
Standard Driver Manual

E-LAB AVRco

Pascal Multi-Tasking for Single Chips

Version for

AVR

© Copyright 1996-2018 by E-LAB Computers



Blaise Pascal Mathematician 1623-1662

The contents of this user guide is copyright protected by E-LAB Computers

Author Rolf Hofmann
Editor Gunter Baab

E-LAB

Mikroprozessor-Technik
Industrie-Elektronik
Hard + Software
8-Bit • 16-Bit • 32-Bit

Computers

E-LAB Computers
Grombacherstr. 27
D74906 Bad Rappenau
Tel 07268/9124-0
Fax 07268/9124-24
<http://www.e-lab.de>
info@e-lab.de

Important information

Everybody tries to write Software without bugs. The emphasis is on tries, because everybody knows that the more complex a Software is, the more likely it is to produce bugs.

We believe that this shouldn't have to be the norm, and that we should not have to live with the problems and mistakes (although some Software giants think like that ☺).

If you should find any errors, we would be grateful for any information. We will try to solve any problems as quickly as possible.

It is also a normal international agreement that the software producer does not accept liability for any costs arising out of errors in software, unless otherwise agreed.

E-LAB Computers do not accept liability for costs resulting out of errors in the software. It is a condition of use of this Software you agree with these terms. If you do not agree, you are not permitted to use the software.

As we have said, before this exclusion of liability is international standard.

This user guide and the software is intellectual property from E-LAB Computers and therefore copyright protected.

This document and the software it relates to are solely for the use of the purchaser. The purchaser is not permitted to give, sell or distribute these products. Distributing copies of these products to a third party is strictly prohibited.

We like to think that you as user of the software can make money from it and therefore also expect maintenance of the product. Illegal copies would make it impossible for us to be able to maintain this service.

As you see it is also in the interest of you, the user, to observe the copyright.

That's it the author

Table of Contents

1	Introduction.....	9
1.1	The Reason for Drivers.....	9
2	Overview.....	10
2.1	AVRco Versions	10
2.2	Driver and Manual Versions.....	10
2.3	Structure of the Documentation	10
3	Driver AVRco Standard Version	11
3.1	SwitchPorts: SwitchPort1, SwitchPort2, SwitchPort_G.....	11
3.1.1	Implementation	12
3.1.2	Imports and Exports.....	13
3.1.3	Key Repeat Support.....	14
3.1.4	Full Auto- Repeat Support	15
3.1.5	Support Functions.....	16
3.1.6	Variable Define of Switchport.....	16
3.2	KeyBoard Library Driver 2x2 ... 4x4	17
3.2.1	Exported Variables.....	18
3.2.2	Exported Functions and Procedures	18
3.2.3	Relation between Buttons and Bits.....	19
3.2.4	Debouncing.....	20
3.2.5	Key Repeat Support.....	20
3.2.6	Full Auto- Repeat Support	21
3.2.7	Support Functions.....	21
3.2.8	UserDevice and MatrixPort.....	22
3.3	KeyBoard Library Driver 2x8 ... 8x8	24
3.3.1	Exported Variables.....	25
3.3.2	Memory property of the keys	26
3.3.3	Exported Functions and Procedures	26
3.3.4	Relation between Buttons and Bits.....	27
3.3.5	Debouncing.....	27
3.3.6	Key Repeat Support.....	27
3.3.7	Full Auto- Repeat Support	28
3.3.7.1	Support Functions.....	29
3.3.8	User Device and KeyBoard8.....	30
3.4	LCD-Display	31
3.4.1	LCDPORT	31
3.4.2	Functions and Procedures	32
3.4.3	LCD-Split.....	35
3.4.4	User defined LCD-IO (LCDUserPort)	36
3.4.5	User defined LCD characters.....	36
3.5	LCD BarGraph Driver	37
3.5.1	Functions.....	39
3.6	LCDmultiPort Driver with up to 8 LCDs	41
3.6.1	Technical data.....	41
3.6.2	Types and Functions.....	43
3.6.3	Unused Port Pins of the LCD Control Port	45
3.6.4	Multi-Processing with the TWI Port.....	46



AVRco Standard Driver

3.7	LED 7seg Display	48
3.7.1	Disp7sPort, Mux NonMux	48
3.7.2	Functions and Procedures	49
3.8	LED 14seg Display Driver.....	52
3.8.1	Variables	53
3.8.2	Functions.....	53
3.9	LED 7seg Display Driver for up to 4 displays	56
3.9.1	Technical Data	56
3.9.2	Types	58
3.9.3	Variables	58
3.9.4	Functions.....	58
3.10	IOexpand Driver for up to 128 digital IOs	62
3.10.1	Technical Data	62
3.10.2	Functions and Variables	63
3.11	RS232/V24 Driver SerPort, SerPort2 and SerPort3, -4	66
3.11.1	Basic Functions (UART and USART)	66
3.11.1.1	Functions and Procedures	67
3.11.1.2	Interrupt Operation.....	69
3.11.1.3	Handshake Operation	71
3.11.1.4	RS485	74
3.11.1.5	TxComplete Callback.....	75
3.11.2	Port Switching	76
3.11.3	UART enable and disable for XMegas	77
3.11.4	IRDA for XMegas UARTs (IRcom).....	78
3.11.5	Advanced Functions for Controller with USART.....	79
3.11.5.1	Types, Procedures and Functions	79
3.11.6	SLIP packet oriented Protocol	80
3.12	Serial Network LAN	82
3.12.1	Implementation	84
3.12.2	Exported Variables.....	85
3.12.2.1	Memory Organization.....	85
3.12.3	Exported Functions and Procedures	87
3.12.4	Multi-Processing	88
3.12.5	Line Drivers.....	88
3.13	USBport Driver USBsmart XMega	90
3.13.1	Import of the USB Driver	90
3.13.2	Definition of the USB Driver	91
3.13.1	Callback Function	91
3.13.2	Exported Functions and Procedures	92
3.13.3	Host/PC Implementation	93
3.13.3.1	Initialisation etc.....	93
3.13.3.2	Device specific	93
3.13.3.3	Support.....	93
3.13.3.4	Data Transfer	94
3.13.1	Testprogram in the IDE PED32	95
3.13.2	Support Tools.....	96
3.14	PWMports.....	97
3.14.1	PWMport1A 1B 1C PWMport3A 3B 3C PWMport4A 4B 4C PWMport5A 5B 5C	97
3.14.2	PWMport2A, PWMport2B	98
3.14.3	Software PWM	99
3.14.4	Software-PWM SoftPWM8 XMega	101
3.15	XMega PWM	102
3.15.1	Functions and Procedures	102
3.16	XMega CRC	103
3.16.1	Functions and Procedures	103
3.17	SPI onBoard Network	104

3.17.1	Mini Network	104
3.17.2	Exported Variables.....	105
3.17.3	Exported Functions and Procedures	106
3.18	SPI Low Level SPIdriver, SPI_C...SPI_F Hardware Version	109
3.18.1	Functions.....	110
3.19	MSPI Low Level SPI driver MSPI_0..MSPI_3 AVR.....	113
3.19.1	Functions.....	114
3.20	MSPI Low Level SPI driver MSPI_C0..MSPI_F1 XMega.....	115
3.20.1	Functions.....	116
3.21	SPI Low Level SPIdriver1/2 Software Version.....	118
3.21.1	Functions.....	119
3.22	Serial SPI Flash AT25DF (XMega).....	120
3.23	TWI (I2C) Drivers Master and Slave.....	122
3.23.1	Master Mode	122
3.23.2	Slave Mode	122
3.23.3	Functions.....	124
3.23.3.1	TWlstat.....	124
3.23.3.2	TWlinp.....	124
3.23.3.3	TWlinpP	124
3.23.3.4	TWlout.....	125
3.23.3.5	TWloutP and TWloutWP.....	125
3.23.4	Multi-Processing with the Master	127
3.23.5	TWI (I2C) Slave	128
3.23.6	Common Functions.....	129
3.23.7	Functions in Handshake Mode	129
3.23.8	Functions in Transparent Mode	131
3.24	TWI-Net Library Driver	133
3.24.1	TWI network.....	133
3.24.2	Implementation	136
3.24.3	Exported Types.....	137
3.24.4	Exported Variables.....	137
3.24.4.1	Memory Organization.....	137
3.24.4.2	Variables only in MasterSlave and Slave Mode	139
3.24.5	Multi-Processing with the Master	140
3.24.6	Multi-Processing with the Slave	141
3.24.7	Exported Functions and Procedures	141
3.24.8	Functions and Procedures only available in MasterSlave mode	143
3.24.9	Functions and Procedures only available in MasterSlave and Master mode	143
3.24.10	Additional Functions.....	146
3.25	AD-Converter	149
3.25.1	ADCPort.....	149
3.25.2	ADCchans, RAMpage.....	149
3.25.3	Functions and Procedures	149
3.25.4	Call-Back Function onADCread	150
3.26	AD-Converter XMega	151
3.26.1	ADC_A ADC_B	151
3.26.2	Defines.....	151
3.26.3	Functions and Procedures	152
3.27	DA-Converter XMega	153
3.27.1	DAC_A DAC_B	153
3.27.2	Defines.....	153
3.27.3	Functions and Procedures	153
3.28	Real Time Clock.....	154
3.28.1	RTC-Functions/Procedures	155
3.28.2	Alarm-Procedures	156
3.28.3	Timer-Procedures	156



AVRco Standard Driver

3.29	I2C-Bus.....	158
3.29.1	I2CPORT.....	158
3.29.2	158	
3.29.3	Functions and Procedures.....	159
3.29.4	The I2C-BUS Interface.....	161
3.29.5	Multi-Processing and I2C.....	161
3.30	I2Cexpand Driver for up to 8 bidirectional Ports	163
3.30.1	Technical Data	163
3.30.2	Types and Functions.....	165
3.30.3	Multi-Processing and TWI Port	165
3.31	Pulse Counter driver PulseCount.....	167
3.31.1	Functions.....	167
3.32	Pulse Counter driver PulseCount XMega.....	168
3.32.1	Functions.....	168
3.33	Incremental Encoder Driver IncrPort	169
3.33.1	Functions.....	169
3.34	Incremental Encoder Driver IncrPort4	171
3.34.1	Functions.....	171
3.35	UP/DOWN Counter Driver XMega.....	173
3.35.1	Functions.....	174
3.36	QDEC Incremental Encoder Driver XMega.....	175
3.36.1	Functions.....	176
3.37	Stepper-Motor.....	177
3.37.1	Fundamentals	177
3.37.1.1	ConstantVoltage Operation.....	177
3.37.1.2	ConstantCurrent Operation.....	177
3.37.1.3	Limitations.....	177
3.37.2	Acceleration Ramp.....	178
3.37.3	Drive Modes	178
3.37.4	Import of the Stepper	179
3.37.5	Parameters of the Stepper.....	179
3.37.6	Commands/Procedures of the Stepper	180
3.38	Stepper-Motor in UserMode	182
3.38.1	Parameters of the Stepper.....	183
3.38.2	Commands/Procedures of the Stepper	184
3.38.3	StepperIOS	187
3.39	Servo Driver for up to 8 digital Servos.....	189
3.39.1	Technical Data	190
3.39.2	Functions.....	190
3.40	DCF-77 Decoder	192
3.40.1	DCF77-Functions/Procedures	193
3.40.2	Hardware.....	194
3.41	AVR Timer Low Level Driver TickTimer	195
3.41.1	Variable	195
3.41.2	Functions.....	196
3.41.3	Interrupt.....	196
3.42	FreqCount Driver - Frequency Counter/Timer	197
3.42.1	Overview Frequency Counter	197
3.42.2	Overview Pulse Timer.....	198
3.42.3	Functions, Procedures, Types	199
3.43	FreqCount driver – Frequeny Counter XMega	201
3.43.1	Import & Define FreqCount	202
3.43.2	Functions, Procedures, Types	202
3.44	RFID125 Receiver (XMega only at this time)	204

3.45 RC5 Decoder/Encoder Driver	205
3.45.1 Receiver	205
3.45.2 Transmitter	205
3.45.3 Functions Receiver	206
3.45.4 Functions Transmitter	206
3.46 SHT11 Temperature and Humidity Sensor Driver.....	208
3.46.1 Variables	209
3.46.2 Functions.....	209
3.47 Sound Generator and Driver	211
3.47.1 Functions.....	211
3.48 SysLEDblink	213
3.48.1 Implementation	214
3.48.2 Functions.....	214
3.49 Line Printer Driver LPTport.....	216
3.49.1 Functions and Types.....	217
3.50 Banking Port.....	219
3.50.1 Implementation	220
3.50.2 Hardware Example	221
3.51 Flash Downloaded/Writer	224
3.51.1 Overview	224
3.51.2 Compiler Switches	224
3.51.3 BootApplication	225
3.51.4 FlashWrite at Runtime	225
3.51.4.1 Functions.....	225
3.51.5 Usage of FlashWrite	227
3.51.5.1 Implementation.....	227
3.51.5.2 Rewrite a Page completely	228
3.51.5.3 FlashDownLoad at RunTime	231
3.51.5.4 FlashDownloader Functions	233
3.51.6 Host Program	234
3.51.7 Boot Area, Optimiser and re-compile.....	235
3.52 BootApplication and MainApplication	236
3.52.1 XMega FlashLoader.....	238
3.53 Device Drivers	240
3.53.1 Organisation.....	240
3.53.2 Formal Declarations.....	241
3.53.3 BlockDevice	244

1 Introduction

1.1 The Reason for Drivers

As with every computer system, a Microcontroller system does not exist in isolation. It communicates with the outside world. This could be a human, another computer system, external media, sensors, actuators etc. There are many options.

This communication can be very simple, e.g. the control of LEDs on a port. It becomes more complicated if the port is used to read mechanical switches, pushbuttons etc. With these debouncing is indispensable. The complexity ends with graphic and file system drivers.

All these tasks are generally done by what are known as *drivers*. The drivers control the resources.

In the embedded sphere these drivers were for years, even tens of years, personal developments, mostly written in assembler. A common development systems offered at best a support for a serial interface. Because of the limited resources of the controllers this was generally sufficient.

In recent years the controllers increased their performance from about 1MIPS to 20MIPS or more and the on-chip memory from typically 1kByte up to 1Mbyte is now possible to realize very complex systems with small controllers. In addition the customers demands for software grew continually. In the past some sensors, LEDs, relays and push buttons were sufficient. Nowadays control panels, LCD or even graphic LCDs, file systems, networks and complex calculations are common demands.

With a PC in mind this is no big problem. Everything is built in including all of the hardware as the necessary drivers.

But even today a typical development system offers nearly nothing. In the best case one can buy libraries for a lot of money and these must frequently be adapted with lots of effort.

This results frequently in “do-it-yourself” jobs and the self-made drivers are the developers “pets” that are carefully maintained. But rarely is the actual driver needed in the library. This leads again to a do-it-yourself job with extensive development and debug time consumption.

For that reason the AVRco system has a very large number of drivers, so the programmer can focus the essentials - his application.

A frequent counter argument is: *“I only know my own software exactly”* and *“I do not know how a foreign driver is precisely working”*.

The answer is that the drivers in a development system are used hundreds of times in a wide area and can essentially be considered as bug-free. Besides that, development time is expensive and can be enormously reduced by available and proven drivers.

E-LAB is proud to have the development system with the greatest number of built-in drivers.

Nearly all drivers are supported by the **AVRco Application Wizard** and also by the **AVRco Simulator**.



2 Overview

2.1 AVRco Versions

All AVRco Versions support all AVR Controller with an internal RAM (for the stack). That means in practice the whole range.

AVRco Profi Version:

the Profi Version contains all available drivers, including very complex ones like e.g. a FAT16 file system and an extensive library for graphic LCDs.

The professional program development is furthermore assisted by the full support of Units.

AVRco Standard Version:

the Standard Version omits only the most complex drivers, and does not support units.

AVRco Demo Version:

the Demo Version supports all controllers and all drivers of the Standard Version.

The **only restriction** is the limitation of the generated code to max. 4 kByte size.

2.2 Driver and Manual Versions

This Manual concerns the drivers contained as well in the AVRco Standard Version as in the AVRco Profi Version.

2.3 Structure of the Documentation

..\E-Lab\DOCs\DocuCompiler.pdf:

contains the Pascal language description and the enhancements compared with Standard Pascal

..\E-Lab\DOCs\DocuStdDriver.pdf:

contains the description of the drivers contained as well in the Standard, as in the Profi Version.

..\E-Lab\DOCs\DocuProfiDriver.pdf:

contains the description of the drivers contained only in the Profi Version.

..\E-Lab\DOCs\DocuReference.pdf:

contains a Short Reference (the the same as the online help)

..\E-Lab\DOCs\DocuTools.pdf:

contains the description of the IDE, the simulator, a tutorial etc.

..\E-Lab\IDE\DataSheets\Release-News.txt:

lists the enhancements in chronological order.

The enhancements are documented in the above mentioned .pdf files (DocuXXX.pdf)

..\E-Lab\AVRco\Demos\ :

contains many test and demo programs

..\E-Lab\DOCs\ :

contains the documentation, the Release-News and further schematics and data sheets

3 Driver AVRco Standard Version

3.1 SwitchPorts: SwitchPort1, SwitchPort2, SwitchPort_G

A SwitchPort in the AVRco System is a debounced port. Debouncing means that a Port bit must be stable and doesn't change its state at least 2 System Ticks so it will be recognized as a stable condition. Then this bit (the state of it) will be copied into a corresponding (stable) byte.

By debouncing the application can be sure that a mechanical switch or key has a stable and unique condition when it is read by the SwitchPort functions. False interpretations are avoided with debouncing.

The debouncing is done in the SysTick so it must be imported. Furthermore a debounce can only be done when the global interrupt is enabled (EnableInts or Start_Processes).

The application can access a single bits of a SwitchPorts with INP_STABLE__(BitNum). This function returns the actual state of the selected bits as true or false.

With most of the applications the function INP_RAISE(BitNum) is more important. Here the status change is returned. If the Port Bit has changed from stable-false to stable-true a true is returned and at the same time this edge state is reset. This means that a positive edge will be latched and can be read by this function. Each additional function call then returns a false until a new positive edge appeared. Then the state of the edge detection remains true until this function is called again, no matter how often the edge appeared and the port bit has changed its state. This function can only say that at least once a transition happened after the last function call. How often this happened is unknown and the current state of the port bit doesn't matter.

An additional possibility for the application is to read out the stable state of the whole port by reading the byte PORT_STABLE. It contains all debounced bits of this port.

The AVRco system knows 3 SwitchPorts: SwitchPort1, SwitchPort2 and SwitchPort_G.

The first two are related to an entire unique port and read and process this port in one part. So up to 16 Switches and keys can be handled, with the limitation that always eight bits must reside in the same physical port of the CPU.

This limitation to one port is not there when SwitchPort_G is used. The 8 bits of this "virtual" port can be placed anywhere into the CPU's address area which includes all ports of the CPU. The disadvantage of SwitchPort_G is that the processing, which done in the SysTick, requires much more work (code size, time) compared to the standard SwitchPort.

In general the SysTick must not be too short in order to have a secure debouncing. One must know that a mechanical switch can bounce between 1 and 20msec. Then a SysTick of 1..5msec can not do a secure debouncing in most cases.

Because of this with SysTicks < debounce time it makes sense to do the debouncing not with each SysTick but in intervals of x SysTicks. For this purpose there is the **Define Debounce** which imports an internal interval timer. So at design time it's possible to define that for example only at each 5th tick the debouncing is executed. This timer, if defined, is then also responsible for all system debouncing which includes the Matrix Port and the Keypoint.

None of the SwitchPort drivers do any port or hardware initialising. It's the job of the application to initialise these physical ports in a correct manner.



AVRco Standard Driver

3.1.1 Implementation

Imports

As always with AVRco the driver must be imported.

Import SysTick, SwitchPort1, ...;

Defines

The port register (in most cases the PIN-register must be used!)

```
Define ProcClock   = 8000000;      {Hertz}
          SysTick    = 10;           {msec}
          StackSize  = $0030, iData;
          FrameSize  = $0030, iData;
          SwitchPort1 = PinB;        {SwitchPort}
```

Optional the polarity of the trigger edge (Edgemask) can be defined with:

```
Define SwitchPort1 = PinB, $17;    {SwitchPort, edgepolarity}
```

An additional **option** is the definition of the port polarity:

```
Define PolarityP1  = %00000110;    {Polarity SwitchPort1}
```

The purpose of the port **Polarity** is to always have positive results with low-active inputs.

```
PolarityP1 = $F0;      // upper 4bits high active, lower 4bits low active
```

With the PolarityP byte the active levels of each input (8bits) can be set. Each bit corresponds to an input bit and defines whether a 0 or a 1 is the result of a closed switch. So it doesn't matter how the switch logic is constructed, a high is always returned for a closed switch and the system can always work with positive logic.

The Edgemask defines the edge for each input. A binary zero means that a positive edge must be latched, or if a correct polarity is defined a 0 means "onSwitchClose" trigger and a 1 means "onSwitchOpen" trigger.

The port polarity definition of course also influences the behaviour of the edge mask because at first the physical port must be read and then processed with PortPolarity (XOR). The result of this operation must then be processed with the Edgemask.

SwitchPort2 has the same definition as SwitchPort1. So the above is also true for this port.

An additional **option** is the define of a debounce interval:

```
Debounce = 5;    {debounce every 5 SysTicks}
```

With very short (fast) SysTicks < 10mSec and long bouncing switches or keys a secure debouncing is not possible. With this optional define Debounce the used interval can be adjusted. If for example the Tick is 1mSec and the debounce interval is 20 the debouncing is done every 20mSec which is a good value. The Define of the Debounce influences also the debounced system drivers like KeyPort and MatrixPort.

SwitchPort_G definition differs significantly from the other two SwitchPorts:

```
Import SysTick, SwitchPort_G, ...;
Define ProcClock    = 8000000;    {Hertz}
        SysTick      = 10;         {msec}
        StackSize    = $0030, iData;
        FrameSize     = $0030, iData;
        SwitchPort_G = [ResetKey, PinC, 4][StartInput, PinC, 5] [Sensor, PinB, 1], %00000001;
        PolarityP_G   = %00000110; {Polarity SwitchPort_G}
```

Because each of the SwitchPort_G bits can reside in a different port, each bit must then completely defined. This must be done with up to eight pin-defines:

[PinName, PortName, BitNum]

PinName is a freely definable name which can be used by the application as byte constant.

PortName is the name of a PIN-register but can be also any numerical address in the RAM area.

BitNum is the desired bit in the (Port/address) PortName.

The bit definitions must be separated by []. A comma or semicolon is not permitted here. At the end of the up to 8 bit-definitions the optional parameter **EdgeMask** can be appended, separated by a comma.

The port Polarity parameter **PolarityP_G** is also an option here, as is the **Debounce** Define.

3.1.2 Imports and Exports

An import of a SwitchPort also imports a number of bytes which are reserved for the system. The port handler is linked into SysTick.

Each SwitchPort exports a variable PORT_STABLE which contains all bits of the ports in their debounced state. This variable can always be read.

A SwitchPort can have unexpected conditions after a CPU reset depending on the port levels, so there is the procedure SWITCHPORTx_CLEAR which clears the port's state.

Because the state of a Switch port is stable at least after 3 SysTicks after the global interrupt enable (EnableInts or Start_Processes) it makes sense to wait a few ticks and then call SWITCHPORTx_CLEAR.

The debounced state of a single port bit can be received with the function INP_STABLEx(bit).

Whether a Switch port bit had an edge change or not can be determined with the function INP_RAISEx(bit), but bear in mind that only that edge which was selected with the EdgeMask definition is latched.

Definitions

```
Var PORT_STABLE1 : byte;
        PORT_STABLE2 : byte;
        _STABLE_G : byte;

Procedure SWITCHPORT1_CLEAR; // SWITCHPORT1
Procedure SWITCHPORT2_CLEAR; // SWITCHPORT2
Procedure SWITCHPORT_G_CLEAR; // SWITCHPORT1_G

Function INP_STABLE1 (bit : byte) : boolean; // SWITCHPORT1
Function INP_STABLE2 (bit : byte) : boolean; // SWITCHPORT2
Function INP_STABLE_G (bit : byte) : boolean; // SWITCHPORT_G

Function INP_RAISE1 (bit : byte) : boolean; // SWITCHPORT1
Function INP_RAISE2 (bit : byte) : boolean; // SWITCHPORT2
Function INP_RAISE_G (bit : byte) : boolean; // SWITCHPORT_G
```



AVRco Standard Driver

3.1.3 Key Repeat Support

The driver also provides an optional support of external repeat functions. In order to implement an auto-repeat function of the keys the controlling part of the application must know how long a key is activated.

To support this each key has its own optional 8bit timer which is incremented in each scan interval as long as the key stays active and stable. The timer stops at 255. If the key becomes released the timer will be reset to zero.

There are 8 timers. These timers are placed into an array and can be read by the application.

If this option must be used the timers must be imported.

```
Import SysTick, SwitchPort1, ... ;           // SwitchPort2, SwitchPort_G
```

```
From System import SwitchPTimer1; // SwitchPTimer2, SwitchPTimer_G
```

This import defines the timer array in this way:

```
Var SwitchTimerArray1 : array[0..7] of byte;
```

Now the application can access the array without any restrictions:

```
if 1[2] > 0 then  
  // key is pressed  
  ...  
endif;
```

or:

```
case SwitchTimerArray1[4] of  
  0 : // key released  
  |  
  1..10 : // key initially activated  
  |  
else  
  // do any repeat action  
endcase;
```

Note:

The timers are incremented in the scan interval. In most cases this is the once per SysTick. If **Debounce** is defined the increments are done in (SysTick * Debounce) intervals.

Because the Switch port always scans an entire 8bit port, unused port bits can change their timers. But more worse is if such pins are written into the pipe and start an event (WaitPipe etc). To prevent this unwanted behaviour it is possible to mask unwanted bits.

This Define supports it:

```
Define SwitchPMask1 = $0F; // SwitchPMask2, SwitchPMask_G
```

Mask bits set to 1 enable this pin for processing, bits set to 0 exclude these bits from processing.

3.1.4 Full Auto- Repeat Support

The driver also provides an optional internal auto-repeat function.
The SwitchPTimerx is used to build the Auto-Repeat function.
The resulting Key events are then stored into a special pipe **KeyboardPipe**.

In most cases the application now does not need any of the standard KeyPort functions.
All operations can be done with the Pipe. All of the common pipe function can be used.

If this option must be used the SwitchPTimers must be imported.

```
Import SysTick, SwitchPort1, ... ;           // SwitchPort2, SwitchPort_G
```

```
From System import Pipes, SwitchPTimer1; // SwitchPTimer2, SwitchPTimer_G
```

The Auto-Repeat function must be declared with a Define:

```
Define SwitchPPipe1 = PipeLen, FirstRepeat, RepeatRate; // SwitchPPipe2, SwitchPPipe_G
```

Then this variable will be exported:

```
SwitchKeyPipe1 : Pipe[PipeLen] of Byte;    // SwitchKeyPipe2, SwitchKeyPipe_G
```

The first parameter **PipeLen** defines the length of the pipe and hence the number of keystrokes which can be stored in it.

The second parameter **FirstRepeat** defines the time in msec elapsed before the repeat starts.

The third parameter **RepeatRate** defines the time in msec between two repeats.

If a key is actually pressed and the debounce time is elapsed the key ID (0..7) is immediately written into the pipe. If the key stays active the auto-repeat starts after the **FirstRepeat** time has elapsed. The repeat writes then in **RepeatRate** msec intervals into the pipe. Each time the ID of this key is written into the pipe. The auto-repeat works until either this key is released or the pipe is full.

If any key becomes inactive the corresponding Key-ID + \$80 (8th bit set) is written into the pipe.

The application can access the pipe at any time with:

```
If PipeStat (SwitchKeyPipe1) <> 0 then  
    key:= PipeRecv (SwitchKeyPipe1);  
    ...  
endif;
```



AVRco Standard Driver

3.1.5 Support Functions

The repeat operation can be enabled or disabled at runtime with these procedures:

Procedure SwitchKeyRepeat1 (rept : boolean);
Procedure SwitchKeyRepeat2 (rept : boolean);
Procedure SwitchKeyRepeat_G (rept : boolean);

Basically in addition all SwitchPort functions can be used like ClearKeyBoard. But in most cases this is not necessary.

Also all the pipe related functions can be used like PipeFlush. It is also possible to write directly into the keyboard buffer with PipeSend, a technique that was often used in the old MSDOS days.

Processes and Tasks can use the Function **WaitPipe**.

3.1.6 Variable Define of Switchport

The Define for SwitchPort1 and SwitchPort2 can also be a byte variable:

Define SwitchPort1 = @name, ...

where "name" can be any identifier because at this time no variable is defined.

The system then implements this variable "name" of type byte automatically.

The main purpose is that for example Processes or Tasks can "feed" this byte with special IO results (e.g. I2C) and the system debounces it.

3.2 KeyBoard Library Driver 2x2 ... 4x4

With many embedded applications there are control keys connected, which the user uses for keying in commands or parameters. As long as these keys can be connected to a single input port, this is no problem in most cases. If there are many keys it's necessary to use many port pins. In most cases the available port pin count of a CPU is too small for connecting a larger keyboard.

Here we can use the multiplex processing of a keyboard. The keys are connected at the cross points of a matrix. The matrix is constructed of columns and rows. The maximal possible count of the keys is determined of the product of columns x rows. With 4 columns and 4 rows the maximum is 16 Keys. With a 4x4 matrix it is possible to halve the necessary pins compared to the non-multiplexed scheme.

With the multiplex scheme the rows are activated one after another, all columns are read in and analysed. The result is stored into bit fields. A 4x4 matrix shows a 16bits result. Only one row is activated at one time, the others remain in TriState condition.

In this implementation each column has a PullUp resistor of 1.. 10kOhm. The impedance of the internal PullUps of the AVR is too large (typ. 80k) for this purpose. Because of the fast scan, the positive edges of the columns are too slow. They are charged by the capacitive loads of the CMOS-inputs, wires, keys etc. and false results must be expected. The scanning is done within the SysTick routine, which must be imported for that reason.

While scanning the keys are debounced. A keys state must be stable for at least 3 SysTicks before a change will be accepted.

Imports

The SysTick driver must be imported.

Import SysTick, MatrixPort, ... ;

By the import of the MatrixPorts the enumeration "Keys" is automatically imported:

Type Keys = (Key1, Key2, ..., KeyN);

Const lastMatrixKey : Keys = KeyN;

Defines

The mode and parameters of the keyboard driver must be defined.

*Define MatrixRow = PortA, 4 {use PortA, start with bit4}
 MatrixCol = PinA, 0; {use PinA, start with bit0}
 MatrixType = 3, 4; {3 Rows at PortA, 4 Columns at PinA}*

The application can process the IO-accesses by itself. But then the Define must be changed:

Define MatrixPort = UserPort;

MatrixRow and *MatrixCol* must not be defined in this case. The application now must provide this Call-Back function:

UserDevice MatrixPortIOS : byte;

Now the drivers internal functions like debouncing and scanning are used where the application feeds them with the result of reading a special, normally not supported hardware.



AVRco Standard Driver

3.2.1 Exported Variables

Memory Organisation

All keyboard related variables are defined sequentially in the memory.

```
_OLDMX2      byte  
_OLDMX1      byte  
_MATRIX2    byte  
_MATRIX1    byte  
_CHANGEMX2   byte  
_CHANGEMX1   byte  
KEYBOARD     semaphore
```

Don't access any of the above vars, except "KeyBoard".

The var "KeyBoard" is an 8-bit Semaphore which can be accessed with the corresponding methods. This semaphore contains as an information the count of key clicks. It is reset with "ClearKeyBoard".

This semaphore is highly recommended if processes are imported to avoid continuous polling of the keyboard. The process sleeps until a key becomes active.

```
process HandleKeyBoard(20, 20 : idata);  
begin  
  WaitSema (KeyBoard);  
  case GetKey of  
    Key1 : ...; |  
  endcase;  
end;
```

3.2.2 Exported Functions and Procedures

Function *ReadKey* (*const key* : *Keys*) : *boolean*;

The actual state of a key is returned with a true or false. If the key is pressed, the result is true, otherwise it's false. The parameter Key must be of type "Keys", e.g. Key1..Key16.

Function *KeyRaised* (*const key* : *Keys*) : *boolean*;

The state of a key is returned by a true or false. If a key was pressed at least once, then a true is returned. There is no information as to whether the key is still pressed or not. The KeyBoard driver has a memory function. This means that a pressed key is latched and stored in a bit until it is cleared with the KeyRaised function or a 'ClearKeyBoard' is called. With this a continuous polling of the keys by the user program can be avoided.

Function *ReadKeyBoard* : *BitSet of Keys*;

The result of this function is a "BitSet of Keys" where each bit corresponds with a key. The result is the static state of the keys and not the latched state. The definition of " BitSet of Keys" is as follows:

```
Type Keys    = (Key1, Key2, ..., KeyN);  
      KeySet = BitSet of Keys
```

There are several possibilities to handle the BitSet-result:

```
bool := [Key1, Key4] in ReadKeyBoard;  
bool := ReadKeyBoard in [Key1, Key4];
```

```
bool := ReadKeyBoard = [Key1, Key4];  
bool := [Key1, Key4] = ReadKeyBoard;
```

Example program:

an example is in the directory ..\E-Lab\AVRco\Demos\BitSets

Function KeyStatRaised : boolean;

The state of the entire KeyBoard is returned with a true or false. If there was at least one key pressed and latched and "KeyRaised" does not read out this key, the result becomes true. In other words, the function returns a true if a key was pressed and is still latched. With this function a complete KeyBoard read-out by many "KeyRaised" can be avoided until this function returns a true.

Function GetKeyRaised : Keys;

This function calls "KeyStatRaised" and if it gets a true, the first latched key found is returned. If no key is latched the function polls KeyStatRaised until a latched key is found. This means, if no key can be found the user program never gains the control, because this function loops endless.

Function KeyStat : boolean;

The state of the entire KeyBoard is returned with a true or false. If there is at least one key pressed the result becomes true. In other words, the function returns a true if a key is still pressed. With this function a complete KeyBoard read-out by many "GetKey" can be avoided until this function returns a true.

Function GetKey : Keys;

This function calls "KeyStat" and if it gets a true, the activated key found is returned. If no key is active the function polls KeyStat until an active key is found. This means, if no key can be found the user program never gains the control, because this function loops endless.

Procedure ClearKeyBoard;

This procedure clears the entire keyboard including the memory "KeyRaised" and the semaphore.

Procedure KeyboardEnable (ena : boolean);

This procedure enables or disables the entire KeyBoard including the Scanner, Timer etc. So it is possible to disable user key operations or with time critical program parts the SysTick latency (interrupt disable duration of the SysTicks) can be temporary reduced to a minimum.

3.2.3 Relation between Buttons and Bits

The button on the cross point of Row1 and Col1 is the most significant bit. Then button on the point Row1/Col2 follows etc.

If the matrix for example has 12 buttons so there are 12 bits relevant, this means that the result of a function call can be Key1..Key12. The button on Row1/Col1 returns Key12. If the layout of the keyboard is 4Cols and 3Rows then the button on Row3/Col4 returns the value Key1.

Row1/Col 1 Key12	Row2/Col 1 Key8	Row3/Col 1 Key4
Row1/Col 2 Key11	Row2/Col 2 Key7	Row3/Col 2 Key3
Row1/Col 3 Key10	Row2/Col 3 Key6	Row3/Col 3 Key2
Row1/Col 4 Key9	Row2/Col 4 Key5	Row3/Col 4 Key1



AVRco Standard Driver

3.2.4 Debouncing

With `SysTick < bounce-time` it makes sense not to do a debouncing with each `SysTick` but to do the debouncing in intervals of `x SysTicks`. For this purpose there is the optional

```
Define Debounce = nn; // nn = SysTicks
```

Which imports an internal interval timer. So the debouncing is done every 5th tick for example. This timer then is also responsible for system wide debouncing, including the `KeyBoard8` and the `SwitchPorts`.

3.2.5 Key Repeat Support

The driver also provides an optional support of external repeat functions.

In order to implement an auto-repeat function of the keys the controlling part of the application must know how long a key is activated.

To support this each key has its own optional 8bit timer that is incremented in each scan interval as long as the key stays active and stable. The timer stops at 255. If the key becomes released the timer will be reset to zero.

Each key has its own timer. These timers are placed into an array and can be read by the application.

If this option is to be used the timers must be imported.

```
Import SysTick, MatrixPort, ... ;
```

```
From MatrixPort import MatrixTimer;
```

This import defines the timer array in this way:

```
Var MatrixTimerArray : array[Keys] of byte;
```

Now the application can access the array without any restrictions:

```
if MatrixTimerArray[Key1] > 0 then  
    // key is pressed  
    ...  
endif;
```

or:

```
case MatrixTimerArray[Key1] of  
    0 : // key released  
    |  
    1..10 : // key initially activated  
    |  
    else  
        // do any repeat action  
    endcase;
```

Note:

The timers are incremented in the scan interval. In most cases this is once per `SysTick`. If **Debounce** is defined the increments are done in `(SysTick * Debounce)` intervals.

3.2.6 Full Auto- Repeat Support

The driver also provides an optional internal auto-repeat function. The MatrixTimer is used to build the Auto-Repeat function.

The resulting Key events are then stored into a special pipe **KeyboardPipe**.

In most cases the application now does not need any of the standard KeyPort functions. All operations can be done with the Pipe. All of the common pipe function can be used.

If this option is to be used the Matrix Timers must be imported.

```
Import SysTick, MatrixPort, ... ;
```

```
From MatrixPort import MatrixTimer;
```

The Auto-Repeat function must be declared with a Define:

```
Define MatrixPipe = PipeLen, FirstRepeat, RepeatRate;
```

Then this variable will be exported:

```
MatrixKeyPipe : Pipe[PipeLen] of Keys;
```

The first parameter **PipeLen** defines the length of the pipe and hence the number of keystrokes that can be stored in it.

The second parameter **FirstRepeat** defines the time in msec elapsed before the repeat starts.

The third parameter **RepeatRate** defines the time in msec between two repeats.

If a key is actually pressed and the debounce time is elapsed the key is immediately written into the pipe. If the key stays active the auto-repeat starts after the **FirstRepeat** time has elapsed. The repeat writes then in **RepeatRate** msec intervals into the pipe. Each time the ID of this key is written into the pipe. The auto-repeat works until either this key is released or the pipe is full.

If any key becomes inactive the corresponding Key-ID + \$80 (8th bit set) is written into the pipe.

The application can access the pipe at any time with:

```
If PipeStat (MatrixKeyPipe) <> 0 then  
  key:= PipeRecv (MatrixKeyPipe);  
  ...  
endif;
```

3.2.7 Support Functions

The auto-repeat operation of the driver can be enabled and disabled at runtime with:

```
Procedure KeyBoardRepeat (rept : boolean);
```

In addition all MatrixPort functions can be used, such as ClearKeyBoard, but in most cases this is not necessary.

Also all the pipe related functions like PipeFlush can be used. It is also possible to write directly into the keyboard buffer with PipeSend, a technique that was often used in the old MSDOS days.

Example program:

an example is in the directory ..\E-Lab\AVRco\Demos\KeyBoardAuto



AVRco Standard Driver

3.2.8 UserDevice and MatrixPort

If an application cannot use the standard ports of an AVR for the KeyBoard-IO it's possible for the user application to do its own special port accesses. The driver controls the scanning and does the debouncing.

Define

```
ProcClock = 16000000;    {Hertz}  
SysTick   = 10;  
StackSize = $0064, iData;  
FrameSize = $0064, iData;  
KeyB8Type = UserPort, 5; {5 Rows, 8 Columns}
```

Implementation

```
{ $NORETURNCHECK }  
UserDevice Keyboard8IOS : byte;  
begin  
  ASM;  
  ...  
  ...  
  EndASM;  
end;
```

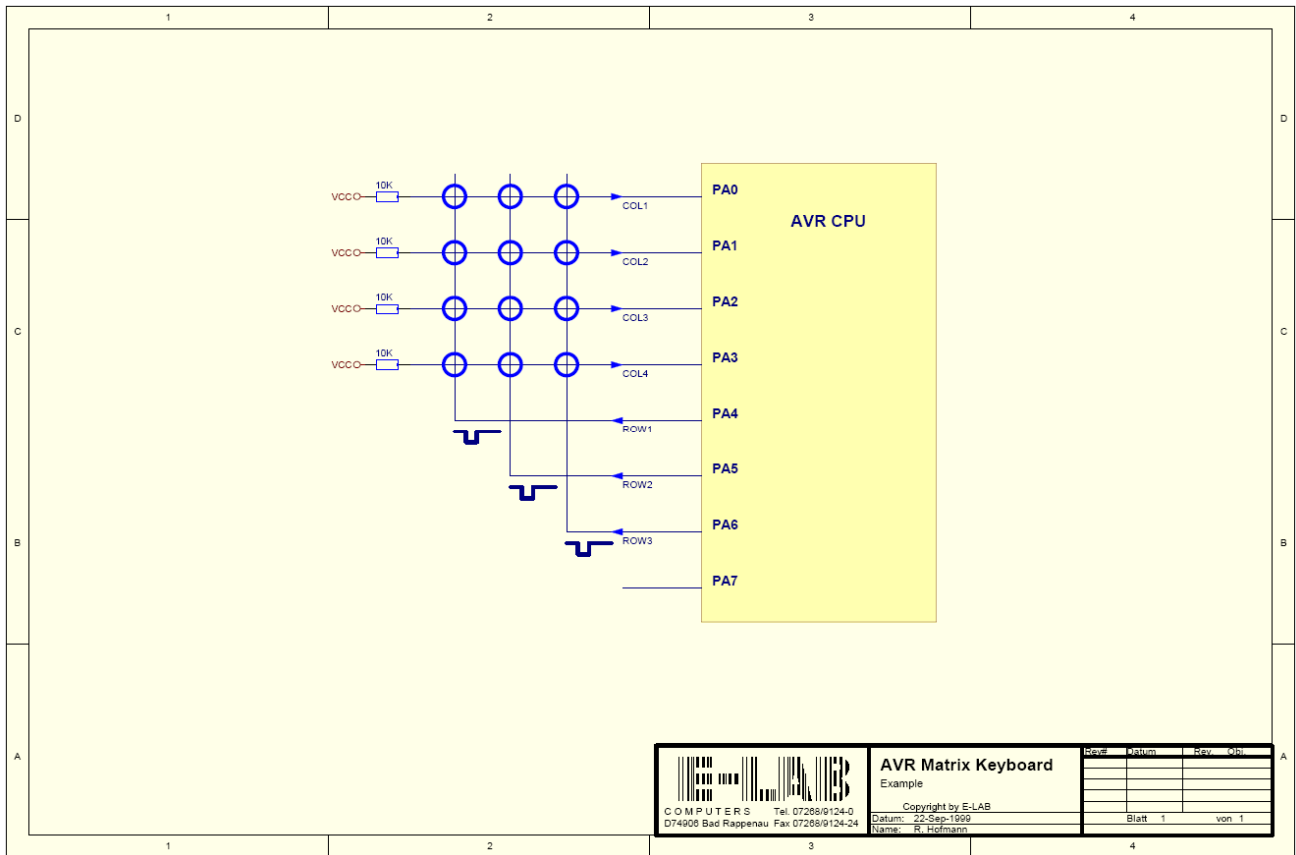
The driver passes the register `_ACCA` (R17) and `_ACCB` (R16) to the `UserDevice` function. Register `_ACCA` contains the actual COL number as 0, 1, 2, 3, 4 etc. As an alternative the register `_ACCB` contains the COL number as a bit position \$1, \$2, \$4, \$8, \$10, \$20 etc. Dependant on the construction of the external hardware `_ACCA` or `_ACCB` can be used to read-in a ROW .
With a multiplexer `_ACCA` is preferable, with a latch or similar `_ACCB` is suitable better.

The application must return in `_ACCA` the column read result to the calling driver.

The program code must be written in assembler in order to be as fast as possible because this function is called from the `SysTick`. Because of this only `_ACCA` and `_ACCB` can be used for the IO-processing. If additional registers must be used they must be saved with a `PUSH` and later restored with a `POP`.

Local variables are forbidden and so is the calling of system drivers or mathematical operations of the system.

AVRco Standard Driver



schematic **KeyBoard**



AVRco Standard Driver

3.3 KeyBoard Library Driver 2x8 ... 8x8

With many embedded applications there are control keys connected, which the user uses for keying in commands or parameters. As long as these keys can be connected to a single input port, this is no problem in most cases. If there are many keys it's necessary to use many port pins. In most cases the available port pin count of a CPU is too small for connecting a larger keyboard..

Here we can use the multiplex processing of a keyboard. The keys are connected at the cross points of a matrix. The matrix is constructed of columns and rows. The maximal possible count of the keys is determined of the product of columns x rows. With 8 columns and 8 rows the maximum is 64 Keys. In general it's possible to use far fewer pins compared to the non-multiplexed scheme.

With the multiplex scheme the rows are activated one after another, all columns are read in and analysed. The result is stored into bit fields. A 8x8 matrix shows a 64bits result. Only one row is activated at one time, the others remain in TriState condition.

In this implementation each column has a PullUp resistor of 1..10kOhm. The impedance of the internal PullUps of the AVR is too large (typ. 80k) for this purpose. Because of the fast scan the positive edges of the columns are too slow. They are charged by the capacitive loads of the CMOS-inputs, wires, keys etc. and false results must be expected. The scanning is done within the SysTick routine, which must be imported for that reason. In order to save code space und execution time a complete input port is used for the column input.

While scanning the keys are debounced. A keys state must be stable for at least 3 SysTicks before a change will be accepted.

Imports

The SysTick driver must be imported.

Import SysTick, KeyPort8, ... ;

By the import of the KeyPort the enumeration "Keys" is automatically imported:

Type Keys = (Key1, Key2, ..., KeyN);
Const lastKeyBoardKey : Keys = KeyN;

Defines

The mode and parameters of the keyboard driver must be defined.

Define KeyB8Row = PortA, 2; {use PortA, start with bit2}
KeyB8Col = PinD; {use PinD complete input port}
KeyB8Type = 4; {4 Rows at PortA, 8 Columns at PinD}
Debounce = 4; {optional debounce in systick counts}

Remarks:

The Row-port must be a bi-directional port.

The Col-port must at least consist of an input-port.

AVRco Standard Driver



The application can process the IO-accesses by itself. But then the Define must be changed:

```
Define KeyB8Type = UserPort, 3;      {3 Rows, 8 Columns}
```

KeyB8Row and KeyB8Col must not be defined in this case. The application now must provide this Call-Back function:

```
UserDevice KeyBoard8IOS : byte;
```

Now the driver internal functions like debouncing and scanning are used where the application feeds them with the result of reading a special, normally not supported hardware.

3.3.1 Exported Variables

Memory Organization

All KeyBoard8 related variables are defined in a row in the memory.

```
KEYBOARD8      .EQU  076h    ; var iData semaphore
_KEYBOARDENA    .EQU  077h    ; var iData boolean
_OLDKB81        .EQU  078h    ; var iData byte
_OLDKB82        .EQU  079h    ; var iData byte
_OLDKB83        .EQU  07Ah    ; var iData byte
_OLDKB84        .EQU  07Bh    ; var iData byte
_KEYB81         .EQU  07Ch    ; var iData byte
_KEYB82         .EQU  07Dh    ; var iData byte
_KEYB83         .EQU  07Eh    ; var iData byte
_KEYB84         .EQU  07Fh    ; var iData byte
_CHANGEKB81     .EQU  080h    ; var iData byte
_CHANGEKB82     .EQU  081h    ; var iData byte
_CHANGEKB83     .EQU  082h    ; var iData byte
_CHANGEKB84     .EQU  083h    ; var iData byte
```

Don't access any of the above vars, except "KeyBoard8".

The var "KeyBoard8" is an 8-bit Semaphore which can be accessed with the corresponding methods. This semaphore contains as information the rough count of key clicks. It is reset with "ClearKeyBoard8". This semaphore is highly recommended if processes are imported to avoid continuous polling of the keyboard. The process sleeps until a key becomes active. Example:

```
process HandleKeyBoard (20, 20 : idata);
begin
  WaitSema (KeyBoard8);
  case GetKey8 of
    Key1 : ...; |
  endcase;
end;
```

The content of the semaphore doesn't show always the real count of key clicks. Sometimes there can be more clicks than the semaphore shows. But it's certain that at least [keyboard8] clicks have occurred.



AVRco Standard Driver

3.3.2 Memory property of the keys

The keyboard driver has a build-in memory function. A clicked key will be stored in a bit until it is read-out and cleared with the “KeyRaised8” or “GetKeyRaised8” functions or a ‘ClearKeyBoard8’ is called. With this a continuous polling of the keys by the user program can be avoided.

Please note that the memory store operation is only executed if a key was pressed and the released. All functions related to the memory feature contain a “RAISED” in their name.

3.3.3 Exported Functions and Procedures

Function ReadKey8 (const key : Keys) : boolean;

The actual state of a key is returned with a true or false. If the key is pressed, the result is true, otherwise it's false. The parameter Key must be of type “Keys”, i.e. Key1..Keyxx.

Function KeyRaised8 (const key : Keys) : boolean;

The state of a key is returned by a true or false. If a key was pressed and released at least once, then a true is returned. There is no information whether the key is again pressed or not.

Function ReadKeyBoard8 (const row : byte) : byte;

The result of this function is a byte where each bit corresponds with a key. The parameter “row” selects an 8bit keyblock where “0” returns the keys 1..8 and “1” the keys 9..16 etc. The result is the static state of the keys and not the latched state.

Function GetKeyRaised8 : Keys;

This function calls “KeyStatRaised8” and if it gets a true, the first latched key found is returned. If no key is latched the function polls KeyStatRaised8 until a latched key is found. This means, if no key can be found the user program never gains the control, because this function loops endless.

Function KeyStatRaised8 : boolean;

The state of the entire KeyBoard is returned with a true or false. If there was at least one key pressed and latched and this key is not read out by “KeyRaised8” etc., the result becomes true. In other words, the function returns a true if a key was pressed and released and is still latched. With this function a complete KeyBoard read-out by many “KeyRaised8” can be avoided until this function returns a true.

Function KeyStat8 : boolean;

The state of the entire KeyBoard is returned with a true or false. If there is at least one key pressed the result becomes true. In other words, the function returns a true if a key is still pressed. With this function a complete KeyBoard read-out by many “GetKey8” can be avoided until this function returns a true.

Function GetKey8 : Keys;

This function calls “KeyStat8” and if it gets a true, the activated key found is returned. If no key is active the function polls KeyStat8 until an active key is found. This means, if no key can be found the user program never gains the control, because this function loops endless.

Procedure ClearKeyBoard8;

This procedure clears the entire keyboard including the memory “KeyRaised8” and the semaphore.

Procedure KeyBoardEnable8 (ena : boolean);

This procedure enables or disables the entire KeyBoard including the Scanner, Timer etc. So it is possible to disable user key operations or with time critical program parts the SysTick latency (interrupt disable duration of the SysTicks) can be temporary reduced to a minimum.

A disable includes a ClearKeyBoard call. If Timer and Pipe are imported they will also be cleared.

3.3.4 Relation between Buttons and Bits

The button on the cross point of Row1 and Col1 is the least significant bit. Then button on the point Row1/Col2 follows etc.

If the matrix for example has 32 buttons so there are 32 bits relevant, this means that the result of a function call can be Key1..Key32. The button on Row1/Col1 returns Key1. If the layout of the keyboard is 8Cols and 4Rows then the button on Row3/Col4 returns the value Key20.

	Row1	Row2	Row3	Row4
Col1	Key1	Key9	Key17	Key25
Col2	Key2	Key10	Key18	Key26
Col3	Key3	Key11	Key19	Key27
Col4	Key4	Key12	Key20	Key28
Col5	Key5	Key13	Key21	Key29
Col6	Key6	Key14	Key22	Key30
Col7	Key7	Key15	Key23	Key31
Col8	Key8	Key16	Key24	Key32

3.3.5 Debouncing

With SysTick < bounce-time it makes sense not to debounce with each SysTick but to do the debouncing in intervals of x SysTicks. For this purpose there is the optional

Define Debounce = nn; // nn = SysTicks

Which imports an internal interval timer. So the debouncing is done every 5th tick for example. This timer then is also responsible for system wide debouncing, including the MatrixPort and the SwitchPorts.

3.3.6 Key Repeat Support

The driver also provides an optional support of external repeat functions.

In order to implement an auto-repeat function of the keys the controlling part of the application must know how long a key is activated.

To support this each key has its own optional 8bit timer that is incremented in each scan interval as long as the key stays active and stable. The timer stops at 255. If the key becomes released the timer will be reset to zero.

Each key has its own timer. These timers are placed into an array and can be read by the application.

If this option must be used the timers must be imported.

Import SysTick, KeyPort8, ... ;

From KeyPort8 import KeyBoardTimer;

This import defines the timer array in this way:

Var KeyboardTimers : array[Keys] of byte;



AVRco Standard Driver

Now the application can access the array without any restrictions:

```
if KeyboardTimers[Key1] > 0 then  
    // key is pressed  
    ...  
endif;
```

or:

```
case KeyboardTimers[Key1] of  
    0 : // key released  
    |  
    1..10 : // key initially activated  
    |  
else  
    // do any repeat action  
endcase;
```

Note:

The timers are incremented in the scan interval. In most cases this is the once per SysTick. If **Debounce** is defined the increments are done in (SysTick * Debounce) intervals.

3.3.7 Full Auto- Repeat Support

The driver also provides an optional internal auto-repeat function. The KeyboardTimer is used to build the Auto-Repeat function. The resulting Key events are then stored into a special pipe **KeyboardPipe**.

In most cases the application does not need any of the standard KeyPort functions. All operations can be done with the Pipe. All of the common pipe function can be used.

If this option must be used then the KeyboardTimers must be imported.

```
Import SysTick, KeyPort8, ... ;
```

```
From KeyPort8 import KeyboardTimer;
```

The Auto-Repeat function must be declared with a Define:

```
Define KeyB8Pipe = PipeLen, FirstRepeat, RepeatRate;
```

Then this variable will be exported:

```
KeyboardPipe : Pipe[PipeLen] of Keys;
```

The first parameter **PipeLen** defines the length of the pipe and hence the number of keystrokes which can be stored in it.

The second parameter **FirstRepeat** defines the time in msec elapsed before the repeat starts.

The third parameter **RepeatRate** defines the time in msec between two repeats.

If a key is actually pressed and the debounce time has elapsed the key is immediately written into the pipe. If the key remains active the auto-repeat starts after the **FirstRepeat** time has elapsed. The repeat writes in **RepeatRate** msec intervals into the pipe. Each time the ID of this key is written into the pipe. The auto-repeat works until either this key is released or the pipe is full.

AVRco Standard Driver



If any key becomes inactive the corresponding Key-ID + \$80 (8th bit set) is written into the pipe.
The application can access the pipe at any time with:

```
if PipeStat (KeyboardPipe) <> 0 then  
  key:= PipeRecv (KeyboardPipe);  
  ...  
endif;
```

3.3.7.1 Support Functions

The auto-repeat operation of the driver can be enabled and disabled at runtime with:

Procedure *KeyBoardRepeat* (*rept : boolean*);

Basically in addition all KeyPort8 functions can be used like ClearKeyBoard8. But in most cases this is not necessary.

Also all the pipe related functions can be used like PipeFlush etc. It is also possible to write directly into the keyboard buffer with PipeSend, which was often used in the old MSDOS days.

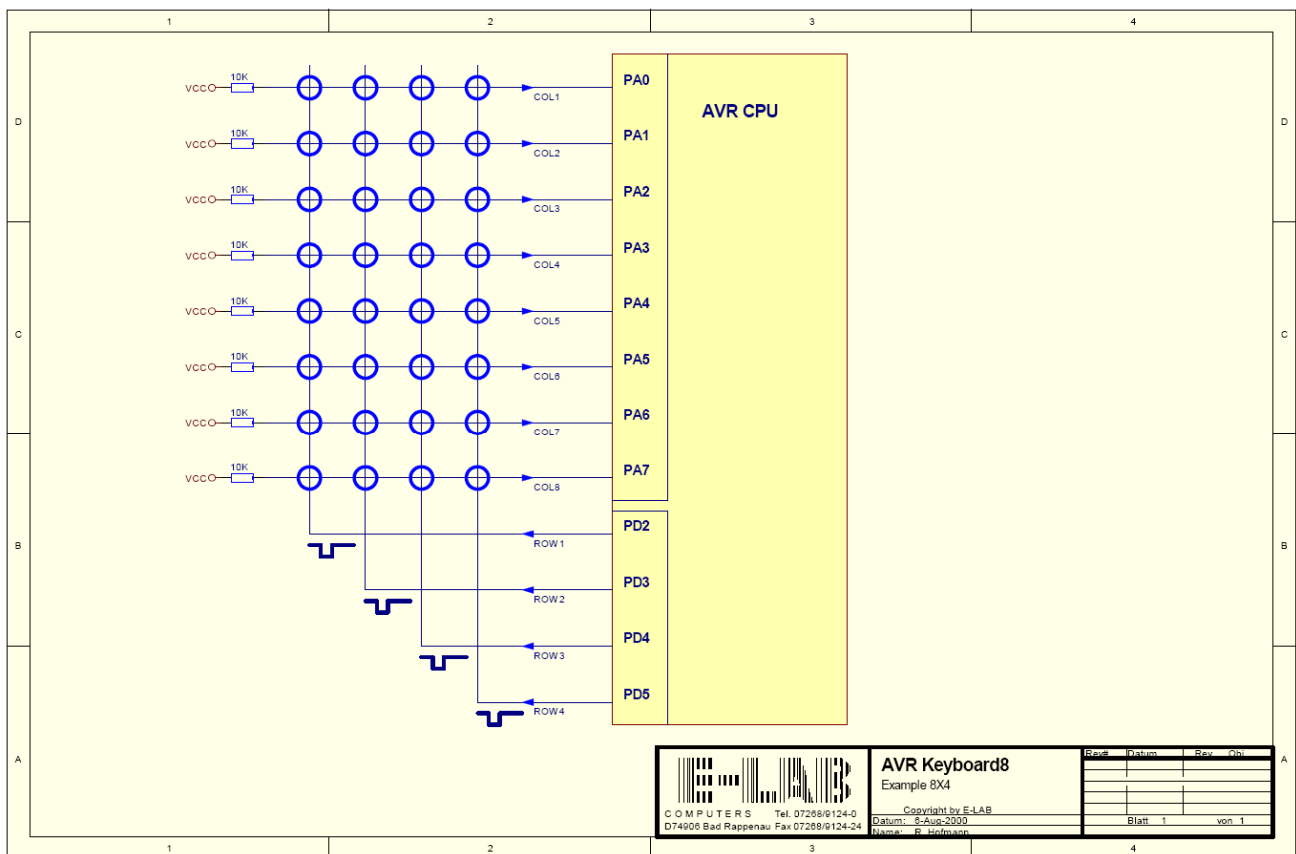
Attention:

With Auto-Repeat each key has an 8bit Timer. This is an additional memory usage of up to 64 Bytes.

Because all operations (scan, debounce, Timer, Pipe and repeat) are in the SysTick, the SysTick is again heavily loaded. For each 8 key group there is a consumption of about 18usec@16MHz. Worst case (64 keys) this is about 150usec. This must be added to the normal interrupt disable time in the SysTick when the debounce counter elapsed. So the CPU should run at 16MHz or faster.

Example program and schematic:

an example is in the directory **..\\E-Lab\\AVRco\\Demos\\KeyBoardAuto8**



schematic **KeyBoard8**



AVRco Standard Driver

3.3.8 User Device and KeyBoard8

If an application cannot use the standard ports of an AVR for the KeyBoard-IO it's possible for the user program to do its own special KeyBoard accesses. The driver controls the scanning and does the debouncing.

Import SysTick, KeyPort8, ... ;

Define

```
ProcClock = 16000000;    {Hertz}
SysTick    = 10;
StackSize  = $0064, iData;
FrameSize  = $0064, iData;
KeyB8Type  = KeyB8Type = UserPort, 5; {5 Rows, 8 Columns}
```

Implementation

```
{ $NORETURNCHECK }
UserDevice KeyBoard8IOS : byte;
Begin
  ASM;
  ...
  ...
EndASM;
end;
```

The driver passes the register _ACCA (R17) and _ACCB (R16) to the UserDevice function. Register _ACCA contains the actual ROW number as 0, 1, 2, 3, 4 etc. As an alternative the register _ACCB contains the ROW number as a bit position \$1, \$2, \$4, \$8, \$10, \$20 etc. Dependant on the construction of the external hardware _ACCA or _ACCB can be used to read-in a column (COL) . With a multiplexer _ACCA is preferable, with a latch or similar _ACCB is suitable better.

The application must return in _ACCA the column read result to the calling driver.

The program code must be written in assembler in order to be as fast as possible because this function is called from the SysTick. Because of this only _ACCA and _ACCB can be used for the IO-processing. If additional registers must be used they must be saved with a PUSH and later restored with a POP.

Local variables are forbidden and so are calling system drivers or mathematical operations of the system.

3.4 LCD-Display

3.4.1 LCDPORT

Import and definition of a LCD-port. 1-, 2- or 4-lined displays can be connected. Also versions with 2 enables or two separate modules are possible. Only HD44780, HD66712, KS0070 and KS0073 or compatible controllers are supported.

The LCD-display operates in the 4bit-mode. The lower 4bits of the chosen port are the data bits D4..D7 of the display, port-bit4 is the control signal *E*, port-bit5 is the signal *RS* and port-bit6 is the signal *RW*. Port-bit7 remains unused, unless a second enable signal is needed. In this case bit7 becomes Enable2.

When choosing the port, be aware that this port has to work bi-directionally.

```
Import SysTick, LCDport;
Define ProcClock    = 4000000;    {4Mhz clock }
        SysTick      = 10;         {10msec Tick}
        LCDtype       = 44780;     {66712}
        LCDport       = PortA;     {Port Address}
        LCDRows       = 2;         {2-lined display}
        LCDcolumns    = 16;        {16-character display}
```

Attention:

If the LCD-display is imported, a functioning display should be connected correctly, otherwise each access to the LCD takes a long time for the functions to return with a timeout.

For test and debug purpose the polling of the BUSY-Flag of the controller can be disabled by **{\$LCDNOWAIT}**. This switch must be removed when the program is run in a real chip, otherwise the program will not work because the display controller is overrun.

LCDTYPE

This definition can be omitted if the controller type 44780 is used. The type 66712 is somewhat different when using LCDxy and therefore it must be defined explicitly if used.

The LCD line displays equipped with the Samsung controller types KS0070, KS0073 are software compatible with the HD44780 but extremely different with their timings. Because of this the exact type of controller must be declared. Some displays with KS0073 can also be operated with a SPI:

```
Define
        LCDtype       = 0073;      {KS0073}
        LCDrows       = 4;         {4 rows}
        LCDcolumns    = 20;        {20 characters/line}
        LCDport       = SPI_Soft, PortB, 1, 2, 3, 0; // PortX, SCK, MOSI, MISO, SS
        // LCDport    = SPI;       // Hardware SPI
        // LCDport    = SPI_C, PortB, 1; // Hardware SPI, SS_port, SS_pin X Mega
```

LCDROWS

Number of lines of the LCD-display (1, 2, 4). It is required for initialisation. Big display units sometimes offer an additional Enable input. This second control signal, if present, must be declared as the second parameter.

```
LCDRows = 4, 2;    {4-row Display with 2 Enables}
```

LCDCOLUMNS

Number of characters per line of the LCD-display (8, 12, 16, 20, 24, 40). It is required for calculating the physical address in the procedure LCDxy.

```
LCDcolumns = 16;    {16 characters per line}
```



AVRco Standard Driver

3.4.2 Functions and Procedures

LCDout

Writes to the LCD DD-Ram and/or CG-Ram with $RS=1$ and $RW=0$.

Output of the data at the display. Passing parameter is of the type byte or char. Example:

```
LCDout ('A');           {indicate char A}  
Write (LCDout, IntToStr(i));
```

LCDinp

Reads of the LCD DD-Ram and/or CG-Ram with $RS=1$ and $RW=1$. The result is of the type byte.

```
c:= LCDinp;
```

LCDctrl

Writes to the LCD-Control Port with $RS=0$ and $RW=0$.

Allows control words to be output to the display, for example *DispClear*, *ReturnHome*, *OnOff*, *CursorPos* etc.

```
LCDctrl ($01);           {Clear Display}  
LCDctrl ($02);           {Cursor Home}  
LCDctrl ($08);           {Display off}  
LCDctrl ($41);           {CursorPos 1}
```

LCDstat

Reads an LCD Status Port with $RS=0$ and $RW=1$.

Allows the state of the display to be interrogated. Bit7 contains the *BusyFlag* and bit 0..6 contain the cursor or DD-Ram address. LCDstat exists primarily to get the actual cursor address. The *BusyFlag* must not be interrogated for the above LCD operations (LCDout, LCDinp, LCDstat), because they are doing it implicit themselves.

```
repeat until (LCDstat and $80) > 0; {not necessary}  
if (LCDstat and $7f) > 15 then  
    LCDctrl (2);           {Cursor Home}  
endif;
```

A Time-out is implemented in the LCD-driver. With a defective or missing display each access is aborted after approx. 2msec. If a time out occurs the function LCDstat returns a byte where bit 7 is high if the busy-bit of the LCDport has a weak Pull-up.

LCDlower, LCDupper

Big multi-row Displays generally need a second enable control for the second controller on the display board. If such a line is present on the display, it must be declared at the definition of **LCDrows** as the second parameter. The normally unused Bit7 of the control port of the CPU is now used as the second enable control line of the displays. The same is true if two separate modules are used. Both parts are connected in parallel, except for the 2 enables.

The Compiler now offers 2 additional procedures *LCDupper* and *LCDlower*. These procedures define which of the 2 Enable-controls become active on all subsequent LCD-accesses. LCDupper ensures that EnableControl1 (PortBit4) is used; LCDlower ensures that EnableControl2 (PortBit7) is used.

In reality two Enable-control behave as two completely independent controllers in the display. The display has to be considered as two separate units on one board!!

All settings, positioning, read- and write accesses etc. are only effective for the controller which is enabled.

AVRco Standard Driver



As an alternative to one big display with 2 enables it's possible to connect two smaller separate displays in parallel with separate enables. These displays must be of the same type.

Procedure *LCDupper;*

Procedure *LCDlower;*

LCDxy

Positioning of the Cursors to column[x] and row[y]. Both values count from "0".

Procedure *LCDxy (column, row : byte);*

LCDxy (0, 0); (* Cursor to line 0 and column 0 *)

LCDxy (2, 1); (* Cursor to line 1 and column 2 *)

LCDcursor

Definition of the Cursor mode. The cursor can be set disabled, static on or flash. The first parameter switches cursor on or off, the 2nd parameter sets flash on/off.

Procedure *LCDcursor (on, blink : boolean);*

LCDcursor (false, false); (* Cursor off *)

LCDcursor (true, false); (* Cursor on, no blink *)

LCDcursor (true, true); (* Cursor on, blink *)

LCDgetXY

Read back the actual cursor position:

The High-Byte of the result then contains the Y-position (line) and the Low-Byte contains the X-position (column).

Function *LCDgetXY : word;*

LCDclr

Deletes the contents of the display and moves the Cursor to position 0,0

Procedure *LCDclr;* (* erase display, cursor to 0, 0 *)

LCDclrLine

Clears the desired Line(0..n) and places the Cursor to the start of the line

Procedure *LCDclrLine (Line : byte);* (*erase current line to end *)

LCDclrEol;

Clears the current line from cursor position to line end. The cursor position is unchanged.

Procedure *LCDclrEOL;* (* erase current line to end *)

LCDhome

Moves the Cursor to position 0,0

Procedure *LCDhome;* (* cursor to 0, 0 *)

LCDoff

Sets the display off, contents, positions and mode stay unchanged.

Procedure *LCDoff;* (* switch display off *)



AVRco Standard Driver

LCDon

Sets the display on and sets cursor to ON and BLINK

***Procedure** LCDon (* switch display on *)*

LCDcharSet

LCDcharSet stores a self defined graphic character into an unused location [0..15] in the Character Generator Ram of the display module. The character can be displayed with the corresponding address. To define a special char the user must supply the desired address or the ordinal value of this character. Then follow 8 bytes which represent the 8 horizontal pixel rows from top to bottom. Because the character width is 5 pixel, only the 5 lsb's are relevant. The Bit0 represents the right most pixel.

***Procedure** LCDcharSet (loc : char; b1, b2, b3, b4, b5, b6, b7, b8 : byte);*

Example how to generate and display the special char "degree" with the ordinal number of 10.

LCDcharSet (#10, \$1C, \$14, \$1C, \$00, \$00, \$00, \$00, \$00);

LCDout (#10); (display user defined degree char *)*

LCDcharSetP

The storing of special user chars into the LCD controller is supported by an additional and more common function. In contrary to the existing function character defines (array[0..7] of byte) can be placed into RAM, ROM or EEprom. The pointer must point to any structure which contains 8 bytes. "srcArea" defines the kind of source: 0=RAM, 1=EEPROM, 2=FLASH.

***Procedure** LCDcharSetP (const loc : char; srcArea : byte; ptr : pointer);*

{ \$LCDNOINIT }

This compiler switch disables the automatic initialisation of the LCD controllers after a program start or reset. The application decides later at runtime, whether or when an initialisation takes place by invoking the system procedure "LCDsetup".

LCDsetup

This procedure initialises the LCD-controller on condition that the compiler switch { \$LCDNOINIT } is activated before the define block. Then after the calling of "LCDsetup" the display becomes accessible.

Attention:

The compiler switch and this procedure work together and have to be used together. This construction is an **Option** and only relevant for special purpose.

3.4.3 LCD-Split

The connection of a LCD display with the standard driver "LCDport" is limited to one port. So all control and data lines must be connected to one port. But sometimes there is no complete port left and so it is necessary to split the lines over two ports. One port can provide the control lines and another one can handle the 4 data lines.

