

Industrial Embedded Systems - Design for Harsh Environment -

Dr. Alexander Walsch
alexander.walsch@ge.com

IN 2244

Part V

WS 2013/14

Technische Universität München

Architecture Patterns

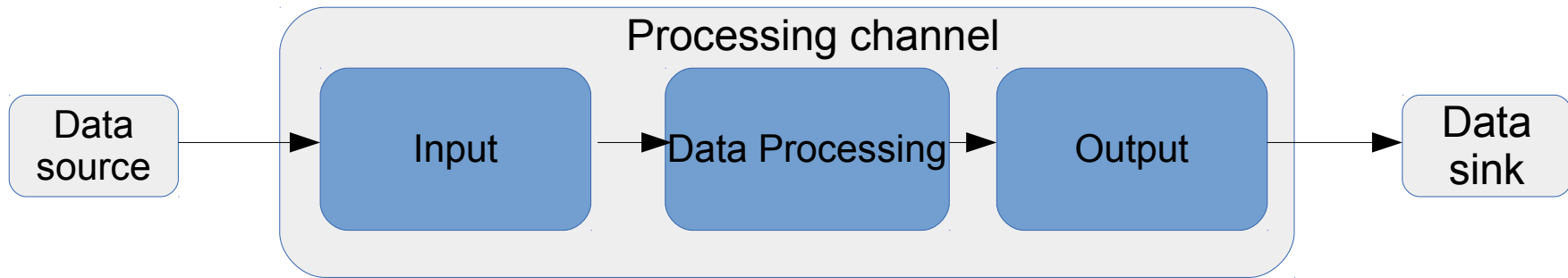
- Recurring Hardware and Software building blocks
- Focus on
 - Reliability – see lecture slides on reliability
 - Safety – see lecture slides on safety
- Keep in mind: faults can be random or systematic
- Design decisions are based on reasoning (FTA, FMEA) and recommendations (e.g. safety architectures)

Software Design Concepts

	Technique/Measure *	Ref.	SIL 1	SIL 2	SIL 3	SIL 4
	Architecture and design feature					
1	Fault detection	C.3.1	---	R	HR	HR
2	Error detecting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Diverse monitor techniques (with independence between the monitor and the monitored function in the same computer)	C.3.4	---	R	R	----
3c	Diverse monitor techniques (with separation between the monitor computer and the monitored computer)	C.3.4	---	R	R	HR
3d	Diverse redundancy, implementing the same software safety requirements specification	C.3.5	---	---	---	R
3e	Functionally diverse redundancy, implementing different software safety requirements specification	C.3.5	---	---	R	HR
3f	Backward recovery	C.3.6	R	R	---	NR
3g	Stateless software design (or limited state design)	C.2.12	---	---	R	HR
4a	Re-try fault recovery mechanisms	C.3.7	R	R	---	---
4b	Graceful degradation	C.3.8	R	R	HR	HR
5	Artificial intelligence - fault correction	C.3.9	---	NR	NR	NR
6	Dynamic reconfiguration	C.3.10	---	NR	NR	NR
7	Modular approach	Table B.9	HR	HR	HR	HR
8	Use of trusted/verified software elements (if available)	C.2.10	R	HR	HR	HR
9	Forward traceability between the software safety requirements specification and software architecture	C.2.11	R	R	HR	HR
10	Backward traceability between the software safety requirements specification and software architecture	C.2.11	R	R	HR	HR
11a	Structured diagrammatic methods **	C.2.1	HR	HR	HR	HR
11b	Semi-formal methods **	Table B.7	R	R	HR	HR
11c	Formal design and refinement methods **	B.2.2, C.2.4	---	R	R	HR
11d	Automatic software generation	C.4.6	R	R	R	R
12	Computer-aided specification and design tools	B.2.4	R	R	HR	HR
13a	Cyclic behaviour, with guaranteed maximum cycle time	C.3.11	R	HR	HR	HR
13b	Time-triggered architecture	C.3.11	R	HR	HR	HR
13c	Event-driven, with guaranteed maximum response time	C.3.11	R	HR	HR	-
14	Static resource allocation	C.2.6.3	-	R	HR	HR
15	Static synchronisation of access to shared resources	C.2.6.3	-	-	R	HR

