Chapter 3
Hardware Considerations

What are you about to learn? 2

3.1 Input/Output Architecture 3

3.2 Sampling and Polling 10

3.3 Interrupts 15

3.4 Sensors and Actuators 24

Points to Remember 30

Objectives

What are you about to learn?

Knowledge Objectives

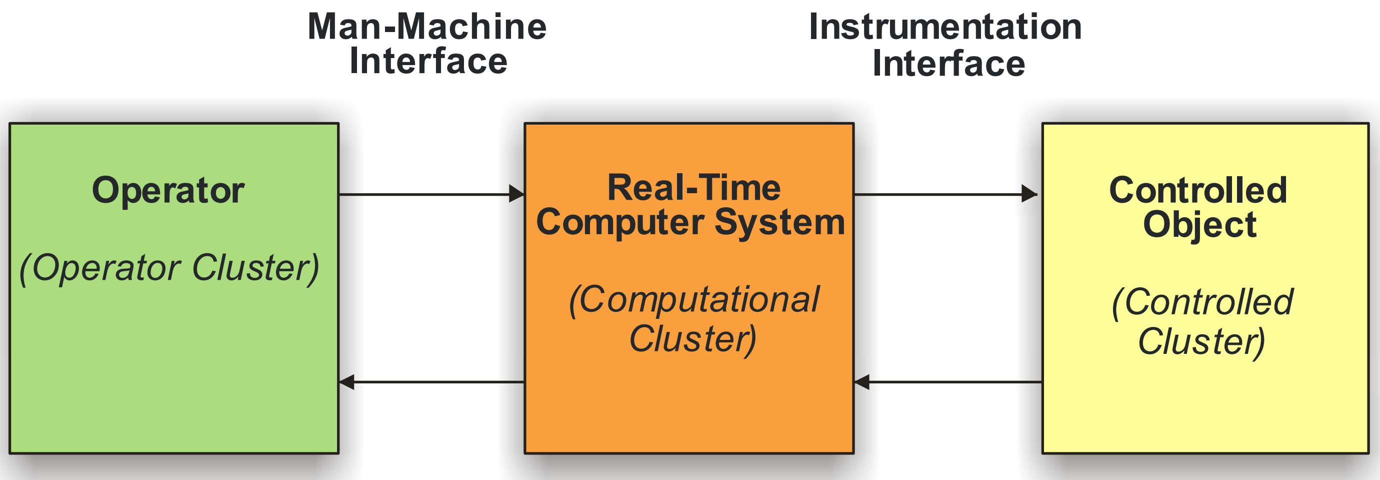
- Understand the different ways to connect computer systems to the real world.
- Understand the difference between sampling and polling.
- Understand how interrupts work directly on the hardware, and in the RMOS operating system.
- Know about typical sensors and actuators for real-time systems.

Skill Objectives

- Ability to write a small C program to control parallel digital I/O.
- Ability to write and install a simple interrupt service routine for a plain bare bones computer system, and for a system with the RMOS operating system installed.
- Ability to connect two real-time computer systems via a real-time communication system (CAN).

3.1 Input/Output Architecture

The real-time computer system interfaces to the controlled object via the instrumentation interface.
The real-time computer system interfaces to the operator via the man-machine interface.



The computer system hardware interfaces to the other objects via input/output devices.

Functional requirements for input/output devices are:

- Buffering of input and output data
- Conversion and transformation of data (e.g., serial/parallel, analogue, digital)
- Generation of control- and handshaking signals
- Picking up and generating interrupt signals

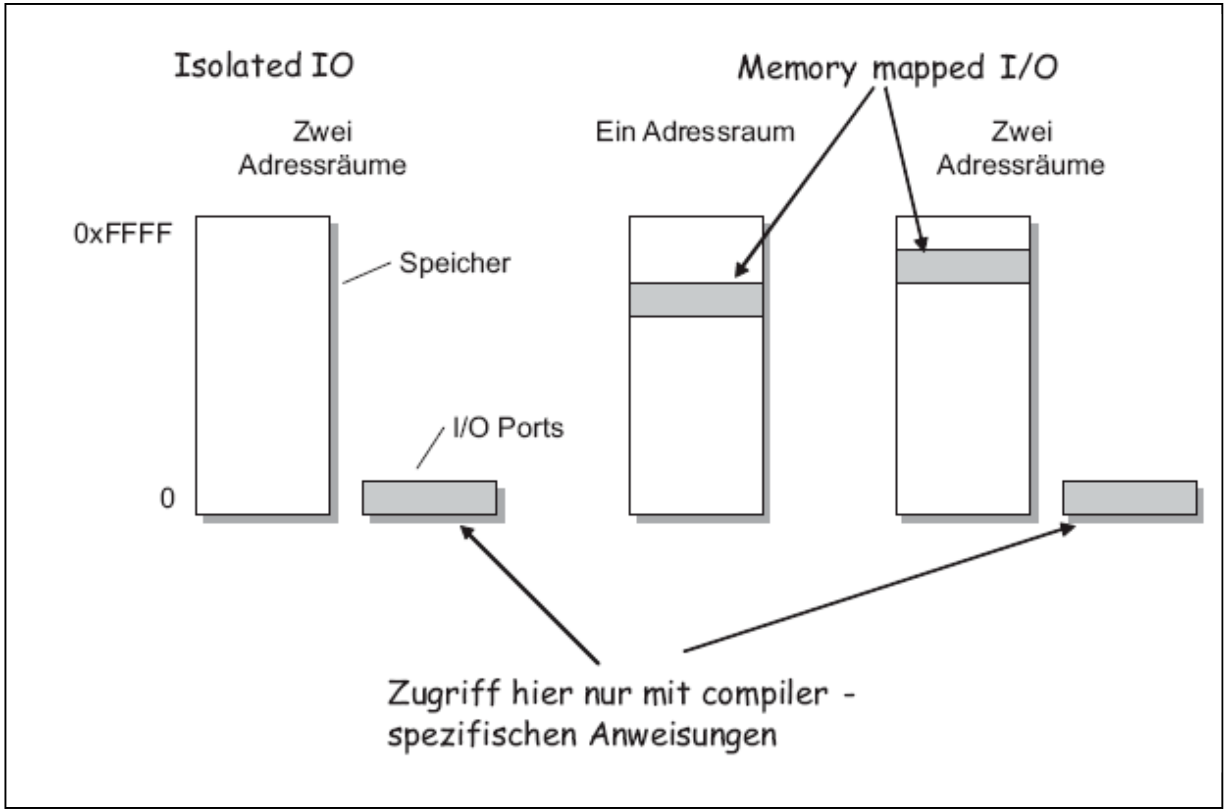
3.2 Sampling and Polling

Input/output devices are

- Integrated in a microcontroller
- Placed on separate circuits (interface controller)

Input/output devices are connected to the computer system core via

- Memory mapped I/O
- Isolated I/O
- Direct Memory Access I/O (DMA)



We classify interfaces as

- Simple interfaces (a single register for input or output)
- Complex interfaces (input register, output register, status register, control register)
- Intelligent interfaces (with own microcomputer, programmable locally)

