GyroAccGlove

1.0

Generated by Doxygen 1.7.3

Sat Sep 10 2011 17:54:13

Contents

1 Main Page

Browning Research Field Emitter Control and Measurement System.

Introduction This code is written in C++. The AVR tools support a limited set of C++ capabilities so there are no fancy constructs such as templates. C++ allows the high level features to be encapsulated into a class and used where needed. In most cases these classes are built around hardware resources. There is a class to work with IO Ports, one for HardwareSerial, etc.

Compiling The compiler and debug environment for the AVR tools is freely available. Several options exist, the simplest on is the AVR Studio. This tool can be downloaded from Atmel's web site. The tool runs on a Windows PC only.

For Unix or Macs there are freely available GNU toolchains. These do not include a GUI, but command line builds work just find.

Controller Board Hardwarew The hardware consists of the following comonents:

- Controller Board.
- Emitter Control Board
- Current Monitor Board

Controller Board Board for controlling all other components and interfacing to host computer.

Emitter Control Board Contains N-Channel FETS to control the current into the emitter elements.

Microprocessor The procssor on the board ia an Atmel ATxmega 128A1.

Some important links for this device are:

- Product Datasheet
- Product Manual
- Product Website

The product manual is very similar to the datasheet, however the manual contains register definitions. These are very important when configuring the hardware resources available within the ATxmega.

2 Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Class Documentation

5.1 CmdProcessor Class Reference

```
#include <CmdProcessor.h>
```

Public Member Functions

- CmdProcessor (HardwareSerial ∗pHW)

  Number of valid parameters.

- ∼CmdProcessor ()

  Destructor. Release memory allocated in constructor.

- bool checkCommands ()
- char ∗ cmdTerm ()

  Return pointer to termination string.

- void cmdTerm (char ∗)
- void resetCmd ()

  Clear the command status values so a new command can be started.

- const char ∗ cmdDelim ()

  Return current delimiter string.

- void cmdDelim (const char ∗)
- const char ∗ getCmd ()

  Return the command string.

- uint8_t paramCnt ()

  Return the number of parameters parsed from the current command.

Parameter Extraction Functions

getParam is overloaded on the variable type, this means that each possible type has a unique function. The type of the parameter you are seeking will determine the exact function that is called, which will then do the right thing to convert the string paramter value to an unsigned int, double etc.

- void getParam (uint8_t idx, uint8_t &p)

  Parse the index parameter into a unsigned 8 bit integer.

- void getParam (uint8_t idx, uint16_t &p)

  Parse the index parameter into a unsigned 16 bit integer.

- void getParam (uint8_t idx, long &l)

  Parse the index parameter into a unsigned 8 bit integer.

- void getParam (uint8_t idx, int &p)

  Parse the index parameter into a unsigned 8 bit integer.

- void getParam (uint8_t idx, double &f)

  Parse the index parameter into a double.

- void getParam (uint8_t idx, char ∗&p, uint8_t maxlen=128)

  Parse the index parameter into a string with the length specified.

Protected Member Functions

- void processCmd ()

Protected Attributes

- HardwareSerial ∗ _pHW
- char ∗ _pTokens [10]

  Store the serial object.

- char ∗ _pCmd

  List of command tokens.

- char ∗ _pCmdString

Command buffer.

- uint8_t _cmdPos

  Current command.

- bool _validCmd

  Current position during serial read.

- char ∗ _pCmdTerm

  Indicates a current valid command.

- char ∗ _pCmdDelim

  Store command terminator.

- uint8_t _paramCnt

  Current command parameter delimiter.

5.1.1 Detailed Description

Definition at line 7 of file CmdProcessor.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 CmdProcessor::CmdProcessor (HardwareSerial ∗ *pHW*)

Number of valid parameters.

Construct a new CmdProcessor. Pass in reference to the HardwareSerial class to use for command processing. Store the serial pointer and then initialize the internal data strings used during command input processing and output processing.

Definition at line 11 of file CmdProcessor.cpp.

References _pCmd, _pCmdDelim, _pCmdString, _pCmdTerm, _pHW, and resetCmd().

```
{
    _pHW = pHW;

    _pCmdString = (char*)malloc(128);
    _pCmd = 0;
    _pCmdTerm = (char*)malloc(3);
    strcpy(_pCmdTerm,"\n\r");
    _pCmdDelim = (char*)malloc(3);
    strcpy(_pCmdDelim," \t");
    resetCmd();
}
```

5.1.2.2 CmdProcessor::∼CmdProcessor ( )

Destructor. Release memory allocated in constructor.

Definition at line 25 of file CmdProcessor.cpp.

References _pCmdDelim, _pCmdString, _pCmdTerm, _pHW, and HardwareSerial::end().

```
{
    if (_pHW) {
        _pHW->end();
    }
    free(_pCmdString);
    free(_pCmdDelim);
    free(_pCmdTerm);
}
```

5.1.3 Member Function Documentation

5.1.3.1 bool CmdProcessor::checkCommands ( )

Read new characters from the serial port

Read any new characters into the command buffer. Look for the command terminator. If the terminator is found, store the command, process the command buffer and return 1 to indicate that a new command is availble. If a full command is not yet present, then return zero.

Definition at line 68 of file CmdProcessor.cpp.

References _cmdPos, _pCmdString, _pCmdTerm, _pHW, HardwareSerial::available(), Print::print(), processCmd(), and HardwareSerial::read().

```
{
    while (_pHW->available() > 0) {
        unsigned char c = _pHW->read();
        if (strchr(_pCmdTerm,c) != 0) {
            if (_cmdPos > 0) {
                // Done with this command.
                _pCmdString[_cmdPos] = 0; // Null terminate command
                processCmd();
                return 1;
            } else {
                _pHW->print("Ok\n");
            }
        } else {
            _pCmdString[_cmdPos++] = c;
        }
    }
    return 0;
}
```

5.1.3.2 const char ∗ CmdProcessor::cmdDelim ( )

Return current delimiter string.

