New capabilities are:

This capability is an ISR. It is therefore highly reliant on processing time and the adherence with the I2C specification. The I2C handler should not user configurable and it is not intended to edited by the user.

As an ISR if the user turns on/off the interrupts in the user routines this WILL cause I2C protocol errors and overruns. Do not use interrupts without considering the impact on the I2C protocol.

When developing solutions adding serial or LCD output must be fully optimised to ensure I2C protocol, in terms of timing, is maintained. Longs strings will cause I2C overruns.

Interfacing to Slave attached devices is relatively simple. Do this in the user code.

**How does this work?**

You have a Master I2C device. It writes and reads using the I2C protocol.

The Slave responds to the writes and read. The Slave code exposed the message in a simple message queue. The user can examine the message queue and take appropriate action or return values to the Master.

**Hardware I2C ISR Handler for GCGB and GCB.**

This library provides an ISR to implement a stateful I2C hardware slave.

This is a GCB implementation of Microchip Application Note AN734. According to AN734, there are 5 possible I2C states. During ISR, each of this states are detected. This ISR provides a standard skeleton to implement an I2C hardware slaves, while client code must implement several callbacks the ISR is expecting to call while processing states.

Callbacks:

HI2CSlave\_State\_1 ( in I2Ctemp as byte ) - called when I2C address matches (master starts a talk)

HI2CSlave\_State\_2 ( in I2CByte as byte) - called when master is writing a byte. Slave is thus receiving this byte. This callback takes this bytes as argument

HI2CSlave\_State\_3 called when master wants to read a byte from slave. Thus, slave should send a byte using HI2C2Send

HI2CSlave\_State\_4 called when master still wants to read a byte from slave. That is, master required to read (state 3) and now still want to read a byte slave should send a byte using HI2C2Send

HI2CSlave\_State\_5 called when master does not want to talk to slave anymore usually a good place to reset data or slave's logic

HI2CSlave\_State\_Error called when something wrong happens. You can do what you want in this case, like resetting the PIC, log some information using usart, ... called any cases other than states 1, 2, 3, 4 or 5

Also, there is an enable the line below to enable testing of the clock stretching feature it will add an additional delay of 200us in the interrupt handler so were sure that clock stretching is required for 100 KHz I2C operation

**The I2C Message Handler**

The I2C Message Handler provides the interface for the user to supply a procedure to process the received message.

Basically, this I2C slave waits for a full message to arrive. Then it calls the User procedures to process the message and (optional) prepare a response. Subsequently, this library will pass the response data over to the master, if it wants to have them.

Callbacks:

HI2C\_Process\_In\_Message( HI2CIndex as byte ) called when I2C a Master starts a talk. HI2CIndex is the length of the incoming packet.

HI2C\_Process\_Outgoing\_Message called when I2C an Master needs to a talk.

The HI2CBUFFER should be populated with the data to be sent.

All other methods are Callbacks from the ISR routine.

**Demonstration Code**

The demonstration code creates I2C Slave on a specific I2C address.

The slave responds to a write of 4 bytes to I2C address 0x4C (which is really Read Address 0x4D) and the Slave responds with two data bytes – from two pots. The results are shown on a terminal.

Callbacks:

HI2C\_Process\_In\_Message ( in HI2CMESSAGESIZE as byte ) called when slave has a full buffer.

HI2C\_Process\_Out\_Message called when slave is requested to respond to a masters request.

Users should redirect all or any of the standard callbacks like HI2CSlave\_State\_Error, HI2CSlave\_State\_1-5 using #define to call specific routines.

#define HI2CSlave\_State\_Error MyHI2CSlave\_State\_Error

**Test Activities**

The tests showcase the new GCB capabilities are:

1. Demonstrate how to configure the I2C Slave address and to configure the I2C queue
2. Respond to I2C discovery with the response from the Write and Read addresses
3. Demonstration code on using how to use the I2C Slave using GCB Message Queue
   1. Receive I2C data from an I2C Master to this I2C Slave – with the I2C Slave handling the incoming via a Message Queue, and,
   2. Send I2C data from this I2C Slave to the I2C Master. The data being sent should be sensor data sourced from the Slave.

Test outcomes are:

1. Tested solution using 16f operating as I2C Slave – non PPS
2. Tested solution using 16f operating as I2C Slave – PPS

Test configuration:

1. Xpress board – 16lf18855 with I2C and Serial connectivity
2. 16f690 – with I2C and TTL Serial connectivity

Software Configuration

1. GCB v0.95.010
2. HW I2C ISR Library ( new .h ). This file is not intended to be edited by a user.
3. HW I2C Message Queue Library ( new .h ). This file is not intended to be edited by a user.
4. Example 16F demonstration code ( new )

Installation of new capabilities

Install GCB v0.95.010, copy the two .h into INCLUDE folder. These files are not intended to be edited by a user.

Using GCB v0.95.010, copy the hwI2C .h into INCLUDE\LOWLEVEL folder. This file is not intended to be edited by a user.

Install the Demo user GCB file into the any suitable code development folder. This file is intended to be edited by the user.

