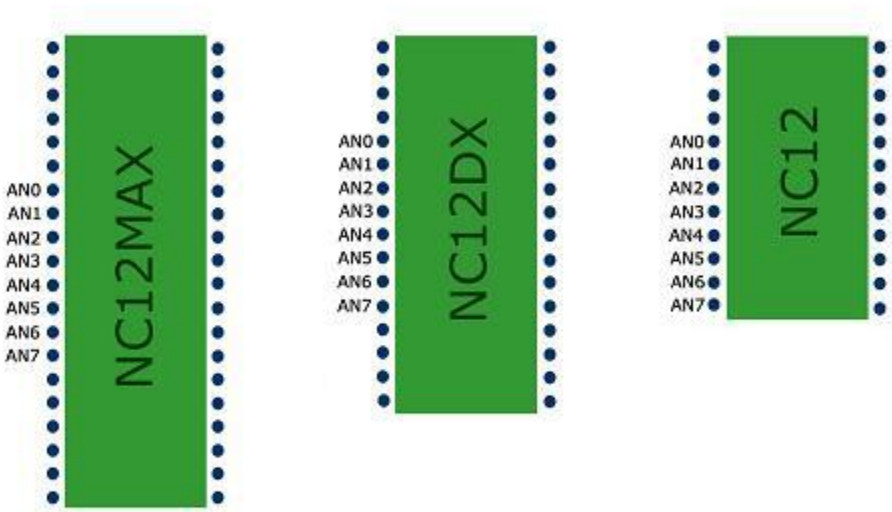
ADC

Analog-to-Digital Converter Subsystem

Version:
1.0.0

Targets:
Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
ADC`(const <Pins>)` One or more analog input(s) can be specified in the constructor of an ADC object.

Object Function Summary
ADC_Read`(const <Pin>, out byte/word result)` Get a value from an analog pin.
ADC_Start`(in byte wait, in byte mode)` Start the Analog to Digital Converter

Class Function Summary
ADC_Done`(out byte done)` Check if the ADC has finished the conversion.

Constructor Function Detail

ADC

ADC(const <Pins>)

More than one analog input can be specified in the constructor of an ADC object, as long as the listed pins form a valid AD sequence for the HCS12. The only restriction is that all the pins specified must be adjacent. Note that PAD07 and PAD00 are also considered to be adjacent analog inputs.

Parameters:

`const <Pins>`- Array of pins for ADC object to be associated with

Example:

```
dim myADC0 as new ADC(PAD00, PAD01)
```

Object Function Detail

ADC_Read

ADC_Read(const <Pin>, out byte/word *result*)

Get a value from an analog pin.

Parameters:

`const <Pin>`- Analog pin part of ADC object to read

`out byte/word` *result*- Result of conversion (use word if getting a 10-bit value)

Example:

```
myADC0.ADC_Read (PAD00, myResult)
```

ADC_Start

ADC_Start(in byte *wait*, in byte *mode*)

Initiate an analog-to-digital conversion

Parameters:

`in byte` *wait*- When WAIT will block until the conversion is complete

`in byte` *mode*-

Four possible modes: ADC_MODE_8ONCE: single 8-bit conversion
ADC_MODE_8CONTINUOUS: continuous 8-bit conversion
ADC_MODE_10ONCE: single 10-bit conversion
ADC_MODE_10CONTINUOUS: continuous 10-bit conversion

Example:

```
myADC0.ADC_Start (WAIT, ADC_MODE_8ONCE)
```

Class Function Detail

`ADC_Done`

`ADC_Done``(out byte` *`done`*`)`

Return the status of the ADC Conversion Complete flag

Parameters:

`out byte` *`done`*- Returns 1 if conversion complete, 0 if still in progress

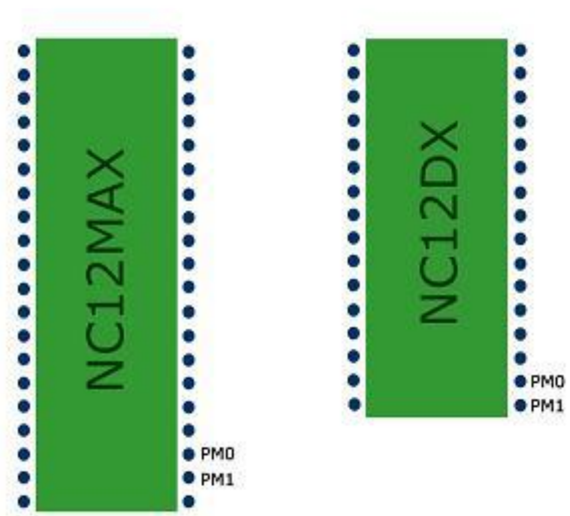
Example:

```
ADC.ADC_Done(myResult)
```

CAN

Controller Area Network Subsystem.
The CAN object uses the crystal oscillator to derive its timing. Hence, modifying PLL settings does not affect the functioning of this object.

Version:
1.0.0
Targets:
Nanocore12DX, Nanocore12MAX



Constructor Function Summary
CAN `(const <CAN RX pin>, const <CAN TX pin>)` CAN Constructor

Object Function Summary
CAN_Filter `(in byte filternumber, in byte mask, in byte value)` Define filter for CAN messages
CAN_Receive `(out byte[16] buffer, out byte length, out byte extended)` Wait for a CAN message to be received.
CAN_Send `(in byte txbuffer, in byte mode, in byte priority, const word <identlow>, const word <identhigh>, in byte datalength, in byte[6] data)` Send a CAN message.
CAN_Setup `(in byte mode, in byte bitrateprescaler, in byte filter)` Setup the CAN subsystem.
CAN_Shutdown `()` Put the CAN device in initialization mode.

Class Function Summary
CAN_Rec_data(in byte[16] *canbuffer*, out byte[8] *data*, in byte *length*) Extract the data buffer from a CAN message
CAN_Rec_filter(in byte[16] *canbuffer*, out byte *filter*) Get the filter number which passed the received CAN message.
CAN_Rec_ident(in byte[16] *canbuffer*, out word *identlow*, out word *identhigh*) Get the CAN identifier of the message
CAN_Rec_RTR(in byte[16] *canbuffer*, out byte *rtr*) Get the RTR (Remote Transmission Request) bit value of a CAN message

Constructor Function Detail

CAN

CAN(const <CAN RX pin>, const <CAN TX pin>)

Create CAN object with the specified parameters.

Parameters:

- const <CAN RX pin>- CAN RX pin
- const <CAN TX pin>- CAN TX pin

Object Function Detail

CAN_Filter

CAN_Filter(in byte *filternumber*, in byte *mask*, in byte *value*)

Define one of 8 filters to be used on received CAN-messages. Each filter is specified with 8 bit values. However, multiple filter values may be concatenated, according to the filter arrangement specified in CAN_Setup.

Parameters:

- in byte *filternumber*- Specify the filter to which the mask applies
- in byte *mask*- The mask to set for the filter
- in byte *value*- Value to use for the filter

CAN_Receive

`CAN_Receive(out byte[16] buffer, out byte length, out byte extended)`

This function waits for a CAN message to be received. (The CAN receiver can buffer up to 5 messages internally in a FIFO). Alternatively, EVENT_CAN may be used to wait for receiving a CAN frame, without busy-waiting in CAN_Receive.

Parameters:

`out byte[16] buffer`- Byte array buffer for incoming data
`out byte length`- Contains the data length of the CAN message
`out byte extended`- Contains 1 if extended format, or 0 if standard format

CAN_Send

```
CAN_Send(in byte txbuffer, in byte mode, in byte priority, const word <identlow>, const
word <identhigh>,
        in byte datalength, in byte[6] data)
```

Send a CAN message.

