Program Structure

sub *name* (**in/out** *type var*) **end sub**	subroutine
task *name* **end task**	task
main **end main**	main program section
/* ***/**	multi line comment
//	single line comment

Data Types

byte	1 byte (-128 to 127)
word	2 byte (-32768 to 32767)

Custom Types

type *name* *type name* **end type**	Custom type declaration

Object Types

ADC	Analog input
CAN	Controller area network
DIO	Digital input/output
I2C	IIC
LCD	LCD
PWM	Pulse width modulation
RTI	Real time interrupt
SCI	Serial Port
SPI	SPI
SYSTEM	System
TIMIO	Timer subsystem
WDT	Watchdog timer
WIRE1	1-Wire

Variable/Object Declaration

variable declaration **dim** *name* **as new** *type*
object declaration **dim** *name* **as new** *object*(*pins*)

Flow Control

if statement

```
if (expr)
        statement
else
        statement
end if
```

while statement

```
while (expr)
        statement
end while
```

do statement

```
do
        statement
until(expr)
```

select statement

```
select var
        case expr:
                statement
        break
        case expr:
        case expr2:
                statement
        break
        default
                statement
        break
end select
```

for statement

```
for var = istart to iend step istep
        statement
next

for var = istart to iend
        statement
next
```

Operators

plus, minus	+, -
multiply, divide	*, /
modulus %	
left, right shift [bit ops]	<<, >>
comparisons	>, <, >=, <=, ==, !=
bitwise and, or, xor	&, |, ^
logical and, or	AND, OR

ADC

ADC(const <Pins>)
One or more analog input(s) can be specified in the constructor of an ADC object.
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
ADC_Done(out byte *done*)
Check if the ADC has finished the conversion.

CAN

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

DIO

DIO(const <Pins>)
DIO Constructor
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.
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.

I2C

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

LCD

LCD(const <LCD-D4>, const <LCD-D5>, const <LCD-D6>, const <LCD-D7>, const <LCD-E>, const <LCD-RS>)
LCD Constructor
LCD_Char(in byte *char*)
Place a character on the display, at the current cursor-position.
LCD_Command(const <LCD COMMAND>, in byte *addrdata*)
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.

PWM

PWM(const &ltPWM PIN>)
PWM Constructor
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.
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.

RTI

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

SCI

SCI(const <RX PIN>, const <TX PIN>)
SCI Constructor
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 &ltFILL 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.

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.

SPI

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

SPI Constructor

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.

SYSTEM

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 &ltKHz SPEED>)

Change the speed of the processor

Sleep(const <WAKEUP ON>)

Put the MCU to sleep.

TIMIO

TIMIO(const <PORTT PIN>)

TIMIO Constructor

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 OC 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 OutputCompare timers to control the behaviour of the specified pin.
TIMIO_Timer_start(const <RESOLUTION>)
Start central timer from which all TIMIO timing is derived.
TIMIO_Timer_stop()
Stop the central timer

WDT

WDT_set(const <WDT TIMEOUT>)
Calling this function will activate the watchdog (also called COP by Motorola).

WIRE1

WIRE1(const <Data Pin>)
WIRE1 Constructor
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.