The standard driver definition is this:

```
LCDport = PortC;           // PortC 0..6(7) = D4, D5, D6, D7, E, RS, RW (, E2 opt.)
```

With port splitting the define must be extended:

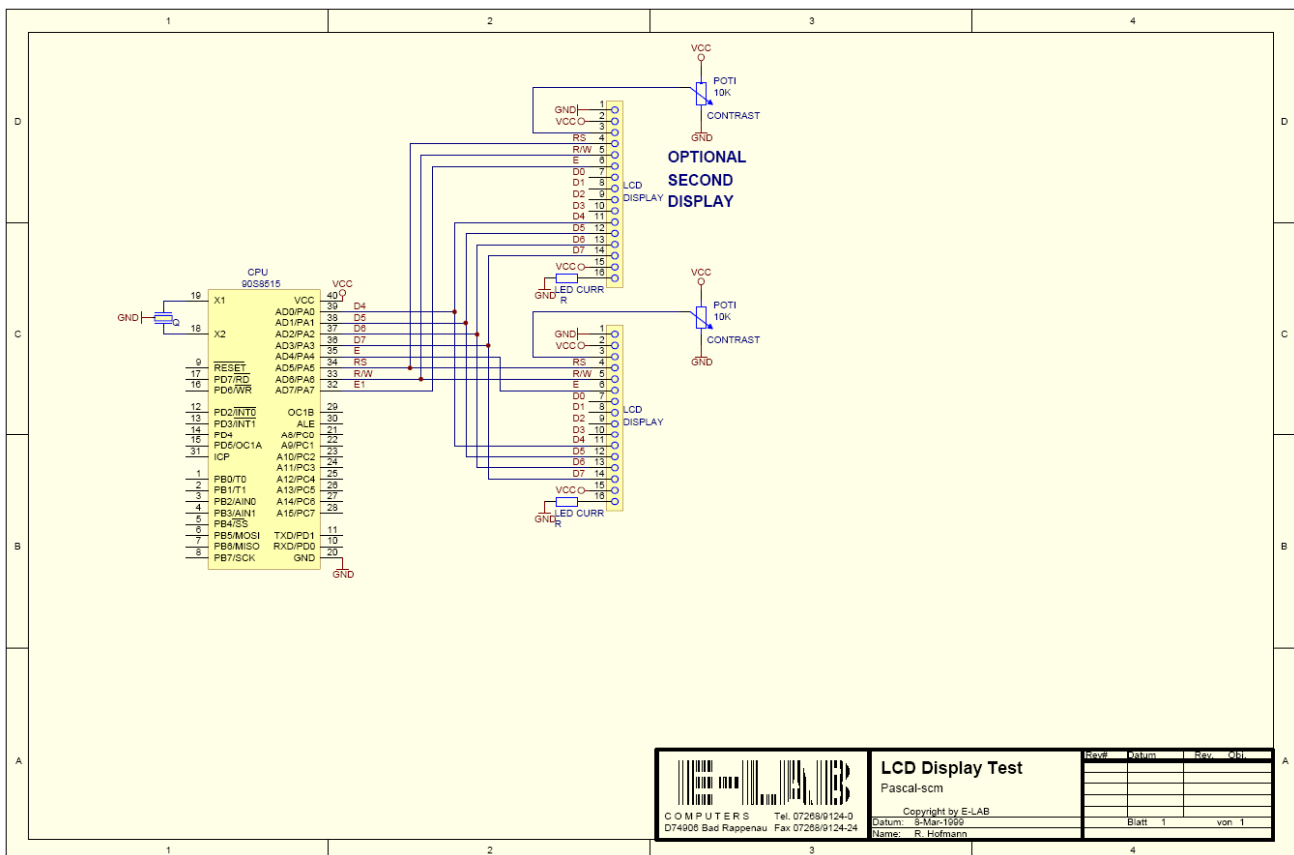
```
LCDport = PortC, 2, PortA, 3; // control port, bit, Data port, bit
```

The first parameter defines the control port with its first used bit. The order of the bits are fixed but **different to the non-split driver**: RS, RW, E (, Enable2 optional).

The data port must come next with its first used bit. The order of the bits also is fixed: D4, D5, D6, D7.

Example program and schematic:

an example is in the directory `..\E-Lab\AVRco\Demos\LCDsplit`



schematic LCD

3.4.4 User defined LCD-IO (LCDUserPort)

As an option the user can also define his own hardware driver

Example program:

An example is in the directory `..IE-Lab\AVRco\Demos\LCD_IOS`

With the import of the LCDUserPort it is necessary that the user must provide the entire display initialisation within the obligatory driver "UserDevice LCDIOS".

If the display operates in 8bit mode, now it is possible to call the system function "LCDSETUP" which does the initialisation of the display thru "LCDIOS".

To use this the compiler switch {\$LCDNOINIT} must be activated in order to disable the automatic initialisation.

3.4.5 User defined LCD characters

The AVRco system contains a specific character generator tool "LCDCharEd.exe" which is intended for designing special chars not present in the character generator of the LCD controller.

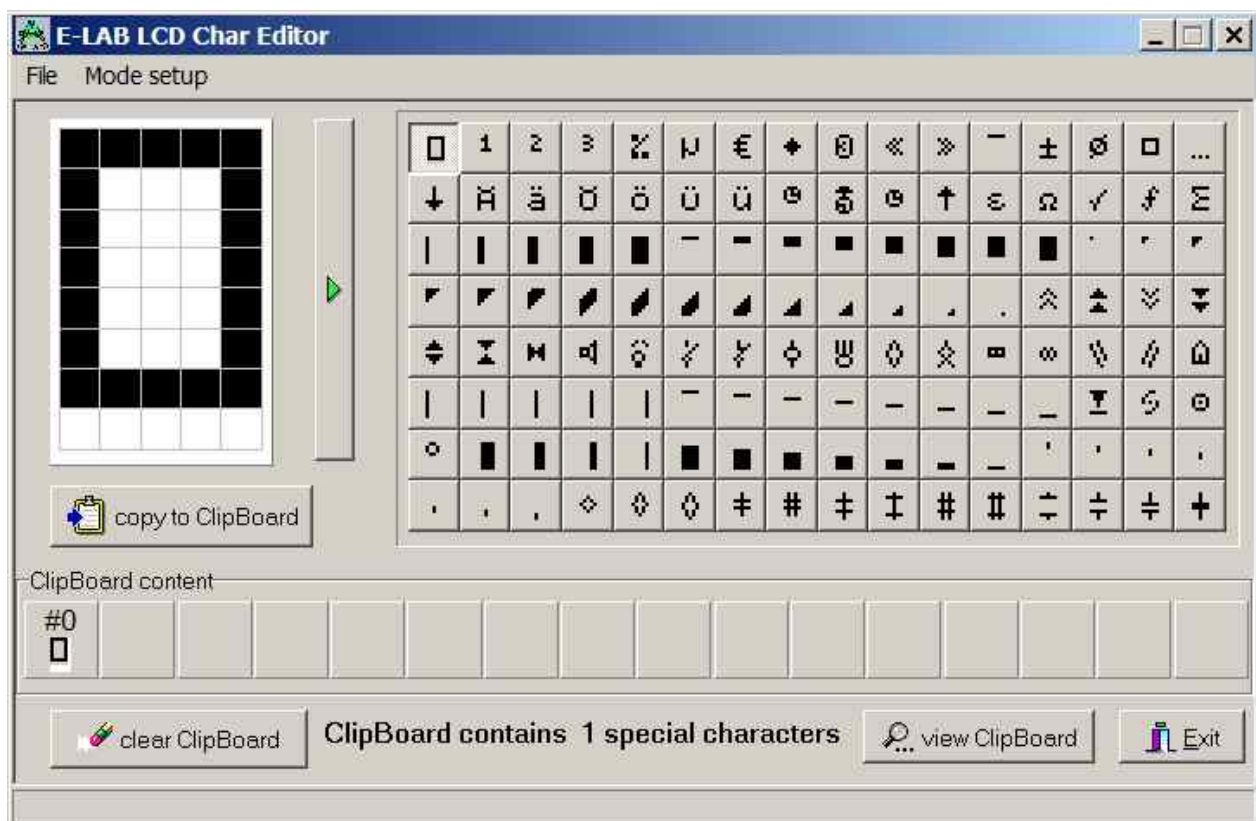
With this tool characters can be interactively designed and the necessary code can be copied into the source with "Shift/Insert".



This tool is also available in the tool bar of the IDE.

The generated special chars can be processed with the Procedure LCDcharset.

The charsets build with this tool can be stored onto the disk with the extension ".lchr" for future use.



3.5 LCD BarGraph Driver

Basics

The representation of values on a display (LCD, 7seg etc) is usually numeric, so measurement results etc. can be displayed in their full precision. The numeric display mode also has some big drawbacks. The number must be interpreted and weighted in the head of the viewer. Finding a trend, e.g. raising or falling, as well needs more computing power from the viewer.

A graphical representation is much more imprecise but the viewer has much less analysis to do than with the numeric one. This is true especially with the evaluation of large or small changes, rising or falling. Here there is a big advantage for the graphical representation.

Graphical representation of numeric values usually are realized with pie charts or bar graphs. With simple displays like the LCDs only bar graphs can be used.

Introduction to LCD BarGraph

This implementation either uses the driver LCDport or the driver LCDmultiPort. To use the BarGraph either the driver **LCDport** or the driver **LCDmultiPort** must be imported. The standard LCDport driver can coexist besides the MultiPort driver.

Up to 4 bar graphs can be defined where each graph has its own "coordinate" system:

Y-Position	line
X-Start	character
X-Len	character
Scale	byte

Each graph has to be connected logically to a LCD driver by its define statement. It's possible to route the graphs 1..3 to the LCDmultiPort and the graph 4 to the LCDport.

With the LCDmultiPort a graph can be placed at runtime into each of the 8 possible physical displays.

For displaying the Bar Graphs the first 5 of 8 user definable characters (#0..#4) of the displays are used by the driver. Each display that must show a Bar Graph then must be set up by a special BarGraph Init. These user characters then are used by the system and must not be overwritten by the user's application.

The driver accepts only bytes as control values. Such a byte then is recalculated by using the graph's scaling and coordinate factors and the result is displayed by the BarGraph.

Attention:

Some exotic LCD controllers (e.g. Samsung S6A0032) are basically compatible to the standard Hitachi controller 44780 but support none or only 2 free definable characters. So there is no way to use this driver for implementing a bar graph with these controllers. In this case the application must implement it itself by using the block character #255 (\$FF).

Imports

As usual with the AVRco system the driver must be defined. In addition the desired LCD driver must be imported and defined.

Either import the LCD driver **LCDmultiPort** and the I2C/TWI driver or import the **LCDport** driver. SysTick is not used.



AVRco Standard Driver

```
Import I2Cport, LCDmultiPort;  
or  
Import TWImaster, LCDmultiPort;  
or  
Import TWInet, LCDmultiPort;           // use Master mode  
or  
Import LCDPort;
```

Defines

Depending on the I2C or TWIport, this must be defined. As with the standard LCDport, the LCD type used must be defined.

Example for I2Cport:

```
Define ProcClock      = 8000000;      {8Mhz clock }  
        I2Cport        = PortC;        {port used}  
        I2Cdat          = 7;           {bit7-PortC}  
        I2Cclk          = 6, 4;         {bit6-PortC, optional delay 4}  
        LCDmultiPort    = I2C_Soft;     {use Software I2Cport}  
        LCDTYPE_M       = 66712;       {LCD controller type}  
        LCDrows_M       = 2;           {2 rows}  
        LCDcolumns_M    = 20;          {20 chars per line}
```

Example for TWImaster:

```
Define ProcClock      = 8000000;      {8Mhz clock }  
        TWIpresc       = TWI_BR100;    {100kBit/sec alt. TWI_BR400}  
        LCDmultiPort   = I2C_TWI;      {use TWIport}  
        LCDTYPE_M      = 44780;        {LCD controller type}  
        LCDrows_M      = 4;            {4 rows}  
        LCDcolumns_M   = 16;           {16 chars per line}
```

Example for TWInetMaster:

```
Define ProcClock      = 8000000;      {8Mhz clock }  
        TWInode        = 05;           {default address in slave mode}  
        TWIpresc       = TWI_BR400;    {400kBit/sec alt. TWI_BR100}  
        TWIframe       = 4, iData;     {buffer/packet size}  
        TWIframeBC     = 6;            {option broadcast buffer/packet size}  
        TWInetMode     = Master;       {use TWIport}  
        LCDmultiPort   = I2C_TWI;      {use TWIport}  
        LCDTYPE_M      = 0070;         {LCD controller type}  
        LCDrows_M      = 1;            {1 rows}  
        LCDcolumns_M   = 12;           {12 chars per line}
```

Example for LCDport:

```
Define ProcClock      = 8000000;      {8Mhz clock }  
        LCDport        = PortB;        {use Port B}  
        LCDTYPE        = 0070;         {LCD controller type}  
        LCDrows        = 2;            {2 rows}  
        LCDcolumns     = 20;           {20 chars per line}
```

After the complete Import and Define of the LCD driver the Define of the LCD BarGraphs must follow:

```
Define ProcClock = 8000000;           {8Mhz clock }  
...  
...  
LCDBargraph1 = LCDmultiPort;  
LCDBargraph2 = LCDmultiPort;  
LCDBargraph3 = LCDmultiPort;  
LCDBargraph4 = LCDmultiPort;  
// LCDBargraph4 = LCDPort;
```

LCDBargraph1..4

Imports the coordinate system for a BarGraph and defines which type of LCD driver has to be used with it. The required basic driver must also be imported and defined.

3.5.1 Functions

The definition of a bar graph publishes one/two LCD dependent common init functions and two BarGraph functions:

LCDbarInit_P

Init for the standard **LCDport**. Each display must be initialised once. With the driver LCDport this is done internally at Power On and CPU-Reset. In order to show the 5 Bar graph special characters they must be stored into the character RAM of the LCD controller. This procedure loads these characters:

Procedure LCDbarInit_P;

LCDbarInit_M

Init for the **LCDmultiPort**. Each display must be initialised once. Unlike the standard driver LCDport this not automatically done at Power On or CPU-Reset. The application must initialise the display with **LCDsetup_M**. After this the Bar Graph initialisation can follow. In order to show the 5 Bar graph special characters they must be stored into the character RAM of the LCD controller. This must be done with this procedure after the display initialisation because LCDbarInit_M has no display-num parameter to set the target display. The current active display is always used. If there are doubts about this one can use the procedure LCDsetPort_M to select the desired display.

Procedure LCDbarInit_M;

LCDbarSet1..4

Before a Bar graph can be used the special character set must be loaded with the procedure **LCDbarInit** described above. Also the coordinate system of a Bar Graph must be initialised at least once with LCDbarSet. The parameters of this function determine the line, position and size of the Bar Graph on a display.

Procedure LCDbarSet1 (**const** Line, PosA, Len, Scal : byte);
LCDbarSet1 (0, 8, 8, 100);

Line defines the display line 0..3

PosA defines the X-start position, in characters

Len defines the total graph length in characters

Scal defines the scaling, i.e. the right end value of the Bar Graph

The sum of PosA + Len must not exceed the line length (columns) of the display.
The value of Scal should be in the range of 50..255 to get best results.



AVRco Standard Driver

LCDbarOut1..4

Changes the Bar Graph to the new value and displays it. The parameter passed is scaled based on the coordinate system of this Bar Graph.

With the LCDmultiPort the actual display is always rewritten. If this is unknown one can use the procedure *LCDsetPort_M* to select the desired display.

*Procedure LCDbarOut1 (const b : byte);
LCDbarOut1 (50);*

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\LCD_BarGraph`

3.6 LCDmultiPort Driver with up to 8 LCDs

Basics

Usually alpha-numeric LCD-displays are directly connected to a port of the CPU and work in the 4- or 8bit mode. With two or more displays the number of required port pins easily exceed the available pin count of a CPU. In addition the necessary cables with at least 10 lines/LCD require a lot of connectors on the board.

The solution of this problem is to use LCDs with a Bus-system, mainly I2C. But these displays are not cheap and also there are only a few types available.

If we don't use intelligent displays with I2C bus we "simply" have to extend the count of the ports of the CPU by I2C I/O-expander chips. So each expander chip can handle one LCD device. Only 2 port pins of the CPU are used, regardless how much displays are connected.

Introduction to LCDmultiPort

This implementation either uses the software I2C-driver (I2Cport) or the internal TWI (I2C) port of the AVR mega CPUs. To use the LCDmultiPort either the driver *I2Cport* or the driver *TWImaster* or the driver *TWInet* must be imported, the latter in Master mode. The standard LCDport driver can coexist with the MultiPort driver. With the **XMegas** instead of the TWI one of the TWI ports must be selected: TWI_C, TWI_D, TWI_E or TWI_F.

As the I2C port-expander one PCA9555 from Philips must be used for one LCD. This chip can be present up to 8 times on the I2C bus. The wiring between the PCA9555 and LCD is fixed by the driver and can be found in the schematic below. The PCA9555 can run up to 400kBit/sec on the I2C Bus. Unlike its predecessors each of the two 8bit ports can be directly read, written or reprogrammed without any affect on the other parts of the chip.

The driver LCDmultiPort controls its remote ports PCA9555 in such a way that the LCDs are directly connected in the 8bit mode to a port of the CPU.

The driver's functions are very similar to the functions of the driver *LCDport* but with the extension that with each function call the desired display number is expected (exception: LCDout).

3.6.1 Technical data

I2C Port	Software I2C imported by I2Cport	
or	CPU-TWI imported by TWImaster	// Mega
or	CPU-TWI imported by TWInet in Master mode	// Mega
or	CPU-TWI imported by TWI_C, TWI_D, TWI_E, TWI_F	// XMega
Displays	up to 8 devices, controller 44780, 66712 or KS0070 8bit mode, only one enable line possible. All displays must be of the same type: controller, line count, column count.	
Hardware	I2C I/O-Expander chip PCA9555 from Philips, 1 piece per display	

I2C addresses The PCA9555 reside at the bus-addresses \$20..\$27

where LCD1 has the address \$20, LCD2 has \$21 etc.

The PCA9555 have three address pins or bits which must be wired in the correct way.



AVRco Standard Driver

Imports

As usual with the AVRco system the driver must be imported and defined. In addition the desired I2C/TWI driver must be imported and defined. SysTick is not used. All functions return after a timeout if no display is available.

```
Import    I2Cport, LCDmultiPort;
or
Import    TWImaster, LCDmultiPort;
or
Import    TWInet, LCDmultiPort;           // use Master mode
```

XMega

```
Import    TW_C, TWI_E, LCDmultiPort;
```

Defines

Dependent of the I2C or TWIport, this must be defined. As with the standard LCDport, the LCD type used must be defined.

Example for I2Cport:

```
Define ProcClock      = 8000000;      {8Mhz clock }
        I2Cport        = PortC;        {port used}
        I2Cdat          = 7;           {bit7-PortC}
        I2Cclk          = 6, 4;        {bit6-PortC, optional delay 4}
        LCDmultiPort    = I2C_Soft;    {use Software I2Cport}
        LCDTYPE_M       = 66712;      {LCD controller type}
        LCDrows_M       = 2;          {2 rows}
        LCDcolumns_M    = 20;         {20 chars per line}
```

Example for TWImaster:

```
Define ProcClock      = 8000000;      {8Mhz clock }
        TWIpresc       = TWI_BR100;    {100kBit/sec alt. TWI_BR400}
        LCDmultiPort   = I2C_TWI;      {use TWIport}
        LCDTYPE_M      = 44780;        {LCD controller type}
        LCDrows_M      = 4;            {4 rows}
        LCDcolumns_M   = 16;           {16 chars per line}
```

Example for TWInetMaster:

```
Define ProcClock      = 8000000;      {8Mhz clock }
        TWInode        = 05;           {default address in slave mode}
        TWIpresc       = TWI_BR400;    {400kBit/sec alt. TWI_BR100}
        TWIframe       = 4, iData;     {buffer/packet size}
        TWIframeBC     = 6;            {option broadcast buffer/packet size}
        TWInetMode     = Master;
        LCDmultiPort    = I2C_TWI;      {use TWIport}
        LCDTYPE_M      = 0070;         {LCD controller type}
        LCDrows_M      = 1;            {1 rows}
        LCDcolumns_M   = 12;           {12 chars per line}
```

Example for **XMega**:

```
Import    TWI_C;
Define    OSCtype      = int32MHz, PLLmul=4, prescB=1, prescC=1;
        TWIprescC      = TWI_BR100;    {100kBit/sec alt. TWI_BR400}
        LCDmultiPort   = TWI_C;        {use TWIport}
        LCDTYPE_M      = 44780;        {LCD controller type}
        LCDrows_M      = 4;            {4 rows}
        LCDcolumns_M   = 16;           {16 chars per line}
```

LCDmultiPort

Requests the I2C-port to use, either software-I2C with *I2C_Soft* or on chip TWIport with *I2C_TWI*. The required basic driver must also be imported and defined.

LCDTYPE_M

Defines the LCD-controller type HD44780, HD66712, KS0070 or KS0073

LCDrows_M

Defines the line or row count of the display 1, 2 or 4

LCDcolumns_M

Defines the column count (characters/row) of the display 8, 12, 16, 20, 24 or 40.

3.6.2 Types and Functions

The import of LCDmultiPort exports an enumeration type which must be used for the driver's function calls:

Type *TLCD_num* = (*LCD_m1*, *LCD_m2*, *LCD_m3*, *LCD_m4*, *LCD_m5*, *LCD_m6*, *LCD_m7*, *LCD_m8*);

LCDsetup_M

Each connected display must be initialised at least once. Unlike the standard driver *LCDport* this doesn't happen at Power On or CPU-Reset. After this initialisation the display stays in the off-mode. The display must be switched on with function *LCDcursor_M*.

If successful LCDsetup_M returns with a true, otherwise a false is returned.

Function *LCDsetup_M* (**const** *LCD_num* : *TLCD_num*) : *boolean*;

If not *LCDsetup_M* (*LCD_m1*) **then** ...

Error ...

Endif;

LCDon_M

Switches the display ON and sets the cursor to ON and BLINK

Procedure *LCDon_M* (**const** *LCD_num* : *TLCD_num*);

LCDon_M (*LCD_m6*);

LCDcursor_M

Switch the display ON and setup of the cursor mode. The cursor can be set off, static on or flash. The 2nd parameter switches the cursor ON or OFF, the 3rd parameter sets flash on/off.

Procedure *LCDcursor_M* (**const** *LCD_num* : *TLCD_num*; *on*, *blink* : *boolean*);

LCDcursor_M (*LCD_m2*, *false*, *false*); (* Cursor off *)

LCDcursor_M (*LCD_m3*, *true*, *false*); (* Cursor on, no blink *)

LCDcursor_M (*LCD_m4*, *true*, *true*); (* Cursor on, blink *)

LCDout_M

Writes into the LCD DD-ram or CG-Ram and displays this data. The parameter is of type char. Because this procedure also must interact with *Write*, there is no display select parameter *LCD_num*. The writing to the display is always directed to the currently selected one. The desired display can also be explicitly set with *LCDsetPort_M*.

Procedure *LCDout_M* (**const** *c* : *char*);

LCDout_M ('A'); {show char A}

write (*LCDout_M*, *IntToStr* (*i*)); {string output}

LCDclr_M

Deletes the contents of the display and moves the Cursor to position 0,0

Procedure *LCDclr_M* (**const** *LCD_num* : *TLCD_num*);

LCDclr_M (*LCD_m5*);



AVRco Standard Driver

LCDclrEOL_M

Deletes the actual line from cursor position to line end. The cursor position stays unchanged.

Procedure *LCDclrEOL_M* (**const** *LCD_num* : *TLCD_num*);
LCDclrEOL_M (*LCD_m6*);

LCDclrLine_M

Deletes the passed line number (0..n) and moves the cursor to the line start position.

Procedure *LCDclrLine_M* (**const** *LCD_num* : *TLCD_num*; **const** *line* : *byte*);
LCDclrLine_M (*LCD_m6*, 1);

LCDhome_M

Moves the cursor to position 0,0

Procedure *LCDhome_M* (**const** *LCD_num* : *TLCD_num*);
LCDhome_M (*LCD_m7*);

LCDxy_M

Positioning of the Cursors to column[x] and row[y]. Both values count from "0".

Procedure *LCDxy_M* (**const** *LCD_num* : *TLCD_num*; *x*, *y* : *byte*);
LCDxy_M (*LCD_m8*, 0, 0); (* Cursor to line 0 and column 0 *)
LCDxy_M (*LCD_m1*, 2, 1); (* Cursor to line 1 and column 2 *)

LCDgetXY_M

Read back the position of the cursor. The line (y) is in the high-byte and column (x) in the low-byte of the result.

Function *LCDgetXY_M* (**const** *LCD_num* : *TLCD_num*) : *word*;

LCDoff_M

Sets the display OFF. Contents, positions and mode stay unchanged.

Procedure *LCDoff_M* (**const** *LCD_num* : *TLCD_num*);
LCDoff_M (*LCD_m2*);

LCDsetPort_M

Optional, switches the driver to a specific LCD. Only makes sense with *LCDout_M*.

Procedure *LCDsetPort_M* (**const** *LCD_num* : *TLCD_num*);
LCDsetPort_M (*LCD_m2*);

LCDgetPort_M

Optional. The function returns the current selected LCD-Port.

Function *LCDgetPort_M* : *TLCD_num*;

LCDctrl_M

Optional low-level function. Writing to the LCD-ControlPort with *RS=0* and *RW=0*.

Used to output control words to the display, for example *DispClear*, *ReturnHome*, *OnOff*, *CursorPos* etc.

Procedure *LCDctrl_M* (**const** *LCD_num* : *TLCD_num*; **const** *b* : *byte*);
LCDctrl_M (*LCD_m1*, \$01); {Clear Display}
LCDctrl_M (*LCD_m2*, \$02); {Cursor Home}
LCDctrl_M (*LCD_m3*, \$08); {Display off}
LCDctrl_M (*LCD_m4*, \$41); {CursorPos 1}

LCDinp_M

Optional low-level function.

Reads the LCD DD-Ram and/or CG-Ram with *RS=1* and *RW=1*. The result is of the type byte.

Function *LCDInp_M* (**const** *LCD_num* : *TLCD_num*) : *byte*;
b := *LCDInp_M* (*LCD_m5*)

LCDstat_M

Optional low-level function. Reads an LCD Status Port with *RS*=0 and *RW*=1. Interrogates the state of the display. Bit7 contains the *BusyFlag* and bits 0..6 contain the cursor or DD-Ram address. *LCDstat_M* exists primarily to get the actual cursor address. The *BusyFlag* must not be interrogated for the above LCD operations (*LCDout*, *LCDinp*, *LCDstat*), because they do it implicitly themselves.

Function *LCDstat_M* (**const** *LCD_num* : *TLCD_num*) : *byte*;
repeat until (*LCDstat_M*(*LCD_m1*) **and** \$80) > 0; {not necessary}
if (*LCDstat_M* (*LCD_m1*) **and** \$7f) > 15 **then**
 LCDctrl_M (*LCD_m1*, 2); {Cursor Home}
endif;

LCDcharset_M

LCDcharset stores a self defined graphic character into an unused location [0..15] in the Character Generator Ram of the display module. The character can be displayed using the corresponding address. To define a special char the user must supply the desired address or the ordinal value of this character. Then follow 8 bytes which represent the 8 horizontal pixel rows from top to bottom. Because the character width is 5 pixel, only the 5 lsb's are relevant. The Bit0 represents the right most pixel, Bit4 the left most.

Procedure *LCDcharset_M* (*LCD_num* : *TLCD_num*; *loc* : *char*; *c1*, *c2*, *c3*, *c4*, *c5*, *c6*, *c7*, *c8* : *byte*);

LCDcharset_M (*LCD_m5*, #10, \$1C, \$14, \$1C, \$00, \$00, \$00, \$00, \$00);

LCDout_M (*LCD_m5*, #10); (* display user defined degree char *)

Note:

Most LCD controllers support 16 private characters but can only display the first 8.

LCDcharset_MP

This procedure is a more general alternative to the version above. With *srcArea* the source type must be selected, 0=RAM, 1=EEPROM, 2=FLASH. The pointer must point to a structure (array[0..7] of byte) in the selected memory area.

Procedure *LCDcharset_MP* (*LCD_num* : *TLCD_num*; *loc* : *char*; *srcArea* : *byte*; *ptr* : *pointer*);

3.6.3 Unused Port Pins of the LCD Control Port

The upper (MSB) 5 bits of the LCD control port are not affected by the driver. These pins are always set as inputs and can be read by the application.

LCDportInp_M

This function reads the upper 5 input pins of the control port. The lower 3 bits (E, RW, RS) are always returned as 0. The parameter *LCD_num* is not stored and so doesn't influence the other LCD operations.

Function *LCDportInp_M* (**const** *LCD_num* : *TLCD_num*) : *byte*;



AVRco Standard Driver

3.6.4 Multi-Processing with the TWI Port

In an application with Processes and/or Tasks in many cases the TWI-Bus is not only used as a network but also for other purposes (LCD, Ports etc). If then different processes access the TWI heavy conflicts are implied because such sequential drivers (I2C, TWI, UART etc) are not re-entrant, i.e. they are not interruptible and cannot re-entered. Because of this the TWI port works with a semaphore of the type DeviceLock.

```
TWI_DevLock : DEVICELock;  
TWI_DevLockTN : DEVICELock; // XMega
```

The TWI driver observes and controls this semaphore. At the entry into the driver the semaphore is checked to see whether the driver is locked or not. If it is free the semaphore becomes active (driver locked) and the job will be executed. After finishing the job the semaphore is released (unlocked).

If the driver is locked (occupied) at entry time then a **Schedule** is executed and the calling Process is put into the Schedulers queue. In one of the next few scheduling slots this Process is restarted and again checks this semaphore. This is repeated until the semaphore becomes free (unlocked).

Important:

Because of the abortion of a TWI access through "Schedule" **Tasks should not** use TWI accesses. A schedule completely aborts and exits a Task and so a job will not be executed if a locked semaphore is found.

It is possible to set a flag if the Task solved the job successfully. If the flag is not set the Task must again be initiated to repeat this job. But this is somewhat complicated and should be avoided.

Furthermore it must be clear that this locking only works at byte level. For example if strings are sent to the LCD then the application is still responsible ensuring that the string transfer is not disturbed by another process which also wants to access the LCD.

Example program and schematic:

an example is in the directory ..\E-Lab\AVRco\Demos\LCDmulti

an XMega example is in the directory ..\E-Lab\AVRco\Demos\XMega_LCDmulti



3.7 LED 7seg Display

3.7.1 Disp7sPort, Mux|NonMux

Import and definition of a port for a 7segment LED display with up to 8 positions/Digits.

Multiplexed:

The display is prosecuted in the Common-Cathode Multiplex Modus. So 5 bits of an Output-Port are required as well as two serial driver-modules, one for the segments, for example UCN5895, and one for the digits, for example UCN5841.

The On-time of a digit is one SysTick. So the time for a complete Refresh id: Digits x SysTick (msec). With a SysTick of 10msec and 4 digits it results a refresh rate of $4 \times 10\text{msec} = 40\text{msec} = 25\text{Hz}$.

Non-Multiplexed:

This mode uses one shift register and 8 resistors for **each** digit. The advantage of this mode is ultra low radiation and therefore no EMR problems.

The Bit0 of the port is connected to the corresponding CLOCK-input and the bit1 to the corresponding data-input of both drivers. Bit2 is the Latch-Signal for the digit-driver and the bit3 is the Latch-Signal for the segment-driver. Bit4 is the output-enable-signal for the digit-driver (multiplexed).

Import *SysTick, Disp7sPort;*

Define *ProcClock = 4000000; {4Mhz clock }*
 SysTick = 10; {10msec Tick}
 Disp7sPort = PortA, Mux; {Port Adresse, multiplexed}
 //Disp7sPort = PortA, NonMux; {Port Adresse, not multiplexed}
 //Disp7sPort = PortA, Mux, startpin; {Port Adresse, multiplexed with Startpin}
 //Disp7sPort = UserPort; {User defined IO-driver, multiplexed}
 DispMode = ShiftLeft; {links durchschieben}
 DispDigits = 4, iData; {4-stelliges Display}

Disp7Port with startpin

The necessary 3..5 Bits used by the 7SegPort must not start with bit0. These bits can be placed anywhere into the used port on condition that all used bits fit into this port, starting with this "Startpin", in a consecutive order. Defining of this bit is optional. If omitted placement starts with bit0 of the port.

If **UserPort** is selected driver always works in multiplexed mode and the application must provide the UserDevice function **Disp7sIOS**.

DispMode

Defines the action of the displays on adding a new char with DispOut. **Wrap** places the internal insert cursor when reaching the end of the line to position 0 (left). With **ShiftLeft** the cursor stays always at right and the character is inserted here, the rest of the contents are shifted one position to the left. With **ShiftRight** this behaviour is exactly the opposite.

An additional option with the multiplex mode is that the display blank line can be disabled. This signal is not necessary if double buffered shift registers are used, because these types don't produce "ghost pictures". The blank-Pin is then unused and free for other purposes.

Define *DispMode = ShiftLeft, noBlank; {shift left}*

DispDigits

Number of digits of the display (1..8) and size of the symbol buffer. It is required for the initialisation.

3.7.2 Functions and Procedures

DispOut

Write into the display buffer. Passing parameter is of type byte or char. The symbol which is at the actual location of the display is overwritten with the new one. The interface can also be used with **Write**. See also description of LCD-display and "Write".

Because a 7seg-display is not able to represent all symbols of the alphabet, some symbols are represented as spaces. Control symbols are ignored, except 'CR' and 'LF'.

Linefeed (LF = \$0A) deletes the display and sets the writing cursor to the position '0'. Return (CR = \$0D) sets the writing cursor to the position '0', but does not change the display. With **wrap** mode if the writing cursor has reached the last position, it is automatically set to '0'. With the **shift** modes the content is shifted one digit to right or left and the new char is added.

```
DISPout ('A');                                {represent char A}  
Write (DispOut, #10 + 'Hello');
```

DispBlink

The whole display blinks or blinking is switched off. The passing parameter is of the type boolean.

```
DISPblink (true);                            {Blink on}
```

DispDigBlink

A single digit blinks or blinking is switched off. The passing parameter is of type byte and indicates the position or the digit (cell). If the digit is blinking, blinking is switched off and vice versa

```
DISPDigBlink (1);                            {Blinking on/off}
```

DispPos

The writing cursor is positioned at a certain location. The passing parameter is of the type byte and indicates the position or the digit (cell). The counting order of the digits is from '0' (left) up to 'Disp7sPort' -1 (right);

```
DISPPos (0);                                {Cursor Home}
```

DispClear

The whole display is deleted. The writing cursor is positioned to the location '0'. All actual blinking functions are reset.

```
DISPclear;                                {Delete + Cursor Home}
```

Disp7Test

Test function. It can be used to switch ON all segments of the display(s):

Procedure *Disp7Test;*

DispClEol

The display is deleted from the actual position of the cursor up to the end of line. The position of the writing cursor is preserved.

```
DISPclEol;                                {Delete up to end of line}
```



AVRco Standard Driver

Disp7Buff

Special user defined characters can be placed directly into the refresh buffer by addressing this buffer as an array[0..DispDigits-1] of byte

```
Disp7Buff[3]:= $63;           {display degree symbol}
```

DispRefresh

If a direct write into Disp7Buff is done, with NonMux mode the external shift registers are not updated. Because of this the user must force an update with this procedure in order to make the changes visible.

```
DISPrefresh;                  {physical update of the drivers}
```

_Disp7sTab

If special characters or a special character set is needed the internal character set can be overwritten. In this case an array[0..60] of byte must be defined. The first entry is the space (\$20). The system always subtracts \$20 from each character passed by DispOut. There are 2 possibilities for implementing this charset.

Define an array with constants by direct keyboard input:

```
_Disp7sTab : array[0..60] of byte = (0, 23, 53, 12, .., .., );
```

Generate an arrays of constants by reading a file:

```
_Disp7sTab : array[0..60] of byte = 'Disp7table.bin';
```

This file must contain all 60 possible characters in binary form.

UserPort

If the UserPort was selected by a define so the application must provide this device driver:

```
UserDevice Disp7sIOS(digit, segment : byte);  
begin  
...  
end;
```

The parameter *digit* passes the digit number (position) and the parameter *segment* passes the active segments of this digit. Each of the digit bits controls one out of eight possible digits. The Bit0 relates to Digit0 = left most digit etc. Active bits switch on the related digit or segment.

The segment definition follows the usual rules shown in the schematics below:

```
SegmentA    = $01  
SegmentB    = $02  
SegmentC    = $04  
SegmentD    = $08  
SegmentE    = $10  
SegmentF    = $20  
SegmentG    = $40  
DecimalPnt  = $80
```

The character "C" then results in the value \$39 in the parameter segment. For example:

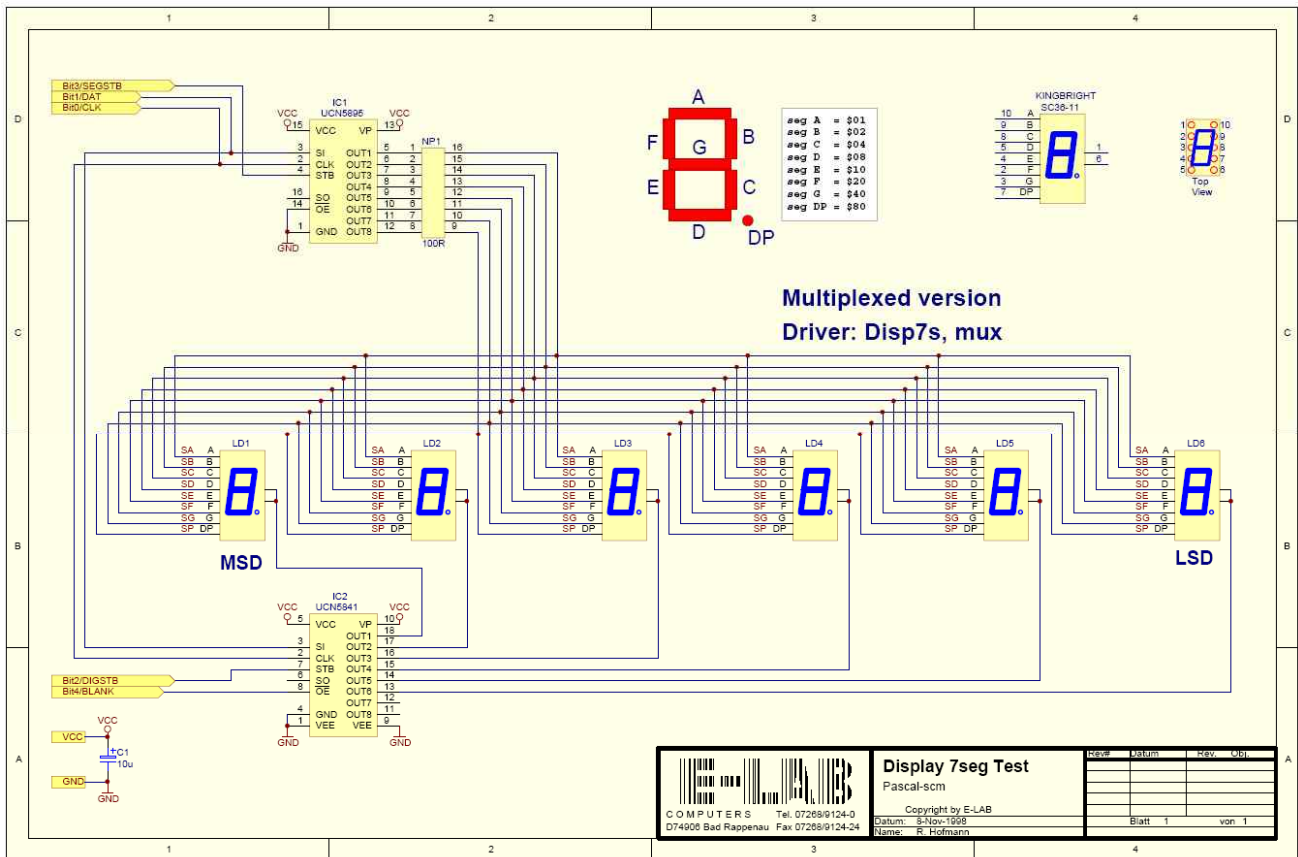
Digit = \$01

Segment = \$39

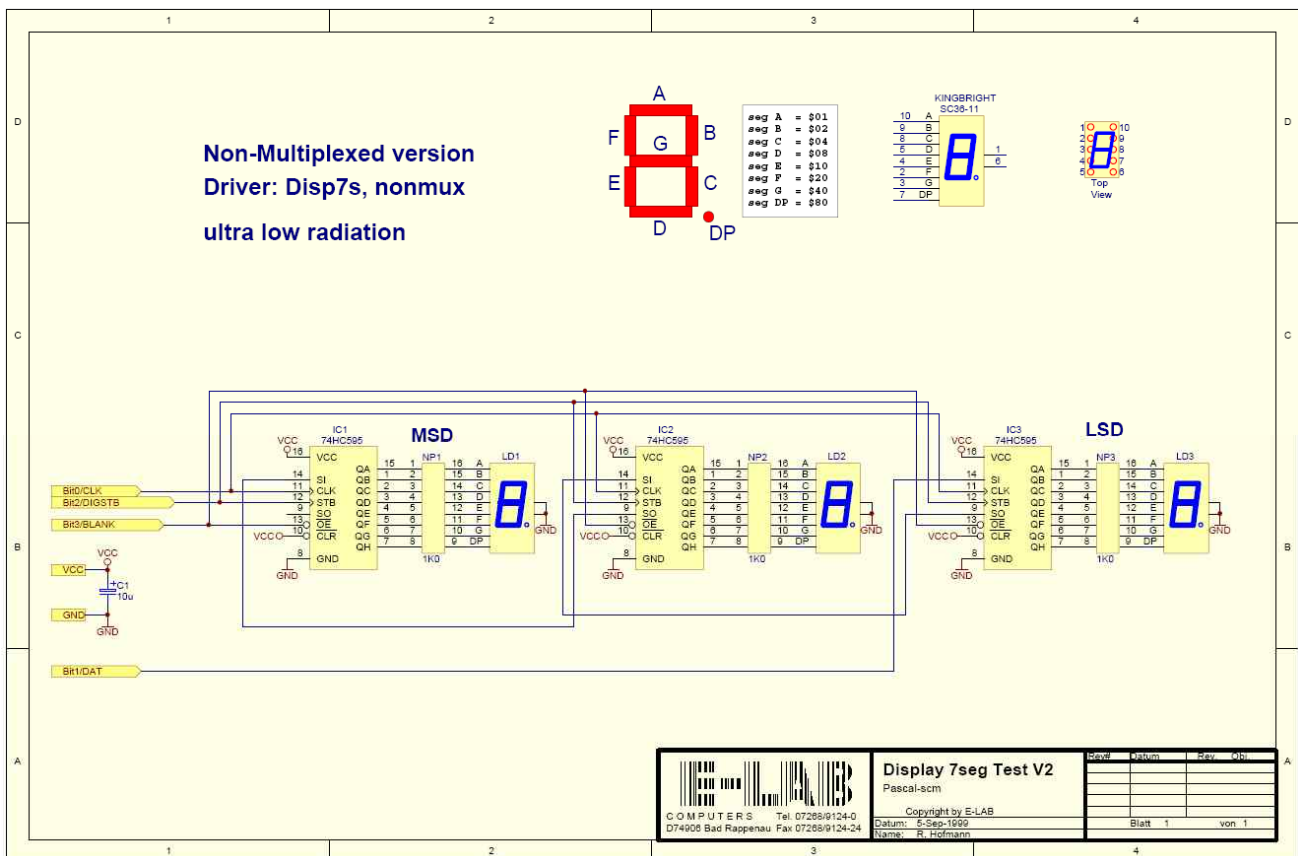
The left most digit then displays "C".

An example program **AVR Disp7sUser** is in the Demos directory in **Disp7sUser**.

AVRco Standard Driver



schematic **Disp7s** (multiplexed)



schematic **Disp7s2** (non multiplexed)



AVRco Standard Driver

3.8 LED 14seg Display Driver

Overview

Although LCD type of displays are standard because they are cheap, flexible and easy to use, 7 or 14segment LED displays are still used. In spite of their limited display possibilities they are used when character size (up to 20cm/4inch height) or high brightness or very good readability at long distances is needed. This is still the domain of LED displays and they have a "bright" future. The 14 segment types in particular offer a better flexibility and readability compared to their smaller brothers, the 7segment types.

14 segment LED displays are connected to a CPU either through shift registers or with special display driver chips. This driver implements the shift register type including the multiplexing operation.

By using ultra bright and low current types the resulting magnetic environment pollution (EMI) caused by the multiplexing are in an acceptable kevel, because the current peaks can be very low $\geq 15\text{mA}$

Introduction Disp14sPort

The driver controls one 14 segment LED display with up to 8 positions/Digits.

The display is controlled in the Common-Anode Multiplex mode. So 5 bits of an Output-Port are required as well as three serial driver-modules, two for the segments, for example UCN5841, and one for the digits, for example 74HC595. Because the HC595 cannot drive a digit it must be supported by small power drivers.

The On-time of a digit is one SysTick. So the time for a complete Refresh is: Digits x SysTick (msec). With a SysTick of 10msec and 4 digits it results a refresh rate of $4 \times 10\text{msec} = 40\text{msec} = 25\text{Hz}$.

Bit0 of the used CPU port is connected to the corresponding CLOCK-input and bit1 to the corresponding data-input of both drivers. Bit2 is the Latch-Signal for the digit-driver and the bit3 is the Latch-Signal for the segment-driver. Bit4 is the output-enable-signal for the digit-driver.

The wiring between the CPU, the driver chips and the 14 segment display is fixed by the driver and can be found in schematic below.

The driver functions are similar to the functions of the driver *Disp7sPort*.

Imports

As usual with the AVRco system the driver must be imported and defined. SysTick is also used.

Import *SysTick*, *Disp14sPort*;

Defines

As with the standard 7seg-Port, the used display size and count must be defined.

```
Define ProcClock    = 8000000;           {8Mhz clock }
          SysTick     = 5;                 {5msec}
          Disp14sPort = PortB, 0;          {Port, start Portbit}
          Disp14Mode  = ShiftRight, Blank;
          Disp14Digits = 6, iData;
```

Disp14sPort

Defines the Port to use. The parameter StartBit defines which bit in the port is the first to use, but the order of the bits is fixed.

Disp14sDigits

Defines the digit count of the driver 1..8 and the memory area of its buffer.

Disp14Mode

Defines the action of the displays with adding a new char with DispOut. **Wrap** places the internal insert cursor when reaching the end of the line to position 0 (left). With **ShiftLeft** the cursor stays at right and the character is inserted here. The rest of the contents are shifted one position to the left. With **ShiftRight** this behaviour is exactly the opposite.

With **Blank** a blanking signal is generated which suppresses ghosting while a digit is switched.

NoBlank disables this features and the port pin is not used.

3.8.1 Variables

All write operations are translated from ASCII into the segment presentation and then copied into the local buffer. The content of this buffer then is transferred to the display drivers through the external shift registers. The application can also directly write into the display buffer. In this way segments can be manipulated or special characters can be written and displayed.

The buffer is defined as **Array[0..Disp14Digits -1] of word**.

```
var Disp14Buff : array[0.. Disp14Digits-1] of word;
```

3.8.2 Functions

I2C_Disp7Pos

The writing cursor is positioned at a particular location. The passing parameter is of the type byte and indicates the position or the digit (cell). The counting order of the digits is from '0' (left) up to Disp14digits-1 (most right);

```
Procedure Disp14Pos (const digit : byte);  
Disp14Pos (2); (* position 2 *)
```

Disp14Out

Write into the display buffer. Passing parameter is of the type char. The symbol, which is at the actual location within the display is overwritten with the new one.

Because a 14seg-display is not able to represent all ASCII symbols, some symbols are represented as spaces. Control symbols are ignored, except 'CR' and 'LF' and #0..#9.

Linefeed (LF = \$0A) deletes the display and sets the writing cursor to the position '0'. Return (CR = \$0D) sets the writing cursor to the position '0', but does not change the display.

With **wrap** mode if the writing cursor has reached the last position, it is automatically set to '0'.

With **shift** mode if the writing cursor has reached the last position, the content is shifted one position left or right and the new character is appended.

```
Procedure Disp14Out (const ch : char);  
Disp14Out ('A'); {show char A}  
Write (Disp14Out, IntToStr (i)); {string output}
```



AVRco Standard Driver

Disp14Clear

The whole display is deleted. The writing cursor is positioned to the location '0'. All actual blinking functions are reset.

Procedure *Disp14Clear;*
Disp14Clear;

Disp14CLREOL

The display is deleted from the actual position of the cursor up to the end of line. The position of the writing cursor is preserved.

Procedure *Disp14CLREOL;*
Disp14CLREOL;

Disp14Blink

The whole display blinks or blinking is switched OFF. The passing parameter is of type boolean.

Procedure *Disp14Blink (const blink : boolean);*
Disp14Blink (true);

Disp14DigBlink

A single digit blinks or blinking is disabled. The passed parameter is of type byte and defines the position or the digit. With "blink" the blinking is switched ON or OFF.

Procedure *Disp14DigBlink (digit: byte; blink: boolean);*
Disp14DigBlink (1, true);

Disp14Test

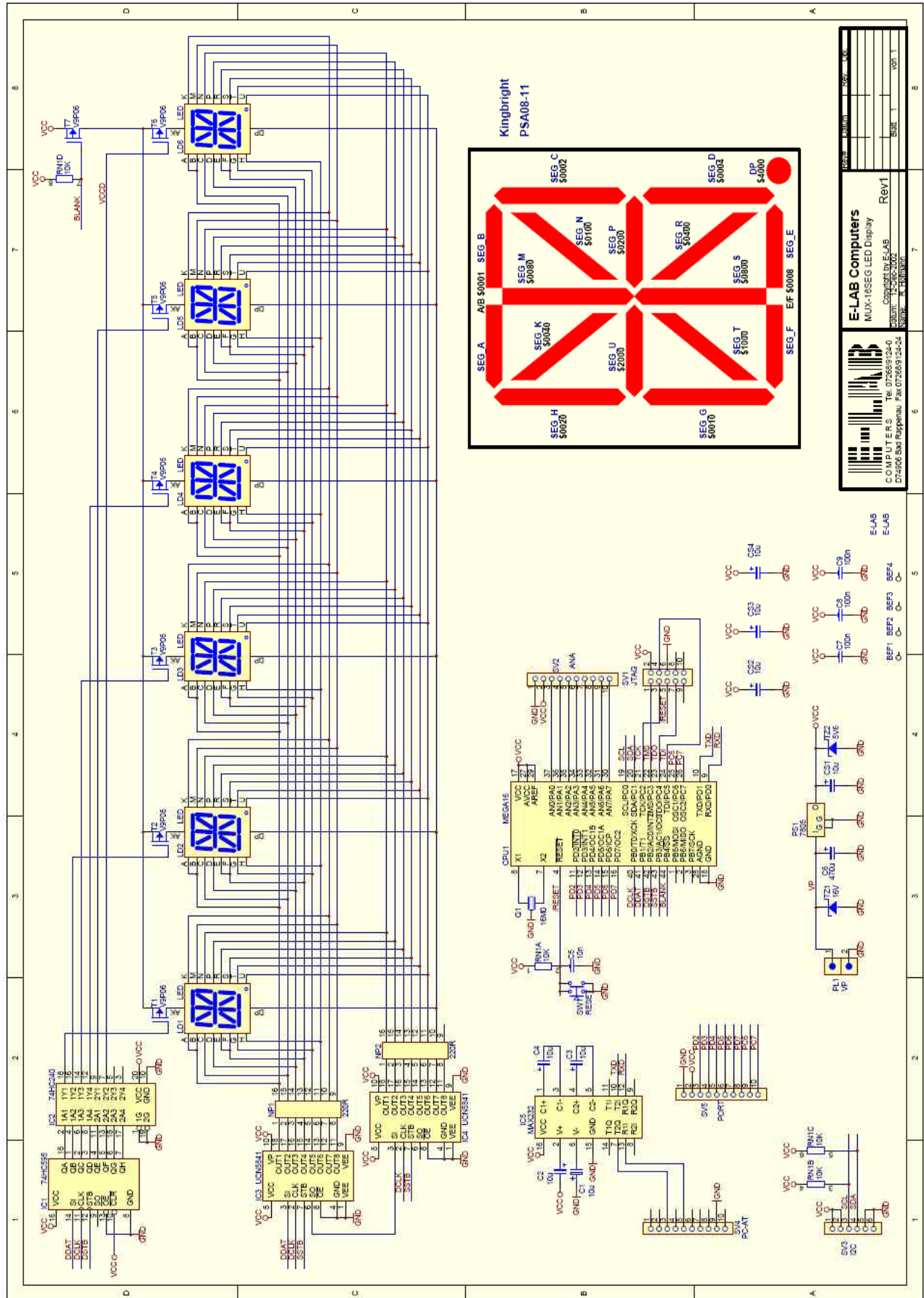
Test function. All segments of the display are switched ON. The writing cursor is positioned to location '0'. All actual blinking functions are reset.

Procedure *Disp14Test;*
Disp14Test;

Example program and schematic:

an example is in the directory `..\E-Lab\AVRco\Demos\Disp14s`

AVRco Standard Driver



schematic Disp14seg



AVRco Standard Driver

3.9 LED 7seg Display Driver for up to 4 displays

Overview

Although LCD type of displays are the standard because they are cheap, flexible and easy to use, 7 segment LED displays are still used. In spite of their limited information possibilities they are used when character size (up to 20cm/4inch height) or high brightness or very good readability at long distances is required. This is still the domain of LED displays and they have a "bright" future.

7 segment LED displays are connected to a CPU either through shift registers or with special display driver chips. The disadvantage of both versions on one hand is big code and CPU-time wasting. On the other hand the pin usage of the CPU increases with each additional display.

The solution is to implement the 7seg-displays on a bus system, usually the I2C. But ready to use 7seg-displays with a bus interface are not available.

So standard I2C LED driver chips must be used, which can directly supply 16 LEDs or 2 digits. Here only 2 port lines of the CPU are used, independent of the count of the connected displays or digits.

Current chips of this type contain a blinker and a dimmer so the CPU only needs to send commands. The blinking, dimming and multiplexing via continuous software loops, for example in the SysTick, is not necessary and the CPU is not burdened with this job.

Introduction I2C_Disp7

This implementation either uses the software I2C-driver (I2Cport) or the internal TWI (I2C) port of the AVR mega CPUs. To use the LCDmultiPort either the driver *I2Cport* or the driver *TWImaster* or the driver *TWInet* must be imported, the latter in Master mode. The standard 7SegPort driver can coexist with the I2C_Disp7 driver.

As the I2C port-expander one PCA9532 from Philips must be used for two digits. This chip can be present up to 8 times on the I2C bus. So the sum of all display digits cannot exceed 16 digits, which can be distributed to up to 4 displays where the digit count of a display cannot exceed 8.

The driver works in non-multiplex mode. Although this results in a larger part count, the EMI problems are minor. This is a big problem with multiplexed LED displays where the radiation can flow through the glass or plastic front without any attenuation.

The wiring between the PCA9532 and the 7seg displays is fixed by the driver and can be found in schematic below. The PCA9532 can run up to 400kBit/sec on the I2C Bus.

The driver functions are similar to the functions of the driver *Disp7sPort* but with the extension that each function has a display number parameter (exception: I2C_Disp7Out).

3.9.1 Technical Data

I2C Port or or	Software I2C imported through I2Cport CPU-TWI imported by the TWImaster CPU-TWI imported by the TWInet in Mastermode
Displays	up to 4, max. 8 digits/display, a total 16 digits max
Hardware	I2C LED-driver chip PCA9532 from Philips, 1 piece for 2 digits
I2C addresses	The PCA9532 are located on the bus-addresses \$60..\$67 MSD..LSD The PCA9532 have 3 address pins and bits which must be connected so that a continuous addressing is possible. Leftmost (MSD) digit has the lower address.

AVRco Standard Driver



Imports

As usual with the AVRco system the driver must be imported and defined. In addition the desired I2C/TWI driver must be imported and defined.

The SysTick is not used. Example:

```
Import I2Cport, I2C_Disp7;
or
Import TWImaster, I2C_Disp7;
or
Import TWInet, I2C_Disp7;           // use Master mode
```

Defines

Whether the I2C or TWIport is used must be defined. As with the standard 7seg-Port, the used display size and count must be defined.

Example for I2Cport:

```
Define ProcClock    = 8000000;           {8Mhz clock }
        I2Cport      = PortC;             {port used}
        I2Cdat       = 7;                 {bit7-PortC}
        I2Cclk       = 6, 4;              {bit6-PortC, optional delay 4}
        I2C_Disp7    = I2C_Soft, iData;    {use Software I2Cport, buffer loc}
        // use 2 Displays
        I2C_7sDig1   = 8;                 {first display 8digits}
        I2C_7sDig2   = 6;                 {second display 8digits}
        I2C_7MODE    = wrap;
```

Example for TWImaster:

```
Define ProcClock    = 8000000;           {8Mhz clock }
        TWIpresc     = TWI_BR100;         {100kBit/sec alt. TWI_BR400}
        I2C_Disp7    = I2C_TWI, xData;    {use TWIport, buffer location}
        // use 4 Displays
        I2C_7sDig1   = 4;                 {first display 4digits}
        I2C_7sDig2   = 4;                 {second display 4digits}
        I2C_7sDig3   = 4;                 {third display 4digits}
        I2C_7sDig4   = 4;                 {fourth display 4digits}
        I2C_7MODE    = shiftLeft;
```

Example for TWInetMaster:

```
Define ProcClock    = 8000000;           {8Mhz clock }
        TWInode      = 05;                {default address in slave mode}
        TWIpresc     = TWI_BR400;         {400kBit/sec alt. TWI_BR100}
        TWIframe     = 4, iData;          {buffer/packet size}
        TWIframeBC    = 6;                {option broadcast buffer/packet size}
        TWInetMode   = Master;
        I2C_Disp7    = I2C_TWI, iData;    {use TWIport, buffer location}
        // use 1 Display
        I2C_7sDig1   = 4;                 {4digits}
        I2C_7MODE    = wrap;
```

I2C_Disp7

Requests the I2C-port to use, either software-I2C with *I2C_Soft* or on-chip TWIport with *I2C_TWI*. The required basic driver must also be imported and defined.



AVRco Standard Driver

I2C_7sDIGn

Defines the import of a displays and its digit (cell) count.

I2C_7MODE

Defines the reaction of the displays with adding a new char with DispOut. **Wrap** places the internal insert cursor when reaching the end of the line to position 0 (left). With **ShiftLeft** the cursor stays always at right and the character is inserted here, the rest of the contents is shifted one position to the left.

3.9.2 Types

The import of the I2C_Disp7 publishes an enumeration which must be used for the function calls of the driver:

Type *TI2C_DISP7 = (Disp7_1, Disp7_2, Disp7_3, Disp7_4);*

For testing, on and off switching of the display this enumeration must be used:

Type *TI2C_Ctrl7 = (Disp7_On, Disp7_Off, Disp7_Test);*

3.9.3 Variables

All write operations are translated from ASCII into the segment presentation and then copied into local buffer. The content of this buffer then is transferred to the display drivers through the I2C bus. The application can also write directly into the display buffer. In this way segments can be manipulated or special characters can be written and displayed.

Each display imported with Define has its own buffer which is defined as an Array[0..I2C_7sDIGn -1].

Var *I2C_7Buff1, I2C_7Buff2, I2C_7Buff3, I2C_7Buff4;*

3.9.4 Functions

All functions with the exception of *I2C_Disp7Out* need as the first parameter the number of the desired display. This parameter is then stored internally and will be used by *I2C_Disp7Out*. Each function starts an internal refresh if necessary.

I2C_Disp7Init

Each connected display (driver chips) must be initialised at least once. The buffer content stays unchanged. The BlinkRate defines the flash rhythms. The accompanying on/off must be defined with the parameter DutyCycle.

Function *I2C_Disp7Init (const Disp : TI2C_Disp7; BlinkRate, DutyCycle : byte);*
I2C_Disp7Init (Disp7_1, 50, 127); // 3Hz rate, 50/50 onoff

I2C_Disp7Ctrl

Each connected display can be switched on, switched off and tested without changing the buffer content. The test function switches on all segments including the decimal point. With the ON-function exits the test mode.

Function *I2C_Disp7Ctrl (const Disp : TI2C_Disp7; const ctrl : TI2C_Ctrl7);*
I2C_Disp7Ctrl (Disp7_1, Disp7_Test);
I2C_Disp7Ctrl (Disp7_2, Disp7_Off);
I2C_Disp7Ctrl (Disp7_3, Disp7_On);

I2C_Disp7Test

Test function. All segments of the display are switched ON. The writing cursor is positioned to location '0'. All actual blinking functions are reset.

Procedure *I2C_Disp7Test (const Disp: TI2C_Disp7);*

I2C_Disp7Pos

The writing cursor is positioned to a specific location. The passing parameter is of the type byte and indicates the position, or the digit (cell). The counting order of the digits is from '0' (left) up to I2C_7sDIGn -1 (most right);

Procedure *I2C_Disp7Pos (const Disp : TI2C_Disp7; const digit : byte);
I2C_Disp7Pos (Disp7_1, 2); (* position 2 *)*

I2C_Disp7Set

Optional, switches the driver to a specific display. Only makes sense with **I2C_Disp7Out**.

Procedure *I2C_Disp7Set (const disp : TI2C_Disp7);
I2C_Disp7Set (Disp7_1);*

I2C_Disp7Get

Optional. The function returns the current selected display.

Function *I2C_Disp7Get : TI2C_Disp7;*

I2C_Disp7Out

Because this procedure also must interact with **Write**, there is no display select parameter Disp. The writing to the display is always directed to the currently active display. The desired display can also be explicitly set with *I2C_Disp7Set*.

Writes into the display buffer. Passing parameter is of the type char. The symbol, which is at the current location in the display is overwritten with the new one. Because a 7seg-display is not able to represent all symbols of the alphabet, some symbols are represented as spaces.

Control symbols are ignored, except 'CR' and 'LF'. Linefeed (LF = \$0A) deletes the display and sets the writing cursor to the position '0'. Return (CR = \$0D) sets the writing cursor to the position '0', but does not change the display. With wrap mode if the writing cursor has reached the last position, it is automatically set to '0'. With a **Shift** mode the display content is shifted one position to the left and the new character is appended.

Procedure *I2C_Disp7Out (const ch : char);
I2C_Disp7Out ('A'); {show char A}
Write (I2C_Disp7Out, IntToStr (i)); {string output}*

I2C_Disp7Clear

The whole display is deleted. The writing cursor is positioned to location '0'. All actual blinking functions are reset.

Procedure *I2C_Disp7Clear (const Disp : TI2C_Disp7);
I2C_Disp7Clear (Disp7_1);*



AVRco Standard Driver

I2C_Dis7CLEOL

The display is blanked from the actual position of the cursor up to the end of line. The position of the writing cursor is preserved.

Procedure *I2C_Dis7CLEOL (const Disp : TI2C_Dis7);*
I2C_Dis7CLEOL (Disp7_1);

I2C_Dis7DigitBlink

A single digit blinks or blinking is disabled. The passed parameter is of type byte and defines the position or the digit (cell). With "blink" the blinking is switched ON or OFF.

Procedure *I2C_Dis7DigitBlink (Disp : TI2C_Dis7; digit: byte; blink: boolean);*
I2C_Dis7DigitBlink (Disp7_1, 1, true);

I2C_Dis7Dim

The displays can be dimmed. The brightness can be set in the range of 0..255. Consider two important points with dimming

1. A blinking display always switches between 0% and 100% brightness. Dim values are ignored with blinking.
2. The dimming is done with pulse width modulation (PWM). Because of this a higher EMI radiation must be expected. But this will be much less than with multiplexed displays.

Procedure *I2C_Dis7Dim (const Disp : TI2C_Dis7; const dim : byte);*
I2C_Dis7Dim (Disp7_1, 127); (* brightness to 50% *)

I2C_Dis7Refresh

All functions and procedures start an internal refresh/update from the display buffer to the digit drivers if necessary.

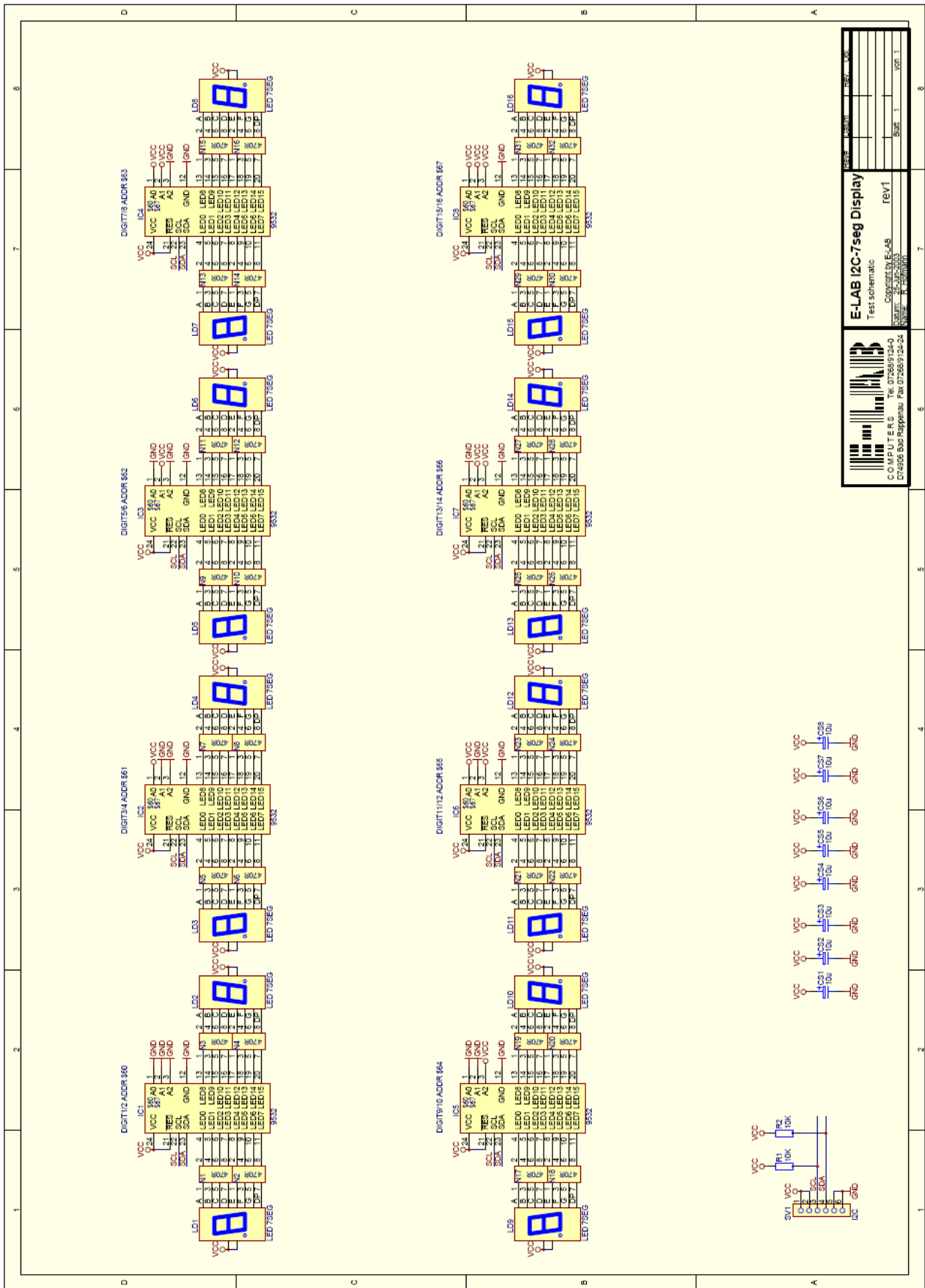
Direct writing to the **I2C_7Bufx** has no direct influence to the display, because the external driver chips are not updated by a direct buffer manipulation. Hence if a direct writing to a buffer was made this procedure must be called in order to make the changes visible.

Procedure *I2C_Dis7Refresh (const Disp : TI2C_Dis7);*
I2C_Dis7Refresh (Disp7_1);

Example program and schematic:

an example is in the directory **..\E-Lab\AVRco\Demos\I2C_Dis7s**

AVRco Standard Driver



schematic I2C_7seg



AVRco Standard Driver

3.10 IOexpand Driver for up to 128 digital IOs

Basics

With some control applications and with a specific CPU used the useable IO-pins are not sufficient especially if two ports must be used for external RAM.

So a larger CPU (pin count) must be used or the current port pins must be expanded with additional hardware. There are several ways to implement such an expansion. Real IO-chips like the 8255 often can't be used because they need a parallel connection (at least 10bits) to the CPU and there are only 20 additional pins. Standard latches must also be connected in parallel and therefore also need many additional Pins of the CPU so the pin saving is less than expected.

If the IO-speed doesn't matter too much but the count of additional pins is high, the best way is to implement external shift registers which solve the problem very well.

Each shift register serves either for 8 input or 8 output bits. The connection of them uses typically 3 control pins. Because the shift registers are cascadeable the control pin count is constant regardless of how many registers are serialized. The drawback of this method is that each in/out bit needs one shift register cycle (clock, data).

But with smart programming of the driver it's possible to save machine cycles and also port pins if the combination of input and output types is used.

Overview IOexpander

This implementation uses 3 bi-directional port bits of the CPU if only one mode is implemented, either input or output. If input and output are combined, 5 bi-directional port pins must be used. All port pins must be in a consecutive order in the same port.

The driver is completely passive, this means that the application must use it by demand (cyclically by a Task, Process, interrupt or simply when necessary). The driver places its results into variables which can be read at any time. Output values which should changed or become updated must be written into the related variables so the driver reads them and stores them into the shift registers.

Each driver call reads the output variables and shifts their content into the output shift registers. At the same time with each single bit shift operation it reads the content of the input shift registers and places the result into the input variables in the memory.

The driver works byte-oriented, this means there must be always 8bit shift registers connected. There are maximal 8 registers possible which results in max. 64 bits. This is true for both, the input and also for the output. The registers can be in any configuration. For example there can be 8 input registers and 1 output register.

3.10.1 Technical Data

Input Register	0..8	Bits	0, 8, 16, 24, 32, 40, 48, 56, 64
Output Register	0..8	Bits	0, 8, 16, 24, 32, 40, 48, 56, 64
Input Databytes	0..8	in iData or xData area	
Output Databytes	0..8	in iData or xData area	
Used Port Bits	Input only	3	
	Output only	3	
Combined	5		

AVRco Standard Driver



Imports

As usual with the AVRco system the driver must be imported and defined.
SysTick is not necessary.