Parameters:

`in byte txbuffer`- Specify which one of the three tx-buffers to use: 0, 1, or 2
`in byte mode`- Mode can be one of the following:
0 - Transmit mode standard 11-bit identifier
1 - Transmit mode extended 29-bit identifier
2 - Transmit mode standard 11-bit identifier *and* set RTR bit
3 - Transmit mode extended 29-bit identifier *and* set RTR bit
`in byte priority`- Value indicating priority of message
`const word <identlow>`- Constant containing low part of CAN message identifier
`const word <identhigh>`- Constant contains the high 13 bits of the identifier
`in byte datalength`- Length of data to transmit
`in byte[6] data`- Data array to transmit

CAN_Setup

`CAN_Setup(in byte mode, in byte bitrateprescaler, in byte filter)`

Setup the CAN subsystem. Make sure you use CAN_Filter to set up the filter values *before* calling CAN_Setup! After CAN_Setup is invoked, the CAN device is no longer in initialization mode, so CAN_Filter calls are ignored.

Parameters:

`in byte` *`mode`*- Not implemented. Just pass 0
`in byte` *`bitrateprescaler`*- Bit rate prescaler
`in byte` *`filter`*- Specifies the filter arrangement:
0 - 2x CAN filter 32-bit
1 - 4x CAN filter 16-bit
2 - 8x CAN filter 8-bit
3 - Close filter

`CAN_Shutdown`

`CAN_Shutdown``()`

Put the CAN device in initialization mode. In initialization mode, the receive filters can be programmed with the CAN_Filter function. While in initialization mode, no communication can occur.

Parameters:

`None`

Class Function Detail

`CAN_Rec_data`

`CAN_Rec_data``(in byte[16]` *`canbuffer`*`, out byte[8]` *`data`*`, in byte` *`length`*`)`

Extract the data buffer from a CAN message

Parameters:

`in byte[16]` *`canbuffer`*- Specifies which CAN buffer to extract data from
`out byte[8]` *`data`*- Data extracted from CAN buffer
`in byte` *`length`*- Length of data to extact

`CAN_Rec_filter`

`CAN_Rec_filter``(in byte[16]` *`canbuffer`*`, out byte` *`filter`*`)`

Get the filter number which passed the received CAN message

Parameters:

`in byte[16]` *`canbuffer`*- CAN buffer to extract data from
`out byte` *`filter`*- The number of the filter that was used to receive the CAN frame

---

CAN_Rec_ident

CAN_Rec_ident(in byte[16] *canbuffer*, out word *identlow*, out word *identhigh*)

Get the CAN identifier of the message

Parameters:

- in byte[16] *canbuffer*- CAN buffer to extract data from
- out word *identlow*- Identifier low of CAN frame
- out word *identhigh*- Identifier high of CAN frame

CAN_Rec_RTR

CAN_Rec_RTR(in byte[16] *canbuffer*, out byte *rtr*)

Get the RTR (Remote Transmission Request) bit value of a CAN message

Parameters:

- in byte[16] *canbuffer*- CAN buffer to extract data from
- out byte *rtr*- Set to 1 if RTR is set in the CAN frame.

DIO

Digital input/output

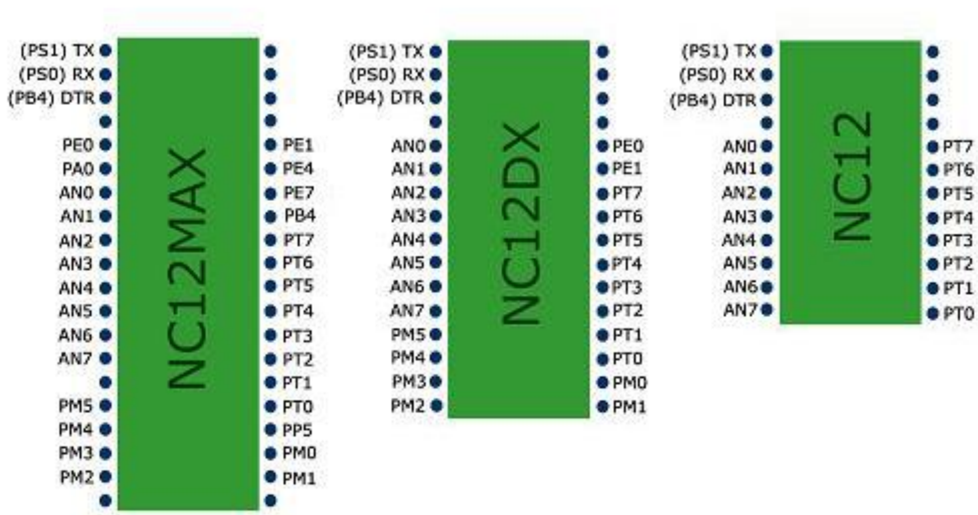
This object gives access to one or multiple I/O pins. The constructor sets all pins up as outputs, by default. However, object functions may be used to change the direction of pins to input or output, "on-the-fly". Multiple pins can be manipulated at the same time, by using the object functions which start with “PORT_”.

Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
DIO(const <Pins>) DIO Constructor

Object Function Summary
PIN_Dir(const <Pin>, in byte *direction*) Set the direction of the pin.
PORT_Dir(in byte *mask*) Set the direction of all pins in a port.

Class Function Summary
PIN_Busy_in(const <Pin>, in byte *value*) This function will block until the <Pin> matches the passed <Value>. Note that the RTI object can abort busy functions like this one.

Constructor Function Detail

DIO

DIO(const <Pins>)

Note that the pins in a DIO object may be a combination of pins from several ports in any order, to a maximum of 8 pins-- a virtual port!.

At startup, DIO pins are setup as output pins. Use PIN_Dir or PORT_Dir to specify (individual) pins as input pins, if desired.

Parameters:

const <Pins>- Array of pins for DIO object to be associated with

Example:

```
dim myDIO0 as new DIO(PT1, PT2)
```

Object Function Detail

PIN_Dir

PIN_Dir(const <Pin>, in byte *direction*)

Set the direction of the pin. INPUT or OUTPUT

Parameters:

const <Pin>- Designated pin to which direction setting applies. Must be part of the current DIO object.

in byte *direction*- 0 = output, 1 = input

Example:

```
myDIO0.PIN_Dir(PT1, OUTPUT)
```

PORT_Dir

PORT_Dir(in byte *mask*)

Set the direction of all pins in a port. (A DIO object with multiple pins is considered a port.) With the port functions, you can manipulate all the pins at once. A “1”-bit in the *Mask* makes the pin an input pin. The bit locations (LSB to MSB) correspond to pins in the DIO constructor (left to right, respectively).

Parameters:

in byte *mask*- Mask to use to define all the directions of a port simultaneously. Eg. a mask of 0b11110000 sets 4 pins as inputs and 4 pins as outputs

Example:

```
myDIO0.PORT_Dir(0b00000011)
```

Class Function Detail

PIN_Busy_in

PIN_Busy_in(const <Pin>, in byte *value*)

This function will block until the <Pin> matches the passed <Value>. Note that the RTI object can abort busy functions like this one.

Parameters:

const <Pin>- Pin to monitor

in byte *value*- Value to wait for

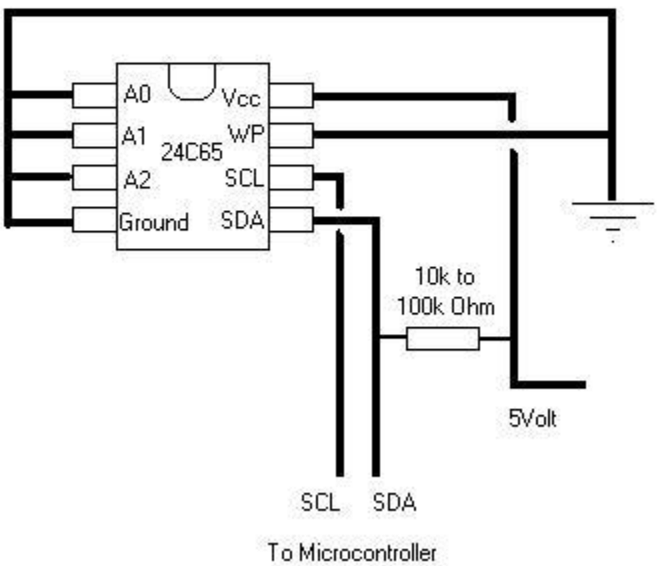
Example:

```
DIO.PIN_Busy_in(PAD04,HIGH)
```

IC2

IIC

This object implements a software ("bit-banged") master for the I2C protocol. It works on every I/O pin. The figure below shows an example of how to interface an I2C serial EEPROM device.