Definition at line 48 of file CmdProcessor.cpp.

References _pCmdDelim.

```
{
    return _pCmdDelim;
}
```

5.1.3.3 void CmdProcessor::cmdDelim (const char ∗ *d*)

Set new delimiter string. Free memory, allocate new memory and copy new value.

Definition at line 55 of file CmdProcessor.cpp.

References _pCmdDelim.

```
{
    free(_pCmdDelim);
    _pCmdDelim = (char*)malloc(strlen(d) + 1);
    strcpy(_pCmdDelim,d);
}
```

5.1.3.4 char ∗ CmdProcessor::cmdTerm ( )

Return pointer to termination string.

Definition at line 36 of file CmdProcessor.cpp.

References _pCmdTerm.

```
{ return _pCmdTerm; }
```

5.1.3.5 void CmdProcessor::cmdTerm (char ∗ *t*)

Set a new command terminator. Free memory for previous value, allocate new memory and save the new value.

Definition at line 40 of file CmdProcessor.cpp.

References _pCmdTerm.

```
{
    free(_pCmdTerm);
    _pCmdTerm = (char*)malloc(strlen(t) + 1);
    strcpy(_pCmdTerm,t);
}
```

5.1.3.6 const char ∗ CmdProcessor::getCmd ( )

Return the command string.

Definition at line 123 of file CmdProcessor.cpp.

References _pCmd.

```
{
    return _pCmd;
}
```

5.1.3.7 void CmdProcessor::getParam (uint8_t *idx*, double & *f*)

Parse the index parameter into a double.

Definition at line 176 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        uint8_t nScans;
        nScans = sscanf(_pTokens[idx],"%lf", &p);
        //p = atof(_pTokens[idx]);
    }
}
```

5.1.3.8 void CmdProcessor::getParam (uint8_t *idx*, uint8_t & *p*)

Parse the index parameter into a unsigned 8 bit integer.

Definition at line 154 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}
```

5.1.3.9 void CmdProcessor::getParam (uint8_t *idx*, uint16_t & *p*)

Parse the index parameter into a unsigned 16 bit integer.

Definition at line 146 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}
```

5.1.3.10 void CmdProcessor::getParam (uint8_t *idx*, int & *p*)

Parse the index parameter into a unsigned 8 bit integer.

Definition at line 161 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}
```

5.1.3.11 void CmdProcessor::getParam (uint8_t *idx*, char ∗& *p*, uint8_t *maxlen* = `128`)

Parse the index parameter into a string with the length specified.

Definition at line 186 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        strncpy(p,_pTokens[idx],maxlen);
    }
}
```

5.1.3.12 void CmdProcessor::getParam (uint8_t *idx*, long & *l*)

Parse the index parameter into a unsigned 8 bit integer.

Definition at line 168 of file CmdProcessor.cpp.

References _paramCnt, and _pTokens.

```
{
    if (idx < _paramCnt) {
        l = atol(_pTokens[idx]);
    }
}
```

5.1.3.13 uint8_t CmdProcessor::paramCnt ( )

Return the number of parameters parsed from the current command.

Definition at line 129 of file CmdProcessor.cpp.

References _paramCnt.

```
{
    return _paramCnt;
}
```

5.1.3.14 void CmdProcessor::processCmd ( ) [protected]

Process the commands in the command buffer Split the command into parameters based on the command delimiter. The maximum number of command tokens is 10.

Definition at line 92 of file CmdProcessor.cpp.

References _paramCnt, _pCmd, _pCmdDelim, _pCmdString, _pTokens, and _validCmd.

Referenced by checkCommands().

```
{
    // See if the command delimiter exists in the
    // command. if it does not, then the command
    // is the entire string.
    if (strpbrk(_pCmdString,_pCmdDelim)) {
        _pCmd = strtok(_pCmdString,_pCmdDelim);
        char* pTok = strtok(0,_pCmdDelim);
        int i = 0;
        while (i < 10 && pTok) {
            _pTokens[i++] = pTok;
            pTok = strtok(0,_pCmdDelim);
        }
        _paramCnt = i;
        _validCmd = true;
    } else {
        _pCmd = _pCmdString;
        _paramCnt = 0;
        _validCmd = true;
    }
}
```

5.1.3.15 void CmdProcessor::resetCmd ( )

Clear the command status values so a new command can be started.

Definition at line 115 of file CmdProcessor.cpp.

References _cmdPos, _paramCnt, and _validCmd.

Referenced by CmdProcessor().