Import *IOexpand;*

Defines

It must be specified which port of the CPU and which bits of this port must be used, where the variables reside and also how many input and output bytes (shift registers) must be used.

Define *ProcClock* = 8000000; {8Mhz clock }
 IOexpand = PortD, 2, iData; {Port, Port-startbit, memory loc}
 IOexplnp = 4; (4x8 = 32 bit input}
 IOexpOutp = 4; (4x8 = 32 bit output}

IOexpand

Defines the port to use, the first useable bit of this port and the data area of the driver. The port must be bi-directional. The startbit in the port must be chosen so that all required control bits for the shift registers (3 or 5) fit into the port. The data area defines where the working variables of the driver reside.

IOexplnp

Defines the count of input bytes and also memory bytes used.

IOexpOutp

Defines the count of output bytes and also memory bytes used.

3.10.2 Functions and Variables

IOexplnpArr

The import of the driver exports this array if input pins/bytes were defined. The size of this array depends of the count of the input bytes. The driver stores its read results in this array in the following order: first byte into IOexplnpArr[0] the seconds byte into IOexplnpArr[1] etc.

The definition of the array is this:

Var *IOexplnpArr* : **array**[0.. *IOexplnp-1*] **of** byte;

IOexplnpX

The bytes of the array also have synonyms so its also possible to read the bytes without array indexing. The bytes, where only [IOexplnp] count exist, are defined:

Var *IOexplnp0* : byte;
 IOexplnp1 : byte;
 IOexplnp2 : byte;
 ...



AVRco Standard Driver

IOexpOutpArr

The import of the driver exports this array if output pins/bytes were defined. The size of this array depends of the count of the output bytes. The driver reads the output bytes from this array in the following order: first byte from IOexpOutpArr[0] the seconds byte from IOexpOutpArr[1] etc. and shifts them out into the output shift registers.

The definition of the array is this:

```
Var IOexpOutpArr : array[0.. IOexpOutp-1] of byte;
```

IOexpOutputx

The bytes of the array also have synonyms so it's also possible to write the bytes without array indexing. The bytes, where only [IOexpOutp] count exist, are defined:

```
Var   IOexpOutp0 : byte;  
      IOexpOutp1 : byte;  
      IOexpOutp2 : byte;  
      ...  
      ...
```

IOexpUpdate

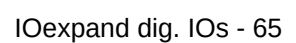
The application calls this procedure which executes the driver. The driver reads all input shift registers and stores them into the input array. In the same process the output array is read out and shifted into the output shift registers.

The definition of IOexpUpdate is:

```
Procedure IOexpUpdate;
```

Example program and schematic:

an example is in the directory ..\E-Lab\AVRco\Demos\IOexpand





AVRco Standard Driver

3.11 RS232/V24 Driver SerPort, SerPort2 and SerPort3, -4

The bigger AVR types have one or more serial interfaces. In the older types (AT90Sxxxx) is a UART implemented; the newer contain so-called USARTs.

The USARTs are much more flexible than the former UART.

The AVRco system supports, dependent of the CPU type up to 4 SerPorts:

SerPort (SerPort1), SerPort2, SerPort3, SerPort4.

The **XMegas** provide upto 8 interfaces: **SerPortC0, SerPortC1, SerPortD0, SerPortD1** etc.

If an AVR CPU has USARTs instead of UARTs the AVRco system offers some additional system functions that support the enhanced functionality of these USARTs. These are changing of the data byte length, parity mode and stopbit count. These functions are available at runtime.

Furthermore the internal **default values** (2 Stopbits, noParity, 8 Databits) can be overwritten at design time with defines. So the system starts up with the desired setup and a runtime setup is not necessary in most cases.

3.11.1 Basic Functions (UART and USART)

Imports

As usual with the AVRco system, the driver must be imported.

Import SysTick, SerPort, SerPort2, ...; // SerPort = SerPort1

Defines

The baud rate, optional Rx and/or TxBuffer, optional Databits, Stopbits or Parity

```
Define  ProcClock  = 4000000;           {4Mhz clock }
          SysTick    = 10;                {10msec Tick}
          SerPort    = 9600, Stop1;        {9600 Baud, 1Stopbit}
          // SerPort = 9600, parEven;      {9600 Baud, even parity}
          // SerPort = 9600, parOdd;       {9600 Baud, odd parity}
          // SerPort1 = 9600, parOdd;      {9600 Baud, odd parity}
          //SerPorC0 = 9600, SyncMaster;   {9600 Baud, synchron mode, XCK output}
          //SerPorC0 = 9600, SyncSlave;    {9600 Baud, synchron mode, XCK input}
          TxBuffer   = 8;                  {8 Byte Buffer and Int}
          // TxBuffer1 = 8;                {8 Byte Buffer and Int}
          RxBuffer   = 8, iData;           {8 Byte Buffer and Int}
          // RxBuffer1 = 8, iData;         {8 Byte Buffer and Int}
```

There are analogous definitions for the other serial interfaces:

```
Define  SerPort2   = 9600, Stop1;        {9600 Baud, 1Stopbit}
          // SerPort2 = 9600, parEven;     {9600 Baud, even parity}
          // SerPort2 = 9600, parOdd;      {9600 Baud, odd parity}
          TxBuffer2  = 8;                  {8 Byte Buffer and Int}
          RxBuffer2  = 8, iData;           {8 Byte Buffer and Int}

Define  SerPort3   = 9600, Stop1;        {9600 Baud, 1Stopbit}
          // SerPort3 = 9600, DataBit5    { DataBit5, DataBit6, DataBit7, DataBit9}
          TxBuffer3 ...
          etc.
```

AVRco Standard Driver



XMega

```
Define ProcClock = 4000000;      {4Mhz clock }
        SysTick   = 10;          {10msec Tick}
        SerPortC0 = 9600, Stop1;  {9600 Baud, 1Stopbit}
        //SerPortC0 = 9600, parEven; {9600 Baud, gerade Parität}
        //SerPortC0 = 9600, parOdd; {9600 Baud, ungerade Parität}
        TxBufferC0 = 8;           {8 Byte Buffer und Int}
        RxBufferC0 = 8, iData;    {8 Byte Buffer und Int}
```

There are analogous definitions for the other serial interfaces:

```
Define SerPortC1 = 9600, Stop1;    {9600 Baud, 1Stopbit}
        //SerPortC1 = 9600, parEven; {9600 Baud, gerade Parität}
        //SerPortC1 = 9600, parOdd;  {9600 Baud, ungerade Parität}
        TxBufferC1 = 8;             {8 Byte Buffer und Int}
        RxBufferC1 = 8, iData;      {8 Byte Buffer und Int}

Define SerPortD0 = 9600, Stop1;    {9600 Baud, 1Stopbit}
        //SerPortD0 = 9600, DataBit5 { DataBit5, DataBit6, DataBit7, DataBit9}
        TxBufferD0 ...
        etc.
```

By the import of the XMega SerPorts this type (Enumeration) is defined by the system:

Type

```
tUSARTenum = (UsartC0, UsartC1, UsartD0, UsartD1, UsartE0, UsartE1, UsartF0, UsartF1, UsartCDC);
```

Notes:

Because with standard UARTs the system uses the 9th bit either as the 2nd Stopbit or as the Paritybit it is only possible to use 2 Stopbits or Parity. Both together is not possible! But there can be exceptions by newer AVR's which support all by hardware.

Please note that if any parity function is enabled the parity must be calculated for each byte by the system if the UART doesn't support it.

Parameter changes at runtime are possible.

RXBUFFER (RXBUFFER1), RAMpage

Definition of the RX-Buffer length (Ringbuffer).

RXBUFFER is used by functions: If defined the required buffer memory locations are allocated (4..254), and the receiver works with buffers. Otherwise it works with polling. Take care with definition of the buffer length by RxBuffer. The most processors only have a small Ram memory!

The buffer (if defined) works as a ringbuffer without watching. That means that symbols may get lost, if they are not fetched by *SerInp* in time. That might lead to a program crash. So programming carefully with buffer execution, too!

TXBUFFER (TXBUFFER1), RAMpage

Definition of the TX-Buffer length. See also above (RxBuffer). It is also valid for TxBuffer, but the exception is that the Txbuffer cannot overflow.

3.11.1.1 Functions and Procedures

SerStat (SerStat1, SerStat 2, SerStat 3, SerStat 4)

XMega

SerStatC0, SerStatC1, SerStatD0, SerStatD1, SerStatE0, SerStatE1, SerStatF0, SerStatF1

Gives the state of the serial interface. It should be called before *SerInp* is called, to avoid unnecessary waiting periods within the procedure.

SerStat returns TRUE, if a character exists. With processes and tasks you should avoid SerStat, but use WaitPipe(RxBuffer) instead.



AVRco Standard Driver

Furthermore the function "PipeFull" can be used with the serial interfaces:

Function *PipeFull (RxBuffer): boolean;*

SerStatP (SerStatP1, SerStatP2, SerStatP3, SerStatP4)

XMega

SerStatPC0, SerStatPC1, SerStatPD0, SerStatPD1, SerStatPE0, SerStatPE1, SerStatPF0, SerStatPF1

With

Define *SerPort = 9600, parEven;*

XMega

Define *SerPortC0 = 9600, parEven;*

an even parity is selected.

With

Define *SerPort = 9600, parOdd;*

XMega

Define *SerPortC1 = 9600, parOdd;*

an odd parity is selected.

If a parity error is detected while receiving, the received byte stays valid and can be read. After reading with the function

SerStatP : boolean;

XMega

SerStatPC0 : boolean;

the result of the parity check can be found. If there was a parity error, the result is true (= Error) otherwise it is false. If the receiver works with RxBuffer it's only possible to get the info that there was an error or not since the last call of "SerStatP". Which byte in the RxBuffer this relates to cannot be found. The call of "SerStatP" always resets the systems internal parity error flag to false (no error).

SerBaud (SerBaud1, SerBaud2, SerBaud3, SerBaud4)

XMega

SetSerBaud

Adjusts the BaudRate of the serial interface at runtime.

SerBaud(19200);

XMega

SetSerBaud(UsartC0, 19200);

SerInp (SerInp1, SerInp2, SerInp3, SerInp4)

XMega

SerInpC0, SerInpC1, SerInpD0, SerInpD1, SerInpE0, SerInpE1, SerInpF0, SerInpF1

Reads the serial interface or buffer.

With polling (RxBuffer = 0) SerInp does not read the buffer but the input register of the UART's. In both cases the procedure waits until a character to fetch is ready. To avoid needless waiting periods or a blockading of the program it normally makes sense to fetch the state of the interface by SerStat first and then the received character.

repeat until *SerStat;* *{wait for RxReady}*

c:= SerInp;

XMega

repeat until *SerStatC0;* *{wait for RxReady}*

c:= SerInpC0;

If processes are imported it is better to do the interface within a process, so the process waits for a character with **WaitPipe** (RxBuffer) and does not waste any run-time. The Receiver buffer can be cleared by **FlushBuffer**(RxBuffer).

XMega

WaitPipe (RxBufferC0) **FlushBuffer**(RxBufferC0)

SerOut (SerOut1, SerOut2, SerOut3, SerOut4)

XMega

SerOutC0, SerOutC1, SerOutD0, SerOutD1, SerOutE0, SerOutE1, SerOutF0, SerOutF1

With polling (TxBuffer = 0) SerOut does not write into the buffer, but directly into the sending-register of the UART. In both cases the procedure waits until the previous character was sent and the sending-register is free, or until the sending-buffer is able to accept a new character. Example:

```
SerOut ('x');
Write (SerOut, ByteToStr (100:6));    {-> ' 100'}
```

XMega

```
SerOutC0 ('x');
Write (SerOutC0, ByteToStr (100:6)); {-> ' 100'}
```

SerInpBlock (SerInpBlock1, SerInpBlock2, SerInpBlock3, SerInpBlock4)

SerInpBlock_P (SerInpBlock1_P, SerInpBlock2_P, SerInpBlock3_P, SerInpBlock4_P)

Procedure SerInpBlock (*var* location: type);

Procedure SerInpBlock_P(ptr : pointer; len : word);

XMega

SerInpBlock

SerInpBlock_P

Procedure SerInpBlock (Usart : tUSARTenum; *var* location: type);

Procedure SerInpBlock_P(Usart : tUSARTenum; ptr : pointer; len : word);

With these functions it is possible to receive any data structure. Possible memory areas are: Procedure- or function-local vars, RAM, EEPROM and Flash. With SerInpBlock_P, SerInpBlock2_P etc. the destination must be a RAM location. The type of the target (location) in RAM is only used for calculation of the block size. The underlying structure is ignored.

SerOutBlock (SerOutBlock1, SerOutBlock2, SerOutBlock3, SerOutBlock4)

SerOutBlock_P (SerOutBlock1_P, SerOutBlock2_P, SerOutBlock3_P, SerOutBlock4_P)

Procedure SerOutBlock(*const* location: type);

Procedure SerOutBlock_P(p : pointer; count : word);

XMega

Procedure SerOutBlock(Usart : tUSARTenum; *const* location: type);

Procedure SerOutBlock_P(Usart : tUSARTenum; p : pointer; count : word);

With these procedures it's possible to send any data structure. Possible memory areas are: Procedure- or function-local vars, RAM, EEPROM and Flash

3.11.1.2 Interrupt Operation

With most applications serial interfaces are interrupt driven. This has the advantage that sending and receiving of characters are done by the system in the background. SerPort interrupts always need a circular buffer where the characters are written to and read from. By the Define of an Rx- or TxBuffer the necessary interrupt system is automatically imported.

```
Define RxBuffer  = 24, iData;
        TxBuffer  = 16, iData;
        RxBuffer2 = 40, xData;
        TxBuffer2 = 8,  xData1;
        RxBuffer3 = 40, xData;
        TxBuffer3 = 8,  xData1;
        RxBuffer4 = 40, xData;
        TxBuffer4 = 8,  xData1;
```

XMega

```
Define RxBufferC0 = 24, iData;
        TxBufferC0 = 16, iData;
        RxBufferF0 = 40, xData;
        TxBufferF1 = 8,  xData1;
```



AVRco Standard Driver

SerInp Timeout

An alternative for receiving of serial data with "if SerStat then SerInp" is using SerInp with an additional timeout parameter. Some jobs can be solved in a better way with timeouts. The TimeOut mode for data receive must be declared:

```
Define SysTick    = 10;           // necessary for timeout handling
        SerPort    = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPort2   = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPort3   = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPort4   = 9600, Stop2, timeout; // Stop2 and timeout are optional
```

XMega

```
Define SysTick    = 10;           // necessary for timeout handling
        SerPortC0  = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPortD1  = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPortE1  = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPortF0  = 9600, Stop2, timeout; // Stop2 and timeout are optional
```

Then these system functions are imported:

```
Function SerInp_TO (var rx : char|byte; const timeout : byte) : boolean;
Function SerInp_TO1 (var rx : char|byte; const timeout : byte) : boolean;
Function SerInp_TO2 (var rx : char|byte; const timeout : byte) : boolean; // 2.chan
Function SerInp_TO3 (var rx : char|byte; const timeout : byte) : boolean; // 3.chan
Function SerInp_TO4 (var rx : char|byte; const timeout : byte) : boolean; // 4.chan
```

XMega

```
Function SerInp_TOC0 (var rx : char|byte; const timeout : byte) : boolean; // SerPortC0
Function SerInp_TOD1 (var rx : char|byte; const timeout : byte) : boolean; // SerPortD1
Function SerInp_TOE1 (var rx : char|byte; const timeout : byte) : boolean; // SerPortE1
Function SerInp_TOF0 (var rx : char|byte; const timeout : byte) : boolean; // SerPortF0
```

The parameter "rx" must be a variable of type byte or char, but EEPROM is not possible. TimeOut is a byte. After the function returns the variable "rx" contains the character/byte received if the result was true. If the result was false the variable "rx" is unchanged because no data has been received.

The parameter "TimeOut" counts in SysTicks. If a zero value is passed to the function the function returns immediately with or without data (true/false). If parameter > 0 then the function waits until the timeout has elapsed or a character has been received.

SerInp_TO is not useable with the Read and ReadLn functions!

SerInpBlock Timeout

The procedures "SerInpBlock" have timeout controlled variants. Here the same is true as in the "SerInp TimeOut" paragraph above. The function returns with a true if "location" is completely filled with received data. If a TimeOut occurs the function result becomes false and the passed variable is not filled or incompletely filled with data. The value of TimeOut is used for each received byte. TimeOut must be imported:

```
Define SysTick    = 10;           // necessary for timeout handling
        SerPort    = 9600, Stop2, timeout; // Stop2 and timeout are optional
        // SerPort1 = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPort2   = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPort3   = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPort4   = 9600, Stop2, timeout; // Stop2 and timeout are optional
```

XMega

```
Define SysTick    = 10;           // necessary for timeout handling
        SerPortC0  = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPortD1  = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPortE1  = 9600, Stop2, timeout; // Stop2 and timeout are optional
        SerPortF0  = 9600, Stop2, timeout; // Stop2 and timeout are optional
```

Then these system functions are imported:

```
Function SerInpBlock_TO (var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO1 (var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO2 (var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO3 (var location : type; const TimeOut : byte) : boolean;
Function SerInpBlock_TO4 (var location : type; const TimeOut : byte) : boolean;
XMega
Function SerInpBlock_TO(Usart : tUSARTenum; var location : type; const TimeOut : byte) : boolean;
```

```
Function SerInpBlockP_TO (ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO1 (ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO2 (ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO3 (ptr : pointer; len : word; const TimeOut : byte) : boolean;
Function SerInpBlockP_TO4 (ptr : pointer; len : word; const TimeOut : byte) : boolean;
```

XMega

```
Function SerInpBlockP_TO(Usart : tUSARTenum; ptr : pointer; len : word; const TimeOut : byte) : boolean;
```

The target of this operation must be RAM. Writing to EEprom is not possible. The type of the target (location) in RAM is only used for calculation of the block size. The underlying structure is ignored.

3.11.1.3 Handshake Operation

Handshake Tx (DTR)

Sometimes an external unit controls the data flow of "SerOut" of the serial interface of the AVR. This is no problem, as long as the SerPort runs with no TxBuffer and so the SerOut is always done in polling mode. But with a TxBuffer it becomes a problem because the application can recognize DTR-line and now can disable the CPU's Tx-interrupt of the UART, but this can be too late and the external unit is overrun. In addition the application must always poll the DTR line which practically impossible.

To support the DTR handshake with Tx-buffer operation the SerPort driver is extended. If the external device deactivates its DTR line the Tx-Buffer driver immediately stops the transmit operation.

This line is watched in the SysTick and if the stop condition is removed the transmission continues automatically. If no SysTick is implemented then it's the job of the application to re-enable the transmission when the DTR line is activated again. This can be done automatically if the DTR pin of the CPU is an edge triggered interrupt pin. Then a DTR enable generates an interrupt which then re-enables the transmit operation with "SerPort_Send". With most cases it is sufficient when the main program loop recognizes the change from disable to enable of the DTR line and then restarts the transmission. The DTR-option of the serial interface must be defined by:

```
Define SerPortDTR    = PinC, 2, Positive;    // first serport
        SerPortDTR1   = PinC, 2, Positive;    // first serport
        SerPortDTR2   = PinC, 3, Negative;    // second serport if available
        SerPortDTR3   = PinC, 4, Positive;    // third serport if available
        SerPortDTR4   = PinC, 5, Negative;    // fourth serport if available
```

XMega

```
Define SerPortDTRC0 = PinC, 2, Positive;    // first serport
        SerPortDTRC1 = PinC, 3, Negative;    // second serport if available
        SerPortDTRD0 = PinC, 4, Positive;    // third serport if available
        SerPortDTRD1 = PinC, 5, Negative;    // fourth serport if available
```

This define also imports the DTR handshake and defines the port pin used. The level statement means "enable at level".

To handle the restart (tx-enable) by the application these procedures are implemented:

```
Procedure SerPort_Send;    or    Procedure SerPort_Send1;
Procedure SerPort_Send2;
Procedure SerPort_Send3;
Procedure SerPort_Send4;
```

XMega

```
Procedure SerPort_Send (Usart : tUSARTenum);
```



AVRco Standard Driver

Handshake Rx (DSR)

In the other direction, on receiving data, it is also possible to overrun the RxBuffer. With a pure text transfer the XON/XOFF protocol can be used. But this is not possible with binary data. So also here a hardware handshake must be used. The receiver signals the transmitter whether its buffer is full or that it can receive more data. The stop occurs when the buffer is $\frac{3}{4}$ full, the ready occurs when the receiver buffer is nearly empty, below $\frac{1}{4}$ buffer size.

To support the DSR handshake with Rx-buffer operation the SerPort driver is extended. If the driver deactivates its DSR line the external transmitter immediately stops its transmit operation until the driver's buffer drops below the lower threshold and it re-enables transmission by setting the DSR line active again.

The DSR-option of the serial interface must be defined by:

```
Define  SerPortDSR    = PortC, 2, Positive;  // first serport
          SerPortDSR1  = PortC, 2, Positive;  // first serport
          SerPortDSR2  = PortC, 3, Negative;  // second serport if available
          SerPortDSR3  = PortC, 4, Positive;  // third serport if available
          SerPortDSR4  = PortC, 5, Negative;  // fourth serport if available
```

XMega

```
Define  SerPortDSRC0 = PortC, 2, Positive;  // first serport
          SerPortDSRC1 = PortC, 3, Negative;  // second serport if available
          SerPortDSRD0 = PortC, 4, Positive;  // third serport if available
          SerPortDSRD1 = PortC, 5, Negative;  // fourth serport if available
```

This define imports the DSR handshake and also defines the used port pin. The level statement means "enable at level".

There are no support functions for the DSR line driver option.

Handshake Rx (XON/XOFF)

As an alternate way to avoid RxBuffer overruns it is also possible to use a software handshake instead of a control line. With a pure text transfer the XON/XOFF protocol can be used, although this is not possible with binary data. The receiver signals the transmitter that its buffer is full by sending a XOFF character (#19) and re-enables receiving by sending an XON character (#17). The stop (XOFF) occurs when the buffer is $\frac{3}{4}$ full, the ready (XON) occurs when the receiver buffer is nearly empty, below $\frac{1}{4}$ buffer size. To support the XON/XOFF handshake with Rx-buffer operation the SerPort driver is extended now. If the driver sends a XOFF the external transmitter immediately stops its transmission until the drivers buffer drops below the lower threshold and it re-enables reception by sending the XON.

The XON/XOFF-option of the serial interface (Rx) must be defined by:

```
Define  RxBuffer    = 20, iData, XON;  // fist serport
          RxBuffer1  = 20, iData, XON;  // fist serport
          RxBuffer2  = 30, iData, XON;  // second serport if available
          RxBuffer3  = 20, iData, XON;  // third serport
          RxBuffer4  = 30, iData, XON;  // fourth serport if available
```

XMega

```
Define  RxBufferC0 = 20, iData, XON;  // fist serport
          RxBufferC1 = 30, iData, XON;  // second serport if available
          RxBufferD0 = 20, iData, XON;  // third serport
          RxBufferD1 = 30, iData, XON;  // fourth serport if available
```

This define imports the XON/XOFF handshake.

There are no support functions for the XON/XOFF option.

If **XON/XOFF** or **DSR handshake** is imported then the application can get informed about the stopped or free condition of the SerPort receive state. For this two Callback functions are implemented:

Procedure *onSerRxStopped;*

The system calls this function every time if the SerPort driver disables the RxBuffer by sending a XOFF or activating the DSR handshake line.

Procedure *onSerRxResumed;*

The system calls this function every time if the SerPort driver re-enables the RxBuffer by sending a XON or de-activating the DSR handshake line.

Implemented functions of the possible 4 SerPorts:

```
Procedure onSerRxStopped;           //SerPort
Procedure onSerRxResumed;          //SerPort
Procedure onSerRxStopped2;         //SerPort2
Procedure onSerRxResumed2;        //SerPort2
Procedure onSerRxStopped3;         //SerPort3
Procedure onSerRxResumed3;        //SerPort3
Procedure onSerRxStopped4;         //SerPort4
Procedure onSerRxResumed4;        //SerPort4
```

XMega

```
Procedure onSerRxStoppedC0;        //SerPortC0
Procedure onSerRxResumedC0;        //SerPortC0
Procedure onSerRxStoppedC1;        //SerPortC1
Procedure onSerRxResumedC1;        //SerPortC1
Procedure onSerRxStoppedD0;        //SerPortD0
Procedure onSerRxResumedD0;        //SerPortD0
Procedure onSerRxStoppedD1;        //SerPortD1
Procedure onSerRxResumedD1;        //SerPortD1
Procedure onSerRxStoppedE0;        //SerPortE0
Procedure onSerRxResumedE0;        //SerPortE0
Procedure onSerRxStoppedE1;        //SerPortE1
Procedure onSerRxResumedE1;        //SerPortE1
Procedure onSerRxStoppedF0;        //SerPortF0
Procedure onSerRxResumedF0;        //SerPortF0
Procedure onSerRxStoppedF1;        //SerPortF1
Procedure onSerRxResumedF1;        //SerPortF1
```

Attention as with all CallBack functions also these ones are called from an interrupt service and so the same rules as with interrupt must be obeyed: no big operations like Float, avoid long global interrupt disablings. Don't forget register savings if necessary!



AVRco Standard Driver

3.11.1.4 RS485

If the UART is being used with RS485 drivers in half-duplex mode in most cases it is necessary to control the line driver (Tristate or direction) In this case the state of the line driver must not be changed while the UART is shifting out its bits. Furthermore the state should only be altered if the TxBuffer of the system is empty. For these reasons a Define and a Procedure is implemented:

```
Define TxBuffer    = 8, iData;  
        SerCtrl     = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrl1   = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrl2   = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrl3   = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrl4   = PortD, 2, positive;    {control line for RS485 driver}
```

XMega

```
Define TxBuffer    = 8, iData;  
        SerCtrlC0    = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrlC1  = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrlD0  = PortD, 2, positive;    {control line for RS485 driver}  
        //SerCtrlD1  = PortD, 2, positive;    {control line for RS485 driver}
```

Because a RS485 support from the system only makes sense with a transmit buffer the TxBuffer must be defined. The Define "SerCtrl" determines the I/O-port, the control pin and the polarity of this pin for enable.

```
Procedure Ser_Enable (const ena : boolean);  
Procedure Ser_Enable1 (const ena : boolean);  
Procedure Ser_Enable2 (const ena : boolean);  
Procedure Ser_Enable3 (const ena : boolean);  
Procedure Ser_Enable4 (const ena : boolean);
```

XMega

```
Procedure SetSerEnable(Usart : tUSARTenum; const ena : boolean);
```

This procedure must be used to control the line driver. With ena = true the driver is immediately enabled. From this point on SerOut or Write can be used. With ena = false the system is informed that the line driver can be disabled if the last byte of the TxBuffer is stored into the UART's Tx-register and the last bit of this byte has left the UART's shift register. The state of the line driver can be found with reading back the corresponding port pin.

```
SER_ENABLE (true);  
write (SerOut, 'hello');  
SER_ENABLE (false);
```

XMega

```
SetSerEnable(UsartC0, true);  
write (SerOutC0, 'hello');  
SetSerEnable(UsartC0, false);
```

3.11.1.5 TxComplete Callback

If only the info is needed that the TxBuffer is completely empty and also the UART has send out all bits then the SerPort driver can inform the application about this. Therefore the Define/Import similar to (alternative) the \ RS485 mode must be used. The SerCtrl, the TxBuffer Define and the function Ser_Enable is necessary. The info is made by the Call Back function. RS485 drivers are not supported in this case!

```
Define TxBuffer    = 16, iData;
        SerCtrl     = onSerTxComplete1;    {necessary callback function}
        //SerCtrl2  = onSerTxComplete2;    {necessary callback function}
        //SerCtrl3  = onSerTxComplete3;    {necessary callback function}
        //SerCtrl4  = onSerTxComplete4;    {necessary callback function}
```

XMega

```
Define TxBuffer    = 32, iData;
        SerCtrlC0   = onSerTxComplete_C0; {necessary callback function}
        //SerCtrlC1 = onSerTxComplete_C1; {necessary callback function}
        //SerCtrlD0 = onSerTxComplete_D0; {necessary callback function}
        //SerCtrlD1 = onSerTxComplete_C0; {necessary callback function}
```

The concerning Callback function must be provided by the application:

Procedure onSerTxCompleteXX;

Support functions:

```
Procedure Ser_Enable (const ena : boolean);
Procedure Ser_Enable1 (const ena : boolean);
Procedure Ser_Enable2 (const ena : boolean);
Procedure Ser_Enable3 (const ena : boolean);
Procedure Ser_Enable4 (const ena : boolean);
```

XMega

```
Procedure SetSerEnable(Usart : tUSARTenum; const ena : boolean);
```

This is used to control the Callback. With ena = true the operation is started. From this point on SerOut or Write can be used. With ena = false the system is informed that if the last byte is fetched from the TxBuffer and all bits are sent by the UART, the Callback function must be called..

```
Ser_Enable (true);
write (SerOut, 'hello');
Ser_Enable (false);
```

XMega

```
SetSerEnable(UsartC0, true);
write (SerOutC0, 'hello');
SetSerEnable(UsartC0, false);
```

If in addition an RS485 driver must be controlled so the application must enable it at SerEnable time and disable it in the Callback function.

Attention as with all CallBack functions also these ones are called from an interrupt service and so the same rules as with interrupt must be obeyed: no big operations like Float, avoid long global interrupt disablings. Don't forget register savings if necessary!



AVRco Standard Driver

3.11.2 Port Switching

With AVRs containing several UARTs it's possible to dynamically switch between the multiple SerPorts. If correctly imported there is a new byte "SerPortSelect". It controls the selection of the interface of the functions SerOut, SerInp, SerStat and FlushBuffer.

For example if SerPortSelect has the value 1 all SerOut calls are redirected to SerPort2, and so are SerInp to SerInp2 and SerStat to SerStat2.

This routing is also valid for the functions Write, WriteLn and Read etc.

Basically this is true: if SerOut, SerInp, SerStat etc. is used here the SerPortSelect routes the selected port. But if a real port is selected with SerInp2 for example the SerPort2 is explicitly used.

In order to enable this switching both SerPorts must be imported. Then in addition the import of "SerPortSelect" is also possible.

```
Device = mega2560, VCC = 5;  
Import SysTick, SerPort, SerPort2, SerPort3;  
From SerPort import SerPortSelect;
```

Now the byte "SerPortSelect" is accessible.

```
SerPortSelect:= 1;    // switch to serport2  
SerOut ('x');        // write 'x' to serport2  
if SerStat then      // check serport2  
  ch:= SerInp;        // read from Serport2  
endif;
```

```
SerPortSelect:= 2;    // switch to serport3  
SerOut ('x');        // write 'x' to serport3  
if SerStat then      // check serport3  
  ch:= SerInp;        // read from Serport3  
endif;
```

```
SerPortSelect:= 0;    // switch to serport  
SerOut ('x');        // write 'x' to serport  
if SerStat then      // check serport  
  ch:= SerInp;        // read from Serport  
endif;
```

```
SerPortSelect:= 0;    // switch to serport  
FlushBuffer (RxBuffer); // clear the selected RxBuffer
```

XMega

Here the variable *SerPortSelect* is not a byte but of type tUSARTenum

```
SerPortSelect:= UsartC1;    // switch to serportC1  
SerOut ('x');              // write 'x' to serportC1  
if SerStat then            // check serportC1  
  ch:= SerInp;              // read from SerportC1  
endif;
```

```
SerPortSelect:= UsartD0;    // switch to serportD0  
SerOut ('x');              // write 'x' to serportD0  
if SerStat then            // check serportD0  
  ch:= SerInp;              // read from SerportD0  
endif;
```

```
SerPortSelect:= UsartC0;    // switch to serportC0  
SerOut ('x');              // write 'x' to serportC0  
if SerStat then            // check serportC0
```

```
ch:= SerInp;           // read from SerportC0  
endif;
```

For example if SerPortSelect has the value UsartC1 all SerOut calls are redirected to SerPortC1, and so are SerInp to SerInpC1 and SerStat to SerStatC1.

This redirection is also valid for drivers and functions using the SerPorts like Write, WriteLn, Read etc.

Basically this is true: if SerOut, SerInp, SerStat etc. is used here the SerPortSelect routes the selected port. But if a real port is selected with SerInpC1 for example the SerPortC1 is explicitly used.

3.11.3 UART enable and disable for **XMegas**

With some cases it maybe necessary to disable and re-enable the Rx part, the Tx part or both of an UART. The system provides these functions to accomplish this:

```
procedure UartEnableRxXX( Ena : boolean); // Rx hardware enable/disable  
procedure UartEnableTxXX( Ena : boolean); // Tx hardware enable/disable
```

where **XX** stays for C0, C1, D0, D1 etc.



AVRco Standard Driver

3.11.4 IRDA for XMegas UARTs (IRcom)

The XMega UARTs provide an IRDA support. IRDA = **I**nfra**R**ed**D**ata**A**ssociation. With IRDA data can be exchanged via infrared light. There are several low-Level protocols and also some protocol stacks/layers based on IRDA. Our implementation uses the simplest and lowest level of IRDA.

This SIR protocol (**S**erial**I**nfra**R**ed) is an asynchron protocol which is also used by all UARTs. But the pulse widths are reduced here to 1/6 to save power consumption.

Basically the SIR works with 115.2kBaud, 1 Startbit, 8 Databits, 1 Stopbit and no parity.

All these requirements are fulfilled by the IRDA hardware in the XMega UARTs. But this hardware can only be used once in an XMega, but is applicable with each UART.

IRDA/SIR works as the same as a MAX232 except that the data is not sent over lines but over invisible infra red light. As a result there is the limitation to half-duplex, which means that only one device can send at the same time. Normally the usable destination with IRDA is >1m. Using a higher infrared tx-power it is possible to extend this to >10m.

Because of the hardware support of the UARTs the IRDA is outclassing the RC5! Furthermore in contrast to RS232 there can be multiple participants. It can be a simple Bus system.

If an IRDA transceiver should be used then the driver must be setup for this with this define:

Define `SerPortC1 = 115200, IRDA;`

Nearly all SerPort functions can be used with IRDA.

With a bi-directional operation only the half-duplex mode is allowed. So only one master can send data at one time. Furthermore all IRDA transceiver always receive their own Tx-data which of course is not welcome. So with a transceiver mode it must be clear that the own Rx must be disabled when sending data.. If the data is completely sent, the receiver must be re-enabled. To support this there are two system functions:

Procedure `IRDATxStartXX;`

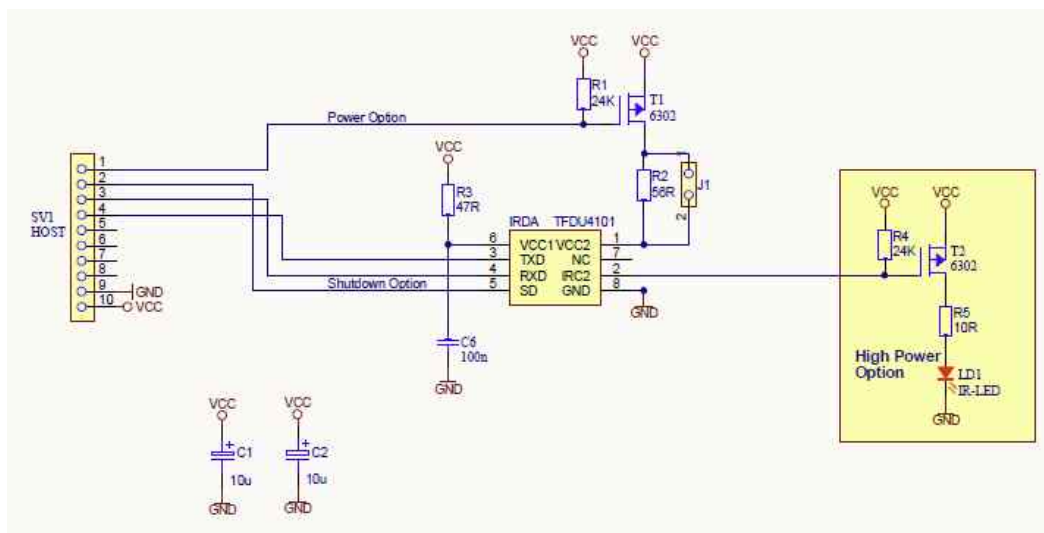
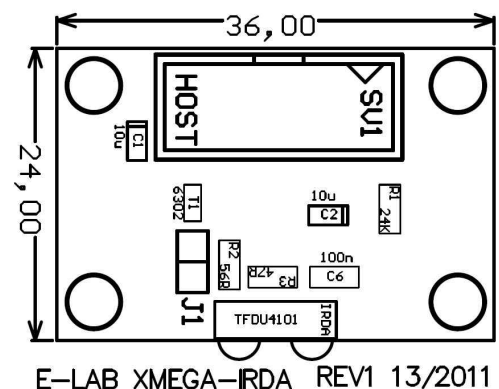
Procedure `IRDATxWaitIdleXX;`

The **XX** stands for the IRDA UART (C0..F1).

Example program:

two examples are in the directory

..E-Lab\AVRco\Demos\XMega_IRDA



3.11.5 Advanced Functions for Controller with USART

The main improvements of controllers with USART are as follows:

DataBits : 5/6/7 or 8 bits. The functionality of the multiprocessor-bit (bit9) is also extended.
 StopBits : 1 or 2. Previously only possible by using the 9th bit.
 Parity : none/even/odd. Previously also only possible by using the 9th bit and must be calculated by the driver.

3.11.5.1 Types, Procedures and Functions

If an AVR CPU has an USART the AVRco system exports some additional types and functions that don't exist with standard UARTs.

Types

For a better handling and readability of the additional functions with the USART the system exports these types:

Type *tParity* = (*parNone*, *parEven*, *parOdd*);
Type *tDataBits* = (*DataBit5*, *DataBit6*, *DataBit7*, *DataBit8*);
Type *tStopBits* = (*StopBit1*, *StopBit2*);

Procedures and Functions

With USARTs it is no problem to change several parameters at runtime.
 This is done by the following procedures:

```
Procedure SerStopBits (bits : tStopBits);           // Stopbits (1/2)   COM1
Procedure SerStopBits1 (bits : tStopBits);         // Stopbits (1/2)   COM1
Procedure SerStopBits2 (bits : tStopBits);         // Stopbits (1/2)   COM2
Procedure SerStopBits3 (bits : tStopBits);         // Stopbits (1/2)   COM3
Procedure SerStopBits4 (bits : tStopBits);         // Stopbits (1/2)   COM4
Procedure SerDataBits (bits : tDataBits);          // Databits (5/6/7/8) COM1
Procedure SerDataBits1 (bits : tDataBits);         // Databits (5/6/7/8) COM1
Procedure SerDataBits2 (bits : tDataBits);         // Databits (5/6/7/8) COM2
Procedure SerDataBits3 (bits : tDataBits);         // Databits (5/6/7/8) COM3
Procedure SerDataBits4 (bits : tDataBits);         // Databits (5/6/7/8) COM4
Procedure SetParity (par : tParity);               // Parity (N/E/O)   COM1
Procedure SetParity1 (par : tParity);              // Parity (N/E/O)   COM1
Procedure SerParity2 (par : tParity);              // Parity (N/E/O)   COM2
Procedure SerParity3 (par : tParity);              // Parity (N/E/O)   COM3
Procedure SerParity4 (par : tParity);              // Parity (N/E/O)   COM4
Procedure SerBaud (var baud : word);               // Baudrate var Parameter COM1
Procedure SerBaud (const baud : byte..longword);  // Baudrate const Parameter COM1
Procedure SerBaud1 (var baud : word);              // Baudrate var Parameter COM1
Procedure SerBaud1 (const baud : byte..longword); // Baudrate const Parameter COM1
Procedure SerBaud2 (var baud : word);              // Baudrate var Parameter COM2
Procedure SerBaud2 (const baud : byte..longword); // Baudrate var Parameter COM2
Procedure SerBaud3 (var baud : word);              // Baudrate var Parameter COM3
Procedure SerBaud3 (const baud : byte..longword); // Baudrate var Parameter COM3
Procedure SerBaud4 (var baud : word);              // Baudrate var Parameter COM4
Procedure SerBaud4 (const baud : byte..longword); // Baudrate var Parameter COM4
XMega
Procedure SetSerStopBits(Usart : tUSARTEnum; bits : tStopBits); // Stopbits (1/2)
Procedure SetSerDataBits(Usart : tUSARTEnum; bits : tDataBits); // Databits (5/6/7/8)
Procedure SetSetParity(Usart : tUSARTEnum; par : tParity);      // Parity (N/E/O)
Procedure SetSerBaud(Usart : tUSARTEnum; const baud : byte..longword); // Baudrate const Parameter
```

Please note that these parameter changes at runtime can result in 1..2 wrong characters if the changes are made when a receive operation is going on.



AVRco Standard Driver

3.11.6 SLIP packet oriented Protocol

Networks basically work with so called packets or frames. This is a block of data which is framed (limited) by a start and end condition. This is absolutely necessary to have a secure data transmission. These frame limiters can be constructed in different ways. For example a well defined pause between two data blocks like ModBUS RTU or there are special control bytes if the data is plain ASCII (A..Z, a..z, 0..9) like (ModBUS ASCII).

With more complex protocols like CAN, Ethernet, I2C/TWI etc. the framing is explicitly implemented in the driver hardware. Without such framing it is a difficult task to identify incoming frames on the receiver side. But also with timeout-controlled protocols this is not trivial. On both sides timers and interrupts are involved to avoid unexpected time slots or timeouts. With ASCII protocols there is a big overhead on both sides to convert binary data into ASCII chars (e.g. HEX) and vice versa.

But with simple bit-serial interfaces like UARTs and SPI building frames, clearly recognizing them is a huge problem. For this purposes there is a powerful protocol called SLIP.

Serial Line Internet Protocol

The word "Internet" can easily be replaced by "Interchange" or "Interface" because SLIP basically and only defines the frame-start and frame-end handling. All the other stuff must be handled by a processing layer above SLIP.

SLIP provides a secure recognition of the frame start and frame end condition and extracts the pure data packet for the application. The data contained in the packet can be of any type and is not changed or analysed by SLIP.

The AVRco system provides a SLIP implementation in conjunction with the serial interfaces (UARTs) 1..4 where they are present in the CPU. There is only one transmit function and one receive function per channel.

```
Procedure SerOutSLIP(src : pointer; count : word); // UART1
Procedure SerOutSLIP1(src : pointer; count : word); // UART1
Procedure SerOutSLIP2(src : pointer; count : word); // UART2
Procedure SerOutSLIP3(src : pointer; count : word); // UART3
Procedure SerOutSLIP4(src : pointer; count : word); // UART4
```

XMega

```
Procedure SerOutSLIPC0(src : pointer; count : word); // UARTC0
Procedure SerOutSLIPC1(src : pointer; count : word); // UARTC1
Procedure SerOutSLIPD0(src : pointer; count : word); // UARTD0
Procedure SerOutSLIPD1(src : pointer; count : word); // UARTD1
Procedure SerOutSLIPE0(src : pointer; count : word); // UARTE0
Procedure SerOutSLIPE1(src : pointer; count : word); // UARTE1
Procedure SerOutSLIPF0(src : pointer; count : word); // UARTF0
Procedure SerOutSLIPF1(src : pointer; count : word); // UARTF1
```

These procedures send a data packet through the selected serial interface/UART. Parameter **src** defines the data and must be a pointer. The parameter **count** defines the packet size in bytes. The procedure returns if all data bytes are placed at least in the TxBuffer of the UART. The parameter count must never be zero.

```
Function SerInpSLIP(dst : pointer; tmo : byte; count : word) : word; // UART1
Function SerInpSLIP1(dst : pointer; tmo : byte; count : word) : word; // UART1
Function SerInpSLIP2(dst : pointer; tmo : byte; count : word) : word; // UART2
Function SerInpSLIP3(dst : pointer; tmo : byte; count : word) : word; // UART3
Function SerInpSLIP4(dst : pointer; tmo : byte; count : word) : word; // UART4
```

XMega

```
Function SerInpSLIPC0(dst : pointer; tmo : byte; count : word) : word; // UARTC0
Function SerInpSLIPC1(dst : pointer; tmo : byte; count : word) : word; // UARTC1
Function SerInpSLIPD0(dst : pointer; tmo : byte; count : word) : word; // UARTD0
Function SerInpSLIPD1(dst : pointer; tmo : byte; count : word) : word; // UARTD1
Function SerInpSLIPE0(dst : pointer; tmo : byte; count : word) : word; // UARTE0
Function SerInpSLIPE1(dst : pointer; tmo : byte; count : word) : word; // UARTE1
```

Function SerInpSLIPF0(dst : pointer; tmo : byte; count : word) : word; // UARTF0

Function SerInpSLIPF1(dst : pointer; tmo : byte; count : word) : word; // UARTF1

These functions wait for an incoming data packet until either it is completely received or the timeout has expired. The parameter **dst** defines the destination address of the received data and must be a pointer of any type. The parameter **tmo** defines the timeout in SysTick counts for each single byte to receive. The parameter **count** defines the maximal possible byte count to receive (limit). If the packet is larger as this value the function returns with an error, result = 0. If a timeout happened the function also returns with a zero value. If successful the result is the packet size in bytes.

In order to work as expected the driver of the used SerPort must be imported and defined. Furthermore the Timeout of this SerPort must be activated. Please note that a SerPort buffer can only be max. 254 bytes in size. This means that a packet can only have a size of max. 252 data bytes because there are two additional frame bytes. In addition the SLIP algorithm must always replace special data bytes by two replacement bytes. This means with worst cases that a RxBuffer with a size of 254 bytes only 126 effective bytes can be transmitted.

But with transmitting there is never a problem even with large packets because the driver waits until all bytes are stored into the TxBuffer.

So there is a problem with receiving large packets. If the application is unable to continuously poll the corresponding RxStat function and then does not immediately call SerInpSLIP the RxBuffer Interrupt driver ignores additional Rx-Bytes and this frame becomes invalid.

There will be absolutely no problems if the RxBuffer is at least twice the size of the largest expected Rx-Frame.

RS485 Line Driver

If the corresponding SerPort uses the RS485 mode then the SerOutSLIP driver handles this automatically without intervention of the application.

SLIP sample:

```
program SLIPtest;
Device = mega128, VCC = 5;
Import SysTick, SerPort;
Define
    ProcClock           = 16000000;           {Hertz}
    SysTick              = 10;                 {msec}
    StackSize            = $040, iData;
    FrameSize            = $060, iData;
    SerPort              = 115200, Stop2, TimeOut;
    RxBuffer             = 64, iData;
    TxBuffer             = 64, iData;
```

Implementation

```
{ $IDATA }
Var
    Test      : word;
    ptrRx     : pointer;
    ptrTx     : pointer;
    BuffRx    : array[0..15] of byte;
    BuffTx    : array[0..15] of byte;

begin
    EnableInts;
    BuffTx[2] := 192;
    BuffTx[6] := 219;
    ptrTx := @BuffTx;
    ptrRx := @BuffRx;
    SerOutSLIP(ptrTx, 8);
    mDelay(20);
    // if RxStat then ...
    test := SerInpSLIP(ptrRx, 10, 8);
end SLIPtest.
```



AVRco Standard Driver

3.12 Serial Network LAN

Not for **XMegas**, use SlipPorts instead.

Distributed Intelligence is not only a slogan. Distributed computing power within a system yield enormous advantages: less wiring, more computing power at the same time and simplified programs. If a computer or a CPU is placed where it is needed, and often there are several such places in a system, then communication between the individual programs/CPU must be established.

This data transfer normally is made by the implementation of a Local Area Network (LAN). There are several ways to realize such a network. Most of them need a special network controller (Ethernet, ArcNet, InterBus-S, ProfiBus, CAN-BUS etc.) or at least a CPU, in which the controller is build-in (e.g. CAN-Bus). These controllers have a good link to the network.

Unfortunately often there is no place for an additional controller circuit or the price is too high. CPUs with build-in controllers are rare at this time or the pincount or case size is too high, plus they are not cheap.

Because most of the single chip CPUs have a serial interface (UART, SCI), it's a good idea to use them for networking purposes. Old mainframes used the async serial interface many years ago for networking.

All of the usable Atmel AVR-CPU's have an UART. A special feature of them is that the transmission can use the ninth bit as a data bit. This 9th bit can be used as a marking for the address of a frame. Some 8051 derivatives also have this feature.

The present implementation is intended mainly for such CPU-CPU connections. The strategy consists of a Single-Master and Multi-Slave system. For a secure communication an appropriate protocol is used.

Basics

A Single-Master / Multi-Slave system is implemented which consists always of a Master and at minimum one Slave.

The Master controls all connected Slaves. No Slave should gain control of the BUS, although this is always possible. All data transfer, whether to or from the Slave should be initiated by the Master. By this means serious difficulties with collisions and priorities are eliminated.

Networks commonly work with packets also called frames. Through the use of frames there is some overhead, but on the other hand the security of the system is increased. To avoid additional problems, there is no full duplex transfer. In that way the cheap and secure RS485 line drivers can be used.

With the half-Duplex operation only one subscriber can send data at one time. All others are listening. For example the master requests a frame from a slave. For that it sends a frame to the slave with the desired contents (address, data, commands). The slave recognizes its address and informs the user program about the reception of the frame. If the user program in the slave must send an answer to the master (not a must), it responds by sending a frame, which inherent address is its one Node address. All other slaves ignore this frame. The Master places the received frame in its buffer and informs the application.

The contents and meaning of a frame is ambiguous and has to be defined by the user, as does whether a slave responds with a responding frame or not. If the master is waiting for an answer, the slave should respond immediately because the master is waiting and sends no further frames until an answer is received to avoid BUS collisions. But this "handshaking" must be done by the application. It is not handled by the master or slave software itself.

Principles of operation:

The **Master** receives all frames that are sent on the BUS and provides it to the application program. There is no analysis or utilization in the master except the checksum checking if any.

Each **Slave** always listens on the BUS, but fetches only frames if the frame inherent address is identical with its own address (node address). There is no analysis or utilization in the slave except the checksum checking if any. It's the job of the application to do this.

The slave doesn't respond with an acknowledge or an answer to the master. This is also the job of the application. There is one exception: if the slave is in **AutoAck** mode. In this case the slave responds with a short frame without contents to the master. It consists of the slave's address, FrameLen(0) and checksum if any. This also is only true if the received Frame had no errors.

Slave addresses must be greater than 0. The address "0" is reserved for the Broadcast to all slaves. Only the master should send a Broadcast because of collisions. All slaves receive the frame, and their userprogram must analyze and interpret it. But there must be no reaction (sending a response frame) of any slave. Also AutoAck is disabled in this case.

The actual frame length (bytes) of a frame can range from 0 bytes up to LANframe - 1. The addresses can be 8 or 16 bits (selected by a Define). The checksum (also determined by a define) can be: none, 8 Bit Check, 16 Bit Check or 16 Bit CRC-Check.

Variable is only the actual frame length. Address length/size and checktype must be determined by defines and can't be changed at runtime.

Structure of a frame:

| ADDRESS | FRAMELENGTH | DATA, DATA ... | CHECKSUM |

Address:

The address can consist of a byte (1..255 slaves) or a word (1..65535 slaves). Normally a byte address is sufficient, because the usual RS485 line drivers can only handle 255 participants at maximum. Define LANadr determines the address width.

Each slave should have a unique address, but there may be exceptions. The master has no address, because it receives all frames, regardless of the inherent address. The address bytes are sent with the 9th Bit set to one. If two byte addresses are used, the low byte is sent first.

FrameLength:

The frame length is determined by define LANframe = nn. If parameter nn < 256 there is one length byte transferred, if nn ≥ 256, 2 length bytes are transmitted. The actual frame length can be between 0 and nn bytes. Because of this, all BUS subscribers must have the same kind of address size (Byte/Word), so the frame descriptors are always identical. The implemented buffer sizes can vary in the range of a byte or word.

DATA:

The data area within a frame is defined by the length byte or word. The data length can be zero so the length byte/word also contains a zero.

CHECKSUM:

The kind of the checksum is a setup option. The check is computed over the complete frame. There can be an 8bit, a 16bit checksum, or a 16bit CRC.

Attention:

The address width, the max. frame length and the checksum type must all be defined at design time with the use of "Define", as must the Master/Slave mode. A change at runtime is not possible. The actual frame buffer sizes can vary in the participants but only in the range of 0..FrameLength. Either all nodes possess a buffer size smaller than 256 Bytes (FrameLength is in one byte) or greater than 255 bytes (FrameLength is in one word). A mix is not allowed.



AVRco Standard Driver

Transfer:

The data transfer is completely handled in interrupt procedures. Therefore it is guaranteed if a frame is completely built, this frame is transferred to the target in a very short time. It is important that interrupts must not be disabled for a long time. This is especially true at high baud rates. At 9600Bd each msec a receiver/transmitter interrupt occurs. At 56kBd this time is 160usec. In the last case for example, disabling the interrupt (e.g. SysTick) for more than 100usec can cause bytes to be lost.

The used baudrate of the net must be identical in all slaves, but need not be in the standard grid (9600, 19200 etc.).

Line driver control:

Regardless whether there are wireless modems, RS485 lines or TTL line drivers, all possess the same property: they are not full duplex. Either there is a transmit going on or a receive, but never at the same time like RS232 does. Therefore each BUS participant must have a tristate or switch off type line driver for listening only. This switching is done with a Port Pin, which is declared by "Define LANctrl = Port, Bit;". The pin is controlled by the system. The application has nothing to do with this.

3.12.1 Implementation

Imports:

The LANport must be imported, like usual in the AVRco system, by an import directive.

Import *LANport;*

Defines:

LANport

Declaration of the used UART

Define *LANport = SerPort; {SerPort2}*

LANctrl

Definition of the control port and pin for the line driver. As an option also None is possible.

Define *LANctrl = PortA, 5; {PortName, bit number}*

LANmode

The operation type of the unit as a master or slave must be defined. A change of this type at runtime is not possible

Define *LANmode = Master; {Master/Slave}*

LANbaud

The operation modes of the LAN must be identical in the Master and Slave and must be defined by the following defines:

Define *LANbaud = 57600; {Baudrate}*

The baudrate can be arbitrary but the same in the whole net. The typical RS232 tolerance of +/- 5% must not be exceeded. The lower the error the higher the possible baudrate and cable length.

LANadr

The address size (Byte/Word) determines the number of possible slaves. Although it's possible to implement more than 255 slaves (LANadr = 16) this is unrealistic. Simple network interfaces (e.g. RS485 driver) are unable to drive more than 128 receivers. With wireless modems this is not a problem.

Define *LANadr = 8 [,Mask]; {8 or 16 bit, [masked by LANADRMASK] }*

If "Mask" is omitted a mask is not defined and used. If the import is present a variable "LANADRMASK" of type Byte or Word is created, dependant of the defined address size (8/16bit). This variable can be accessed:

LANADRMASK := \$0F;

All high (log 1) bits are treated as DON'T CARE. This means that these bits are ignored in the address comparator when the incoming frame address is compared with the local node address.

LANframe

Master and Slave both have a receive- and a transmit buffer. All slaves should possess the same buffer size as the master. The defined buffer size determines the required internal memory. The transferred frames can be less than but never larger than this parameter.

Because the actual frame length is transmitted as a parameter in the frame, all frames must be managed with a byte or a word as the length information. A mix of them is not possible. This limitation simplifies the protocol and its handling dramatically and is not a serious limitation. All frame size definitions therefore must fit either into a byte or a word, depending on whether the frame is smaller than 256 bytes or larger.

Define LANframe = 16, iData; {Framesize max. 16 bytes in iData}

LANcheck

A check of the received frames is very important with networks. It is recommended to use the 8bit check. If the frames are relatively long (>50 bytes), the 16bit check should be selected. The CRC check is only recommended with very long frames and/or noisy environment. CRC checks consume a great deal of time and code space Code.

Define LANcheck = ChkSum8; {ChkSum8, ChkSum16, CRC16}

3.12.2 Exported Variables

3.12.2.1 Memory Organization

All the LAN related variables reside one after another in the defined memory area. The order is, beginning with the lowest address, the following:

LANNODEADR	Byte at < 256 LAN nodes, Word at > 255 LAN nodes. SLAVE only
_LANRXPTR	Byte at a frame length < 256 bytes, word at a frame length > 255 bytes
LANRXSTATREG	Byte
LANRXADR	Byte at < 256 LAN nodes, Word at > 255 LAN nodes
LANRXLEN	Byte at a frame length < 256 bytes, word at a frame length > 255 bytes
LANRXBUFF	Array[0..LANframe-1] of Byte;
_LANRXCHK	byte if ChkSum8, word if ChkSum16/CRC16
_LANTXPTR	Byte at < 256 LAN nodes, Word at > 255 LAN nodes. SLAVE only
LANTXSTATREG	Byte
LANTXADR	Byte at < 256 LAN nodes, Word at > 255 LAN nodes
LANTXLEN	Byte at a frame length < 256 bytes, word at a frame length > 255 bytes
LANTXBUFF	Array[0..LANframe-1] of Byte;
_LANTXCHK	byte if ChkSum8, word if ChkSum16/CRC16

LANnodeAdr

This variable is only present in the slave. The type is of byte at < 256 LAN nodes and a word with > 255 LAN nodes. If the slave recognizes an address (9. Bit set) it compares the incoming frame address with the variable "LANnodeAdr". If both are equal the frame is received completely and stored into the locations LANrxAdr.._LANrxChk. Addresses with the value "0" are recognized as Broadcast Frames and always stored, but there should never be a response from the slave.



AVRco Standard Driver

Having more than one slave with the same address should be avoided. The address should be unique. If there are several slaves with the same address, none of these devices must respond to a frame from the master, otherwise BUS collisions are guaranteed. There is no simple way to handle such collisions.

_LANRxPtr, _LANTxPtr

These two variables (Byte or Word) must not be changed! They are for internal use only while a receive or transmit is occurring. They are byte sized if "Define LANframe" < 256 bytes or word sized with "Define LANframe" > 255 bytes.

LANRxStatReg, LANTxStatReg

These two variables (Byte) must only be read! Both pseudo status registers are continuously updated while a receive or transmit of a frame is going on. If idle, all bits are zero. If the action proceeds the bits become a "1" in turn from right to left. If the action is finished successfully, all bits are "1" and the byte has the value \$FF. Values between \$00 and \$FF mean either an error has occurred or that the frame isn't completely received or transmitted yet. If a send frame is processed, there can be no error. If a frame was received a \$9F means that a checksum error occurred. The user program must poll the bit7 of this register to see if a frame is complete or not. A correct and complete frame shows a \$FF.

```
(* bit0,1      : 00 = idle          *)
(*             01 = first adr       *)
(*             11 = second adr      *)
(* bit2,3      : 00 = idle          *)
(*             01 = first len       *)
(*             11 = second len      *)
(* bit4        : 0 = rx/tx frame    *)
(*             1 = rx/tx frame      *)
(* bit5,6      : 00 = idle          *)
(*             01 = first check     *)
(*             11 = second check    *)
(* bit7        : 0 = processing     *)
(*             1 = rx/tx complete   *)
```

LANRxAdr, LANTxAdr

These vars are of type byte if "Define LANadr = 8" and a word if "Define LANadr = 16"

Master:

The master receives all frames, which were send by another BUS-member. The user's program can identify the sender with the contents of the var "LANRxAdr".

If the master sends a frame, it must define the desired slave by writing the slave's address into the variable "LANTxAdr". The address "0" defines a Broadcast Message, this means all members of the network receive this frame, process it but never send a response (danger of collision).

Slave:

A slave only receives those frames whose internal address is equal to its own node address. This inherited address of the incoming frame is stored into the location "LANRxAdr". Normally this address is the same as the node address of the slave or it is a Broadcast frame. The user's program must check this address to find out whether it's a unique frame or a Broadcast frame and makes the relevant decisions.

In order to transmit a frame the Node address of the slave is stored into "LANTxAdr". The slave only should send frames with its own node address. Although it's possible that a slave can send Broadcast messages or it can address another slave, this is very dangerous and must be arranged with the Master to avoid any collisions on the BUS.

LANRxLen, LANTxLen

These variables are of type byte if "Define LANframe" < 256 bytes and of type word if "Define LANframe" > 255 bytes. Frames can have a zero length. The maximum length cannot exceed the number of "Define LANframe = nn". Basically all slaves and also the master should have the same "LANframe" sizes. But it's possible that some slaves have a shorter value of "LANframe". The master's user program must know this and only send frames to this nodes with a length so that they fit into the receive buffers of that slaves.

LANtxLen contains at send and receive the **effective length of the telegram**. Address, Checksum, etc. are do not count.

LANrxLen is updated whilst receiving a frame with the value which is received internal in the frame.

LANtxLen is set by the function "LANtxFrame(Node, len)" and also transmitted within the frame.

LANRxBuff, LANTxBuff

The size of these two buffers is determined by the Define LANframe. The buffers can be seen as an Array of byte and therefore can be also accessed in this manner.

The indexes of this arrays are 0 to LANframe -1.

```
LANtxBuff[0]:= $56;
```

```
X:= LANrxBuff[6];
```

It's recommended and very useful to overlay a structure (record) over the Rx part and the Tx part of the LAN memory. By that a symbolic access to the complete data structure is possible.

```
type tLANRec = record
```

```
    LANstate : byte;      // LAN state size and loc fixed  
    LANnode  : byte;      // rx/tx address size and loc fixed  
    LANlen   : byte;      // rx/tx framelen size and loc fixed  
    LANusr1  : byte;      // user defined  
    LANusr2  : word;      // user defined  
    LANdata  : array [0..LANframe-4] of char; // user  
end;
```

```
var
```

```
    LANRxRec[@LANrxStatReg] : tLANRec;
```

```
    LANTxRec[@LANtxStatReg] : tLANRec;
```

```
// Frame belegen
```

```
    LANTxRec.LANusr1 := $30;
```

```
    LANTxRec.LANusr2 := $3231;
```

```
    LANTxRec.LANdata[0] := '3';
```

_LANRxChk, _LANTxChk

Their type is a byte if ChkSum8 and a word if ChkSum16/CRC16 is defined. The checksum is calculated over the complete frame (LANRXADR + LANRXLEN + LANRXBUFF). The same is true for both Tx and Rx. The checks are processed automatically while receiving and transmitting.

3.12.3 Exported Functions and Procedures

```
Function LANRxStat : boolean;
```

This function returns a true if a frame has been received complete with or without errors and is not invalidated with "LANrxClear". Then the program must analyze the LANrxStatReg to check for errors. As long a frame is complete, also with errors, a new incoming frame is discarded. It's the job of the user program to process a received frame as fast as possible and then invalidate it so a new frame can be received.

Otherwise frames can be lost.

If the user's program in the master and slave are working with an acknowledge, this means if the master sends a frame and then waits for a response from the slave, no more frames are sent by the master until the slave sends an acknowledge. This way no frames can be lost, but the throughput is somewhat (or greatly) reduced. It is the job of the participating devices and/or the programmer to implement such a method.

```
Procedure LANRxClear;
```

The statusbyte of the RxBuffer is reset. Another frame can be received.



AVRco Standard Driver

Function *LANtxStat* : *boolean*;

This function checks whether a frame is already send or not. If the frame is send or invalidated by LANtxClear the function returns with a true, otherwise a false is returned. A new frame can only be send if the actual frame is completely transmitted or the state is reset with "LANtxClear".

Procedure *LANtxClear*;

The status byte of the TxBuffer is reset. Another frame can be transmitted.

Function *LANtxFrame* (*node* : *byte[word]*; *len* : *byte[word]*) : *boolean*;

If the transmit operation was successful a true is returned, otherwise a false is returned. With LANtxClear a frame can be reset without deleting the data and LANtxFrame can be called again. If the frame can be transmitted the "len" parameter is stored in "LANtxLen". The parameter "node" should have always the same value as the slave's variable "LANnodeAdr". It is stored into "LANtxAdr". The function then starts a transmit operation by enabling the TxInterrupt.

Procedure *LANrxAutoAck* (*const OnOff* : *boolean*);

This procedure only exists in slave mode. If the AutoAck is enabled, the slave immediately responds with an empty frame to the master (LANtxLen = "0"). The slave uses its Txframe for this. Then the master finds out whether a slave is still "alive" and received the frame without errors. With a Broadcast frame no acknowledge will be sent.

3.12.4 Multi-Processing

If processes are used, sometimes it is necessary to suspend a waiting process until a frame is received. This can be done with "WaitPipe(LANrxBuff)". The process waits until the first byte of the frame is received. Then the process must poll via LANrxStat and/or LANrxStatReg.

3.12.5 Line Drivers

Basically the LANport uses the CPU internal asynchronous serial interfaces. They must have a controllable 9th bit to distinguish between addresses and data.

V24/RS232 line driver (e.g. MAX232) are not suitable for this application because they don't have the tristate property. All line drivers, including the master's, are tristated if the device is idle, so every member of the system is always listening. Only if a frame has to be transmitted the driver is activated and gains control of the BUS.

Normally if a wired connection between the participants exists the RS485 line drivers are used. They are cheap, robust, very fast and very long distances are possible. They fulfill the important tri-state property very well. A simple two-wire connection can be used. Common ground is not absolutely necessary. A good idea is to use twisted pair cable, but this also is not necessary. Even shielding can be eliminated, even with a speed > 100kBaud.





AVRco Standard Driver

3.13 USBport Driver USBsmart **XMega**

Embedded systems often need a connection to a host, eg a PC. For a long time this was the serial interface through a COMport/UART. Today's PCs and mostly Laptops and Notebooks often don't provide a COMport. It is expected that every peripheral device provides an USB interface.

There are several solutions to realise an USB connection with an AVR:

1. a FTDI Chip on board which builds a virtual COMport on the PC. On the side of the AVR the FTDI behaves like a MAX232. Disadvantage: board place, costs and a possible conflict with COMports on the PC side.
2. HID and USB-AVRs. Simulation of a mouse or a keyboard on the AVR side. Advantage: cheap, a minimal cost on the PC side. Drawback: weak properties of the system.
3. CDC and USB-AVRs. The system provides a virtual COMport on the PC, similar to FTDI. Advantage: only an INF file must be constructed for the PC side, but this is absolutely not trivial. Drawback: conflict with other COMports because it is not sure that the device is always located on the same COMport and so it must always be searched.
4. Unique USB and USB-XMegs. This is the most elegant but also the difficultest way for an USB connection. Here the XMega is recognised as an absolutely new and unique USB device. Drawback: a DLL, a SYS and an INF file for the PC must be build and provided. Very difficult. The communication on the PC is much more complex compared with the other solutions. Advantage: all features of the USB can be used: sending commandos through the controlpipe (Endpoint0), one data pipe with 64 bytes, flexible packet sizes. High speed, upto 500kByte/sec! All drivers and files for the PC are build by two mouse clicks!

The implementation provided here use the unique interface described in 4. On the AVR side the XMega-USB driver of the AVRco system provides powerful but simple functions. All XMegas with internal USB interface are supported. Interrupts are not used here and the Endpoint count is limited to one. So a very small (<2kB) but powerful and fast driver is provided. Because there are no Interrupts all USB functions must be polled.

Control Pipe. The Control Pipe (endpoint0) can be used to force a real hardware reset on the AVR send by the PC. Furthermore the Control Pipe can pass private commands to the AVR. This data is passed via a Callback function to the AVR application. Also small data can be received or transmitted here. In order to satisfy the USB requirements the application should continuously poll the control pipe (Endpoint0).

Simple Interface. There is one 64byte sized pipe provided (Endpoint1 = Rx/Tx). The important data transfer must be accomplished through this endpoint. This is done by receiving or transmitting data blocks with upto 64 bytes. In order to satisfy the USB requirements the application should continuously poll these Endpoints.

3.13.1 Import of the USB Driver

As always in the AVRco System the driver must be imported:

Import SysTick, USBsmart, ...; // smart version oft the XMega USB driver

The SysTick is optional and not needed here.

3.13.2 Definition of the USB Driver

The USB driver needs some essential Defines which determine the way the driver connects to the Host.

```
// The XMega USB occupies the internal 32MHz source, so the internal 2MHz osc must be used here
OSCtype          = int2MHz,
                  PLLmul = 16,
                  prescB = 1,
                  prescC = 1;      // CPU runs with 32MHz

SysTick          = 10;            // msec
StackSize        = $0064, iData;
FrameSize        = $0064, iData;

USBmanufact      = 'E-Lab Computers'; // max 31 bytes
USBprodName      = 'xMiniUSBApp';     // " "
USBpid           = 30;                // product ID
USBvid           = $1234;             // vendor ID
USBprodRel       = 201;               // product release
USBcurrent       = 200;               // current consumption, max 500mA
USBvBUS          = PortB.7;           // port and pin to check the USB power connection. optional
```

SysTick is optional.

USBvid is the Vendor/Manufacturer ID. Provided from the USB.org (costs).

USBpid is the Product ID. Can be any.

USBprodRel is the actual version of the product. Can be any.

USBmanufact is the name of the producer as a string.

USBprodName is the name of the product as a string.

USBcurrent is the expected current consumption through the USB cable in mA (max 500mA)

USBvBUS the Port Pin where the application can identify a connected Host (5Volt). Optional.

USBsernum is the serial number of the device and **should not** be defined. Instead the XMega internal serial number is used. As an **option** the user can provide a fixed/constant serial number.

All items are mandatorial though only USBvid and USBsernum are really minded by the Host.

If the serial number of all devices of this manufacturer and type are constant then there is no action when a new device of this kind is connected. Only once at the very first time the PC asks for a driver. If the serial number changes so with each new serial number a question for a correct driver is raised.

3.13.1 Callback Function

The Host program is able to send some private commands through the Control Pipe (endpoint0). To handle this in the device the main must provide a Callback Function. The address of this function must be passed to the driver:

```
// User function to receive EP0 ControlRequests from the Host
Function myUSB_ControlRequest(bRequest : Byte; wValue : word) : boolean;
Begin
  // ...
  // this function returns with user defined commands in bRequest and optional parameters
  // in wValue send by the host
end;
...
  USB_SetControlCallback (@myUSB_ControlRequest);
```

If *myUSB_ControlRequest* is not present or not passed to the driver by USB_SetControlCallback then there is no callback and incoming commands are ignored. If the application can handle the request type defined by "bRequest" then the function must return a true, otherwise a false. The requests 0..15 should be reserved for the system.