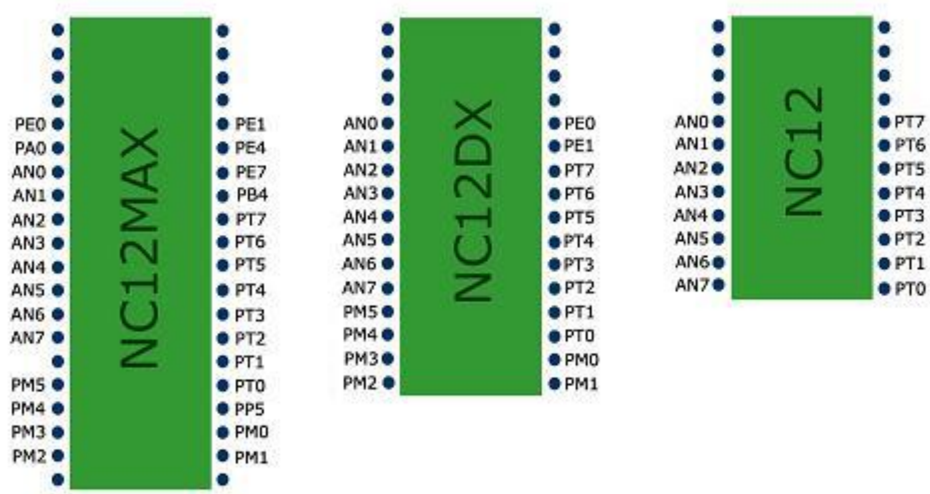


Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
I2C`(const <SDA pin>, const <CLK pin>)` I2C Constructor

Object Function Summary
I2C_Receive`(in byte *ack*, in byte *received*)` Receive a byte on the I2C bus.
I2C_Send`(in byte *data*)` Send a byte on the I2C bus.
I2C_Start`()` Send I2C start-bit
I2C_Stop`()` Send I2C stop-bit.

Constructor Function Detail

I2C

I2C(const <SDA pin>, const <CLK pin>)

Create I2C object using the specified pins.

Parameters:

- const <SDA pin>- Serial data pin
- const <CLK pin>- Clock pin

Example:

```
dim myI2C0 as new I2C (PT6, PT7)
```

Object Function Detail

I2C_Receive

I2C_Receive(in byte *ack*, in byte *received*)

Receive a byte on the I2C bus.

Parameters:

- in byte *ack*- If 0, will acknowledge
  if 1, will not acknowledge
- in byte *received*- Byte of received data

Example:

```
myI2C0.I2C_Receive(0,myData)
```

I2C_Send

I2C_Send(in byte *data*)

Send a byte on the I2C bus.

Parameters:

`in byte` *`data`*- Send data over I2C

Example:

```
myI2C0.I2C_Send(0x55)
```

`I2C_Start`

`I2C_Start``()`

Send I2C start-bit

Parameters:

`None`

Example:

```
myI2C0.I2C_Start()
```

`I2C_Stop`

`I2C_Stop``()`

Send I2C stop-bit.

Parameters:

`None`

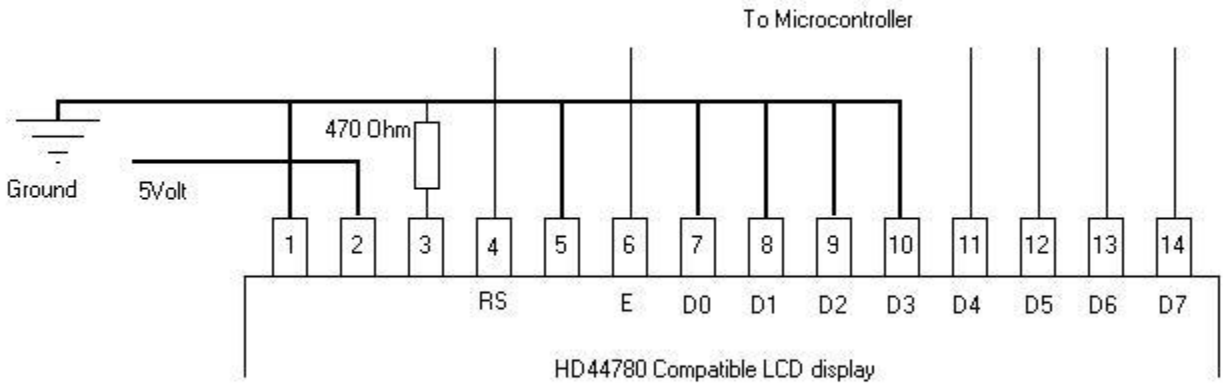
Example:

```
myI2C0.I2C_Stop()
```

LCD

Liquid Crystal Display

This object implements a software ("bit-banged") HD44780-compatible character-LCD interface. The LCD is connected via 6-pins, in 4-bit mode.

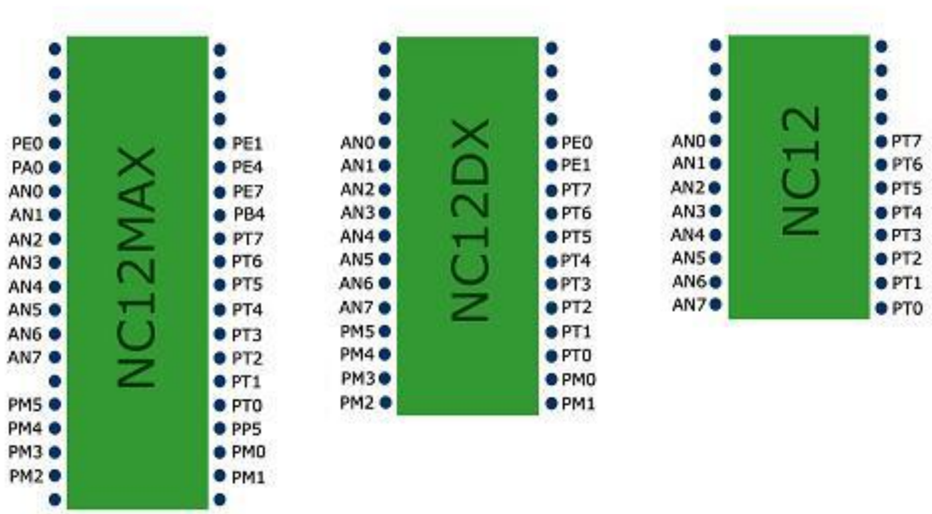


Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
LCD(const <LCD-D4>, const <LCD-D5>, const <LCD-D6>, const <LCD-D7>, const <LCD-E>, const <LCD-RS>) LCD Constructor

Object Function Summary
LCD_Char(in byte *char*) Place a character on the display, at the current cursor-position.
LCD_Command(const <LCD COMMAND>, in byte *adrdata*) Send a command to the LCD.
LCD_Decimal(in byte/word *data*, const <FILL TYPE>) Displays the value of a variable in readable decimal text.
LCD_Hex(in byte/word *data*, const <FILL TYPE>) Displays the value of a variable in readable hexadecimal text.
LCD_Init(const mode) Initialize the LCD
LCD_String(const <STRING>) Display a 0-terminated string-constant.

Constructor Function Detail

LCD

LCD(const <LCD-D4>, const <LCD-D5>, const <LCD-D6>, const <LCD-D7>, const <LCD-E>, const <LCD-RS>)

Create LCD object using the specified pins.