```
{
    _cmdPos = 0;
    _validCmd = false;
    _paramCnt = 0;
}
```

5.1.4 Member Data Documentation

5.1.4.1 uint8_t CmdProcessor::_cmdPos [protected]

Current command.

Definition at line 14 of file CmdProcessor.h.

Referenced by checkCommands(), and resetCmd().

5.1.4.2 uint8_t CmdProcessor::_paramCnt [protected]

Current command parameter delimiter.

Definition at line 18 of file CmdProcessor.h.

Referenced by getParam(), paramCnt(), processCmd(), and resetCmd().

5.1.4.3 char* CmdProcessor::_pCmd [protected]

List of command tokens.

Definition at line 12 of file CmdProcessor.h.

Referenced by CmdProcessor(), getCmd(), and processCmd().

5.1.4.4 char* CmdProcessor::_pCmdDelim [protected]

Store command terminator.

Definition at line 17 of file CmdProcessor.h.

Referenced by cmdDelim(), CmdProcessor(), processCmd(), and ∼CmdProcessor().

5.1.4.5 char∗ CmdProcessor::_pCmdString `[protected]`

Command buffer.

Definition at line 13 of file CmdProcessor.h.

Referenced by checkCommands(), CmdProcessor(), processCmd(), and ∼CmdProcessor().

5.1.4.6 char∗ CmdProcessor::_pCmdTerm `[protected]`

Indicates a current valid command.

Definition at line 16 of file CmdProcessor.h.

Referenced by checkCommands(), CmdProcessor(), cmdTerm(), and ∼CmdProcessor().

5.1.4.7 HardwareSerial∗ CmdProcessor::_pHW `[protected]`

Definition at line 10 of file CmdProcessor.h.

Referenced by checkCommands(), CmdProcessor(), and ∼CmdProcessor().

5.1.4.8 char∗ CmdProcessor::_pTokens[10] `[protected]`

Store the serial object.

Definition at line 11 of file CmdProcessor.h.

Referenced by getParam(), and processCmd().

5.1.4.9 bool CmdProcessor::_validCmd `[protected]`

Current position during serial read.

Definition at line 15 of file CmdProcessor.h.

Referenced by processCmd(), and resetCmd().

The documentation for this class was generated from the following files:

- CmdProcessor.h

- CmdProcessor.cpp

5.2 Fifo Class Reference

Fifo Class for unsigned 8 bit values.

```
#include <fifo.h>
```

Public Types

- typedef uint8_t FifoType

Public Member Functions

- Fifo (uint8_t size)
- int8_t push (FifoType ∗)
- int8_t pop (FifoType ∗pData)
- uint8_t count ()
- bool full ()

  Return true if the fifo is full.

- bool empty ()

  Return true if the fifo is empty.

- void clear ()

  Clear the fifo by resetting the start and end pointer.

Private Attributes

- FifoType ∗ _pdata
- FifoType ∗ _start
- FifoType ∗ _end
- uint8_t _size

5.2.1 Detailed Description

Fifo Class for unsigned 8 bit values. Construct a fifo and specify the number of elements to store. The fifo constructor will allocate memory for the specified number of values. The Fifo class contains member functions for pusshing, popping and checking the status of the fifo.

Definition at line 16 of file fifo.h.

5.2.2 Member Typedef Documentation

5.2.2.1 typedef uint8_t Fifo::FifoType

Definition at line 20 of file fifo.h.

5.2.3 Constructor & Destructor Documentation

5.2.3.1 Fifo::Fifo (uint8_t *size*)

Construct the fifo object. Allocate memory for the specified number of elements and set the internal value to indicate the size of the fifo. Reset the start and end data points to their clear state. The clear function is called to maintain consitency and insure that clear() always does the right thing.

Definition at line 14 of file fifo.cpp.

References _pdata, _size, and clear().

```
{
    _size = size;
    _pdata = (FifoType*)malloc(_size * sizeof(FifoType));
    clear();
}
```

5.2.4 Member Function Documentation

5.2.4.1 void Fifo::clear ( )

Clear the fifo by resetting the start and end pointer.

Definition at line 22 of file fifo.cpp.

References _end, _pdata, and _start.

Referenced by Fifo(), and FifoTest().

```
{
    _start = _end = _pdata;
}
```

5.2.4.2 uint8_t Fifo::count ( )

Return the number of elements currently in the fifo if the end and start pointers are the same then the fifo is empty and count == 0. If they differ, then we need to check for wrap-around in order to properly determine the size. In the following examples a = marks empty spots, while an x marks filled spots.

```
       s                       e
=======xxxxxxxxxxxxxxxxxxxxxxxxx========================
```

In this case end > start, so count is equal to the distance between them or $end - start$.

```
       e                       s
xxxxxxxx=======================xxxxxxxxxxxxxxxxxxxxxxxxx
```

In this case end < start, so data wraps around. The total count is equal to the size of the buffer, minus the number of blank spots, or $size - (start - end)$.

The total number of possible elements that can be stored is size -1, so

Definition at line 50 of file fifo.cpp.

References _end, _size, and _start.

Referenced by FifoTest().

```
{
    if (_end == _start) return 0;
    if (_end > _start) {
        return _end - _start;
    }
    return _size - (_start - _end);
}
```

5.2.4.3 bool Fifo::empty ( )

Return true if the fifo is empty.

Definition at line 66 of file fifo.cpp.

References _end, and _start.

Referenced by pop().

```
{
    return (_start == _end);
}
```

5.2.4.4 bool Fifo::full ( )

Return true if the fifo is full.

Definition at line 60 of file fifo.cpp.

References _end, _size, and _start.

Referenced by push().

```
{
    return (_start - _end)  == 1 || (_end - _start) == _size;
}
```

5.2.4.5 int8_t Fifo::pop (FifoType ∗ *pD*)

Remove the top value from the Fifo. We do not have exceptions in this simple C++ implementation, so this function is not able to do anything to indicate that the called tried to pop a value from an empty fifo. In that case, a zero value is returned, which is not unique so the caller will have to insure that pop is never called on an empty fifo.

Definition at line 92 of file fifo.cpp.

References _pdata, _size, _start, and empty().

Referenced by FifoTest().

```
{
    if (empty()) {
        return -1; // Nothing else to do
    }
    *pD = *(_start++);
    if ((_start - _pdata) > _size) {
        _start = _pdata;
    }
    return 0;
}
```

5.2.4.6 int8_t Fifo::push (FifoType ∗ *d*)

Push a new value onto the fifo. This function returns 0 if the operation succeeds, and a negative value if the operation fails.

Definition at line 74 of file fifo.cpp.

References _end, _pdata, _size, and full().

Referenced by FifoTest().