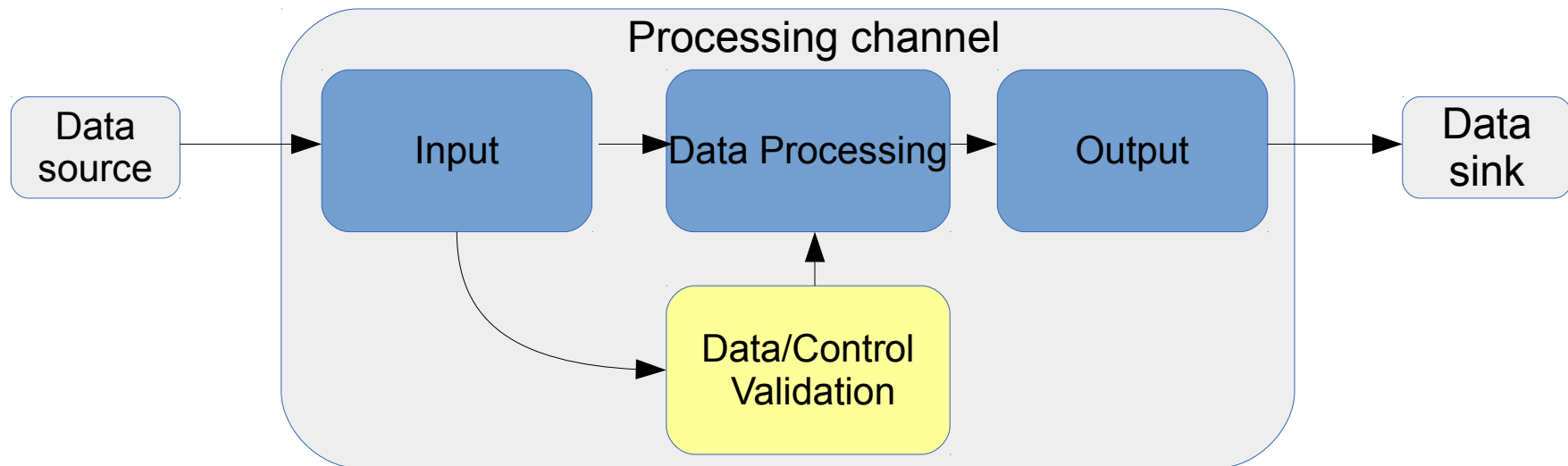
Source: IEC61508-3

Base Channel



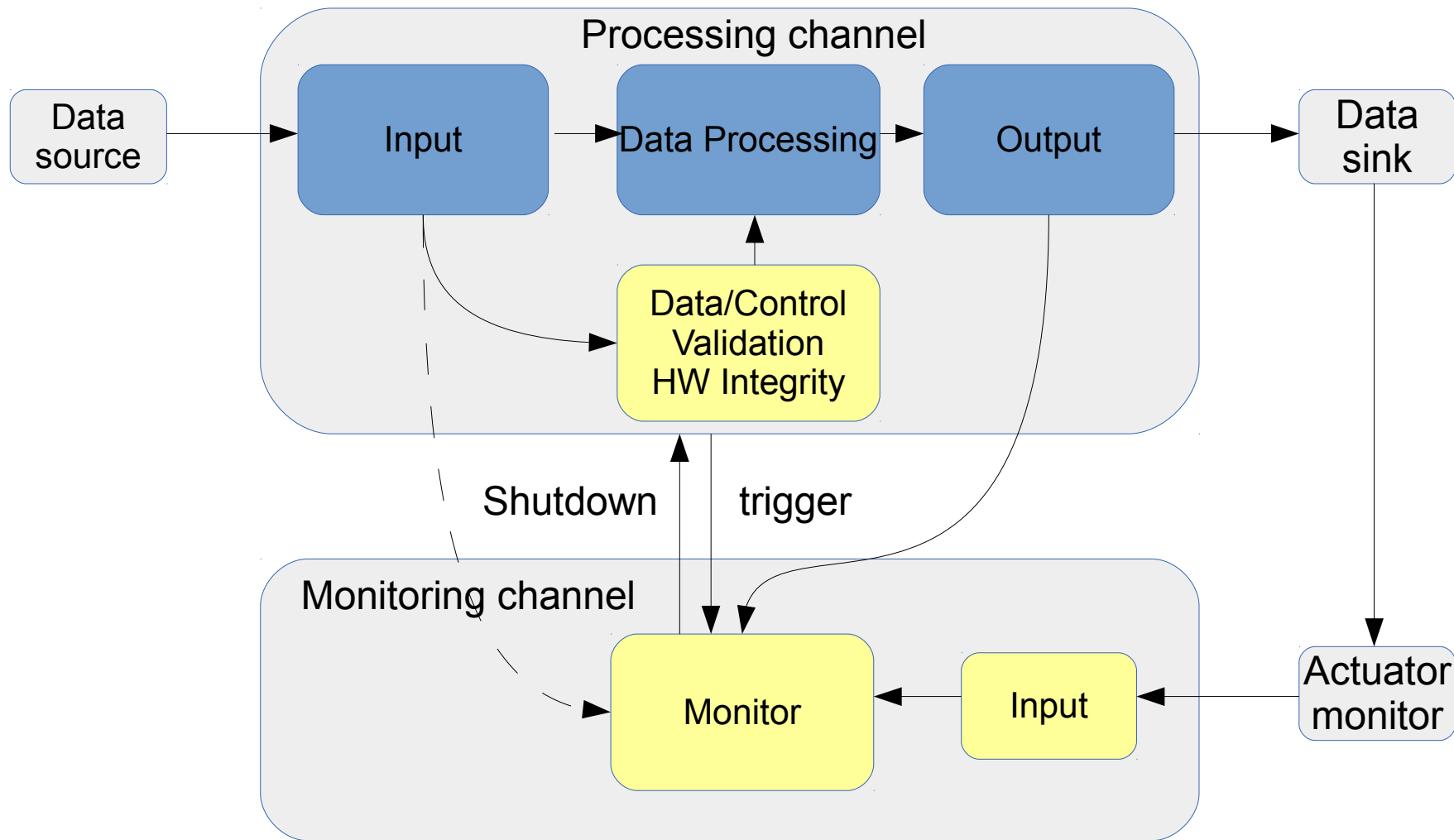
- Reliability (random faults): see previous calculations
- Reliability (systematic faults): highly affected
- Safety: 1oo1 architecture, not used

Protected Channel



- Still 1oo1.
- Provides some data and control flow checks (self-monitoring)
 - Internal watchdog, acceptance tests
- Use: not used in safety-related applications, reliability increase (depends on application)

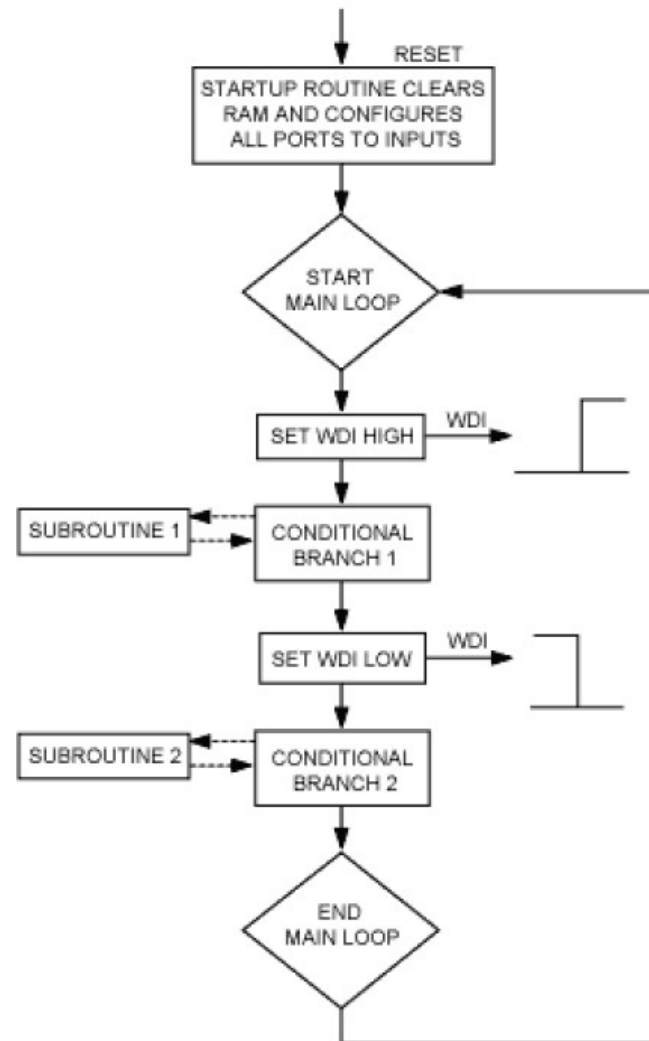
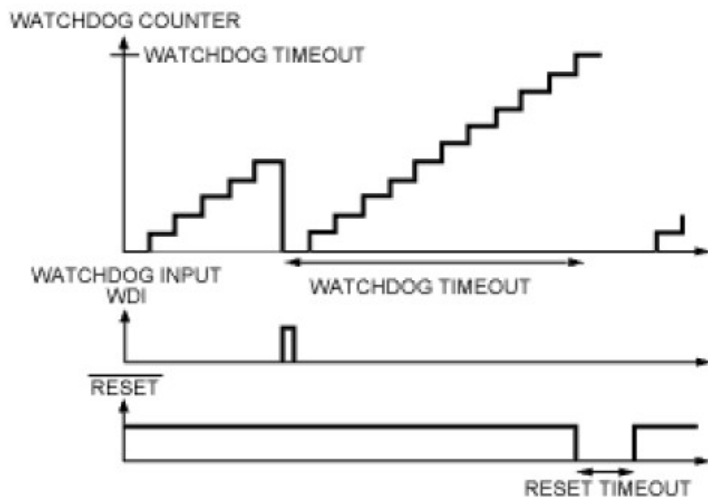
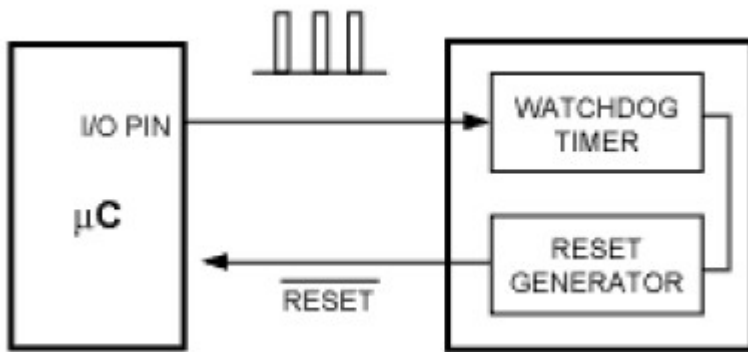
External Diagnostics (MooND Architectures)



Watchdog Circuits

- A watchdog timer is a supervisory component which must be triggered in regular intervals in order to avoid system reset
- Embedded processors usually come with internal watchdog circuits.
- A failure mode (drift) of the oscillator (account for in FMEA) makes a second external one with a separate clock source highly advisable for robust systems.
- Internal watchdogs can be disabled accidentally by software
- Set and reset the watchdog in different parts of the software to disallow stuck-at watchdog pulse loops