Parameters:

- const <LCD-D4>- LCD D4 pin
- const <LCD-D5>- LCD D5 pin
- const <LCD-D6>- LCD D6 pin
- const <LCD-D7>- LCD D6 pin
- const <LCD-E>- LCD enable pin
- const <LCD-RS>- LCD RS pin

Example:

```
dim myLCD0 as new LCD (PT0, PT1, PT2, PT3, PT4, PT5)
```

Object Function Detail

LCD_Char

LCD_Char(in byte *char*)

Place a character on the display, at the current cursor-position.

Parameters:

- in byte *char*- Char to display

Example:

```
myLCD0.LCD_Char('A')
```

`LCD_Command`

`LCD_Command``(const <LCD COMMAND>, in byte` *`adrdata`*`)`

Send a command to the LCD. (eg. to clear the display, to change the cursor mode/position, etc.)

Parameters:

`const <LCD COMMAND>`- LCD command to send

Command	Description
LCD_CLEAR_DISPLAY	Clear display; **NOTE: includes 1.64ms delay!**
LCD_HOME	Home (cursor to first position on first line) **NOTE: includes 1.64ms delay!**
LCD_AUTO_BACK	Entry auto--
LCD_AUTO_SHIFT_BACK	Entry shift auto--
LCD_AUTO_FORW	Entry auto++
LCD_AUTO_SHIFT_FORW	Entry shift auto++
LCD_DISPLAY_OFF	Display off
LCD_CURSOR_OFF	Display no cursor
LCD_CURSOR	Display cursor
LCD_CURSOR_BLINK	Display blink cursor
LCD_CURSOR_LEFT	Cursor Left
LCD_CURSOR_RIGHT	Cursor Right
LCD_SCROLL_LEFT	Scroll Left
LCD_SCORLL_RIGHT	Scroll Right
LCD_SET_CGRAM	Set CGRAM
LCD_SET_DDRAM	Set DDRAM
LCD_UPLOAD_RAM	Upload XXXRAM

`in byte` *`adrdata`*- Address or data to use with command, (if needed)

Example:

```
myLCD0.LCD_Command (LCD_HOME,0)
```

`LCD_Decimal`

`LCD_Decimal``(in byte/word` *`data`*`, const <FILL TYPE>)`

Displays the value of a variable in readable decimal text. A byte variable will always result in three ASCII digits being displayed (eg. “255”), while a word variable will always result in five ASCII digits being displayed (eg. “65535”).

Parameters:

`in byte/word` *`data`*- Data to display (can be either byte or word)

`const <FILL TYPE>`- 0 to fill high spaces with "0", 1 to fill with " "(blanks)

Example:

```
myLCD0.LCD_Decimal(res,FILLUP_ZERO)
```

LCD_Hex

LCD_Hex(in byte/word *data*, const <FILL TYPE>)

Displays the value of a variable in readable hexadecimal text. Byte variables will always result in two ASCII digits being displayed (eg. “FF”), and word variables will always result in four ASCII digits being displayed (eg. “FFFF”). The fill-type will determine what the most-significant digits will contain, if the number is too small to generate digits in these positions.

Parameters:

`in byte/word data`- Data to display (can be either byte or word)

`const <FILL TYPE>`- 0 to fill high spaces with "0", 1 to fill with " "(blanks)

Example:

```
myLCD0.LCD_Hex(res,FILLUP_ZERO)
```

LCD_Init

LCD_Init(const mode)

Initialize the LCD control registers with the supplied parameters. Typical parameters include the display configuration (eg. single- or multi-line display), the display mode (eg. blinking cursor), etc.

Parameters:

`const mode`- LCD_MODE_ONE_LINE for single line display, LCD_MODE_MORE_LINES for multi-line display.

Example:

```
myLCD0.LCD_Init(LCD_MODE_MORE_LINES)
```

LCD_String

LCD_String(const <STRING>)

Display a 0-terminated string-constant, beginning at the current cursor postion.

Parameters:

`const <STRING>`- Null-terminated const string to display

Example:

```
myLCD0.LCD_string("HelloWorld")
```

PWM

Pulse Width Modulator subsytem

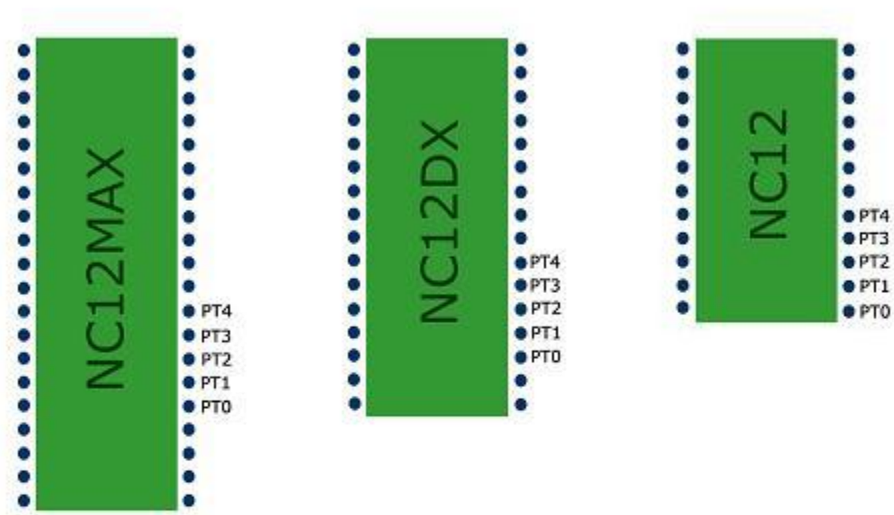
Note: the PWM channels are referred to as PP0 through PP5. However, the lower five channels (PP0 - PP4) are multiplexed onto the Port T pins, so they actually appear on pins PT0 through PT4. The sixth channel is not multiplexed, and appears on pin PP5 (only present on the 40-pin module).

Version:

  1.0.0

Targets:

  Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
PWM(const <PWM PIN>) PWM Constructor

Object Function Summary
PWM_Start(const <CLOCK>, const <LEVEL>, in byte *period*, in byte *duty*) Start a pulse-train on the pin of this object.
PWM_Start_ext(const <CLOCK>, const <LEVEL>, in word *period*, in word *duty*) Start an extended PWM pulse on the pin of this object.
PWM_Stop() Stop the PWM pulse.

Class Function Summary
PWM_Res_PP0145(const <BUS CLOCK DIV>, const <SCALED DIV>) This function sets up the possible clock rates for PWM signals on pins PP0, PP1, PP4 and PP5.

PWM_Res_PP23(const <BUS CLOCK DIV>, const <SCALED DIV>)
This function sets up the possible clock rates for PWM signals on pins PP2 and PP3.

Constructor Function Detail

PWM

PWM(const <PWM PIN>)

Creates a PWM object on the specified pin.

Parameters:

`const <PWM PIN>`- PWM pin

Example:

```
dim myPWM0 as new PWM(PP0)
```

Object Function Detail

PWM_Start

PWM_Start(const <CLOCK>, const <LEVEL>, in byte *period*, in byte *duty*)

Start generating a pulse-train on the pin of this object.

Parameters:

`const <CLOCK>`- 0 for main clock, 1 for scaled clock.
`const <LEVEL>`- 0 for normal, 1 for inverted.
`in byte` *period*- Period of the PWM waveform
`in byte` *duty*- Duty cycle of the PWM waveform

Example:

```
myPWM0.PWM_Start(PWM_MAIN_CLK, PWM_NORMAL, 255, 120)
```

PWM_Start_ext

PWM_Start_ext(const <CLOCK>, const <LEVEL>, in word *period*, in word *duty*)

Activate a pulse train on the pin of this object. Extended PWM concatenates two 8-bit pulse registers into one 16-bit pulse register, resulting in a higher range/resolution for a single pin PWM.