```
{
    if (full()) return -1;
    *(_end++) = *d;

    // Wrap the end back to the beginning.
    if ((_end - _pdata) > _size) {
        _end = _pdata;
    }

        return 0;
}
```

5.2.5 Member Data Documentation

5.2.5.1 FifoType∗ Fifo::_end [private]

Definition at line 25 of file fifo.h.

Referenced by clear(), count(), empty(), full(), and push().

5.2.5.2 FifoType∗ Fifo::_pdata [private]

Definition at line 23 of file fifo.h.

Referenced by clear(), Fifo(), pop(), and push().

5.2.5.3 uint8_t Fifo::_size [private]

Definition at line 26 of file fifo.h.

Referenced by count(), Fifo(), full(), pop(), and push().

5.2.5.4 FifoType∗ Fifo::_start [private]

Definition at line 24 of file fifo.h.

Referenced by clear(), count(), empty(), full(), and pop().

The documentation for this class was generated from the following files:

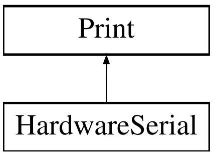
- fifo.h
- fifo.cpp

5.3 HardwareSerial Class Reference

HardwareSerial implementation.

```
#include <HardwareSerial.h>
```

Inheritance diagram for HardwareSerial:



Public Member Functions

- HardwareSerial (USART_t ∗usart, PORT_t ∗port, uint8_t in_bm, uint8_t out_bm)
- ∼HardwareSerial ()
- void begin (long baudrate, int8_t bscale=0)
- void begin2x (long baudrate, int8_t bscale=0)
- void end ()
- uint8_t available (void)
- int read (void)
- void flush (void)
- virtual void write (uint8_t)
- void enable (bool bEn)

Interrupt Handlers

There are three possible interrupts for the USART. Receive done, Transmit done and Data Register Ready.

- void rxc ()
- void dre ()
- void txc ()

Protected Attributes

- ring_buffer ∗ _rx_buffer
- USART_t ∗ _usart
- PORT_t ∗ _port
- uint8_t _in_bm
- uint8_t _out_bm
- uint8_t _bsel
- int8_t _bscale
- long _baudrate
- bool _bEn

5.3.1 Detailed Description

HardwareSerial implementation. This class was originally copied form the Arduino source directory but has been modified somewhat to customize it for the CFA project.

Ths class wraps the hardware serial resource in the ATXmega The class handles an interupt driven receive with a fixed size receive buffer of 128 bytes. The current implementation uses a synchronous send, but a buffered send would be a great enhancement for performance purposes.

Definition at line 23 of file HardwareSerial.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 HardwareSerial::HardwareSerial (USART_t ∗ *usart,* PORT_t ∗ *port,* uint8_t *in_bm,* uint8_t *out_bm*)

Definition at line 112 of file HardwareSerial.cpp.

References _baudrate, _bEn, _bscale, _bsel, _in_bm, _out_bm, _port, _rx_buffer, _usart, RX_BUFFER_SIZE, and SetPointer().

```
{
    _rx_buffer = (ring_buffer*)malloc(RX_BUFFER_SIZE+2*sizeof(int));
    _usart      = usart;
    _port       = port;
    _in_bm      = in_bm;
    _out_bm     = out_bm;
    _bsel       = 0;
    _bscale     = 0;
    _baudrate   = 9600;
    _bEn        = true;
    SetPointer(_usart,this);
}
```

5.3.2.2 HardwareSerial::∼HardwareSerial ( )

Definition at line 130 of file HardwareSerial.cpp.

References _rx_buffer, _usart, end(), and SetPointer().

```
{
    end();
    free(_rx_buffer);
    _rx_buffer = 0;
    SetPointer(_usart,0);
}
```

5.3.3 Member Function Documentation

5.3.3.1 uint8_t HardwareSerial::available (void)

Definition at line 213 of file HardwareSerial.cpp.

References _rx_buffer, ring_buffer::head, RX_BUFFER_SIZE, and ring_buffer::tail.

Referenced by CmdProcessor::checkCommands().

```
{
    return (RX_BUFFER_SIZE + _rx_buffer->head - _rx_buffer->tail) %
      RX_BUFFER_SIZE;
}
```

5.3.3.2 void HardwareSerial::begin (long *baudrate*, int8_t *bscale* = `0`)

Definition at line 140 of file HardwareSerial.cpp.

References _baudrate, _bscale, _in_bm, _out_bm, _port, and _usart.

Referenced by main().

```
{
    uint16_t BSEL;
    _bscale = bscale;
    _baudrate = baud;

    float fPER = F_CPU;
    float fBaud = baud;

    _port->DIRCLR = _in_bm;  // input
    _port->DIRSET = _out_bm; // output

    // set the baud rate
    if (bscale >= 0) {
        BSEL = fPER/((1 << bscale) * 16 * baud) - 1;
        //BSEL = F_CPU / 16 / baud - 1;
    } else {
        bscale = -1 * bscale;
        BSEL = (1 << bscale) * (fPER/(16.0 * fBaud) - 1);
    }

    _usart->BAUDCTRLA = (uint8_t)BSEL;
    _usart->BAUDCTRLB = ((bscale & 0xf) << 4) | ((BSEL & 0xf00) >> 8);

    // enable Rx and Tx
    _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
    // enable interrupt
    _usart->CTRLA = USART_RXCINTLVL_HI_gc;

    // Char size, parity and stop bits: 8N1
    _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
}
```

5.3.3.3 void HardwareSerial::begin2x (long *baudrate*, int8_t *bscale* = `0`)

Definition at line 173 of file HardwareSerial.cpp.

References _baudrate, _bscale, _in_bm, _out_bm, _port, _usart, and SetPointer().