AVRco Standard Driver

3.13.2 Exported Functions and Procedures

Procedure *USB_Init(pFirstRxBuf : pointer);*

This procedure initialises the USB hardware in the XMEGA. The pointer is necessary because a Host can send data after the init. This pointer must point to a 64byte memory area.

Procedure *USB_SetControlCallback(myUserProc : tUserSetupProc);*

This procedure passes an optional Call-Back function address to the driver which then the driver calls if a Vendor Requests (private command) is received in the Control Pipe (Endpoint0).

Procedure *USB_ControlJob;*

This procedure must be called continuously (polling). It checks for request, commands etc. coming from the Endpoint0 and handles them. If the time gaps are too long (>50msec) the Host can run into time-outs which then can result in a complete communication abort. It makes sense to place this job into a Task.

Procedure *USB_ControlSend(pData : Pointer; size : word); // send Controlrequest Data*

If a bRequest is handled in the ControlCallBack then it must be always closed with this procedure. Then the pointer can point to a data block and in "size" the byte count is returned. If there is no data to return then the pointer must be NIL and size must be zero.

Function *USB_RxDataAvail : boolean; // check if Data on Endpoint1 received*

This is the Rx poll function. It must be called continuously. If the time gaps are too long (>1sec) the Host can run into time-outs which then can result in a complete communication abort. With a true returned the byte count in the RxBuffer must be called with USB_RxCount. Then the buffer data can be accessed. If this is done then the RxBuffer must be released with USB_RxSetBuf. After this the driver accepts new data from the Host via Endpoint1.

Function *USB_RXcount : byte;*

If data is available the this function must be used to find out the blocksize/bytecount.

Procedure *USB_RxSetBuf(Buf : Pointer); // connect Endpoint1 do your buffer*

If any data was received through the Endpoint1 (USB_RxDataAvail) then the RxBuffer must be released so the driver can accept new data from the Host. The pointer must point to a 64byte sized buffer. The buffer must be global and on even address, **Align2**

Function *USB_TxSend(pData : Pointer; size : byte) : boolean; // Result send OK or Timeout*

If the Host requests data then the pointer must point to a TxBuffer and with size the byte count must be defined. If there is nothing to send then this pointer must be NIL and size must be zero. The buffer must be global and on even address, **Align2**

Function *USB_TXcomplete : boolean;*

After USB_TxSend is executed, this function should be polled until the Host has fetched the data. Any other operation until then doesn't make sense. The user can implement a timeout in this poll loop.

Procedure *USB_Detach;*

Disconnects the USB driver from the host (PC).

Typical program flow:

USB_Init ...

USB_RxSetBuf ...

Loop

```

    USB_ControlJob;
    if USB_RxDataAvail then
        Count:= USB_RxCount;
        // do something with RXData .....
        USB_RxSetBuf(@Rxbuf);                // we need more data ..
    endif;
    // fill TxBuf with data ...
    USB_TxSend(@TXbuf, 64);
    Repeat until USB_TxComplete;
```

A Demo and Testprogram can be found in the Demo directory "XMEGA_USBsmart" and "XMEGA_BootUSB".

3.13.3 Host/PC Implementation

USB interfaces are pure Master/Slave systems like I2C or SPI. The PC is always the Host/Master and the device is always the Slave. A Slave never can send data to the Host, but the Host must read data out of the Slave. This must be taken in account with each communication between Host and Slave.

An USB device always needs a suitable driver on the Host/PC side. These maybe already existing ones like HID for mouse or Keyboard. Here there is no need for additional drivers. Then there is the mostly unknown and rarely used CDC interface (virtual comport) which only needs a specific INF-File. Proprietary devices like printers or the E-LAB programmer always need a complete driver set consisting of a SYS-File, INF-File and sometimes a DLL.

The AVRco USBsmart implementation for XMegas also need these three driver parts. Because the constructions of them, mainly the SYS file, needs tremendous knowledge, the AVRco system provides a so called generic driver. It can be configured with parameters for nearly every device. This is the **libUSB** system. It provides a complete configurable driver set, including the automatic creation of the INF file. This text file contains the description of the driver and the capabilities of the device. The manual creation of such an INF-Files needs a huge knowhow and can be a nightmare for unexperienced users.

When creating a Windows application for USB controlled AVRco devices the DLL build by the libUSB system must be used. This DLL provides all necessary interface functions. The DLL functions used by the included Delphi application „USBtester“ are provided by the Unit „LibUSB.pas“. And there are much more. Used functions are:

3.13.3.1 Initialisation etc.

Procedure *usb_init*;

Function *usb_find_busses* : longword;

Function *usb_find_devices* : longword;

Function *usb_get_busses* : pusb_bus;

3.13.3.2 Device specific

Function *usb_open*(dev : pusb_device) : pusb_dev_handle;

Function *usb_close*(dev : pusb_dev_handle) : longword;

Function *usb_set_configuration*(dev : pusb_dev_handle; configuration : longword) : longword;

Function *usb_claim_interface*(dev : pusb_dev_handle; iinterface : longword) : longword;

Function *usb_release_interface*(dev : pusb_dev_handle; iinterface : longword) : longword;

3.13.3.3 Support

Function *usb_get_descriptor_by_endpoint*(udev : pusb_dev_handle; ep: longword; ttype : byte; index : byte; var buf; size : longword) : longword;

Function *usb_get_descriptor*(udev: pusb_dev_handle; ttype: byte; index: byte; var buf; size: longword): longword;

Function *usb_get_string_simple*(dev: pusb_dev_handle; index: longword; var buf; buflen: longword) : longword;



AVRco Standard Driver

3.13.3.4 Data Transfer

Function *usb_control_msg*(dev : *pusb_dev_handle*; requesttype, request, value, index : *longword*; var bytes; size, timeout : *longword*) : *longword*;

This function must be used to send commands through the Control-Pipe (endpoint0).

The following parameters are mandatory to send „private“ commands and data to the Slave:

requesttype = \$40

request = \$00

value is the actual command

index is an optional word parameter

bytes and *size* are only used for reading data out of the controlpipe. Not used here

timeout is in msec

The commands (value) 0..15 are reserved for the AVRco system.

With the ControlMessage/endpoint0 it is possible to send several commands or data (Vendor Requests).

Commands 16..255 (*msgUser+x*) can be used in the user application and are processed by a Callback function in the X Mega. The index parameter is optional. See the "RESET" example in the Delphi source.

Function *usb_bulk_write*(dev: *pusb_dev_handle*; ep : *longword*; var bytes; size, timeout: *longword*): *longword*;

This is the the actual tx-function. Data is send to endpipe1.

Parameter:

ep = endpoint1

bytes = Buffer which contains the packet to send

size = count of the bytes to send. Any count with endpoint1, but the Slave must be able to receive and store this packet in one piece. This means ist Rx-Buffer must be large enough. On the PC the packet size can be random. They are internally split into 64 byte sized blocks/packets and then send. The result of the function is the total transmitted byte count. If the Slave is unable to read all bytes because of memory limitations, the result is always the count of the bytes send. Will be fixed later.

Function *usb_bulk_read*(dev: *pusb_dev_handle*; ep: *longword*; var bytes; size, timeout: *longword*): *longword*;

Das ist die eigentlich Empfangs Funktion. Daten werden von der endpipe1 abgeholt.

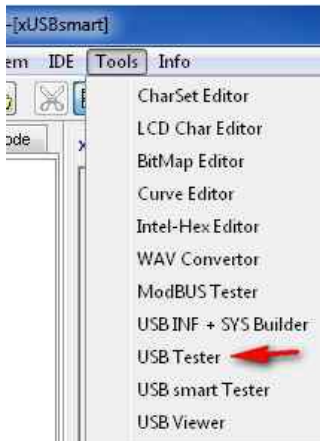
Parameter:

ep = endpoint1

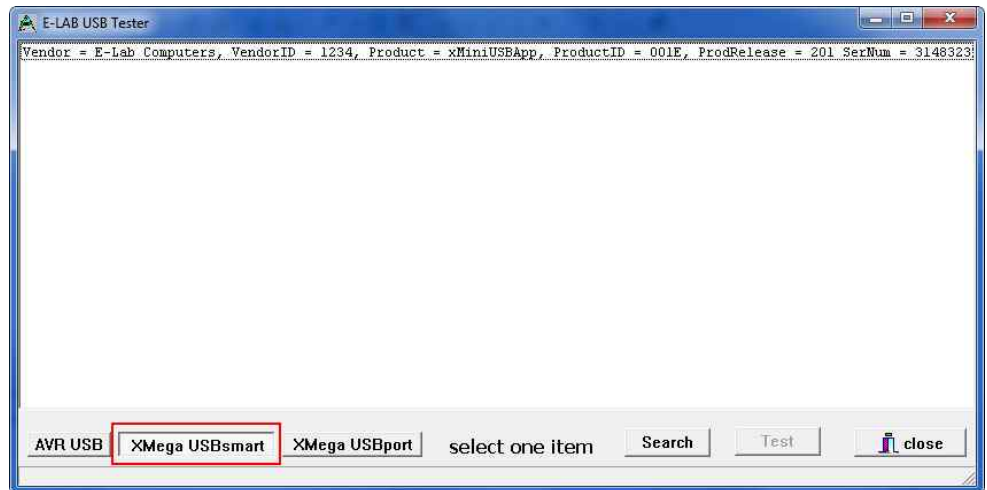
bytes = Buffer which must contain the received data

size = count of bytes to receive. Any count with endpoint1. But the Slave must be able to send this packet in one piece. Means ist tx-buffer must be large enough. On the PC the packet size can be random. They are fetched in 64byte blocks and then returned in one packet. The result of the function is the total received byte count.

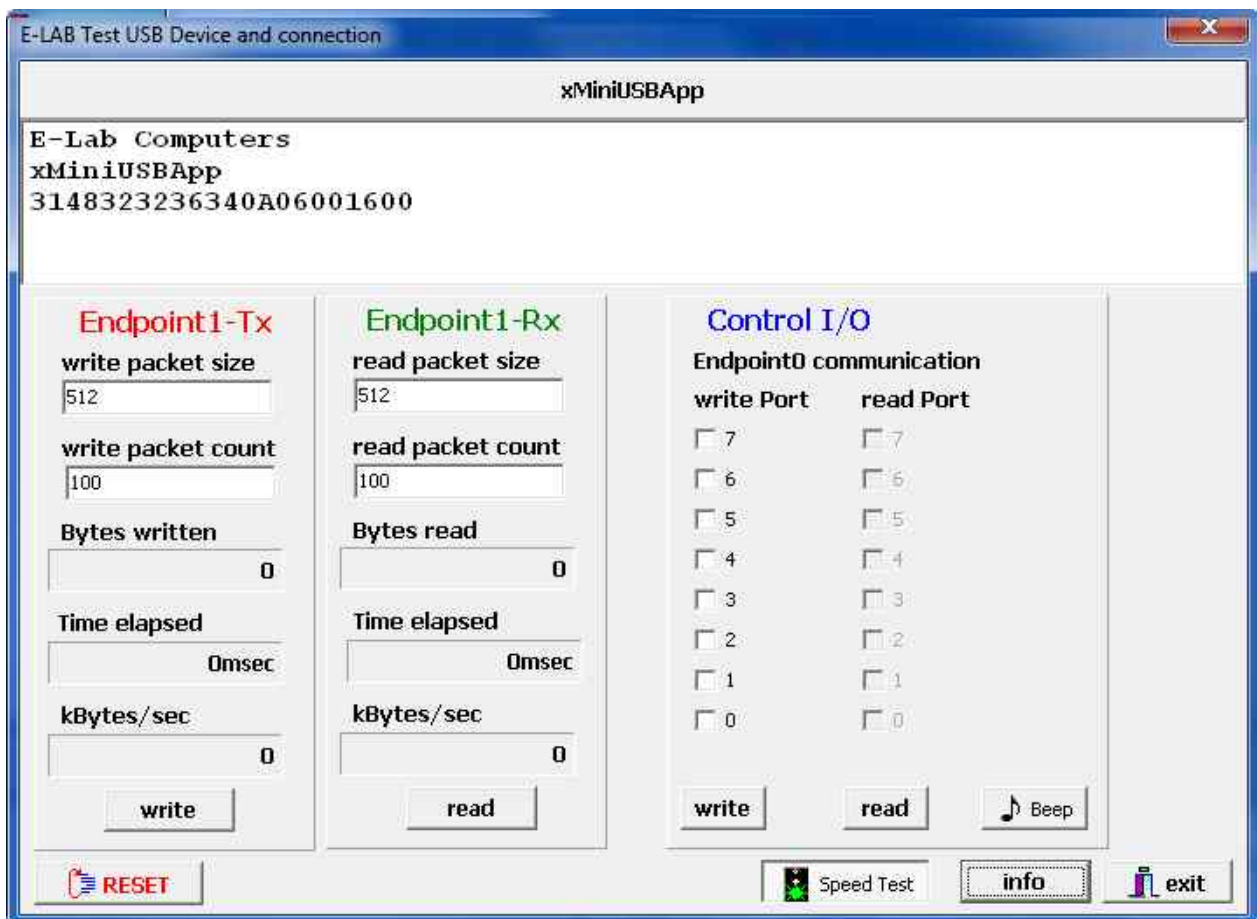
3.13.1 Testprogram in the IDE PED32



The AVRco installation provides a test program „USBtester“ which can be started by the left menu. The the form below is shown:



After selecting a device the working dialog is shown.



Some transfers can be started.

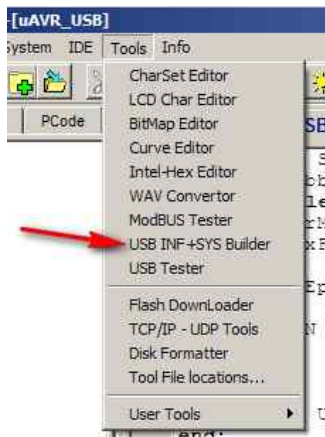
The source of this Delphi program can be found in the installation directory in

[..\\AVRco\\IDE\\USBtester\\](#)



AVRco Standard Driver

3.13.2 Support Tools



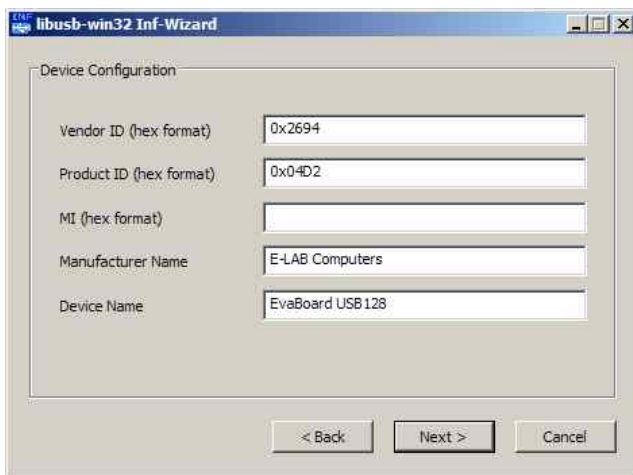
Inf-Wizard Start



Creating the necessary SYS, INF and DLL files for the Host/PC is an enormous challenge for the developer. For many not doable without the right knowhow and experiences. Therefore AVRco system contains a tool which exactly does this for you!. It will be started in the menu above.

Attention:

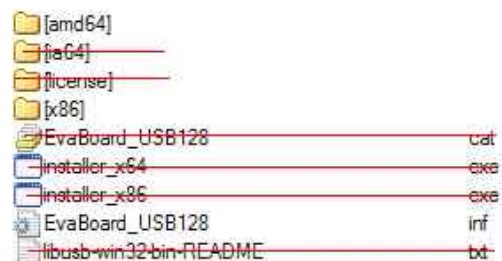
The AVR Device must be connected and working properly! Any driver isn't necessary at this time. Please select only the device you have build! Accidently selecting the wrong device and installing it will result in a failure of this (wrong) device.Maybe it will never work again!!



The next step is checking the data and parameters of this device drivers. With „Next“ the driver files are build and stored.

Optionally this driver can directly be installed on the current system. Then this device is ready to operate.

The Wizard generates several files:



But for the enduser only these are important:



These files and directories must be given to the enduser exactly as they are (directories etc).

If such a device is connected to a Host the first time the PC ask for a suitable driver. The this **INF-File** must be selected. The rest will be done automatically.

Never give the away the Inf-Wizard. Inexperienced user can destroy or cripple the whole PC system.

3.14 PWMports

3.14.1 PWMport1A 1B 1C PWMport3A 3B 3C PWMport4A 4B 4C PWMport5A 5B 5C

All AVR's contain at least one 16bit timer (Timer1) which supports up to 3 PWM channels. Bigger CPUs often have a second 16bit timer (Timer3) which also can support up to 3 PWM channels. Which timers are available in a specific CPU can be found in the corresponding datasheet. How much PWMs such a timer supports is not very easy to find out. It is best to take a look at the chip pinout of the datasheet. Search for the pin names "OC1A, OC1B, OC1C" for the timer1. With the timer3 these pins are named "OC3A, OC3B, OC3C". If such a pin is present the relevant PWM can be imported.

Single channel timers are named without a trailing character (e.g. OC2).

Because such a timer can control up to 3 PWMs its setup (prescaler, resolution) is valid for all of its PWMs. PWMport4 and 5 only supported in Rev and higher.

Imports

As usual with AVRco the driver must be imported.

The timers to use (Timer1, Timer3 etc) and their PWMports are declared by the imports of the PWMports.

Import SysTick, PWMport1A, ..., PWMport3C, ...;

Defines

The values of the prescaler and the pwm resolutions must be defined.

```
Define ProcClock    = 8000000;      {Hertz}
          SysTick     = 10;          {msec}
          StackSize   = $0030, iData;
          FrameSize   = $0030, iData;
          PWMpresc1   = $2;          {prescaler timer1}
          PWMres1     = $8;          {resolution timer1}
          PWMpresc3   = $4;          {prescaler timer3}
          PWMres3     = $10;         {resolution timer3}
```

PWMres1 (Timer1),

PWMres3 (Timer3)

PWMres4 (Timer4)

PWMres5 (Timer5)

Defines the common resolution for one of the two possible PWMport groups.

With the AVR 8, 9 or 10 Bits are possible.

PWMpresc1 (Timer1)

PWMpresc3 (Timer3)

PWMpresc4 (Timer4)

PWMpresc5 (Timer5)

Defines the common prescaler for one of the two possible PWMport groups. The prescaler controls the repeat rate (frequency) of the PWMs. The values must be within 1..5.

PWMmode1 (Timer1)

PWMmode3 (Timer3)

PWMmode4 (Timer4)

PWMmode5 (Timer5)

Optional defines. Select the standard or fast mode of the timer and the pulse polarity.

PWMport1A, PWMport1B, PWMport1C (Timer1)

PWMport3A, PWMport3B, PWMport3C (Timer3)

The import of PWMPORTxx also imports a variable *PWMPORTxx*

These variables are always of type Word independent of the selected resolution.

This variable is the only existent interface between the application and the driver.

PWMport1A:= 127; {50% Duty Cycle at 8bit res}

PWMport3B:= 511; {50% Duty Cycle at 10bit res}



AVRco Standard Driver

PWMport4A, PWMport4B, PWMport4C (Timer4)

PWMport5A, PWMport5B, PWMport5C (Timer5)

Timer4 and Timer5 if present. These PWMs are supported with Compiler revision 4.xx and higher.

```
PWMport4A:= 127;    {50% Duty Cycle at 8bit res}  
PWMport5B:= 511;    {50% Duty Cycle at 10bit res}
```

Note:

Possible existent 8bit PWMs of the 8bit timers are not supported.

The old AVRco PWMport names PWMport1 and PWMport2 are still valid but should not be used with new designs.

3.14.2 PWMport2A, PWMport2B

some newer AVR's have a timer2 that supports upto 2 PWM channels (mega644 / 2560).

If such a timer exists and supports PWM it can also be used and the statements for timer1/timer3 are valid in an analogous way.

```
Import SysTick, PWMport2A, PWMport2B;
```

Define

```
...  
    PWMpresc2 = $8;           {prescaler timer2}  
    PWMres2   = $8;           {resolution timer2}  
    PWMmode2  = fast, negative {optional define}  
...
```

```
PWMport2A:= 84;  
PWMport2B:= 10;
```

3.14.3 Software PWM

A software PWM is implemented. There are up to 8 channels possible. An 8bit timer (Timer0 or Timer2) is used which runs with interrupts. The resolution can be set between 10 and 254 points. The PWM frequency (cycle time) can be set from 5msec/200Hz to 50msec/20Hz. These PWMs are well suited for brightness control of lamps and LEDs, for speed control of DC motors and for temperature control.

Please note that with an 8MHz CPU and 10msec cycle time (100Hz) the update on all 8 channels needs 16usec in the timer interrupt. The repeat rate of the interrupt is 100usec if you select a resolution of 100. So the SoftPWM needs 15% of the CPU performance.

With a resolution of 254 the repeat rate of the timer interrupt is increased to 40usec and the SoftPWM needs 40% of the CPU performance.

More channels need more performance. A greater resolution needs additional CPU performance and shorter PWM cycles also need more CPU performance.

Imports

Import SysTick, SoftPWM;

Defines

Define

<i>ProcClock</i>	<i>= 8000000;</i>	<i>{Hertz}</i>
<i>SysTick</i>	<i>= 10;</i>	<i>{msec}</i>
<i>StackSize</i>	<i>= \$0032, iData;</i>	
<i>FrameSize</i>	<i>= \$0032, iData;</i>	
<i>SoftPWMport</i>	<i>= PortA;</i>	<i>{use PortA for PWM output}</i>
<i>SoftPWMchans</i>	<i>= 4, 0;</i>	<i>{4 channels, bit0/PortA is the first}</i>
<i>//SoftPWMchans</i>	<i>= 4, 2, negative;</i>	<i>{4 channels, bit2/PortA 1.bit, low pulsed}</i>
<i>SoftPWMtimer</i>	<i>= timer2, 10;</i>	<i>{use timer2, PWM cycle time 10msec = 100Hz}</i>
<i>SoftPWMres</i>	<i>= 16;</i>	<i>{PWM resolution is 16 points}</i>

SoftPWMport

specifies the output port

SoftPWMchans

specifies the number of PWM channels and the first bit of the port = PWM1 bit. As an option the output polarity *positive/negative* can be defined.

SoftPWMtimer

specifies the 8-bit timer (Timer0 or Timer2) and the cycle time in 1mSec steps. **XMega** one of the timers (Timer_C0.. Timer_F1) must be specified.

SoftPWMres

specifies the resolution of the PWMs, 10..254.

There are two special functions for this driver:

Procedure *SoftPWMstop;*

The Timer will be stopped. The PWM pins become idle/inactive.

Procedure *SoftPWMstart;*

The Timer will be restarted and all PWM values are zeroed.



Var

Furthermore it is possible to address the PWMs as an array of byte. The array is defined as:

var

```
SoftPWM2:= 8;  
SoftPWMar[1]:= 5;
```

Example program and schematic:

an example is in the directory `..\E-Lab\AVRco\Demos\SoftPWM`
 an **XMega** example is in the directory `..\E-Lab\AVRco\Demos\XMega_SoftPWM`



3.14.4 Software-PWM SoftPWM8 XMega

With the XMegas another software PWM is implemented. Upto 16 channels can be defined. These channels can be placed onto any port and port-pin. The advantage of this PWMs is an extreme low CPU load and a very short interrupt latency (interrupt disabling time) **~2.5usec total at 32MHz 8 channels**. The disadvantage is the low resolution of 3.5bits. This means 8 values can be set plus complete off. But with many applications this is sufficient.

Imports

Import SysTick, SoftPWM8;

Defines

Define

```
// The XMegas don't provide any Oscillator fuses.
// So the application must setup the desired values
// possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz
//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCtype = int32MHz,
        PLLmul=4,
        prescB=1,
        prescC=1;
SysTick = 10, adj; // msec, correct the RTC32K timer for exact mSec timing
StackSize = $80, iData;
FrameSize = $100, iData;

// Chan0 Chan1 Chan2 Chan3 Chan4 Chan5 ...
SoftPWM8chans = PortR.1, PortF.0, PortF.1, PortF.2, PortF.3, PortF.4,
               PortF.5, PortF.6, PortF.7; // 8 channels, mixed ports
SoftPWM8timer = Timer_D0, 10; // use Timer_D0, PWM cycle time 10msec = 100Hz
```

SoftPWM8chans

Here upto 16 ports with pins can be defined.

SoftPWM8timer

This defines the necessary XMega timer which is needed for this PWM. The additional parameter defines the cycle time of the PWMs. (1..100msec)

Exported Functions

procedure SoftPWM8set(chan, pwm : byte);

This procedure sets the desired on-time for the selected channel. The time must be in the range 0..8 . 0 = off, 8 = full on. The channel number starts with 0 and can go upto the defined channel number -1

procedure SoftPWM8start;

Starts all PWM channels. Timer interrupt becomes enabled.

procedure SoftPWM8stop;

Stops all PWM channels. Timer interrupt becomes disabled. The preset channel values are not changed.

procedure SoftPWM8clear;

Alle preset channel values are set to 0. Timer interrupt is not changed.

A sample can be found in the directory **..E-Lab\AVRco\Demos\XMega_SoftPWM8**



AVRco Standard Driver

3.15 XMega PWM

Each of the XMega 16bit timers provides upto 4 PWM ports which are all supported by this driver. Because a timer can control upto 4 PWMs so its setup (prescaler, resolution) is common for all its PWMs.

Imports

As usual the driver must be imported into the AVRco.

The timers to be used (TimerC0, TimerC1 etc) and PWMports are defined by the import of a PWMport. In the following context a **TT** stands for a timer (C0, C1, D0, D1 etc) and **C** for the channel (A..D).

Import SysTick, PWM_ **TT****C**;

Import SysTick, PWM_ C0A, PWM_ C0B, PWM_ C0C, PWM_ C0D, PWM_ C1A, PWM_ C1B;

Defines

Defines the prescaler and the resolution of the PWMs and also the output polarity of the channels.

PWMpresc_**TT**

Defines the prescaler of the used timer (1..7).

The prescaler defines the repeatrate or frequency of the PWMs.

PWMres_**TT**

Defines the resolution of a timer (8..16bits).

PWMPol_**TT****C**

defines the output polarity of a PWM channel (positive or negative).

Define

```

//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCType          = int32MHz,
                  PLLmul=4,
                  prescB=1,
                  prescC=1;

StackSize        = $0030, iData;
FrameSize        = $0060, iData;
PWMpresc_ C0     = $1;           // prescaler timerC0
PWMres_ C0       = 8;           // pwm resolution timerC0
PWMPol_ C0A      = negative;    // output polarity PWM_ C0A

```

3.15.1 Functions and Procedures

Procedure EnablePWM_ **TT****C**(ena : boolean);

Starts or stops the PWM channel. If disabled, the output is forced to a zero value.

EnablePWM_ C0A(true);

Procedure SetPWM_ **TT****C**(pw : byte|word);

Controls the dutycycle of a PWM channel. With a resolution > 8bit a Word is expected, otherwise a Byte.

SetPWM_ C0A(123);

Please note that a 100% can not be reached. There will always be a small peak. The timer C0, D0, E0 and F0 provide 4 PWM channels, timer C1, D1, E1 and F1 have 2 channels. So with bigger XMegas there are upto 24 PWMs. Then a SoftPWM is not necessary with the XMegas.

The PWM outputs (pins) are fixed to dedicated port pins and can not be changed.

O**T****C** output-compare-timer-chan pin

A sample can be found in the directory **..E-Lab\AVRco\Demos\XMega_PWM**

3.16 XMega CRC

If data errors with any kind of transmissions must be recognized then a checksum is mandatory. It must be appended to the transmitted data block. The most secure way is a CRC checksum which must be build with a known polynome over the entire packet. If the receiver got a different checksum then the operation must be repeated or discarded. An error correction doesn't take place.

There are two CRCs:

CRC16 (CRC-CCITT, Polynom $x^{16}+x^{12}+x^5+1$)

CRC32 (IEEE 802.3, Polynom $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$)

Both of them are supported by hardware of most of the XMegas. Where the CRC32 basically is the most secure.

Imports

As usual the driver must be imported into the AVRco.

Import *SysTick*, *XMega_CRC*, ..;

3.16.1 Functions and Procedures

Function *CRC16_Block*(*seed* : word; *source* : pointer; *count* : word) : word;

Function *CRC32_Block*(*seed* : longword; *source* : pointer; *count* : word) : longword;

These two functions are similar except of the result, 16 or 32bit.

Seed defines the start value for the CRC operation

Source must point to a memory in the internal SRAM.

Count defines the amount of bytes to be checked.

The checksum is returned as a word or longword.

Streaming

The functions above are working with data blocks. Sometimes it is necessary to work with data streams. For this purpose the functions below are implemented. With **StreamInit** the CRC is started. Then with **StreamAdd** bytes are continuously add to the CRC. If the streaming is finished the result can be fetched with **StreamGet**.

Procedure *CRC16_StreamInit*(*Seed* : word);

Starts resp. initialises the CRC stream.

Procedure *CRC16_StreamAdd*(*b* : byte);

Always adds one byte tot he CRC.

Function *CRC16_StreamGet* : word;

Requests the result (CRC) oft he operations above.

The 32bit functions are similar to the 16bit functions above

Function *CRC32_StreamInit*(*seed* : longword);

Procedure *CRC32_StreamAdd*(*b* : byte);

Function *CRC32_StreamGet* : longword;

A sample can be found in the directory ..\E-Lab\AVRco\Demos\XMega_CRC

3.17 SPI onBoard Network

Not for **XMegas**, use the serial SlipPorts instead.

The SPI-Interface of the Atmel AVR family is well suited for the interconnection of several CPUs. The communication between a CPU and special SPI-chips is also possible.

The intention of this implementation is basically the interconnection of CPU to CPU. Simple byte transfers between CPU and SPI-circuits are not implemented, these types can be realized with few assembler statements by the programmer himself. The strategy consists of a Single-Master and Multi-Slave System. For secure communication a frame oriented protocol is implemented.

3.17.1 Mini Network

As quoted above, the single byte transfer on a SPI is not a problem. Difficult and more complex is the link between two or more CPUs. This implementation takes this in account. The implementation is a single master-multi slave system, which consists basically of one master and at minimum one slave CPU.

The master controls all connected slaves. No slave can gain the control of the BUS. All data traffic, regardless of the direction master-slave or slave-master is initiated and controlled by the master. With this all the common serious problems with collisions, priorities etc. are dropped.

This implementation can be called an "On-Board Network". You can also call it a "Mickey Mouse" network.

Networks in general are working with telegrams, also called frames. By the usage of frames there is some overhead, but on the other side the security and handability is very improved. To avoid more typical SPI problems, the SPI inherent full duplex transfer is not used. Instead of this the system works with a quasi half-duplex mode. For example, the master requests a frame from a slave. The slave responds with an acknowledge and the master fetches the frame from the slave. On the other way, if the master wants to send a frame to a slave, the master first sends a request, the slave responds also with an acknowledge and then the master sends its frame to the slave.

The request/acknowledge technique can be seen as a polling by the master. The master in principle works without SPI-interrupts, while the slave always uses SPI-interrupts. If the requ-ack handshake is done, all interrupts of the master are disabled and the slave is in its SPI-interrupt-service routine, which is only exited if the frame is completely transferred or its SS-Pin gets inactive. This guarantees that the frame is transferred in the shortest possible time (Master) and no byte is lost (Slave). Byte losses are the main problems with SPI CPU-CPU links, because an interrupt in the slave, while the transfer is in progress at high data rates, leads with certainty to byte losses in the slave. On the other hand, if a slave can't be interrupted whilst receive/transmit operations, this can't happen. The disadvantage of this is that all interrupts are disabled as long as a transfer is going on. To avoid to disabling interrupts for a long time, the master must not be disturbed by own interrupts while transferring (e.g. Task-changes) so the frame is transferred with the maximum speed without any delays between the bytes.

The frame lengths can be arbitrary, in the limits of 0..SPIbufferLen -1.

Each slave needs for selection a control line from the master, which is connected to the **SS**-pin of the slave. Because of this there is no address information in the frame. The application program determines in the master how and/or which port has to build these select lines. The AVRco SPI driver can't support this function. It's the job of the master application to activate this control line and select a slave before a communication starts. This control line should be disabled immediately after the transfer is finished.

If a slave has an inactivated SS-input (log. 1) this device disables its SPI logic and tristates its MISO and MOSI lines. Hence it's possible to connect several slaves to the SPI BUS, provided that not more than one slave has an activated SS-line.

Basics

The SPIport must be imported by an Import instruction, as usual with the AVRco system

Import *SPIport;*

The operation mode of the SPI peripheral, master or slave must be declared with a Define. A change of this mode at runtime is impossible.

Define *SPIport = Master;*

or

Define *SPIport = Slave;*

Master and slave both own a receive- and a transmit-buffer. All slaves should have the same buffer size like the master. The length of this buffers (min1, max 254) and the memory area is determined by this Define:

Define *SPIbuffer = nn, iData;*

The internal mode of the SPI peripheral must be identical in the master and the slaves and is declared in this manner:

Define *SPIOrder = MSB; {LSB}*
SPICPOL = 0; {0/1}
SPICPHA = 0; {0/1}
SPIpresc = 0; {0/1/2/3}

If the master is instructed to receive or transmit a frame from/to the slave, it polls the slave for an acknowledge. If the slave is not ready or doesn't respond in any way, this can lead to total lock-up of the system. To avoid this there must be a retry timeout defined by Define SPIretry. This parameter determines how many times the master sends a poll command until it returns with a Timeout error (SPIrxFrame, SPItxFrame) :

Define *SPIretry = 10;*

3.17.2 Exported Variables

SpiTxBuff, SpiRxBuff

The length of these buffers are determined by the Define SPIbuffer. The buffers can be treated as an array[1..BuffLen] of byte. By that a read or write access with array indexes is possible.

SPItxBuff[0]:= \$56;
X:= SPIrxBuff[6];

With the help of the function SPIout(x) it's possible to write continuously into the Tx-Buffer. A write pointer is automatically updated. The Rx-Buffer can be read with SPIinp, a read pointer is automatically updated.

for *i:= 0 to 5 do*
SPIout (\$ff);
endfor;

for *i:= 1 to SPIRXLEN do*
bb:= SPIinp;
endfor;



AVRco Standard Driver

SpiRxLen, SpiTxLen

These two byte variables are only for read purpose! SPIrxLen is updated after the complete reception of a frame with the actual frame length. SPITxLen is written by the command "SPITxFrame(n)".

3.17.3 Exported Functions and Procedures

Function SpiRxStat : boolean;

This function returns a true if a frame was received and the ReadOut pointer (changed by SpiInp) has not reached the end of the received frame. (SPIrxLen > 0) and (SPIrxOutP < SPIrxLen)

Function SpiInp : byte;

This function reads a byte from the RxBuffer at the position SPIrxBuff[SPIrxOutP] and increments the read pointer (SPIrxOutP). It should only be used in conjunction with SPIrxStat.

Function SpiRxFrame : boolean;

This function has a different meaning with Master and Slave. In case of the **Slave** it's identical with "SPIrxStat".

With **Master** a true is returned if a frame was received and this frame is not completely read out with SpiInp. See SPIrxStat.

If the RxBuffer is empty or cleared with "SPIrxClear", the master starts its polling operation and tries to receive a frame from the actual slave. For that the master continuously sends a \$FF (max. SPIretry) to the slave. If the slave doesn't respond the slave's hardware always echoes the \$FF. If the slave has nothing to send it responds with a \$00.

If there is a frame to send by the slave, the slave sends a byte > \$00 and < \$FF.

In the last case the slave has a frame ready to send and this byte is the length byte of the frame which follows. The master now receives this count of bytes from slave, stores them into the RxBuffer, resets the read pointer and stores the frame length into SPIrxLen. The function then returns with a true. If the master reaches the Retry Timeout with its polling, the function returns a false.

While receiving the frame from slave the master must send always a byte, so that the slave also can send a byte. These bytes from master are ignored by the slave, because they are only dummies. Throughout the whole operation the slave stays in its interrupt service routine. It exits only if all bytes are transferred or its SS-pin gets inactive.

Function SpiRxClear : boolean;

The read and write pointers and the Framelength byte of the RxBuffer are cleared. The contents of the buffer itself is not changed. If the state of the frame was "empty" a false is returned, otherwise a true.

Function SpiOnLine : boolean;

The **Master** transmits a \$00-request to the activated Slave (SS-pin active!). The Slave responds with a \$00. The Master can find out whether the selected Slave is online, that means it is ready to receive or send data. The result is true, if the Slave is active, otherwise it's false. The result has no meaning about the Slave's internal state e.g. Frame ready to send.

Neither in the Slave nor in the Master a Buffer or state will be changed.

Function SpiTxStat : boolean;

This function checks whether a frame to send is sent or not. If the frame is sent, the function returns with a true, otherwise with a false.

Function SpiOut (const b : byte) : boolean;

This function stores a byte into the TxBuffer at the location SPIrxBuff[SPItxInP] and increments the write pointer (SPItxInP). The store is only done if the TxBuffer is ready, that means a previous send operation with "SPITxFrame" has returned successfully. Furthermore in the TxBuffer there must be place for an additional byte (SPItxInP < BufferLen). After a successful byte store a true is returned, otherwise a false.

Function *SpiTxFrame (const len : byte) : boolean;*

This function has a different meaning with Master and Slave.

With the **Slave** it checks whether the last frame was transmitted successfully. If not, the function returns a false. If the last frame was sent, the "len" parameter is stored into "SPITxLen". The result of the function is then true.

If the master now requests a frame, the slave checks its SPI-interrupt-service parameter SPITxLen. The slave sends this byte as an acknowledge to the master. If this byte is \$00, the slave exits the interrupt, because there is nothing to send. Otherwise the slave waits for the dummy bytes from the master and responds for each master byte with a new byte from its TxBuffer. If the frame is sent completely the slave clears its SPITxLen to zero and exits the interrupt service. If the SS-pin goes inactive while transmitting the slave exits its interrupt service and resets its Tx state to an unsent but ready to send frame.

In case of **Master** mode the function checks whether the last frame was sent successfully. If not, the function returns a false. A possible reason was a TimeOut during the previous SPITxFrame. With SPITxClear the frame can be reset without destroying the frame itself and SPITxFrame can be invoked again. If the frame was sent successfully, the "len" parameter is stored in "SPITxLen".

In order to send a frame the master polls the active slave with \$00, until the TimeOut is reached (SPIretry) or the slave responds with a byte > \$00 and < \$FF. This byte is the max. possible frame length that the slave can accept. No further analysis of this byte takes place. The assumption is that the master and all slaves have the same buffer lengths.

In case of a timeout (SPIretry) the function returns with a false.

If the slave has responded with an acknowledge, the master first transmits the frame length (SPITxLen) and then the remaining bytes from the TxBuffer. The echo of the slave will be ignored, because the slave is in its interrupt service routine and leaves it only when the correct count of bytes is transferred or its SS-pin goes inactive.

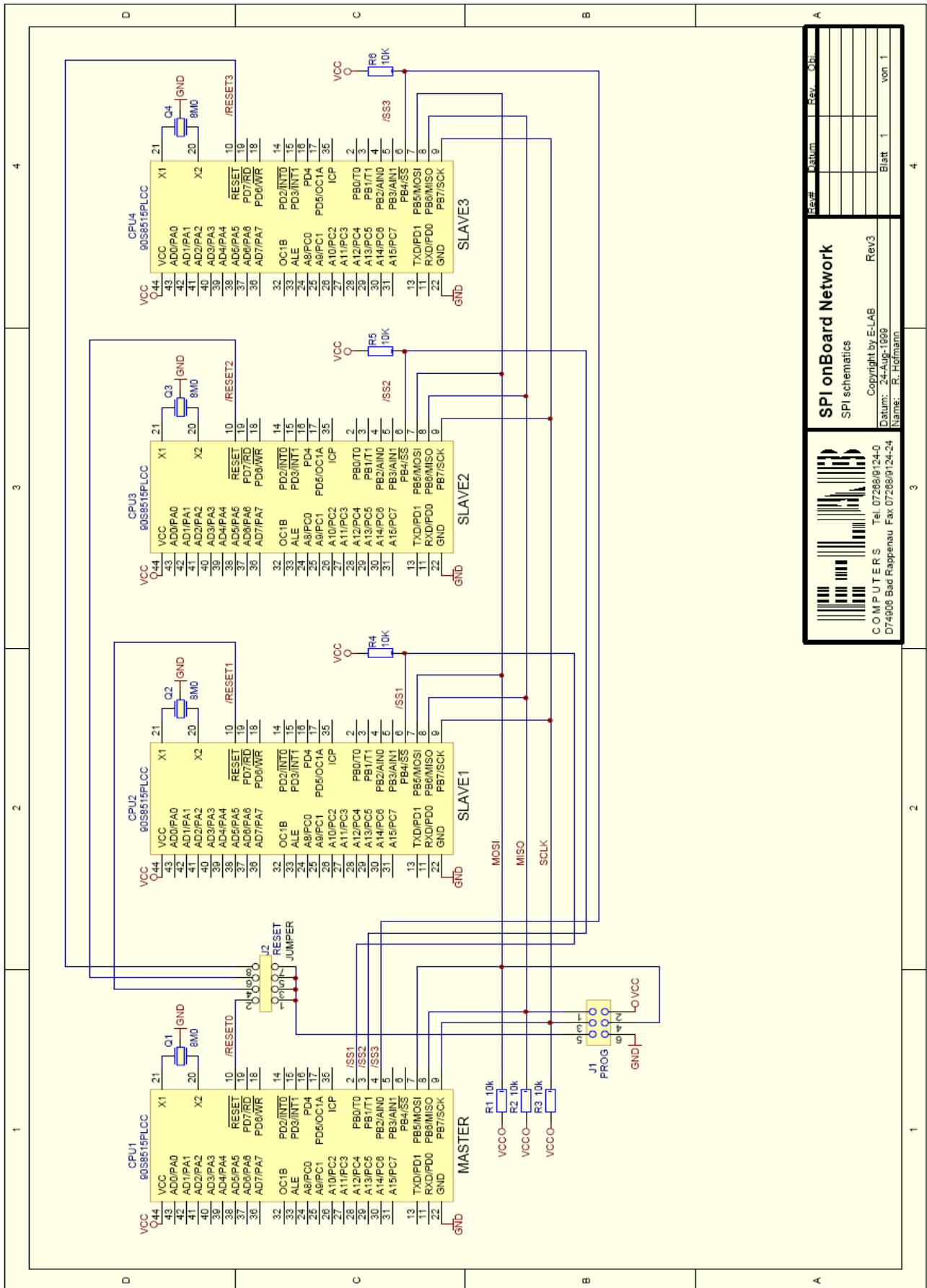
Procedure *SpiTxClear;*

The read and write pointers and the Framelength byte of the TxBuffer are cleared. The contents of the buffer itself are not changed. If the state of the frame was "empty" a false is returned, otherwise a true is returned.

Remarks

The Slave continuously looks at the SS-pin while a transfer is processed. If the SS-pin becomes inactive the Slave aborts the action. If a frame reception was occurring, the received frame is invalid. If the Slave is sending a frame the action is aborted and the frame length is reset to the original length. This means that the current frame is unsent and a new transmit is possible.

The used pins MISO, MOSI, SCLK of the Master and each SS-Pin of the slaves should be connected to a weak pullup resistor, typ. 10k. If the electrical connection fails, at least the "SPIONLine" function then will return a false.



schematic SPI

3.18 SPI Low Level SPIdriver, SPI_C...SPI_F Hardware Version

In addition to the Mini-Network **SPIport** the AVRco system also supports direct data transfers through the SPI interface of the AVR in Master Mode. This driver provides the low-level functions for reading and writing of datablocks with the SPI in Master Mode. Also runtime changes of the SPI modes are supported.

The setup of the SPI is achieved by the import of the driver and the associated defines. There are three different data transfer functions. The mandatory chip select for the slave is handled within the driver. As usual the PortBit **SS** of the SPI port is used. It is possible for the application to assert additional chip selects before calling the input and output functions and then deactivate them after a function returns.

The source and destination of the transferred data blocks must always reside in RAM. EEprom or Flash can not be used.

Imports

As usual with AVRco the driver must be imported.

Import SysTick, SPIdriver, ...;

XMega

Import SysTick, SPI_C, ...; // SPI_D, SPI_E, SPI_F

Defines

The Bitrate, ClockPhase, ClockPolarity and MSB/LSB must first be set with the Define, depending on the requirements of the SPI slave.

```
Define ProcClock = 8000000;           {Hertz}
          SysTick   = 10;               {msec}
          StackSize = $0030, iData;
          FrameSize = $0030, iData;
          SPIorder   = MSB;
          SPIcpol     = 1;
          SPIcpa      = 1;
          SPIpresc    = 1[, double];    // presc = 0..3 -> 4/16/64/128. Double = optional
          SPI_SS      = false;          // don't use SS pin as chipselect, not for Xmega
```

Alternative to the *SPIcpol* and *SPIcpa* the desired mode can also be defined in a more general way:

```
Define SPImode = 0; // 0, 1, 2, 3
```

XMega

```
Define OSCtype = int32MHz, PLLmul=4, prescB=1, prescC=1; // all 32MHz
          SysTick = 10;                               {msec}
          StackSize = $0030, iData;
          FrameSize = $0030, iData;
          SPIorderC = MSB;
          SPImodeC   = 0; // Clock Phase and Polarity
          SPIprescC  = 1; // presc = 0..3 -> 4/16/64/128
          SPI_SSC    = PortB, 3; // use this pin as SS chipselect
          // SPI_SSC = none;     // application generates the SS chipselect
```

With the **XMegas** there are upto 4 separate SPI ports. Each of them is included into an I/O Port. PortC, PortD, PortE, PortF. Because of this these SPIs are named SPI_C, SPI_D, SPI_E and SPI_F. Please also note the switch **{\$REUTILIZE xxx}** in the Compiler Manual.

Attention:

Basically the standard SS-pin must be programmed as an output with all AVR and XMegas. The system does this. If this pin is not used as the SS, so it can be used as a normal output pin.



AVRco Standard Driver

3.18.1 Functions

These SPI Defines can be changed at runtime:

```
Procedure SetSPOrder (msb : boolean);
Procedure SetSPIClkPol (pol : byte);           // pol = 0/1
Procedure SetSPIClkPha (phase : byte);        // phase = 0/1
Procedure SetSPIPresc (presc : byte);         // presc = 0..3 -> 4/16/64/128
Procedure SetSPImode(mode : byte);           // mode = 0..3
```

XMega

```
Procedure SetSPOrderC(msb : boolean);
Procedure SetSPOrderD(msb : boolean);
Procedure SetSPOrderE(msb : boolean);
Procedure SetSPOrderF(msb : boolean);
```

```
Procedure SetSPIClkPolC(pol : byte);          // pol = 0/1
Procedure SetSPIClkPolD(pol : byte);          // pol = 0/1
Procedure SetSPIClkPolE(pol : byte);          // pol = 0/1
Procedure SetSPIClkPolF(pol : byte);          // pol = 0/1
```

```
Procedure SetSPIClkPhaC(phase : byte);        // phase = 0/1
Procedure SetSPIClkPhaD(phase : byte);        // phase = 0/1
Procedure SetSPIClkPhaE(phase : byte);        // phase = 0/1
Procedure SetSPIClkPhaF(phase : byte);        // phase = 0/1
```

```
Procedure SetSPIPrescC(presc : byte);         // presc = 0..3 -> 4/16/64/128
Procedure SetSPIPrescD(presc : byte);         // presc = 0..3 -> 4/16/64/128
Procedure SetSPIPrescE(presc : byte);         // presc = 0..3 -> 4/16/64/128
Procedure SetSPIPrescF(presc : byte);         // presc = 0..3 -> 4/16/64/128
```

```
Procedure SetSPImodeC(mode : byte);           // mode = 0..3
Procedure SetSPImodeD(mode : byte);           // mode = 0..3
Procedure SetSPImodeE(mode : byte);           // mode = 0..3
Procedure SetSPImodeF(mode : byte);           // mode = 0..3
```

```
Procedure SPIout (source : pointer; count : word);
```

XMega

```
Procedure SPIoutC(source : pointer; count : word);
Procedure SPIoutD(source : pointer; count : word);
Procedure SPIoutE(source : pointer; count : word);
Procedure SPIoutF(source : pointer; count : word);
```

Writes a datablock from the RAM address **source** with the length **count** into the SPI-Slave.
SPIout (@ArrXY, sizeof (ArrXY));

```
Procedure SPIinp (dest : pointer; count : word);
```

XMega

```
Procedure SPIinpC(dest : pointer; count : word);
Procedure SPIinpD(dest : pointer; count : word);
Procedure SPIinpE(dest : pointer; count : word);
Procedure SPIinpF(dest : pointer; count : word);
```

Reads a datablock out of the SPI-Slave into the memory location **dest** with the length **count**
SPIinp(@ArrXY, sizeof (ArrXY));

XMega

```
SPIinpC(@ArrXY, sizeof (ArrXY));
```

Procedure *SPlinOut*(source, dest : pointer; count : word);

XMega

Procedure *SPlinOutC*(source, dest : pointer; count : word);

Procedure *SPlinOutD*(source, dest : pointer; count : word);

Procedure *SPlinOutE*(source, dest : pointer; count : word);

Procedure *SPlinOutF*(source, dest : pointer; count : word);

Writes a datablock from the RAM address **source** with the length **count** into the SPI-Slave and reads a datablock out of the SPI-Slave into the memory location **dest** with the length **count**

SPlinOut(@ArrXY, @RecordAB, 24);

XMega

SPlinOutC(@ArrXY, @RecordAB, 24);

Procedure *SPloutByte*(b : byte);

XMega

Procedure *SPloutByteC*(b : byte);

Procedure *SPloutByteD*(b : byte);

Procedure *SPloutByteE*(b : byte);

Procedure *SPloutByteF*(b : byte);

Writes a byte into the SPI-Slave.

SPloutByte (\$32);

XMega

SPloutByteC(\$40);

Procedure *SPloutWord*(w : word);

XMega

Procedure *SPloutWordC*(w : word);

Procedure *SPloutWordD*(w : word);

Procedure *SPloutWordE*(w : word);

Procedure *SPloutWordF*(w : word);

Writes a word into the SPI-Slave.

SPloutWord (\$4080);

XMega

SPloutWordC(\$4080);

Procedure *SPloutLong*(L : LongWord);

XMega

Procedure *SPloutLongC*(L : LongWord);

Procedure *SPloutLongD*(L : LongWord);

Procedure *SPloutLongE*(L : LongWord);

Procedure *SPloutLongF*(L : LongWord);

Writes a Longword into the SPI-Slave.

SPloutLong(\$12345678);

XMega

SPloutLongC(\$12345678);

Function *SPlinpByte* : byte;

XMega

Function *SPlinpByteC* : byte;

Function *SPlinpByteD* : byte;

Function *SPlinpByteE* : byte;

Function *SPlinpByteF* : byte;

Reads a byte out of the SPI-Slave.

bb:= *SPlinpByte*;

XMega

bb:= *SPlinpByteC*;



AVRco Standard Driver

Function *SPlinpWord* : word;

XMega

Function *SPlinpWordC* : word;

Function *SPlinpWordD* : word;

Function *SPlinpWordE* : word;

Function *SPlinpWordF* : word;

Reads a word out of the SPI-Slave.

ww:= *SPlinpWord*;

XMega

ww:= *SPlinpWordC*;

Function *SPlinpLong* : longword;

XMega

Function *SPlinpLongC* : longword;

Function *SPlinpLongD* : longword;

Function *SPlinpLongE* : longword;

Function *SPlinpLongF* : longword;

Reads a longword out of the SPI-Slave.

Lw:= *SPlinpLong*;

XMega

Lw:= *SPlinpLongC*;

Function *SPlinOutByte*(*b* : byte) : byte;

XMega

Function *SPlinOutByteC*(*b* : byte) : byte;

Function *SPlinOutByteD*(*b* : byte) : byte;

Function *SPlinOutByteE*(*b* : byte) : byte;

Function *SPlinOutByteF*(*b* : byte) : byte;

Writes a byte into the SPI-Slave and reads a byte out of the SPI-Slave.

bb:= *SPlinOutByte* (\$24);

XMega

bb:= *SPlinOutByteC*(\$33);

Procedure *SetSPIdoubleSpeed*(*ds* : boolean);

XMega

Procedure *SetSPIdoubleSpeedC*(*ds* : boolean);

Procedure *SetSPIdoubleSpeedD*(*ds* : boolean);

Procedure *SetSPIdoubleSpeedE*(*ds* : boolean);

Procedure *SetSPIdoubleSpeedF*(*ds* : boolean);

Switches the SPI bit rate between double and standard.

SetSPIdoubleSpeed (true); // switch to double speed

XMega

SetSPIdoubleSpeedC(true); // switch to double speed

XMega only

Function *SPlinpLong64C* : longword;

Function *SPlinpLong64D* : longword;

Function *SPlinpLong64E* : longword;

Function *SPlinpLong64F* : longword;

Reads a 64bit word out of the SPI-Slave.

W64:= *SPlinpLong64C*;

Procedure *SPloutLong64C*(*L* : Word64);

Procedure *SPloutLong64D*(*L* : Word64);

Procedure *SPloutLong64E*(*L* : Word64);

Procedure *SPloutLong64F*(*L* : Word64);

Writes a 64bit word into the SPI-Slave.

SPloutLong64(\$123456780ABCD);

The SPI-SS pin is default used as the chipselect for the slave. With the SPI_SS Define this can be disabled. Then the application must handle the chipselects itself.

3.19 MSPI Low Level SPI driver MSPI_0..MSPI_3 AVR

Not for XMegas!

Some AVR types provide the SPI Master operation of their UARTs. Which types support this can be found in the data sheets in „USART in SPI Mode“. They call it the **MSPIM** mode.

In addition to the standard hardware **SPI** the AVRco system also supports direct data transfers through the USART-SPI interface of the AVR_s **in Master Mode**. This driver provides the low-level functions for reading and writing of datablocks with the USART-SPI in Master Mode. Also runtime changes of the SPI modes are supported. The drivers are called MSPI_0, MSPI_1, MSPI_2 etc. dependend of the USART used.

Because there can be upto 4 USARTS in an AVR and all, if implemented, can be switched into the MSPIM mode the System supports upto 4 MSPI drivers which are identical but distinguished by an appended number 0..3. For example MSP0, MSP1 or MSPIout0 or MSPIout1 etc. In the following context the possible indexes 0..3 are replaced by a **n**.

The setup of the SPI is achieved by the import of the driver and the associated defines. There are three different data transfer functions. The mandatory chip select for the slave must be handled by the application which must assert a chip select before calling the input and output functions and then deactivate them after a function returns.

The TxD**n** pin becomes MOS**n**, the RXD**n** pin becomes MISO**n** and the XCK**n** pin becomes SCK**n**.

The source and destination of the transferred data blocks must always reside in RAM. EEprom or Flash can not be used.

Imports

As usual with AVRco the driver must be imported.

Import SysTick, MSPI_0, MSPI_1, ...;

Defines

The Bitrate, ClockPhase, ClockPolarity and MSB/LSB must first be set with the Define, depending on the requirements of the SPI slave. They can be changed at runtime.

```
Define  ProcClock   = 8000000;           {Hertz}
          SysTick     = 10;                {msec}
          StackSize   = $0030, iData;
          FrameSize   = $0030, iData;
          MSPIorder0  = MSB;
          MSPIcpol0   = 1;
          MSPIcpol0   = 1;
          MSPIpha0    = 1;
          MSPIpresc0  = 1;                 // presc = 0..255

          MSPIorder1  = MSB;
          MSPIcpol1   = 1;
          MSPIcpol1   = 1;
          MSPIpha1    = 1;
          MSPIpresc1  = 1;                 // presc = 0..255
          ....
```

Alternative to the *MSPIcpolx* and *MSPIcpahx* the desired mode can also be defined in a more general way:

```
Define  MSPImode0   = 0;                 // 0, 1, 2, 3
```



AVRco Standard Driver

3.19.1 Functions

These MSPI Defines can be changed at runtime:

```
Procedure MSetSPlordern (msb : boolean);  
Procedure MSetSPclkPoln (pol : byte); // pol = 0/1  
Procedure MSetSPclkPhan (phase : byte); // phase = 0/1  
Procedure MSetSPprescn (presc : byte); // presc = 0..255  
Procedure SetMSPI0mode(mode : byte); // mode = 0..3  
Procedure SetMSPI1mode(mode : byte); // mode = 0..3  
Procedure SetMSPI2mode(mode : byte); // mode = 0..3  
Procedure SetMSPI3mode(mode : byte); // mode = 0..3
```

Procedure MSPIoutn (*source* : pointer; *count* : word);

Writes a datablock from the RAM address **source** with the length **count** into the MSPI-Slave.

MSPIout0 (@ArrXY, sizeof (ArrXY));

Procedure MSPIinp (*dest* : pointer; *count* : word);

Reads a datablock out of the MSPI-Slave into the memory location **dest** with the length **count**

MSPIinp1 (@ArrXY, sizeof (ArrXY));

Procedure MSPIInOutn (*source*, *dest* : pointer; *count* : word);

Writes a datablock from the RAM address **source** with the length **count** into the MSPI-Slave and reads a datablock out of the MSPI-Slave into the memory location **dest** with the length **count**

MSPIInOut2 (@ArrXY, @RecordAB, 24);

Procedure MSPIoutByten (*b* : byte);

Writes a byte into the MSPI-Slave.

MSPIoutByte3 (\$32);

Procedure MSPIoutWordn (*w* : word);

Writes a word into the MSPI-Slave.

MSPIoutWord0 (\$4080);

Procedure MSPIoutLongn (*L* : LongWord);

Writes a Longword into the MSPI-Slave.

MSPIoutLong1 (\$12345678);

Function MSPIinpByten : byte;

Reads a byte out of the MSPI-Slave.

bb:= MSPIinpByte1;

Function MSPIinpWordn : word;

Reads a word out of the MSPI-Slave.

ww:= MSPIinpWord2;

Function MSPIinpLongn : LongWord;

Reads a longword out of the MSPI-Slave.

Lw:= MSPIinpLong3;

Function MSPIInOutByten (*b* : byte) : byte;

Writes a byte into the MSPI-Slave and reads a byte out of the MSPI-Slave.

bb:= MSPIInOutByte0 (\$24);

3.20 MSPI Low Level SPI driver MSPI_C0..MSPI_F1 X Mega

For XMegas only!

All X Mega types provide the SPI Master operation of their UARTs. This can be found in the data sheets in „USART in SPI Mode“. They call it the **MSPIM** mode.

In addition to the standard hardware **SPI** the AVRco system also supports direct data transfers through the USART-SPI interface of the XMegas **in Master Mode**. This driver provides the low-level functions for reading and writing of datablocks with the USART-SPI in Master Mode. Also runtime changes of the SPI modes are supported. The drivers are called MSPI_C0, MSPI_C1, MSPI_D0 etc. dependend of the USART used. The suffix C0 etc. is renamed to **xx** for simplicity. MSPI_**

All XMegas support the MSPIM mode with their USARTs. So there there can be upto 8 USARTS in a X Mega and all, if implemented, can be switched into the MSPIM mode. The System supports upto 8 MSPI drivers which are identical but distinguished by an appended suffix C0..F1. For example MSP_C0, MSP_C1 or MSPIout_C0 or MSPIout_C1 etc. In the following context the possible suffixes are replaced by a **xx**.

The setup of the SPI is achieved by the import of the driver and the associated defines. The mandatory chip select for the slave is handled by the driver itself. So the application which must define a chip select port and pin.

The TxD** pin becomes MOSI**, the RXD** pin becomes MISO** and the XCK** pin becomes SCK**.

The source and destination of the transferred data blocks must always reside in RAM. EEprom or Flash can not be used. For block transfers DMA channels can be used.

Imports

As usual with AVRco the driver must be imported.

Import SysTick, MSPI_C0, MSPI_C1, ..;

Defines

The Bitrate, SPI mode (ClockPhase, ClockPolarity) and MSB/LSB must first be set with the Define, depending on the requirements of the SPI slave. They can be changed at runtime. The desired pin and port for the /CS (SS) must be defined. If “**none**” is given then the application must handle the **SS** by itself.

Define

// The XMegas don't provide any Oscillator fuses.

// So the application must setup the desired values

//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz

```

    OSCtype      = int32MHz,
                  PLLmul=4,
                  prescB=1,
                  prescC=1;

    SysTick      = 10;           {msec}
    StackSize    = $0030, iData;
    FrameSize    = $0030, iData;
    MSPIorder_C0 = MSB;
    MSPIMode_C0  = 1;           // 0, 1, 2, 3
    MSPIpresc_C0 = 1;           // presc = 0..255
    MSPI_SS_C0   = PortE, 4;

```

```

    MSPIorder_C1 = MSB;
    MSPIMode_C1  = 0;           // 0, 1, 2, 3
    MSPIpresc_C1 = 3;           // presc = 0..255
    MSPI_SS_C1   = none;

```

Optional

MSPIDMA_OUT_C1 = DMAch0; // DMAch0 for BlockOut functions

....



AVRco Standard Driver

3.20.1 Functions

These MSPI Defines can be changed at runtime:

```
Procedure SetMSPIorder_XX(msb : boolean);  
Procedure SetMSPIclkPol_XX(pol : byte);           // pol = 0/1  
Procedure SetMSPIclkPha_XX(phase : byte);         // phase = 0/1  
Procedure SetMSPIpresc_XX(presc : byte);         // presc = 0..255  
Procedure SetMSPImode_XX(mode : byte);           // mode = 0..3
```

DMA:

With OutBlock functions one of the 4 possible DMA channels can be used.

With InpBlock functions one of the 2 possible DMA pairs (0+1 or 2+3) can be used.

The used DMA must be defined in the Define section.

```
Define MSPIDMA_OUT_XX = DMAch0;  DMAch0 for Block Out functions  
Define MSPIDMA_INP_XX = DMAch2;  DMAch2+3 for Block Inp functions
```

Procedure MSPIout_XX (*source* : pointer; *count* : word); // **DMA** can be used

Writes a datablock from the RAM address **source** with the length **count** into the MSPI-Slave.

```
MSPIout_C0(@ArrXY, sizeof (ArrXY));
```

Procedure MSPIinp_XX (*dest* : pointer; *count* : word); // **DMA** can be used

Reads a datablock out of the MSPI-Slave into the memory location **dest** with the length **count**

```
MSPIinp_C1(@ArrXY, sizeof (ArrXY));
```

Procedure MSPIinOut_XX(*source*, *dest* : pointer; *count* : word); // no DMA possible

Writes a datablock from the RAM address **source** with the length **count** into the MSPI-Slave and reads a datablock out of the MSPI-Slave into the memory location **dest** with the length **count**

```
MSPIinOut_D0(@ArrXY, @RecordAB, 24);
```

Procedure MSPIoutByte_XX(*b* : byte);

Writes a byte into the MSPI-Slave.

```
MSPIoutByte_D1($32);
```

Procedure MSPIoutWord_XX(*w* : word);

Writes a word into the MSPI-Slave.

```
MSPIoutWord_E0($4080);
```

Procedure MSPIoutLong_XX(*L* : LongWord);

Writes a Longword into the MSPI-Slave.

```
MSPIoutLong_E1($12345678);
```

Function MSPIinpByte_XX : byte;

Reads a byte out of the MSPI-Slave.

```
bb:= MSPIinpByte_E1;
```

Function MSPIinpWord_XX : word;

Reads a word out of the MSPI-Slave.

```
ww:= MSPIinpWord_F0;
```

Function MSPIinpLong_XX : LongWord;

Reads a longword out of the MSPI-Slave.

```
Lw:= MSPIinpLong_F1;
```

Function MSPIinOutByte_XX(*b* : byte) : byte;

Writes a byte into the MSPI-Slave and reads a byte out of the MSPI-Slave.

```
bb:= MSPIinOutByte_E1($24);
```

AVRco Standard Driver



DMA:

A DMA transfer can only be used with block transfer functions.

If a DMAout channel is defined

Define *MSPIDMA_OUT_xx = DMAchX*

So the DMA is only used for the function

Procedure *MSPIout_xx(..)*

If a DMAin channel is defined

Define *MSPIDMA_INP_xx = DMAchX*

So the DMA is only used for the function

Procedure *MSPIin_xx(...);*

Both functions return only if the complete transfer is done. This waiting can be disabled by calling a separate (optional) busy function:

Function *MSPI_DMAready_xx : boolean;*

This function returns with a false as long as the transfer is running. If the transfer has finished then a true is returned.

Attention:

Because the DMA uses the same address and databus as the CPU the DMA transfer is slowed down. The transfer time of the DMA is 3 times longer as without DMA!

Example programs:

Two examples are in the directories **..\E-Lab\AVRco\Demos\XMega_MSPI** and **..\E-Lab\AVRco\Demos\XMega_MSPI_DMA**



AVRco Standard Driver

3.21 SPI Low Level SPIdriver1/2 Software Version

In addition to the direct SPIdriver that uses the SPI port of the AVR the system also supports two pure software SPI interfaces in Master Mode. This driver provides the low-level functions for reading and writing of datablocks via the software interfaces in Master Mode.

The setup of the SPI is achieved by the import of the driver and the associated defines. There are three different functions. The mandatory chip select for the slave is as usual handled within the driver.

The source and destination of the transferred data blocks must always reside in RAM. EEprom or Flash can not be used.

Imports

As usual with AVRco the driver must be imported.

Import SysTick, SPIdriver1, ...; //or SPIdriver2 or both

Defines

The ClockPhase, ClockPolarity and MSB/LSB first must be set with the Define, dependant of the requirements of the SPI slave.

Define

```
ProcClock = 16000000;      {Hertz}
SysTick    = 10;           {msec}
StackSize  = $0030, iData;
FrameSize  = $0040, iData;

SPIdriver1 = PortA, 0, 1, 2, 3; // SCK, MOSI, MISO, SS
SPIdriver1 = MSB;
SPIdriver1 = 1;
SPIdriver1 = 1;
```

Alternative to the *SPIdriver1* and *SPIdriver2* the desired mode can also be defined in a more general way:

```
Define SPImode1 = 0; // 0, 1, 2, 3

SPIdriver2 = PortC, 7, 4, 1, 5; // SCK, MOSI, MISO, SS
SPIdriver2 = MSB;
SPIdriver2 = 0;
SPIdriver2 = 0;
```

SPIdriverX selects the bit functions of the used IO-port. First is the port name. Next are the positions of the clock pin, the MOSI and the MISO pins, and finally the position of the SS-pin (chip select). The latter is optional.

All pins or bits must be in one port. Splitting is not possible but you can use any bits of the port at any position.

The function of the SPIdriver (SPI Clock Polarity) and SPIdriver (SPI Clock Phase) or the SPI modes is explained in the AVR datasheets.

The definition of the SS pin (chip select) can also be omitted. Then the driver generates no chip selects and this becomes the job of the application. If the SS pin is defined, the driver activates the SS pin before an access to the slave and deactivates it after the access.

The SPI transfer rate is not selectable. It is about 1/16 of the processor clock. So with 16MHz the bitrate is about 1Mbit/sec.

3.21.1 Functions

These SPI Defines can not be changed at runtime:

Procedure SPIout1 (source : pointer; count : word);

Procedure SPIout2 (source : pointer; count : word);

Writes a datablock from **source** with the length **count** into the SPI-Slave.

SPIout1 (@ArrXY, sizeof (ArrXY));

Procedure SPIinp1 (dest : pointer; count : word);

Procedure SPIinp2 (dest : pointer; count : word);

Reads a datablock out of the SPI-Slave into the memory location **dest** with the length **count**

SPIinp2 (@ArrXY, sizeof (ArrXY));

Procedure SPIinOut1 (source, dest : pointer; count : word);

Procedure SPIinOut2 (source, dest : pointer; count : word);

Writes a datablock from **source** with the length **count** into the SPI-Slave and reads a datablock out of the SPI-Slave into the memory location **dest** with the length **count**

SPIinOut1 (@ArrXY, @RecordAB, 24);

Procedure SPIoutByte1 (b : byte);

Procedure SPIoutByte2 (b : byte);

Writes a byte into the SPI-Slave.

SPIoutByte2 (\$40);

Procedure SPIoutWord1 (w : word);

Procedure SPIoutWord2 (w : word);

Writes a word into the SPI-Slave.

SPIoutWord1 (\$4080);

Procedure SPIoutLong1 (L : longword);

Procedure SPIoutLong2 (L : longword);

Writes a longword into the SPI-Slave.

SPIoutLong2 (\$12345678);

Function SPIinpByte1 : byte;

Function SPIinpByte2 : byte;

Reads a byte out of the SPI-Slave.

bb:= SPIinpByte1;

Function SPIinpWord1 : word;

Function SPIinpWord2 : word;

Reads a word out of the SPI-Slave.

ww:= SPIinpWord2;

Function SPIinpLong1 : longword;

Function SPIinpLong2 : longword;

Reads a longword out of the SPI-Slave.

Lw:= SPIinpLong1;

Function SPIinOutByte1 (b : byte) : byte;

Function SPIinOutByte2 (b : byte) : byte;

Writes a byte into the SPI-Slave and reads simultaneously a byte out of the SPI-Slave.

bb:= SPIinOutByte2 (\$33);

Single-Bit Output:

Procedure SPIoutBit1(b : boolean);

Procedure SPIoutBit2(b : boolean);

Writes one bit into a Slave. Can be used for a 9bit output..

Example in XMega_SoftSPI



AVRco Standard Driver

3.22 Serial SPI Flash AT25DF (XMega)

Some applications need a huge amount of constant data, for example the EVE graphic system where big bitmaps and huge fonts are used. Also HTML pages can have very much kB to send. Store and restore large amounts of data in a very cheap SPI driven serial Flash device is a solution for these requirements. The device supported here is the AT25DF321 32MBit = 4MByte and the AT25DF641 64Mbit = 8MByte device. Both devices are identical 8pin SOIC chips. Optional 2 ROMs can be controlled. As all flash devices the read operations are always random and high speed. Single bytes and blocks upto 64kBytes can be read with one access with 50Mbit/sec speed. The XMega used here is limited to 16MBit/sec.

The drawback with flash devices is that non-empty blocks must be erased before they can be written. So the main purpose of such devices is mostly limited to a read-only device. The contents of the flash is written once and read often.

Import

As usual with AVRco the driver must be imported.