Watchdog Circuits II

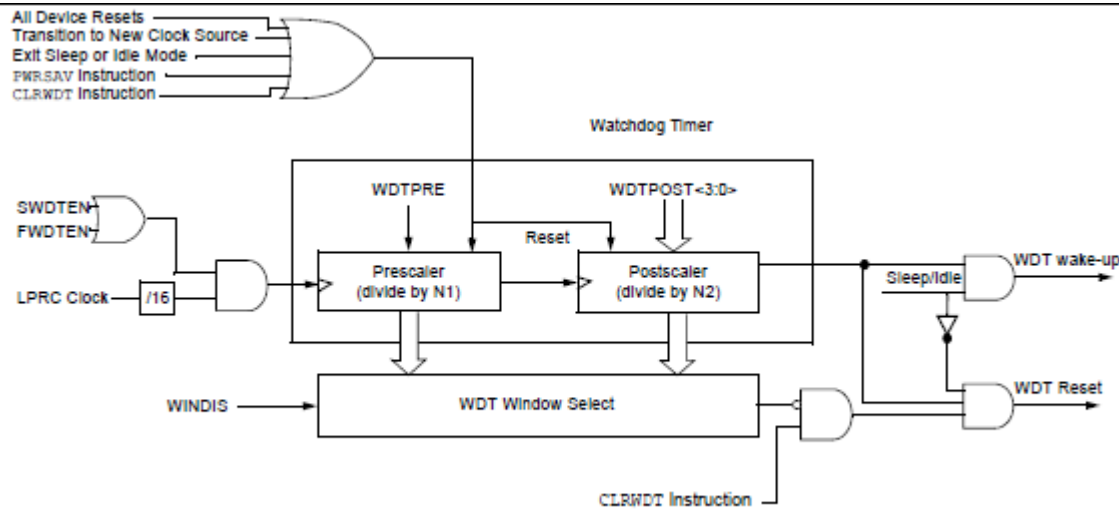


Source:
Maxim AN1926

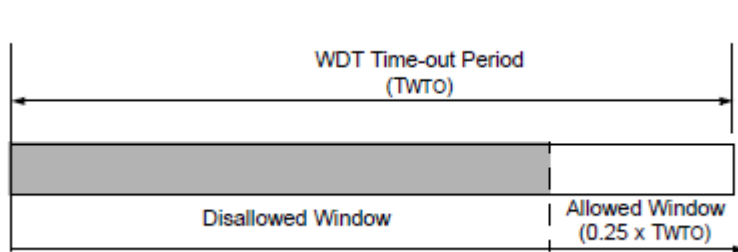
Watchdog Circuits III

- Standard watchdog
- Windowed watchdog

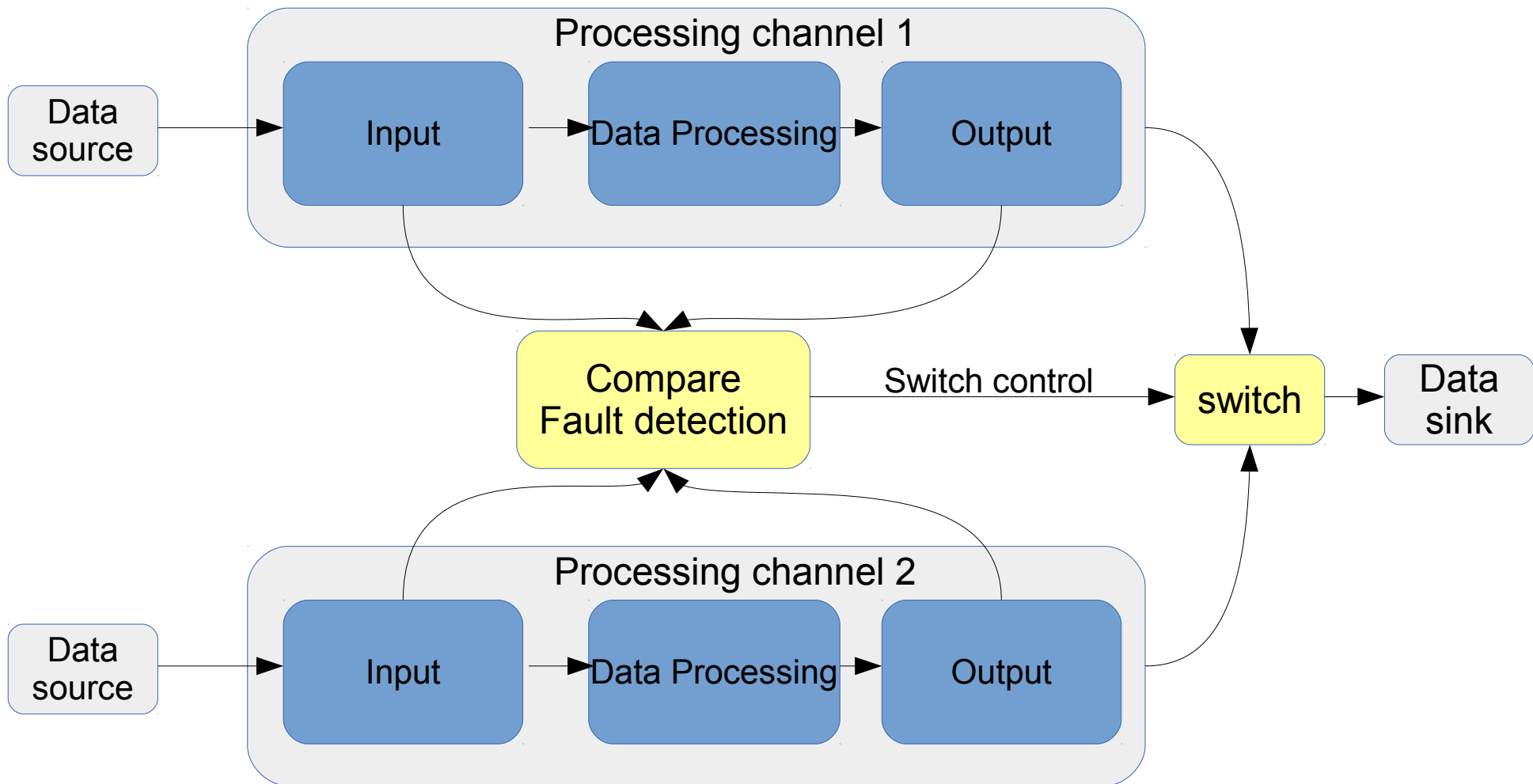
Source:
Microchip, dsPIC30F



Note: See Table 36-2 for the Prescaler divider ratio (N1) and Table 36-3 for the Postscaler divider ratio (N2).



Multiple Channels



Operating Systems (widely used)

OS	Vendor	Domain certification
VxWorks CERT	Windriver	Industry, Aviation
Integrity	Greehills	Industry, Railway, Aviation, Healthcare
Neutrino Safe	QNX	Industry
SafeRTOS	Wittenstein	Industry
PikeOS	SYSGO	Industry, Aviation, Automotive, Railway

Scheduling

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

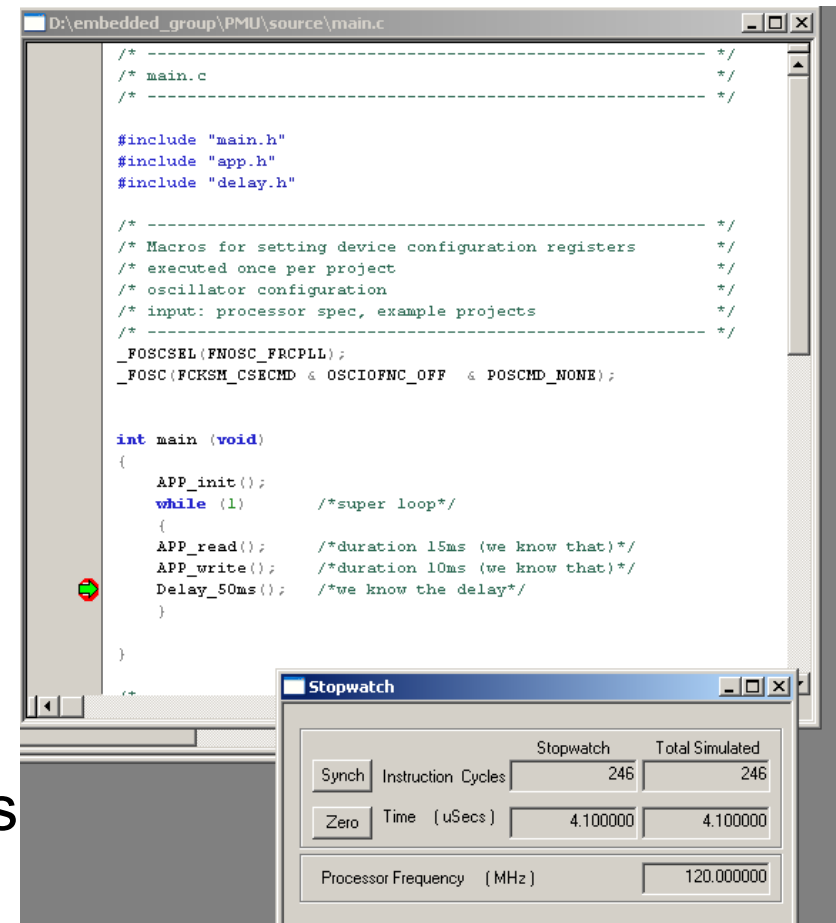
PMUSysRQ 8: Pressure readings communicated via CAN shall not be older than 100ms.