**Message Queue Structure**

Essentially the I2C messages are exposed in the **Message Queue**. The Message Queue is a buffer. The size of the buffer is user determined but this will be based on the maxiumum I2C message to be communicated.

The methods expose the buffer and with very simple GCB logic users can respond to the specifics of the I2C message.

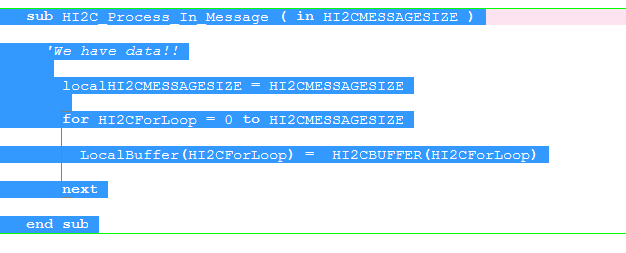
|  |  |  |
| --- | --- | --- |
| **Use Case** | **Sample Code** | **Queue Structure** |
| Master Write Operation | HI2CStart ;generate a start signal  HI2CSend(TargetGCBI2CAddress) ;WriteOp  HI2CSend(outvar) ;first value = 1  HI2CSend(outvar+1) ;then the value = 2  HI2CSend(outvar+2) ;then the value  HI2CSend(outvar+3) ;then the value  HI2CSend(outvar+4) ;then the value  HI2CSend(outvar+5) ;then the value=6  HI2CStop | **83:01:02:03:04:05:06:**  The queue is 7 bytes long. Address via  HI2CBUFFER(1) thru to HI2CBUFFER(N) where N is HI2CMESSAGESIZE.  HI2CMESSAGESIZE will confirm message size.  HI2CBUFFER(0) will contain the first sent byte (the Write instruction) |
| Master Write then Read (aka an EEPROM addressing scheme) | All Master messages of this type MUST be the same structure in terms of the length to the Send/Write component of the message.  HI2CStart ;generate a start signal  HI2CSend(TargetGCBI2CAddress) ;WriteOp  HI2CSend(outvar) ;first value = 1  HI2CReStart ;generate a restart signal  HI2CSend(TargetGCBI2CAddress XOR 1) ;inidicate a read  HI2CReceive(Val1, ACK) ;read one byte  HI2CReceive(Val2, NACK) ;read one byte and conclude  HI2CStop | **82**  The queue is 1 bytes long.    HI2CMESSAGESIZE will confirm message size.  Address via HI2CBUFFER(0)  Data to be READ/Published to the Master thru to HI2CBUFFER(1..N)  You **MUST ACK** and **NACK** the message. |

**Reading the Incoming Queue**

HI2C\_Process\_In\_Message ( in HI2CMESSAGESIZE as byte ) is called when slave has a full buffer.

The variable HI2CMESSAGESIZE will specify the buffer input queue length.

If the demo code the incoming queue is copied to a local queue for display purpose.

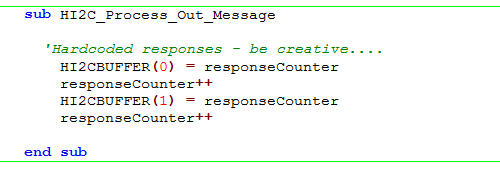


A use case could be read the byte pattern or first byte and respond with specific sensors results – which would be placed in the output queue.

**Writing to the outgoing Queue**

HI2C\_Process\_Out\_Message is called when slave is requested to respond to a masters request for data.

Data bytes should be placed in the buffer in the correct order where buffer element 0 is the first data byte transmitted.



The supporting libraries are intended to hide the complexities of the state engine, and the buffer is intended to make things easy.

**Use Case #1**

Emulate DS1307 where code would be

do

HI2CReStart ;generate a start signal

HI2CSend(AddrWrite) ;inidcate a write

loop While HI2CAckPollState

HI2CSend(0) ;begin with address 0

HI2CSend(DecToBcd(DS\_Sec)) ;then set the three

HI2CSend(DecToBcd(DS\_Min)) ;consecutive values

HI2CSend(DecToBcd(DS\_Hour))

HI2CStop

This could update the 3 memory addresses in the slave device. Slave code would be….

**Sub** HI2C\_Process\_In\_Message ( in HI2CMESSAGESIZE as byte )

Select CASE HI2CBUFFER[0]

Case 0x00

if (HI2CMESSAGESIZE = 4) then

SecVar = HI2CBUFFER[1]

MinVar = HI2CBUFFER[2]

HourVar= HI2CBUFFER[3]

end if

….

….

**Use Case #2**

Response to various message types. This is working/tested code.

sub HI2C\_Process\_In\_Message ( in HI2CMESSAGESIZE )

'We have data!!