Parameters:

`const <CLOCK>`- 0 for main clock, 1 for scaled clock
`const <LEVEL>`- 0 for normal, 1 for inverted.
`in word` *period*- Period of the PWM waveform
`in word` *duty*- Duty cycle of the PWM waveform

Example:

```
myPWM0.PWM_Start_ext (PWM_MAIN_CLK, PWM_NORMAL, 10000, 2000)
```

PWM_Stop

PWM_Stop()

Stop the PWM pulse train.

Parameters:

None

Example:

```
myPWM0.PWM_Stop()
```

Class Function Detail

PWM_Res_PP0145

PWM_Res_PP0145(const <BUS CLOCK DIV>, const <SCALED DIV>)

This function sets up the possible clock rates for PWM signals on pins PP0, PP1, PP4 and PP5. The scaled clock is derived from the PWM main clock. For each of the four PWM signals, you can choose either clock source.

Parameters:

const <BUS CLOCK DIV>- Bus clock divider
const <SCALED DIV>- Scaled divider

Example:

```
PWM.PWM_Res_PP0145 (TIMER_DIV_8, 0)
```

PWM_Res_PP23

PWM_Res_PP23(const <BUS CLOCK DIV>, const <SCALED DIV>)

This function sets up the possible clock rates for PWM signals on pins PP2 and PP3. The scaled clock is derived from the PWM main clock. For each of the PWM signals, you can choose either clock source.

Parameters:

const <BUS CLOCK DIV>- Bus clock divider
const <SCALED DIV>- Scaled clock divider

Example:

```
PWM.PWM_Res_PP23(TIMER_DIV_8,0)
```

RTI

Real time interrupt

This object gives access to the real-time timer (RTI) of the MCU. EVENT_RTI can be used to WAIT for timer expiration. Note that the RTI is driven from the crystal oscillator, so using the PLL does not affect its speed. Caution: when the MCU is in Active BDM Mode, the RTI timer is NOT running!

Version:
    1.0.0

Targets:
    Nanocore12, Nanocore12DX, Nanocore12MAX

Class Function Summary
RTI_Start`(const <PRESCALER DIV>, const <FINE DIV>, const <EXPIRATION ACTION>)` Start the real-time timer.
RTI_Stop`()` Disables the timer interrupt.

Class Function Detail

RTI_Start

RTI_Start`(const <PRESCALER DIV>, const <FINE DIV>, const <EXPIRATION ACTION>)`

Start the real-time timer.

Parameters:

`const <PRESCALER DIV>`- Prescaler divider.

Constant in stdlib.ncb	Devides clock by	Period with 8mHz crystal	Actual value passed
RTI_PRESCALE_OFF	Timer OFF		0
RTI_PRESCALE_1024	1024	128 usec	1
RTI_PRESCALE_2048	2048	256usec	2
RTI_PRESCALE_4096	4096	0.5 msec	3
RTI_PRESCALE_8192	8192	1 millisec	4
RTI_PRESCALE_16384	16384	2 millisec	5
RTI_PRESCALE_32768	32768	4.1 millisec	6
RTI_PRESCALE_65536	65536	8.2 millisec	7

`const <FINE DIV>`- Fine Divide
`const <EXPIRATION ACTION>`- Expiration action

Example:

```
RTI.RTI_Start(RTI_PRESCALE_1024,0,RTI_EXPIRE_RESTART)
```

RTI_Stop

RTI_Stop()

Disables the timer interrupt (however, the timer keeps running). Only necessary when RTI_Start called with expiration action, which restarted the timer.

Parameters:

`None`

Example:

```
RTI.RTI_Stop()
```

SCI

Serial Port

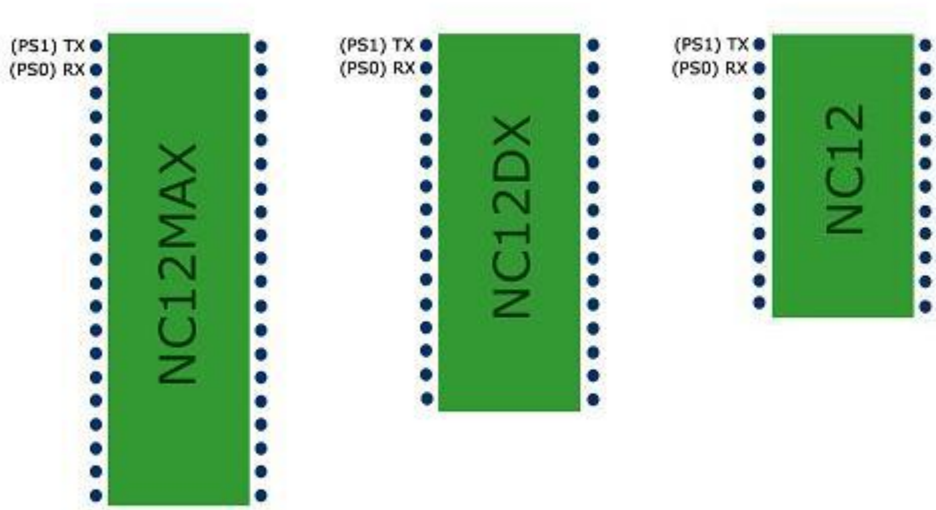
This object gives access to the Serial Communications Interface subsystem of the MCU (ie. UART).

Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
SCI(const <RX PIN>, const <TX PIN>) SCI Constructor

Object Function Summary
SER_Control(const <SUSPEND>) Disable/Enable the receiver (and its interrupt handler).
SER_Flush_in() Empties the serial receive buffer.
SER_Get_char(const <WAIT>, out byte *received*) Gets a character from the serial input receive buffer.
SER_Put_char(in byte *char*) Transmits a character over the serial port.
SER_Put_decimal(in byte/word *data*, const <FILL TYPE>) Transmits the value of a variable in readable decimal text.
SER_Put_hex(in byte/word *data*, const <FILL TYPE>) Transmits the value of a variable in readable hexadecimal text.

SER_Put_string(const <STRING>) Outputs a 0-terminated string-constant on the serial port.
SER_Setup(const <BUFFER SIZE>, const <BAUDRATE>) Sets up the SCI for 8N1 with selected baudrate.

Class Function Summary
SER_Busy_get(const <PIN>, const <LOGIC>, const <BAUDRATE>, const <WAIT>, out byte *received*) This function will attempt to receive a serial (RS232-like) character on any pin that is configured as input-pin.

Constructor Function Detail

SCI

SCI(const <RX PIN>, const <TX PIN>)

SCI Constructor

Parameters:

const <RX PIN>- SCI RX pin
const <TX PIN>- SCI TX pin

Example:

```
dim mySCI0 as new SCI(PS0,PS1)
```

Object Function Detail

SER_Control

SER_Control(const <SUSPEND>)

SER_DISABLE_RECEIVER to disable the receiver (and its interrupt handler). Pass SER_ENABLE_RECEIVER to enable it again.

Parameters:

const <SUSPEND>- Disable or enable receiver (enable only needed if you disable the receiver)

Example:

```
mySCI0.SER_Control(SER_DISABLE_RECEIVER)
```

SER_Flush_in

SER_Flush_in()

Empties the serial port's receive buffer.

Parameters:

None

Example:

```
mySCI0.SER_Flush_in()
```

SER_Get_char

SER_Get_char(const <WAIT>, out byte *received*)

Gets a character from the input receive buffer of the SCI.

Parameters:

const <WAIT>- If you want the function to wait till it has received a charater or not.

out byte *received*- Received data

Example:

```
mySCI0.SER_Get_char(1, Char)
```

SER_Put_char

SER_Put_char(in byte *char*)

Transmits a character over the serial port (SCI).

Parameters:

in byte *char*- Character to ouput on the SCI

Example:

```
mySCI0.SER_Put_char ('A')
```

SER_Put_decimal

SER_Put_decimal(in byte/word *data*, const <FILL TYPE>)

Transmits the value of a variable in readable decimal text. Bytes will always result in three ASCII digits being transmitted (eg. “255”) and words will always result in five ASCII digits being transmitted (eg. “65535”)

Parameters:

in byte/word *data*- Data to display (can be either byte or word)

const <FILL TYPE>- FILLUP_ZERO to fill high spaces with "0", FILLUP_SPACE to fill with " "(blank spaces).