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)

Super Loop

- 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



```
D:\embedded_group\PMU\source\main.c
/* ----- */
/* main.c */
/* ----- */

#include "main.h"
#include "app.h"
#include "delay.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) /*super loop*/
    {
        APP_read(); /*duration 15ms (we know that)*/
        APP_write(); /*duration 10ms (we know that)*/
        Delay_50ms(); /*we know the delay*/
    }
}
```

		Stopwatch	Total Simulated
Synch	Instruction Cycles	246	246
Zero	Time (uSecs)	4.100000	4.100000
Processor Frequency (MHz)		120.000000	

Timer Interrupts

- Timer based interrupts:

$$f_{osc} = 2 * f_{cy}$$

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

```

D:\embedded_group\PMU\source\interrupt.c
/* -----
/* private function prototypes
/* -----
/* -----
/* private constants
/* -----
/* -----
/* private variables
/* -----
/* -----
/* public function bodies
/* -----
void T2_init(void)
{
    uint16 match_value;

    ConfigIntTimer2(T2_INT_PRIOR_1 & T2_INT_ON);
    WriteTimer2(0);
    match_value = 0xE9E8; /* 1ms assuming Tcy = 16.7ns*/
    OpenTimer2(T2_ON & T2_GATE_OFF & T1_IDLE_STOP &
               T2_PS_1_1 & T2_SOURCE_INT, match_value);
}

void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void)
{
    APP_read();
    APP_write();
    IFS0bits.T2IF = 0; /*reset interrupt flag*/
}

D:\embedded_group\PMU\source\main.c
/* -----
/* main.c
/* -----

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

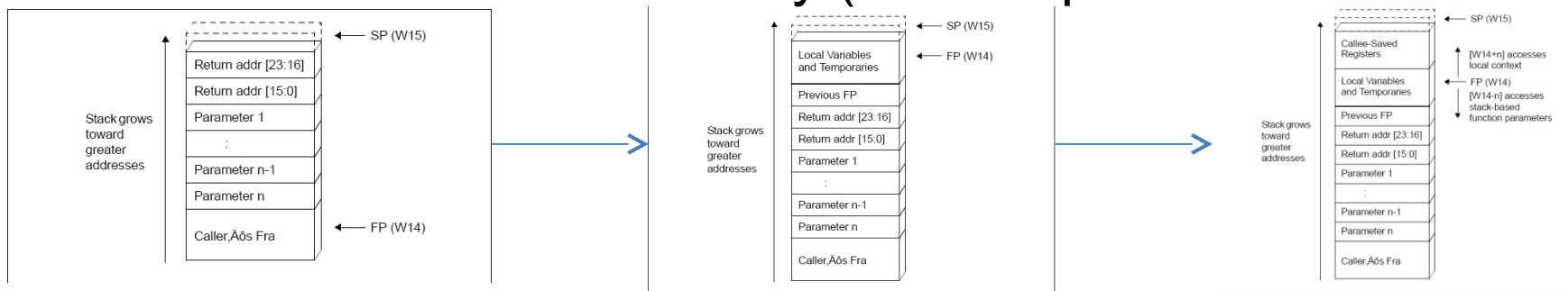
/* -----
/* Macros for setting device configuration registers
/* executed once per project
/* oscillator configuration
/* input: processor spec, example projects
/* -----
_FOSCSEL(FNOSC_FRCPLL);
_FOSC(FCKSM_CSECMD & OSCIOFNC_OFF & POSCMD_NONE);

int main(void)
{
    APP_init();
    T2_init();
    while (1) /*super loop*/
    {
        /*empty super loop or background task*/
        /*processor sleeps*/
    }
}

Stopwatch
Stopwatch      Total Simulated
Synchronizing Instruction Cycles 59881 898378
Zero Time (µsecs) 998.016667 14572.966667
Processor Frequency (MHz) 120.000000
    
```

Context Switch

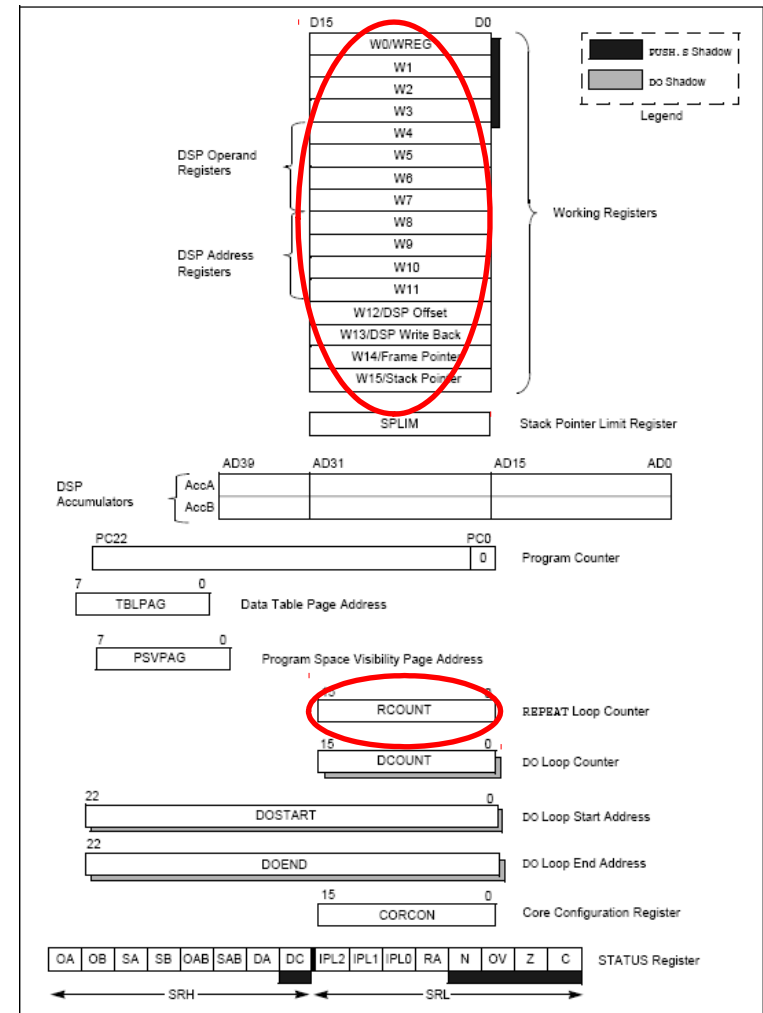
- Context switch
 - Switch from one task to another (P1 to P2)
 - Store P1 context (stack pointer if it is a multi-stack implementation, program counter, registers) – if we switch stacks we need assembly language
 - Restore P2 context
- Is there a „natural“ context switch?
 - If we work on one stack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



call instruction (taken from microchip.com)

Calling Conventions

- Interrupts save context in their handlers 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