Import SysTick, AT25DFxx, ..;

Define

The driver needs the define of a Hardware or Software SPI:

Define

```
AT25DFxx    = SPI_E, PortE.3;    // hardware SPI, /CS
or Software SPI
//                      SS      SCLK  MOSI  MISO
AT25DFxx    = SPI_Soft, PortE.3, PortE.7, PortE.5, PortE.6;
AT25DF_ROM2 = PortE.2;          // optional 2nd chip select
```

Low Level Functions

These functions are normally not used by the application. They serve for very specific purposes.

procedure AT25_CS_Hi;

The SPI-chipselect output is set to log1 (inactive).

procedure AT25_CS_Lo;

The SPI-chipselect output is set to log0 (active).

procedure AT25_outByte(b : byte);

The byte b is send by the SPI-MOSI output.

function AT25_inpByte : byte;

This function reads one byte through the SPI-MISO input.

procedure AT25_outAddr(a : longword);

The Flash address is written into the AT25.

procedure AT25_write_enable;

The write-enable in the AT25 becomes activated.

procedure AT25_write_disable;

The write-enable in the AT25 becomes deactivated.

Support Functions

procedure AT25_SelectROM(rom : byte);

If the optional 2nd ROM is defined, so the actual desired ROM must be given here.

procedure AT25_checkBusy;

The busy flag in the AT25 is checked until it becomes inactive.

AVRco Standard Driver



procedure AT25_Reset;

This function aborts all running operations in the AT25 (write, erase etc).

function AT25_readStatus : word;

The status register of the AT25 is read out. Find further infos in the AT25 datasheet.

procedure AT25_protect_sector(sect : longword);

A 64kb flash sector in the AT25 becomes write- and erase protected. "sect" must address a 64kB sized area. Supplying the start address of this block is not necessary.

procedure AT25_unprotect_sector(sect : longword);

A 64kb flash sector in the AT25 becomes unprotected for write- and erase operations. "sect" must address a 64kB sized area. Supplying the start address of this block is not necessary.

procedure AT25_protect_all;

The entire flash in the AT25 becomes write- and erase protected.

procedure AT25_unprotect_all;

The entire flash in the AT25 becomes write- and erase protected.

function AT25_GetDeviceDensity : byte;

This function returns the flash size of the AT25 in Mbits. 32..64

Main Functions

procedure AT25_chip_erase;

This function executes a complete chip-erase. This means that all flash cells are set to \$FF. This can last upto 25sec.

procedure AT25_4kb_erase(addr : longword);

A 4kByte block in the flash becomes erased. "addr" must point into the desired block.

procedure AT25_32kb_erase(addr : longword);

A 32kByte block in the flash becomes erased. "addr" must point into the desired block.

procedure AT25_64kb_erase(addr : longword);

A 64kByte block in the flash becomes erased. "addr" must point into the desired block.

procedure AT25_read_array(addr : longword; buf : pointer; count : word);

This function reads an upto 64kB sized flash content out of the AT25. "addr" defines an arbitrary start address. "buf" is a pointer into the iData (dest) where the readout must be written. "count" is the amount of bytes to read.

procedure AT25_write_page(addr : longword; buf : pointer);

This function writes a 256 byte sized area out of iData into the AT25. "addr" is the destination address in the flash and must point to the start of a 256byte block in the AT25 (a multiple of 256).

procedure AT25_write_bytes(addr : longword; buf : pointer; count : byte);

This function writes an upto 256 byte sized area out of iData into the AT25. "addr" is the destination address in the flash. The maximal byte count is 256. If "count" = 256 then "addr" must point to the begin of a 256byte block in the flash. If "count" = 1 then "addr" can point to any location in the concerned 256byte block. "buf" points to the source in the iData area.

As with all flash write operations it must be sure that these memory locations which will be written must contain (\$FF). This can be done with a chip-Erase or a block-erase. If some bytes must be patched the in a block which already contains data, then the involved 4kB block must be readout. Then this contents are patched in the iData. Then this 4kB block must be erased in the AT25 and now the patched block can be rewritten into the AT25.

Example Program:

A sample is in the directory ..\E-Lab\AVRco\Demos\XMega_SerFlash

A second sample is in the directory ..\E-Lab\AVRco\Demos\XMega_SerFlash2



AVRco Standard Driver

3.23 TWI (I2C) Drivers Master and Slave

The I2C (TWI) interface is an international standardized, synchronous serial interface for the connection of peripheral modules to a processor as well as for the communication of processors. This implementation supports both the Master and Slave mode. With both modes the AVR must have a TWI module on chip, like the mega16.

Several slaves can be connected to a TWI master, where every slave has a special internal address, which should be unique within the so-called net. Theoretically more than 100 modules can be accessed, so you could talk about a network. The variant with 10-bit addresses is not supported.

The connection between each unit is made by two control lines: Data and Clock. These two lines are bi-directional. Each of the two lines must have external pull-up resistors of 1..10kOhm to +5Volt.

Slave Addresses

With slave addresses there is an important characteristic that one must bear in mind when these addresses are calculated or defined. In the datasheets the addresses of slave chips always include the read/write bit at location 0. For example if a slave has the address R/W \$80..\$81 in its datasheet, with AVRco this address must be changed to \$40, because the R/W bit must not be included here.

Slave address \$00 is not implemented in any component. It is a **Global Call** address, which is similar to a Broadcast Message in networks. Most of the slave CPUs can handle this address. More details are given in the slave section.

3.23.1 Master Mode

In this implementation a Master always uses polling scheme; there are no interrupts used. There are no retries if a function fails.

Lock-up in Master Mode

The internal TWI hardware of the mega CPUs is very sensitive to external distortions, especially on the CLK line. This can result in a complete malfunction. This affects the TWImaster mode driver of the library. To avoid these deadlocks some Timeout functions are implemented into these drivers. If such a Timeout occurs the TWI hardware will be reset and re-initialized and the library function will return with a false. If the external reason is removed the system continues without any intervention of the application. In order to use the Timeout support the **SysTick** must be imported.

3.23.2 Slave Mode

In this implementation a Slave always runs with interrupts.

TWI (I2C) Master

The TWI-interface, which is imported by **TWImaster** and defined by **TWIpresc** offers the three high-level functions TWIstat, TWIinp and TWIout. These functions are generalized, i.e. they do not depend on a particular slave chip. The programmer must observe the software protocol of the slave, by use and parameters, so that the telegram protocol (Hardware-Address, Clock-Generation etc) is implemented by the library driver.

All functions detect if the slave exists and is ready or if it doesn't exist or is BUSY. If the function failed, FALSE is returned, otherwise TRUE.

Both transfer-functions offer the capability of reading and writing. The type of the parameter data automatically determines the block length. For further information see TWIinp and TWIout.

AVRco Standard Driver



Imports

TWImaster

Defines and imports the Master TWI-Bus interface. With XMegas one of the TWI ports must be imported.

The TWI-Bus uses the onChip TWI module. The implemented interface can only work as Bus-master. If a Master/Slave switching is necessary the AVRco TWI-LAN driver should be used.

If the TWImaster is imported by the Import clause, then the corresponding TWI prescaler must be defined which controls the speed of the TWI channel.

The import of the TWImaster automatically imports the library functions TWIstat, TWIinp and TWIout. These functions not only support single byte transfers, but also block transfers. The functions return a true or a false according to the result.

Import SysTick, TWImaster;

Define ProcClock = 4000000; {4Mhz clock }
SysTick = 10; {10msec Tick}
TWIpresc = 32 (TWI speed)

XMega

Import SysTick, TWI_C; // TWI_C, TWI_D, TWI_E, TWI_F
Define OSCtype = int32MHz, PLLmul=4, prescB=1, prescC=1;
SysTick = 10; {10msec Tick}
TWIprescC = TWI_BR400; (TWI speed)
optional
TWInoTimeOutC = false; // or true, default = false

TWIpresc

Defines the value to load into the TWI's internal prescaler which controls the bitrate of the TWIbus in master mode. A large value reduces the speed, a small value increases the speed. But note that most of the I2C slaves can handle a bitrate of 100kBit/sec. In the real world the achievable bitrate can be limited by the wire length, the pullup resistors and other factors.

New I2C chip types can handle up to 400kBit/sec. But then the line length becomes critical. At the very least the pullups must be reduced to 1..2kOhm (this value is the sum of all resistors in parallel).

The system exports some related constants in master mode

Const TWI_BR100 : byte = nn; // nn = prescaler value for 100kBits/sec
TWI_BR400 : byte = nn; // nn = prescaler value for 400kBits/sec
TWI_BR500 : byte = nn; // nn = prescaler value for 500kBits/sec
TWI_BR600 : byte = nn; // nn = prescaler value for 600kBits/sec
TWI_BR800 : byte = nn; // nn = prescaler value for 800kBits/sec

TWInoTimeOut XMega

Normally the variable *TWI_TimeOut***tn** defines the Timeout with the TWI operations. With very fast slaves this Timeout in x-SysTick counts can be avoided. With this special and optional Define (true) the Timeout becomes reduced to about 4msec.

AVR+Mega driver export the byte variable *TWI_TimeOut* which can be in the range of 0..255 Systicks

Attention:

With the **XMega** the MasterSlave and Slave modes are not supported!

With the following functions the suffix **tn** is used for identifying the selected TWI_C, TWI_D, TWI_E, TWI_F
tn can be C, D, E or F

With the XMega TWI the variable *TWI_TimeOut***tn** is exported. This value is set to 100msec at system start, but can be set to any count of SysTicks (1..255) by the user.



AVRco Standard Driver

3.23.3 Functions

3.23.3.1 TWIstat

The function TWIstat returns a true if the chosen slave exists and is responding. In case of an error the function returns as FALSE. An EEPROM, for example, ignores any further accesses for a while after it was written. In this case it would return a false. In the case of error the function returns with a false. With a real TWI-Slave the function returns a false if the Slave is internally busy. In this case a false doesn't necessarily mean that the Slave not exists.

The definition of TWIstat is:

```
Function TWIstat (SlaveAdr : byte) : boolean;  
Function TWIstatTN (SlaveAdr : byte) : boolean; // XMega
```

The parameter SlaveAdr is the physical module address of the TWI-Bus.

3.23.3.2 TWIinp

The function TWIinp reads at least one byte of the selected slave. If the function fails, the function returns with the result FALSE, otherwise the result is TRUE.

The definition of TWIinp is:

```
Function TWIinp(SlaveAdr : byte; var Data) : boolean;  
Function TWIinpTN(SlaveAdr : byte; var Data) : boolean; // XMega
```

The parameter **SlaveAdr** is the physical module address of the TWI-Bus. Module-internal addresses (e.g. EEPROM) have to be adjusted with a write bit by TWIout.

The variable **Data** has to be defined as a variable, where the type of the variable is arbitrary. The SizeOf(Data) and/or the number of occupied memory locations determine the number of bytes, which are read within the block and are passed to the variable data. With a byte, char or boolean it is 1 byte, with a word or integer it is 2 bytes, where the low byte is read before the high-byte.

If Data is a type of array, then the length of the array is established and a corresponding number of bytes is read, starting with the lowest address of the array (lowest index).

If the Data is a string, the max. possible length of the string is established and a corresponding number of bytes is read, starting with the index string[0]. That means, that a string is always read and written incl. length byte. Reading the length byte determines the number of the bytes, which have to be read. But the max. length of the destination string will not be exceeded.

3.23.3.3 TWIinpP

A similar function with a pointer destination is also available:

```
Function TWIinpP(SlaveAdr : byte; dst : pointer; len : word) : boolean;  
Function TWIinpPTN(SlaveAdr : byte; dst : pointer; len : word) : boolean; // XMega
```

Attention:

By choosing the data variable pay attention, that the slave is able to handle a corresponding length of a blocktransfer. An TWI module, which only consists of an 8-bit parallel-port, is generally able to transfer one byte.

3.23.3.4 TWIout

The function TWIout writes at least one byte into the selected slave. If the function fails (TimeOut), the function returns with the result FALSE, otherwise the result is TRUE.

The definition of TWIout is:

Function TWIout (SlaveAdr : byte; Command : byte|word [; Data]) : boolean;

Function TWIout^{TN}(SlaveAdr : byte; Command : byte|word [; Data]) : boolean; // XMega

The parameter **SlaveAdr** is the physical module address of the TWI-Bus.

The parameter **Command** is depending on the slave, but has to be specified. It could be a byte variable, an 8-bit constant or a word. Often this command is a module-intern address (e.g. EEPROM).

With Slave CPUs the parameter Command must always be a Byte!!

Bigger EEPROMs (>256 bytes) have a 16-bit address. Because it would not make sense to implement special library functions, the function **TWIout must be** informed whether the address parameter (2nd parameter) consists of one or two bytes, i.e. if it is 8- or 16-bits long. Because of this the parameter has to be definite and has to be declared as byte or word (as appropriate).

3.23.3.5 TWIoutP and TWIoutWP

A similar function with a pointer source is also available:

Function TWIoutP(SlaveAdr : byte; Command : byte; src : pointer; len : word) : boolean;

Function TWIoutP^{TN}(SlaveAdr : byte; Command : byte; src : pointer; len : word) : boolean; // XMega

Function TWIoutWP(SlaveAdr : byte; Command : word; src : pointer; len : word) : boolean;

Function TWIoutWP^{TN}(SlaveAdr : byte; Command : word; src : pointer; len : word) : boolean; // XMega



AVRco Standard Driver

```
{ eeprom 8kBytes 24C65 }  
Var w : word;           { 16 bit adr }  
  
w:= 0000;  
b2:= 0;  
bool:= TWIout ($50, w);   { set adr for read = 0000 }  
while b2 < $FF do       { read eeprom with adr auto incr }  
    bool:= TWIinp ($50, b1); { result into b1 }  
    inc(b2);  
endwhile;
```

For components, which possess an additional internal address, for example EEproms and AD-converter, the required reading-address has to be set by a write-access before reading. The next reading-access then goes to this address. If the module possesses an auto-increment of the address (ADC, EEprom), it can be read continuously, without an output of a new address.

```
{ eeprom 256*8bit PCF8582E }  
Var b2 : byte;           { 8 bit adr }  
  
B2:= 00;  
bool:= TWIout ($53, b2);   { set adr for read = 00 }  
while b2 < $FF do       { read eeprom with adr auto incr }  
    bool:= TWIinp ($53, b1); { result into b1 }  
    inc (b2);  
endwhile;
```

The variable **Data** is optional and can be an 8- or 16-bit constant or any variable. Arrays, strings and string literals in ROM or EEprom are also possible. This parameter is optional and can, but does not need to, exist. The slave and/or the required slave-action determine the type of or omitting of data. The SizeOf(Data) or the number of the memory locations, which are occupied by this variable and/or constant, determine the number of bytes which are read out of the variable within the block and written into the slave. With a byte, char or boolean it is 1 byte, with a word or an integer it is 2 bytes, with the low order byte written before the high-byte.

If the result is a type of array, then the length of the array is established and a corresponding number of bytes is written, starting with the lowest address (lowest index) of the array.

If the Data is a type of string, the length of the string is established and a corresponding number of bytes is written, starting with the index string[0]. That means, that a string is always read and written incl. length byte. Within writing, it is the the length byte of the source string that determines the number of bytes written:

Count := length(Source).

Basically **Data** is treated as a block regardless whether it is a byte, word, string, array or record. Sometimes it can be useful to pass the result of an expression instead of firstly storing this result in a byte, word etc. but directly inserting expressions as Data into TWIout is not possible. In order to notify the compiler that **Data** is an expression and not a data area we use a special construction where **Data** is marked in a special way. To do this **Data** must be enclosed in parentheses.

```
bool:= TWIout ($53, b2, (a and b or c));
```

The result of this expression must be of byte, word, integer etc. Chars, boolean, strings etc are not allowed.

If the parameter **Data** was not specified, only the command byte is transferred.

Note:

When choosing the data variable/constant, make sure that the slave is able to handle a corresponding length of a block transfer. An I2C module, which only consists of an 8-bit parallel port, is generally able to transfer one byte.

The slave address 00 is the so-called **General Call** address. Normal I2C devices ignore this address. TWI-Slaves can, if enabled, respond to it. But a read access makes no sense because all slaves then send bytes and this gives an unexpected result.

3.23.4 Multi-Processing with the Master

In an application with Processes and/or Tasks in many cases the TWI-Bus is not only used as a network but also for other purposes (LCD, Ports etc). If so then different processes access the TWI and heavy conflicts inevitable, because sequential drivers (I2C, TWI, UART etc) are not re-entrant, i.e. they are not interruptible and cannot be re-entered. Because of this the TWI port works with a semaphore of the type DeviceLock.

```
TWI_DevLock : DEVICELock;  
TWI_DevLockTN : DEVICELock;
```

The TWI driver observes and controls this semaphore. At the entry into the driver the semaphore is checked to see whether the driver is locked or not. If it is free the semaphore becomes activated (driver locked) and the job will be executed. After finishing the job the semaphore becomes released (unlocked).

If the driver is locked (occupied) at entry time then a **Schedule** is executed and the calling Process is put into the Schedulers queue. With one of the next few schedulings this Process is restarted and again checks this semaphore. This is repeated until the semaphore becomes free (unlocked).

Attention:

Because of the abort of a TWI access through "Schedule" **Tasks should not** use TWI accesses. A schedule completely aborts and exits a Task and so a job will not be executed if a locked semaphore is found.

It is possible to set a flag if the Task solved the job successfully. If the flag is not set the Task must again be initiated to repeat this job, but this is somewhat complicated and should be avoided.

Example program:

An example is in the directory `..\E-Lab\AVRco\Demos\TWItest`

An XMega example is in the directory `..\E-Lab\AVRco\Demos\XMega_TWI`



AVRco Standard Driver

3.23.5 TWI (I2C) Slave

XMega: the slave modes are not implemented.

The TWI-interface, which is imported by **TWIslave** and defined by **TWIBuffer** and **TWImode** offers the four high-level functions **TWIRxStat**, **TWITxStat**, **TWISetRdy** and **TWIGetRdy**, **TWISetBUSY**, **TWIGetBUSY** and **TWIGetCMD**. These functions are generalized, i.e. they are not dependant on a particular Master chip. The programmer/application must control a handshaking by the proper use of the state functions.

A TWI/I2C Slave never can send or receive data by itself. All transfers from and to the Slave are processed by the Master.

Handshake operation

The Slave uses its state functions to avoid current data being overwritten by newly incoming data and also ensuring that previously transmitted data is only fetched once by the Master. Functions: **TWIGetRDY** and **TWISetRDY**.

Transparent operation

The RxBuffer and the TxBuffer act as a Dual port Memory. The Slave continuously reads data or commands out of the RxBuffer and also writes continuously new data into the TxBuffer. The Master reads and writes also both buffers. There is no additional information exchange, so the Slave cannot determine whether the Master has sent new data or fetched data. Functions: **TWISetBUSY** and **TWIGetBUSY**.

Support functions are **TWISetGC** and **TWISetSlaveAddr**.

Imports

TWIslave

Defines and imports the Master TWI-Bus interface.

The TWI-Bus uses the onChip TWI module. The implemented interface can only work as a Bus-slave. If a Master/Slave switching is necessary the AVRco TWI-net driver should be used.

If the TWIslave is imported then the Rx/Tx buffersize must be defined. These buffers are used for transmitting and receiving data from/to the Master.

A default Slave address must also be defined. This address can be changed at runtime.

The import of the TWIslave automatically imports all accompanying library functions.

Import *SysTick, TWIslave;*

Defines

Define *ProcClock = 4000000; {4Mhz clock }*
SysTick = 10; {10msec Tick}
TWladdr = 2; (TWI slave address)
TWlbuffer = 8, iData; (TWI rx/tx buffersize, location)
TWImode = Handshake; (TWI handshake or transparent)

TWladdr

Defines the default BUS-address of the Slave. Address 0 is not valid and neither are addresses greater than 127. The address can be changed at runtime.

TWlbuffer

The data exchange between Master and Slave goes through an RxBuffer and a TxBuffer. Both have an identical size and must be defined with this Define. The buffersize can be set between 2 and 255 bytes. The memory type (iData, xData) must also be stated. The buffers **TWIRxBuffer** and **TWITxBuffer** of array type are built.

TWImode

This Define installs the basic working method of the Slave. With the Transparent Mode there is no Handshake. Master and Slave both have unlimited access to the buffers of the Slave.

With the Handshake Mode only the Slave determines when the Master can read or write data.

3.23.6 Common Functions

TWIssetGC

The procedure TWIssetGC determines whether the Slave can receive Broadcasts (Slave addr = 0).

The handling of such a telegram is the job of the application. Only the master should make write accesses with address 0.

Procedure TWIssetGC (BroadcastEnable : boolean);

TWIssetSlaveAddr

The procedure TWIssetSlaveAddr changes the Slave address. Only address 1..127 is allowed.

Function TWIssetSlaveAddr (SlaveAddr : byte);

3.23.7 Functions in Handshake Mode

The Define TWImode determines the basic working method of the Slave. With the handshake mode the Slave always controls the communication and enables/disables the Master for send or receive.

Automatic disable

If the Master executes a read or write access to the Slave, the slave's driver disables the TWI interface completely and sets its state accordingly. From now on **each** further access of the Master will be denied with a NACK.

Because of this the slave's application must always check the RxState and the TxState. If one of the two functions returns with a true the interface is disabled and the slave's program must at least call "TWIssetRDY(true)" in order to re-enable the interface. But if the "TWIgetRxStat" resulted in a true the receive buffer must be analyzed first.

If the slave has filled its transmit buffer for a transmit to the Master, it should wait until the function "TWIgetTxStat" returns with a true. Only then has the Master fetched data and new data can be written into the TxBuffer. After this with the function "TWIssetRDY(true)" the interface should be enabled again so the Master access the slave again.

The function "TWIgetRDY" informs about the state of the interface in a more common mode then rxStat or txStat.

It is the job of the programmer to establish a principal communication that controls the behavior of the slave as to whether it should send or receive something. It's a good idea to use the command byte for this purpose.

Manual disable/enable

If the slave disables the TWI-interface with a call "TWIssetRDY(false)" then the Master gets a **NACK** from its I2C/TWI interface with the next access to this slave and a false is returned to its application. A disabled TWI-interface must be enabled by the slave with "TWIssetRDY(true)".

Internal addresses

Each read or write access of the Master resets the internal address-pointer of the slave to 0. The slave increments automatically this internal address with each byte transfer until the buffer end is reached or a completely new access is started by the Master.

The command byte of the Master with "TWIout" is stored separately in the Slave and can always be read by the slave with the function "TWIgetCMD". A write to the command byte of the slave without changing its RxBuffer looks like this:



AVRco Standard Driver

Master

TWIout (SlaveAdr, \$AA);

The command of the slave now contains \$AA. Please note that the slave also with a write access to its command byte disables its interface to inform the application that there was a write access from the Master.

TWIgetRxStat

The function TWIgetRxStat checks whether new data has been received since the last call of TWIsetRDY(true). If there is new data available a true is returned, otherwise a false. If the result is true the TWI-interface is also completely disabled.

Function TWIgetRxStat : boolean;

TWIgetTxStat

The function TWIgetTxStat checks whether the Master had data fetched from the Slave since the last call of TWIsetRDY(true). If all data has been transmitted a true is returned, otherwise a false. If the result is true the TWI-interface is also completely disabled.

Function TWIgetTxStat : boolean;

TWIgetRDY

The function TWIgetRDY returns the state of the TWI-interface (enabled/disabled). If enabled the result is true, otherwise false. If the result is false and if there was no manual disable then a Master access caused the disabled status. What kind of access this was can be found by calling the function TWIgetRxStat or TWIgetTxStat.

Function TWIgetRDY : boolean;

TWIsetRDY

The function TWIsetRDY(true) resets the state of the RxBuffer to empty and the TxBuffer to full, so the Master can receive or transmit a new packet/data. The TWI-interfaces becomes enabled.

With TWIsetRDY(false) the TWI interface is completely disabled but the state of the buffers stay unchanged.

This function is only successful if there is no access from the Master at this time. If the function was successful a true is returned otherwise a false.

Function TWIsetRDY (ready : boolean) : boolean;

TWIgetCommand

The function TWIgetCommand returns the last command passed by the Master with **TWIout**. The content of this command has no meaning with the Handshake Mode. The internal write and read pointer will also be reset to 0 by each command. If the command was a part of a complex write function each successive write of the Master automatically increments the internal pointer until the end of the buffer is reached or a new command has been received.

Function TWIgetCommand : byte;

3.23.8 Functions in Transparent Mode

The Define TWImode determines the basic working method of the Slave. If the Transparent Mode is selected the Master and the Slave both have unlimited access to the Slave's Rx and TxBuffer.

But if structures like words, integer etc. have to be transmitted there can be problems when both access an integer to the lo or hbyte because the TWI/I2C always transfers bytes. To also make this secure there are two disable/Handshake functions.

TWIssetBusy

The function TWIssetBusy disables or enables the buffers for Master accesses. IF the Slave accesses any buffer and it wants to prevent the Master doing a concurrent access, the Slave disables the buffer and the Master gets a **NACK** if it tries to access any buffer at the same time. This function should only be used if a transfer contains complex types. With simple bytes it makes no sense.

The function returns with a false if the blocking was not successful which is the case if the Master was already accessing a buffer at this time.

Function *TWIssetTxBusy (busy : boolean) : boolean;*

TWIsgetBusy

This function returns the state of the TWI-interface that was set with TWIssetBusy. The function returns a true if the TWI is disabled/busy otherwise a false is returned.

Function *TWIsgetBusy : boolean;*

Attention:

The command parameter of the Master-function TWIout has a special meaning in the Transparent Mode. It defines the start address of the following read or write access in the related buffer.

Master

TWIout (SlaveAdr, 2, \$AA);

Slave

The data \$AA is stored in Slave RxBuffer memory location 2. RxBuffer[2]

Master

TWIout (SlaveAdr, 3); / set read addr

TWIinp (SlaveAdr, bb);

Slave

The data in memory location 3 is read. TxBuffer[3]

The Slave automatically increments the command until the end of the buffer is reached or a new command has been received.

With TWIs slave mode without handshake there are two status bytes:

TWIsTxCount and *TWIsRxCount*. With each received or transmitted byte the corresponding count is then incremented by 1.

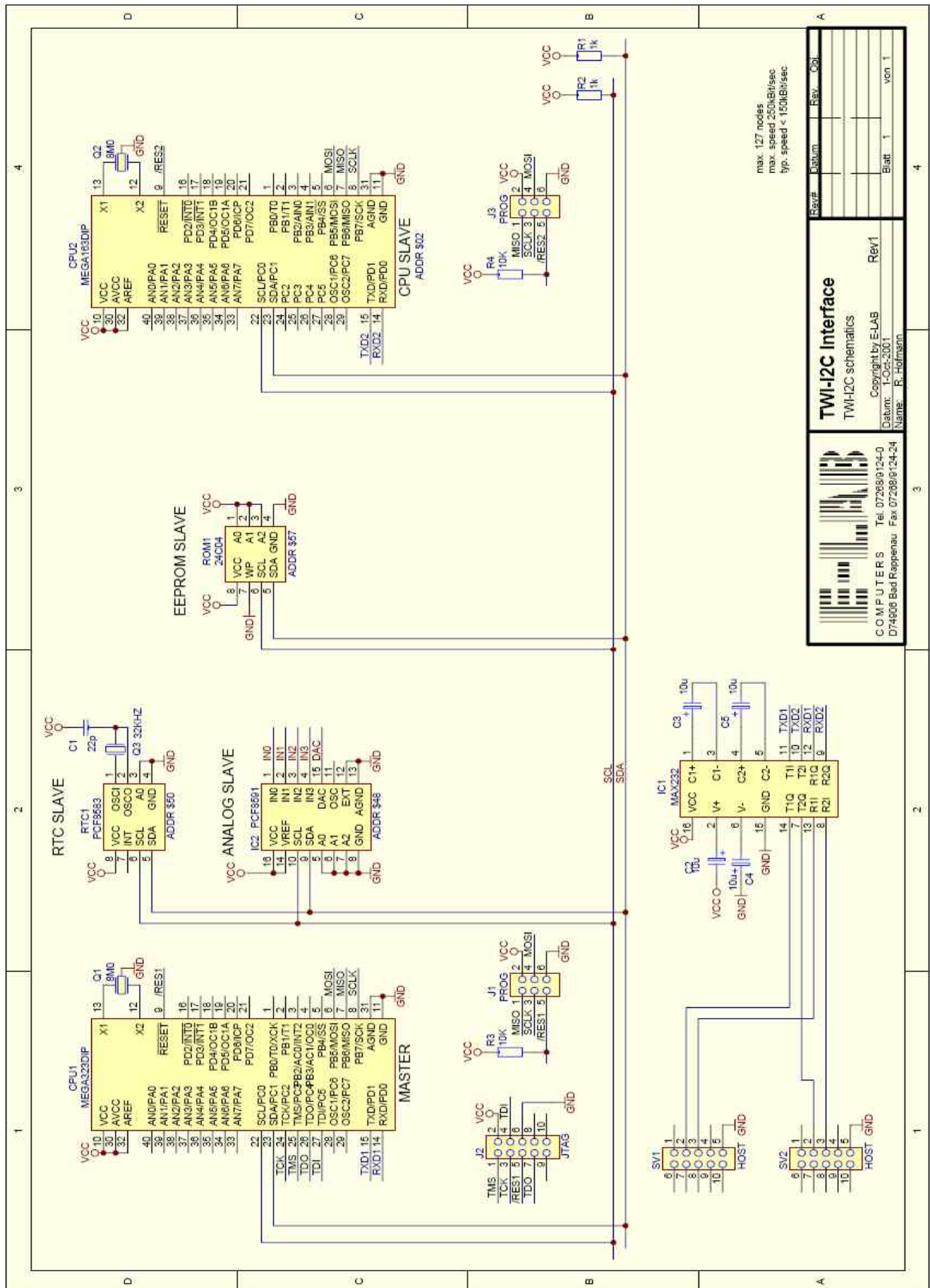
Example programs and schematic:

An example of a Master implementation is **TWImaster**.

An example of a Slave-transparent implementation is **TWIs slaveTrn**.

An example of a Slave- handshake implementation is **TWIs slaveHsk**.

These are in the directory **..\E-Lab\AVRco\Demos\TWItest**.



schematic **TWI**

3.24 TWI-Net Library Driver

There are several ways how to establish a communication between CPUs in a multi-processor system. There is for example DualPort memory, SPI, RS232, CAN-Bus and also Ethernet.

Some of them must be discounted because of limited space or cost. A very simple but efficient way is to use the I2C bus for this purpose. All newer Atmel megas have an I2C build in which Atmel calls a **TWI**.

I2C is a simple but secure and moderate fast bus implementation. But using as a network the nodes are limited to CPUs which have the I2C/TWI implemented in hardware on the Chip. With pure onBoard nets the TWI-network should preferred. With inSystem networks which connects several boards the TWI is better than the SPI version and at least equals the RS485/LAN version. With longer distances and speed is a need the LAN network fits better. If speed is not so much important the TWI network is also useable with longer distances (some meters).

The only drawback of the TWI/I2C is its open-collector construction which in conjunction with the line length determines the maximal possible bitrate.

The AVRco implementation is mainly for such CPU-CPU connections. The strategy consists of a Multi-Master-Multi-Slave system. For a secure communication a proper frame protocol is implemented.

3.24.1 TWI network

A Multi-Master / Multi-Slave system is implemented which consist always of at least one Master and at minimum one Slave. Each Slave can become a Master and each Master can become a Slave, but more than one active Master at a time leads to possible collisions and should be avoided or at least must be handled with extreme care.

The Master controls all connected Slaves. No Slave should gain control of the BUS as a Master, although this is always possible. All data transfer, whether from or to the Slave are initiated and executed by the Master. By this method the serious difficulties with collisions and priorities are overcome. The mode change of a Slave to Master is supported by the library driver but must be handled with care by the application. The best way is that the active Master transfers the control of the bus to a Slave and the changes itself to a Slave or at least stays inactive. Thus collisions, which are not really detectible, can be avoided and are manageable.

Networks commonly work with telegrams, also called frames or packets. Through the usage of frames there is some overhead, but on the other hand the security of the system is increased. The TWI also implements a Broadcast feature, which this network uses as a command or info channel.

With the I2C/TWI operation only one subscriber can transfer data at one time. All others are listening or inactive. For example the master requests a frame from a slave. For that it sends a frame or command to the slave with the desired contents (address, data, commands). The slave recognizes its address and informs its user program about the reception of the frame/command. If the user program in the slave must send an answer to the master (not a must), it responds by initializing a frame, which the Master then reads out of the Slave. The Master places the received frame in its buffer and informs its application.

The contents and meaning of a frame is ambiguous and has to be defined by the user. So too is whether a slave responds with a responding frame or not. If the master is waiting for an answer, the slave should respond immediately because the master application is waiting and sends no further frames/commands until an answer is ready in the Slave. But this "handshaking" must be done by the application. It is not handled by the master or slave software itself.



AVRco Standard Driver

The slave can also initiate a frame transmit by building a txFrame. The master recognizes this by polling the slaves and informs its application. The master always can send a frame to a slave provided the receive buffer of the slave is empty and the slave is active.

The slaves are completely interrupt driven; the master works in polling mode.

Principles of operation:

The **Master** transmits and receives all frames that are transferred on the BUS and provides it to its application program. There is no analysis or utilization in the master except the checksum checking.

Each **Slave** always listens on the BUS, but fetches only frames if the inherent hardware address is identical with its on address (node address). There is no analysis or utilization in the slave except the checksum checking. It's the job of the application to interpret the frame content.

The slave also doesn't respond with an acknowledge or an answer to the master, except for the TWI/I2C built-in hardware checks. This is also the job of the application.

Slave addresses must be greater than 0 and lower than 128. The address "0" is reserved for the Broadcast Command to all slaves. All slaves receive the broadcast frame (2 bytes, + optional frame), and their user program must analyze and interpret it. There must be no bus reaction of any slave.

The actual frame length (bytes) of a frame can vary from 0 bytes up to TWIframe -1. The address is max 7 bits (one byte) and therefore 126 nodes are possible. The checksum is always an 8 Bit Check.

Only the actual frame length is variable. Address size and checksum are fixed.

Structure of a frame:

| ADDRESS | FRAMELENGTH | DATA, DATA ... | CHECKSUM |

Address:

The address consists of a byte (1..127 slaves).

Each slave should have a unique address. The master has no address, because it transmits and receives all frames, regardless of the inherent address. But the Master also needs its own address because it can be switched to a Slave. If the Master transmits a packet it inserts its own slave address into the packet. If a Slave prepares a frame that will be fetched by the Master it also inserts its own address into the packet. This means that the internal address in a packet is always the address of the sender/initiator of the packet.

FrameLength:

The frame length is determined by define TWIframe = nn. If parameter nn < 256 there is one length byte transferred, if nn >= 256, 2 length bytes are transmitted. The actual frame length can be between 0 and nn bytes. In spite of this, all BUS subscribers must have the same kind of address size (Byte/Word), so the frame descriptors are always identical. The implemented buffersizes can vary in the range of a byte or word.

DATA:

The data area within a frame is defined by the length byte or word. The data length can be zero so the length byte/word also contains a zero.

CHECKSUM:

The checksum is computed over the complete frame. It is always an 8bit checksum.

Attention:

The max. frame length must be defined at design time with the use of "Define". Also the Master, Slave or MasterSlave mode. A change at runtime is only possible for the MasterSlave mode. The actual frame buffer sizes can vary in the participants but only in the range of 0..FrameLength-1. Either all nodes possess a buffer size smaller than 256 Bytes (FrameLength is in one byte) or greater than 255 bytes (FrameLength is in one word). A mix is not allowed.

Security

The user's program in the master and slave can use an acknowledge cycle. This means that the master sends a frame and then waits for a response from the slave. This results in a very secure system, but the throughput is somewhat (or greatly) reduced. It is the job of the participating devices and/or the programmer to implement such a method.

Structure of a Broadcast Commands:

| SUBADDR | COMMAND |
or optional:
| SUBADDR | COMMAND | FRAMELENGTH | DATA, DATA ...

SubAddress:

This address is a byte and is used by the Slave to validate a Broadcast Command.

Command:

The Command is a byte and will be recognized and processed by the Slave after a validation (SubAdr, Mask etc).

FrameLength:

The parameter FrameLength is always a byte. To use the optional/additional data frame with broadcasts the broadcast frame must be defined with Define TWIframeBC = nn; Then it is possible for a dataframe with up to 255 bytes to be appended. The FrameLength can be 0..255.

The main job of a Broadcast is control a specific or all Slaves with commands, independent of their actual TxFrame or RxFrame state. So for example it is possible to break deadlocks caused by incomplete or destroyed Frames when running in a multiprocessing system and using semaphores.

Transfer:

With the slave the data transfer is completely handled in interrupt procedures. Therefore it is guaranteed that if a frame is completely built, then this frame is transferred to the target in a very short time. It is important that in the slave the interrupts must not be disabled for a long time. This is especially true at high bit rates. With 100kBit/sec each byte transfer generates an interrupt in 100usec cycles.

Line considerations:

The I2C/TWI bus is an open collector bus. This means all drivers only can switch to ground. A logical 1 is always generated from the pullups. The reason is that both SCL and SDA lines always act bidirectionally. Each time the Slave can insert a clock stretch by pulling down the clock line. In addition to this the Slave must acknowledge some states by pulling down the data line. All this leads to a simple bus hardware but also supports a secure communication.

The disadvantage with I2C/TWI is the limited speed imposed by the open collector-pullup combination. The lower the pullup resistor is the higher the speed and/or line length can be. But also the power consumption increases. The pullups must not fall below 500Ohms; otherwise the port pin drivers of the CPU can be damaged (overcurrent).

Instead of Pullups, constant current sources can also be used. These provide at the same power consumption with a much shorter rise time and therefore allow a faster bitrate.

Example program:

An example is in the directory ..\E-Lab\AVRco\Demos\TWI.net



AVRco Standard Driver

New I2C chip types can handle up to 400kBit/sec. But then the line length becomes critical. At a minimum the pullups must be lowered to 1..2kOhm, where this value is the sum of all resistors in parallel.

The system exports some related constants in master mode

Const *TWI_BR100 : byte = nn; // nn = prescaler value for 100kBits/sec*
 TWI_BR400 : byte = nn; // nn = prescaler value for 400kBits/sec

These constants can be used in the define section:

Define *ProcClock = 4000000; {4Mhz clock }*
 SysTick = 10; {10msec Tick}
 TWIpresc = TWI_BR100; {TWI speed 100kBit/sec}

3.24.2 Implementation

Imports

The TWIport must be imported, as usual in the AVRco system, by an import directive.

Import *TWInet;*

Defines

Define *TWInode = \$12; {node addr, always necessary}*

Declaration of the node address. Can be changed at runtime.

Define *TWInetMode = Master; {Master, Slave or MasterSlave}*

The operation type of the unit as a master, slave or MasterSlave must be defined. A change of this type at runtime is only possible if MasterSlave has been defined.

Define *TWIpresc = 32; {prescaler, bitrate for Master mode}*

Defines the bitrate of the Master. Must be >= 8

The frame type of the TWI must be **identical** in the Master and Slave and must be defined by the following define:

Define *TWIframe = 16, iData; {Framesize max. 16 bytes in iData}*

Both Master and Slave have a receive- and a transmit buffer. All Masters and Slaves should possess the same buffer size (not an absolute must). The defined buffer size determines also the required internal memory. The transferred frames can be less than but never larger than this parameter.

Because the actual frame length is transmitted as a parameter in the frame, all frames must be managed with either a byte or a word as the length information. A mix of them is not possible. This limitation simplifies the protocol and its handling dramatically and is not a serious limitation. All frame size definitions therefore must fit either into a byte or a word, and thus must be smaller than 256 bytes or can be larger depending on this.

The size of these buffers and the memory location are determined by the above Define.

With Broadcast telegrams it is possible send an optional data frame in addition to the two standard parameters **SubAddr** and **Cmd**. This option must be imported with the following Define:

Define *TWIframeBC = 64; {Broadcast Framesize max. 16 bytes}*

Also here either all or none of the Bus participants (Master and Slaves) must have this Import. If imported then all must have the same Framesize.

3.24.3 Exported Types

Type *tTWInetmode* = (TWInetSlave, TWInetMaster);

Type *tTWIStates* = (TWIRxEmpty, TWIRxBusy, TWIRxFull, TWITxEmpty, TWITxBusy, TWITxFull, TWIbcCMD, TWIstatFail);

Type *tTWIState* = set of *tTWIStates*;

3.24.4 Exported Variables

3.24.4.1 Memory Organization

All the TWI related user accessible variables reside one after another in the defined memory area. The order is, beginning with the lowest address, the following:

TWIRxStatReg	Byte
TWIRxADR	Byte
TWIRxLEN	Byte at a Framelength < 256 bytes, word at a Framelength > 255 bytes
TWIRxBUFF	Array[0..TWIframe-1] of Byte;
_TWIRxCHK	Byte
TWITxStatReg	Byte
TWITxADR	Byte
TWITxLEN	Byte at a Framelength < 256 bytes, word at a Framelength > 255 bytes
TWITxBUFF	Array[0..TWIframe-1] of Byte;
_TWITxCHK	Byte

TWIRxStatReg, TWITxStatReg

These two variables (Byte) must only be read! Both pseudo status registers are continuously updated while a receive or transmit of a frame is going on. If idle, all bits are zero. If the action proceeds the bits become a "1" from right to left. If the action is finished successfully, all bits are "1" and the byte has the value \$FF. Values between \$00 and \$FF mean either an error has occurred or the frame isn't completely received or transmitted yet. If a send frame is processed, there can be no error. If a frame was received a \$9F means that a checksum error occurred. The user program must poll the bit7 of this register to see if a frame is complete or not. A correct and complete received frame shows a \$FF after a receive process, and with the Master and Slave this byte becomes "00" after a successful complete transmit operation.

(* bit0,1	:	00 = idle	*)
(*		01 = start	*)
(*		11 = node addr	*)
(* bit2,3	:	00 = idle	*)
(*		01 = first len	*)
(*		11 = second len	*)
(* bit4	:	0 = rx/tx frame	*)
(*		1 = rx/tx frame	*)
(* bit5,6	:	00 = idle	*)
(*		01 = start check	*)
(*		11 = check ready	*)
(* bit7	:	0 = processing	*)
(*		1 = rx/tx complete	*)



AVRco Standard Driver

The following show the states of the Status Register during receiving or transmitting a Frame, if there is no error.

Master transmits: TWItxStatReg
 00h idle.
 01h Frame is completely assembled
 02..FFh transmit in progress
 00h transmit complete

Master receives: TWIrxStatReg
 00h idle
 01..7Fh receive in progress
 FFh receive is completed

If the application has processed the received Frame it must invalidate it with "TWIRXCLEAR".
 The Status Register then contains 00h

Slave transmits: TWItxStatReg
 00h idle.
 01h Frame is completely assembled
 02..FFh transmit in progress
 00h transmit complete

Slave receives: TWIrxStatReg
 00h idle
 01..7Fh receive in progress
 FFh receive is completed

If the application has processed the received Frame it must invalidate it with "TWIRXCLEAR".
 The Status Register then contains 00h

TWIrxFr, TWItxFr

These vars are of type byte

Master:

The Master transmits and polls all frames. Because the I2C/TWI bus Master transmits all frames to the Slave and also fetches available frames from the Slave, the Master and its application program always know where the frame is sent to or received from.

If the Master sends a frame it must insert its own node address into the variable "TWItxFr". The address "0" is reserved for broadcast commands and must not be occupied by a master or slave for its own node address.

Slave:

A slave only receives those frames whose hardware address is equal to its own node address. The node address of the sender (Master) is stored into the location "TWIrxFr". Broadcasts with the target address "0" are processed separately. See below.

"TWItxFr" always contains the node address of the Slave. This is done by the driver.

TWIrxFr, TWItxFr

These variables are of type byte if "Define TWIframe" < 256 bytes and of type word if "Define TWIframe" > 255 bytes. Frames can have a zero length. The maximum length cannot exceed the number of "Define TWIframe = nn". Basically all slaves and the master should have the same "TWIframe" sizes, but it's possible that some slaves have a shorter value of "TWIframe". The master's user program must know this and can only send frames to these nodes with a length that will fit into the receive buffers of those slaves.

TWIrxFr is updated whilst receiving a frame with the value of which is received internally in the frame.

TWItxFr is set by the function "TWItxFr(Node, len)" and also transmitted within the frame.

TWIrxBuff, TWItxBuff

The size of these two buffers is determined by the Define TWIframe. The buffers can be seen as an Array of byte and therefore can be also accessed in this manner.

The indexes of these arrays are 0 to TWIframe -1.

```
TWItxBuff[0] := $56;
X := TWIrxBuff[6];
```

It's recommended and very useful to overlay a structure (record) over the Rx part and the Tx part of the TWI memory. By that a symbolic access to the complete data structure is possible.

```
type   tTWIRec = record
        TWIstate : byte;           // TWI state, size and loc fixed
        TWInode : byte;           // node, size and loc fixed
        TWIlen  : byte/word;      // framelen, size and loc fixed
        TWIusr1 : byte;           // user defined
        TWIusr2 : word;           // user defined
        TWIdata : array[0..TWIframe-4] of char; // user
    end;
```

```
var
{$NOOVRCHECK}
    TWIRxRec[@TWIrStatReg] : tTWIRec;
{$NOOVRCHECK}
    TWITxRec[@TWItStatReg] : tTWIRec;
```

```
// write Frame
    TWITxRec.TWIusr1:= $30;
    TWITxRec.TWIusr2:= $3231;
    TWITxRec.TWIdata[0]:= '3';
```

_TWIrChk, _TWItChk

These are of type byte. The checksum is calculated over the complete frame (TWIRXADR + TWIRXLEN + TWIRXBUFF). The same is true for both Tx and Rx. The checks are processed automatically while receiving and transmitting. These two vars are only for information and to analyse any occurring errors.

3.24.4.2 Variables only in MasterSlave and Slave Mode

The following variables are only present in MasterSlave mode and in Slave mode:

```
Define TWIInetMode = MasterSlave;
or
Define TWIInetMode = Slave;
```

In addition they are only relevant if also the Slave mode is activated if the MasterSlave id defined.

The Broadcast variables control the behavior of a Slave if a Broadcast Command of a Master is received. I2C/TWI Broadcasts are initiated by the hardware address 00h. All Slaves receive all Broadcasts. To make further decisions a sub-address and a mask are implemented. The Slave driver uses them to find out whether a Broadcast Command is of interest or not. The Master which starts the Broadcast at first sends a data byte which contains the command byte followed by the sub-address. As an option the additional Broadcast frame is sent. All Slaves then internally have to perform the following operation:



AVRco Standard Driver

```
if (SubAddr or TWIadrMask) = (NodeAdr or TWIadrMask) then  
    TWIbroadcastAdr:= SubAdr;  
    TWIbroadcastCMD:= CMD;  
    Receive optional Broadcast frame;  
    Inc (TWIBROADCASTSEMA);  
endif;
```

TWIadrMask

This type is a byte. In case of a Broadcast the Slave uses this mask in conjunction with its NodeAdr to find out whether the received sub-address is sufficient or not.

TWIBroadcastADR

This type is a byte. This variable contains the sub-addr of the last accepted Broadcast Command.

TWIbroadcastCMD

This type is a byte. This variable contains the command part of the last accepted Broadcast Command.

Broadcast Option

If Broadcast Frames were imported by Define "TWIframeBC" there are additional parameter and data in the Master and Slave applications.

TWIBroadCastCount

This type is a byte. This variable contains the initial count (in bytes) of the frame send by the master.

TWIBCStatReg

This type is a byte. This variable contains the count (in bytes) that was received in the frame by the slave.

TWIBroadCastBuffer

This type is a Array[0.. TWIframeBC-1] of Byte. This variable contains the received frame.

A check of the BroadCast frame can be done by comparing the two variables TWIBCStatReg and TWIBroadCastCount. If there is difference the frame invalid or defective.

3.24.5 Multi-Processing with the Master

In an application with Processes and/or Tasks in many cases the TWI-Bus is not only used as a network but also for other purposes (LCD, Ports etc). If so, then different processes access the TWI, and severe conflicts are inevitable because such sequential drivers (I2C, TWI, UART etc) are not re-entrant, i.e. they are not interruptible and cannot re-entered. Because of this the TWI port works with a semaphore of the type DeviceLock.

TWI_DevLock : DEVICELOCK;

The TWI driver observes and controls this semaphore. On entry into the driver the semaphore is checked as to whether the driver is locked or not. If it is free the semaphore becomes activated (driver locked) and the job will be executed. After finishing the job the semaphore is released (unlocked).

If the driver is locked (occupied) at entry time then a *Schedule* is executed and the calling Process is put into the Schedulers queue. With one of the next few schedulings this Process is restarted and again checks this semaphore. This is repeated until the semaphore becomes free (unlocked).

Attention:

Because of the suspension of a TWI access through "Schedule" *Tasks should not* use TWI accesses. A schedule completely aborts and exits a Task and so a job will not be executed if a locked semaphore is found.

It is possible to set a flag to determine if the Task solved the job successfully. If the flag is not set the Task must again be initiated to repeat this job. But this is somewhat complicated and should be avoided.

3.24.6 Multi-Processing with the Slave

TWlbbroadcastSEMA

This type is a semaphore. With each accepted Broadcast Command from a Master the Slave increments this sema by one. The content of the variables "TWlbbroadcastAdr" and "TWlbbroadcastCMD" are overwritten with new values. The same is also true for the optional "TWlbbroadcastBuffer"

TWlrxSEMA

This type is a semaphore. With each received Frame from a Masters the Slave increments this sema by one. Note that a checksum error also leads to an increment because the Frame is complete.

TWltxSEMA

This type is a semaphore. With each transmitted Frame to a Masters the Slave increments this sema by one.

If processes are used it makes sense that a specific process waits for the reception of a Frame. This can be done with "WaitSema(TWlrxSema)". The process then waits until a Frame has been completely received. Because a checksum error also increments the sema the process must use "GetTWlslaveSTAT(0)" or "TWlrxSTAT" to find out the real state of the received Frame. The "TWlrxStatReg" then contains 00h or, in case of an error, an errorcode.

A process can also wait for an incoming Broadcast Command with "WaitSema(TWlbbroadcastSema)".

For the Slave it's also possible to let a process wait until the Master has completely fetched a provided Frame. This can be done with "WaitSema(TWltxSema)". The process is woken up if the Frame sent. Then the "TWltxStatReg" contains a 00h.

If there are invalid and incomplete Frames caused through corruptions then it's possible that a semaphore will be never incremented and the accompanied process will never get woken up.

3.24.7 Exported Functions and Procedures

Procedure *SetTWlnodeAddr (sAddr : byte);*

Each TWI Master and Slave must have its own unique node address which must exist only once in a network. This address is assigned by the Define TWlnode = nn. However it is possible to change this address by this procedure.

This address is used as a hardware address when reading or writing a TWI Slave. The hardware address is copied into the Frame internal addresses TWlrxAdr and TWltxAdr by the sender when a Frame is built for a transmit operation. Then a receiver of a Frame always can detect the sender of this Frame.

Function *GetTWlslaveSTAT (node : byte) : tTWlNetState;*

This function serves to retrieve the state of the local Rx and Tx buffers and Frames and with a Master also to check the state of a Slave's Frames. If the system is in slave mode or defined as a pure Slave the parameter "node" will be ignored and the local state is always returned.

If the system is in Master Mode or defined as a pure Master then with the parameter "node" = 0 the function returns the local state and with "node" <> 0 it returns the state of the Slave which has the internal hardware address "node".

The result of the function is a bitset that is constructed in this manner:

Type tTWlStates = (TWlRxEEmpty, TWlRxEBusy, TWlRxEFull, TWlTxEmpty, TWlTxBusy, TWlTxFull, TWlbcCMD, TWlstatFail);

Type tTWlNetState = set of tTWlStates;



AVRco Standard Driver

If the system is a Master and the result is retrieved from a Slave the specific values have the following meaning:

TWIRxEmpty	the RxBuffer of the Slave is empty and can receive a Frame.
TWIRxBusy	the RxBuffer is occupied but has errors
TWIRxFull	the RxBuffer is full but not processed by the Slave
TWITxEmpty	the TxBuffer of the Slave is empty
TWITxBusy	the Slave is assembling a new Frame or the last Frame had errors
TWITxFull	the Slave has built a new Frame that can be fetched by the Master
TWlbcCMD	the Slave has not invalidated the last Broadcast Command
TWlstatFail	the Slave does not respond to any request

The processing and analysing of the result can be done in this way:

```
TWlstate:= GetTWlslaveStat (02);  
if TWlstatFail in TWlstate then  
..  
else if [TWlrxEmpty, TWltxFull] in TWlstate then  
  // either send a frame or receive a frame  
else ...
```

If a TWI node checks itself the value “TWlstatFail” can never occur.

Function TWlrxStat : boolean;

This function returns a true if a frame has been received complete with or without errors and has not been invalidated with “TWlrxClear”. Then the program must analyse the **TWlrxStatReg** to check for errors. As long as a frame is complete, or if there are errors, a new incoming frame is discarded. It is the job of the user program to process a received frame as fast as possible and then invalidate it so a new frame can be received. Frames cannot get lost but the system can be blocked.

Procedure TWlrxClear;

The statusbyte of the RxBuffer is reset. With the slave another frame can be received. If the system is in MasterSlave or Slave Mode the semaphore “TWlrxSema” is also reset.

Function TWltxStat : boolean;

This function checks whether a frame is already sent or not. If the frame is sent or invalidated by TWltxClear the function returns with a true; otherwise a false is returned. A new frame can only be sent if the current frame is completely transmitted or the state is reset with “TWltxClear”.

Procedure TWltxClear;

The statusbyte of the TxBuffer is reset. Another frame can be transmitted. If the system is in MasterSlave or Slave Mode the semaphore “TWltxSema” is also reset.

Function TWltxFrame (node : byte; len : byte[word]) : boolean;

This function first checks whether the last frame was sent successfully and also that the parameter “node” is valid (0 < node < 128). If this fails a false is returned. In this case by using TWltxClear a frame can be reset without deleting the data and TWltxFrame can be called again. If the frame can be transmitted the “len” parameter is stored in “TWltxLen”. The parameter “node” should have always the same value as the target’s variable “TWlnodeAdr”. It is used as the hardware address. The sender’s node address is stored into “TWltxAdr”.

1. In the case of the Master the transmitting starts. If the Slave responds the entire Frame will be sent. If successful, the “TWltxStatReg” now contains 00h. If the Slave doesn’t respond or there is a transmission error the “TWltxStatReg” now contains a value > 00h and < FFh. For example if the Slave doesn’t respond the resulting value is 80h. The Master’s application also can use the function “GetTWlslaveStat(slavenode)” to check the current state of the Slave before transmitting a Frame.

2. If a Slave calls TWItxFrame this function initializes the Frame and changes "TWItxStatReg" to 01h and returns with a true, but the Frame is still not transmitted. Only if a Master reads this Frame with its "TWIrxFrame" function will the Slave's status register is reset to 00h and also the semaphore "TWItxSema" be incremented. The parameter "node" is not used by a Slave transmit, but this must also be valid.

3.24.8 Functions and Procedures only available in MasterSlave mode

Procedure *SetTWImode (const twimode : tTWInetmode);*

This procedure switches the device mode from Master to Slave and vice versa. A Master should only switch itself into Slave mode if it is ensured that a Slave becomes a Master after this. A Slave should switch itself into the Master mode if it gets the permission from the current Master. With two or more Masters on the bus collisions are certain, and these cannot be handled in an easy way.

tTWInetmode = (TWInetSlave, TWInetMaster);

The current mode can be found by reading the system variable `_TWInetMode`.

Important:

In MasterSlave mode a TWI system always starts up in slave mode. If this system is to become a Master this procedure must be called.

If called, this procedure clears all buffers, frames, states and semaphores.

3.24.9 Functions and Procedures only available in MasterSlave and Master mode

These functions must only be used if the system is in Master Mode or if it is defined as a pure Master.

Function *TWItxBroadcast (cmd : byte; subnode : byte) : boolean;*

This function generates an I2C Broadcast to all Slaves. Each I2C port of an active Slave always receives all Broadcasts by definition. I2C Broadcasts are pure write accesses. The Broadcast consists of the hardware address 00h, an appended Command byte and the sub-node address. Hence the Frame consists of 2 bytes.

If the "TWIframeBC" was defined/imported, a variable length Frame is appended. The function below must be used:

Function *TWItxBroadcast (cmd, subnode, count : byte): boolean;*

The Master fills the Array TWIBroadCastBuffer with data and passes the bytecount in the function above in the parameter "count". The slave then reads these bytes out of its own TWIBroadCastBuffer.

The Broadcast variables control the behavior of a Slave if a Broadcast Command of a Master is received. I2C/TWI Broadcasts are initiated by the hardware address 00h. All Slaves receive all Broadcasts. To make further decisions a sub-address and a mask are implemented. The Slave driver uses them to find out whether a Broadcast Command is of interest or not. The Master which starts the Broadcast first sends a data byte which contains the command byte followed by the sub-address. All Slaves must then internally do the following operation:

```
if (SubAddr or TWIadrMask) = (NodeAdr or TWIadrMask) then  
    TWIbroadcastAdr:= SubNode;  
    TWIbroadcastCMD:= CMD;  
    //Optional receive data frame;  
    Inc (TWIbroadcastSema);  
endif;
```



AVRco Standard Driver

With each accepted Broadcast Command from a Master the semaphore "TWIbroadcastSema" is incremented by one. The content of the variables "TWIbroadcastAdr" and "TWIbroadcastCMD" are overwritten with new values. The same is also true for the optional "TWIBroadCastBuffer".

A Broadcast will be received regardless of what the internal Rx or Tx state of the Slave is. It also doesn't change these states because a Broadcast with the Master and the Slaves is processed totally independently to the common Frames.

Function *TWIrxFram* (*node : byte*) : *boolean*;

With a Master this function reads a Frame from a Slave. First the parameter "node" is checked. If it is zero or > 127 the function returns with a false. Then it checks whether the last received Frame was processed by the Master's application (TWIrxFramStatReg = 00h). If this fails a false is returned. In this case by using TWIrxFramClear a frame can be reset and TWIrxFram can be called again. The parameter "node" should have always the same value as the target's variable "TWInodeAdr". It is used as the hardware address.

1. The Master now starts reading the Slave. If the Slave responds the entire Frame will be received. If successful, the "TWIrxFramStatReg" now contains FFh. If the Slave doesn't respond or there is an transmission error the "TWIrxFramStatReg" now contains a value > 00h and < FFh. For example if the Slave doesn't respond the resulting value is 80h. The Master's application also can use the function "GetTWIslaveStat(slavenode)" to check the current state of the Slave before receiving a Frame.

If the frame is completely received the actual byte count of the RxBuffer is stored in "TWIrxFramLen". The sender's node address is stored into "TWIrxFramAdr".

2. If a Master has read an entire Frame with its "TWIrxFram" function the Slave's status register "TWIrxStatReg" is reset to 00h and also the semaphore "TWIrxSema" will be incremented.

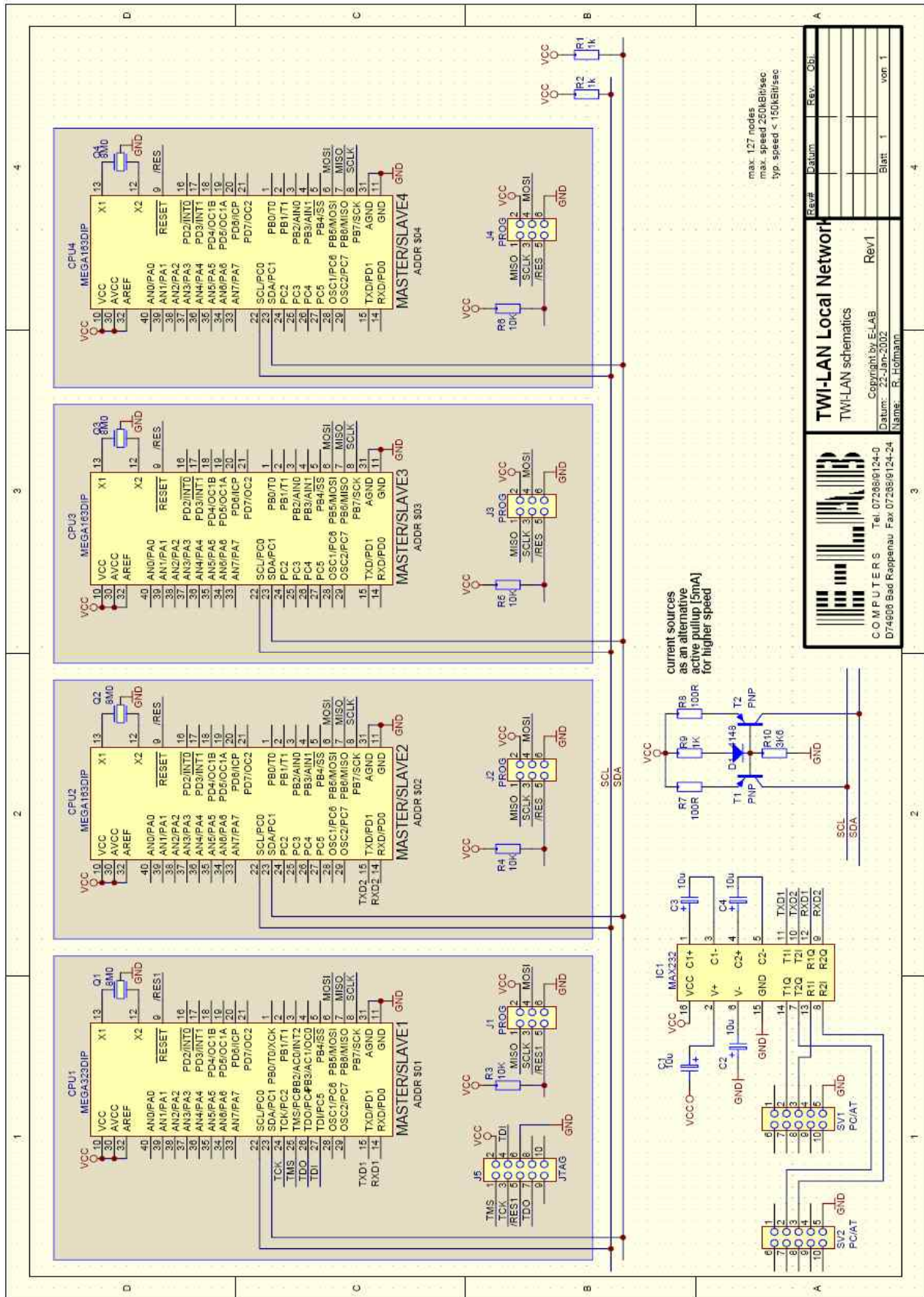
Example programs and schematic:

three examples

TWInetMastr,
TWIslave and
TWInetMsSI

are in the directory ..\E-Lab\AVRco\Demos\TWInet

AVRco Standard Driver



schematic TWI_Net

3.24.10 Additional Functions

In order to also have an access to standard I2C-devices in the TWInet the general functions for I2C chip access are also implemented. But here "Slave" means a normal I2C-chip and not network slave! So don't access net-slaves with these functions and also don't access standard I2C chips with the network functions!

TWlstat

The function TWlstat returns a true if the chosen slave exists and is responding. In case of an error the function returns as FALSE. An EEprom, for example, ignores any further accesses for a while after it was written. In this case it would return a false. In case of error the function return with a false.

With a real TWI-Slave the function returns a false if the Slave is busy internally. In this case a false doesn't mean that the Slave does not exist.

The definition of TWlstat is:

***Function** TWlstat (SlaveAdr : byte) : boolean*

The parameter **SlaveAdr** is the physical module address of the TWI-Bus.

TWlinp

The function TWlinp reads at least one byte of the selected slave. If the function fails, the function returns with the result FALSE, otherwise the result is TRUE.

The definition of TWlinp is:

***Function** TWlinp (SlaveAdr : byte; var Data) : boolean;*

The parameter **SlaveAdr** is the physical module address of the TWI-Bus. Module-internal addresses (e.g.. EEprom) have to be adjusted with a write by a TWIout.

The variable **Data** has to be defined as a variable, but the type of the variable is arbitrary. The SizeOf(Data) or the number of occupied memory locations also determines the number of bytes which are read within the block and are passed to the variable data. With a byte, char or boolean it is 1 byte, with a word or integer it is 2 bytes, where the higher order byte is read before the low-byte.

If Data is a type of array, then the length of the array is established and a corresponding number of bytes is read, starting with the lowest address of the array (lowest index).

If the Data is a string, the max. possible length of the string is established and a corresponding number of bytes is read, starting with the index string[0]. That means, that a string is always read and written incl. length byte. The length byte does not determine the number of the bytes that have to be read, but rather the length of the destination string string does:

Count := sizeOf(Destination).

Important:

By choosing the data variable make sure that the slave is able to handle a corresponding length of a block transfer. A TWI module, which only consists of an 8-bit parallel-port, is generally able to transfer one byte.

TWlout

The function TWlout writes at least one byte into the selected slave. If the function fails (TimeOut), the function returns with the result FALSE, otherwise the result is TRUE.

The definition of TWlout is:

Function *TWlout (SlaveAdr, Command : byte|word [, Data]) : boolean;*

The parameter **SlaveAdr** is the physical module address of the TWI-Bus.

The parameter **Command** depends on the slave, but has to be specified. It could be a byte variable, an 8-bit constant or a word. Often this command is a module-internal address (e.g. EEPROM).

With Slave CPUs the parameter Command must always be a Byte!!

Bigger EEPROMs (>256 bytes) have a 16-bit address. Because it would not make sense to implement special library functions, the function **TWlout must be** informed if the address parameter (2nd parameter) consists of one or two bytes, that is whether it is 8- or 16-bits long. Because of this the parameter has to be definite and has to be declared that way.

```
{ eeprom 8kBytes 24C65 }
Var   w : word;                { 16 bit adr }

w:= 0000;
b2:= 0;
bool:= TWlout ($50, w);          { set adr for read = 0000 }
while b2 < $FF do              { read eeprom with adr auto incr}
    bool:= TWlout ($50, b1);     { result into b1 }
    inc (b2);
endwhile;
```

For components that have an additional internal address, for example EEPROMs and AD-converter, the required reading-address has to be set by a write-access before reading. The next read-access then goes to this address. If the module has an auto-increment of the address (ADC, EEPROM), it can be read continuously, without an output of a new address. For example:

```
{ eeprom 256*8bit PCF8582E }
Var   b2 : byte;               { 8 bit adr }
B2:= 00;
bool:= TWlout ($53, b2);        { set adr for read = 00 }
while b2 < $FF do              { read eeprom with adr auto incr}
    bool:= TWlout ($53, b1);     { result into b1 }
    inc (b2);
endwhile;
```

The variable **Data** is optional and can be an 8- or 16-bit constant or any variable (but arrays, strings and string literal are not allowed). This parameter is optional and can, but does not have to exist. The slave and/or the required slave-action determine the type of or omitting of data. SizeOf(Data) or the number of the memory locations which are occupied by this variable or constant determine the number of bytes that are read out of the variable within the block and written into the slave. With a byte, char or boolean it is 1 byte, with a word or an integer it is 2 bytes, where the higher byte is written before the low-byte.

If the result is a type of array, then the length of the array is established and a corresponding number of bytes is written, starting with the lowest address (lowest index) of the array.



AVRco Standard Driver

If the Data is a type of string, the length of the string is established and a corresponding number of bytes is written, starting with the index string[0]. That means, that a string is always read and written incl. length byte. Within writing, the length byte does not determine the number of the bytes that have to be written but rather the length of the source string determines this:

Count := sizeof(Source)

If the parameter data was not specified, only the command byte is transferred.

Important:

When choosing the data variable/constant, make sure that the slave is able to handle a corresponding length of a block transfer. An I2C module, which only consists of an 8-bit parallel-port, is generally able to transfer just one byte.

Lock-up in Master mode

The internal TWI hardware of the mega CPUs is very sensitive to external distortions, especially on the CLK line. This can result in a complete malfunction. This concerns the TWInetmaster mode driver of the library. To avoid these deadlocks some Timeout functions are implemented into these drivers. If such a Timeout occurs the TWI hardware will be reset and re-initialized and the library function returns with a false. If the external cause is removed the system continues without any intervention of the application.

In order to use the TimOut support the **SysTick** must be imported.

3.25 AD-Converter

3.25.1 ADCPORT

Defines and imports the AD-converter (only if it exists in the chip).

The AD-converter is read in the background of the interrupt-service *SysTick* and is restarted. The reason is, that a free-running ADC (depending on the processor) would cause an interrupt every 10 to 40µsec. Because of the necessary administration work the system would be 'dense' with these interrupts. A polling of the converter would have the same result in the end. The ADC administration by the *SysTick* e.g. all 10msec makes nearly no load.

Import *SysTick, ADCPort;*

Define *ProcClock = 4000000; {4Mhz clock }*
SysTick = 10; {10msec Tick}
ADCchans = 2, iData; {use 2 Channels}
ADCpresc = 16; {prescaler 16}

3.25.2 ADCchans, RAMpage

Required number of ADC-channels.

It has to be between 1 and the max defined number of channels, which is in the description file (xxx.dsc) of the processor. The recorded values are stored in reserved memory locations, and they can be read by *GetAdc* any time. A value of 4 means, that channels 1..4 are used.

Sometimes it's useful that only one or two channels are imported, e.g. 4 and 7. To save resources one or two channels can be random defined:

ADCchans = [4, 7], iData;

or only one channel:

ADCchans = [5], iData;

or a range of channels

ADCchans = [2..5], iData;

Result integration:

ADCchans = 2, iData, int2; {2-times integrate}

With "Int2" the previous result is added to the new result and then the sum is divided by two.

Result:= (prevValue + newValue) div 2;

ADCpresc

Required prescaler of the AD-converter. This value determines conversion time of the ADCs and also indirectly the resilience to spikes and noise in the signal.

3.25.3 Functions and Procedures

GetADC

Returns data of the ADC. Function to indirectly read the ADC-channels.

ADCport and *SysTick* **have to be** imported. The passing parameter indicates the required channel (1..*ADCport*). The parameter runs from 1 to n. The result of the function always is an unsigned word.

a:= GetAdc (1); {read first channel}

With only **one** active channel the corresponding function is

a:= GetAdc; {read a single channel}



AVRco Standard Driver

SetADCfixed

With an ADC-import if there are more than 1 channel defined, the ADC-mux will be advanced in every SysTick by one step.