Example:

```
mySCI0.SER_Put_decimal(0x23,FILLUP_SPACE)
```

SER_Put_hex

SER_Put_hex(in byte/word *data*, const <FILL TYPE>)

Transmits the value of a variable in readable hexadecimal text. Byte variables will always result in two ASCII digits being transmitted (eg. “FF”) and word variables will always result in four ASCII digits being transmitted (eg. “FFFF”). The fill-type will determine what the most-significant digits will contain if the number is too small to generated digits in these positions.

Parameters:

`in byte/word data`- data to display, can be either byte or word

`const <FILL TYPE>`- FILLUP_ZERO to fill high spaces with "0", FILLUP_SPACE to fill with " "(blanks)

Example:

```
mySCI0.SER_Put_hex(0x23,FILLUP_SPACE)
```

SER_Put_string

SER_Put_string(const <STRING>)

Transmits a null-terminated string-constant from the SCI.

Parameters:

`const <STRING>`- Null-terminated const string to display

Example:

```
mySCI0.SER_Put_string("Hello World")
```

SER_Setup

SER_Setup(const <BUFFER SIZE>, const <BAUDRATE>)

Sets up the SCI for 8 bits data, no start bit, and one stop bit, with selected baudrate.

Parameters:

`const <BUFFER SIZE>`- Value, in bytes, can be SER_BUFFER_2, SER_BUFFER_4, or SER_BUFFER_8.

`const <BAUDRATE>`- Any of the predefined baudrates

Example:

```
mySCI0.SER_Setup(SER_BUFFER_4,BAUD19200)
```

Class Function Detail

SER_Busy_get

SER_Busy_get(const <PIN>, const <LOGIC>, const <BAUDRATE>, const <WAIT>, out byte *received*)

This function will attempt to receive a serial (RS232-like) character on any pin that is configured as input-pin.

Parameters:

`const <PIN>`- Pin to wait for incoming signal on
`const <LOGIC>`- 0 = normal, 1 = inverted
`const <BAUDRATE>`- Baudrate select
`const <WAIT>`- 0 if no start bit needed; 1 otherwise
`out byte` *`received`*- Received character

Example:

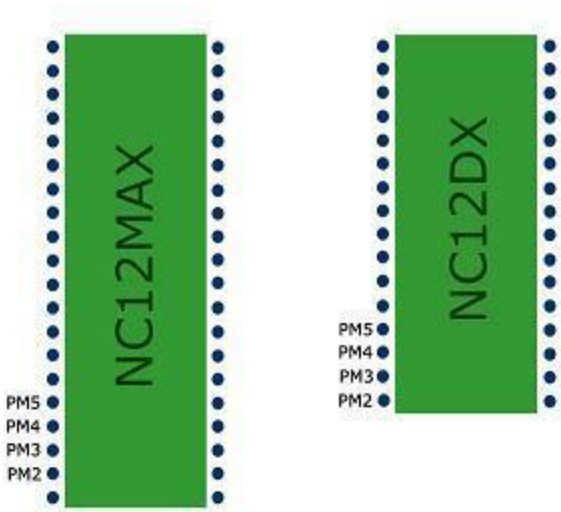
```
SCI.SER_Busy_get(PAD02,0,BAUD19200,1,Result)
```

SPI

Serial peripheral interface
General purpose serial synchronous communication device (SPI = Synchronous Peripheral Interface).

Version:
1.0.0

Targets:
Nanocore12DX, Nanocore12MAX



Constructor Function Summary
SPI`(const <MISO PIN>, const <MOSI PIN>, const <SCK PIN>, const <SS PIN>)` SPI Constructor

Object Function Summary
SPI_Done`(out byte` *done*`)` Check if the SPI is finished transferring
SPI_Received`(out data` *received*`)` Get the last received byte.
SPI_Reply`(in byte` *replydata*`)` This function is used to setup the reply-data.
SPI_Setup`(const <MAS/SLV>, const <PRESCALER>, const <FINE DIV>, const <MODE>, const <BITDIRECTION>)` Setup the SPI device.
SPI_Transfer`(in byte` *sendbyte*`, const <WAIT>, out byte` *received*`)` Initiate an SPI transfer to send a byte to the slave.

Constructor Function Detail

SPI

SPI(const <MISO PIN>, const <M0SI PIN>, const <SCK PIN>, const <SS PIN>)

SPI Constructor

Parameters:

- const <MISO PIN>- MISO pin
- const <M0SI PIN>- MOSI pin
- const <SCK PIN>- SCK pin
- const <SS PIN>- SS pin

Example:

```
dim mySPI0 as new SPI(PM2,PM4,PM5,PM3)
```

Object Function Detail

SPI_Done

SPI_Done(out byte *done*)

Check if the SPI is finished transferring (applicable to both MASTER and SLAVE setups).

Parameters:

- out byte *done*- 1 if done, 0 if otherwise

Example:

```
mySPI0.SPI_Done(myResult)
```

SPI_Received

SPI_Received(out data *received*)

Get the last received byte. Applicable to both MASTER and SLAVE setups. EVENT_SPI can be used to WAIT for completion of transfer.

Parameters:

- out data *received*- Received data

Example:

```
mySPI0.SPI_Received(myResult)
```

SPI_Reply

`SPI_Reply(in byte replydata)`

Only relevant when SPI is setup as SLAVE. This function is used to setup the reply data to send to the MASTER next time it initiates an SPI transfer to the board.

Parameters:

`in byte replydata`- Data to reply with

Example:

```
mySPI0.SPI_Reply(0x34)
```

SPI_Setup

`SPI_Setup(const <MAS/SLV>, const <PRESCALER>, const <FINE DIV>, const <MODE>, const <BITDIRECTION>)`

Setup the SPI device.

Parameters:

`const <MAS/SLV>`- 0: Master, 1: Slave

`const <PRESCALER>`- Prescaler divisor

Constant in stdlib.ncb	Devides bus-clock by	Resulting rate with 8mHz crystal	Actual value passed to <prescale>
SPI_DIV_2	2	2 mHz	0
SPI_DIV_4	4	1 mHz	1
SPI_DIV_8	8	500 kHz	2
SPI_DIV_16	16	250 kHz	3
SPI_DIV_32	32	125 kHz	4
SPI_DIV_64	64	62.5 kHz	5
SPI_DIV_128	128	31.25 kHz	6
SPI_DIV_256	256	15.625 kHz	7

`const <FINE DIV>`- Fine divisor rate

`const <MODE>`- One of the predefined modes

Constant in stdlib.ncb	Description	Value in <mode>
SPI_CLK_HIGH_SS_OR_CLK	Clock is active HIGH and Slave Select line is OR-ed with clock	0
SPI_CLK_HIGH_SS_LOW	Clock is active HIGH and Slave Select line is active LOW	1
SPI_CLK_LOW_SS_OR_CLK	Clock is active LOW and Slave Select line is OR-ed with CLK	2
SPI_CLK_LOW_SS_LOW	Clock is active LOW and	3

	Slave Select line is active LOW	

`const <BITDIRECTION>`- 1 if high bits first, 0 if low bits first.

Example:

```
mySPI0.SPI_Setup(SPI_MASTER,SPI_DIV_128,0,SPI_CLK_HIGH_SS_LOW,SPI_HIGH_BIT_FIRST)
```

SPI_Transfer

SPI_Transfer`(in byte` *`sendbyte`*`, const <WAIT>, out byte` *`received`*`)`

Initiate a SPI-transfer to send a byte to the slave. (For use only with an SPI set up as a MASTER.)