ISRs

- 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

Code Example

```

D:\embedded_group\PMU\source\interrupt.c
/* ----- */
/* private variables */
/* ----- */
static uint16 counter;

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

void T2_init(void)
{
    uint16 match_value;

    ConfigIntTimer2(T2_INT_PRIOR_1 & T2_INT_ON);
    WriteTimer2(0);
    match_value = 0xE9E8; /*1ms assuming Tcy = 16.7ns*/
    OpenTimer2(T2_ON & T2_GATE_OFF & T1_IDLE_STOP &
               T2_PS_1_1 & T2_SOURCE_INT, match_value);
}

void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void)
{
    uint16 mod10, mod20;

    mod10 = counter&CYCLE10;
    mod20 = counter&CYCLE20;

    switch(mod10)
    {
        case 0:
            /*time slot 0*/
            /*execute every 10ms*/
            APP_read();
            break;
        case 1:
            /*time slot 1*/
            /*execute every 20ms*/
            if (1 == mod20)
                APP_write();
            break;
        default:
            break;
    }
    counter++; /*increment ISR call counter*/
}
    
```

Synch	Instruction Cycles	Stopwatch	Total Simulated
Zero	Time (µsecs)	9.980167	60.83083
Processor Frequency (MHz)		120.000000	

10ms cycle

```

D:\embedded_group\PMU\source\interrupt.c
/* ----- */
/* private variables */
/* ----- */
static uint16 counter;

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

void T2_init(void)
{
    uint16 match_value;

    ConfigIntTimer2(T2_INT_PRIOR_1 & T2_INT_ON);
    WriteTimer2(0);
    match_value = 0xE9E8; /*1ms assuming Tcy = 16.7ns*/
    OpenTimer2(T2_ON & T2_GATE_OFF & T1_IDLE_STOP &
               T2_PS_1_1 & T2_SOURCE_INT, match_value);
}

void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void)
{
    uint16 mod10, mod20;

    mod10 = counter&CYCLE10;
    mod20 = counter&CYCLE20;

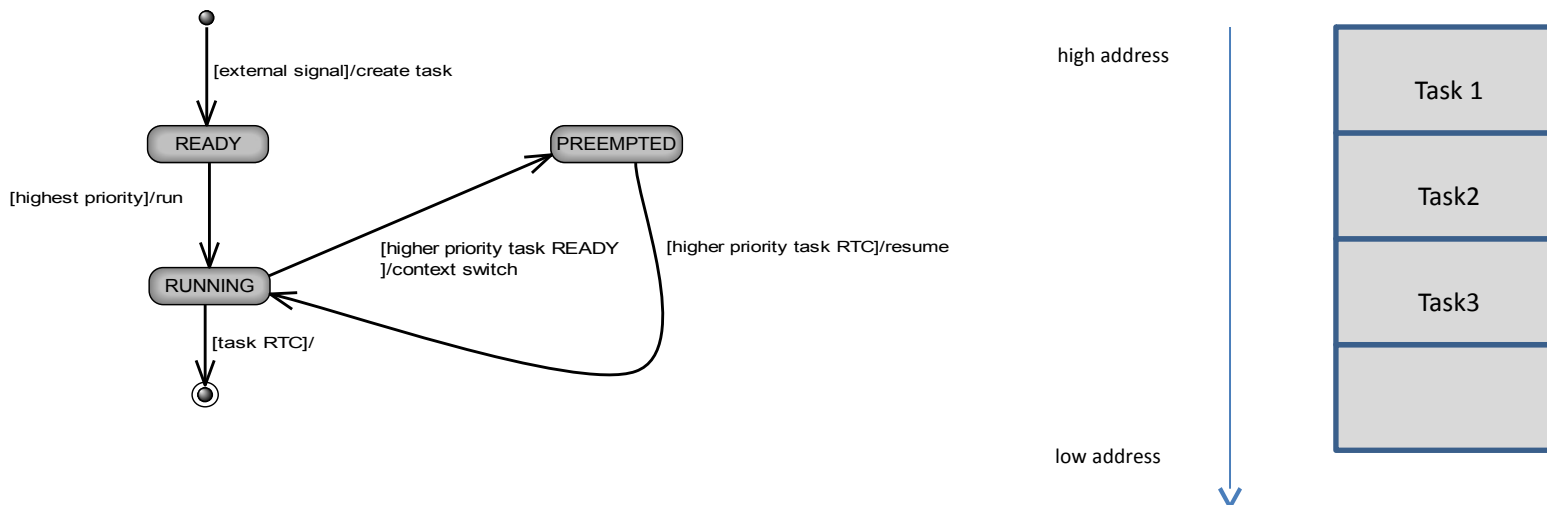
    switch(mod10)
    {
        case 0:
            /*time slot 0*/
            /*execute every 10ms*/
            APP_read();
            break;
        case 1:
            /*time slot 1*/
            /*execute every 20ms*/
            if (1 == mod20)
                APP_write();
            break;
        default:
            break;
    }
    counter++; /*increment ISR call counter*/
}
    
```

Synch	Instruction Cycles	Stopwatch	Total Simulated
Zero	Time (µsecs)	19.960333	121.762200
Processor Frequency (MHz)		120.000000	

20ms cycle

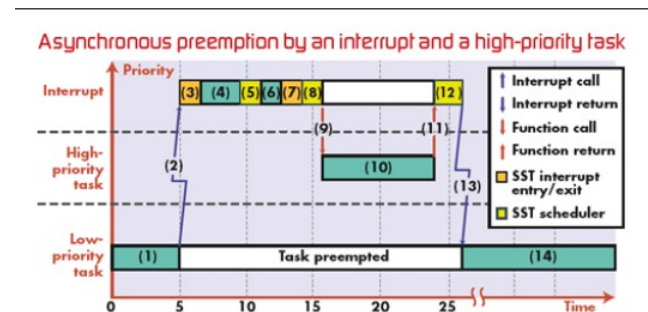
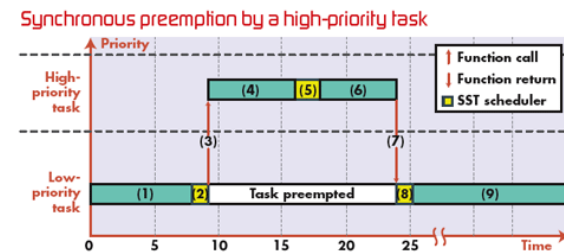
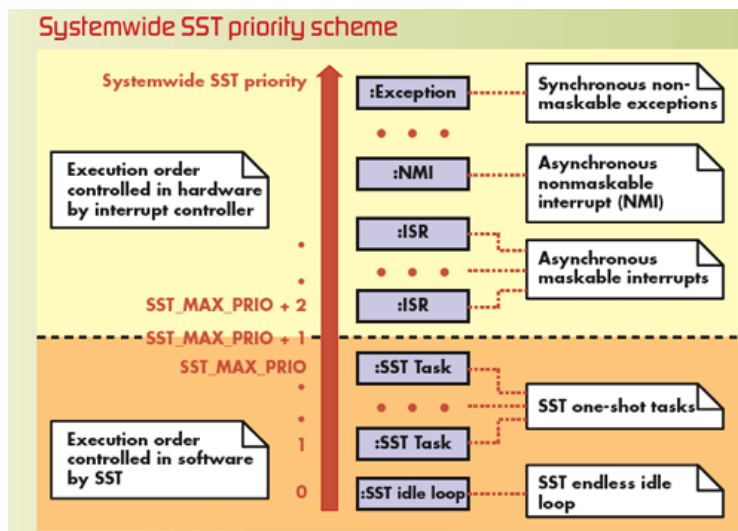
Simple Scheduler

- 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



Example

- 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>



Fault Models

- A fault model determines what possible effects of faults on the behaviour of a system model are considered
- Hardware fault models are established
- Software fault models mainly deal with corruption of data flow or control flow
- Special attention is on communication (inter-task or via networks)

Fault Detection

- 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 method for fault detection evaluation is FMEA/FMEDA (hardware integrity and functionality)
- 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).