```
{
    uint16_t baud_setting;
    _bscale = bscale;
    _baudrate = baud;

    // TODO: Serial. Fix serial double clock.
    long fPER = F_CPU * 4;

    _port->DIRCLR = _in_bm;  // input
    _port->DIRSET = _out_bm; // output

    // set the baud rate using the 2X calculations
    _usart->CTRLB |= 1 << 1; // the last 1 was the _u2x value
    baud_setting = fPER / 8 / baud - 1;

    _usart->BAUDCTRLA = (uint8_t)baud_setting;
    _usart->BAUDCTRLB = baud_setting >> 8;


    // enable Rx and Tx
    _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
    // enable interrupt
    _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
      ;

    // Char size, parity and stop bits: 8N1
    _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
    SetPointer(_usart,this);
}
```

5.3.3.4 void HardwareSerial::dre ( )

Definition at line 100 of file HardwareSerial.cpp.

```
{
}
```

5.3.3.5 void HardwareSerial::enable (bool *bEn*)

Definition at line 248 of file HardwareSerial.cpp.

References _bEn.

Referenced by main().

```
{
    _bEn = bEn;
}
```

5.3.3.6 void HardwareSerial::end ( )

Definition at line 203 of file HardwareSerial.cpp.

References _usart, and SetPointer().

Referenced by CmdProcessor::∼CmdProcessor(), and ∼HardwareSerial().

```
{
    SetPointer(_usart,(HardwareSerial*)0);

    // disable Rx and Tx
    _usart->CTRLB &= ~(USART_RXEN_bm | USART_TXEN_bm);
    // disable interrupt
    _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
      ;
}
```

5.3.3.7 void HardwareSerial::flush (void)

Definition at line 230 of file HardwareSerial.cpp.

References _rx_buffer, ring_buffer::head, and ring_buffer::tail.

```
{
    // don't reverse this or there may be problems if the RX interrupt
    // occurs after reading the value of rx_buffer_head but before writing
    // the value to rx_buffer_tail; the previous value of rx_buffer_head
    // may be written to rx_buffer_tail, making it appear as if the buffer
    // were full, not empty.
    _rx_buffer->head = _rx_buffer->tail;
}
```

5.3.3.8 int HardwareSerial::read (void)

Definition at line 218 of file HardwareSerial.cpp.

References _rx_buffer, ring_buffer::buffer, ring_buffer::head, RX_BUFFER_SIZE, and ring_buffer::tail.

Referenced by CmdProcessor::checkCommands().

```
{
    // if the head isn't ahead of the tail, we don't have any characters
    if (_rx_buffer->head == _rx_buffer->tail) {
        return -1;
    } else {
        unsigned char c = _rx_buffer->buffer[_rx_buffer->tail];
        _rx_buffer->tail = (_rx_buffer->tail + 1) % RX_BUFFER_SIZE;
        return c;
    }
}
```

5.3.3.9 void HardwareSerial::rxc ( )

Definition at line 94 of file HardwareSerial.cpp.

References _rx_buffer, _usart, and store_char().

```
{
    unsigned char c = _usart->DATA;
    store_char(c,_rx_buffer);
}
```

5.3.3.10 void HardwareSerial::txc ( )

Definition at line 104 of file HardwareSerial.cpp.

```
{
}
```

5.3.3.11 void HardwareSerial::write (uint8_t *c*) [virtual]

Implements Print.

Definition at line 240 of file HardwareSerial.cpp.

References _bEn, and _usart.

```
{
    if (_bEn) {
        while ( !(_usart->STATUS & USART_DREIF_bm) );
        _usart->DATA = c;
    }
}
```

5.3.4 Member Data Documentation

5.3.4.1 long HardwareSerial::_baudrate [protected]

Definition at line 33 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.2 bool HardwareSerial::_bEn [protected]

Definition at line 34 of file HardwareSerial.h.

Referenced by enable(), HardwareSerial(), and write().

5.3.4.3 int8_t HardwareSerial::_bscale [protected]

Definition at line 32 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.4 uint8_t HardwareSerial::_bsel [protected]

Definition at line 31 of file HardwareSerial.h.

Referenced by HardwareSerial().

5.3.4.5 uint8_t HardwareSerial::_in_bm [protected]

Definition at line 29 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.6 uint8_t HardwareSerial::_out_bm [protected]

Definition at line 30 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.7 PORT_t* HardwareSerial::_port [protected]

Definition at line 28 of file HardwareSerial.h.

Referenced by begin(), begin2x(), and HardwareSerial().

5.3.4.8 ring_buffer* HardwareSerial::_rx_buffer [protected]

Definition at line 26 of file HardwareSerial.h.

Referenced by available(), flush(), HardwareSerial(), read(), rxc(), and ~HardwareSerial().

5.3.4.9 USART_t* HardwareSerial::_usart [protected]

Definition at line 27 of file HardwareSerial.h.

Referenced by begin(), begin2x(), end(), HardwareSerial(), rxc(), write(), and ∼HardwareSerial().

The documentation for this class was generated from the following files:

- HardwareSerial.h
- HardwareSerial.cpp

5.4 I2C_Master Class Reference