If for example 6 channels are imported, a specific channel will be converted each 6th SysTick.

This could be a problem if a control loop is activated at limited time. It can be too slow.

With

Procedure SetAdcFixed (fix : boolean; chan : byte);

the ADC can be fixed to a single channel (fix =true) and the mux does not advance. The other channels remain with their previous results until this procedure is called again with fix = false.

In this case the parameter "chan" will be ignored

3.25.4 Call-Back Function onADCread

With some special AD-Conversions sometimes it is necessary that the application gets informed if an ADC channel is read. For this reason the procedure "onADCread" is implemented.

Procedure onADCread;

So if the application implements this procedure:

Procedure onADCread;

begin

...

end;

then "onADCread" is called with each SysTick (Timer Interrupt) at the point where the ADC result is read. ACCB/ACCA (R16/R17) (lo/hi) contain the ADC result and ACCALO (R18) contains the actual channel number (0..7). For example it is possible to use only one ADC channel of the CPU and expand it with an external analog multiplexer.

Because the ADC driver always places this result into the same memory location (variable) the application must handle the result and must store it by using the current multiplexer setting. Of course the external multiplexer must also be handled here.

The same is true here as with other Call-Back functions from SysTick:

no big data management, no function calls etc. If possible write all in assembler.

Important:

1. Local parameters are prohibited.
2. The register SREG, _ACCA/R17, _ACCB/R16, _ACCALO/R18, _ACCCLO/R30 and _ACCCHI/R31 are saved always. If Pascal statements or other registers are used, then the additional register must be saved manually before their use and then restored.

Example program:

an example is in the directory ..\E-Lab\AVRco\Demos\NonLinear

3.26 AD-Converter XMega

3.26.1 ADC_A ADC_B

Defines and imports the AD-Converters. (as far as present in the chip).

The XMegas provide upto 2 separate AD-Converters. They are always connected to PortA and PortB and are named ADC_A and ADC_B.

These converters can work absolutely autarc. This means they can scan continuously upto 8 channels and store the results into the channel's result register. Because a Port can have upto 8 inputs the application must decide which of the 8 pins must be scanned. With this implementation there is no Interrupt and no DMA used. The SysTick is only used if more than 4 pins/channels should be scanned;

The settings are a resolution of 12bit, single-ended, unsigned and right justified.

Import *SysTick, ADC_A, ADC_B;*

Define

```
// The XMegas don't provide any Oscillator fuses.
// So the application must setup the desired values
// possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCType    = int32MHz,
            PLLmul=4,
            prescB=1,
            prescC=1;

SysTick     = 10;                // {msec
StackSize   = $0032, iData;
FrameSize   = $0064, iData;

ADCCrefA    = REF100;            // intern 1.0V reference, input pins at PortA, 4 channels
ADCprescA   = 256;               // prescaler 256
ADCchansA   = [0..7];           // 8 channels in use, using SysTick

ADCCrefB    = REFextB;           // extern reference, input pins at PortB, 3 channels
ADCprescB   = 32;                // prescaler 32
ADCchansB   = [1];              // single channel, no SysTick used
```

3.26.2 Defines

ADCCrefA, ADCCrefB

These defines set the reference voltage source.

```
REF100      = internal 1.0Volt reference
REFintVCC   = internal reference VCC div 1.6
REFintVCC2  = internal Referenz VCC div 2
REFextA     = external reference at PortA
REFextB     = external reference at PortB
```

ADCprescA, ADCprescB

Required prescaler of the AD-converter. This value determines conversion time of the ADCs and also indirectly the resilience to spikes and noise in the signal (4, 8, 16, 32, 64, 128, 256, 512).

As an option the desired resolution can be defined here:

```
ADCprescA = 4, 8;                // prescaler 4, resolution 8bits (8 or 12)
```

Without this parameter the resolution is always 12bit.



AVRco Standard Driver

ADCchansA, ADCansB

Defines the channel count and also the associated port pins.

Possible Defines are:

```
ADCchansA = [0..7];           // 8 channels in use PortA.0 .. PortA.7
ADCchansA = [1..3];           // 3 channels in use PortA.1 .. PortA.3
ADCchansA = [1, 3, 5, 7];     // 4 channels in use PortA.1, PortA.3, PortA.5, PortA.7
ADCchansA = [0];              // 1 channel in use PortA.0
ADCchansA = 1;                // 1 channel in use PortA.1
```

Please note that if more than 4 channels/pins are defined then the SysTick must also be imported!

3.26.3 Functions and Procedures

GetADC

Returns data of the ADC. Function to direct/indirectly read the ADC-channels.

ADC_A or ADC_B **have to be** imported. The passing parameter indicates the required channel (0..7). The result of the function is always an unsigned word.

Function GetAdcA(chan : byte) : word;

Function GetAdcB(chan : byte) : word;

The parameter chan must be in the range of 0..7. But if there are only 2 pins defined as inputs then chan must be in the range of 0..1. For example if the pins 2,4,5,7 are set as inputs then there is this relation:

```
Port.Pin2 > chan0
Port.Pin4 > chan1
Port.Pin5 > chan2
Port.Pin7 > chan3
```

Illegal values of chan will have a result of \$0000.

```
w:= GetAdcA(0);    {read first channel}
w:= GetAdcB(3);
```

Restrictions with the XMEGAE5 types:

More than one channel can only be defined with the [m..n] scheme:

```
ADCchansA = [1..3];           // 3 channels in use PortA.1 .. PortA.3
```

ADC_B does not exist and then also no Defines for it.

Example program:

an example is in the directory ..\E-Lab\AVRco\Demos\ XMEGA_ADC

3.27 DA-Converter XMega

3.27.1 DAC_A DAC_B

Defines and imports the DA-Converters. (as far as present in the chip).

The XMegas provide upto 2 separate DA-Converters. They are always connected to PortA and PortB and are named DAC_A and DAC_B. The settings are a resolution of 12bit and right justified.

Import SysTick, DAC_A, DAC_B;

From System Import ;

Define

*// The XMegas don't provide any Oscillator fuses.
 // So the application must setup the desired values
 // possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz*

*//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
 OSCtype = int32MHz, PLLmul=4, prescB=1, prescC=1;
 SysTick = 10; {msec}
 StackSize = \$0032, iData;
 FrameSize = \$0064, iData;*

*DAC_A = chan01, REF100; // DAC_A channel 0 + 1 used
 DAC_B = chan0, REFextB; // DAC_B channel 0 used*

XMega "U" Typen:

*DAC_A = chan1, REFextB; // DAC_A channel 1 only used
 DAC_B = chan1, REFextB; // DAC_B channel 1 only used*

3.27.2 Defines

DAC_A, DAC_B

Each DAC provides upto 2 channels, chan0 and chan1. If there is only one channel present or only one channel should be used so chan0 must be selected. If there are two channels and both should be used the chan01 must be defined. The second parameter defines the reference voltage source:

REF100 = internal 1.0Volt reference
 REFaVCC = external reference at AVCC pin
 REFextA = external reference at PortA
 REFextB = external reference at PortB

3.27.3 Functions and Procedures

SetDAC

Writes a new output value (12bit) into the DAC register. DAC_A or DAC_B or both **must be** imported and defined.

Function SetDacA_chan0(val : word);
Function SetDacA_chan1(val : word);
Function SetDacB_chan0(val : word);
Function SetDacB_chan1(val : word);

SetDacA_chan0(0); // write to DACA chan0
 SetDacB_chan1(www); // write to DACB chan1

Example program:

an example is in the directory ..\E-Lab\AVRco\Demos\ XMEGA_DAC



AVRco Standard Driver

3.28 Real Time Clock

Some AVR's (4434, 8535, mega83, mega161, mega163, mega103 etc) feature a so called Real Time Counter, but which is not comparable with a RTC-Chip. It's possible to build a Real Time Clock in software without much code size, provided that the CPU has an "ASSR" register in the IO-area and also two pins to connect a watch quartz crystal. This is available in at least the above CPU-types. With CPU-types which don't support an RTC (8515) or where the timers are in use for other jobs, with a minimal additional expense a precise RTC can be build with the help of the SysTick.

Our **RTC** has the functionality for adjusting time (hh:min:sec) and optional date (yy.mm.dd) and the corresponding readback functions. For the RTC-time it's possible to implement so called CallBack functions. If implemented they are called each new second, minute or hour ticks.

With additional optional imports there is one or more **Timer** (DownCount Alarm) and an **Alarm** (time/date). Both supplementary functions also have a CallBack which calls the related procedure if an event occurs.

The RTC needs the ASSR register, the related timer (Timer0 or Timer2) and an external 32768Hz watch quartz. If the SysTick is used for the RTC the timer and quartz are not necessary.

As with all RTCs there is not only the precision problem but also setting the clock to the correct time and date. In Europe there is the DCF77 time transmitter which is always used for the initial setup of RTCs. The DCF77 driver of AVRco can be used for this. Please see the DCF77 documentation.

Imports

The driver must be imported as usual with AVRco.

Import SysTick, RTclock, ...;

If needed the optional event procedures must be imported.

From RTclock Import RTctimer, RTCalarm;

The type of the RTC (Time-only or Date+Time) and also the memory location must be given with the Define:

Defines (mega103)

```
Define ProcClock = 6000000;      {Hertz}
          SysTick   = 10 , Timer2; {msec}
          RTclock    = iData, Time; {Time or DateTime}
```

As an option it is possible to import more than one timer channels

```
Define RTctimer = 4;              {1..8 Channels}
```

As a replacement for the internal RTC channel:

```
Define RTCsource = SysTick[, adj]; {optional}
```

Attention:

If the SysTick is selected as the RTC-source then the SysTick must fulfill certain conditions. The SysTick must not be lower than 1, it must be an integer number and $1000 / \text{SysTick}$ must not have a remainder ($1000 \bmod \text{SysTick} = 0$).

3.28.1 RTC-Functions/Procedures

Basic and implementation dependent procedures and functions.

Procedure *RTCsetSecond (sec : byte);*

Loads the seconds-register with the value "sec"

Procedure *RTCsetMinute (min : byte);*

Loads the minutes-register with the value "min"

Procedure *RTCsetHour (hour : byte);*

Loads the hours-register with the value "hour"

Procedure *RTCsetDay (day : byte);*

Loads the days-register with the value "day". Only implemented if Define was used with "DateTime".

Procedure *RTCsetWeekDay (wday : byte); {0 = Monday}*

Loads the weekday-register with the value "wday". Only implemented if Define was used with "DateTime".

Procedure *RTCsetMonth (month : byte);*

Loads the month-register with the value "month". Only implemented if Define was used with "DateTime".

Procedure *RTCsetYear (year : byte);*

Loads the year-register with the value "year". Only implemented if Define was used with "DateTime".

Attention: Adjusting the RTC should only be done with interrupts disabled!

Function *RTCgetSecond : byte;*

Reads the seconds-register

Function *RTCgetMinute : byte;*

Reads the minutes-register

Function *RTCgetHour : byte;*

Reads the hours-register

Function *RTCgetDay : byte;*

Reads the days-register. Only implemented if Define was used with "DateTime".

Function *RTCgetWeekDay : byte; {0 = Monday}*

Reads the weekday-register. Only implemented if Define was used with "DateTime".

Function *RTCgetMonth : byte;*

Reads the months-register. Only implemented if Define was used with "DateTime".

Function *RTCgetYear : byte;*

Reads the year-register. Only implemented if Define was used with "DateTime".

Attention: Reading of the RTC should only be done with interrupts disabled!

The three procedures below are the exported **CallBack** procedures. This means, if these are present in the source, they will be called at every second, minute or hour overflow. Pay attention that these functions are called directly from the Timer-Hardware-Interrupt.

As with all interrupt procedures only the few main working registers are saved (`_ACCA/R16`, `_ACCB/R17` and the Z-register `_ACCCLO/HI`). Because of this only the above register can be used within this procedures. Otherwise additional registers must be saved manually by the programmer.

These procedures must be build in the application source!



AVRco Standard Driver

Procedure *RTtickSecond;* {Callback procedure}
Is called at every second overflow.

Procedure *RTtickMinute;* {Callback procedure}
Is called at every minute overflow.

Procedure *RTtickHour;* {Callback procedure}
Is called at every hour overflow.

3.28.2 Alarm-Procedures

If **RTCalarm** is imported an alarm function is exported which calls the Callback procedure "RTCalarm" if the preset Time or preset Time+Date is reached.

Procedure *RTCalarm_Time (hour, min, sec : byte);*
Loads the time-compare registers of RTCalarm. Compare function becomes stopped.

Procedure *RTCalarm_Date (year, month, day : byte);*
Loads the date-compare registers of RTCalarm. Only implemented if Define was used with "DateTime". Compare function becomes stopped.

Procedure *RTCalarm_Start (mode : byte); {mode 0= stop, 1= Time, 2= DateTime}*
Starts or stops RTCalarm.

Procedure *RTCalarm_Stop;*
Stops RTCalarm.

Procedure *RTCalarm;* {Callback procedure}
Will be called at a compare-match, this means onetime at most.
This procedure must be build in the application!

3.28.3 Timer-Procedures

If **RTctimer** is imported one or up to 8 timer function can be used which decrement the preset value by one at each second tick. If this value is zero the Callback procedure "RTctimer" is called.

Procedure *RTctimer_Load (seconds : word[longword]);* // single timer defined
Procedure *RTctimer_Load (chan : byte; seconds : word[longword]);* // multiple timer defined
Loads the time-compare registers of RTctimerX. Compare function becomes stopped.
If LongWord or LongInts are imported then seconds is a LongWord (32bits), otherwise it's a Word (16bits).

Procedure *RTctimer_Start;* // single timer imported
Procedure *RTctimer_Start (chan : byte);* // multiple timer imported
Starts RTctimer.

Procedure *RTctimer_Stop;* // single timer imported
Procedure *RTctimer_Stop (chan : byte);* // multiple timer imported
Stops RTctimer.

Procedure *RTctimer;* {Callback procedure}
Procedure *RTctimer (chan : byte);* {Callback procedure}
This Callback procedure is called once if the Downcounter = 0. The timer is then stopped.
This procedure must be build in the application!

AVRco Standard Driver



Remarks:

The RTC must run with Timer0/2 **Interrupt**. If the interrupt is disabled more then a second the second gets lost for the RTC and the time gets wrong.

CallBack procedures are directly called from within the timer interrupt and there is only a minimal register saving. Because of this one must carefully inspect the generated assembler code in the callback procedure.

In doubt save additional registers (ASM) or write the procedure's code with assembler.

CallBack procedures must be build by the programmer.

Procedure RTctimer; {CallBack procedure, only one timer imported}

Begin

Elapsed:= true;

End;

Procedure RTctimer(chan : byte); {CallBack procedure, more than one timer imported}

Begin

Elapsed:= true;

End;

Preset and **Readback** of RTC registers should be done with disabled interrupts! Otherwise the results can be faulty if an interrupt occurs whilst reading or writing.

DisableInts;

RTCsetSecond (x);

RTCsetMinute (y);

RTCsetHour (z);

EnableInts;

With all other operations (also RTctimer_Load, RTCalarm_Time and RTCalarm_Date) a disabled interrupt is not necessary.

SysTick uses an 8bit timer and the RTC does. The timer for the RTC is fixed in the CPU. Because of this the SysTick must use the other 8bit timer of the CPU (Timer0 or Timer2).

If the RTC is driven by the SysTick "Define RTCsource = SysTick" a Timer is not needed.

The optional parameter "adj" in the definition of RTCsource = SysTick serves as a fine adjust for the RTC.

With repeatedly checking and changing this value (min -100, max +100) that correcting value can be found where the RTC's error is a minimum. A precision of +/- 1sec/day is possible. A precise clock, at best a radio clock must be used for the adjusting process.

Example programs:

examples are in the directories

..\E-Lab\AVRco\Demos\RTclock and ..\E-Lab\AVRco\Demos\RTC8564



AVRco Standard Driver

3.29 I2C-Bus

3.29.1 I2CPORT

Defines and imports the I2C-Bus interface.

The I2C-Bus is software moderated implemented. The interface can only work as Bus-master. An eventually existing hardware-I2Cbus-interface is not used and is not supported.

If the I2Cport is imported by the Import clause, so the corresponding parallel-port and both required bits I2Cclk and I2Cdat have to be defined by Define. Both bits must reside in the same port. The portbits must be able to work bidirectional.

The import of the I2Cport automatically imports the library functions I2Cstat, I2Cinp and I2Cout. These functions do not only support single byte transfers, but also block transfers. The functions return a true or a false according to the result.

Import *SysTick, I2Cport;*

```

Define ProcClock   = 4000000;    {4Mhz clock }
          SysTick    = 10;         {10msec Tick}
          I2Cport     = PortB;     {use port B}
          I2Cclk      = 0;         {clock-pin = port B bit 0}
          //I2Cclk    = 0,NOPs    {NOPs1, NOPs2...NOPs10}
          //I2Cclk    = 0, 2      {2x sDelay}
          //I2Cclk    = 0, @myDel {x sDelay} // myDel = variable which can be changed at runtime
          I2Cdat      = 3;         {data-pin = port B bit 3}

```

I2Cclk

Defines the portpin 'Clock' of the port, which was specified by the I2Cport for the I2Cbus prosecution. The specified bit has to be switched as in- or output.. See also I2Cdat. Value 0..7.

The speed of the I2C-Bus can be set with additional optional Define parameter:

Define *I2Cclk = PortBit, NOPs*

"NOPs" is a either a constant value between 1..255 or a variable name. Both define the count of CPU-cycles which are inserted between each state change of the I2C data and clock line.

In case of the variable it's possible to change the speed at runtime for the needs of the different I2C chips. In reality these NOPs are executed by a **sDelay**. This delay needs about 3 CPU cycles for one given value. If a value of 10 is given, then the result is about 30 CPU cycles. Instead of the constant or variable it is possible to give the NOP count directly:

Define *I2Cclk = 0, NOPs3; // PortBit, 3 NOPs [0..10]*

I2Cdat

Defines the portpin 'Data' of the port, which was specified by the I2Cport for the I2Cbus prosecution. The specified bit has to be switched as in- or output.. See also I2Cclk Value 0..7.

3.29.2

Alternatively the complete define can be given in one line:

Define *I2Cport = Port, DataPin, ClockPin[, NOPs]; // NOPs = optional parameter*

Example:

Define *I2Cport = PortC, 2, 3, NOPs2;*

Define *I2Cport = PortC, 2, 3, 2; // 2 sDelays into the clock delay, XMegas*

3.29.3 Functions and Procedures

I2Cstat

The function I2Cstat establishes, if the chosen slave exists and is responding. In case of error the function returns as FALSE, otherwise the result is true. An EEprom, for example, ignores any further access for a while after it was written. In this case it would return a false. There are max. 255 tries until abort. In case of error the function return after ca. 6msec with a false.

Function I2Cstat (SlaveAdr : byte) : boolean;

The parameter **SlaveAdr** can be a byte variable or a 8-bit constant. According to the definition the 8th Bit of the address is the Read-Write Bit and because of that it is overwritten or ignored by the function. This address is the physical module address of the I2C-Bus.

I2Cinp

The function I2Cinp reads at least one byte of the chosen slave. If the try fails (TimeOut), the function returns with the result FALSE, otherwise the result is TRUE. There are max. 255 tries until the abort. In case of error the function returns after ca. 6msec with false.

Function I2Cinp (SlaveAdr : byte; var Data) : boolean;

The parameter **SlaveAdr** can be a byte variable or an 8-bit constant. According to the definition the 8th Bit of the address is the Read-Write bit and because of that it is overwritten or ignored by the function. This address is the physical module address of the I2C-Bus. Module-inter addresses (e.g.. EEprom) have to be adjusted with a write by a I2Cout.

The variable **Data** has to be defined as a variable, whereby the type of the variable is arbitrary. The SizeOf(Data) or the number of occupied memory locations also determine the number of bytes, which are read within the block and are passed to the variable data. With a byte, char or boolean it is 1 byte, with a word or integer it is 2 bytes, whereby the lower byte is read before the high-byte.

If Data is a type of array, so the length of the array is established and a corresponding number of bytes is read, starting with the lowest address of the array (lowest index).

If the Data is a string, the max. possible length of the string is established and a corresponding number of bytes is read, starting with the index string[0]. That means, that a string is always read and written incl. length byte. At reading the length byte determines the number of the bytes, which have to be read. But the max. length of the destination string will not be exceeded.

Attention:

By choosing the data variable pay attention, that the slave is able to handle a corresponding length of a blocktransfer. An I2C module, which only consists of an 8-bit parallel port, is generally able to transfer one byte.

I2Cout

The function I2Cout writes at least one byte into the choosen slave. If the try fails (TimeOut), the function returns with the result FALSE, otherwise the result is TRUE. There are max. 255 tries until the dropout. In case of error the function returns after ca. 6msec with false.

Function I2Cout (SlaveAdr : byte; Command : byte[word [: Data]) : boolean;

The parameter **SlaveAdr** can be a byte variable or an 8-bit constant. According to the definition the 8th Bit of the address is the Read-Write bit and because of that it is overwritten or ignored by the function. This address is the physical module address of the I2C-Bus.



AVRco Standard Driver

The parameter **Command** is depending on the slave, but has to be specified. It could be a byte variable, an 8-bit constant or a word. Often this command is a module-intern address (e.g. EEPROM)

Bigger EEPROMs (>256 bytes) have a 16-bit address. Because it would not make sense to implement special library functions, the function **I2Cout** must be informed, if the address parameter (2nd parameter) consists of one or two bytes, so if it is 8- or 16-bits long. Because of this the parameter has to be definite and has to be declared that way.

```
{ eeprom 8kBytes 24C65 }
Var w : word;           { 16 bit adr }

w:= 0000;
b2:= 0;
bool:= I2Cout ($50, w);   { set adr for read = 0000 }
while b2 < $FF do         { read eeprom with adr auto incr}
    bool:= I2Cin ($50, b1); { result into b1 }
    inc (b2);
endwhile;
```

For components, which possess an additional intern address, for example EEPROMs and AD-converter, the required reading-address has to be put out by a write-access before reading. The next reading-access then goes to this address. If the module possesses an auto-increment of the address (ADC, EEPROM), it can be read continuous, without an output of a new address.

```
{ eeprom 256*8bit PCF8582E }
Var b2 : byte;           { 8 bit adr }

B2:= 00;
bool:= I2Cout ($53, b2);  { set adr for read = 00 }
while b2 < $FF do         { read eeprom with adr auto incr}
    bool:= I2Cin ($53, b1); { result into b1 }
    inc (b2);
endwhile;
```

The variable **Data** is optional and can be an 8- or 16-bit constant or any variable or constant. This parameter is optional and can, but must not exist. The slave and/or the required slave-action determines the type or omitting of data. The SizeOf(Data) or the number of the memory locations, which are occupied by this variable or constant, determine the number of bytes, which are read out of the variable within the block and written into the slave. With a byte, char or boolean it is 1 byte, with a word or an integer it is 2 bytes, whereby the lower byte is written before the high-byte.

If the result is a type of array, so the length of the array is established and a corresponding number of bytes is written, starting with the lowest address (lowest index) of the array.

If the Data is a type of string, the length of the string is established and a corresponding number of bytes is written, starting with the index string[0]. That means, that a string is always read and written incl. length byte. Within writing, the length byte determines the number of the bytes, which have to be written:

Count := length(string).

If the parameter data was not specified, only the command byte is transferred.

Attention:

By choosing the data variable/constant pay attention, that the slave is able to handle a corresponding length of a blocktransfer. An I2C module, which only consists of an 8-bit parallel port, is generally able to transfer one byte.

3.29.4 The I2C-BUS Interface

The I2C interface is an international standardized, synchrone serial interface for the connection of periphery-modules to a processor as well as for the communication of processors. This implementation mainly supports the used master-slave configuration, the CPU is the master and the connected module is the slave.

Several slaves can be connected, whereby every slave possesses a special intern address, which should appear only once in the so called net. Theoretically more than 100 modules can be prosecuted, so you could talk about a network. The variant with 10-bit addresses is not supported.

The connection with each other is made by two control lines: Data and Clock. These both lines are bi-directional, so they work in both directions. Because of that the designated port and/or the used pins have to be switchable into the respective data direction or it must possess a TRI-State function. Each of the both lines must have external pull-up resistors of 10..50kOhm to +5Volt.

The I2C-interface, which is imported by I2Cport and defined by I2clk and I2Cdat offers the three high-level functions I2Cstat, I2Cinp and I2Cout. These functions are generalized, i.e. they are not depending on a special chip. The programmer can and/or has to observe the software protocol of the slave, by a corresponding use and parameters, whereby the telegram protocol (Hardware-Address, Clock-Generation etc) is realized by the library driver.

All functions detect if the slave exists and is ready or if it not exists or is BUSY. If the function failed, a FALSE is returned, otherwise a TRUE.

Both transfer-functions offer the possibility of reading and writing. The type of the parameter data automatically determines the block length. Further information see I2Cinp and I2Cout.

3.29.5 Multi-Processing and I2C

In an application with Processes and/or Tasks in many cases the I2C-Bus is not only used as a network but also for other purposes (LCD, Ports etc). If so, then different processes access the I2C, and severe conflicts are inevitable because such sequential drivers (I2C, TWI, UART etc) are not re-entrant, i.e. they are not interruptible and cannot re-entered. Because of this the I2C port works with a semaphore of the type DeviceLock.

I2C_DevLock : DEVICELOCK;

The I2C driver observes and controls this semaphore. On entry into the driver the semaphore is checked as to whether the driver is locked or not. If it is free the semaphore becomes activated (driver locked) and the job will be executed. After finishing the job the semaphore is released (unlocked).

If the driver is locked (occupied) at entry time then a *Schedule* is executed and the calling Process is put into the Schedulers queue. With one of the next few schedulings this process is restarted and again checks this semaphore. This is repeated until the semaphore becomes free (unlocked).

Attention:

Because of the suspension of a I2C access through "Schedule" *Tasks should not* use I2C accesses. A schedule completely aborts and exits a Task and so a job will not be executed if a locked semaphore is found.

It is possible to set a flag to determine if the Task solved the job successfully. If the flag is not set the Task must again be initiated to repeat this job. But this is somewhat complicated and should be avoided.



AVRco Standard Driver

```
Import    SysTick, I2Cport;
Define    ProcClock = 4000000;    {4Mhz clock }
           SysTick    = 10;        {10msec Tick}
           2Cport     = PortB;     {use port $05}
           I2Cclk     = 0;        (clock-pin = port B bit 0}
           I2Cdat     = 3;        (data-pin  = port B bit 3}

Var       ar1       : array[1..4] of byte;
           b          : byte;
           bool       : boolean;

{adr 0 adjust to read}
if not I2Cout (%01010000, $0) then {Eeprom select, intern adr 0}
  Error;
else {read}
  bool:= I2Cinp (%01010000, b);    {Eeprom select, read adr 0 -> b}
endif;

{read a block, Eeprom decrements autom. Adr.}
bool:= I2Cinp (%01010000, ar1);   {Eeprom select, read adr 1..4 -> ar1}

{Eeprom write operation}
if not I2Cout ($50, $0, b) then   {Eeprom select, intern adr 0, Byte b prog}
  Error;
else {write block}
  bool:= I2Cout ($50, 1, ar1);    {Eeprom select, ar1 -> Eeprom-adr 1..4 prog}
endif;

{wait for ok examples}
while not I2Cstat(...) do
  ...;
  ...;
endwhile;

repeat
  ...;
  ...;
until I2Cstat(...);
```

3.30 I2Cexpand Driver for up to 8 bidirectional Ports

Basics

With some control applications and with a specific CPU used the useable IO-pins are not sufficient especially if two ports must be used for external RAM.

So a larger CPU (pin count) must be used or the current port pins must be expanded with additional hardware. There are several possibilities to implement such an expansion. Real IO-chips like the 8255 often can't be used because they need a parallel connection (at least 10bits) to the CPU and there are only 20 additional pins. Standard latches must also be connected in parallel and therefore also need many additional Pins of the CPU so the pin saving is less than expected.

If the IO-speed doesn't matter too much but the count of additional pins is high, the simple way is to implement external ports based on the I2C bus system which solve the problem very well.

There are many freely programmable remote-IO port chips available for the I2C bus (TWI). The Philips chips PCA9554 or PCA9554A are suitable very well. These contain a PORT-register, a PIN-register and a DDR-register. So they are absolute equivalent to an AVR port. Up to 8 devices can be connected to the bus.

Introduction to the I2Cexpander

This implementation either uses the software I2C-driver (I2Cport) or the internal TWI (I2C) port of the AVR mega CPUs. To use the I2Cexpander either the driver **I2Cport** or the driver **TWImaster** or the driver **TWInet** must be imported, the latter in Mastermode. With **XMegas** one of these TWIs must be used/imported: TWI_C, TWI_D, TWI_E or TWI_F.

As the I2C port-expander one PCA9554A from Philips must be used for one port. This chip can be present up to 8 times on the I2C bus. The PCA9554A can run up to 400kBit/sec on the I2C Bus. In contrary to its predecessors the 8bit port can be directly read, written or reprogrammed without any influence to the other parts of the chip.

The base address of the PCA9554A is \$38..\$3F. The PCA9554A can be replaced by the PCA9554 which base address is \$20..\$27. But these addresses are reserved for 16bit I2C-devices of type PCA9555 in the AVRco system. So the PCA9554 should only be used in special cases.

The possible ports are named PORT0...PORT7, PIN0...PIN7, DDR0...DDR7 where PORT0/PIN0/DDR0 has the I2C-address \$38, PORT1/PIN1/DDR1 has the I2C address \$39 etc.

As a specialty these I2C chips can invert the polarity of the input pins. To handle this the special ports INP_POL0..INP_POL7 are exported. A log1 inverts the corresponding input bit.

3.30.1 Technical Data

I2C Port	Software I2C imported by I2Cport	
or	CPU-TWI imported by TWImaster	// Mega
or	CPU-TWI imported by TWInet in Mastermode	// Mega
or	CPU-TWI imported by TWI_C, TWI_D, TWI_E oder TWI_F	// XMega

Hardware	I2C I/O-Expander chip PCA9554A from Philips, 1 piece per port
----------	---

I2C addresses	The PCA955A reside at the bus-addresses \$38..\$3F where PORT0 has the address \$38, PORT1 has \$39 etc.
---------------	---

The PCA9554 reside at the bus-addresses \$20..\$27 where PORT0 has the address \$20, PORT1 has \$21 etc.

The PCA9555 have three address pins or bits which must be wired in a correct way.



AVRco Standard Driver

Imports

As usual with the AVRco system the driver must be imported and defined. In addition the desired I2C/TWI driver must be imported and defined.

The SysTick is not used.

Import *I2Cport, I2Cexpand;*

or

Import *TWImaster, I2Cexpand;*

or

Import *TWInet, I2Cexpand;* // use Master mode

XMega

Import *TWI_C, I2Cexpand;* // use TWI_C

Defines

Dependent of the I2C or TWIport, this must be defined.

Example for *I2Cport*:

```
Define ProcClock    = 8000000;      {8Mhz clock }
        I2Cport      = PortC;       {port used}
        I2Cdat       = 7;           {bit7-PortC}
        I2Cclk       = 6, 4;        {bit6-PortC, optional delay 4}
        I2Cexpand    = I2C_Soft, $38; {use Software I2Cport, 9554A}
        I2CexpPorts  = Port0, Port4; {use Port0 and Port4}
```

Example for *TWImaster*:

```
Define ProcClock    = 8000000;      {8Mhz clock }
        TWIpresc     = TWI_BR100;    {100kBit/sec alt. TWI_BR400}
        I2Cexpand    = I2C_TWI, $38; {use TWIport, 9554A}
        I2CexpPorts  = Port1, Port2; {use Port1 and Port2}
```

Example for *TWInetMaster*:

```
Define ProcClock    = 8000000;      {8Mhz clock }
        TWInode      = 05;           {default address in slave mode}
        TWIpresc     = TWI_BR400;    {400kBit/sec alt. TWI_BR100}
        TWIframe     = 4, iData;     {buffer/packet size}
        TWIframeBC   = 6;            {option broadcast buffer/packet size}
        TWInetMode   = Master;
        I2Cexpand    = I2C_TWI, $20; {use TWIport, 9554}
        I2CexpPorts  = Port7;        {use Port7}
```

Example for *XMega*:

Import *TWI_C, I2Cexpand;* // use TWI_C, TWI_D, TWI_E or TWI_F

Define

```
OSctype    = int32MHz, PLLmul=4, prescB=1, prescC=1;
TWIpresc   = TWI_BR100;    {100kBit/sec alt. TWI_BR400}
I2Cexpand  = TWI_C, $38;   {use TWIportC, 9554A}
I2CexpPorts = Port1, Port2; {use Port1 and Port2}
```

I2Cexpand

Requests the I2C-port to use, either software-I2C with **I2C_Soft** or onchip TWIport with **I2C_TWI**.

The required basic driver must also be imported and defined.

The second parameter defines the I2C-chip: \$20 for the PCA9554 or \$38 for the PCA9554A.

I2CexpPorts

Defines which and how much ports must be supported. Valid defines are: Port0 ... Port7.

3.30.2 Types and Functions

The import of I2Cexpand exports a special type:

Type *TI2CPORT* = *internal*;

This type can be used to rename a ports and give it a more relevant name. Example:

Var *myName[@Port1]* : *TI2Cport*;

Now *myName* can be used to access the Port1.

I2CexpStat

At power-up it makes sense to check the ports for a correct connection and working. To do this, the I2C-state of a port can be checked.

Function *I2CexpStat (Port: TI2Cport)* : *boolean*;

This function returns a true if the select of the PCA9554 chip was successful.

With the programs context such a port can be handled in the same way as a normal port of the AVR. But there are some restrictions. The port can not be addressed by pointers. It can not be a part of a construct like an array or record. It can not be procedure-local and also can not be passed as a parameter to procedures/functions. The only special operation is *inc(port)* and *dec(port)*.

The bit by bit addressing of the port bits is possible. Then the bits must be overlaid over a port.

Var *iBit1[@Port2, 1]* : *bit*;

Now single bits can be addressed and manipulated:

```
if not iBit1 then  
  iBit1 := true;  
  iBit1 := false;  
  toggle (iBit1);  
endif;
```

The same or similar is also true with the standard AVR ports.

3.30.3 Multi-Processing and TWI Port

In an application with Processes and/or Tasks in many cases the TWI-Bus is not only used as a network but also for other purposes (LCD, Ports etc). If then different processes access the TWI heavy conflicts are build in because such sequential drivers (I2C, TWI, UART etc) are not re-entrant, i.e. they are not interruptible and can not re-entered. Because of this the TWI port works with a semaphore of the type DeviceLock.

TWI_DevLock : *DEVICELOCK*;
*TWI_DevLock**TN* : *DEVICELOCK*; // *XMega TN* = *C, D, E* or *F*

The TWI driver observes and controls this semaphore. At the entry into the driver the semaphore is checked whether the driver is locked or not. If it is free the semaphore becomes activated (driver locked) and the job will be executed. After finishing the job the semaphore becomes released (unlocked).

AVRco Standard Driver

If the driver is locked (occupied) at entry time then a **Schedule** is executed and the calling Process is put into the Schedulers queue. With one of the next few schedulings this Process is restarted and again checks this semaphore. This is repeated until the semaphore becomes free (unlocked).

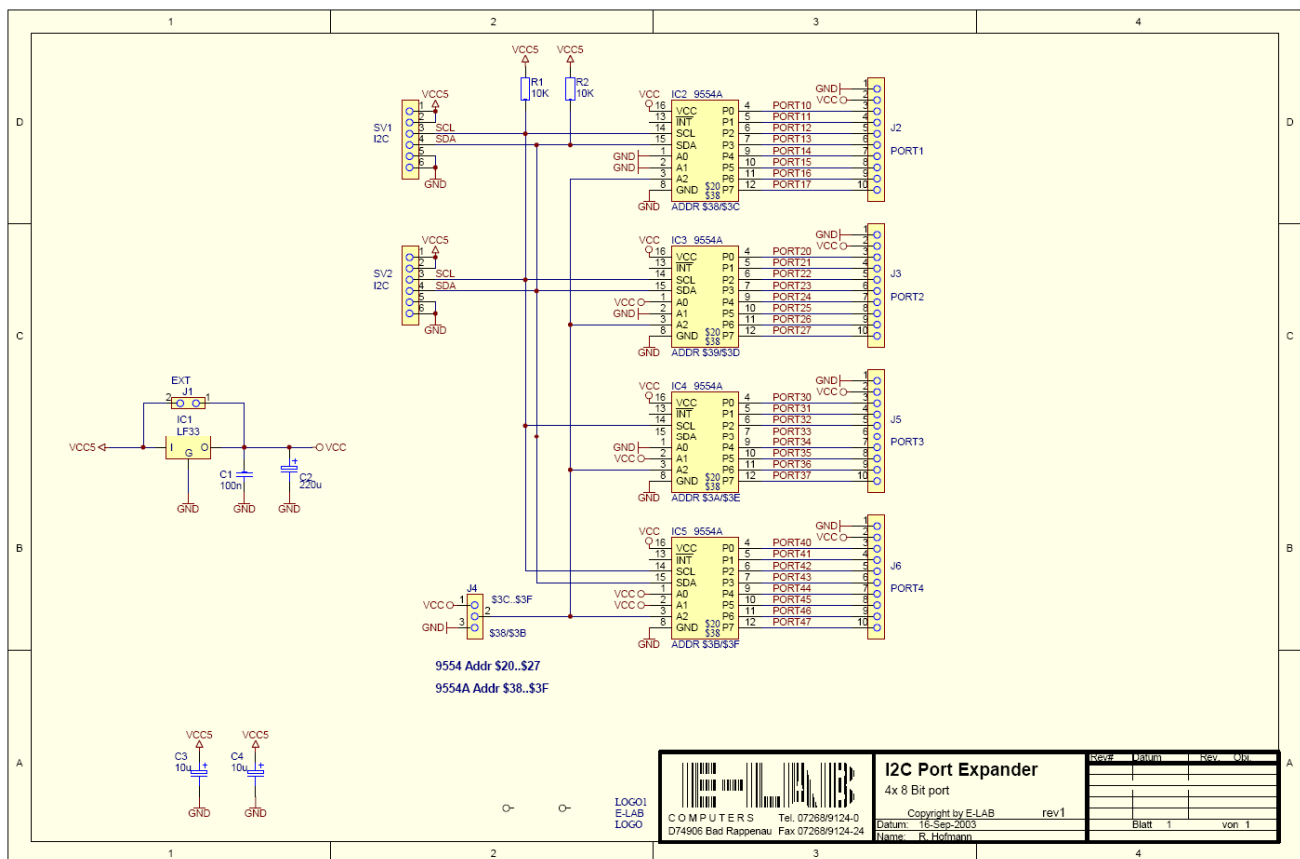
Attention:

Because of the cancellation of a TWI access through "Schedule" **Tasks should not** use TWI accesses. A schedule completely aborts and exits a Task and so a job will not be executed if a locked semaphore is found. It is possible to set a flag if the Task executed the job successfully. If the flag is not set the Task must again be initiated to repeat this job. But this is somewhat complicated and should be avoided.

Example program and schematic:

an example is in the directory `..\E-Lab\AVRco\Demos\I2Cexpand`

an X Mega example is in the directory `..\E-Lab\AVRco\Demos\XMega_I2Cexpand`



schematic I2Cexpand

3.31 Pulse Counter driver PulseCount

Not for XMegas

Simple and event/pulse counter can be build by continuously watching of a port pin which can be done in a task for example. If these events follow with short time delays so it makes sense to use an interrupt pin and then increment a variable with each interrupt. But then with high speed pulses the system is heavily loaded with interrupts and with interrupts also disabled for a short time pulses maybe lost.

In general but also especially with high speed pulses it is better to use an internal 16bit counter for this job. The advantage hereby is that these counters running completely self contained and the system is only minimal burdened with interrupts. Most of the 16bit counters provide an external clock input pin which must be used to connect the external pulse source. The application the must take care that this pin is always set as an input pin. For a correct function of the driver the global interrupt must be enabled.

The counter increments with each rising edge of the clock input pin. If an overflow of the 16bit counter occurs the counter wraps to 0 and rises an overflow interrupt. In this interrupt service then a 16bit variable is incremented by 1. Reading the PulseCount result then returns a longword (32bit) which consists of the counter content and the variable. Upto two counter channels can be imported and defined.

Imports

As usual with AVRco the driver must be imported.

Import SysTick, PulseCount, { PulseCount2} ..;

The necessary 16bit timer/counter must be defined:

Defines

Define

```
ProcClock   = 8000000;           {Hertz}
SysTick      = 10;                {msec}
StackSize    = $0010, iData;
FrameSize    = $0010, iData;
PulseCount   = Timer1;            {or Timer3..5 if present}
// PulseCount2 = Timer3;          {or Timer3..5 if present}
```

3.31.1 Functions

The driver provides 4 functions per channel:

Function GetPulseCount : longword;

Function GetPulseCount2 : longword;

This function returns the actual internal counter value. The internal value is not changed and the counter is not stopped. Continuously polling this function should be avoided because of the inherited temporary interrupt disable in this function which can result in a slowdown of the entire interrupt system.

Procedure PulseCountStart;

Procedure PulseCountStart2;

This procedure starts the counter. This is necessary after a program Start, a Stop or Clear.

Procedure PulseCountStop;

Procedure PulseCountStop2;

This procedure stops the counter without changing the internal counter.

Procedure PulseCountClear;

Procedure PulseCountClear2;

This procedure stops the counter and clears it to 0.



AVRco Standard Driver

3.32 Pulse Counter driver PulseCount XMega

Simple and event/pulse counter can be build by the use of an internal 16bit counter. These counters are running completely self contained and the system is only minimal burdened with interrupts. All of the 16bit counters provide an external clock input pins which must be used to connect the external pulse source. The application the must take care that this pin is always set as an input pin. For a correct function of the driver the global interrupt must be enabled.

The counter increments with each rising or falling edge of the clock input pin. If an overflow of the 16bit counter occurs the counter wraps to 0 and rises an overflow interrupt. In this interrupt service then a 16bit variable is incremented by 1. Reading the PulseCount result then returns a longword (32bit) which consists of the counter content and the variable. Upto 8 counter channels (C0..F1) can be imported and defined.

Imports

Import SysTick, PulseCount_C0, { PulseCount_C1, PulseCount_D0} ..; // C0..F1 counters

Defines

Define

```
// The XMegas don't provide any Oscillator fuses.
// So the application must setup the desired values
// possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

//>> CPU      = 32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCtype        = int32MHz,
PLLmul         = 4,
prescB         = 1,
prescC         = 1;
SysTick        = 10;           {msec}
StackSize      = $0060, iData;
FrameSize      = $0080, iData;
PulseCount_C0  = PortA, 0, PullUp; // clock input Port, Pin, Pullup/PullDown/None
PulseCountEv_C0 = 0;           // event channel to be used
```

Each of the Ports A..F and every Pin of these Ports can be used. One of the 8 Event Channels must be used.

3.32.1 Functions

The driver provides 4 functions per channel where **XX** stands for the timer/channel used, C0..F1:

Function PulseCountRead_XX : longword;

This function returns the actual internal counter value. The internal value is not changed and the counter is not stopped.

Procedure PulseCountStart_XX;

This procedure starts the counter. This is necessary after a program Start, a Stop or Clear.

Procedure PulseCountStop_XX;

This procedure stops the counter without changing the internal counter.

Procedure PulseCountClear_XX;

This procedure stops the counter and clears it to 0.

Example program:

A sample program is in the directory ..\E-Lab\AVRco\Demos\XMega_PulseCount

3.33 Incremental Encoder Driver IncrPort

Not for XMegas, use QDEC instead

Positions, movements or rotation directions are often controlled by so called Incremental Encoders. These are mechanical or optoelectronic devices, which supply 2 or 3 digital signals shifted by 90(120)degrees. With the help of these it's possible to find out the rotation direction or the direction of a move, and by counting the edges of all signals a relative position can be found out. If there is a Null-Point signal it is possible to count in absolute positions.

This implementation needs one (two) XOR-gates, which sum all signals and feed one input of the **Analog Comparator** of the AVR. The other input of the ACOMP is fixed to 50% VCC. This circuit allows each edge of all signals to generate an interrupt which implements four times higher resolution of the input (Quadrature). This signal establishes a fast pulse counting, but direction information is not build in.

The direction of the sensor can be found out by checking the phase relation of the input signals. All lines must be connected to any **input port** of the AVR. The lines must be connected to the same port in a consecutive order. The port pins are arbitrary, but must be useable as inputs.

Imports

The driver must be imported as usual with AVRco.

Import *SysTick, IncrPort, ..;*

The resolution (16 or 32bits, 2 or 3 phases), the desired port and the two (three) portpins must be defined:

Defines

Define

```
ProcClock   = 8000000;           {Hertz}
SysTick     = 10;                 {msec}
StackSize   = $0010, iData;
FrameSize   = $0010, iData;
IncrCounter = 16, 2;              {16 bit integer, 2 phases}
IncrPort    = PinD, $C0;          {PinD, Portpin 6 + 7}
```

3.33.1 Functions

There are 4 different functions exported:

Function *GetIncrementVal : integer [longint];*

This function returns the actual internal counter value as an integer if a 16bit counter is defined. The result is a longint if a 32bit counter is defined. The internal value is not changed with a read out.

Function *GetIncrementRel : integer [longint];*

This function returns the actual internal relative counter value as an integer if a 16bit counter is defined. The result is a longint if a 32bit counter is defined. The returned value is relative to the last call of this function. If there was no count pulse since the last call a zero is returned. If this function is called with fixed intervals it is possible to use the value to get a speed or rotate information.

Procedure *ClearIncrementVal;*

This procedure resets the absolute and relative internal counter.

Procedure *SetIncrementVal (val : integer [longint]);*

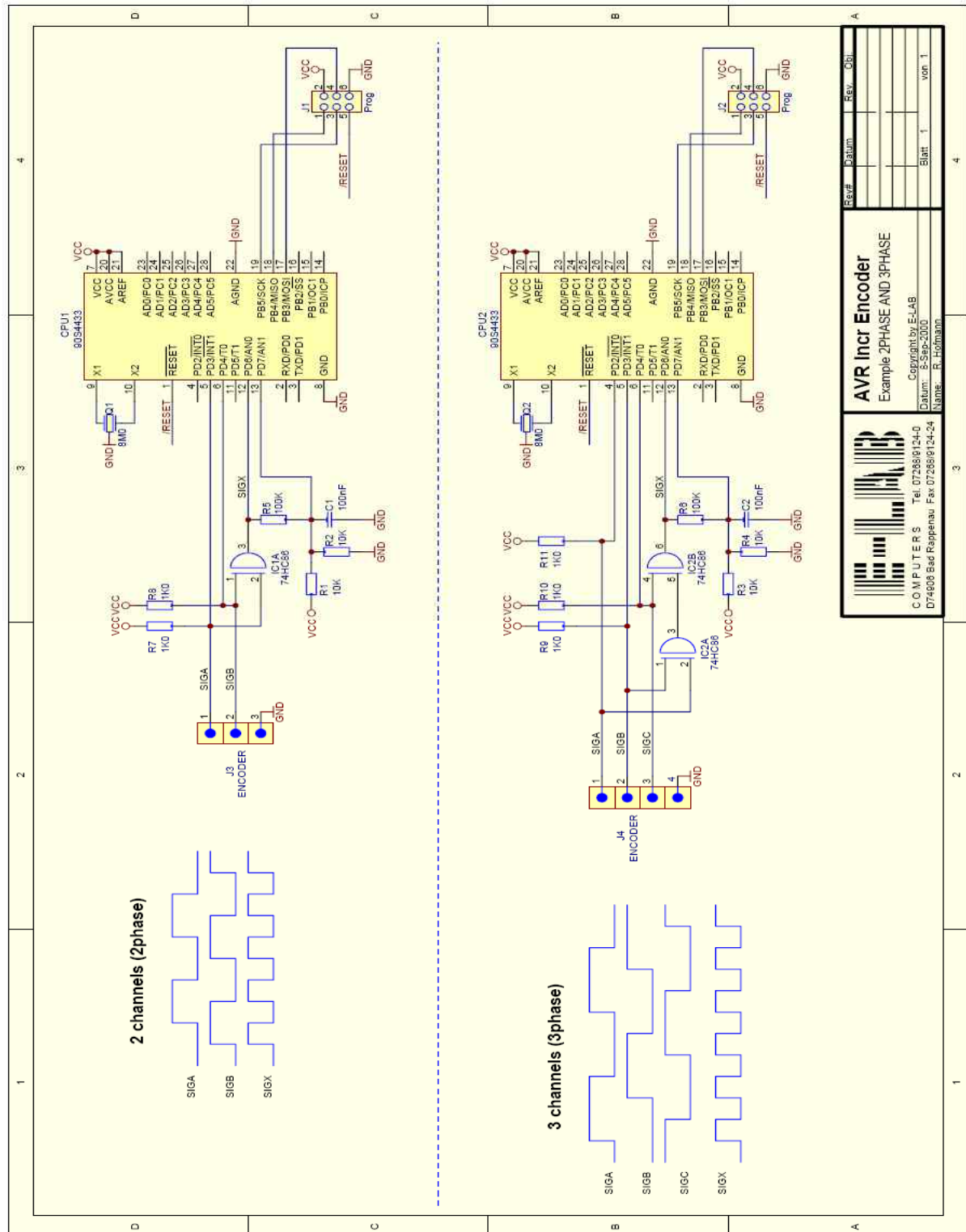
This procedure resets the absolute internal counter to "val".

Remarks:

This driver is interrupt controlled. If the general interrupt is disabled even for a short time incorrect results are possible.

Example program and schematic:

an example is in the directory `..\\E-Lab\\AVRco\\Demos\\Increment`



schematic **Increment**

3.34 Incremental Encoder Driver IncrPort4

The previously introduced driver uses the Analogue Comparator and therefore only supports one encoder. A more common implementation is the *IncrPort4* driver which supports up to 4 encoder channels. It uses a timer and an 8bit port. External logic is not necessary, but there is no zero position recognition.

This implementation uses any input port which is able to connect up to 4 of 2-phase encoders. All encoders must be connected to the same port. Encoder-0 is connected to portx.0 and portx.1, encoder-1 to portx.2 and portx.3 etc. The portpins used must be set as inputs. The four-folding of the input lines (quadrature) is always active.

Unused port pins can be used in any way.

The driver uses the cyclical timer interrupt for channel scanning and analysing. In order to get stable results the scan rate should be at least 4 times the max. expected pulse rate of the encoders. On the other hand a scan rate much higher than necessary wastes processing power and can slow down the system dramatically. Also a continuous polling of the channels by the application can result to problems with the interrupt system because each read access implies a short interrupt disable.

Imports

The driver must be imported as usual with AVRco.

Import SysTick, IncrPort4, ..;

The resolution (16 or 32bit), the desired port (PIN), the number of encoders, the time and the scan rate have to be defined:

Defines

Define

```
ProcClock   = 16000000;           {Hertz}
StackSize   = $0020, iData;
FrameSize   = $0040, iData;
IncrPort4    = PinA, 2, 32;       // pin-reg used, channels, 16 or 32bit integer
IncrScan4    = Timer3, 10;        // timer used, scan rate 10kHz (1..100)
```

3.34.1 Functions

There are 7 different functions exported. The parameter *chan* counts from 0 (0..3).

Procedure IncrCount4start;

After a Reset or PowerOn the Scan-Timer does not run. It must be started with this procedure. This procedure does not change any counter states.

Procedure IncrCount4stop;

The Scan-Timer can be stopped and no interrupts will be generated by the timer. The counter states are not changed.

Function GetIncrVal4 (*chan* : byte) : integer [longint];

This function returns the actual internal counter value of *chan* as an integer if a 16bit counter is defined. The result is a longint if a 32bit counter is defined. The internal value is not changed by a read out.



AVRco Standard Driver

Function *GetIncrRel4 (chan : byte) : integer [longint];*

This function returns the actual internal relative counter value of *chan* as an integer if a 16bit counter is defined. The result is a longint if a 32bit counter is defined. The returned value is relative to the last call of this function. If there was no count pulse since the last call a zero is returned. If this function is called with fixed intervals it is possible to use the value to get a speed of rotate information.

Procedure *ClearIncrVal4 (chan : byte);*

This procedure resets the absolute and relative internal counters of *chan*.

Procedure *ClearIncrAll4;*

This procedure resets all absolute and relative internal counters.

Procedure *SetIncrVal4 (chan : byte; val : integer [longint]);*

This procedure resets the absolute internal counter at *chan* to "val".

Remarks:

This driver is interrupt controlled. If the general interrupt is disabled too long faulty results are possible.

If the parameter *chan* is passed by the functions it is always compared to the channel count defined with *IncrPort4*. If this value is out of range the function does nothing and returns a zero if necessary. The channel parameter *chan* counts from zero.

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\Increment4`

3.35 UP/DOWN Counter Driver XMega

A basic method to sample mechanical position is a simple Up-Down counter. In contrast to the increment counter there is a pulse input and a direction input.

This implementation uses one of the upto 8 XMega Timers. The two necessary inputs (clock, dir) can be in any port (PortA..PortF). With a resolution of 16bit (Integer) only one of the 8 Event Channels are used. With 32bit (LongInt) an additional interrupt is used. The input lines must be connected to the same port. The port pins can be arbitrary.

Imports

The driver must be imported as usual with AVRco.

```
Import SysTick, UPDOWN_C0, ..;      // UPDOWN_C1, UPDOWN_D0, UPDOWN_D1, UPDOWN_E0
```

Where C0, C1, D0, D1 etc. select the desired Timer. The “XX” stands for the selected timer.
The resolution (16 or 32bits), the desired port, the two portpins and the event channel must be defined:

Defines

Define

```
// The XMegas don't provide any Oscillator fuses.  
// So the application must setup the desired values  
// possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz  
  
//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz  
OSCtype      = int32MHz,  
              PLLmul=4,  
              prescB=1,  
              prescC=1;  
  
SysTick      = 10, adj;      // msec, correct the RTC32K timer for exact mSec timing  
StackSize    = $0064, iData;  
FrameSize    = $0064, iData;  
  
UPDwnClkDir_C0 = PortD, 0, 1;  // Port, clock, dir input pin  
UpDwnRes_C0    = 16;          // Resolution UPDOWN_C0, 16 or 32  
UpDwnEvChan_C0 = 0;          // 0..6
```

In the following context the Defines, Functions and Procedures are marked with XX. The XX stands for the UpDwn UpDwn_C0..QDEC_F1) and for the Timer channel (C0..F1).

```
UPDwnClkDir_XX  defines the Port to use (PortA..PortF) and its two Pins.  
UpDwnRes_XX     defines the resolution, 16 or 32 bits.  
UpDwnEvChan_XX  defines the necessary Event Channel, 0..6.  
                Two consecutive channels are used.
```



AVRco Standard Driver

3.35.1 Functions

There are 4 different functions exported:

Procedure *UPDWNenable_XX(ena : boolean);*

This procedure starts or stops the internal counter. The actual counter value is not changed.

Function *UPDWNgetPos_XX : integer [longint];*

This function returns the actual internal counter value as an integer if a 16bit counter is defined. The result is a longint if a 32bit counter is defined. The internal value is not changed with a read out.

Procedure *UPDWNclearpos_XX;*

This procedure resets the internal counter. The counter is stopped and must be re-enabled.

Procedure *UPDWNsetPos_XX(val : integer [longint]);*

This procedure resets the absolute internal counter to "val". The counter is stopped and must be re-enabled.

The direction and the change speed can be found out by reading the current counter value and comparing it with the last one. This must be done in a fixed time interval.

Remarks:

This driver is interrupt controlled when a 32bit resolution is used. If the general interrupt is disabled for a time then incorrect results are possible.

Example program:

A sample program is in the directory `..\E-Lab\AVRco\Demos\XMega_UpDownCount`

3.36 QDEC Incremental Encoder Driver XMega

Positions, movements or rotation directions are often controlled by so called Incremental Encoders. These are mechanical or optoelectronic devices, which supply 2 digital signals shifted by 90degrees. With the help of these it's possible to find out the rotation direction or the direction of a move, and by counting the edges of all signals a relative position can be found out.

This implementation uses one (two) of the upto 8 XMega Timers. The two necessary Phase inputs can be in any port (PortA..PortF). With a resolution of 16bit (Integer) only two of the 8 Event Channels are used. With 32bit (LongInt) one additional timer and two additional event channels are used. The driver works in quadrature mode. The lines must be connected to the same port in a successive order. The port pins are arbitrary.

Imports

The driver must be imported as usual with AVRco.

Import SysTick, QDEC_C0, QDEC32_F1, ...; // QDEC_C1, QDEC_D0, QDEC_D1, QDEC_E0

Where C0, C1, D0, D1 etc. select the desired Timer. The import QDEC_xx or QDEC32_xx defines the resolution, 16 or 32 bits.

The desired port, the two portpins and the event channels must be defined:

Defines

Define

```
// The XMegas don't provide any Oscillator fuses.
// So the application must setup the desired values
// possible OSC types: extXTAL, extClock, ext32kHz, int32Khz, int2MHz, int32MHz

//>> CPU=32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCType      = int32MHz,
              PLLmul=4,
              prescB=1,
              prescC=1;

SysTick      = 10;           // msec
StackSize    = $0064, iData;
FrameSize    = $0064, iData;

// 16 bit resolution
QDECphase_C0 = PortD, 0, 1;   // Port, Phase0, Phase90 input pin
QDECevChan_C0 = 0;           // 0, 2, 4 or 6

// 32 bit resolution
QDECphase_F1 = PortE, 4, 5;   // Port, Phase0, Phase90 input pin
QDECevChan_F1 = 2;           // event channel 0, 2, 4 or 6
QDECcntHi_F1 = TCC1, 5, 6;    // Timer, two event channels
```

In the following context the Defines, Functions and Procedures are marked with **XX**. The **XX** stands for the QDEC (QDEC_C0..QDEC_F1) and for the Timer channel (C0..F1).

QDECphase_XX defines the Port to use (PortA..PortF) and its two Pins.
QDECevChan_XX defines the necessary Event Channel, 0, 2, 4 or 6.
QDECcntHi_XX 32bits, defines the necessary additional timer and two event channels



AVRco Standard Driver

3.36.1 Functions

There are 4 different functions exported:

Procedure *QDECenable_XX(ena : boolean);*

This procedure starts or stops the internal counter. The actual counter value is not changed.

Function *QDECgetPos_XX : integer [longint];*

This function returns the actual internal counter value as an integer if a 16bit counter is defined. The result is a longint if a 32bit counter is defined. The internal value is not changed with a read out.

Procedure *QDECclearpos_XX;*

This procedure resets the internal counter.

Procedure *QDECsetPos_XX(val : integer [longint]);*

This procedure resets the absolute internal counter to "val".

The direction and the change speed can be found out by reading the current counter value and comparing it with the last one. This must be done in a fixed time interval.

Remarks:

The event channels of the basic timer must start at even addresses. Two channels are used. The event channels of the additional (32bit) timer should be in a consecutive order.

Example program:

A sample program is in the directory `..\\E-Lab\\AVRco\\Demos\\XMega_QDEC`

3.37 Stepper-Motor

3.37.1 Fundamentals

Using stepper motors one can easily design precise and fast positioning and transport actuators. Of all of this kind of motor systems the stepper motor system is the cheapest type. However this drive also has a significant disadvantage: stepper motors are synchron motors and these have the problem of falling out of synchronization, they stop or, which is also a dangerous behavior, they lose steps. Under practical conditions one must observe the start step rate, the acceleration and the maximum step rate. These parameters depend on the motor type, the operation mode (full/halfstep etc), the power output stage, the inertia of the system and in general on the load and friction losses.

Only stepper motors with 2 coils are supported (2-phase motors).

The developer must know the advantages and disadvantages of the various power driver types and also the operation modes of the stepper motor. Tutorials etc. about these topics can be downloaded from the homepages of Ericsson and ST. It cannot be the target of this manual to make a stepper motor workshop, but here are some explanations:

3.37.1.1 ConstantVoltage Operation

This is the simplest way and can be implemented with 4 transistors. But the motor must be suitable; that means coil resistance > 10 ohms. Only full or halfstep is possible as the operation mode. The start step rate is relatively low and also the possible max. step rate. Motor and system resonances are a big problem.

3.37.1.2 ConstantCurrent Operation

This mode is costly, but has the advantage that the higher the step resolution is (e.g. sine current mode with "StepMicro8"), the lower is the risk of getting into resonance problems. In addition the motor runs smoothly like a DC-motor, including at low step rates.

A disadvantage is that both the software and the hardware driver must work with a 6..8fold speed (StepMini6, StepMicro8). In addition the necessary acceleration table in the ROM must also be prepared to include the much higher step rate.

The result is that the lookup table size instead of 1..2kByte (Full/Halfstep) now increases to 8..10kByte. At a step rate of 10Khz the used 16bit timer runs with a 100usec interrupt. With an AVR at 8Mhz there is a computing time of appr. 30usec to get the next step. This is ca 30% of the whole time. Also other interrupts are disabled during this time. If there are other interrupts implemented, the interrupt time of them must be small to avoid a frequency -jitter and consequent step losses.

3.37.1.3 Limitations

From the above explanations one can see that with the lookup table size in ROM and the time consumption of the interrupt the CPU can't be burdened with large additional jobs, **because:** The higher the step resolution, the higher the necessary step rate, the larger the lookup table and the higher the computing time consumption.

With the AVR the CPU should run with 8MHz in order to obtain the parameter values and/or step rates in reality. A fixed internal resolution is obtained by a fixed prescaler (div 8) for the used **Timer1**, which is the base for all other internal calculations.

3.37.2 Acceleration Ramp

In order to have no step losses or to get out of synchronization, a stepper motor must be accelerated and decelerated linearly (Hz/sec). This ramp can be calculated in realtime or readout from a lookup table. A realtime calculation is only possible with a CPU power $\gg 20\text{Mips}$. The used lookup table has the disadvantage of much higher ROM consumption, but is absolutely necessary if there a random acceleration and steprates and the CPU is not a Highend part.

A standard lookup table normally starts at 100Hz (**StepMinFreq**) because this value can be achieved with most operation modes and motors. The end of the table (**StepMaxFreq**) is mostly determined by the operation mode of the motor. With full- or halfsteps 1kHz is possible. With mini- or microsteps mode the max. steprate can reach 10kHz.

Both values define the size of the table and also the **possible** start- and max steprate with ramping.

3.37.3 Drive Modes

The drive mode (**StepType**) defines operation mode of the motor (Full, Half, Micro Step etc) and also the construction of the stepper power stage. There are 8 possible combinations:

Name	Driver	Steps		Port pins	Ics
StepFull 2	StepF2	Full Step		2	PBL3717..3777, L6219
StepFull 4	StepF4	Full Step		4	L6203, L6204
StepHalf 4	StepH4	Half Step		4	PBL3717..3777, L6219
StepHalf 6	StepH6	Half Step		6	L6203, L6204
StepMini 4	StepM4	Quart Step	Trap ez	6	L6219, TEA3738
StepMini 6	StepM6	Hexa Step	Trap ez	6	L6219, TEA3738
StepMicro 2	StepM2	8 Micro Steps	Sinus	2	A3955, IMT901
StepMicro 8	StepM8	8 Micro Steps	Sinus	8	PBL3717..3777, L6219, TLE5250
StepS4	StepS4	32 Micro Steps	Sinus	10	LMD 18245

More about trapezoidal-mode in Siemens datasheet TCA3727

More about sinusoidal-mode in Allegro datasheet A3955

StepFull 2

Fullstep operation with 2 phase outputs

StepFull 4

Fullstep operation with 4 phase outputs

StepHalf 4

Halfstep operation with 2 phase outputs and 2 Enable (low active!)

StepHalf 6

Halfstep operation with 4 phase outputs and 2 Enable (high active!)

StepMini 4

Quarterstep operation (trapez) with 2 phase outputs and 2*2 binary value outputs

StepMini 6

Hexastep operation (trapez) with 2 phase outputs and 2*2 binary value outputs

StepMicro 2

One pulse- and 1 direction output for sine wave operation (8 Microsteps) and/or intelligent powerstage

StepMicro 8

Sine operation (8 Microsteps) with 2 phase outputs and 2*3 binary value outputs

StepS4

Sine operation (32 Microsteps)

3.37.4 Import of the Stepper

In order to use the stepper driver it must be imported within the common import clause (Import **StepPort**;). Because the driver does internal calculations using LongWord, and also the number of steps to move is also given as LongWord, the type LongWord must be imported (From System Import **LongWord**;).

The output **port** to use must be declared in the Define-Block. For building the acceleration lookup tables the parameter **StepMinFreq** and **StepMaxFreq** must be defined. The desired soft driver also must be declared with **StepType**.

```
Device = 90S8515;
```

```
Import StepPort;
```

```
From System Import longword;
```

Define

```
ProcClock      = 8000000;      {Hertz}  
StackSize      = $0030, iData;  
FrameSize      = $0010, iData;  
StepPort       = PortA;  
StepMinFreq    = 100;  
StepMaxFreq    = 5000;  
StepType       = StepM6;
```

XMega

With the XMegas Timer_C0...Timer_F1 must be defined, if present.

```
StepTimer      = Timer_C0;      // use Timer_C0
```

3.37.5 Parameters of the Stepper

For the acceleration and deacceleration ramps at runtime there several parameter needed. These are variables which are published automatically if a StepPort is imported.

Warning: a parameter should only be changed if the driver is inactive! !

```
Var StepStartFreq : word;
```

Designates the start steprate at ramp start and the steprate which is achieved if the ramp down has ended. The value of this variable should not be lower than the value of "StepMinFreq" and also should be lower than "StepEndFreq".

```
Var StepEndFreq : word;
```



AVRco Standard Driver

Designates the end step rate where the ramp/acceleration should end and the step rate, at which the deceleration starts. The value of this variable should be smaller than "StepMaxFreq" and also should be larger than "StepStartFreq".

Var StepAccValue : word;

Designates the acceleration in Hz/sec. Realistic values reside between 1000 and 10000.

In order to make a definite number of steps with the help of acceleration and deceleration ramps (positioning), the additional parameter

Var StepCount : LongWord; {max 2³² Steps}

is needed. StepCount must be loaded with the desired step count before starting the move. StepCount is the essential parameter for positioning. After the move StepCount contains the value 0.

If the time is the limit of the move (StepRampCW or StepRampCCW) StepCount contains the number of steps done with this command.

Type TStepMode = (StepStop, StepUp, StepRun, StepDown);

Var StepMode : TStepMode;

The variable StepMode always contains the actual state of the driver. This var can be changed by the application, but there is a risk of some strange behaviour if you do this, so please read only! If the driver is idle "StepMode" contains the value "StepStop". If accelerating StepMode shows "StepUp". Whilst ramp-down the value is "StepDown". In the possible linear phase (constant StepRate) the value "StepRun" is exhibited.

Because of this the application can increase the current of the motor (boost) while acceleration or deceleration is in progress.

Because a parameter only should be changed in idle mode it is a **must** to check the driver state by reading the variable **StepMode** and test for **StepStop** before changing any var or inserting a new command. Normally the parameters are set once at startup and remain unchanged. An exception is the var StepCount, which must be always setup before a destination move command.

3.37.6 Commands/Procedures of the Stepper

The stepper driver knows three logical operation modes with the accomplished commands and/or procedures. These are **SingleStep**, **Ramp** and **Positioning**.

Important: a procedure should only be called in idle mode (StepMode = StepStop) with the exception of the command "StepRampStop" in the Ramp mode.

StepperOn

Switches the power stage active outputs the last step combination at the output port. This command can be invoked before each action/command, but is not a must.

StepperOn; {switch power stage active}

StepperOff

Switches the power stage inactive or removes the current from motor. This command can be invoked after each action to avoid overheating of the power stage or motor. If a reduction of the motor current is possible this should be preferred over "StepperOff".

StepperOff; {Switch power stage inactive}

StepOneCW

Step one step clock wise. No parameters are needed or changed. StepCount is not changed.

StepOneCW; {one step CW}

StepOneCCW

Step one step counter clock wise. No parameters are needed or changed. StepCount is not changed.

StepOneCCW; {one step back}

StepRampCW

Step up ramp clockwise. The parameters StepStartFreq, StepEndFreq, and StepAccValue are needed but not changed. At first StepCount is reset to 0 then each step is counted. The acceleration ramp is processed until the maximum step rate (StepEndFreq) is exceeded or the command "StepRampStop" is given. In the first case the driver proceeds stepping with End frequency until the command "StepRampStop" is invoked, then a ramp-down follows until StepStartFreq is reached and the motor is stopped. If the command "StepRampStop" is invoked whilst ramping up the deacceleration is started immediately. A StepRampXX command always needs a StepRampStop command

StepRampCW; {step until stop command}

StepRampCCW

Step up ramp counter clockwise. The parameters StepStartFreq, StepEndFreq, and StepAccValue are needed but not changed. At first StepCount is reset to 0 then each step is counted. The acceleration ramp is processed until the maximum step rate (StepEndFreq) is exceeded or the command "StepRampStop" is given. In the first case the driver proceeds stepping with End frequency until the command "StepRampStop" is invoked, then a ramp-down follows until StepStartFreq is reached and the motor is stopped. If the command "StepRampStop" is invoked whilst ramping up the deacceleration is started immediately. A StepRampXX command always needs a StepRampStop command

StepRampCCW; {step until stop command}

StepRampStop

If a StepRamp command is in progress the command switches to ramp down mode. If the command "StepRampStop" is invoked whilst ramping up the de-acceleration is started immediately. A StepRampXX command always needs a StepRampStop command

StepRampStop; {ramp down}

StepDestCW

Step up ramp clock wise, step a definite amount of steps deaccelerate and stop. The parameters StepStartFreq, StepEndFreq, and StepAccValue are used but not changed. StepCount must be loaded with the desired step count and will be decremented at each step. The acceleration ramp is processed until the maximum Step rate (StepEndFreq) is reached or the half amount of steps is done, whatever comes first. If the half step amount is reached before the maximal speed is reached the driver immediately switches to ramp down until StepCount is zero. In case speed maximum is reached first the driver remains on max speed until the remaining steps to do are satisfactory to do a secure ramp down until StepCount is zero.