Parameters:

`in byte` *`sendbyte`*- Data to send
`const <WAIT>`- If 1: will wait until completed
`out byte` *`received`*- Received data

Example:

```
mySPI0.SPI_Transfer(0x24,0,myResult)
```

SYSTEM

This object provides various system-related class functions only.

Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX

Class Function Summary
CRC_Calc`(in byte[..] databuffer, in byte size, out byte crc)` Calculates 8bit CRC
Delay`(in word time)` Delays a number of microseconds.
Delay_cycles`(in word delay)` Delays a number of CPU cycles.
INTS_Off`()` Disables the interrupts.
INTS_On`()` Enables the interrupts
PLL_Set`(const <KHz SPEED>)` Changes the speed of the processor - for advanced users only
Sleep`(const <WAKEUP ON>)` Puts the MCU to sleep.

Class Function Detail

CRC_Calc

CRC_Calc`(in byte[..] databuffer, in byte size, out byte crc)`

Calculates 8-bit CRC. This function can be used for packet-validation for the 1-wire protocol. The CRC algorithm is the same as used by 1-wire devices. (Note that use of CRC-validation is optional for most 1-wire devices).

Parameters:

`in byte[..] databuffer`- Data buffer

`in byte size`- Size of databuffer

`out byte crc`- 8-bit CRC calculated

Example:

```
SYSTEM.CRC_Calc(myData,12,myResult)
```

Delay

Delay(in word *time*)

Delays a number of microseconds. Parameter is a WORD; hence, the maximum possible delay is 65535 microseconds (65.5 milliseconds).

Parameters:

in word *time*- Period to wait, in microseconds

Example:

```
SYSTEM.Delay(200)
```

Delay_cycles

Delay_cycles(in word *delay*)

Delays a number of CPU cycles. Parameter is a WORD, hence max. 65535 cycles delay.

Parameters:

in word *delay*- Delay time, in cycles

Example:

```
SYSTEM.Delay_cycles(200)
```

INTS_Off

INTS_Off()

Disables the interrupts. Example use: to start a (time) critical section of code, which should not be disturbed. Use with care, since other software (eg. timers), might depend on the handling of interrupts. Use this function if timing is critical and you want to make sure the MCU spends no time on other code (interrupt-handlers), while your section of code is running. Make sure you use INTS_On to restore interrupt-processing again.

Parameters:

None

Example:

```
SYSTEM.INTS_Off()
```

INTS_On

INTS_On()

Enables the interrupts (end of critical section). See also INTS_Off above.

Parameters:

None

Example:

```
SYSTEM.INTS_On()
```

PLL_Set

PLL_Set(const <KHz SPEED>)

Changes the speed of the processor. For example, it can be used to reduce power consumption at IDLE times. Speed is passed in kHz. (eg. 4000 means equivalent of 4 MHz crystal; hence, 2 MHz bus-frequency). Note that the devices which run the bootloader/monitor, run at 24 MHz bus frequency (or the equivalent of 48 MHz crystal). Note that not all frequencies are valid! An error will be reported if a frequency parameter was passed which cannot be created with the current crystal.

WARNING: ALL object libraries (except WTD, RTI and CAN) are clock-speed dependent. The project-PLL-setting is used to calibrate these libraries at compile-time. Changing the PLL-speed will change the timing of these libraries, which may result in faulty or unexpected behavior. Especially sensitive are asynchronous communication objects, which require fixed data rates, such as SCI, LCD and WIRE1. These might not work properly at different speeds.

Parameters:

`const <KHz SPEED>`- Speed in KHz for PLL clock

Example:

```
SYSTEM.PLL_Set(8000)
```

Sleep

Sleep(const <WAKEUP ON>)

Puts the MCU to sleep.

Parameters:

`const <WAKEUP ON>`- 0: any interrupt will wake up the MCU
1: only external interrupts will wake the MCU

Example:

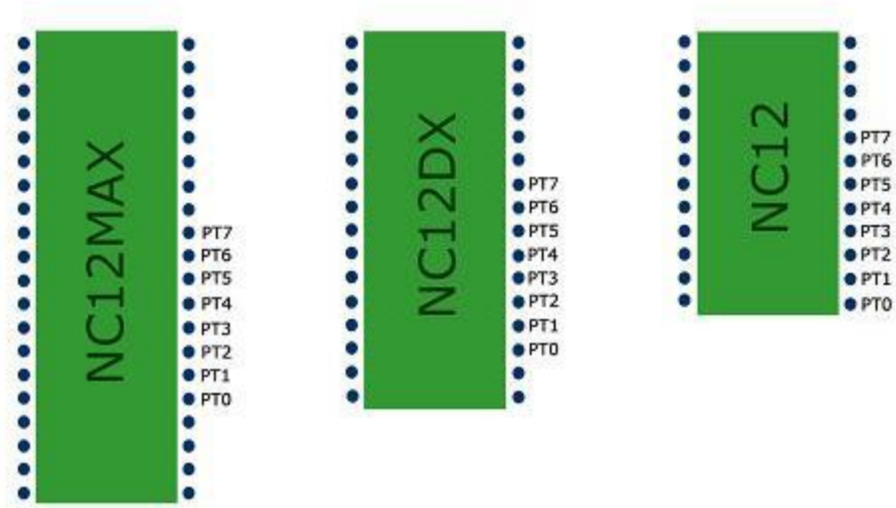
```
SYSTEM.Sleep(SLEEP_UNTIL_ANY_INT)
```

TIMIO

Timer I/O Object

Version:
1.0.0

Targets:
Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
TIMIO(const <PORTT PIN>) TIMIO Constructor

Object Function Summary
TIMIO_Capture(const <SIGNAL>) Captures the event timestamp of the specified transition-type for the specified pin.
TIMIO_Get_time(out word *timestamp*) Read the timestamp of a pin.
TIMIO_In(out byte *value*) Makes the pin an input pin and returns its value.
TIMIO_Kill() Stop the Output Compare timer of the specified pin.
TIMIO_Out(in byte *value*) Makes the pin an output pin, and sets its level according to the passed value.
TIMIO_Output(const <TIMEDELAY>, const <ACTION>) Use one of the Output Compare timers to control the behaviour of the specified pin.

Class Function Summary
TIMIO_Timer_start`(const <RESOLUTION>)` Start central timer from which all TIMIO timing is derived.
TIMIO_Timer_stop`()` Stop the central timer

Constructor Function Detail

TIMIO

`TIMIO``(const <PORTT PIN>)`

TIMIO Constructor

Parameters:

`const <PORTT PIN>`- Port T pin to associate with TIMIO

Example:

```
dim myTIM0 as new TIMIO(PT0)
```

Object Function Detail

TIMIO_Capture

`TIMIO_Capture``(const <SIGNAL>)`

Captures the event timestamp of the specified transition-type for the specified pin. When the transition event is detected on the pin, the timestamp is stored in the timer register corresponding to that pin.

Parameters:

`const <SIGNAL>`- Signal condition upon which to capture

Example:

```
myTIM0.TIMIO_Capture(TIMIO_EDGE_ANY)
```

TIMIO_Get_time

`TIMIO_Get_time``(out word` *`timestamp`*`)`

Read the timestamp of a pin. Can be used with TIMIO_Capture, after EVEN_IOC occurred, indicating successful capture *or* to see how far the timer proceeded in the timer delay period of a TIMIO_Output function call.

Parameters:

`out word` *`timestamp`*- Read the timestamp of a pin

Example:

```
myTIM0.TIMIO_Get_time(myResult)
```

TIMIO_In

TIMIO_In(out byte *value*)

Makes the pin an input pin and returns its value (0 or 1) in the passed variable. You can use this function to use the pin as input, if the TIMIO object is used for its timer only. (TIMIO_Output function with TIMIO_PIN_NONE action).

Parameters:

`out byte value`- Logic value returned from the port pin (0 = low, 1 = high)

Example:

```
myTIM0.TIMIO_In(myResult)
```

TIMIO_Kill

TIMIO_Kill()

Stop the OutputCompare timer of the specified pin. Aborts both TIMIO_Capture and TIMIO_Output activity.