3.2 Sampling and Polling

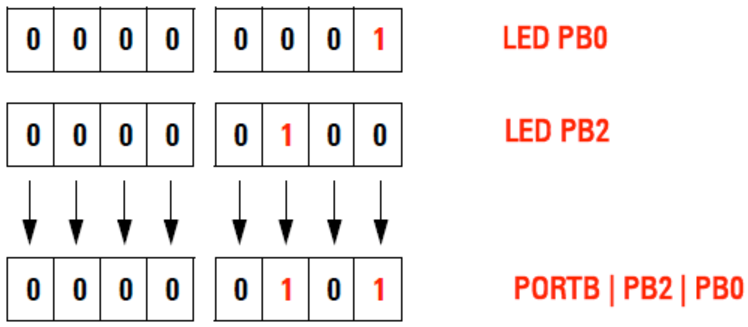
Interface Data Transfer Rates

Interface	Data transfer rates
System bus	> 1 GBytes/sec
Byte move in memory	> 1 GByte/sec
Ethernet	> 10 MBytes/sec
ISDN phone line	8 kBytes/sec
Serial line (RS232)	1 kBytes/sec
Printer	100 bytes/sec
Keyboard	4 bytes/sec

Memory Mapped I/O and Bit Manipulations in the C Language

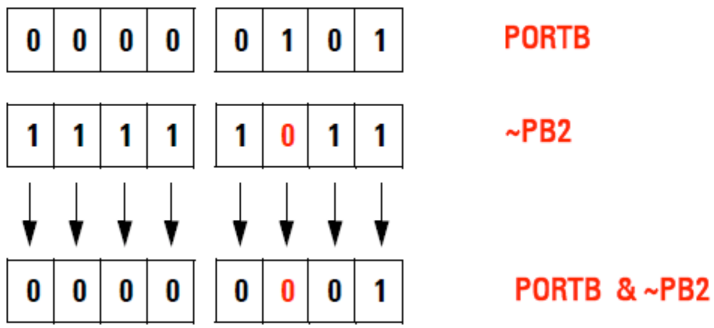
Example 1: setting bits on Port B of the Dragon 12 board (PB0 and PB2 are called “mask”):

```
enum leds {PB0 = 0x01, PB1 = 0x02, PB2 = 0x04, PB3 = 0x08,
PB4 = 0x10, PB5 = 0x20, PB6 = 0x40, PB7 = 0x80};

PORTB = PORTB | PB0 | PB2;
```



Example 2: resetting bits on Port B of the Dragon 12 board (PB2 is called a "mask"):

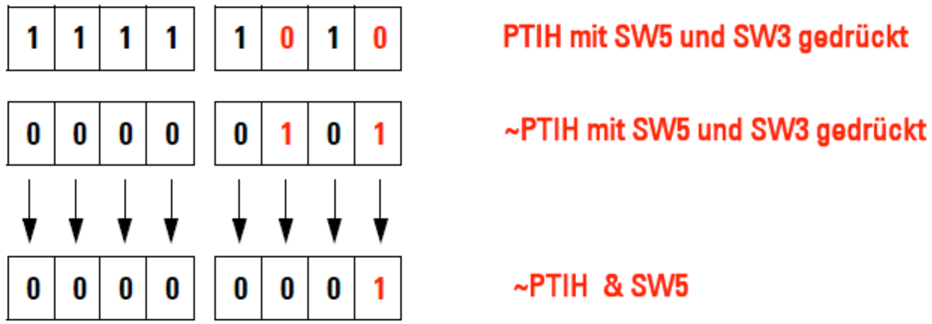
```
enum leds {PB0 = 0x01, PB1 = 0x02, PB2 = 0x04, PB3 = 0x08,
PB4 = 0x10, PB5 = 0x20, PB6 = 0x40, PB7 = 0x80};

PORTB = PORTB & ~PB2;
```



3.2 Sampling and Polling

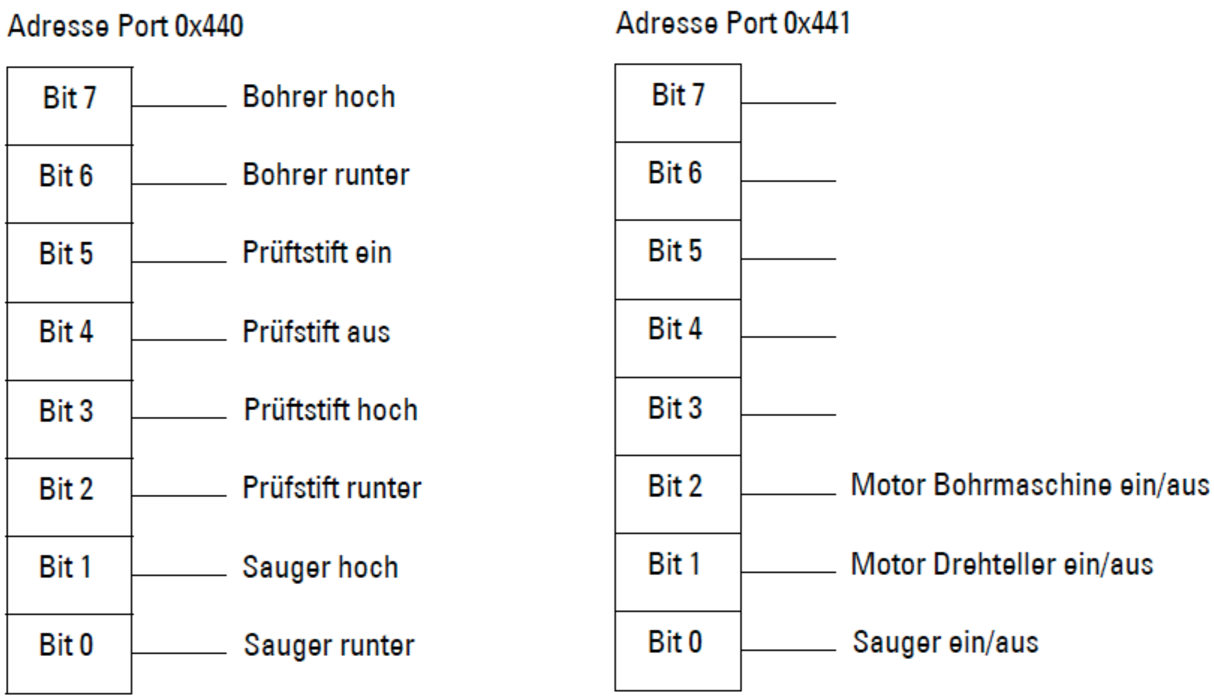
Example 3: testing bits on Port H of the Dragon 12 board (0x20 is called a “mask”):

```
enum switches {SW5 = 0x01, SW4 = 0x02, SW3 = 0x04, SW2 = 0x08};

if (~PTIH & SW5) { … /* SW5 pressed */
```



Isolated I/O on SICOMP Industrial PC

Separate I/O address space, in the following example ports 0x440 to 0x443:



The following code fragment shows how to control these ports via a utility routine:

3.2 Sampling and Polling

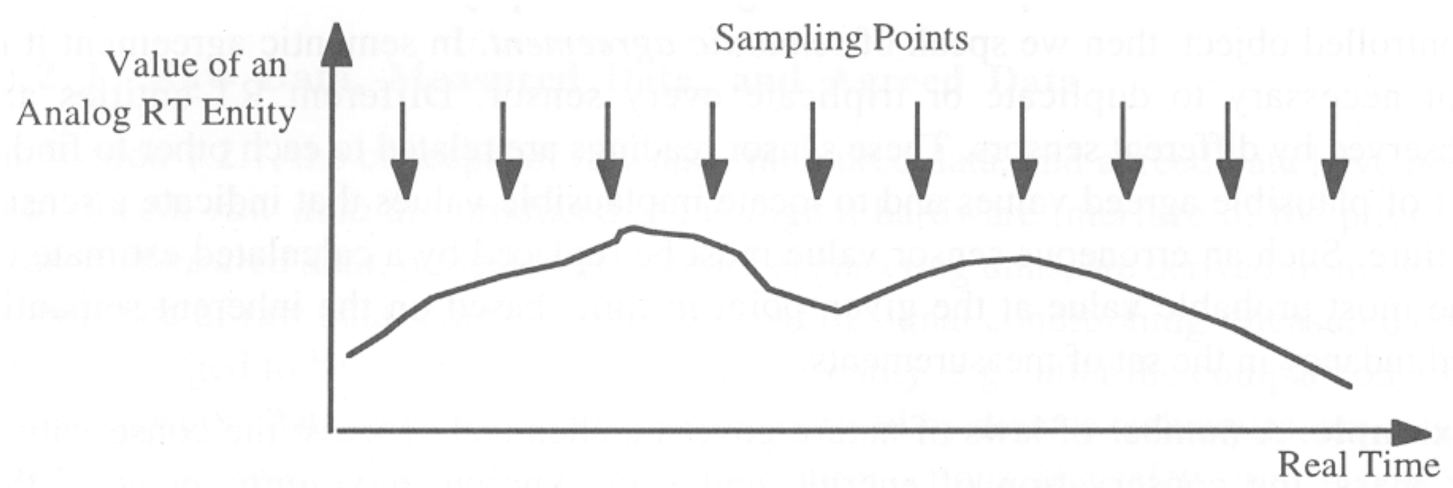
```
...
/*---------------------------------------------------------------*/
/* Digital Input/Output 24V AMS-P218 */
/*---------------------------------------------------------------*/
#define P218_BASE 0x0440 /* base address */
#define P218_CHN0 0x0440 /* channel group 0 (output) */
#define P218_CHN1 0x0441 /* channel group 1 (output) */

unsigned char portWrite(unsigned int io_address,
                        unsigned char outputValue,
                        unsigned char mask) {
...
unsigned char outv, x;
outv = outputValue & mask ;
x = inbyte (io_address); /* Requires special function, can only be
                            written in assembly language */
x = ( x & ~mask ) | outv ;
outbyte (io_address , x ) /* Requires special function, can only be
                             written in assembly language */
ret = inbyte (io_address) ;
return (ret);
}
```

3.2 Sampling and Polling

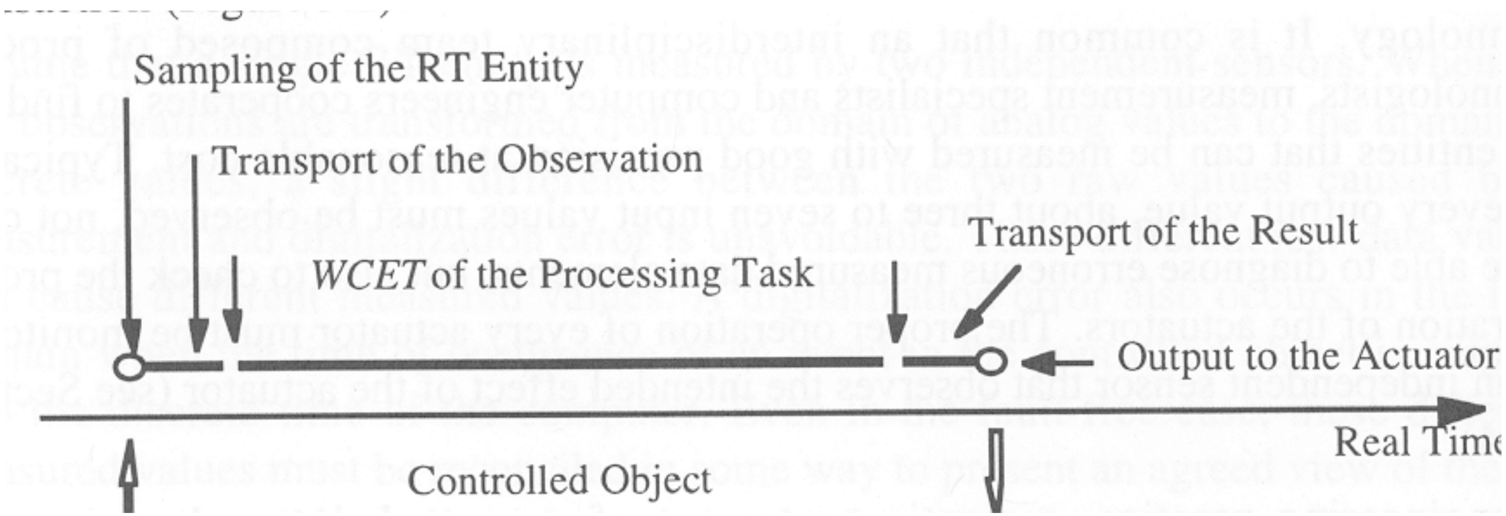
In sampling, the state of an RT entity is periodically interrogated by the computer system at points in time called sampling points. The constant interval between sampling points is called the sampling interval.

Sampling of Analog Values

The most recent current value of an analog RT entity is observed at a moment determined by the computer system.



In a time-triggered architecture, the sampling points can be coordinated with the transmission schedule to generate phased-aligned transaction. This ensures the shortest possible response time.



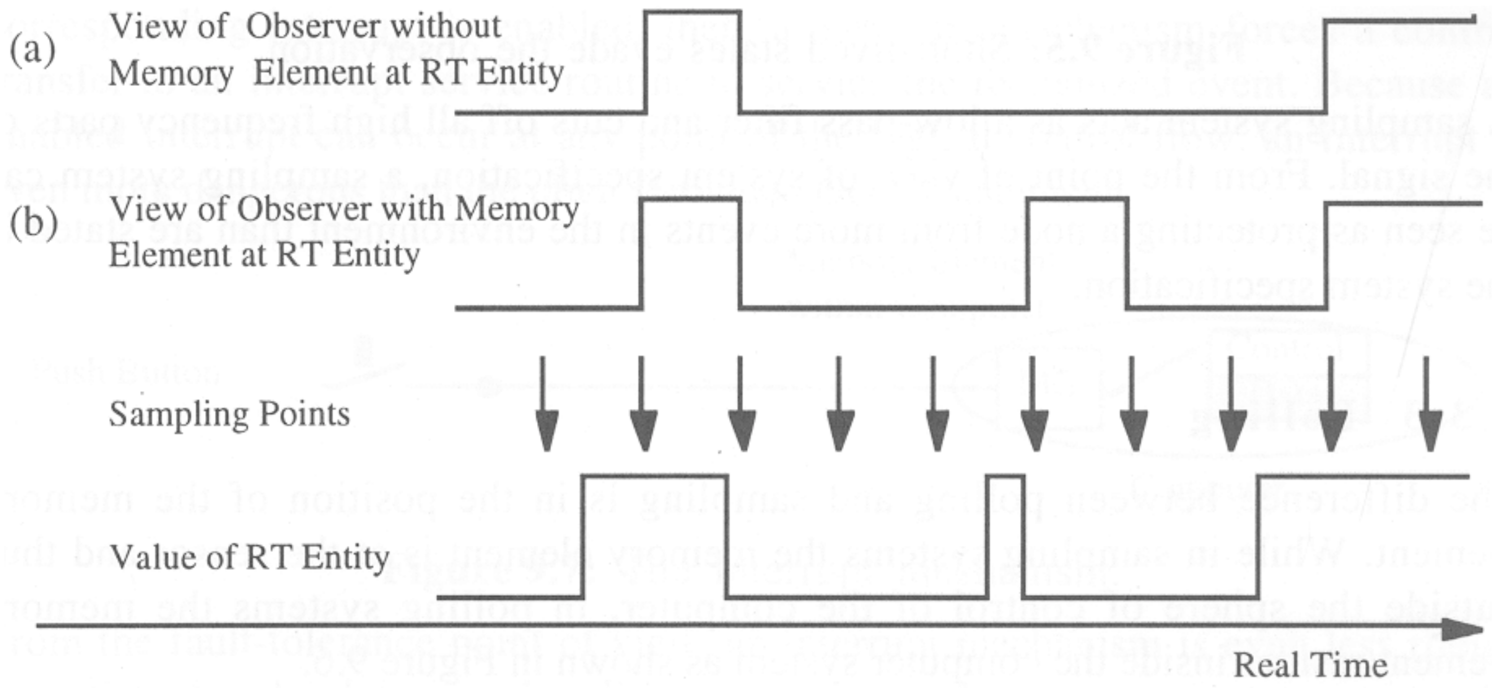
Sampling of Digital Values

When sampling a digital value, the current state and the temporal position of the most recent state change are of interest.

