** | 64 | 0 | $2^{64}-1$ |

| Type | Bits | E Min | E Max | N Min | N Max |
|---|---|---|---|---|---|
| float | 32 | -126 | 127 | $2^{-126}$ | $2^{128}$ |
| double* | 32 | -126 | 127 | $2^{-126}$ | $2^{128}$ |
| long double | 64 | -1022 | 1023 | $2^{-1022}$ | $2^{1024}$ |

E = Exponent
N = Normalized (approximate)
* double is equivalent to long double if -fno-short-double is used.

# Code Structure Ctd.

D:\embedded_group\PMU\source\app.h

```
/* ------------------------------------------------------------ */
/* app.h                                                        */
/* ------------------------------------------------------------ */

#ifndef _APP_H
#define _APP_H

/* ------------------------------------------------------------ */
/* public constants                                             */
/* ------------------------------------------------------------ */

#define APP_PUBLIC_CONSTANT_1 1
#define APP_PUBLIC_CONSTANT_2 2
#define APP_PUBLIC_CONSTANT_3 3

/* ------------------------------------------------------------ */
/* public function prototypes                                   */
/* ------------------------------------------------------------ */

void APP_init();
void APP_write();
void APP_read();

#endif
/* ------------------------------------------------------------ */
/* END OF FILE                                                  */
/* ------------------------------------------------------------ */
```

D:\embedded_group\PMU\source\app.c

```
/* ------------------------------------------------------------ */
/* app.c                                                        */
/* ------------------------------------------------------------ */

#include "main.h"
#include "app.h"

/* ------------------------------------------------------------ */
/* public variable definitions                                  */
/* ------------------------------------------------------------ */

uint16 public1;
uint16 public2;

/* ------------------------------------------------------------ */
/* private function prototypes                                  */
/* ------------------------------------------------------------ */

static void APP_private();

/* ------------------------------------------------------------ */
/* private constants                                            */
/* ------------------------------------------------------------ */

#define APP_PRIVATE_CONSTANT_1 1
#define APP_PRIVATE_CONSTANT_2 2
#define APP_PRIVATE_CONSTANT_3 3

/* ------------------------------------------------------------ */
/* private variables                                            */
/* ------------------------------------------------------------ */

static uint16 private1;
static uint16 private2;

/* ------------------------------------------------------------ */
/* public function bodies                                       */
/* ------------------------------------------------------------ */

void APP_init()
{}
void APP_write()
{}
void APP_read()
{}
/* ------------------------------------------------------------ */
/* END OF FILE                                                  */
/* ------------------------------------------------------------ */
```

# Questions?