Select CASE HI2CBUFFER(0)

Case 0x80

if (HI2CMESSAGESIZE = 1) then

' cmd 0x80 - request version

HI2CBUFFER(1) = REV\_HI\_BYTE 'application code version, write one

HI2CBUFFER(2) = REV\_LO\_BYTE 'byte, read three bytes

end if

CASE 0x81

if (HI2CMESSAGESIZE = 1) then

' cmd 0x81 - turn on power to temperature sensors

I2C\_TS\_EN = on 'power to temp sensors "on"

HI2CBUFFER(1) = 0xFF 'report status of power to TS

end if

CASE 0x82

if (HI2CMESSAGESIZE = 1) then

' cmd 0x82 - turn off power to temperature sensors

I2C\_TS\_EN = off ' power to temp sensors "off"

HI2CBUFFER(1) = 0x00

end if

CASE 0x55

' cmd 0x55 - set PWM duty on 7 channels

if (HI2CMESSAGESIZE = 8) then 'write 8 bytes, first command,

BB = HI2CBUFFER(1) ' seven bytes with values from

GG = HI2CBUFFER(2) '0 to 100 for setting PWM in persents.

HR = HI2CBUFFER(3) '85,xx,xx,xx,xx,xx,xx,xx"

EQW = HI2CBUFFER(4) ' read 8 bytes for verification

FR = HI2CBUFFER(5) '

WW = HI2CBUFFER(6) '

UV = HI2CBUFFER(7) '

end if

' cmd 0x83 - request User ID

CASE 0x83

if (HI2CMESSAGESIZE = 8) then

' User\_ID () 'Call procedure reading User ID to write 1 byte

BB = HI2CBUFFER(1) ' seven bytes with values from

GG = HI2CBUFFER(2) '0 to 100 for setting PWM in persents.

HR = HI2CBUFFER(3) '85,xx,xx,xx,xx,xx,xx,xx"

EQW = HI2CBUFFER(4) ' read 8 bytes for verification

FR = HI2CBUFFER(5) '

WW = HI2CBUFFER(6) '

UV = HI2CBUFFER(7) '

end if

End Select

end sub

The GCB code related to this is as follows:

do

HI2CStart ;generate a start signal

HI2CSend(TargetGCBI2CAddress) ;inidcate a write

I2CRetry++

loop While HI2CAckPollState and I2CRetry <> 255

HI2CSend(0X55)

HI2CSend(outvar) ;then the value

HI2CSend(outvar+1) ;then the value

HI2CSend(outvar+2) ;then the value

HI2CSend(outvar+3) ;then the value

HI2CSend(outvar+4) ;then the value

HI2CSend(outvar+5) ;then the value

HI2CSend(outvar+6) ;then the value

HI2CStop

outvar++

‘init the vars

eepromVal1 = 0X55

eepromVal2 = 0X55

do

HI2CReStart ;generate a start signal

HI2CSend(TargetGCBI2CAddress) ;indicate a write

loop While HI2CAckPollState

HI2CSend(0x80)

HI2CReStart

HI2CSend(TargetGCBI2CAddress XOR 1) ;set the read flag

HI2CReceive(eepromVal1, ACK) ;read one byte

HI2CReceive(eepromVal2, ACK) ;read one byte

HI2CReceive(eepromVal3, NACK) ;read one byte and conclude

HI2CStop

HSerPrint eepromVal1

HSerPrint ":"

HSerPrint eepromVal2

HSerPrint ":"

HSerPrint eepromVal3

HSerPrintCRLF

do

HI2CReStart ;generate a start signal

HI2CSend(TargetGCBI2CAddress) ;indicate a write

loop While HI2CAckPollState

HI2CSend(0x81) ;number of bytes to request or low address

HI2CReStart

HI2CSend(TargetGCBI2CAddress XOR 1) ;set the read flag

HI2CReceive(eepromVal1, ACK) ;read one byte

HI2CReceive(eepromVal2, NACK) ;read one byte

HI2CStop

HSerPrint eepromVal1

HSerPrint ":"

HSerPrint eepromVal2

HSerPrintCRLF

do

HI2CReStart ;generate a start signal

HI2CSend(TargetGCBI2CAddress) ;indicate a write

loop While HI2CAckPollState

HI2CSend(0x82)

HI2CReStart

HI2CSend(TargetGCBI2CAddress XOR 1) ;set the read flag

HI2CReceive(eepromVal1, ACK) ;read one byte

HI2CReceive(eepromVal2, NACK) ;read one byte

HI2CStop

HSerPrint eepromVal1

HSerPrint ":"

HSerPrint eepromVal2

HSerPrintCRLF

do

HI2CStart ;generate a start signal

HI2CSend(TargetGCBI2CAddress) ;indicate a write

I2CRetry++

loop While HI2CAckPollState and I2CRetry <> 255

HI2CSend(0X83)

HI2CSend(outvar) ;then the value

HI2CSend(outvar+1) ;then the value

HI2CSend(outvar+2) ;then the value

HI2CSend(outvar+3) ;then the value

HI2CSend(outvar+4) ;then the value

HI2CSend(outvar+5) ;then the value

HI2CSend(outvar+6) ;then the value

HI2CStop

outvar++

Results transmissions with the left hand terminal showing the Master results and the right terminal showing the slave received data.