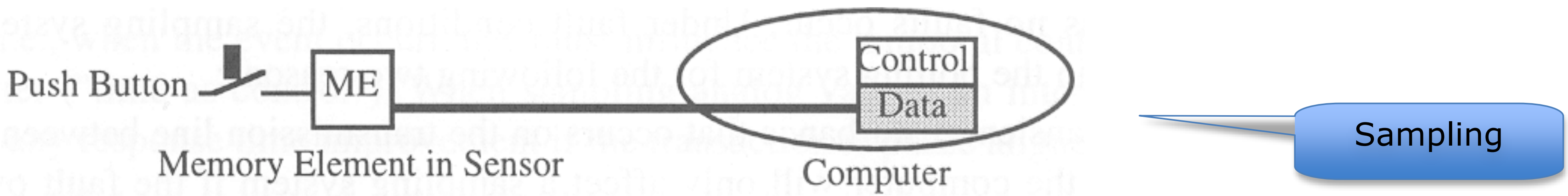
The current state is observed at the sampling point.

The temporal position of the most recent state change can only be inferred by comparing the current observation with the most recent observation.

This can lead to problems when the state of the RT entity can change more than once during the sampling interval.

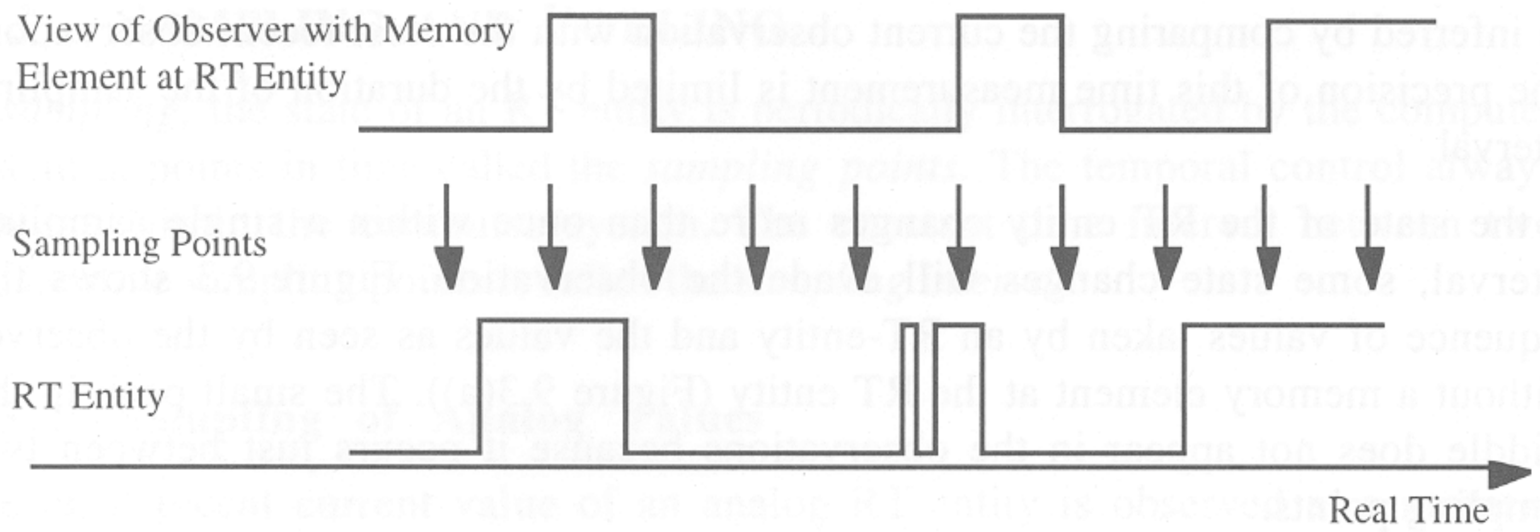


If every event is significant, a memory element must be inserted directly at the RT entity. This memory element stores any state change until the next sampling point.



3.2 Sampling and Polling

This does not solve all problems:



A sampling system acts as a low pass filter and cuts off all high frequency parts of the signal.
A sampling system can protect a computer system from more events than are stated in the specification.

Polling

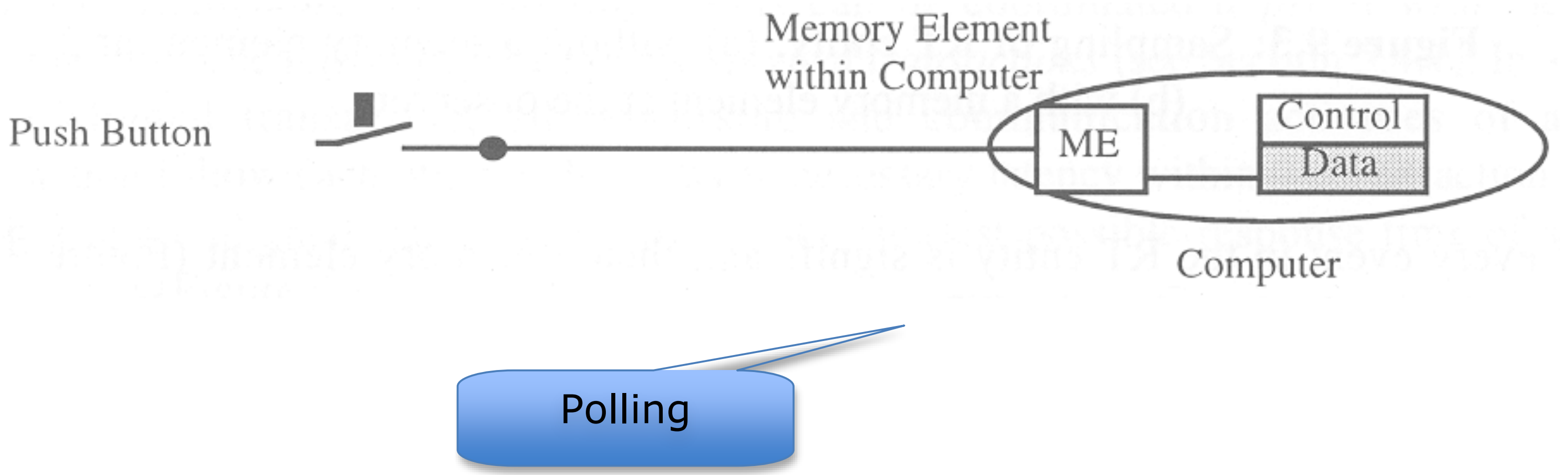
In a sampling system the memory element is placed at the sensor.
In a polling system the memory element is placed at the computer.

Functionally this makes no difference as long as there are no faults.