```
#include <I2C_Master.h>
```

Public Types

- enum DriverState {
  sIdle, sBusy, sError, sArb,
  sIDScan, sIDCheck }
- enum DriverResult {
  rOk, rFail, rArbLost, rBussErr,
  rNack, rBufferOverrun, rUnknown, rTimeout }
- enum ErrorType {
  eNone = 0, eDisabled = -1, eBusy = -2, eNack = -3,
  eArbLost = -4, eBusErr = -5, eTimeout = -6, eSDAStuck = -7,
  eSCLStuck = -8, eUnknown = -9 }
- typedef enum I2C_Master::ErrorType ErrorType

Public Member Functions

- I2C_Master (TWI_t ∗twi)
- ∼I2C_Master ()
- void begin (uint32_t freq)
- void end ()
- ErrorType Write (uint8_t ID, uint8_t ∗Data, uint8_t nBytes)
- ErrorType WriteSync (uint8_t ID, uint8_t ∗Data, uint8_t nBytes)
- ErrorType Read (uint8_t ID, uint8_t nBytes)
- ErrorType ReadSync (uint8_t ID, uint8_t nBytes)
- ErrorType WriteRead (uint8_t ID, uint8_t ∗wrData, uint8_t nWriteBytes, uint8_t nReadBytes)
- ErrorType WriteReadSync (uint8_t ID, uint8_t ∗wrData, uint8_t nWriteBytes, uint8_t nReadBytes)
- void master_int ()
- void slave_int ()
- void WriteHandler ()
- void ReadHandler ()

- void ArbHandler ()
- void ErrorHandler ()
- void MasterFinished ()
- int testack (uint8_t ID)
- void dumpregs ()
- I2C_Master::DriverResult Result ()
- I2C_Master::DriverState State ()
- uint8_t ReadData (uint8_t ∗pData, uint8_t maxcnt)
- uint8_t ReadData (uint8_t index)
- uint8_t nReadBytes ()
- ErrorType CheckID (uint8_t ID)
- void Stop ()
- ErrorType ForceStartStop ()
- ErrorType WigglePin (uint8_t cnt, uint8_t pinSel, uint8_t otherState)
- void CleanRegs ()
- void loop ()
- bool busy ()
- void ∗ isReserved ()
- bool Reserve (void ∗)
- void NotifyMe (I2CNotify ∗pMe)
- bool IsIdle ()

Protected Member Functions

- uint8_t busState ()
- void showstate ()

Private Attributes

- TWI_t ∗ _twi
- PORT_t ∗ _twiPort
- bool _bEnabled
- DriverState _State
- DriverResult _Result
- void ∗ _pReserved
- I2CNotify ∗ _pNotifyClient
- uint8_t _DeviceID
- uint8_t _nBytesWritten
- uint8_t _nWriteBytes
- uint8_t _nReadBytes
- uint8_t _nBytesRead
- uint8_t ∗ _WriteData
- uint8_t _wrBufferLen
- uint8_t ∗ _ReadData
- uint8_t _rdBufferLen
- uint8_t _idScanCurrent
- uint8_t _IDList [128]
- bool _ScanComplete

5.4.1 Detailed Description

Definition at line 25 of file I2C_Master.h.

5.4.2 Member Typedef Documentation

5.4.2.1 typedef enum I2C_Master::ErrorType I2C_Master::ErrorType

5.4.3 Member Enumeration Documentation

5.4.3.1 enum I2C_Master::DriverResult

Enumerator:

- ***rOk***
- ***rFail***
- ***rArbLost***
- ***rBussErr***
- ***rNack***
- ***rBufferOverrun***
- ***rUnknown***
- ***rTimeout***

Definition at line 37 of file I2C_Master.h.

```
                 {
        rOk,
        rFail,
        rArbLost,
        rBussErr,
        rNack,
        rBufferOverrun,
        rUnknown,
        rTimeout
    } DriverResult;
```

5.4.3.2 enum I2C_Master::DriverState

Enumerator:

- ***sIdle***

sBusy

sError

sArb

sIDScan

sIDCheck

Definition at line 28 of file I2C_Master.h.

```
                 {
        sIdle,
        sBusy,
        sError,
        sArb,
        sIDScan,
        sIDCheck
    } DriverState;
```

5.4.3.3 enum I2C_Master::ErrorType

Enumerator:

eNone

eDisabled

eBusy

eNack

eArbLost

eBusErr

eTimeout

eSDAStuck

eSCLStuck

eUnknown

Definition at line 78 of file I2C_Master.h.

```
                           {
        eNone          = 0,
        eDisabled      = -1,
        eBusy          = -2,
        eNack          = -3,
        eArbLost       = -4,
        eBusErr        = -5,
        eTimeout       = -6,
        eSDAStuck      = -7,
        eSCLStuck      = -8,
        eUnknown       = -9
    } ErrorType;
```

5.4.4 Constructor & Destructor Documentation

5.4.4.1 I2C_Master::I2C_Master (TWI_t ∗ *twi*)

5.4.4.2 I2C_Master::~I2C_Master ( )

5.4.5 Member Function Documentation

5.4.5.1 void I2C_Master::ArbHandler ( )

5.4.5.2 void I2C_Master::begin (uint32_t *freq*)

Referenced by main(), and IMU::Reset().

5.4.5.3 uint8_t I2C_Master::busState ( ) `[protected]`

5.4.5.4 bool I2C_Master::busy ( )

Referenced by IMU::Run().

5.4.5.5 ErrorType I2C_Master::CheckID (uint8_t *ID*)

Referenced by IMU::CheckIDs(), and IMU::QueryChannels().

5.4.5.6 void I2C_Master::CleanRegs ( )

5.4.5.7 void I2C_Master::dumpregs ( )

5.4.5.8 void I2C_Master::end ( )

Referenced by IMU::Reset().

5.4.5.9 void I2C_Master::ErrorHandler ( )

5.4.5.10 ErrorType I2C_Master::ForceStartStop ( )

Referenced by IMU::ForceStartStop().

5.4.5.11 bool I2C_Master::IsIdle ( ) [inline]

Definition at line 153 of file I2C_Master.h.

References _twi.