Parameters:

None

Example:

```
myTIM0.TIMIO_Kill()
```

TIMIO_Out

TIMIO_Out(in byte *value*)

Makes the pin an output pin and sets its level according to the passed value (0 or 1 constant or via variable). You can use this function to use the pin as an output, if the TIMIO object is used for its timer only. (TIMIO_Output function with TIMIO_PIN_NONE action).

Parameters:

`in byte value`- Logic-level to send to the pin (0 = low, 1 = high)

Example:

```
myTIM0.TIMIO_Out(1)
```

TIMIO_Output

TIMIO_Output(const <TIMEDELAY>, const <ACTION>)

Use one of the OutputCompare timers to control the behaviour of the specified pin.

Parameters:

const <TIMEDELAY>- Number of ticks before the timer restarts, after performing the specified action
const <ACTION>- Action to take

Example:

```
myTIM0.TIMIO_Output(1)
```

Class Function Detail

TIMIO_Timer_start

TIMIO_Timer_start(const <RESOLUTION>)

Start central timer from which all TIMIO timing is derived.

Parameters:

const <RESOLUTION>- Resolution of the timer subsystem

Example:

```
TIMIO.TIMIO_Timer_start (TIMER_DIV_128)
```

TIMIO_Timer_stop

TIMIO_Timer_stop()

Stop the central timer (and hence all TIMIO functions).

Parameters:

None

Example:

```
TIMIO.TIMIO_Timer_stop()
```

WDT

Watchdog timer

Watchdog Timer is based on the COP feature of the MCU (COP=Computer Operating Properly). When the WDT_Set class-function is called, the watchdog countdown timer will be activated. The micro-kernel of nqBasic will reset the watchdog timer, whenever it gets control. This means that care should be taken when executing long, non-deterministic loops in a single task (eg. while-FOREVER) or busy functions, since the watchdog timer might expire before the nqBasic micro-kernel has a chance to reset it.

WARNING: DO NOT use this function if you are NOT USING TASKS! If you are only using the *main* function, the nqBasic micro-kernel will not be able to reset the watchdog timer, inevitably resulting in expiration and reset of the device!

Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX

Class Function Summary
WDT_set`(const <WDT TIMEOUT>)` Calling this function will activate the watchdog (also called COP by Freescale).

Class Function Detail

WDT_set

WDT_set`(const <WDT TIMEOUT>)`

Calling this function will activate the watchdog (also called COP by Freescale).

Parameters:

`const <WDT TIMEOUT>`- Watchdog (COP) timeout period

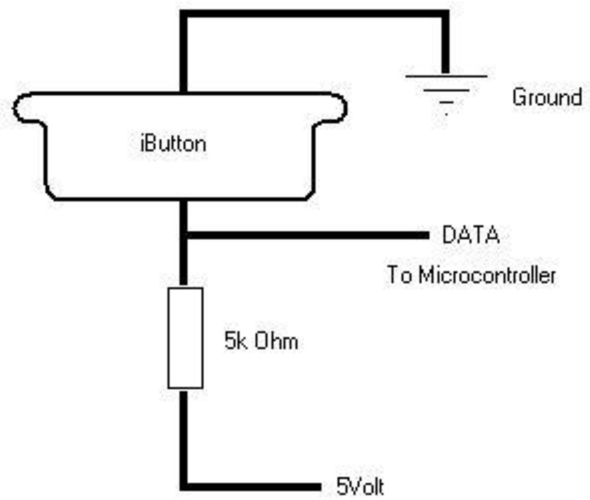
Example:

```
WDT.WDT_set(2000)
```

WIRE1

1-Wire

This object implements a software ("bit-banged") Dallas 1-Wire master protocol. It works on every I/O pin. The 1-wire protocol requires only a single pin. The figure below shows how to wire a DS1921 temperature iButton. NOTE: each specific 1-Wire device has its own protocol (i.e. commands it supports, parameters it expects, etc). Refer to the datasheet of the 1-Wire device you are using for details on the protocol it requires.

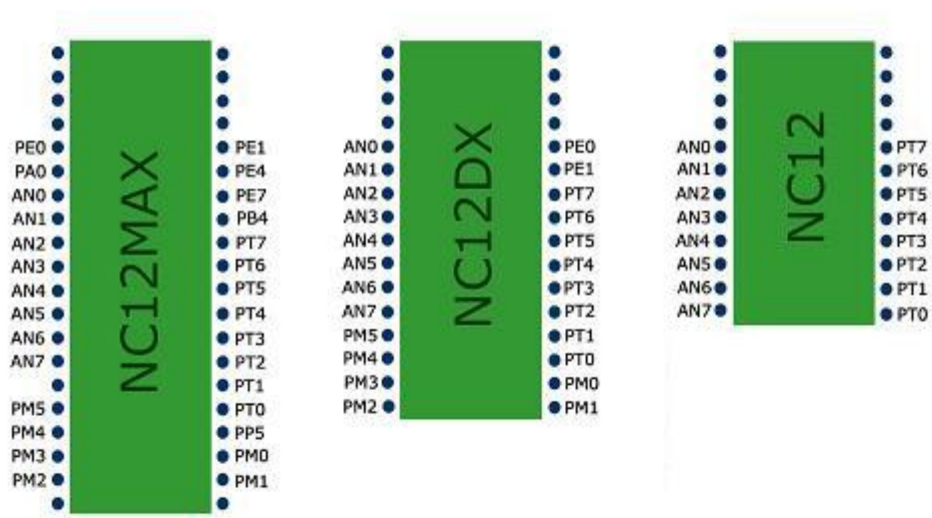


Version:

1.0.0

Targets:

Nanocore12, Nanocore12DX, Nanocore12MAX



Constructor Function Summary
WIRE1`(const <Data Pin>)` WIRE1 Constructor

Object Function Summary
WR1_High`()` Low-level function which brings the 1-Wire® bus to a HIGH state.
WR1_Init`()` Initialize the 1-Wire® bus.
WR1_Low`()` Low-level function which brings the 1-Wire® bus to a LOW state.
WR1_Read`(out byte *result*)` Read a byte of data from a 1-Wire® device.
WR1_Write`(in byte *data*)` Send a byte to a 1-Wire® slave-device.

Constructor Function Detail

WIRE1

WIRE1`(const <Data Pin>)`

WIRE1 Constructor

Parameters:

`const <Data Pin>`- Data pin

Example:

```
dim myWIRE10 as new WIRE1(PT3))
```

Object Function Detail

WR1_High

WR1_High`()`

Low-level function which brings the 1-Wire® bus to a HIGH state. You will hardly ever need to call this function yourself. (The higher level WR1_Write and WR1_Read functions do most of the work).

Parameters:

None

Example:

```
myWIRE10.WR1_High
```

WR1_Init

WR1_Init()

Initialize the 1-Wire® bus. (Resets all slave devices).

Parameters:

None

Example:

```
myWIRE10.WR1_Init
```

WR1_Low

WR1_Low()

Low level function which brings the 1-Wire® bus to a LOW state. You will hardly ever need to call this function yourself. (The higher level WR1_Write and WR1_Read functions do most of the work).

Parameters:

None

Example:

```
myWIRE10.WR1_Low
```

WR1_Read

WR1_Read(out byte *result*)

Read a byte of data from a 1-Wire® device. This device must already know that it has to send the byte, by a command it received via WR1_Write.

Parameters:

out byte *result*- Byte retrieved from 1-Wire device

Example:

```
myWIRE10.WR1_Read(myResult)
```

WR1_Write

WR1_Write(in byte *data*)

Send a byte to a 1-Wire® slave device. The protocol/capabilities of the device determine the meaning (i.e. a command or a parameter). Each specific 1-Wire® device has its own protocol and command list as shown on its datasheet.

Parameters:

in byte *data*- data to write to 1-Wire device

Example:

```
myWIRE10.WR1_Write(0x55)
```