3.2 Sampling and Polling

We consider two faults:

- A transient disturbance on the transmission line between the sensor and the computer has no effect on the measurement with a sampling system, unless it occurs directly at the sampling point (low probability of occurrence). In a polling system every single fault will be stored in the memory element.
- In case of a node shutdown and restart, all data from volatile memory is lost in a polling system. In a sampling system the stored event in the sensor can survive a computer reset.

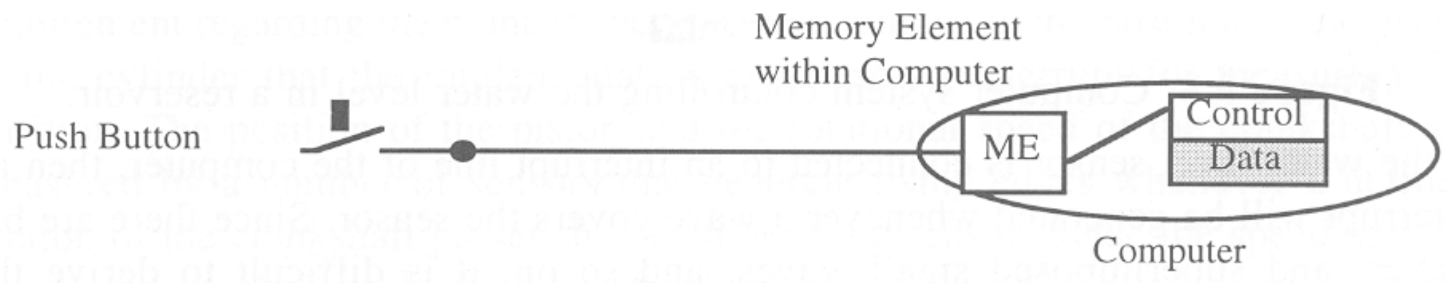


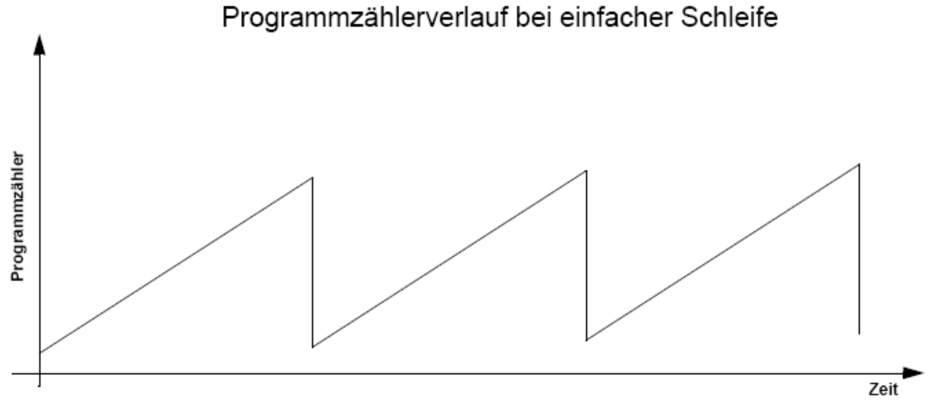
3.3 Interrupts

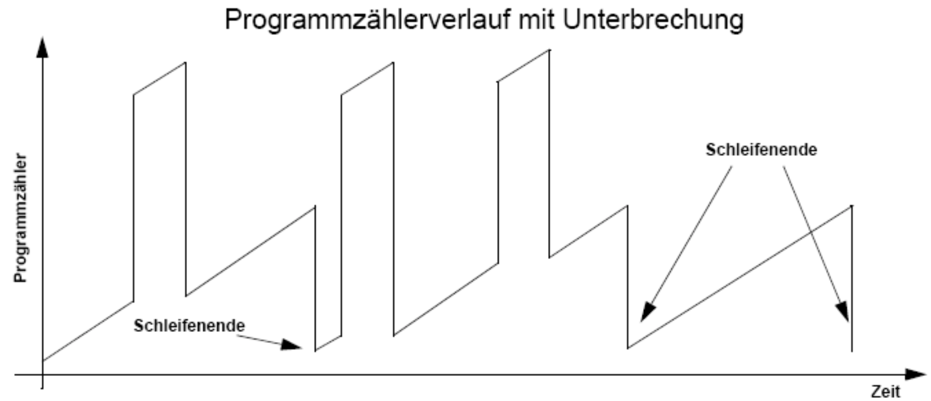
The interrupt mechanism permits a device outside the sphere of control of the computer to make the computer execute a specific "Jump Subroutine" instruction any time that interrupt is enabled.

The flow of control is thus determined by an external device.

The target of the "Jump Subroutine" is called an Interrupt Service Routine (ISR). After the ISR has been executed, control is given back to the regular program flow.



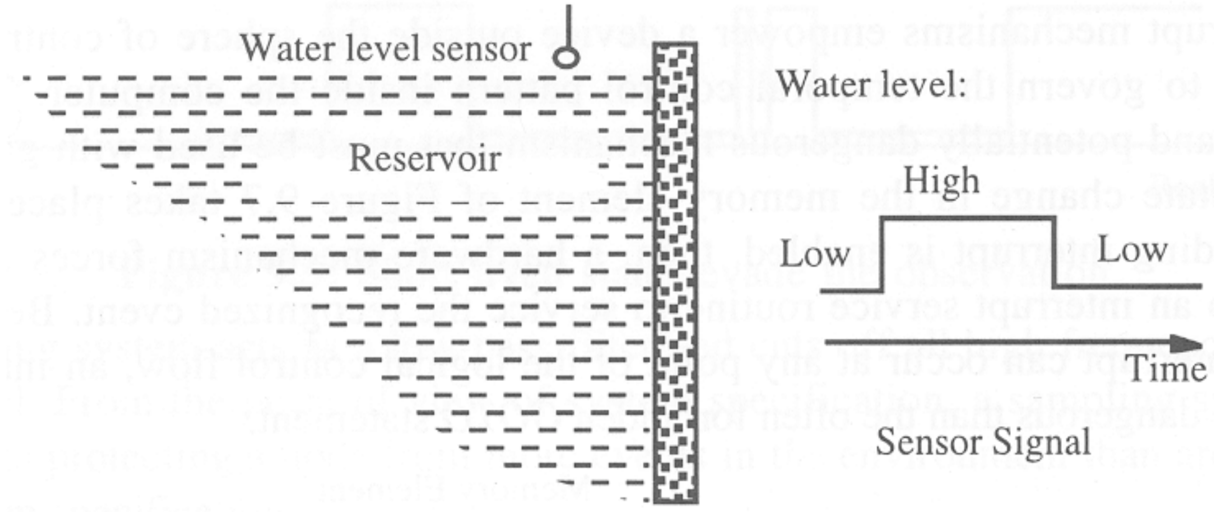




When Are Interrupts Needed?

Interrupts are needed when external events require short reaction times that cannot be efficiently met with a sampling implementation.
An alternative to interrupts are trigger tasks. The trigger task itself requires some time to run. As a rule of thumb, if the WCET of the trigger task is larger than 10 times the required response time, interrupts should be avoided.

Example 1:



An overflow valve is controlled by a computer and is to be opened when the water level reaches the high-level mark. When the sensor is connected to an interrupt line of the computer, small waves can produce high frequency interrupts.

Attaching the sensor to a digital input line, and sampling and filtering the values, would lead to a more robust behaviour.

Example 2: Bouncing switch

On our lab boards we have switches connected to digital input lines. When these switches are closed or opened, they “bounce”, i.e. oscillate for some time (typically a few ms up to 50 ms) rapidly between the “open” and “closed” state.

It would not be wise to connect such switches directly to an interrupt line.