Hardware Integrity

- 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; $\lambda_{dd} = \lambda_d \times DC/100$
 - If we can avoid a dangerous system failure by detecting dangerous component faults (λ_d) in advance we can transfer λ_d into λ_s (if the application system allows for that).
 - DC comes in four categories: no (<60%), low (60% < DC < 90%), medium (90% < DC < 99%), high (DC > 99%)

Hardware Integrity Examples

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 Welded contacts	Does not energize or de-energize Individual contacts welded	Does not energize or de-energize Individual contacts welded No positive guidance of contacts (for relays this failure is not assumed if they are built and tested according to EN 50205 or equivalent) 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 drift and oscillation
Analogue I/O		Stuck-at	DC fault model drift and oscillation	DC fault model drift and oscillation
Power supply		Stuck-at	DC fault model drift and oscillation	DC fault model drift and oscillation
Bus				
General	A.3 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 Wrong access time	All faults which affect data in the memory Wrong data or addresses 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 Dynamic cross-over for memory cells 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	No or continuous interrupts 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 Change of information caused by soft-errors for DRAM with integration 1 Mbits and higher	DC fault model for data and addresses Dynamic cross-over for memory cells No, wrong or multiple addressing 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 No transmission	All faults which affect data in the memory Wrong data or addresses Wrong transmission time Wrong transmission sequence	All faults which affect data in the memory Wrong data or addresses Wrong transmission time Wrong transmission sequence
Sensors	A.14	Stuck-at	DC fault model Drift and oscillation	DC fault model Drift and oscillation
Final elements	A.15	Stuck-at	DC fault model Drift and oscillation	DC fault model 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

Hardware Integrity Examples

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	
<p>NOTE 1 This table does not replace any of the requirements of annex C.</p> <p>NOTE 2 The requirements of annex C are relevant for the determination of diagnostic coverage.</p> <p>NOTE 3 For general notes concerning this table, see the text preceding table A.1.</p> <p>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.</p>			

Source: IEC61508-2

Hardware Integrity Examples

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) 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.
 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.

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.
 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.

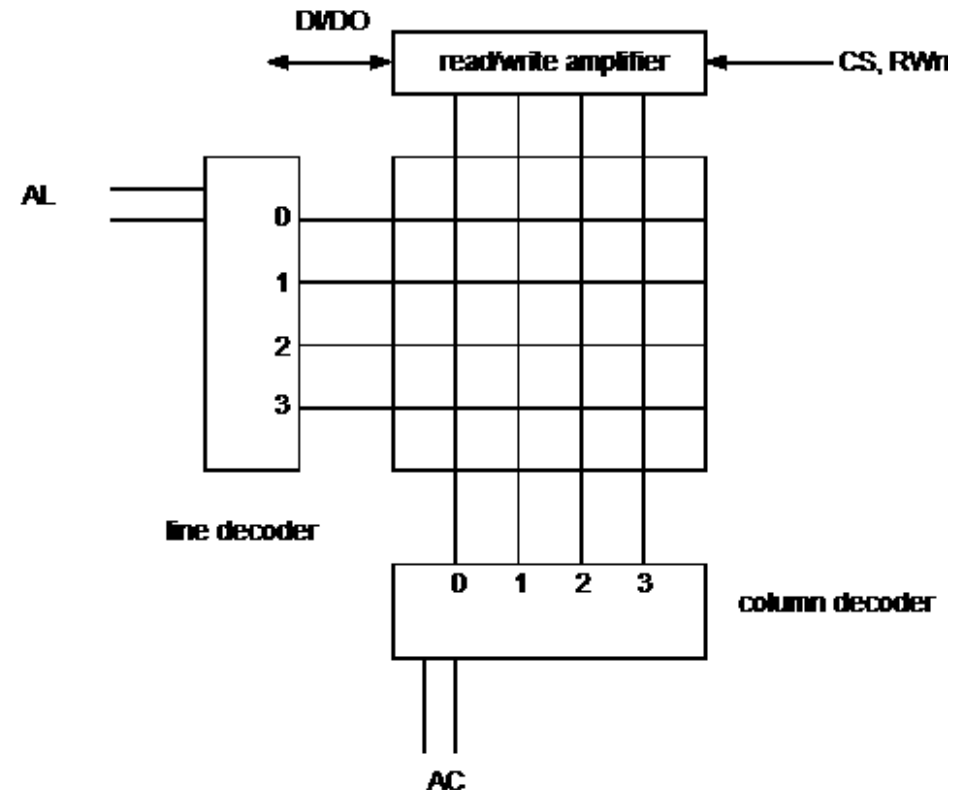
Source: IEC61508-2

Memory

- Parameter memory (non volatile)
 - EEPROM – byte wise read and write – holds e.g. configuration parameters, run-time parameters (hour meter, status)
- Program memory (non volatile)
 - Flash (NOR)– word wise read, write requires a block erase - holds executable (XIP – execute in place)
- Data memory (volatile)
 - RAM (SRAM) – word wise read and write addressable - holds data and stack

Fault Detection - Memory Model-

- 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

A

1	0	1	1
0	1	0	1
0	0	1	0
0	1	0	1

D

1	0	1	1
0	1	1	1
0	0	1	0
0	1	0	1

B

1	0	1	1
1	0	1	0
1	0	0	0
1	0	1	0

E

1	0	1	1
1	1	1	0
1	0	0	0
1	0	1	0

C

0	0	1	1
---	---	---	---

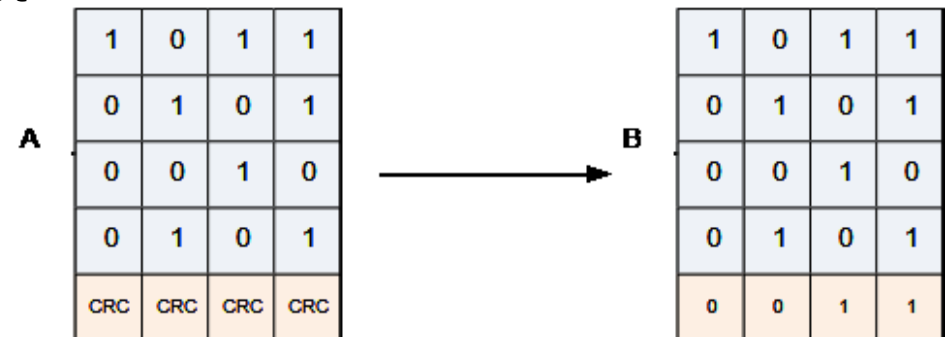
F

0	1	1	1
---	---	---	---

Fault Detection

- Non-variable Memory (program memory) II -

- Signature of one word test (CRC), based on Modulo-2 arithmetic
- Memory content is interpreted as a bit stream
- Division by a defined polynomial yields zero, $P(X) = 11001$ in this 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