```
{
    return (_twi->MASTER.STATUS & TWI_MASTER_BUSSTATE_gm)
        == TWI_MASTER_BUSSTATE_IDLE_gc;
}
```

5.4.5.12 void* I2C_Master::isReserved ( )

5.4.5.13 void I2C_Master::loop ( )

5.4.5.14 void I2C_Master::master_int ( )

5.4.5.15 void I2C_Master::MasterFinished ( )

5.4.5.16 void I2C_Master::NotifyMe (I2CNotify ∗ *pMe*)

Referenced by IMU::IMU(), and IMU::Reset().

5.4.5.17 uint8_t I2C_Master::nReadBytes ( )

5.4.5.18 ErrorType I2C_Master::Read (uint8_t *ID,* uint8_t *nBytes*)

5.4.5.19 uint8_t I2C_Master::ReadData (uint8_t ∗ *pData,* uint8_t *maxcnt*)

Referenced by IMU::Rd(), IMU::ReadWord(), IMU::StoreAccData(), and IMU::StoreGyroData().

5.4.5.20 uint8_t I2C_Master::ReadData (uint8_t *index*)

5.4.5.21 void I2C_Master::ReadHandler ( )

5.4.5.22 ErrorType I2C_Master::ReadSync (uint8_t *ID,* uint8_t *nBytes*)

5.4.5.23 bool I2C_Master::Reserve (void ∗)

5.4.5.24 I2C_Master::DriverResult I2C_Master::Result ( )

5.4.5.25 void I2C_Master::showstate ( ) [protected]

5.4.5.26 void I2C_Master::slave_int ( )

5.4.5.27 I2C_Master::DriverState I2C_Master::State ( )

5.4.5.28 void I2C_Master::Stop ( )

Referenced by IMU::ResetDevices().

5.4.5.29 int I2C_Master::testack (uint8_t *ID*)

5.4.5.30 ErrorType I2C_Master::WigglePin (uint8_t *cnt*, uint8_t *pinSel*, uint8_t *otherState*)

Referenced by IMU::FailRecovery().

5.4.5.31 ErrorType I2C_Master::Write (uint8_t *ID*, uint8_t ∗ *Data*, uint8_t *nBytes*)

5.4.5.32 void I2C_Master::WriteHandler ( )

5.4.5.33 ErrorType I2C_Master::WriteRead (uint8_t *ID*, uint8_t ∗ *wrData*, uint8_t *nWriteBytes*, uint8_t *nReadBytes*)

Referenced by IMU::RdAsync(), and IMU::WrAsync().

5.4.5.34 ErrorType I2C_Master::WriteReadSync (uint8_t *ID*, uint8_t ∗ *wrData*, uint8_t *nWriteBytes*, uint8_t *nReadBytes*)

Referenced by IMU::Rd().

5.4.5.35 ErrorType I2C_Master::WriteSync (uint8_t *ID*, uint8_t ∗ *Data*, uint8_t *nBytes*)

Referenced by IMU::Wr().

5.4.6 Member Data Documentation

5.4.6.1 bool I2C_Master::_bEnabled [private]

Definition at line 51 of file I2C_Master.h.

5.4.6.2 uint8_t I2C_Master::_DeviceID [private]

Definition at line 58 of file I2C_Master.h.

5.4.6.3 uint8_t I2C_Master::_IDList[128] [private]

Definition at line 73 of file I2C_Master.h.

5.4.6.4 uint8_t I2C_Master::_idScanCurrent [private]

Definition at line 72 of file I2C_Master.h.

5.4.6.5 uint8_t I2C_Master::_nBytesRead [private]

Definition at line 62 of file I2C_Master.h.

5.4.6.6 uint8_t I2C_Master::_nBytesWritten [private]

Definition at line 59 of file I2C_Master.h.

5.4.6.7 uint8_t I2C_Master::_nReadBytes `[private]`

Definition at line 61 of file I2C_Master.h.

5.4.6.8 uint8_t I2C_Master::_nWriteBytes `[private]`

Definition at line 60 of file I2C_Master.h.

5.4.6.9 I2CNotify∗ I2C_Master::_pNotifyClient `[private]`

Definition at line 55 of file I2C_Master.h.

5.4.6.10 void∗ I2C_Master::_pReserved `[private]`

Definition at line 54 of file I2C_Master.h.

5.4.6.11 uint8_t I2C_Master::_rdBufferLen `[private]`

Definition at line 67 of file I2C_Master.h.

5.4.6.12 uint8_t∗ I2C_Master::_ReadData `[private]`

Definition at line 66 of file I2C_Master.h.

5.4.6.13 DriverResult I2C_Master::_Result `[private]`

Definition at line 53 of file I2C_Master.h.

5.4.6.14 bool I2C_Master::_ScanComplete `[private]`

Definition at line 74 of file I2C_Master.h.

5.4.6.15 DriverState I2C_Master::_State [private]

Definition at line 52 of file I2C_Master.h.

5.4.6.16 TWI_t* I2C_Master::_twi [private]

Definition at line 49 of file I2C_Master.h.

Referenced by IsIdle().

5.4.6.17 PORT_t* I2C_Master::_twiPort [private]

Definition at line 50 of file I2C_Master.h.

5.4.6.18 uint8_t I2C_Master::_wrBufferLen [private]

Definition at line 65 of file I2C_Master.h.

5.4.6.19 uint8_t* I2C_Master::_WriteData [private]

Definition at line 64 of file I2C_Master.h.

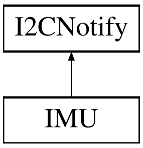
The documentation for this class was generated from the following file:

- I2C_Master.h

5.5 I2CNotify Class Reference

```
#include <I2C_Master.h>
```

Inheritance diagram for I2CNotify:



Public Member Functions

- virtual void I2CWriteDone ()=0
- virtual void I2CReadDone ()=0
- virtual void I2CBusError ()=0
- virtual void I2CArbLost ()=0
- virtual void I2CNack ()=0

5.5.1 Detailed Description

Definition at line 14 of file I2C_Master.h.

5.5.2 Member Function Documentation

5.5.2.1 virtual void I2CNotify::I2CArbLost ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.2 virtual void I2CNotify::I2CBusError ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.3 virtual void I2CNotify::I2CNack ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.4 virtual void I2CNotify::I2CReadDone ( ) `[pure virtual]`

Implemented in IMU.

5.5.2.5 virtual void I2CNotify::I2CWriteDone ( ) `[pure virtual]`

Implemented in IMU.

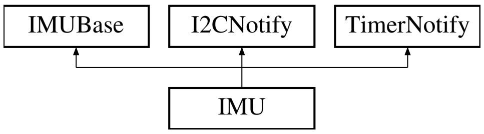
The documentation for this class was generated from the following file:

- I2C_Master.h

5.6 IMU Class Reference