Example: Implementing a 10 ms timer tick

This is the interrupt service routine for the ticker:

```
/*
 * Interrupt service routine for timer channel 4.
 *
 */
interrupt 12 void isr_tc4(void) {

 /* increment timer count register (16 bits) */
 /* current count + increment = new count */
 TC4 = TC4 + TENMS;
 if ((void *) tickerFunctionPointer > NULL) {
    tickerFunctionPointer();
 }
 /* clear the interrupt: write a 1 to bit 4 */
 TFLG1 = TIMER_CH4;
}
```

„interrupt“ is a keyword that instructs the C-compiler to generate an RTI instead of an RTS instruction at the end of the function.

The magic „12“ comes from the position of the timer channel interrupt vector in the controllers interrupt vector table.

3.4 Sensors and Actuators

This is the routine to initialize the hardware for the ticker:

```
/*
 * Initialize the timer 4 hardware and interrupt
 *
 */
void initTicker(void) {

  tickerFunctionPointer = 0;
  ticks = 0;
  /* Timer master ON switch */
  TSCR1 = TIMER_ON;

  /* Set channel 4 in "output compare" mode */
  TIOS = TIOS | TIMER_CH4; /* bit 4 corresponds to channel 4 */

  /* Enable channel 4 interrupt; bit 4 corresponds to channel 4 */
  TIE = TIE | TIMER_CH4;

  /* Set timer prescaler (bus clock : prescale factor) */
  /* In our case: divide by 2^7 = 128. This gives a timer */
  /* driver frequency of 187500 Hz or 5.3333 us time interval */
  TSCR2 = (TSCR2 & 0xf8) | 0x07;

  /* switch timer on */
  TCTL1 = (TCTL1 & ~TCTL1_CH4) | TCTL1_CH4_DISCONNECT;
}
```

Don’t worry if you do not completely understand this example. You will learn how the timer module operates in the CA3 lecture later on. This shall just show you how you can access all these marvelous registers with a few lines of C-code, and how to install a call-back routine.

This is the main program:

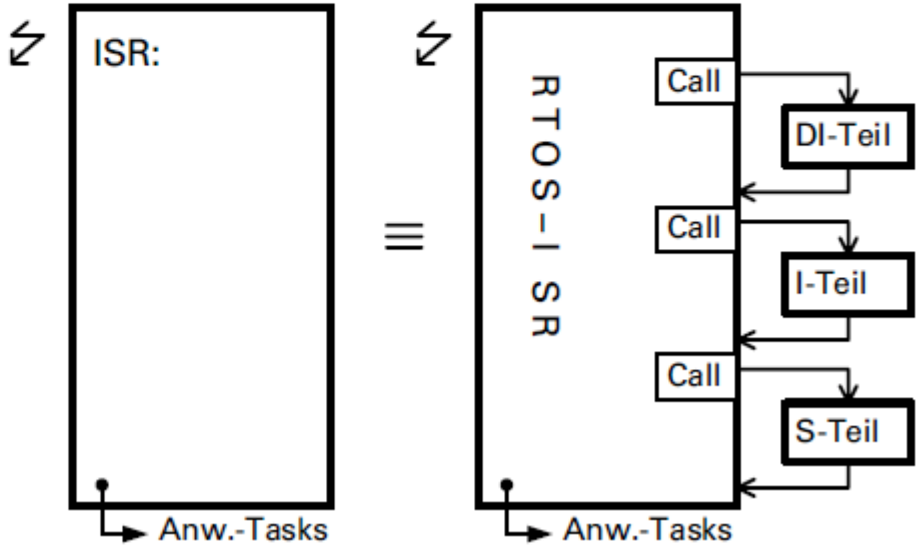
```
#include <hidef.h>            /* some macros and defines        */
#include <mc9s12dp256.h>      /* processor specific definitions */
#include "ticker.h"

void tickRoutine(void);

#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"

static unsigned long cc = 0;

void main(void) {

  EnableInterrupts;  /* We need this for the debugger */

  DDRB = 0xff;       /* Set PORTB to output */

  PORTB = 0x00;      /* Turn off all LEDs initially */

    initTicker();
    registerTickRoutine(&tickRoutine);  /* Register callback routine */

    for(;;) {
      /* Just go crazy */
    }
}
```

This is the callback routine:

```
void tickRoutine(void) {
	++cc;
	if (cc > 100) {
	 cc = 0;
	 PORTB = ~PORTB & 0x01;
	}

}
```

Example: Implementing interrupts on RMOS and the SICOMP IPC

The RMOS operating systems provides a wrapper for interrupt service routines:



DI part: (disabled interrupts)
I-part: interruptible by higher priority interrupts
S-part: interruptible by any other interrupts, but runs before any task

The DI part is usually not programmed by the application programmer. It must be very short and is provided by the operating system itself.

3.4 Sensors and Actuators

The I-part is a simple function:

```
/*-----------------------------------------------------------
 * ISR, I-part
 *-----------------------------------------------------------
 */
int i_part (void) {
  outbyte (...);    /* do something here, delete interrupt flag */
  return (1) ;      /* "1" : continue with S-part                */
}
```

The S-part is a simple function, too. It is only executed when there are no more pending I-parts:

```
/*-------------------------------------------------------------- */
/* ISR, S-part                                        */
/*-------------------------------------------------------------- */
void s_part (void) {
   RmSetFlag (EFG_ISR, FLAG_0);/* Event-Flag setzen */
   i++;
   s_msg.m1 = i;
   RmSendMail (RM_CONTINUE, prio, MBOX, &s_msg);
   /* Send message */
 return;}
```

3.4 Sensors and Actuators

All S-parts are being executed before any task resumes operation. The I- and S-parts need to be installed and de-installed:

```
/*------------------------------------------------------------------*/
/* Install I- and S-parts                                           */
/*------------------------------------------------------------------*/

...
sts = RmSetIntISHandler(
            IRQ(16),            /* SMP interrupt 0 = HW interrupt no. 16*/
            (rmfarproc) i_teil, /* address for I-part */
            (rmfarproc) s_teil) /* address for S-part */
...
clrint(IRQ(16)); /* delete SMP interrupt */
eint(IRQ(16));   /* enable interrupt */
...
sts = RmSetIntDefHandler(IRQ(16))       /* deinstall interrupt handler*/
```

The installation and de-installation should be done at program startup and program shut-down, respectively.

3.4 Sensors and Actuators

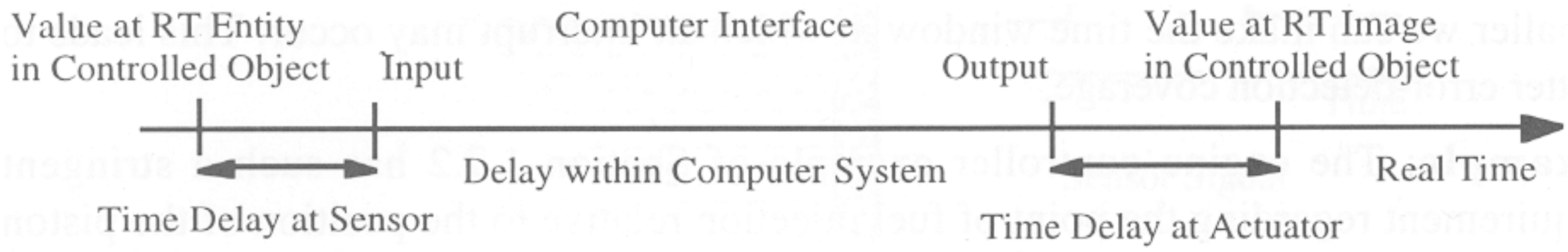
Controlled objects contain transducers (sensors and actors) that measure RT entities, or accept RT images from the controlling computer.