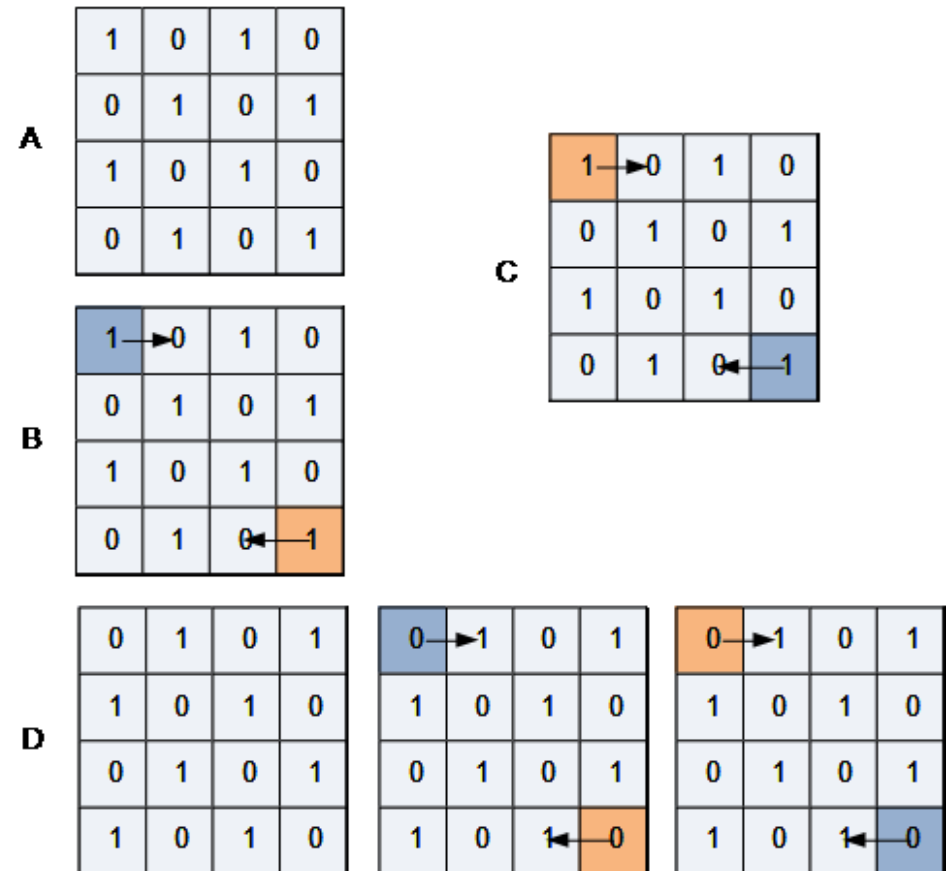
- Non-variable memory (EEPROM) -

- 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) -

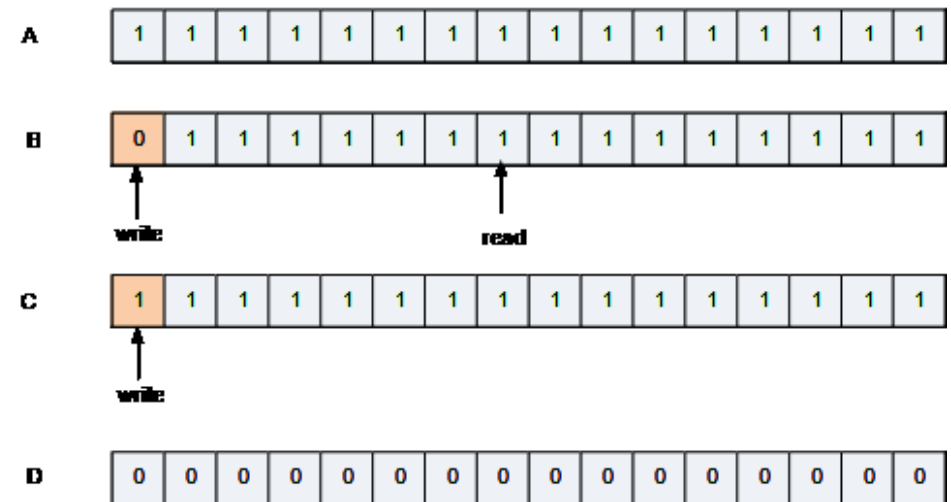
- Checkerboard test – low diagnostic coverage
- Cells are checked for correct content in pairs
- Initialization, upward test, downward test, inverse initialization, upward test, downward test -> $10 * n$ complexity (number of load store operations)
- Pairs are address inverse



Fault Detection

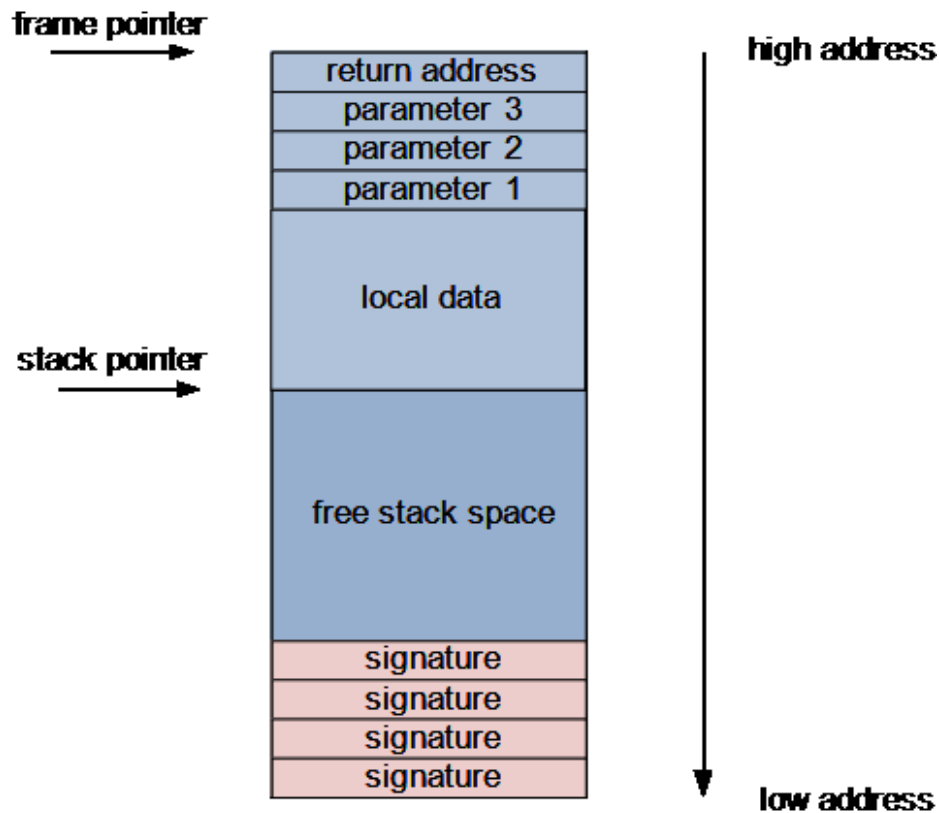
- Variable memory (SRAM) II -

- 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 -

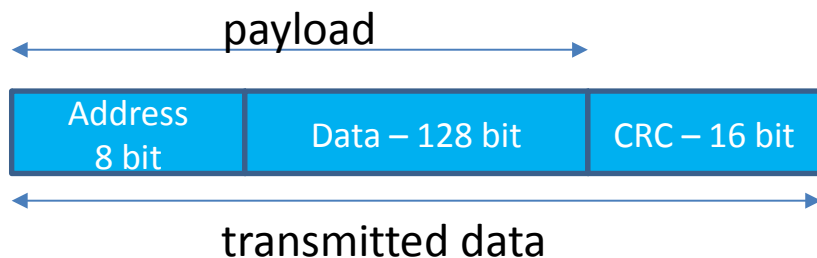
- RAM tests are destructive – therefore we need to save the original data in advance

The screenshot shows a debugger window with two panes. The left pane displays the source code of a C program named `ram_test.c`. The code defines a function `Walking_Pattern()` that tests memory. It first writes `0x5555` to memory, then reads it back to verify correctness. A red circle labeled 'B' highlights a `for` loop that reads the memory. Another red circle highlights a status bar icon. The right pane is the 'Watch' window, which shows a list of memory addresses and their values. The value `0xAAAA` at address `0x816` is circled in red, and a red arrow points to it from a box labeled 'bit flip'.