StepDestCW; {step an amount of steps}

StepDestCCW

Step up ramp counter clock wise, step a definite amount of steps deaccelerate and stop. The parameters StepStartFreq, StepEndFreq, and StepAccValue are used but not changed. StepCount must be loaded with the desired step count and will be decremented at each step. The acceleration ramp is processed until the maximum Step rate (StepEndFreq) is reached or the half amount of steps is done, whatever comes first. If the half step amount is reached before the maximal speed is reached the driver immediately switches to ramp down until StepCount is zero. In the case where speed maximum is reached first the driver remains on max speed until the remaining steps to do are satisfactory to do a secure ramp down until StepCount is zero.

StepDestCCW; {step an amount of steps}



AVRco Standard Driver

3.38 Stepper-Motor in UserMode

Fundamentals

The standard stepper motor driver generates up- and down ramps and places step and phase information onto the chosen port of the CPU. These signals serve for direct controlling the H-bridges, and possibly the additional DA-converters for micro steps.

This kind of operation is not suitable for smart power drivers which often need a special command protocol. Because of this the driver is enhanced by the *UserPort* mode. In this mode there is no phase or step information and also a physical port for output is not necessary. Only the ramp generating and processing and the step counting is implemented.

The communication with the application is executed by a special Call-Back function of type *UserDevice*. This function must be provided by the application. With each step of the driver this function is called. Inside of the function the application must build the phases and eventually the DAC signals and must pass them to the power driver. Now it's possible to handle a stepper motor controller such as Trinamic TMC239 with its SPI interface.

Internally within the driver the forward/backward steps are added/subtracted correctly.

Imports

In order to use the stepper driver it must be imported within the common import clause **Import StepPort**. Because the driver calculates internally with LongWord, and also the number of steps to move is given as LongWord, the type LongWord must be imported (From System Import **LongWord**);).

Because there is no real port for the driver the output port must not be declared in the Define-Block. For building the acceleration lookup tables the parameter **StepMinFreq** and **StepMaxFreq** must be defined. The desired soft driver also must be declared with **StepType**.

Device = mega16;

Import StepPort;

From System Import longword;

Define

```
ProcClock    = 16000000;    {Hertz}
StackSize    = $0050, iData;
FrameSize    = $0020, iData;
StepMinFreq  = 100;
StepMaxFreq  = 10000;
StepType     = UserPort;
```

If a driver chip needs only Step and Dir lines then the *StepperIOS* function can completely be removed if the two necessary pins are defined in the Define section.

Define

```
...
StepType     = UserPort;
StepPort     = PortE.0, PortE.1;    // Clock, Dir, no StepperIOS
```

An example for this specific mode can be found in the Demos directory in *XMEGA_StepperU*.

3.38.1 Parameters of the Stepper

For the acceleration and deacceleration ramps at runtime there several parameter needed. These are variables which are published automatically if a StepPort is imported. Important: a parameter should only be changed if the driver is inactive!!

Var StepStartFreq : word;

Designates the start steprate at ramp start and the steprate that is achieved if the ramp down has ended. The value of this variable should not be lower than the value of "StepMinFreq" and also should be lower than "StepEndFreq".

Var StepEndFreq : word;

Designates the end steprate where the ramp/acceleration should end and the steprate at which the deacceleration starts. The value of this variable should be smaller as "StepMaxFreq" and also should be larger than "StepStartFreq".

Var StepAccValue : word;

Designates the acceleration in Hz/sec. Realistic values are between 1000 and 10000.

In order to make a definite number of steps with the help of acceleration and deacceleration ramps (positioning), the additional parameter

Var StepCount : longword; {max 2^32 Steps}

is needed. StepCount must be loaded with the desired step count before starting the move. StepCount is the essential parameter for positioning. After the move StepCount contains the value 0. If only the time is the limit of the move (StepRampCW or StepRampCCW) StepCount contains the number of steps done with this command.

Type TStepMode = (StepStop, StepUp, StepRun, StepDown, RampDown);

Var StepMode : TStepMode;

The variable StepMode always contains the actual state of the driver. This var can be changed by the application, but some strange behavior can result from this, so please read only!

If the driver is idle "StepMode" contains the value "StepStop". If accelerating StepMode shows "StepUp". Whilst ramp-down the value is "StepDown". In the possible linear phase (constant StepRate) the value "StepRun" is exhibited. Because of this the application can increase the current of the motor (boost) while acceleration or deacceleration is in progress.

Because a parameter only should be changed in idle mode it is imperative to check the driver state by reading the variable **StepMode** and test for **StepStop** before **changing** any var or inserting a new command. Normally the parameters are set once at startup and remain unchanged. An exception is the var StepCount, which must always be setup before a destination move command.

With the UserMode selected the application must provide the CallBack function *StepperIOS*. This function must build the whole of the IOs and the phases.

UserDevice StepperIOS (cw : boolean);

begin

doTheJob;

end;

The parameter CW defines the current rotation direction, clockwise or counterclockwise.



AVRco Standard Driver

3.38.2 Commands/Procedures of the Stepper

With the **UserMode** the stepper driver knows two logical operation modes with the accomplished commands and/or procedures. These are **Ramp** and **Positioning**.

Attention:

a procedure should only called in idle mode (StepMode = StepStop)!! With the exception of the command "StepRampStop" and "StepVelocity" in the Rampmode.

StepperOn
StepperOff

These functions above make no sense in **UserMode** and are not implemented.

StepRampCW

Step up ramp clock wise. The parameters StepStartFreq, StepEndFreq, and StepAccValue are needed but not changed. At first StepCount is reset to 0 then each step is counted. The acceleration ramp is processed until the maximum step rate (StepEndFreq) is exceeded or the command "StepRampStop" is given. In the first case the driver proceeds stepping with End frequency until the command "StepRampStop" is invoked, then a ramp-down follows until StepStartFreq is reached and the motor is stopped. If the command "StepRampStop" is invoked whilst ramping up the deacceleration is started immediately. A StepRampXX command always needs a StepRampStop command

StepRampCW; *{step until stop command}*

StepRampCCW

Step up ramp counter clock wise. The parameters StepStartFreq, StepEndFreq, and StepAccValue are needed but not changed. At first StepCount is reset to 0 then each step is counted. The acceleration ramp is processed until the maximum step rate (StepEndFreq) is exceeded or the command "StepRampStop" is given. In the first case the driver proceeds stepping with End frequency until the command "StepRampStop" is invoked, then a ramp-down follows until StepStartFreq is reached and the motor is stopped. If the command "StepRampStop" is invoked whilst ramping up the deacceleration is started immediately. A StepRampXX command always needs a StepRampStop command

StepRampCCW; *{step until stop command}*

StepRampStop

If a StepRamp command is in progress the command switches to ramp down mode. If the command "StepRampStop" is invoked whilst ramping up the de-acceleration is started immediately. A StepRampXX command always needs a StepRampStop command

StepRampStop; *{ramp down}*

StepVelocity(v : word) : boolean;

With ramp mode this function can be called everytime. The parameter V then temporary replaces the value of StepEndFreq. If the system is in ramp-up or ramp-down state normally there is no immediate influence of the behaviour. If the system is in constant velocity so this will be changed. If necessary an additional ramp-up or ramp-down is executed.

The value of StepEndFreq will not be changed. The value of "v" must not be smaller than StepMinFreq

StepVelocity (1000);

StepDestCW

Step up ramp clock wise, step a definite amount of steps deaccelerate and stop. The parameters StepStartFreq, StepEndFreq, and StepAccValue are used but not changed. StepCount must be loaded with the desired step count and will be decremented at each step. The acceleration ramp is processed until the maximum Step rate (StepEndFreq) is reached or the half amount of steps is done, whatever comes first.

AVRco Standard Driver



If the half step amount is reached before the maximal speed is reached the driver immediately switches to ramp down until StepCount is zero. In case speed maximum is reached first the driver remains on max speed until the remaining steps to do are satisfactory to do a secure ramp down until StepCount is zero.

StepDestCW; {step an amount of steps}

StepDestCCW

Step up ramp counter clock wise, step a definite amount of steps deaccelerate and stop. The parameters StepStartFreq, StepEndFreq, and StepAccValue are used but not changed. StepCount must be loaded with the desired step count and will be decremented at each step. The acceleration ramp is processed until the maximum Steprate (StepEndFreq) is reached or the half amount of steps is done, whatever comes first. If the half step amount is reached before the maximal speed is reached the driver immediately switches to ramp down until StepCount is zero. In case speed maximum is reached first the driver remains on max speed until the remaining steps to do are satisfactory to do a secure ramp down until StepCount is zero.

StepDestCCW; {step an amount of steps}

StepPanicStop

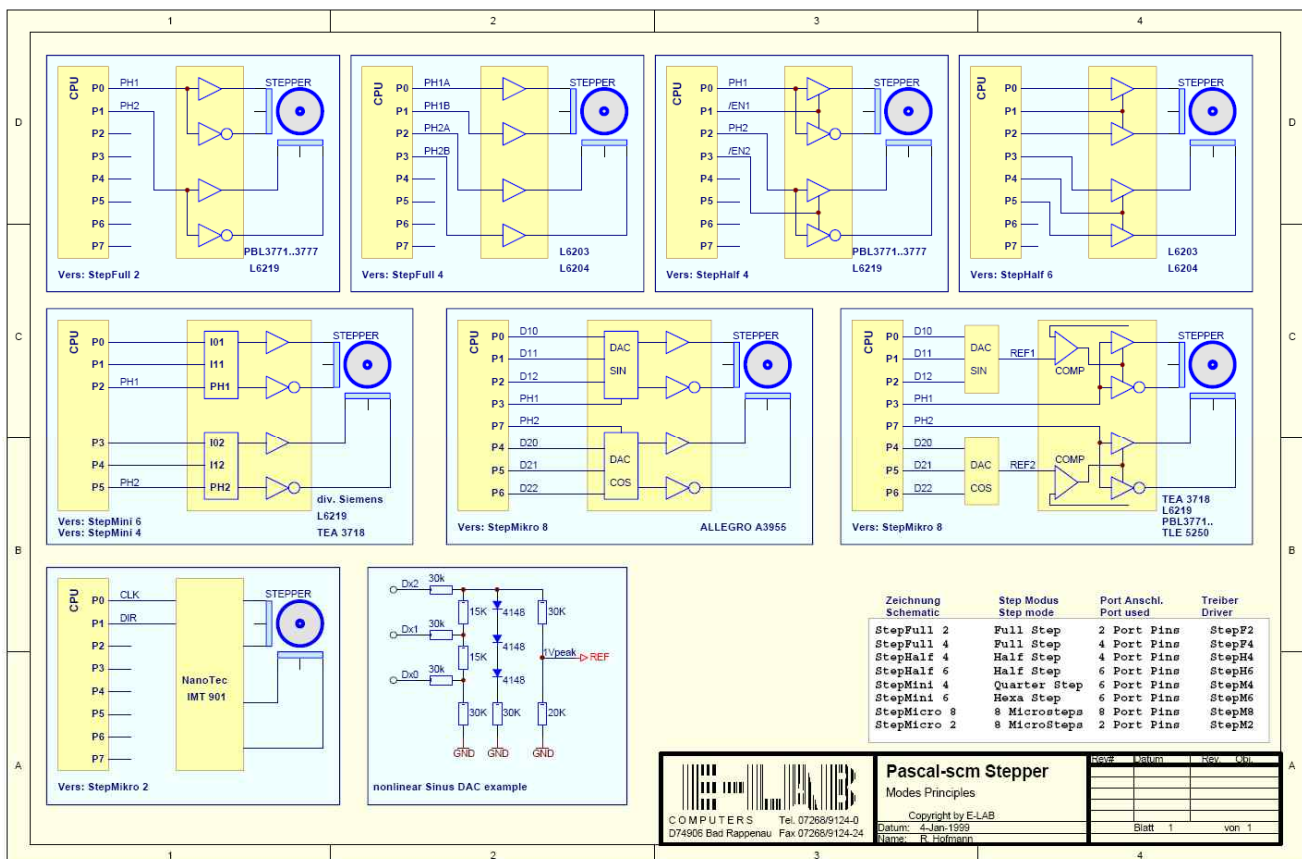
Immediately stops a ramp or destination move. In most cases steps are lost by this hard stop so "StepCount" always contains 0.

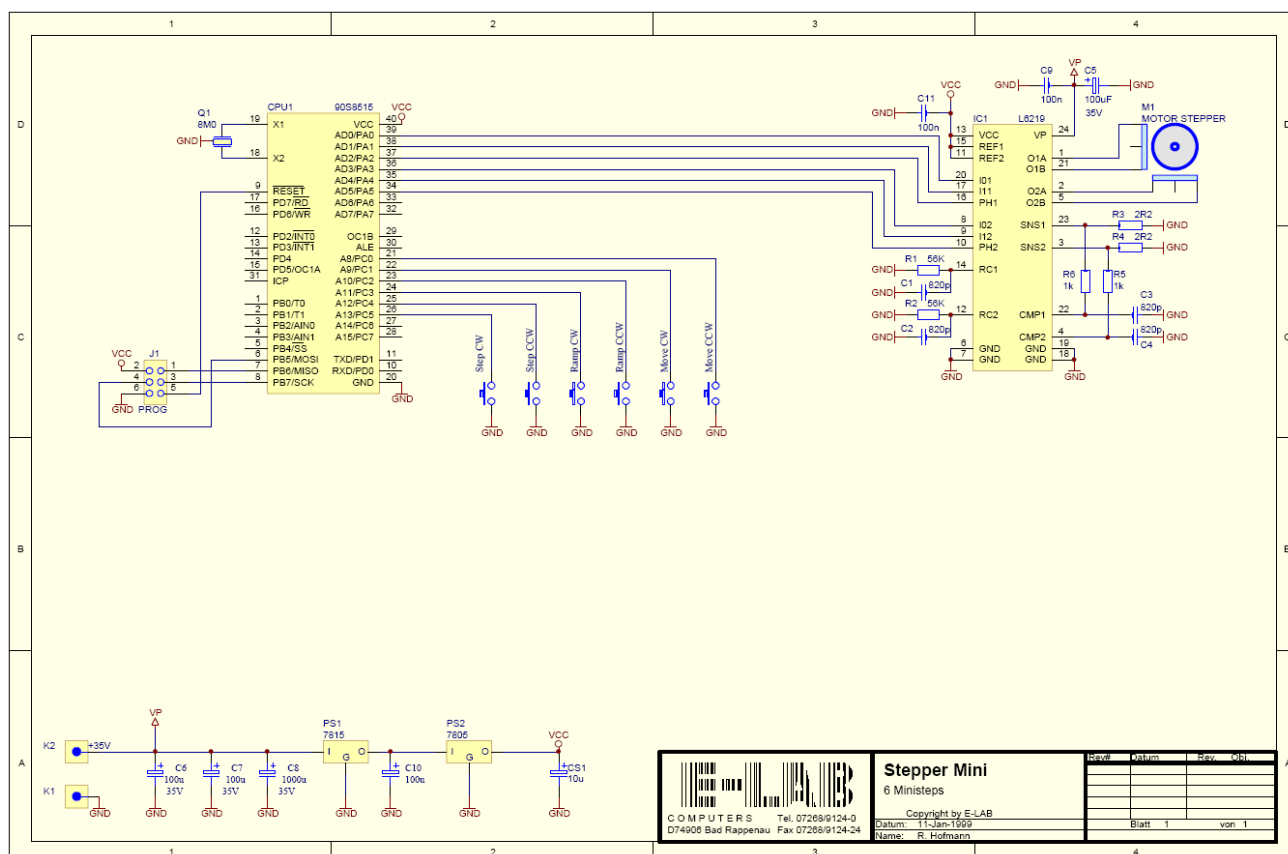
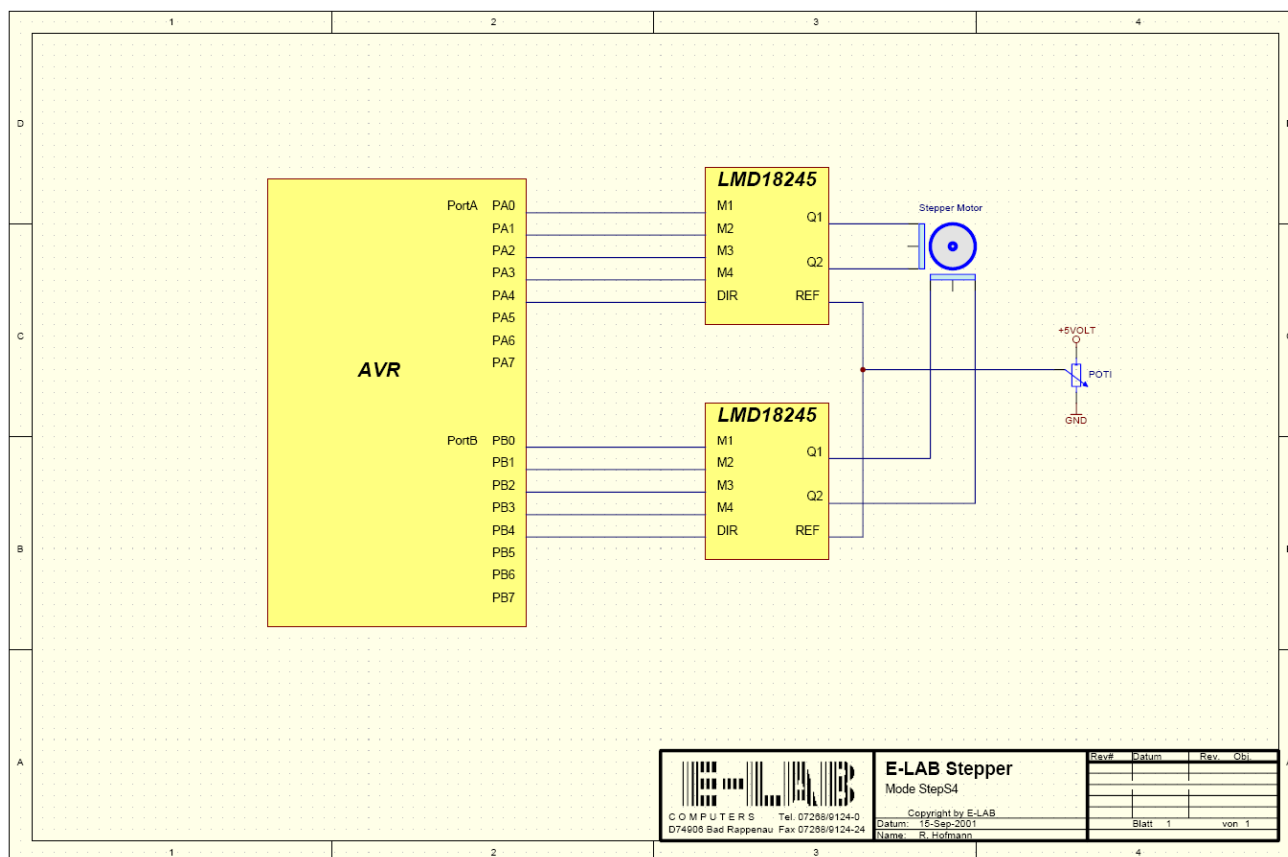
StepperSema

The Stepper driver can handle a semaphore. If used it must be imported with:

From StepPort Import StepperSema;

If the motor stops after a function StepDest or StepRampStop this semaphore is set to 1. So processes or tasks can use WaitSema to wait until the motor has stopped.





3.38.3 StepperIOS

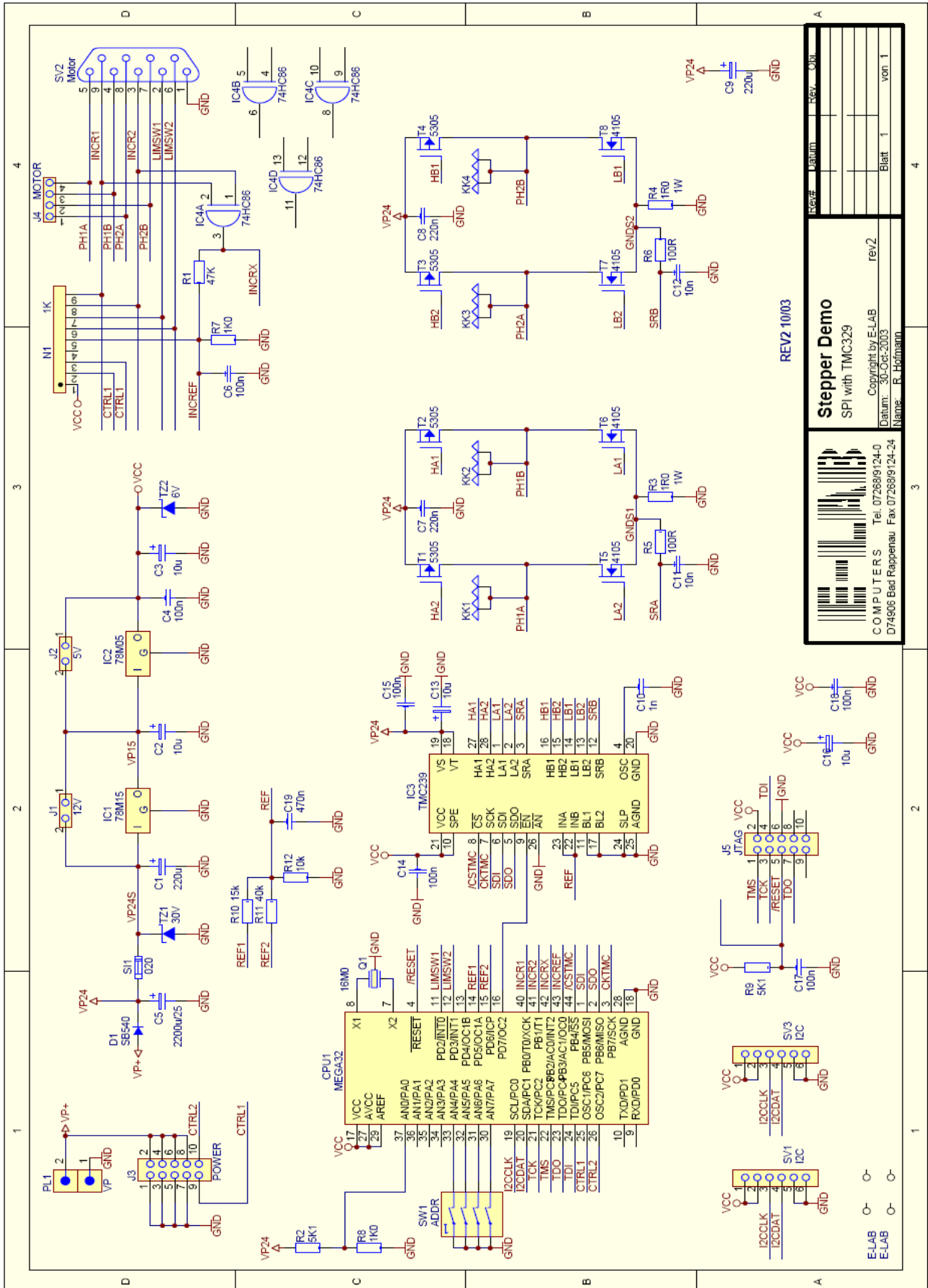
The UserDevice StepperIOS is called out of the Timer Interrupt. Because of this used registers must be saved and restored. The system supports the application by always saving the following registers which then can be used by the device driver function without any limitations:

_ACCGLO	R0
_ACCGHI	R1
_ACCB	R16
_ACCA	R17
_ACCALO	R18
_ACCAHI	R19
_ACCDLO	R20
_ACCDHI	R21
_ACCELO	R22
_ACCEHI	R23
_ACCFLO	R24
_ACCFHI	R25
_ACCBLO	R26
_ACCBHI	R27
_ACCCLO	R30
_ACCCHI	R31

Example program and schematic:

an example is in the directory `..\E-Lab\AVRco\Demos\StepperDemo`

an **XMega** example is in the directory `..\E-Lab\AVRco\Demos\XMega_Stepper`



schematic **StepDemo** with Trinamic Chip TMC239

3.39 Servo Driver for up to 8 digital Servos

Basics

With small robots, manipulators and handling systems the simple but effective positioning jobs must be solved. Here the well known model (aircraft, vessel or car) servos are preferred. These are available with a torque up to 100Ncm and more.

Equipped with ball bearings and metal gears they are very robust and precise, but the main advantage of them is the price.

In most cases the servos are controlled by pulses where the pulse duration contains the desired position. Over the years a standard has been established:

Cycle time	: 20msec
neutral pulse width	: 1.5msec
max positive position	: 2msec
max negative position	: 1msec
pulse polarity	: positive

With some exotic types the cycle time can be longer or shorter. The same is true for the neutral position.

The pulsewidth variation is the same with all types: $\pm 0.5\text{msec}$

The servos provide a trimming of the neutral position within limits. This must be done in the impulse generator (servo controller) which can change its variation for example from $1.5\text{msec} \pm 0.5\text{msec}$ to $1.6\text{msec} \pm 0.5\text{msec}$

The cycle time of 20msec and the maximal pulsewidth of 2msec ($\pm$ neutral offset) allow the control of up to 8 servos because with radio controls the pulses are sent sequentially and serially and an additional pause of a few msec must be inserted to allow the receiver to synchronize the packet.

Overview ServoPort

This implementation doesn't build a serial sequential pulse packet but uses a separate port pin for each servo channel. The pulses of the channels are staggered so simultaneous motor starts of the channels are avoided; otherwise high current peaks will be generated. The advantage of this implementation is that all pulses can be built with one hardware timer of the CPU. This timer then must be interrupt driven.

The driver runs transparently in the background. With two exported functions the servo position and offset of each channel can be controlled. The system calculates and scales the parameters and stores them in an array. The timer interrupt cyclically processes this array and generates the pulses.

The timer is completely reserved for the servo driver and cannot be used for other jobs. With the AVR only timer1 or timer3 (mega64, mega128) can be used.

Important:

Because the timer does its job in an interrupt each interrupt response delay results in a jitter of the generated Impulse that can possibly lead to a creak or purr of the servos.

The sources for such interrupt delays can be:

1. A disabled global interrupt by the application. This should be avoided or must not last more than a few usec.
2. System interrupts, UART, Timer etc. The system itself avoids long lasting interrupt procedures. But with the SystemTick with many jobs like SystemTimer, RTC, MultiTasking Scheduling etc. there can be fluctuations of 10..30usec. In some cases it's necessary to check whether the jitter is acceptable.
3. Application interrupts. Here the common principle must be borne in mind: the longer an interrupt service lasts the more other interrupts are delayed or in worst case will be skipped and get lost.



AVRco Standard Driver

3.39.1 Technical Data

Cycle time	20msec fixed
Neutral pulse time	adjustable from 1.0msec to 2.0msec
Pulse variation	+/- 0.5msec default, definable between +/-0.5 and 1.0msec
Neutral offset	+/- 30% of the pulse variation
Channels	1..8 chan0..chan7
Timer	Timer1 or Timer3 (XMegas Timer_C0...Timer_F1)

Imports

As usual with the AVRco system the driver must be imported and defined.

The SysTick is not used.

Import *ServoPort;*

Defines

The used CPU port must be specified and also the first useable bit of this port. The used memory area must be defined where the variables are located, how many channels must be used and their polarity. The neutral pulse width and the hardware timer must also be defined.

Define *ProcClock = 8000000; {8Mhz clock }*
ServoPort = PortB, 2, iData; {Port, Startbit in Port, data area}
ServoChans = 4, Positive; {4 channels, positive pulse}
ServoNeutral = 1.5, Timer1; {1.5msec neutral position, use Timer1}
ServoSwing = 1.0; {adjustable between 0.5 and 1.0msec, Resolution 100 pts}
 Optional resolution of 1000 points
ServoSwing = 1.0, 1000; {adjustable between 0.5 and 1.0msec, Resolution 1000 pts}

ServoPort

Defines the port to be used, the first useable bit of it and the memory area of the driver's variables. The port can be an output-only type. The startbit in the port must be set so that all desired channels fit into this port. The memory location defines the memory area where the working variables of the driver should be placed.

ServoChans

Defines the channel or servo count (1..8) and consequently defines the necessary internal memory usage. The pulse polarity must be defined with "Positive" or "Negative" common for all channels.

ServoNeutral

Defines the neutral pulsewidth (1.0 ... 2.0) common for all channels/servos. A 16bit timer is needed which with an AVR is usually Timer1. With new types (mega64, mega128) Timer3 is also useable. With XMegas Timer_C0...Timer_F1 must be used, if present.

ServoSwing

Optional, default 0.5msec. Defines the max. swing of all channels (+/-0.5 ... +/-1.0msec). As a additional option the servo resolution (precision) can be extended to 1000 points.

3.39.2 Functions

SetServoChan

Procedure to set the servo position. The parameter chan selects the channel (0..ServoChans-1). The parameter pulse defines the new position in % (-100...+100). For the 1000 resolution the pulse values can vary between -1000 and +1000. Illegal channel numbers are discarded. Pulse values exceeding the +/-100 (+/-1000) limit are clamped to +/-100 (+/-1000).

Procedure *SetServoChan (chan : byte; pulse : integer);*

AVRco Standard Driver



SetServoOffs

Changes the servo neutral position. The parameter chan selects the channel (0..ServoChans-1). The parameter offs defines the new neutral position in % (-30...+30). Illegal channel numbers are discarded. Offs values exceeding the +/-30 limit are clamped to +/-30

Procedure *SetServoOffs* (*chan* : *byte*; *offs* : *integer*);

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\Servo`

an **XMega** example is in the directory `..\E-Lab\AVRco\Demos\XMega_Servo`



AVRco Standard Driver

3.40 DCF-77 Decoder

Real Time Clocks become more and more important with medium and large systems. Because of this the AVRco library contains a RTC driver, which supports the RTC of several AVRs and which can also use the SysTick as its heartbeat. The internal RTC is somewhat more precise than the version with the SysTick. But in most cases the precision of the SysTick version is sufficient.

The precision is no problem. More difficult is the initialization of the RTC. After each Power-Up the RTC must be setup for correct time and date. As well as the additional hardware and software that is needed to setup the clock there is also the risk of a wrong or forgotten setup after a power up.

To avoid these problems and costs there are two solutions:

1. Use of an additional RTC that is battery backed. A setup of the RTC is only necessary once, but several port pins are needed (6..8), or with an I2C-chip an additional driver must be installed. Furthermore a long term deviation must be expected and also the leap years and winter/summer time changes must be handled manually.
2. Usage of a DCF77 receiver (Europe only). With DCF77 there are absolutely no setup or precision problems. The only disadvantage is the setup-time of up to 2 minutes after power up. Also the DCF77 receiver must be placed to a location with low radiation distortion to assure useable receive data. The DCF77 driver inserts itself into the SysTick.

If one has a choice the DCF77 clock should be preferred. With good receiving conditions an absolute error free and problem free operation can be expected. (DCF77 is only applicable in Europe)

The DCF77 clock is not a real-time-clock. It must always work in connection with another clock. The AVRco system provides three possibilities:

1. DCF77 controls the RTC of an AVR CPU. The RTC driver must be imported.
2. DCF77 controls the RTC which is clocked by the SysTick.
3. DCF77 controls an external clock chip

Items 1 and 2 are identical from the DCF77 driver's point of view. If an error free time has been received each minute the RTC will be adjusted. There is no user program intervention. The DCF77 driver recognizes the RTC automatically.

Item 3 (RTC Chip) is supported by a **CallBack** procedure. If this procedure is present, it is called at each full minute, provided that a complete telegram was received error free. The CallBack procedure, which must be provided by the programmer, now can synchronize the external chip by writing the proper memory content of the DCF77 into the chip. This procedure is called from within the SysTick, so please obey register savings and time consumption because the global interrupt is disabled for the whole procedure.

But the external chip is a special case. With most of the cases the RTC driver is precise enough to solve all possible jobs.

Imports

As usual with AVRco the driver must be imported.

Import SysTick, DCFclock, ...;

Defines

The port pin used as the DCF-input and also the memory area of the DCF registers must be defined:

Define {mega103}

```

ProcClock      = 8000000;           {Hertz}
SysTick        = 10 , Timer2;       {msec}
DCFclock       = iData;
DCFport        = PinD, 2, negative; {Port, bitnumber, polarity}
DCFfieldMode   = reset;             // decrement, optional "reset"
    
```

Important:

The SysTick must fulfill several conditions. The SysTick must not be lower than 1msec and not higher than 100msec, it must be an integer number and $1000 / \text{SysTick}$ must not have a remainder ($1000 \bmod \text{SysTick} = 0$).

3.40.1 DCF77-Functions/Procedures

Basic and implementation dependent procedures and functions.

Function DCFready : *boolean*;

The result of this function becomes true if the DCF77 driver received a complete error free telegram and stored it into the RTC. From this moment on one can assume that the RTC's data is absolutely correct. After power-up this process lasts at least 1 minute and with good receiving quality it lasts 2 minutes at maximum. Poor reception level delays the ready by up to several minutes.

Function DCFfield : *byte*;

The result of this function represents the quality of the receiving level. A zero equals no data, a 255 is optimal. It lasts some Minutes until this value has stabilized.

With each valid frame (1sec) this value is incremented by one. With an invalid frame there are two possible reactions:

If the Define **DCFfieldMode** is set to "decrement" each error decrements the value by one.

If the Define **DCFfieldMode** is set to "reset" each error resets the value to zero.

Function DCFDayLightSave : *boolean*;

Returns the current daylight saving. With summer time the result = true.

Procedure DCFupdate; // Callback Procedure

If the DCF77 driver can't locate the import of the RTC, it calls the procedure "DCFupdate" every minute, if this procedure is implemented. This procedure can control and update an external RTC chip. The memory locations/variables

DCF_SECOND, DCF_MINUTE, DCF_HOUR, DCF_DAY, DCF_MONTH, DCF_YEAR, DCF_WEEKDAY

can be used for this. Please note that this function is called from within the SysTick. Used register must be saved under some circumstances. It's a good idea to use assembler code for this purpose.



AVRco Standard Driver

3.40.2 Hardware

DCF77 receivers are available as small boards and also for wall mounting or racks. The wall is the best location because the place and the distance to a possible distortion source (PC etc) is very important. A distorted receiver can lead the many minutes until the DCF77 driver has synchronized itself. During the synchronization phase the connected RTC runs with invalid data.

A DCF77 receiver module sends a bit-serial telegram to the CPU via one wire. With the CPU a single input pin is used for the reception.

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\DCF77`

3.41 AVR Timer Low Level Driver TickTimer

Often a cyclical interrupt or pulse train is needed which must be able to be precisely setup in its timing. The System Timers are not suited for these purposes, especially if a very constant and precise timing is necessary. Then a hardware timer is a necessity.

The introduced driver uses one 16bit timer of the AVR CPU, either TIMER1 or timer3, 4, 5 if present. The selected timer then is not available for other jobs (Stepper etc). With both timers the COMPARE-MATCH-A function is used. It's also possible to use the 8bit Timer2 if present.

The timing can be set between 10usec to 4sec in 1usec increments. With the 8bit Timer2 the resolution strongly depends of the processor clock. As an option the driver can support an interrupt. As another option the output of the timer can be routed to the relevant Compare-Match pin.

By the import of the driver and its defines the setup of the timer is preset. Then there are several functions that can be used by the application. The tick time can be changed. The timer can be stopped and restarted. The optional output pin can be disabled or enabled.

Imports

As usual with AVRco the driver must be imported.

```
Import SysTick, TickTimer, ...; // Xmega TickTimer and/or TickTimer2
```

```
From System Import LongWord; // needed for calculations
```

Defines

The timer that shall be used by the driver must be defined

```
Define ProcClock = 16000000; {Hertz}
        SysTick    = 10;      {msec}
        StackSize  = $0030, iData;
        FrameSize  = $0030, iData;
        TickTimer  = Timer1;   // use Timer1.COMPA and no PortPin
        //TickTimer = Timer1, pinout; // use Timer1.COMPA and its PortPin
        //TickTimer = Timer2;   // use Timer2.COMPA and no PortPin
        //TickTimer = Timer2, pinout; // use Timer2.COMPA and its PortPin
        //TickTimer = Timer3;   // use Timer3.COMPA and no PortPin
        //TickTimer = Timer3, pinout; // use Timer3.COMPA and its PortPin
```

With the TickTimer define an 8/16bit timer is selected. With most of the AVR's there is at least Timer1 present. With newer and bigger types there also timer3 or the 8bit Timer2 or the two 16bit versions Timer4 and Timer5 which are also supported if present.

The optional Define extension **pinout** enables and connects the COMPA output pin of the timer and the timer toggles this pin at runtime. The function **TickTimerOutpEnable** is only available with this Define mode as is the variable **TickTimerPin**.

XMega

With the XMega's a Timer_C0...Timer_F1 must be defined, if present. In addition the desired Output Port and Pin can be defined. PinOut is not valid here.

```
TickTimer  = Timer_C0; // use Timer_C0 and no PortPin
TickTimer2 = Timer_C1, PortE, 5; // use Timer_C1 and PortE, pin 5
```

3.41.1 Variable

```
Var TickTimerPin : bit; { TickTimer2Pin}
```

This variable can be used to "manually" switch the level of the of the timer pin provided the timer is stopped. The variable is only available if **pinout** is defined.



AVRco Standard Driver

3.41.2 Functions

Function *TickTimerTime (time : longword) : boolean;* // *XMega also TickTimer2Time*
Function *TickTimerReload (time : longword) : boolean;* // *XMega also TickTimer2Reload*
Procedure *TickTimerRawVal (presc : byte; cmp : word|byte);* // *XMega also TickTimer2RawVal*
Procedure *TickTimerStart;* // *XMega also TickTimer2Start*
Procedure *TickTimerStop;* // *XMega also TickTimer2Stop*
Procedure *TickTimerOutpEnable (enable : boolean; Level : byte);* // *XMega also TickTimer2OutpEnable*

Function *TickTimerTime (time : longword) : boolean;* { *TickTimer2Time*}

This function stops a running timer and calculates the new setting given by the time argument. Then the values are written into the timer's prescaler and compare registers. The argument **time** defines the tick-time (pause between two interrupts or pin toggles) in usec. The minimum useable value of a tick is 10usec. The possible maximum value depends on the processor clock. With 16MHz this is about 4sec, with 8MHz it is about 8sec. With the 8bit Timer2 these lower limits must be increased in relation. The processor clock must not be below 1MHz. The result becomes false if the given time argument leads to illegal timer settings.

Function *TickTimerReload (time : longword) : boolean;* { *TickTimer2Reload*}

This function does not stop or restart the timer like the function above. The purpose of this function is the same as the above except that the timing can be changed without stopping or disturbing the Timer.

Procedure *TickTimerRawVal (presc : byte; cmp : word|byte);* { *TickTimer2RawVal*}

This is an alternative to the function *TickTimerTime*. Here the prescaler and the compare register can be written directly. The prescaler value must be in the range of 1..5 for the Timer1 and 3. For the Timer2 values between 1..7 are possible. The compare value is of type word for the two 16bit Timers and a byte for the 8bit Timer2.

Procedure *TickTimerStart;* { *TickTimer2Start*}

Starts the timer and its interrupt, if present. This function must be used after one of the above two first functions in order to restart the timer. Also after a program Start/Reset, when the timer is stopped.

Procedure *TickTimerStop;*

Stops the timer and its interrupt, if present. Restart the timer with *TickTimerStart*.

Procedure *TickTimerOutpEnable (enable : boolean; Level : byte);* { *TickTimer2OutpEnable*}

If the timer output is imported by the Define **pinout** the system provides this function. A timer stop or start has no influence on the state of the COMPA pin. In the stop-condition this pin retains its last state given by the timer. If the timer is started the pin toggles at its given rate.

With this function the output pin can either be connected to the timer with **enable = true** or disconnected from the timer with **enable = false**. In the last case the timer has no control of the pin but the pins stays in its last state. In order to have now a definite output level if the timer is disconnected, the desired pin state must be passed in the argument **level**.

With level <> 0 the pin then becomes a logical 1. If the pin is connected to the timer (enable = true) the argument level is meaningless.

3.41.3 Interrupt

The timer generates an interrupt if desired. In order to do so the Callback procedure must be implemented. This procedure then can serve for other additional jobs. Please note that this is a Callback out of an Interrupt.

Procedure *onTickTimer;* // *onTickTimer(SaveAllRegs);* // *onTickTimer2*
begin

...

end;

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\TickTimer`

an *XMega* example is in the directory `..\E-Lab\AVRco\Demos\XMega_TickTimer`

3.42 FreqCount Driver - Frequency Counter/Timer

Not for XMegas

In some applications a function is necessary which can measure or count an external frequency, or the ontime of a pulse must be found. For an experienced programmer this is no problem. Take a 16bit hardware counter, an appropriate time gate and the frequency counter or timer is ready. But often the little details cause some problems.

So it makes sense to implement such a function completely as a library driver. Then the programmer need not worry and can do his main work, building the application.

3.42.1 Overview Frequency Counter

This implementation uses an existing 16bit timer (Timer1 or Timer3 in the mega128) as an pulse counter. With the frequency counter the gate is handled by the SystemTick. This saves resources of the CPU. The timer gate can be set to 10sec, 1sec, 100msec and 10msec. The result of the measure cycle is a 16bit/32bit value.

The drawback of the SystemTick usage (Freq. counter) is the somewhat reduced precision of the result and some limitations with the choice of the SysTick value. The inherent precision is in the range of 0.1%..0.2% depended of the processor clock and the SysTick value. The higher the CPU clock and the lower the SysTick value the better the result (within some limits). An optimal result can be achieved with a CPU clock $\geq 8\text{MHz}$ and a SysTick of 2msec. If several interrupts are running there will be a jitter in the results.

Some limitations exist when the SysTick value has to be defined. Because of the gate times the following limitations apply:

100Hz area

resolution	0.01Hz
max frequency	655.35Hz = 0.65535kHz
gate time	100sec
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec

1kHz area

resolution	0.1Hz
max frequency	6553.5Hz = 6.5535kHz
gate time	10sec
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec

10kHz area

resolution	1Hz
max frequency	65535Hz = 65.535kHz
gate time	1sec
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec

100kHz area

resolution	10Hz
max frequency	655350Hz = 655.35kHz
gate time	100msec
SysTick	1msec, 2msec, 2.5msec, 4msec, 5msec, 10msec

1MHz area

resolution	100Hz
max frequency	6553500Hz = 6.5535MHz
gate time	10msec
SysTick	1msec, 2msec, 2.5msec, 5msec, 10msec

In the MHz area the maximum frequency is also limited by the processor clock. The external clock of a timer must not exceed $\text{ProcClock}/4$. With an 8MHz CPU clock this is then 2MHz.



AVRco Standard Driver

The clock or frequency signal that must be measured must be connected to the Clock input of the used timer. With Timer1 this is the PIN T1 and with the Timer3 this is PIN T3. The application must ensure that this port pin is programmed as an input.

3.42.2 Overview Pulse Timer

The implementation of the pulse timer uses the same 16bit Timer as the frequency counter part. The SysTick is not used and because of this there is no SysTick restriction with the Timer function.

The possible resolution and precision with the timer mode is completely dependant on the CPU-clock and the prescaler of the used 16bit timer. So we can say: the higher the CPU-clock is, the shorter the pulses to measure can be, but also the longest possible pulse will be reduced. The lower the CPU clock is, the longer the pulse times can be.

Contrary to the frequency counter part, which is dependant on the SysTick, the Timer part depends totally on the processor clock. The driver is optimized for clocks between 4 and 16MHz.

The minimum resolution can be adjusted in 4 steps: 10usec, 100usec, 1msec and 10msec
The typical maximal pulse times then are: 100msec, 1sec, 10sec, 100sec.

100msec area

resolution 10usec
max time min 100msec max 500msec

1sec area

resolution 100usec
max time min 1sec max 4sec

10sec area

resolution 1msec
max time min 4sec max 10sec

100sec area

resolution 10msec
max time min 40sec max 100sec

The pulse signal that must be measured must be connected to the capture input of the used timer. With Timer1 this is the PIN ICP1 and with the Timer3 this is PIN ICP3. The application must ensure that this port pin is programmed as an input.

The driver samples either a complete cycle or a positive pulse, depending on the selected mode.

TTimeBasexx = cycle capture

TPulseBasexx = pulse capture (positive pulse).

The timer mode uses the InputCapture mode of the selected 16bit timer. The readback of the timers is done in the InputCaptureInterrupt and doesn't load the CPU very much, but with long interrupt disable times in other parts of the application or the system, sometimes the result can be wrong. The system cannot see this problem and so cannot prevent it. The application must take care with of and check the results for plausibility.

Two Timer or Counter channels can be imported: FreqCounter and FreqCounter2.

FreqCounter2 is fixed to the Timer3 and so is only possible if the CPU has this Timer on chip, for example the mega64 and the mega128.

Imports

As usual with the AVRco system the driver must be imported and defined. For a complete implementation the import of the SysTick is also necessary.

FreqCount and/or FreqCount2

Imports the frequency counter/timer **FreqCount** or **FreqCount2**

The import of FreqCount also imports automatically several library functions and procedures.
The SysTick must also be imported.

Import SysTick, FreqCount [, FreqCount2];

Defines

The driver needs an internal 16bit timer, either Timer1 or Timer3, if it exists.

Define ProcClock = 8000000; {8Mhz clock }
 SysTick = 2.0; {2msec Tick}
 FreqTimer = Timer1; (used 16bit Timer)
 FreqTimer2 = Timer3; (restricted to Timer3)

FreqTimer, FreqTimer2

Defines the 16bit timer to be used. FreqTimer2 is fixed to Timer3. The SysTick should only have the following values: 1msec, 2msec, 2.5msec, 4msec, 5msec, 8msec, 10msec. All other values can lead to greater or lesser precision loss. See above the declarations in the overview.

3.42.3 Functions, Procedures, Types

tFreqCountMode

The import of the driver exports the type tFreqCountMode which is an enumeration. It serves as a select of the gate time or the prescaler of the counter. The following settings are possible:

TFreqBaseNone	Frequ = 0.01Hz...655.35Hz	gate time = 100sec
TFreqBase100Hz	Frequ = 0.1Hz...6.5535kHz	gate time = 10sec
TFreqBase1kHz	Frequ = 1Hz...65.535kHz	gate time = 1sec
TFreqBase10kHz	Frequ = 10Hz...655.35kHz	gate time = 100msec
TFreqBase100kHz	Frequ = 100Hz...6.5535MHz	gate time = 10msec
TFreqBase1MHz	Time = 10msec...100sec	period/pulse
TTimeBase100s/ TPulseBase100s	Time = 1msec...10sec	period/pulse
TTimeBase10s/ TPulseBase10s	Time = 100usec...1sec	period/pulse
TTimeBase1s/ TPulseBase1s	Time = 10usec...100msec	period/pulse
TTimeBase100ms/ TPulseBase100ms		

The definition of tFreqCountMode is:

Type tFreqCountMode = (TFreqBaseNone, TFreqBase100Hz, TFreqBase1kHz, TFreqBase10kHz, TFreqBase100kHz, TFreqBase1MHz, TTimeBase100s, TTimeBase10s, TTimeBase1s, TTimeBase100ms, TPulseBase100s, TPulseBase10s, TPulseBase1s, TPulseBase100ms);

FreqCountSema, FreqCountSema2

This semaphore will be incremented with each new result. Useful not only with MultiProcessing but also with standard applications to save CPU power.

SetFreqCountMode, SetFreqCountMode2

With this procedure the gate time/resolution or prescaler of the counter must be set. This must be done at least once at program start.

Procedure SetFreqCountMode (mode : tFreqCountMode);

The parameter **mode** is the desired resolution. Please note that at least the first measure cycle after a mode change is invalid.

GetFreqCountMode, GetFreqCountMode2

The function GetFreqCountMode returns as its result the actual mode of the counter

Function GetFreqCountMode : tFreqCountMode;



AVRco Standard Driver

GetFreqCounter, GetFreqCounter2

The function GetFreqCounter returns the result (16bit value) of the last frequency measurement cycle. How the result must be interpreted depends on the selected mode because it is a fixed decimal point value where the decimal point position depends on the selected mode. If an overflow occurs an \$FFFF is returned. A call to this function also clears the semaphore.

Function *GetFreqCounter: word;*

Important:

This function must only be used if the frequency counter mode is selected. Otherwise the function **GetTimeCounter** must be used.

GetFreqCounterL, GetFreqCounter2L

The function GetFreqCounterL returns the result (32bit value) of the last measure cycle. The result must be interpreted depending of the selected mode because it is a fixed decimal point value where the decimal point position depends on the selected mode. If an overflow occurs a \$FFFFFFFF is returned.

A call to this function also clears the semaphore.

Function *GetFreqCounterL: longword;*

Important:

This function must only be used if the frequency counter mode is selected. Otherwise the function **GetTimeCounter** must be used.

GetFreqCountOvrFlow, GetFreqCountOvrFlow2

The function GetFreqCountOvrFlow returns the overflow, if any, of the last conversion cycle. This value should be ZERO. If not then with the frequency counter the gate time is too long and the frequency is too high. But if 32bit values are used (this means the result is fetched with **GetFreqCounterL**) there is never an Overflow and this function is not necessary.

With the Pulse Timer mode the external pulse is too long because the selected mode expects a shorter pulse. A call to this function also clears the semaphore.

Function *GetFreqCountOvrFlow: byte;*

FreqCountRestart, FreqCountRestart2

This procedure resets the frequency counter to the initial state, so the next result will be valid. It can and should be used after SetFreqCountMode call for example.

Procedure *FreqCountRestart;*

GetTimeCounter, GetTimeCounter2

The function GetTimeCounter returns the result (16bit value) of the last pulse time measure cycle. The result must be interpreted depending of the selected mode because it is a fixed decimal point value where the decimal point position depends on the selected mode. If an overflow occurred a \$FFFF is returned.

A call to this function also clears the semaphore.

Function *GetTimeCounter: word;*

GetTimeCounterP, GetTimeCounterP2

The function GetTimeCounterP returns the result (16bit value) the plain counter content of the last pulse time measure cycle. The values Count1 and Count2 contain the timer value counted in the "high" and "low" time of the cycle. So for example it is possible to get the on/off relation of a PWM cycle. If an overflow occurred a false is returned. The *FreqCountMode* must be set to *PulseMode*.

A call to this function also clears the semaphore.

Function *GetTimeCounterP(var CountP, CountTot : word): boolean;*

Important:

These two function must only be used if the pulse/timer mode is selected. Otherwise the function **GetFreqCounter** must be used.

Example program:

an example is in the directory `..\\E-Lab\\AVRco\\Demos\\FreqCounter`

3.43 FreqCount driver – Frequency Counter X Mega

In some applications a function is necessary which can measure or count an external frequency. This implementation uses one of the eight existing 16bit timer (TimerC0 .. TimerF1) as an pulse counter. With the frequency counter the gate is handled by the SystemTick. This saves resources of the CPU. The timer gate can be set to 100sec, 10sec, 1sec, 100msec and 10msec. The result of the measure cycle is a 16bit value.

The drawback of the SystemTick usage is the somewhat reduced precision of the result and some limitations with the choice of the SysTick value. The inherent precision is in the range of 0.1%..0.2% depended of the processor clock and the SysTick value. The higher the CPU clock and the lower the SysTick value the better the result (within some limits). An optimal result can be achieved with a CPU clock $\geq 16\text{MHz}$ and a SysTick of 5msec and the **ADJ** option. If several interrupts are running there can be a jitter in the results.

Some limitations exist when the SysTick value has to be defined. Because of the gate times the following limitations apply:

The SysTick must be 2msec, 5msec or 10msec. 5msec preferred!

100Hz area

resolution	0.01Hz	
max frequency	655.35Hz	= 0.65535kHz
gate time	100sec	

1kHz area

resolution	0.1Hz	
max frequency	6553.5Hz	= 6.5535kHz
gate time	10sec	

10kHz area

resolution	1Hz	
max frequency	65535Hz	= 65.535kHz
gate time	1sec	

100kHz area

resolution	10Hz	
max frequency	655350Hz	= 655.35kHz
gate time	100msec	

1MHz area

resolution	100Hz	
max frequency	6553500Hz	= 6.5535MHz
gate time	10msec	

The clock or frequency signal that to be measured must be connected to any Port Pin of Ports A..F. One of the 8 Event Channels is used.

In the following context an **XX** denotes one of the Timers C0..F1.



AVRco Standard Driver

3.43.1 Import & Define FreqCount

Import *SysTick, FreqCount_C0, FreqCount_C1, FreqCount_D0, ...;*

Imports one or more frequency counter. The import of *FreqCount_XX* also imports automatically several library functions and procedures. The SysTick must also be imported.

The driver needs one of the internal 16bit timer, TimerC0..TimerF1, if it exists.

Define

```
//>> CPU           = 32MHz, PeripherX4=32MHz, PeripherX2=32MHz
OSCType             = int32MHz,
PLLMul              = 4,
prescB              = 1,
prescC              = 1;
SysTick             = 5, adj;           {msec} // ,adj
StackSize           = $0032, iData;
FrameSize           = $0064, iData;

FreqCount_D0        = PortD, 0, PullUp; // clock input port/pin, PullUp, PullDown, none
FreqCountEv_D0       = 0;               // event channel
```

SysTick

The SysTick should only have the following values: 2msec, 5msec, or 10msec. All other values are rejected.

FreqCount_XX

Defines the 16bit timer usage. The clock input port, pin and pullup must be defined. The PortA..PortF can be used and every pin (0..7) in these ports.

FreqCountEv_XX

Defines the Event Channel to be used (0..7).

3.43.2 Functions, Procedures, Types

Type tFreqCountRange

The import of the driver exports the type tFreqCountRange which is an enumeration. It serves as a select of the gate time of the counter. The following settings are possible:

TFreqBaseNone		
TFreqBase100Hz	Frequ = 0.01Hz...655.35Hz	gate time = 100sec
TFreqBase1kHz	Frequ = 0.1Hz...6.5535kHz	gate time = 10sec
TFreqBase10kHz	Frequ = 1Hz...65.535kHz	gate time = 1sec
TFreqBase100kHz	Frequ = 10Hz...655.35kHz	gate time = 100msec
TFreqBase1MHz	Frequ = 100Hz...6.5535MHz	gate time = 10msec

The definition of tFreqCountRange is:

Type *tFreqCountRange* = (TFreqBaseNone, TFreqBase100Hz, TFreqBase1kHz, TFreqBase10kHz, TFreqBase100kHz, TFreqBase1MHz);

Var FreqCountSema_XX

This semaphore will be incremented with each new result. Useful not only with MultiProcessing but also with standard applications to save CPU power.

AVRco Standard Driver



Procedure SetFreqCountRange_XX(range : tFreqCountRange);

With this procedure the gate time of the counter must be set. This must be done at least once at program start. The parameter **range** is the desired resolution. Please note that at least the first measure cycle after a mode change is invalid. It clears all internal data.

Function GetFreqCountRange_XX : tFreqCountRange;

The function GetFreqCountRange returns as its result the actual resolution of the counter

Function GetFreqCounter_XX: word;

The function GetFreqCounter returns the result (16bit value) of the last frequency measurement cycle. How the result must be interpreted depends on the selected mode because it is a fixed decimal point value where the decimal point position depends on the selected mode. If an overflow occurs an \$FFFF maybe returned. A call to this function also clears the semaphore.

Function GetFreqCountOvrFlow_XX: byte;

The function GetFreqCountOvrFlow returns the overflow, if any, of the last conversion cycle. This value should be ZERO. If not then with the frequency counter the gate time is too long and the frequency is too high. A call to this function also clears the Overflow.

Procedure FreqCountRestart_XX;

This procedure resets the frequency counter to the initial state, so the next result will be valid. It clears all internal data.

an example is in the directory `..\E-Lab\AVRco\Demos\XMega_FreqCount`



AVRco Standard Driver

3.44 RFID125 Receiver (*XMega only at this time*)

RFID RadioFrequencyIDentification is widely used to identify goods, tools, animals, persons etc. Also for access control, electronic locks, for example with keyfobs or plastic cards RFID is mainly used. There are several ways and techniques which mainly differ in the carrier frequency. In our case we use the commonly used 125kHz variant.

The binary number stored in the passive 125kHz transponders has a total length of 64bit. But the usable data length is only 40bit (5 Bytes), where the MSB is the ID of the transponder manufacturer. The other 4 bytes build a unique number which exists only once world wide. So there are a 2^{40} IDs possible.

An RFID125 receiver needs a high quality antenna (coil) and a sensitive receiver chip. Both parts are not expensive and are available in the commerce. But it makes only sense to build an own receiver in high volumina. Because of this the AVRco driver is based on the active antenna TowiTek TWT2021 (Conrad Electronic). This device provides an async/serial output with 9600Bd, 8N1 and works from 3.3V to 5V. With a valid transponder signal every 200msec a 5Byte packet is send to the host.

Though this device works from 3.3V on one must know that the maximal distance from the transponder to the antenna can be 1cm at 3.3V and 5cm at 5V.

Only the Rx channel of the UART is used, so the Tx pins can be used as GPIOs.

Imports

As usual with the AVRco system the driver must be imported and defined. The SysTick is used for TimeOuts.

Import *SysTick, RFID125;*

Defines

The UART, the baudrate and the packet size must be defined:

Define ...
 SysTick = 10; // 5msec
 RFID125 = SerPort0, 9600, 5; // SerPort, Baudrate, Buffersize
XMega
 RFID125 = SerPortD1, 9600, 5;

Driver variables

The driver exports an array with the 5 ID bytes:

Var
 RFIDbuff : array[0..Buffersize-1] of byte;

Functions

There is only one function. It returns a true if a valid ID has been read. With a call of this function the internal Ready becomes a false.

function *RFIDready* : boolean;

A sample is in the directory ..\E-Lab\AVRco\Demos\XMega_RFID



AVRco Standard Driver

Imports

As usual with AVRco the driver (Rx, Tx or both) must be imported.

Import SysTick, RC5Rxpport, ...;

or

Import SysTick, RC5Txport, ...;

or

Import SysTick, RC5Txport, RC5Rxpport, ...;

For the receiver part the desired input port (PINx) and the PortPin must be defined.

For the transmitter part the output polarity must be defined:

Defines

```
Define ProcClock      = 8000000;           {Hertz}
          SysTick       = 10;                {msec}
          StackSize     = $0010, iData;
          FrameSize     = $0010, iData;
          RC5TXPORT     = negative, 36;      { opt Timer, pulse polarity, opt carrier freq}
          RC5RxPort     = PinD, 7;           {PinD, Portpin 7, opt polarity}
          // RC5RxPort  = PinD, 7, negative; {negative or positive pulse}
          RC5mode       = rc_7bit;           // rc_6bit = default
```

3.45.3 Functions Receiver

Function RecvRC5 (*var rxAdr : byte; var rxCmd : byte*) : *boolean*;

This function returns a false after a Timeout of 130msec. Neither variable will be changed. If the function was successful a true is returned and both variables contain the received values. Bit6 (ext mode bit7) in rxCMD contains the Toggle bit.

RxAdr = 0 0 0 A4 A3 A2 A1 A0

RxCmd in Standard Mode

TOG CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

RxCmd in Extended Mode

TOG CMD6 CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

3.45.4 Functions Transmitter

Procedure SendRC5 (*const txAdr : byte; const txCmd : byte*);

This procedure sends the two bytes via the IR-transmitter. The Toggle bit6 (ext mode bit7) in txCMD must be set or reset by the application.

TxAdr = 0 0 0 A4 A3 A2 A1 A0

TxCmd in Standard Mode

TOG CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

TxCmd in Extended Mode

TOG CMD6 CMD5 CMD4 CMD3 CMD2 CMD1 CMD0

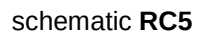
Attention:

Too many or too long interrupts while sending or receiving can corrupt the protocol so that eventually the telegram becomes invalid. With **MultiTasking** the Scheduler must be disabled while receive or transmit.

Example program and schematic:

an AVR example is in the directory ..\E-Lab\AVRco\Demos\RC5

an X Mega example is in the directory ..\E-Lab\AVRco\Demos\XMega_RC5





AVRco Standard Driver

3.46 SHT11 Temperature and Humidity Sensor Driver

Overview

Room temperature and humidity are the two essential factors of the room climate. The sampling of these values is important for air conditioning and heating systems. But also the simple display of these values is an important task.

But there are always problems with the relative humidity. All known sensors are extremely imprecise and unstable and many additional parts are necessary. Generally the precision and the response time are very disappointing.

Furthermore both common sensors need OP-amps and reference circuits and a linearization, either in hardware or software.

With the help of the new intelligent sensors from Sensirion now it's possible measure both values with a superior precision and stability. The linearization is build into the sensor. With temperature the system error is +/- 0.5 °Celsius at room temperature. The humidity error is about +/-3.5%, which is an excellent value. The readouts are very stable. By using parts with tighter tolerance the accuracy can be improved. By implementing some additional linearization algorithms in software the precision can be improved further.

The temperature range of the sensors is from -40grad to +120grad Celsius.
In the end ranges the max. error is +/-3 °Celsius

The humidity range is 0% to 100%RH. Throughout the range the max. error is +/-5%RH

Introduction SHT11drv

The SHT11 sensor works in the range of 2.5 to 5.5Volt. The connection between sensor and CPU is a two-wire interface similar to the I2C interface but incompatible with this.

The communication uses two control lines (SCK and DATA) which can be connected to any port pins of the CPU. However, the DATA line must be bi-directional (DDR register) and must be pulled high by a pullup resistor of 2..3kOhm.

The data is read and written by a bit serial protocol. After each start command a conversion in the sensor runs. The sensor shows the end-of-conversion by pulling low the data line.

The conversion time for a single value is between 10 and 250msec dependent of the operating voltage and the selected resolution. Because of this in the real world it's not practicable and makes no sense to wait for conversion complete for up to 250msec after the start command until the data line goes low. There must be more important things to do in the meantime. Because the sensor has a fully static design it pulls low its data line until it sees the first read-clock pulse from the CPU.

So it's possible to start the conversion, do other jobs and after a time return back and read the state of the data line for READY (SHT11ConvState). With MultiTasking applications the driver sets a semaphore to "1", if the ready state of the data line appears. By that method the reading process can suspend itself with "WaitSema". It will be awakened by the Scheduler when the result from the sensor is readable.

Because of this the temperature and the humidity read functions are implemented in two parts, a conversion start part (SHT11startTemp and SHT11startHum) and readout part (SHT11getTemp and SHT11getHum). Before the "SHT11getTemp" function can be called either the state of the conversion must be polled by "SHT11ConvState" or the application must use "WaitSema".

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\SHT11`

In this directory there is also a helper file that contains additional inearising functions.

AVRco Standard Driver



Imports

As usual with the AVRco system the driver must be imported and defined.
The SysTick is not used.

Import *SysTick, SHT11drv;*

Defines

The status mode and the two port pins must be defined.

Define *ProcClock* = 8000000; // 8Mhz clock
 SysTick = 5; // 5msec
 SHT11drv = *polled [, Delay][, crc];* // polling only
 // *SHT11drv* = *SysTickChecked;* // polling + semaphore
 SHT11clk = *PortD, 5;* // port and pin for clock
 SHT11dat = *PortD, 6;* // port and pin for data

SHT11drv

Defines the polling mode. With “SysTickChecked” a semaphore will be implemented which can be used to control a process via WaitSema. The polling mode using “SHT11ConvState” is then also useable. The optional parameter “Delay” defines the bit rate (default 0, min 0, max 255).
The optional parameter “CRC” defines whether the SHT11 CRC byte must be readout and stored.

SHT11clk

Defines the port and pin used for the Clock line. The port can be an output only port.

SHT11dat

Defines the port and pin used for the Data line. The port must be a bi-directional port.

3.46.1 Variables

If the “SysTickChecked” mode is defined a semaphore is implemented which can be used by processes in conjunction with WaitSema. The conversion start functions clear this semaphore. In the SysTick this semaphore is set to “1” if the sensor is ready with the conversion.

Var *SHT11sema* : *Semaphore;*

If the option “CRC” is given in the Define section, a data read out of the SHT11 always includes the CRC byte. This byte then is stored in the location SHT11crc.

Var *SHT11crc* : *byte;*

Attention:

If a conversion is running there must be no other sensor access except “SHT11ConvState”.

3.46.2 Functions

SHT11synchronize

Resets the 2-wire interface. Is also called from the system init.

Procedure *SHT11synchronize;*
 SHT11synchronize



AVRco Standard Driver

SHT11ConvState

Returns a true if the sensor is ready for read out after a conversion start.

Function *SHT11ConvState* : *boolean*;

repeat

until *SHT11ConvState*;

SHT11startTemp

Starts the temperature conversion.

Procedure *SHT11startTemp*;

SHT11startTemp;

SHT11getTemp

Reads out the measuring result after a conversion is finished.

Function *SHT11getTemp* : *word*;

Ww:= SHT11getTemp;

SHT11startHum

Starts the humidity conversion.

Procedure *SHT11startHum*;

SHT11startHum;

SHT11getHum

Reads out the measuring result after a conversion is finished.

Function *SHT11getHum* : *word*;

Ww:= SHT11getHum;

SHT11getStatus

Reads back the Status/Control register of the sensor. If a \$FF is returned the sensor maybe defect or not connected. This check should be made at start up.

Function *SHT11getStatus* : *byte*;

if *SHT11getStatus* = \$ff **then**

SensorFailed ...

endif;

SHT11setStatus

Some special functions of the sensor can be executed e.g. changing the resolution, heater switch etc.

Procedure *SHT11setStatus*(*s* : *byte*);

SHT11setStatus(\$01); *// reduce solution to 8/12bit*

SHT11softReset

Reset the Status/Control register to its internal default values.

Procedure *SHT11softReset*;

SHT11softReset; *// clear status reg to default values*

3.47 Sound Generator and Driver

The communication between the controller/electronic and the environment often is an important task. Beside displays, keyboards and computer interfaces sometimes an acoustic signal makes sense or indeed is necessary.

These needs are supported by this Sound Driver. Of course it can't play music but it is able to generate some fixed sounds and also a variable sound. Furthermore it can generate a desired frequency for a user-defined duration. As a playback device often a small piezo beeper is sufficient, which is connected through a capacitor (1uF) to the desired Port Pin. If more volume is necessary an external audio amp must be added.

Imports

The driver must be imported, as usual with AVRco. It also needs the import of SysTick:

Import SysTick, BeepPort, ...;

Defines

The desired Port and the PortPin has to be declared with a Define:

Define

```
ProcClock = 8000000;      {Hertz}
SysTick    = 10;          {msec}
StackSize  = $0020, iData;
FrameSize  = $0030, iData;
BeepPort   = PortB, 0;
```

3.47.1 Functions

There are 8 different Sound functions:

Procedure BeepSiren (*const mode : byte; repTimes : byte*);

The call of this procedure generates one siren cycle. The kind of the siren has to be defined with the parameter mode (0 or 1). The parameter repTimes defines how often the sound is repeated.

Procedure BeepClick;

This procedure generates a short click.

Procedure BeepOutLH;

Procedure BeepStepLH;

These procedures generate a short inclining sound sequence with 3 or 5 tones.

Procedure BeepOutHL;

Procedure BeepStepHL;

These procedures generate a short declining sound sequence with 3 or 5 tones.

Procedure BeepChirpH (*repTimes : byte*);

This procedure generates a short declining high sound sequence (chirp). The parameter repTimes defines how often the sound is repeated.

Procedure BeepChirpL (*repTimes : byte*);

This procedure generates a short declining lower sound sequence (chirp). The parameter repTimes defines how often the sound is repeated.



This procedure generates a short creaking sound

This procedure generates a sound with the frequency passed in parameter freq. The minimum frequency is 40Hz; the maximum is 10kHz. Dependant of the processor clock it's possible that the maximum or minimum frequency is not achievable. With 16MHz both limits are legal. The duration of the sound is determined by the parameter ticks, which counts in SysTicks.

an example is in the directory `..\E-Lab\AVRco\Demos\Sound`



3.48 SysLEDBlink

Overview

The controlling of LEDs (Light Emitting Diodes) is a typical job of an embedded system. In most cases this is the first experiment of a beginner because it's a simple program and the results are visible immediately. A port must be programmed as an output and single bits of this port are set to low or high. Ready!

But if LEDs should be flashed there is a big problem where even professionals run into difficulties. Because LEDs must also blink rhythmically while the program is running and a shortened or prolonged On or OFF time is irritating, basically a free timer is used. In the timer interrupts the blink job must be done. But what to do if no timer is available? Or if a free timer must be reserved for future enhancements? Then there is a huge problem.

Because most applications with the AVRco system import the **SysTick** this tick can be used. Implementing the Callback function **OnSysTick** the LED handling can be done here. But this should be written in assembler in order to avoid unwanted increase of the interrupt disable time by this handler. Furthermore the problem of "register save" must be considered.

With the arguments above in mind it makes sense to implement the LED blinker as a driver into the system. This driver is linked into the SysTick and does its job completely transparently without any intervention by the application. But note that this driver is a heavy duty job in the systick.

Introduction SysLEDBlink

The driver serves as a controller for up to 8 LEDs where each LED can be controlled separately. The functions exported by SysLEDBlink are:

1. LED on
2. LED off
3. all LEDs on
4. all LEDs off
5. LED blink on
6. LED blink off
7. all LED blink on
8. all LED blink off
9. Blinktimer setup in SysTicks
10. Blink LED x-times
11. Driver On/Off
12. LED 1xblink

The particular LEDs can be distributed to any ports and Pins also the common memory is possible if it is R/W. A simple latch as a port is not possible.

The common blink interval is selectable in a wide range. The polarity of each LED is also selectable e.g. active LOW or active HIGH.

An alternating blinking of some LEDs is possible by changing the basic state (On/Off) of these LEDs. For example if LED1 is ON and LED2 is OFF and both are set to blink so these two LEDs are flashed alternately.

Note:

There is **no initialisation** of the used ports (DDRx etc). It's the job of the application to do this in a proper way. In addition this port setup should not be changed by the application at runtime. The driver doesn't check or support this.

If the function **SysLEDflashMsg** will be used then also the Message Mode must be imported:
From SysLEDBlink Import LEDmessage;

If the function **SysLEDflashOnce** will be used then it must be imported with:
From SysLEDBlink Import FlashOnce;



AVRco Standard Driver

3.48.1 Implementation

Imports

As usual with the AVRco system the driver must be imported.