Analog input/output

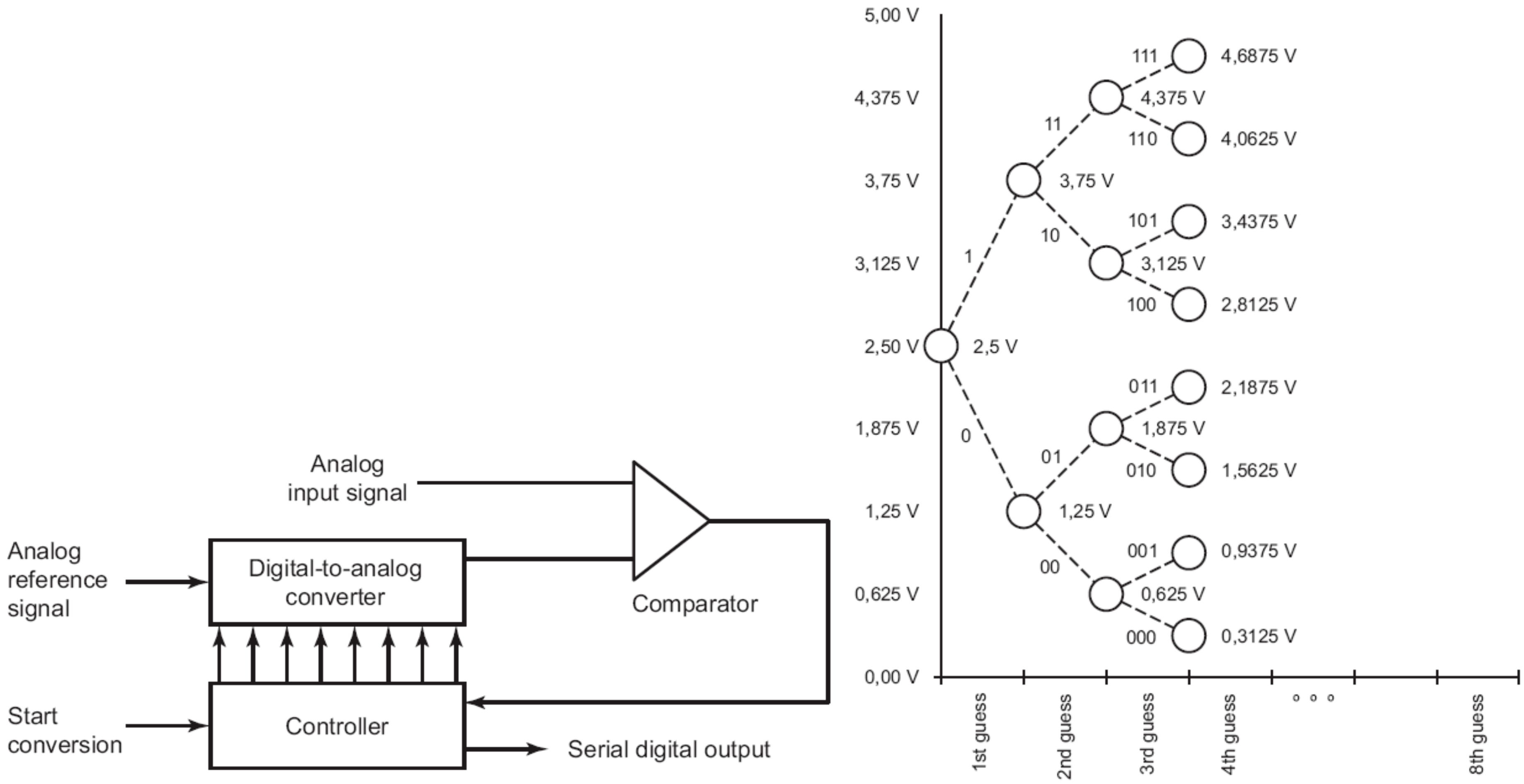
Many RT entities are observed by sensors that produce analog values. A typical industry standard for analog value encoding is the 4-20 mA range current encoding, 4 mA meaning 0% and 20 mA meaning 100% of the value. This scheme permits to detect broken wires (i.e. a current of 0 mA).

The accuracy of any analog control signal is limited by the electrical noise level, to about 0.1%, corresponding to an A/D converter resolution of about 11 bit (2^{-11}). Many microcontrollers have 10 bit A/D converters, and a 16-bit word is more than sufficient to encode analog values.

The time interval required for the transducer to present the value of an RT entity at the sensor/computer interface influences the temporal accuracy interval for that RT entity and its corresponding image. The smaller the time interval for the sensor, the more time remains for the computer to create a corresponding output action.



A commonly used strategy for fast A/D conversion is the sucessive approximation method.



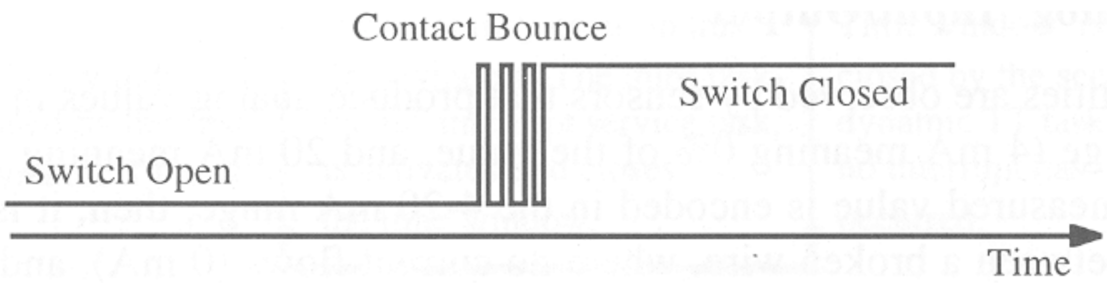
This type of A/D converter can be found on the 68HCS12 microcontroller in the lab.

Digital Input/Output

A digital I/O signal transits between the two states “0” and “1” or “TRUE” and “FALSE”. Typical representations of “0” are 0 Volts, or open contact. Typical representations for “1” are 3.3, 5.0, or 24 Volts, or a closed contact.

In some applications not just the state itself is important, but also the duration of that state or the moment when a state transition occurs.

Mechanical contacts do not reach their new state immediately but only after a number of random oscillations (contact bounce) caused by the mechanical vibrations of the switch contact.