Update	Address	Symbol Name	Value
	0800	mem	
	0800	[0]	0x5555
	0802	[1]	0xAAAA
	0804	[2]	0xAAAA
	0806	[3]	0xAAAA
	0808	[4]	0xAAAA
	080A	[5]	0xAAAA
	080C	[6]	0xAAAA
	080E	[7]	0xAAAA
	0810	[8]	0xAAAA
	0812	[9]	0xAAAA
	0814	[10]	0xAAAA
	0816	[11]	0xAAAA
	0818	[12]	0xAAAA
	081A	[13]	0xAAAA
	081C	[14]	0xAAAA
	081E	[15]	0xAAAA
	0820	[16]	0xAAAA
	0822	[17]	0xAAAA
	0824	[18]	0xAAAA
	0826	[19]	0xAAAA
	0828	[20]	0xAAAA
	082A	[21]	0xAAAA
	082C	[22]	0xAAAA
	082E	[23]	0xAAAA
	0830	[24]	0xAAAA
	0832	[25]	0xAAAA
	0834	[26]	0xAAAA
	0836	[27]	0xAAAA
	0838	[28]	0xAAAA
	083A	[29]	0xAAAA
	083C	[30]	0xAAAA
	083E	[31]	0xAAAA
	0840	[32]	0xAAAA
	0842	[33]	0xAAAA
	0844	[34]	0xAAAA

Communication - Error Detection -

- We usually use standard protocols to transmit data. Correctness is guaranteed by error detection mechanisms (e.g. parity, CRC)
- Sometimes error detection capability not sufficient
 - Hamming distance of n : $n-1$ bit errors can be detected.
 - Residual error: If we do know the Hamming distance and do know the bit error rate (bit flips are statistically independent) we can calculate a residual error.
 - CRC: an additional piece of data is added to the existing bit stream. The additional piece of data allows error detection

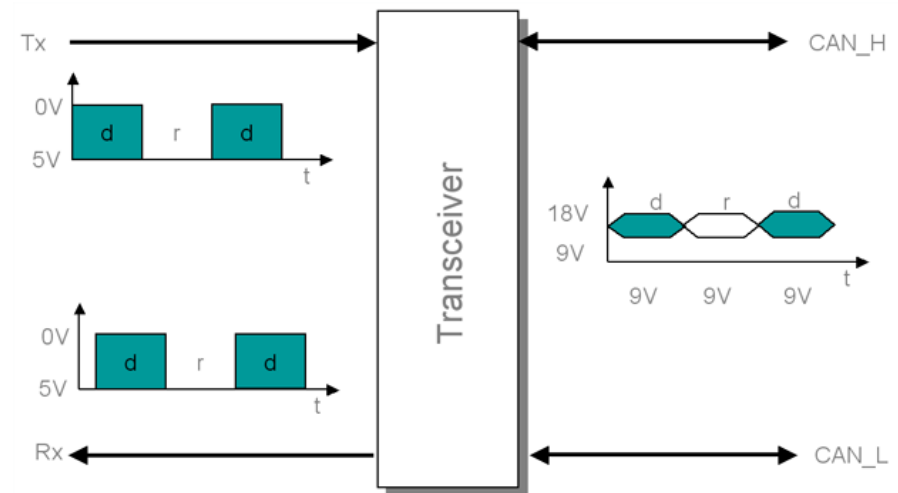


Probability of bit failures p	Transmission medium
$> 10^{-3}$	Transmission path
10^{-4}	Unscreened data line
10^{-5}	Screened twisted-pair telephone circuit
$10^{-6} - 10^{-7}$	Digital telephone circuit (ISDN)
10^{-9}	Coaxial cable in local defined application
10^{-12}	Fibre optic cable

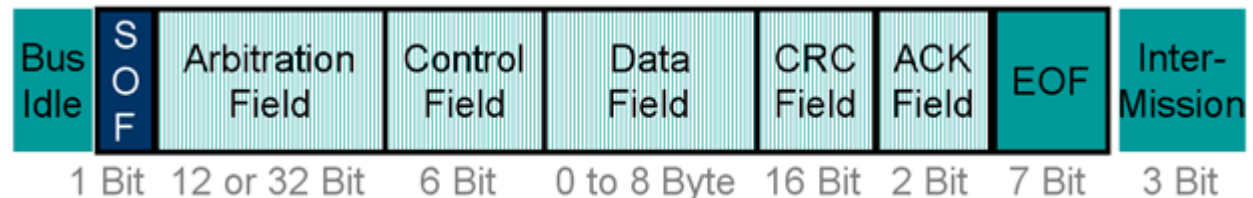
Source:
Börcsök, HIMA

Communication - CAN -

- CAN: Controller Area Network, ISO 11898 (PHY, DLL)
- Protocol controller available as peripheral of embedded processors, line driver external (creates differential signals, adds protection circuits)
- Serial protocol, up to 1 Mbit/s
- Bit-wise arbitration
- Error detection

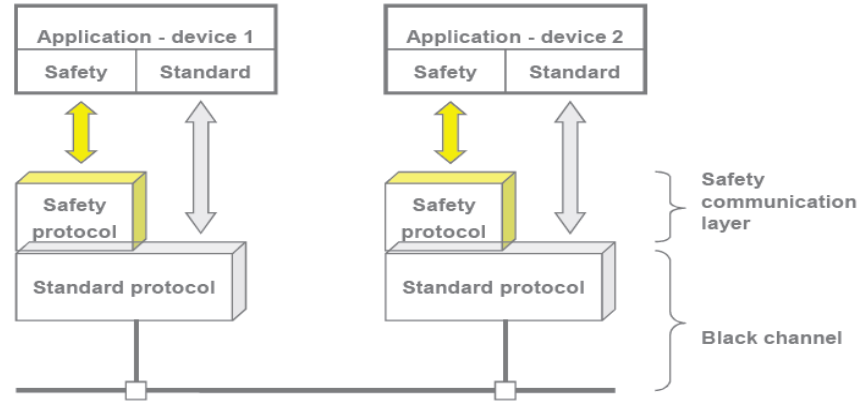


Source:
Softing



Black Channel

Source:
 MESCO Engineering,
 Forum Funktionale
 Sicherheit 2013



Error	Deterministic remedial measures							
	Sequence number	Time stamp	Time expectation (Watchdog)	Connection authentication	Feedback message (Echo)	Data Integrity assurance (CRC)	Redundancy with cross checking	Different data Integrity assurance systems
Unintended repetition	X	X					X	
Loss	X				X		X	
Insertion	X			X	X		X	
Incorrect sequence	X	X					X	
Corruption						X		
Unacceptable delay		X	X					
Masquerade				X	X			X
Adressing				X				

Proven in use Software (FAQs – www.iec.ch)

D11) Can an E/E/PE safety-related system contain hardware and/or software that was not produced according to IEC 61508, and still comply with the standard (proven in use)?

It may be possible to use a *proven in use* argument as an alternative to meeting the design requirements for dealing with systematic failure causes in IEC 61508, including hardware and software. But it is essential to note that proven in use cannot be used as an alternative to meeting the requirements for:

- architectural constraints on hardware safety integrity (see 7.4.2.1 of IEC 61508-2);
- the quantification of dangerous failures of the [safety function](#) due to random hardware faults (see 7.4.3.2 of IEC 61508-2); and
- system behaviour on detection of faults (see 7.4.6 of IEC 61508-2).

See 7.4.2.2 of IEC 61508-2 for a summary of design requirements, including references to more detailed systematic hardware requirements in the standard.

A proven in use claim relies on the availability of historical data for both random hardware and systematic failures, and on analytical techniques and testing if the previous conditions of use of the subsystem differ in any way from those which will be experienced in the [E/E/PE safety-related system](#). 7.4.7.6 of IEC 61508-2 requires that:

- the previous conditions of use of the subsystem are the same as, or sufficiently close to, those which will be experienced in the E/E/PE safety-related system (see 7.4.7.7 of IEC 61508-2);
- if the above conditions of use differ in any way, a demonstration is necessary (using a combination of appropriate analytical techniques and testing) that the likelihood of unrevealed systematic faults is low enough to achieve the required [safety integrity level](#) of the safety functions which use the subsystem (see 7.4.7.8 of IEC 61508-2);
- the claimed failure rates have sufficient statistical basis (see 7.4.7.9 of IEC 61508-2);
- failure data collection is adequate (see 7.4.7.10 of IEC 61508-2);
- evidence is assessed taking into account the complexity of the subsystem, the contribution made by the subsystem to the risk reduction, the consequences associated with a failure of the subsystem, and the novelty of design (see 7.4.7.11 of IEC 61508-2); and
- the application of the proven in use subsystem is restricted to those functions and interfaces of the subsystem that meet the relevant requirements (see 7.4.7.12 of IEC 61508-2).

7.4.2.11 of IEC 61508-3 allows the use of standard or previously developed software without the availability of historical data but with the emphasis on analysis and testing. This concept should be distinguished from the proven in use concept described above.