Import SysTick, SysLEDBlink, ..;

From SysLEDBlink **Import** LEDmessage, FlashOnce, FastBlink; // these are options

Defines

the blink rate/interval in SysTick units

up to 8 port registers and portpins PORT, BIT, ACTIVE_LEVEL

```
Define ProcClock      = 8000000;           {Hertz}
        SysTick        = 10;               {msec}
        StackSize      = $0030, iData;
        FrameSize      = $0030, iData;
        SysLEDBlink    = 30;               {30*SysTick = 300msec}
        // alternative
        //SysLEDBlink  = mSec300;          {10..1000 msec}
        SysLEDBlink0    = PortA, 0, high;   {LEDon = high level}
        SysLEDBlink1    = PortA, 1, low;    {LEDon = low level}
        SysLEDBlink2    = PortA, 2, low;
        SysLEDBlink3    = PortA, 3, low;
        SysLEDBlink4    = PortA, 4, low;
        SysLEDBlink5    = PortA, 5, low;
        SysLEDBlink6    = PortA, 6, low;
        SysLEDBlink7    = PortA, 7, low;
```

Attention: at least the *SysLEDBlink0* must be defined! There should be no gaps in this defines.

As an alternative to the SysLEDBlinkX defines there can be a byte variable in RAM instead of the Port bits:

```
SysLedPort = @LEDram, $00; // byte-var, polarity
```

With this define the generated variable then contains the Blink bits which can be passed to a virtual or I2C/TWI port.

Imports and Exports

A SysLEDBlink import also imports some bytes which are reserved for the system. The blink handler is linked into the SysTick.

3.48.2 Functions

Procedure SysLEDon (b : byte); // b= 0..7 LED number

Procedure SysLEDonOff(b : byte; on : boolean); // b= 0..7 LED number

Procedure SysLEDallOn;

Procedure SysLEDOff (b : byte); // b= 0..7 LED number

Procedure SysLEDallOff;

Procedure SysLEDflashOn (b : byte); // b= 0..7 LED number

Procedure SysLEDflashOnOff(b : byte; on : boolean); // b= 0..7 LED number

Procedure SysLEDflashAllOn;

Procedure SysLEDflashOff (b : byte); // b= 0..7 LED number

Procedure SysLEDflashAllOff;

AVRco Standard Driver



Procedure SysLEDflashMsg (led, msg, rept : byte); // expects LEDmessage Import

Procedure SetSysBlinkTimer (t : byte); // t= 2..255

Procedure SysLEDfastBlink (fast : boolean); // enable/disable fast mode

Procedure SysLEDenable (ena : boolean); // enable/disable the driver

Procedure SysLEDflashOnce (b : byte); // b= 0..7 LED number

Details

Procedure SetSysBlinkTimer (t : byte); // t= 2..255 in SysTicks

Initially the blinkrate (on/off Time) must be set by the define **SysLEDblink**, but the blinkrate can be changed at runtime with this function.

Procedure SysLEDfastBlink (fast : boolean); // enable/disable fast mode

The blinkrate can be switched to fast or normal without changing the timer value.

Procedure SysLEDenable (ena : boolean); // enable/disable the driver

The processing of the LEDs and their timers is done in the SysTick. This consumes CPU power/time. If none of the blink functions is used then blink handling can be completely disabled temporary.

Procedure SysLEDflashMsg (led, msg, rept : byte); // expects LEDmessage Import

This function for example supports the blinking of an error message. The parameter **led** defines the LED that must be used. The parameter **msg** defines the count of blinks of this telegram, where the max value is 126. The parameter **rept** defines how often this telegram must be repeated. The value "255" forces an unlimited repeat. In this case the blinking must be aborted with the function **SysLEDflashOff** or **SysLEDflashAllOff**

Note:

only one LED can show a message at a time.

Hint:

It is possible to give each LED a name. Example:

Type myLEDs = (LEDenter, LEDexit, LEDpower);

This enumeration now can be used with the functions above:

SysLEDflashOn (byte (LEDpower));

This is another way:

Const

LEDenter : byte = 0;

LEDexit : byte = 1;

LEDpower : byte = 2;

These constants now can be used with the functions above:

SysLEDflashOn (LEDpower);

Example program:

an example is in the directory ..\E-Lab\AVRco\Demos\SystemLEDs



AVRco Standard Driver

3.49 Line Printer Driver LPTport

Also with embedded applications sometimes it is necessary to control a printer in order to print out information, measurement results or other data on paper. In most cases these printers provide a serial interface, but some printers are equipped with the old fashioned but still used parallel-interface, also called the Centronics Interface.

The Centronics Interface consists of an 8bit dataport and an 8bit controlport. Basically the dataport always is an output type while the controlport has state lines (input) and also control lines (input/output). The LPTport driver implements this standard configuration. In addition there are some support functions that allow a bi-directional service.

The driver always uses 2 complete 8bit ports. The driver implements either two bi-directional ports of the AVR or the Philips I2C dual-port chip PCA9555. The PCA can be controlled through the software I2Cport and also with the TWIport of the AVRco.

The operation mode of the LPTport driver must be selected by its import and its define.

Imports

As usual with the AVRco the driver must be imported:

Import SysTick, LPTport, ..;

The operation mode AVRport, I2C or TWI must be set by the Define. Example for the port mode:

Define

```
ProcClock = 8000000;      {Hertz}
SysTick    = 10;          {msec}
StackSize  = $0010, iData;
FrameSize  = $0010, iData;
LPTport    = PortA, PortB; // DataPort, ControlPort
```

If the TWI mode is used then the TWI driver must also be imported:

Import SysTick, TWImaster, LPTport, ..; // TWInetMaster is also possible

Define

```
ProcClock = 8000000;      {Hertz}
SysTick    = 10;          {msec}
StackSize  = $0010, iData;
FrameSize  = $0010, iData;
TWIpresc   = TWI_BR400;
LPTport    = TWI_I2C, $24; // $24 = TWIaddr
```

XMega

Import SysTick, TWI_C, LPTport, ..; // TWI_D, TWI_E, TWI_F are also possible

Define

```
OSctype    = int32MHz, PLLmul=4, prescB=1, prescC=1;
SysTick     = 10;          {msec}
StackSize   = $0020, iData;
FrameSize   = $0040, iData;
TWIprescC   = TWI_BR400;
LPTport     = TWI_C, $24; // $24 = TWIaddr
```

AVRco Standard Driver



If the Soft-I2C mode is used then the I2Cport driver must also be imported:

Import SysTick, I2Cport, LPTport, ...;

Define

```
ProcClock = 8000000;      {Hertz}
SysTick    = 10;          {msec}
StackSize  = $0010, iData;
FrameSize  = $0010, iData;
I2Cport    = PortA;
I2Cclk     = 0;           // bit0, porta
I2Cdat     = 1;           // bit1, porta
LPTport    = Soft_I2C, $24; // $24 = I2Caddr
```

3.49.1 Functions and Types

The driver exports 2 types:

Type

tLPTlines = (IpStrobe, IpError, IpInit, IpSelect, IpACK, IpBusy, IpSelected, IpPaper);

tLPTlineSet = BitSet of tLPTlines;

The enumeration type **tLPTlines** exactly corresponds with the lines/connections of the control port.

Name	Bit #	dir	active	connector	databit #	connector
IpStrobe	0	output	low	1	0	2
IpError	1	input	low	15	1	3
IpInit	2	output	low	16	2	4
IpSelect	3	output	low	17	3	5
IpACK	4	input	low	10	4	6
IpBusy	5	input	high	11	5	7
IpSelected	6	input	high	13	6	8
IpPaper	7	input	high	12	7	9

The BitSet **tLPTlineSet** contains the above 8 bits in a byte.

Functions

Procedure LPTinit;

Initialises the printer by the Init line and the Select line. Should be called before each printing cycle.

Procedure LPTreset;

Generates a short pulse on the Init line and resets the printer.

Function LPTstat : tLPTlineset;

Returns the state of the control port. Please note that the result always includes the active states of the lines and not the binary state. If the ACK is active (=0) the IpAck bit in the result is set. If the IpBusy line is active (=1) then the IpBusy bit in the result is set.

The application can start printing only when the **IpSelected** bit is active and the **IpPaper** bit is inactive.



AVRco Standard Driver

Procedure *LPTout (dat : byte);*

This is the printing function. This function immediately returns without any further operation when the **IpSelected** line is inactive. Otherwise it waits until the **IpBusy** line is inactive and then sends the byte to the printer.

LPTinit;

Repeat *until (IpSelected in LPTstat);*

WriteLn (*LPTout, 'Hello'*);

Support Functions

The following functions and procedures are not necessary for the printer operations but are reserved for special implementations like bi-directional data transfers.

Procedure *LPTctrl (ctrl : tLPTlineSet);*

Controls the control port and the control lines to the printer.

Procedure *LPTdir (inp : boolean);*

Controls the data direction of the data port. True = output, false = input.

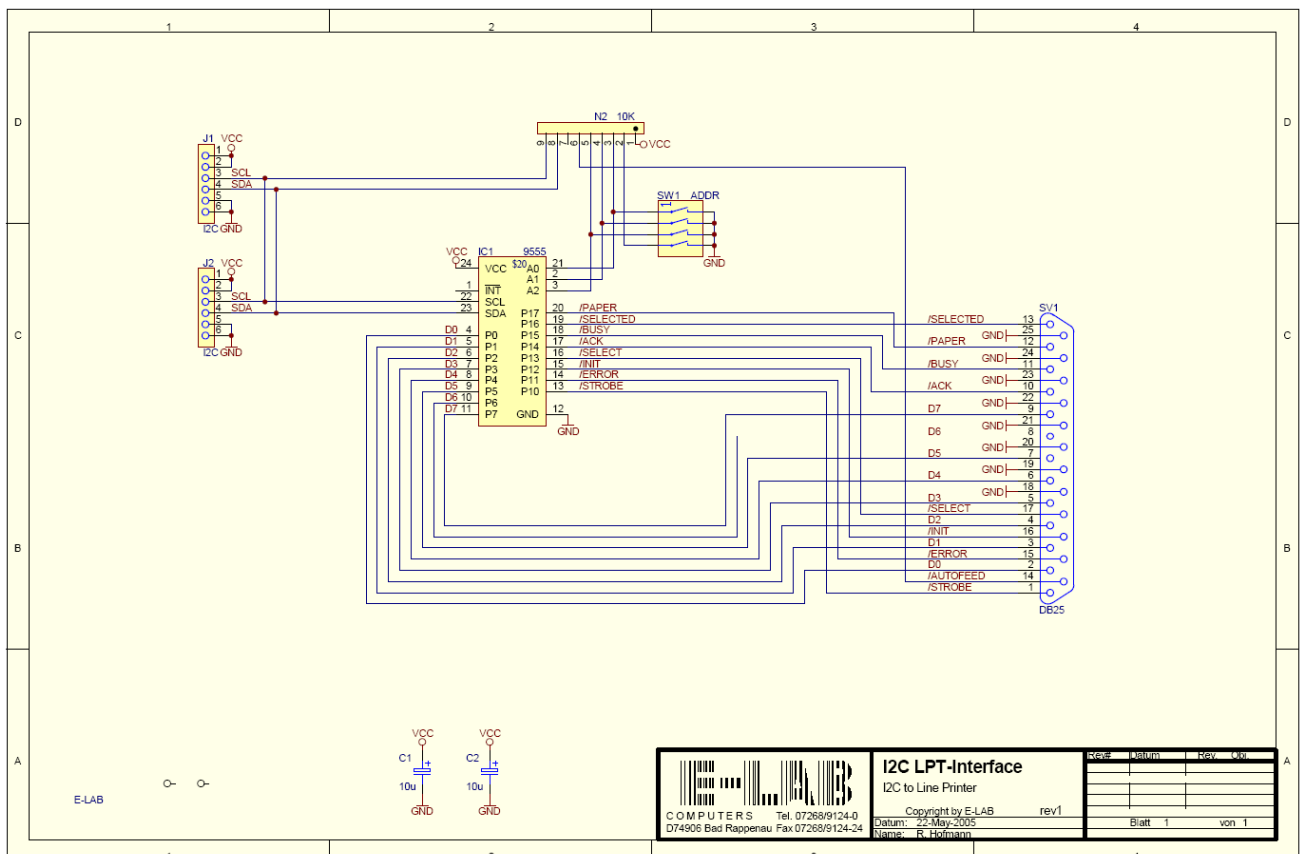
Function *LPTinp : byte;*

Reads the data port.

Example program and schematic:

an example is in the directory `..\\E-Lab\\AVRco\\Demos\\I2C_LPT`

an XMega example is in the directory `..\\E-Lab\\AVRco\\Demos\\XMega_LPT`



schematic **I2C_LPT**

3.50 Banking Port

With some kinds of application there is a need to hold a huge amount of variables as constant and fast access to the program. A file system can help, but because of its complexity it is not very fast and handling is also not so easy as a simple access to a common variable.

Between the linear address area of the AVR and a filesystem there is a memory extension called banking system.

Here a theoretically practically unlimited memory expansion can be implemented with mapping a small part of the large external memory into the standard address area of the CPU. The external memory is divided into equal sized parts. Hardware connects a part (bank) under software control into a free memory area.

The user can place all types of variables into such a bank and can access them fully transparently without any interventions (except pointers to bank vars, see below). In order to populate the different banks with any variables, the user must define where (in which bank) a specific var resides. This must be done with the compiler switch `{ $BDATA # }` where `#` is the desired bank number.

In this banking implementation up to 16 banks with 32kB each can be defined and accessed. This results in an additional memory size of 512kBytes.

In general, the external Device can be a data storage type of any kind. The device is logically divided into up to a maximum of 16 banks, where each bank has a size of 32kBytes. The user must implement the hardware drivers for read a byte and write a byte, where the system provides a relative address (0..\$7fff) and a banknumber (0..15).

The user's drivers must generate an appropriate access to the storage using these 2 parameters. Address mapping and combining the bank number with the relative address in any fashion is possible. So this schema can also be used for flashcards or floppies.

Restrictions

Complex arrays and records are not supported (arrays of records, arrays in records etc).

Strings:

In general only string copy and string manipulation (indexed) is implemented.

No string concat with a banked string in source or destination like:

st:= banked_string + 'abc';

No string conversions with a banked string in source or destination like

int:= StrToInt (banked_string);

or

banked_string:= IntToStr (num);

Imports

The driver must be imported as usual with AVRco.

Import SysTick, BankPort, ...;

Defines

the BankPort bank count in 32kByte pages

Define *ProcClock* = 8000000; {Hertz}
 SysTick = 10; {msec}
 StackSize = \$0030, *iData*;
 FrameSize = \$0030, *iData*;
 BankPort = 3; {Bank Ports}



AVRco Standard Driver

Populate

The additional Bank pages are populated with variables by using the compiler switch "{\$BDATA x}", here x is the bank number for the following defines.

```

{$BDATA 0}
var
  bb0    : byte;
  ww0    : word;

{$BDATA 1}
var
  bb1    : byte;
  ww1    : word;

```

3.50.1 Implementation

The "UserDevice BankDevIni" special procedure should be used to perform any initialisation necessary.

```

UserDevice BankDevIni;      // Bank Device init
begin
  (* is called at System Init Time *)
  (* initialize Device Hardware *)
end;

```

Implement the "UserDevice BankDevInp" special procedure. The desired data address is passed in a word argument and the bank number as a byte. The procedure must return a byte value

```

UserDevice BankDevInp (bank : byte; adr : word) : byte;
begin
  ...
  return(xxx);
end;

```

Implement the "UserDevice BankDevOut" special procedure. The bank number is passed in the first parameter as a byte. The desired data address is passed in the second argument = word. The Byte-value to write to the external DATA is passed in the third argument

```

UserDevice BankDevOut (bank : byte; adr : word; arg : byte);
begin
  ...
end;

```

Special access cases:

Banked variables in expressions etc. which are referenced by their names are handled like others. The compiler knows where they reside and generates a banked access. Because **pointers** are always 16bits there is no information about the memory area that it should point to. So a pointer implies always an access into the CPU's linear address area from 00..\$FFFF.

The user can assign the address of a banked var to a pointer like:

```
ptr := @banked_var;
```

But if the pointer is now dereferenced like:

```
ptr^ := xxx;
```

there is no information that a banked area should be addressed. The user must tell the compiler this with:

```
Function BankDevPtr (bank : byte; ptr : pointer) : pointer;
```

e.g.:

```
BankDevPtr(bank#, ptr)^:= xxx;  
xxx:= BankDevPtr(bank#, ptr)^;
```

Now the pointer is redirected into the bank represented by "bank#" and the access goes into a bank.

Another special case is when a banked var is passed to a procedure or function by its **address** like:

```
Procedure func_name (var bbbb : byte);
```

e.g.:

```
Func_name (banked_var);
```

The called procedure/function expects a byte represented by its address. If the calling statement passes a banked var, the function can't know that the received address should point into the banking area, where the original var resides. The passed address can be interpreted as a pointer to byte. But unlike a pointer, which is dereferenced using: ptr^, a var parameter is only dereferenced by its name, the "^" is not used. In Pascal the compiler knows these circumstances.

Therefore the above used construction "BankDevPtr" is not applicable here! So **within** such a function the user must declare such a VAR-parameter to a banked-Var-parameter with

```
^Bank[bank#].var_name:= xxx;
```

The driver only provides a single function:

```
Function GetBankNum(bankedVar) : byte;
```

The function returns the bank number of the banked variable "bankedVar".

For clarification please carefully read and analyse the **test program** for banking in the directory
..\E-Lab\Demos\Banking

3.50.2 Hardware Example

In the example given in the **test program** and the schematics below the banks always reside at address \$8000..\$FFFF.

An external 512kByte SRAM (16 banks) is implemented. The first physical bank is used as a XDATA area and therefore not useable for the banking scheme. The advantage of this is that standard memory as XDATA doesn't produce any overhead compared to banking.

If the XDATA page is used, the banking cannot use bank 0. Because of this, in this example the passed bank number must be incremented by one. So the logical bank0 is mapped into the physical bank1 etc. etc. Another simple way to use the XDATA is do not use the logical bank0 (\$BDATA 0).

If XDATA is used, please note that **interrupts** can access the XDATA area. If so, the interrupt must be disabled for the duration of the bank access. On exiting the drivers it must be guaranteed, that always bank0 is addressed.

Because there must be any way to generate the additional addresses necessary for the 512kB SRAM, a banking port is implemented at address \$7FFF (memory mapped). This can be a simple latch, a PLD etc. If there are some pins free in the CPU's ports, these pins also can be used to generate the required address lines.

In this example each bank starts at location \$8000 so this constant must be added to the address passed.



AVRco Standard Driver

program BankTest;

Device = 90S8515, VCC=5;

Import SysTick, BankPort;

From System **Import** ;

Define

```
ProcClock = 6000000;      {Hertz}
SysTick    = 10;          {msec}
StackSize  = $0064, iData;
FrameSize  = $0064, iData;
BankPort   = 3;           {Bank Ports}
```

Implementation

```
{-----}
{ Var Declarations }
{$IDATA}
var bb : byte;

{$BDATA 0}           // variables into bank 0
var bb0 : byte;

{$IDATA}
{-----}
{ functions }
(* for a BankDevice import the following 3 function must be implemented *)
```

UserDevice BankDevIni; // Bank Device init

begin

(* is called at System Init Time *)

(* initialize Device Hardware *)

end;

UserDevice BankDevInp (bank : byte; adr : word) : byte;

begin

...

return(xxx);

end;

UserDevice BankDevOut (bank : byte; adr : word; arg : byte);

begin

...

end;

```
{-----}
{ Main Program }
```

begin

EnableInts;

loop

NOP;

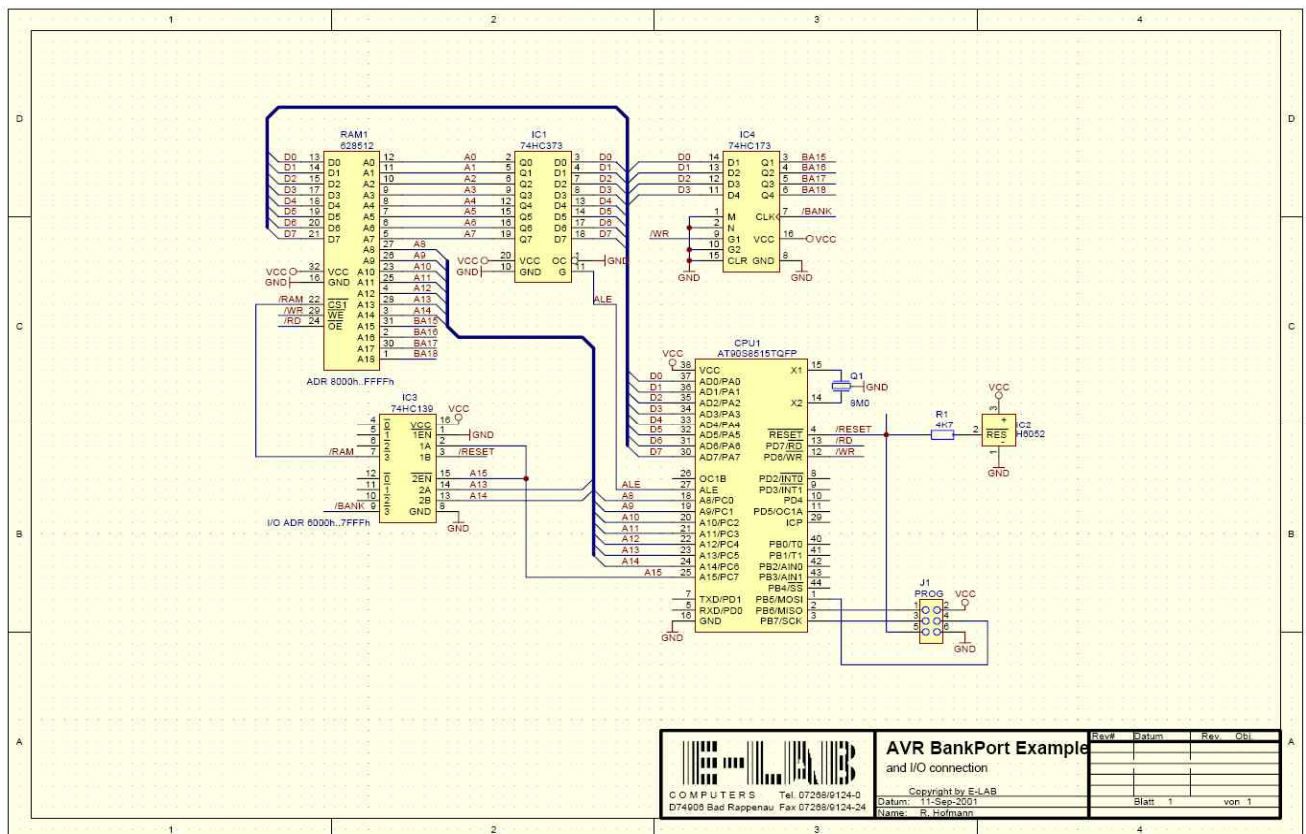
endloop;

end BankTest.

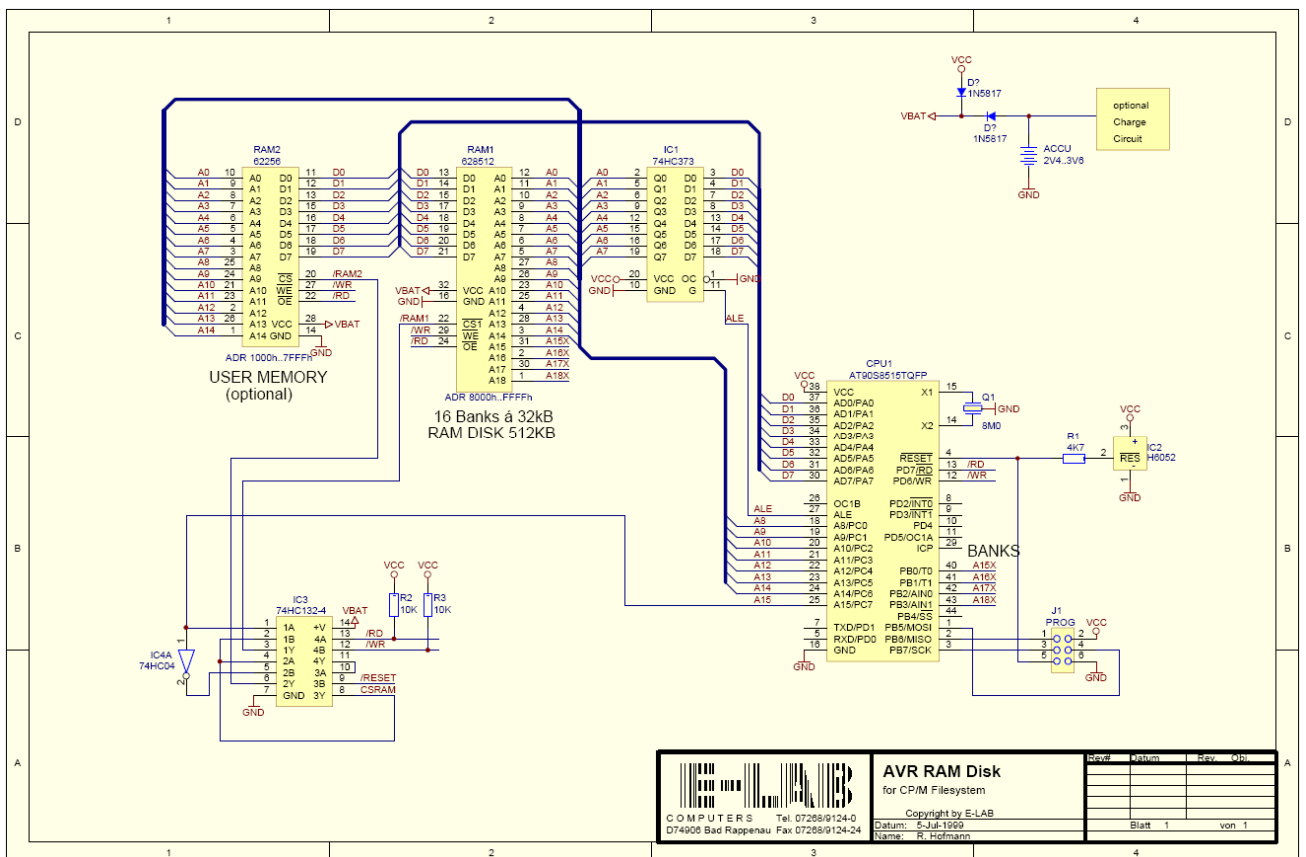
Example program:

an example is in the directory **..\E-Lab\AVRco\Demos\Banking**

AVRco Standard Driver



schematic **BankPort**



schematic **AVRRamDisk**



AVRco Standard Driver

3.51 Flash Download/Writer

3.51.1 Overview

The new AVR CPUs (mega8, mega16, mega128 etc) allow changes of the onChip Flash content at runtime by the application program. This opens many new possibilities for programmer from a simple change of ROM-constants and Code-modification up to download of an updated or new application.

For this purpose the new CPUs provide the epode **SPM** (StoreProgramMemory) and a specific area in the Flash memory, the so-called **BootLoader Section**. With the help of the SPM instruction writing into the Flash is possible now. The SPM instruction itself normally can only reside and be executed in the BootLoader section.

It is essential that all the operations concerning the SPM instruction always use a block of bytes (page). With the mega128 this blocksize is 128Bytes. Furthermore each access into such a page must be Word-based.

AVRco supports these features of the new AVR CPUs with the Import instruction *Import FlashWrite* and the Compiler switches **{\$BOOTRST \$nnnn}**, **{\$PHASE BOOTBLOCK \$nnnn}** and **{\$DEPHASE BOOTBLOCK}**.

In addition two new library groups (drivers) are implemented: FlashWrite at Runtime and FlashDownLoad at Runtime.

Import

In order to use these library functions the group **FlashWrite** must be imported:

Import SysTick, FlashWrite, ...;

Please observe that this **import must be placed at least at the second position** in the importlist. This import generates a word variable **FLASH_ADDR** at the beginning of *iData*. If the Flash is larger than 64kB an additional byte variable **FLASH_PAGE** is generated. These variables are used by all internal FlashWrite functions and must not be changed when such operations are running (with one exception). Otherwise they can be freely used by the application.

3.51.2 Compiler Switches

{\$BOOTRST \$nnnn}

This switch informs the Debugger/Simulator E-LAB AVRsim whether the Hardware Reset uses the vector at Addr \$0000 or the reset jumps into the Bootblock. The parameter \$nnnn defines, if activated, the Reset Jump address for the Simulator at reset time. With the real CPU the Fusebit *BOOTRST* programmed by the InCircuit Programmer defines the behavior of the CPU with a hardware reset. This switch is meaningless for the generated program and therefore also has no influence of the working of the CPU itself. The address parameter can be arbitrary, but only makes sense if it corresponds to the possible Jump addresses of the used CPU. This switch must be placed after the Device declaration.

mega8 \$nnnn = \$0C00, \$0E00, \$0F00, \$0F80;
mega16 \$nnnn = \$1C00, \$1E00, \$1F00, \$1F80;

{\$PHASE BOOTBLOCK \$nnnn}

This switch assigns the following Code Block to the address \$nnnn and up. In addition it serves to inform the Simulator to allow normally flagged Stack- and Frame Pointer manipulations. If the switch \$BOOTRST is activated, address parameters of both these switches should be identical. The above address ranges are also valid here. If the FuseBit *BOOTRST* is active and the Reset Jump of the CPU is the same as this address parameter, the first statement after this switch is also the first statement executed after a hardware reset.

{\$DEPHASE BOOTBLOCK}

This Compiler switch closes the Bootblock and is mandatory. It installs the FlashWrite drivers into the BootBlock where they must reside, because the SPM instruction only works in this section. The following code and/or statements are placed normally, i.e. they are appended to the code that was generated before the switch {\$PHASE ...}.

3.51.3 BootApplication

{\$BootApplication \$nnnn} with XMegas { \$BootApplication}

As an alternative for including the BootBlock into the application, with this switch it is possible to write an application which only contains code for the boot area and completely runs in the boot area. This switch must be placed after the Device declaration.

3.51.4 FlashWrite at Runtime

The implemented functions support the modification of single bytes or whole code pages in the Flash memory. Be aware that the recent CPUs only can read/write entire pages (32..128 words). Furthermore only words can be placed into such a page and they must be placed on an even address.

XMegas: These function are not recommended for use with XMegas. Use [FlashDownloader](#) instead.

3.51.4.1 Functions

FlashInitPage

FlashInitPage(const addr : LongWord) prepares a page for further operations. The parameter addr is a byte address and must be a constant. A manipulation of this constant at call time (e.g. addr + 2) is not possible. It is not absolute necessary that this parameter points to a page boundary. Normally addresses of ROM-constants are passed. The parameter will be placed into the variables *FLASH_ADDR* and if necessary in *FLASH_PAGE*. This function should be called each time before a new page is opened.

Procedure *FlashInitPage (const addr : LongWord);*

FlashReadPage

Reads the Flashpage, where *FLASH_ADDR* points to, from ROM into the temporary CPU Buffer. Then the buffer contains an exact copy of the Flashpage. The content of the temporary buffers now can be manipulated with *WritePage*. Attention: a *FlashProgPage* or *FlashErasePage* implies also an erase of this buffer.

Procedure *FlashReadPage;*

FlashClearPage

Erases the temporary buffer of the CPU with \$FF.

Procedure *FlashClearPage;*

FlashWritePage

Writes a word into the temporary buffer of the CPU. The variable *FLASH_ADDR* serves as an address/pointer into this buffer. Bit0 is ignored. This variable defines the destination address of this operation. Because of this an operation like:

inc(FLASH_ADDR, 2);

is allowed and also necessary in order to manipulate the buffer pointer. But bear in mind that this pointer must never exceed the page boundary, otherwise a crash is certain.

Finally this function always increments the pointer *FLASH_ADDR* by 2. Because of this the variable *FLASH_ADDR* must be set to its original value after all buffer accesses are finished.

Procedure *FlashWritePage (const parm : word);*



AVRco Standard Driver

FlashErasePage

This procedure directly erases this page in the Flash that *FLASH_ADDR* points to. The temporary buffer is automatically cleared. Note: before programming the content of the temporary buffer into the Flash with *FlashProgPage*, the target page should be erased with *FlashErasePage*. If the target page is already erased (i.e. free Flash memory) or the desired locations contain \$FF the erase can be skipped.

Procedure *FlashErasePage*;

FlashProgPage

The temporary buffer of the CPU will be programmed into the Flash memory. The variable *FLASH_ADDR* again determines the target page. The temporary buffer then automatically becomes erased with \$FF.

Procedure *FlashProgPage*;

FlashReadFuses

The Fusebits of an AVR can be read back at runtime with this function. The parameter *FuseGroup* defines the byte that must be read. Which value is valid for a specific fuse byte must be read in the datasheet of the used AVR. The parameter *FLASH_ADDR* is not used by this function.

Function *FlashReadFuses (FuseGroup : byte) : byte*;

FlashWriteFuses

The Fusebits of an AVR can be written at runtime with this function. The parameter *FuseGroup* defines the fuse byte which must be written. Which value is valid for a specific fuse byte must be read in the datasheet of the used AVR. At this time only the LockByte is accessible and also bits can be programmed to 0 but not to 1. The parameter *FLASH_ADDR* is not used by this function.

Procedure *FlashWriteFuses (FuseGroup, fsBits : byte)*;

FlashCopyF2R

If parts of already programmed pages must be changed, e.g. only a byte, then a Page Erase is still mandatory. The content of this page then must be saved, the page must be erased, the changes must be done in the saved page and then this page must be re-programmed into the Flash.

The function *FlashCopyF2R* copies the flash page where *FLASH_ADDR* points to into the buffer in RAM where the parameter *p* points. The RAM buffer must be capable (size) to hold one flash page (mega128 = 256bytes).

Now the application can manipulate the content of the RAM buffer.

Procedure *FlashCopyF2R (p : pointer)*;

FlashCopyR2F

This function copies flash page previously saved with *FlashCopyF2R* into the temporary buffer of the CPU. This buffer now can be restored into the flash page with *FlashProgPage*.

Procedure *FlashCopyR2F (p : pointer)*;

const

StringC[\$10200] : string = 'AF_ZKG';

{ \$IDATA }

var

ar : array[0..255] of byte;

*{ \$Validate FLASHERASEPAGE * }*

*{ \$Validate FLASHPROGPAGE * }*

{ \$Validate FlashCopyF2R }

{ \$Validate FlashCopyR2F }

{ \$PHASE BootBlock \$0F000 }

Procedure *FlashProg*; // at \$0F000

begin

```
// dummy
end FlashProg;
{$DEPHASE BootBlock}

// main
begin
  FlashInitPage (ADDR(StringC));
  FlashCopyF2R (@ar);
  // ... do the changes in "ar"
  FlashErasePage;
  FlashCopyR2F (@ar);
  FlashProgPage;
  ...
```

3.51.5 Usage of FlashWrite

3.51.5.1 Implementation

Calls to **FlashWrite** functions can be done from everywhere in the program. The only condition is that **after** the compiler switch

```
{ $DEPHASE BOOTBLOCK }
```

there are no further FlashWrite calls. Because of this there are three possible implementations of FlashWrite:

1. FlashWrite in common Code Area

Call the driver functions before the definition of the BootBlock:

```
Procedure WriteTest;
begin
  FlashInitPage (Addr (KeyLookUp));
  FlashReadPage;
  FlashErasePage;
  FlashWritePage ($5857);
  inc (FLASH_ADDR, 2);
  FlashWritePage ($5A59);
  FlashProgPage;
end WriteTest;
```

```
Procedure ABC;
begin
  ...
end;
```

```
{ $PHASE BootBlock $01F80 }
{$DEPHASE BootBlock}           // inserts Flash drivers at $1F80
```

2. FlashWrite within the BootBlock

```
{ $PHASE BootBlock $01F80 }
Procedure WriteTest;
begin
  FlashInitPage (Addr (KeyLookUp));
  FlashErasePage;
  FlashWritePage ($5857);
  inc (FLASH_ADDR, 2);
  FlashWritePage ($5A59);
```



AVRco Standard Driver

```
FlashProgPage;  
End WriteTest;  
{ $DEPHASE BootBlock }
```

3. Validation

```
// validate the used driver functions before the BootBlock is defined  
{ $Validate FLASHREADPAGE }  
{ $Validate FLASHERASEPAGE }  
{ $Validate FLASHPROGPAGE }  
{ $Validate FLASHWRITEPAGE }
```

```
{ $PHASE BootBlock $0F000 }  
{ $DEPHASE BootBlock }
```

```
(* main *)  
FlashInitPage (Addr (KeyLookUp));  
FlashErasePage;
```

Type 3 is the most versatile way because the driver calls can be from nearly anywhere in the application.

Notes

As described above, a write access into the temporary buffer of the CPU must be word aligned (even addresses) and the parameter must be a word. This buffer always resides at page boundaries (mega16: 128Bytes or 64 words).

All operations use the internal variable *FLASH_ADDR*. When manipulating this value pay attention to the page boundaries. Do not exceed them. If it is necessary, use the procedure *FlashInitPage*.

Also take care of the correct settings of the BOOTSZx fusebits. With an incorrect setting every Flash operation of the driver is inhibited.

3.51.5.2 Rewrite a Page completely

No flash bit or byte can be written from 0 to 1. If this is necessary then the entire page must be erased to \$FF. If a Flash page must be completely rewritten or in a completely filled page a single byte must be changed a trick must be used. In the early times (Mega163) it was possible to read a page into the temporary buffer, then erase the flash page, then manipulate the buffer and then write back the buffer into the flash. Now this processing is not possible anymore. All current AVR's delete their temporary buffer if a page erase or a page program is executed. The previously read back buffer is emptied before it is possible to write it back to the flash.

A work-around is to read the desired flash page into the RAM and not into the temporary buffer. Then execute the page erase, then copy the RAM buffer content into the temporary buffer and then execute the page program. This is supported by the driver functions *FlashCopyF2R* and *FlashCopyR2F*

```
Program WriteTest;
```

```
{ $BOOTRST $0F000 }      { Reset Jump to $0F000 }
```

```
Device = mega128, VCC=5;
```

```
Import FlashWrite;
```

```
From System Import longword;
```

```
Define
```

```
ProcClock   = 16000000;    {Hertz}  
StackSize   = $40, iData;  
FrameSize   = $80, iData;
```

Implementation

```
{ $IDATA }
{-----}
{ Type Declarations }
type

{-----}
{ Const Declarations }
const
    StringC[$10200] : string = 'AF_ZKG';
    // example how a fixed const can be
    // placed into the bootblock below
    BootConst[$1FFFC] : word = $1234;

{-----}
{ Var Declarations }
{ $IDATA }
var
    bb : byte;
    ww : word;

(*****
(* Because all system Flash Write functions must be *)
(* placed into the BootBlock part of the Flash we must *)
(* import the used functions by the "validate" switch *)
(*****)

{ $Validate FLASHREADPAGE }
{ $Validate FLASHERASEPAGE }
{ $Validate FLASHPROGPAGE }
{ $Validate FLASHWRITEPAGE }
{ $Validate FlashReadFuses }
{ $Validate FlashWriteFuses }

// The definition of the correct BootBlock location
// is a must because all FlashWrite operation can only
// be executed from this area.

// "FlashProg" is an unused dummy function.
// It is not necessary but shows that there
// can be also user functions in the bootblock

{ $PHASE BootBlock $0F000 } // !!!!!!!!!!!!!!!
Procedure FlashProg;           // at $0F000
begin
end FlashProg;
{ $DEPHASE BootBlock }        // !!!!!!!!!!!!!!!

{-----}
{ Main Program }
begin
    // read the fuse and lockbits
    bb:= FlashReadFuses (1);    // lock bits
    bb:= FlashReadFuses (0);    // Fusebits 0
    bb:= FlashReadFuses (3);    // Fusebits 1
    bb:= FlashReadFuses (2);    // Fusebits 2

    // Flash manipulations
    FlashInitPage (ADDR (StringC));

    // either do a FlashReadPage
```



AVRco Standard Driver

```
FlashReadPage;  
// or do a FlashErasePage  
FlashErasePage;  
// but not both at the same time
```

```
ww:= FLASH_ADDR;  
FlashWritePage ($4203);  
inc (FLASH_ADDR, 32);  
FlashWritePage ($4443);  
FLASH_ADDR:= ww;  
FlashProgPage;
```

```
loop  
  Nop;  
endloop;  
end WriteTest.
```

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\FlashWrite`

3.51.5.3 FlashDownLoad at RunTime

In contrast to *FlashWrite* there are no functions published which the application can call, but a DownLoad Monitor is implemented. With the help of this Monitor it is possible to download the entire Flash and burn it into the Flash memory, except the Boot area. The only condition is that there is hardware that supports the transfer, and the user provides some support routines. With this, remote Updates of the firmware by PC, modem etc. are possible without problems.

To use FlashDownLoad the library group **FlashWrite** must be imported:

Import SysTick, FlashWrite, ...;

Please observe that this **import must be placed at least at the second position** in the importlist.

The programmer must provide 4 functions:

FlashLoaderInit, FlashLoaderRecv, FlashLoaderTransm and (optional) *FlashLoaderExit*.

Support functions

FlashLoaderInit

This procedure must be provided by the user. Its job is to initialise the interface that should be used by the DownLoad Monitor. There is no restriction of the kind of the interface. Its only job is to send and receive bytes. Please note that interrupts are not possible here.

This function is a Callback, so this is true: **no local variables**

Procedure FlashLoaderInit;

FlashLoaderRecv

This procedure must be provided by the user. Its job is to fetch a byte from the interface and pass it back to the caller as the result. Because this procedure should be written in assembler the result must be put into register `_ACCA` (R17). A RETURN statement is not implemented.

This function is a Callback, so this is true: **no local variables**

Procedure FlashLoaderRecv;

Attention:

If additional registers must be used, then the registers `_ACCB/R16`, `_ACCALO/R18`, `_ACCAHI/R19`, `_ACCCLO/R30` and `_ACCCHI/R31` must not be used unless they are saved and restored by the user.

FlashLoaderTransm

This procedure must be provided by the user. Its job is to pass a byte to the interface and transmit it. Because this procedure should be written in assembler the parameter is passed to the procedure in register `_ACCA` (R17). This function is a Callback, so this is true: **no local variables**

Procedure FlashLoaderTransm;

Important:

If additional registers must be used, then the registers `_ACCB/R16`, `_ACCALO/R18`, `_ACCAHI/R19`, `_ACCCLO/R30` and `_ACCCHI/R31` must not be used unless they are saved and restored by the user.

FlashLoaderExit

This procedure can be provided by the user as an option. Its job is to do a System Restart after a successful download. This Restart normally is done with a JMP/RJMP to the Reset Vector at address `SYSTEM.VectTab`. If the system doesn't find this procedure the loader does a JUMP to `0000h = SYSTEM.VectTab` if it terminates.

Procedure FlashLoaderExit;

Remarks:

These user-supplied functions can reside anywhere in the ROM/Flash area, but it is convenient to place them into the BOOTBLOCK so the Download cannot overwrite them.



AVRco Standard Driver

Usually the CPU-internal I/Os e.g. UART, SPI, CAN, I2C or Ports are used for the communication with the Host. But also external interfaces can be used without problems.

It is impossible to run these drivers with interrupts as the vector table (or at least the corresponding interrupt drivers) are overwritten during download.

Loader identification

In most cases the Loader Program is programmed once into the target CPU and remains unchanged at least for a long time. Because of this it's a good idea to place the hardware revision (or similar) of the board or system into it. This 16bit number can be interrogated before a download to make sure that the downloaded firmware is executable on this hardware. This number is included into the machine code as an immediate constant when the BootLoader is generated. This must be done by the definition of a global constant named „DownloaderID“:

Const DownLoaderID : word = 010213;

If this constant is not defined this value is stored with \$0000 in the Loader. The ID-number can be recalled at runtime from the loader using the command ‚i‘. The Host requests the DownLoader ID.

There are upto 4 possible ID-words which then are sequential uploaded with the i-command:

const

DownLoaderID : word = \$1234;

DownLoaderID1 : word = \$5678;

DownLoaderID2 : word = \$9ABC;

DownLoaderID3 : word = \$DEF0;

Security

In order to disable unauthorized read back of the Flash or EEPROM content through the Loader there must be an optional password defined:

Const DownLoaderPWD : word = \$1A2B;

If a password is defined the upload tool must always provide this password if any upload command is used.

3.51.5.4 FlashDownloader Functions

FlashDownloader

The Download Monitor is hidden behind this procedure. Calling this procedure disables the global interrupt, the StackPointer is initialised to top of Ram, as is the normally unused FramePointer. The variable **FLASH_ADDR** and possibly **FLASH_PAGE** are initialised. After this follows the buffer for Up- and Downloads. The size of this area is an array[0..PageSize-1] of byte; The RAM memory between this array and the Frame is left free and is unused.

After the internal initialisation of the Loader a call to the user supplied procedure *FlashLoaderInit* follows. If the Loader gets back control over the system, it loops in its command loop. This means it calls continuously the function *FlashLoaderRecv* and interprets the received commands and parameters. Each command is acknowledged by the Loader at least with a handshake byte through the procedure *FlashLoaderTransm*.

If the Loader receives the Exit command it jumps to the user supplied optional procedure *FlashLoaderExit* and closes itself. If the system doesn't find this procedure the loader does a JUMP to 0000h if it terminates.

Procedure FlashDownloader;

FlashDownloader Commands

The Loader knows several commands. They consist of an ASCII character and possibly some following parameters. Each commands is answered by a CR (\$0D or #13) after execution. Exceptions from this are Block Transfer Commands. Commands are always send from Host to Loader. The Loader only executes them. Except for Initialisation each action must be initiated by the Host.

The commands of the Host can set a Page Address, up or download a page, erase a page in the Flash, read a page from Flash into the buffer and burn the buffer into the Flash.

A substantial part is the request for the Loader State/Parameter with the ? command. With this the Host finds out which CPU-ID the target has, the size of the Flash Pages and where the BootBlock starts. All parameters that determine sizes and addresses are word parameters. This means that the parameter itself does not have to be a word, but it counts in words. If, for example, the Loader returns a Page size of 64, it means 64 words (or 128 bytes). The same is true for the Address command of the Host. If the command A is followed by a 256, the Loader must change its Page address to 256. This is the address 512 when counted in bytes. The Host program must observe this absolutely.

Example program:

an example is in the directory `..\E-Lab\AVRco\Demos\SelfProg`



AVRco Standard Driver

3.51.6 Host Program

The installation of AVRco contains a PC-based DownLoader Program **FlashLoader.exe**.
This is documented in **..E-Lab\DOCs\DocuTools.pdf** in the chapter "*Flash Down Loader / Writer*".

Here only the communication protocol again:

FlashLoader Command List

- ? Host requests Loader ID.
Loader responds with
FD FlashDownLoader
- A** Host sends page adr in word representation. All action relate to this Page
aa1 page addr loByte
aa2 page addr hiByte
Loader responds with
CR Command executed
- B** Host sends EEprom adr in byte representation and EEprom data (byte).
aa1 EEprom adr loByte
aa2 EEprom adr hiByte
data 1 Byte into EEprom
Loader programs this byte into the EEprom and responds with
CR Command executed
- C** Host sends EEprom adr in byte representation.
aa1 EEprom adr loByte
aa2 EEprom adr hiByte
If an (optional) password is given (see below) the Host must then send the correct password:
pw1 Password loByte
pw2 Password hiByte
Loader reads the EEprom byte from this address in the EEprom and sends it back as the response
- D** Host stores a new Page of Flash data into the Loader's buffer
XMega 2bytes blocksize in Bytes lo/hi byte
data following (**ps** x 2) bytes = Pagesize x 2 (mega8..meg16 = 128bytes)
Loader stores this Page into its array and builds an 8Bit checksum by adding all bytes.
Loader sends the computed checksum as a result of the operation to the Host:
cc Checksum
- E** Host requests erase of the actual Page
Loader erases the actual Page in the Flash and responds with
CR Command executed
- I** Host requests Loader Info.
Loader responds with
I Info ID = Loader-ID, see below
id1 hiByte Processor ID
id2 midByte Processor ID
id3 loByte Processor ID
ps words pro page
bs1 Bootblock start addr lobyte \
bs2 Bootblock start addr hiByte / = Bootblock start addr in word count
bs3 Bootblock start addr extbyte / = Bootblock start addr in word count if Flash > 128KB
- P** Host request programming of the Loader's buffer into the Flash
Loader overwrites the actual Page in the Flash with the content of its buffer. A verify is not implemented.
The Loader responds with:
CR Command executed

- U** **XMega** UserRow programming.
CR Command executed
- aa1** page adr loByte
aa2 page adr hiByte
aa3 Page Extend Byte
 The Loader responds with:
CR Command executed
- data** following (**ps** x 2) bytes = Pagesize x 2
 Loader stores this Page into its array and builds an 8Bit checksum by adding all bytes.
 Loader sends the computed checksum as a result of the operation to the Host:
cc Checksum
- R** Write downloaded page into the UserRow.
 The Loader responds with:
CR Command executed
- W** Host sends a relative page addr in word representation. Then a word follows which the Loader must write into its buffer with the use of this address
aa page addr relative (Byte!!)
ww1 loByte
ww2 hiByte
 Loader responds with
CR Command executed
- X** End of communication. The Loader Monitor jumps to the optional user supplied procedure *FlashLoaderExit*. If the system doesn't find this procedure the loader does a JUMP to 0000h if it terminates.
 There is no response from the Loader.

3.51.7 Boot Area, Optimiser and re-compile

If the Optimiser is used or the Boot area contains large drivers there can be problems that the function addresses expected by the application can be changed after a re-compile. This concerns the application because the Boot itself cannot be changed by a self-flash. The result is that the application jumps to the now new calculated addresses in the Boot, but these have not changed in the Boot because they are "constant".

A remedy is here to implement Boot Traps like described in the Compiler Manual. So Boot Traps must be imported:

From System Import Traps;

This import places a jump table into the Boot area. Now the procedures needed by the application must be placed as Traps into the Boot Block. Basically the first procedure in the Boot Block should be the Boot Test function (Procedure BootTest). So the Trap functions must be placed after it:

```
{ $PHASE BootBlock $01F00 }
```

```
Procedure BootTest;
```

```
begin
```

```
...
```

```
end;
```

```
Procedure BootEntry; Trap;
```

```
begin
```

```
FlashDownloader;
```

```
end;
```

```
...
```

```
{ $DEPHASE BootBlock }
```

Because at least a Boot entry for re-flashing for the application is necessary but the Boot function "FlashDownloader" maybe not reachable securely the Trap function "BootEntry" now provides an entry point that is always secure. Application:

```
if xxx then
```

```
BootEntry;
```

```
endif;
```



AVRco Standard Driver

3.52 BootApplication and MainApplication

As an alternative for including the BootBlock into the application, with this switch

{\$BootApplication \$nnnn} with **XMegas** **{\$BootApplication}**

it is possible to write an application which only contains code for the boot area and completely runs in the boot area. This switch must be placed after the Device declaration.

With standard AVR applications the Interrupt Vector Table always resides on address \$0000 in the code area (Flash). The same also is true with the Codestart. This means that also the generated code starts with address \$0000, but more exactly immediately behind the vector table.

With this switch the system can be forced to start the address generation and vector table in the boot area. So now it is possible to build an application which completely runs in the boot sections of the controller and still has access to all of the system resources and drivers because they are also placed into the boot section. Also interrupts are supported here. The address parameter for this switch is always a word address!

A main Application can be joined with the boot application in the **define_fuses** block:

AddApp instructs the programmer to load additional hexfiles build by another project:

```
AddApp = 'pathname\projectname';
```

With the **XMegas** **only** the BootApplication mode is supported. The XMega drivers are much more complex compared to the AVRs and it makes sense to have the complete runtime system and the necessary drivers in the boot area.

Support function for the Boot Application to enter the standard application. This procedure also disables the an active WatchDog, disables the Interrupts and switches the Interrupts from Boot-vectors to the Main-vectors (\$0000).

Procedure *Application_Startup*;

Support function for directly switching the vector tabels:

Procedure *SetVectTabBoot*(boot : boolean);

Support function for directly starting and jumping from MainApp into the BootApp:

Procedure *BootStart*;

The **MainApp**, which always begins at start address \$0000 and also its vector table starts at \$0000, now can communicate with the BootApp with the help of Traps. To implement this the **BootApp** must import Traps and export the desired functions. Sample in a BootApp:

From System **Import** *Traps*;

...

function *BootFunction* : boolean; **Trap**;

begin

// do anything

return(true);

end;

procedure *BootProc*(x : byte); **Trap**;

begin

// do anything

end;

AVRco Standard Driver



If there are any Traps defined in a **BootApp** Compiler/Assembler builds a file "BootAppName.traps". In order to use these Traps/Jumps in the **MainApp** this file must be imported in the MainApp.

From System Import Traps;

ExternalTraps 'BootAppName.traps'; // max 16 ext traps !

Define

...

For information a content of a Trap file:

```
BOOTSTART $040000
1479 20241 .TRAPTAB
1480 20241 SYSTEM.TrapTab:
1481 20241 0200FE .ADDR BootApp.BootFunction //function BootFunction : boolean; Trap;
1482 20242 020105 .ADDR BootApp.BootProc //procedure BootProc(x : byte); Trap;
1483 20244 000000 .ADDR 0
1484 20245 000000 .ADDR 0
1485 20247 000000 .ADDR 0
1486 20248 000000 .ADDR 0
1487 2024A 000000 .ADDR 0
1488 2024B 000000 .ADDR 0
1489 2024D 000000 .ADDR 0
1490 2024E 000000 .ADDR 0
1491 20250 000000 .ADDR 0
1492 20251 000000 .ADDR 0
1493 20253 000000 .ADDR 0
1494 20254 000000 .ADDR 0
1495 20256 000000 .ADDR 0
1496 20257 000000 .ADDR 0
```

Now the compiler knows these functions, their parameters and the absolute jump-addresses into the Boot. And now the MainApp can call these Trap functions:

```
if BootFunction then
// ...
endif;
BootProc($33);
```

Attention:

In the BootApp then there are all interrupts disabled! This means that for example all functions using TimeOuts can not work because the interrupts only serve the MainApp. So one must carefully check in the MainApp and Traps to avoid dead-locks.

For security reasons the interrupts become disabled before a jump into a Boot function. After the return from the Boot area the interrupts are set to their previous state.

Furthermore one must take in mind that a re-compile of the MainApp is possible any time. But after a re-compile of the BootApp also the MainApp must be re-compiled.

Sample applications can be found in the Demos directories in "BootApp"



AVRco Standard Driver

3.52.1 XMega FlashLoader

The XMegas are completely different in the Bootloader compared to the “normal” Megas. Concerned are the Flash operations in the Boot but also the fact that the XMegas have a separate Boot Flash. Because of this the Phase/Dephase mode is not supported here but the separate BootApp application, defined by the switch **{\$BootApplication}**. This switch must be placed immediate after the Device Define. Find more infos in the chapter above.

Furthermore some constants and functions are a “must”:

```
const
    // this constant must be the same as in the Main app
    DownLoaderID : word    = $1234;           // mandatory constant

procedure FlashLoaderInit;                    // mandatory
begin
end;

function FlashLoaderRecv : byte;              // mandatory
begin
    return(SerInpC0);
end;

procedure FlashLoaderTransm(arg : byte);      // mandatory
begin
    SerOutC0(arg);
end;

// mandatory
procedure FlashLoaderExit;
begin
    if Application_Valid then                // Flag is cleared at Loader start and set by the
                                                // Bootloader if successful
        EEprom[EEpromEnd]:= $00;           // validate application for Boot usage
    Endif;
    mDelay(100);
    HardwareReset;                         // restart with a jump into the Boot
end;
```

The constant **DownLoaderID** must be present and must be the same in the BootApplication and in the downloadable Main.

The procedure **FlashLoaderInit** is always called at the Download Start and can be used to implement some init operations.

The procedure **FlashLoaderRecv** fetches a byte from the download interface.

The procedure **FlashLoaderTransm** passes a byte to the download interface.

The procedure **FlashLoaderExit** must flag a successful Download. At best this is done by a flag in the EEprom which can be accessed by both, the Downloader and also the downloaded Main. In the Main of the Bootloader with each restart this flag must be checked whether the last Download was successful. If ok then the Loader jumps directly into the Main application. If not ok then it waits for a new download. By a **HardwareReset** in the LoaderExit a complete restart is forced which leads to a check of this flag in the boot.

But therefore the **BootRst** fuse always must be active:

```
Define_Fuses
    Override_Fuses;
    COMport    = USB;
    ProgMode   = PDI;
    LockBits0  = [];
    FuseBits0  = [];
    FuseBits1  = [];
    FuseBits2  = [BootRst];                    // mandatory !!
    // Brown-out is obligatory with USB !!!
    FuseBits5  = [BODACT0, BodLevel0, BodLevel1, BodLevel2];
```

AVRco Standard Driver



The Main in the BootLoader must execute some checks:

```
{-----}
{ Main Program }
{$IDATA}
begin
  // optional a port pin can be checked for a forced download
  // if Pin.x = false then ...
  // ...
  if EEprom[EEpromEnd] = $00 then
    // if a Download failed or the app was never programmed then there is no $00
    // if the main app forces a download then the last byte in the EEprom must be $FF
    //
    Application_Startup; // Jump into the Main Application
  endif;
  EnableInts($87);
  loop
    FlashDownloader;
    // Downloader itself does an exit to the main app if
    // a valid main has been downloaded
  endloop;
end XMega_BootAppSer.
```

The Main application must also provide the same DownloaderID as the boot:

```
const
  // this constant must be the same as in the Main app
  DownloaderID : word = $1234; // mandatory constant
```

If the **Main Application** runs then it is sure that the last download was valid. In the Main App it is possible to force a new Download:

```
EEprom[EEpromEnd] := $FF; // invalidate application for Boot usage
mDelay(100);
HardwareReset; // restart with a jump into the Boot
```

Forcing a **HardwareReset** a complete system restart is executed which always runs the Boot. In the Boot this flag is checked (described above) and because it is now $\neq$ \$00 immediately the FlashDownloader is called. An essential condition for this is that with programming the BootApp also the fuse **BootRst** is activated.

USB download mode

Please note that an USBprodName can not be defined here. This define is internally fixed to "BootLoader".

```
Import USBboot, FlashLoader, ...;
Define
  ...
  USBmanufact = 'E-Lab Computers'; // max 31 bytes
  USBpid      = 30;
  USBvid      = $2345;
  USBprodRel  = 201;
  USBcurrent  = 200;
  USBvBUS     = PortB.7; // port and pin
  // USBvBUS   = none; // not used
```

Building a complete Boot Application

1. Build and compile the BootApp
2. Build and compile the MainApp.
3. With AVRprog make a PAC-file from the MainApp.
4. Program the BootApp. If USB Boot used build the USB drivers with the tool USB Inf/Sys Builder
5. Start the target system. It now expects a firmware download of the MainApp
6. With the tool FlashDownloader the generated PAC-file must be downloaded

Sample applications with a **COMport** download can be found in the Demos directories in "XMega_Boot". In "XMega_Boot" you can also find the corresponding MainApp.

Sample applications with **USB** download can be found in the Demos directories in "XMega_BootUSB". In "XMega_BootUSB" you can also find the corresponding MainApp.



AVRco Standard Driver

3.53 Device Drivers

Overview

Often there is a need for additional memory because the internal memory for variables of the SingleChips (RAM/EEPROM) is too small. This problem can be solved in two ways: either change the current CPU for a larger one with more memory or use external memory. The first possibility often is not feasible because of price or size. On the other hand, which single chip has 32kByte of internal EEprom?

External memory extension with complete addresses and data lines is often impossible because of space and ports usage. So in many cases one must use external devices with 3-wire interfaces, e.g. I2C, SPI or microWire. But these chip have always different functionality so a common interface can't be build or used and the Compiler normally can't place variables into this area and an addressing and/or read or write in HLL is impossible.

With current Compilers therefore it was necessary that the user had to organize and build the accesses for himself, which is not an easy task. The external memory commonly was treated as an array of byte. Because of this there was no direct relationship between the external address and the type stored in the external memory. An insertion of a single byte between others results in a disaster if the whole program was not checked for changes dependent on this insertion.

It's a good idea to overlay a structure on this "exotic" memory, just like a "normal" RAM. The Compiler can do that for you. The user simply must provide the physical access with certain procedures and functions. They are called a "Device Driver".

These Device Drivers work byte oriented as opposed to the Compiler, which works type oriented. The user's program can access these vars in external memory like those that reside in the internal memory of the CPU (RAM, EEPROM).

Another way of accessing external memory can be implemented with block transfers. But here the individual access to types etc. is lost again, but for some applications it can be a useful technique. See below under **BlockDevice**.

3.53.1 Organisation

1. The Compiler must be informed about such a Device: **Import** UserPort ;
2. The size in bytes of the external memory must be defined: **Define** UserPort = nnn;
3. A compiler switch **{ \$UDATA }** locates the following var declarations into the external device. An additional switch e.g. **{ \$IDATA }** switches back to the internal area.
4. A procedure that initialises the device must be written. This procedure is called once at system start or reset : **UserDevice** UsrDevIni
5. A procedure that writes single bytes into the device is required: **UserDevice** UsrDevOut
6. A function that reads single bytes from the device is required: **UserDevice** UsrDevIn

Import aa, bb, UserPort;

This import directive forces the Compiler to provide a UserPort and the related support.

Define UserPort = nnn;

With this define the size in bytes of the User Devices is declared

{ \$UDATA }

All following variable declarations after this compiler switch are located into the UserDevice.

3.53.2 Formal Declarations

UserDevice *UsrDevIni*;

If necessary, port and device inits must be done here.

UserDevice *UsrDevOut* (**const** *adr* : word; **const** *outp* : byte);

The system passes a 16Bit address and a byte to this procedure. The procedure the stores the byte at the desired address into the device.

UserDevice *UsrDevInp* (**const** *adr* : word) : byte;

The system passes a 16Bit address to this procedure. The procedure the reads the byte at the desired address from the device and returns it to the system.

The naming for the 3 procedures above is **mandatory**.

Program *AVR_UsrDataI2C*;

Device = 90S8515;

Import SysTick, UserPort, I2Cport;

From System **Import** longword;

Define

```

    ProcClock  = 8000000;    {Hertz}
    SysTick    = 10;         {msec}
    StackSize  = $0020, iData;
    FrameSize  = $0010, iData;
    UserPort   = 2048;        // define Mem expansion size
    I2Cport    = PortB;       {24C16}
    I2Cdat     = 0;
    I2Cclk     = 1;
```

Implementation

```

{$IDATA}
{-----}
{ Type Declarations }
```

type

```

    ptB  = pointer to byte;
    ptW  = pointer to word;
```

```

{-----}
{ Const Declarations }
```

const

```

    I2Cadr  = $50;
```



AVRco Standard Driver

```
{-----}
{ Var Declarations }
var
    Timer1      : SysTimer8;
    bb          : byte;
    ww          : word;
    testB       : Byte;
    testCh      : char;
    testW       : word;
    testI       : integer;
    testL       : longword;
    testStr     : string[8];
    st          : string[5];
    wxyz        : Word;
    pp          : ptw;

{$UDATA}                      // place vars into User Device
var
    usrb        : byte;
    usri        : integer;
    usrl        : longword;
    usrArr      : array[0..10] of byte;
    uStr        : string[8];
    uRec        : record
        ub : byte;
        uw : word;
        uSt: string[4];
    end;
    pwDirT1     : ptw;
    pbEepAdr1   : ptb;
    prDir       : pointer to word;

{$IDATA}                      // reset to internal RAM
{-----}
{ functions }

(* for a UserDevice the following 3 function must be implemented *)
UserDevice UsrDevIni;
begin
    (* is called at System Init Time *)
    (* initialize Device Hardware *)
    (* for I2C nothing required *)
end;

UserDevice UsrDevInp (adr : word) : byte;
var ok : boolean;
    inp : byte;
begin
    // DisableInts;

    (* because of a possible previous write access *)
    (* the I2Cout function can fail up to 10msec *)
    (* and a timeout loop must be implemented *)
    (* don't forget your watchdog *)
    SetSysTimer (Timer1, 2);
    repeat
        ok:= I2Cout (I2Cadr +hi (adr), lo (adr));    // set the read address
    until isSysTimerZero (Timer1) or ok;
```

AVRco Standard Driver



```
if ok then
  I2Cinp (I2Cadr, inp);           // read the byte
  return(inp);
else
  // error ...
endif;

// EnableInts;
return($ff);
end;
```

```
UserDevice UsrDevOut (adr : word; outp : byte);
var ok : boolean;
begin
  // DisableInts;
  (* because of a possible previous write access *)
  (* the I2Cout function can fail up to 10msec *)
  (* and a timeout loop must be implemented *)
  (* don't forget your watchdog *)
  SetSysTimer(Timer1, 2);
  repeat
    ok:= I2Cout (I2Cadr +hi (adr), lo (adr), outp);
  until isSysTimerZero(Timer1) or ok;
  // EnableInts;
  // if not ok then ....
end;
```

```
{-----}
{ Main Program }
```

```
begin
  uStr[1]:= 'A';
  // uStr:= st;           // not implemented
  // uStr:= 'ABCD';       // not implemented
  // TestStr:= uStr;      // not implemented
```

```
  uRec.uw:= $1234;
  uRec.ub:= $AA;
  uRec.uSt[2]:= 'X';
```

```
  testB:= uRec.ub;
  testW:= uRec.uw;
  testCh:= uRec.uSt[2];
```

```
  usrb:= 155;
  usri:= -155;
  usrl:= 123456;
  usrArr[3]:= 44;
```

```
  testB:= usrb;
  testI:= usri;
  testL:= usrl;
  testB:= usrArr[3];
```

```
  loop
    NOP;
  endloop;
end AVR_UsrDataI2C.
```



AVRco Standard Driver

The UserDevice/UDATA supports also record, array and string copy functions:

```
UserRec:= Record1;  
Record1:= UserRec;  
UserArr:= Array1;  
Array1:= UserArr;  
UserStr:= String1;  
String1:= UserStr;
```

Not possible are Pointer to Record, Array, String.
Also a string-concat is not possible with the User Device.

3.53.3 BlockDevice

Contrary to a so called UserDevice a BlockDevice doesn't work byte or type oriented, but with entire blocks. This means that always whole memory areas are copied. The copy operation must also be executed by a user provided service function. The source or destination of such a copy operation is defined by the construction of the service routine and can be arbitrary, also external devices are possible.

The system function **ReadBlock** at least needs one concerned user function. The same is true for the **WriteBlock** function. A specific Import or Define is not necessary.

The **ReadBlock** uses a procedure/function with the following principles. The name is arbitrary. If it's absolutely secure, that all bytes can be completely copied, only a procedure is needed.

```
Procedure DriverInpP (adr : pointer; size : word);  
begin  
  (* very simple implementation of a Block Device *)  
  (* because the calling library block function *)  
  (* doesn't receive a transfer count from here *)  
  (* this procedure !must! transfer "size" bytes *)  
  (* The destination address is "adr" *)  
  (* The source adr is the job of this service *)  
  CopyBlock (@source, adr, size);          // possible action  
end;
```

If it's possible that the user provided implementation can not generate the desired count of bytes, a function is needed which returns the exact byte count:

```
Function DriverInpF (adr : pointer; size : word) : word;  
begin  
  (* very simple implementation of a Block Device *)  
  (* requested transfer count in "size" *)  
  (* actually transferred bytes count returned *)  
  (* The destination address is "adr" *)  
  (* The source adr is the job of this service *)  
  
  CopyBlock (@source, adr, size);          // possible action  
  Return(size); // actual transferred byte count  
end;
```

The system function **ReadBlock** now can be used with one of the above function/procedure as an argument:

```
(* read-in the var arr2 with max count2 bytes      *)
(* count2 is limited to sizeof(arr2)                *)
(* return the number of bytes read in result         *)
```

```
Count2:= 3;
result:= ReadBlock (DriverInpF, arr2, count2);
result:= ReadBlock (DriverInpP, arr2, count2);
```

```
(* read-in the var arr2 with the sizeof(arr2)      *)
(* return the number of bytes read in result         *)
```

```
result:= ReadBlock (DriverInpF, arr2);
result:= ReadBlock (DriverInpP, arr2);
```

The system function **WriteBlock** uses a procedure/function in the following manner. The name is arbitrary. If the user provided routine can ensure that all desired bytes can be written, the transfer can be done in a procedure like this:

Procedure DriverOutP (adr : pointer; size : word);
begin

```
(* very simple implementation of a Block Device *)
(* because the calling library block function    *)
(* doesn't receive a transfer count from here    *)
(* this procedure !must! transfer "size" bytes   *)
(* The source address is "adr"                  *)
(* The destination adr is the job of this service *)
```

```
CopyBlock (adr, @dest, size);           // possible action
end;
```

If it's possible that the user provided implementation cannot write the desired count of bytes at once, a function is needed which returns the exact byte count written:

Function DriverOutF (adr : pointer; size : word) : word;
begin

```
(* very simple implementation of a Block Device *)
(* requested transfer count in "size"             *)
(* actually transferred bytes count returned      *)
(* The source address is "adr"                   *)
(* The destination adr is the job of this service *)
```

```
CopyBlock (adr, @dest, size); // possible action
return(size);                // actual transferred byte count
end;
```



AVRco Standard Driver

```
(* write-out the var arr1 *)  
(* pass the numbers of bytes to write in count1 *)  
(* count1 is limited to sizeof(arr1) *)  
(* return the number of bytes written in result *)
```

```
count1:= 3;  
result:= WriteBlock(DriverOutF, arr1, count1);  
result:= WriteBlock(DriverOutP, arr1, count1);
```

```
(* write-out the var arr1 with the sizeof(arr1) *)  
(* return the number of bytes written in result *)
```

```
result:= WriteBlock (DriverOutF, arr1);  
result:= WriteBlock (DriverOutP, arr1);
```

A typed pointer can also be used as the compiler knows the structure and therefore the size of the referenced variable.

```
result:= WriteBlock (DriverOutF, Ptr^, count1);  
result:= WriteBlock (DriverOutP, Ptr^, count1);  
result:= WriteBlock (DriverOutF, Ptr^);  
result:= WriteBlock (DriverOutP, Ptr^);
```

Notes



AVRco Standard Driver

Notes

©1996-2018 ***E-LAB Computers***

Grombacherstr. 27
D74906 Bad Rappenau

Tel. 07268/9124-0
Fax. 07268/9124-24

Internet: www.e-lab.de
e-mail: info@e-lab.de