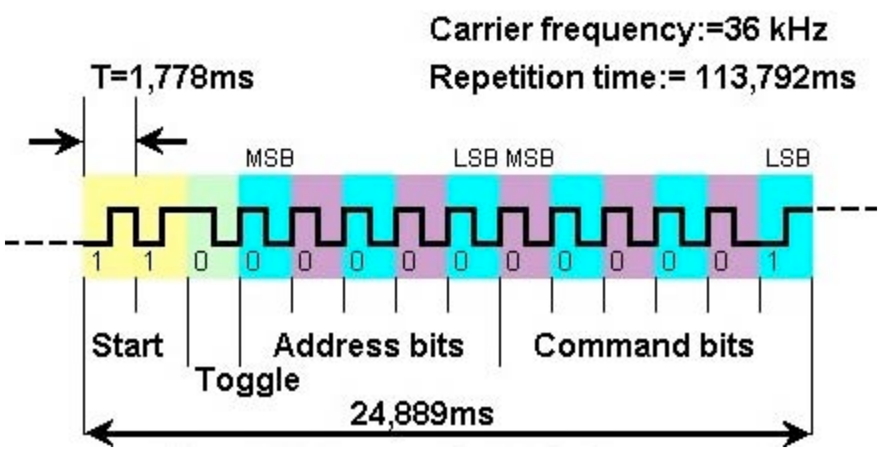


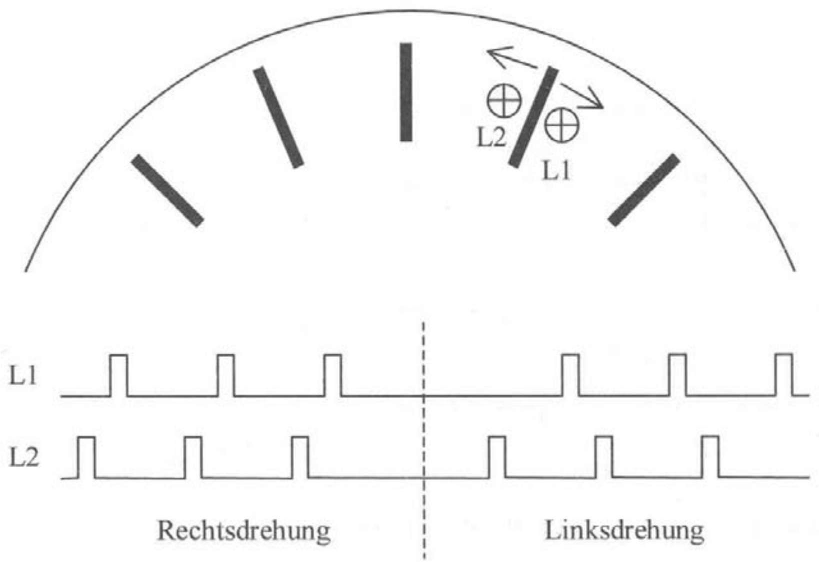
This contact bounce must be eliminated by either an electric low pass filter, or more commonly due to cost reasons, by appropriate debouncing routines in software.

Some sensors generate a sequence of pulses as outputs, where each pulse carries information about an event. Example: distance measurement using a wheel that generates pulses. The pulse frequency indicates the speed.

Time encoded signals: Many output devices are controlled by pulse sequences of well-specified shape (pulse width modulation-PWM). For example, a control signal for a stepping motor must adhere precisely to the temporal shape prescribed by the motor hardware suppli-er.

Another example is the encoding for remote controls, here the RC5 protocol from Philips for TV remote controls. A “1” is encoded as a low-high transition, a “0” as a high-low transition.



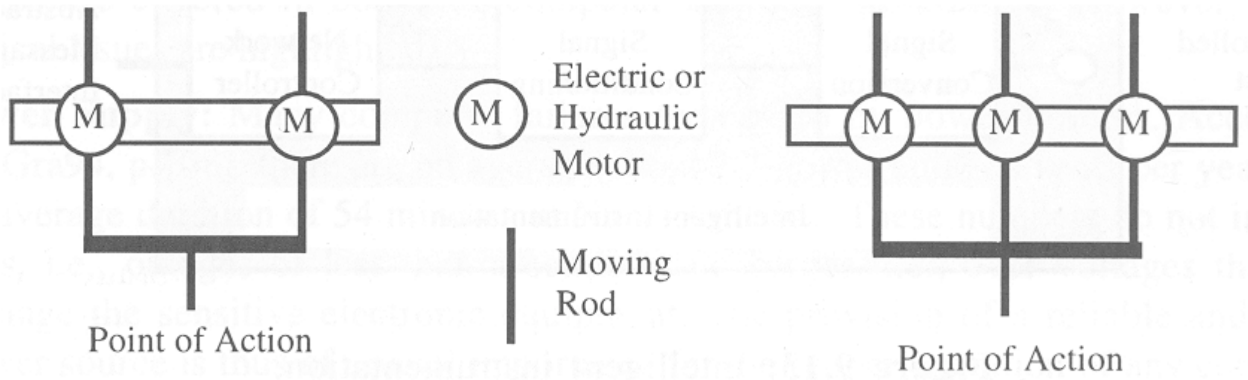


Fault-Tolerant Actuators

An actuator transduces the electrical signal at the output interface of the computer into some action in the controlled object (e.g., opening of a valve, running a motor, lighting an LED).

In a fault tolerant system, the actuators must also be fault-tolerant to avoid a single point of failure.

Example: positioning of a mechanical lever. At the point of action there may be for example the piston of a control valve (TMR: Triple-modular redundancy).

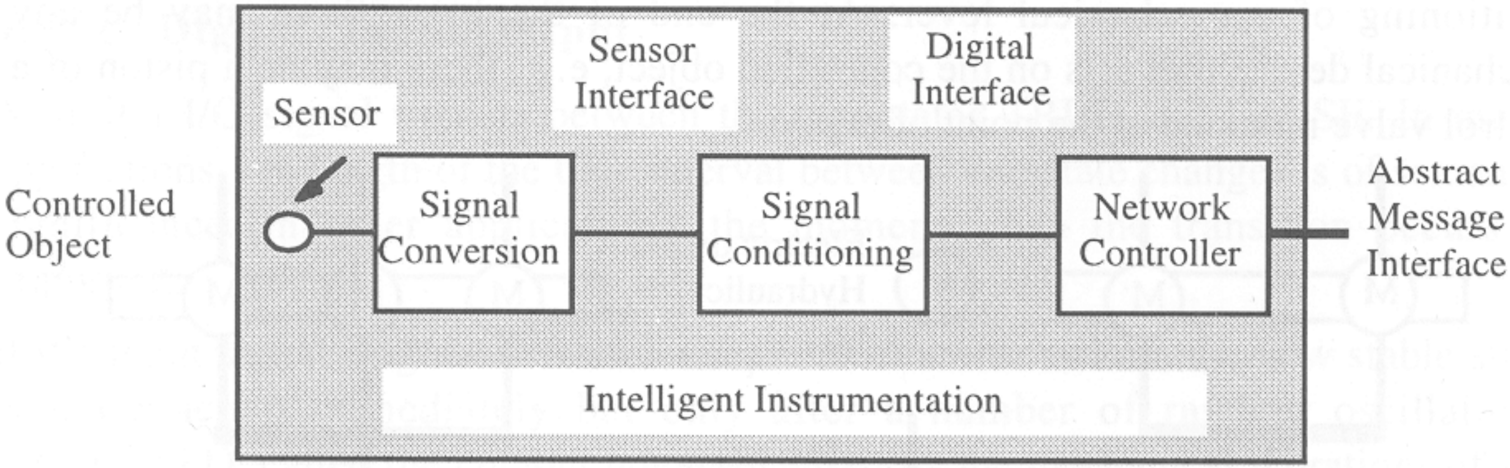


(a) Fail-Silent Actuators (b) TMR Actuator

A fail-silent actuator will either produce the correct output action or no result at all.
A TMR actuator can be viewed as a “voter”, where the majority of the votes determine the output action.

Intelligent Instrumentation

With decreasing silicon cost, there is a tendency towards encapsulating sensors/actuators and the associated microcontroller into a single physical housing to provide a standard abstract message interface. Such a unit is called an intelligent instrument.



The intelligent instrument hides the concrete sensor interface. Its microcontroller performs signal conditioning, signal smoothing, and local error detection.

Intelligent instruments simplify the connection of the plant equipment to the computer.
To make a measured value fault-tolerant, a number of independent sensors can be packaged into a single intelligent instrument.

Examples: acceleration sensor, airbag actuator, have a look at http://www.rn-wissen.de/index.php/Sensorarten

Points to Remember