```
#include <IMU.h>
```

Inheritance diagram for IMU:



Classes

- struct regWrite

Public Member Functions

- IMU (I2C_Master ∗pMas)
- IMU (I2C_Master ∗pMas, uint8_t gID, uint8_t aID)

  Constructor for a single IMU I2C Channel.

- IMU (I2C_Master ∗pMas, uint8_t gID, uint8_t aID, uint8_t gID2, uint8_t aID2)

  Constructor for a double IMU I2C Channel.

- void QueryChannels ()
- void SetDebugPort (DebugPort ∗pPort)
- void SetDebugPort2 (DebugPort ∗pPort)
- virtual void Reset ()
- virtual void SampleRate (uint16_t)
- virtual int Setup ()
- virtual int Start ()
- virtual int Stop ()
- virtual int ForceStartStop ()
- virtual bool Busy ()
- virtual void ResetTimer ()
- virtual void UseGyro (bool bEnable)
- virtual void NextIMU (IMUBase ∗pNext)
- virtual int BeginRead ()
- virtual bool DataReady ()
- virtual uint8_t ∗ GetPacketData (uint8_t ∗)
- virtual void CheckIDs (HardwareSerial ∗pSerial)
- virtual void ResetDevices ()
- void SetTimer (TimerCntr ∗pTimer)
- void SetTimerPeriod ()

- virtual void I2CWriteDone ()
- virtual void I2CReadDone ()
- virtual void I2CBusError ()
- virtual void I2CArbLost ()
- virtual void I2CNack ()
- void FailRecovery ()

Interrupt Handlers

These handlers receive interupts from the Timer class. We registered to received these calls with the Notify function. In some cases we might have 2 or more objects that will send us interrupt notifications, in this case we give each object an ID that is passed back so that we know which one caused the interrupt.

For the IMUManager, there is only a single Timer, so the ID is always 0.

- virtual void err (uint8_t id)

  Timer Error - ignored.

- virtual void ovf (uint8_t id)
- virtual void ccx (uint8_t id, uint8_t idx)

  Timer Capture Compare - not used.

Protected Member Functions

- void Run ()
- int StartTransaction ()
- void ProcessTransaction ()
- int Configure (uint8_t idx)
- int Wr (uint8_t ID, uint8_t addr, uint8_t data)
- int Rd (uint8_t ID, uint8_t addr, uint8_t cnt, uint8_t ∗pData)
- int WrAsync (uint8_t ID, uint8_t addr, uint8_t data)
- int RdAsync (uint8_t ID, uint8_t addr, uint8_t cnt)
- void ReadWord (uint16_t ∗pData)
- void StoreGyroData (uint8_t idx)
- void StoreAccData (uint8_t idx)
- void PushData (uint8_t idx)

  Push the data in the temporary buffer onto the appropriate fifo.

- void SetState (StateType s)
- void MarkPos (PosType p)
- void ResetBusyTime ()
- bool BusyTimeout ()
- void ResetFailStats ()

Private Types

- enum StateType {
  sIdle = 0, sConfigure = 1, sConfigured = 2, sWait = 5,
  sReadGyro1 = 8, sReadAcc1 = 9, sReadGyro2 = 10, sReadAcc2 = 11,
  sErrRecover = 12 }
- enum PosType {
  pStart = 0, pRun = 1, pWriteDn = 2, pReadDn = 3,
  pNack = 4, pBusErr = 5, pArbLost = 6, pSetup = 7 }
- enum FailType { fNone = 0, fNack = 1, fBusErr = 2, fArbLost = 3 }
- enum ProcessType { ptTimer, ptI2CWrite, ptI2CRead, ptI2CNack }
- typedef enum IMU::StateType StateType
- typedef enum IMU::PosType PosType
- typedef enum IMU::FailType FailType
- typedef struct IMU::regWrite RegWriteType

Private Attributes

- StateType _State
- StateType _previousState
- FailType _failType
- I2C_Master ∗ _pMas
- bool _bDualChan
- uint8_t _numChans
- bool _configOkay [2]
- uint8_t _gID [2]
- uint8_t _aID [2]
- uint8_t _DLPF
- uint8_t _FullScale
- uint8_t _ClkSel
- uint16_t _Rate
- bool _bUseGyro
- uint8_t _dataBuffer [2][20]

  This buffer is used to store data until we push it into the fifo.

- bool _bDataReady [2]
- TimerCntr ∗ _pTimer
- bool _bRun
- uint16_t _failCount
- uint8_t _nackCount

  used for recover logic

- bool _bFailDetected
- unsigned int _busyWaitTime
- DebugPort ∗ _pDBGPort
- DebugPort ∗ _pDBGPort2
- IMUBase ∗ _pNextIMU

5.6.1 Detailed Description

Definition at line 40 of file IMU.h.

5.6.2 Member Typedef Documentation

5.6.2.1 typedef enum IMU::FailType IMU::FailType `[private]`

5.6.2.2 typedef enum IMU::PosType IMU::PosType `[private]`

5.6.2.3 typedef struct IMU::regWrite IMU::RegWriteType `[private]`

5.6.2.4 typedef enum IMU::StateType IMU::StateType `[private]`

5.6.3 Member Enumeration Documentation

5.6.3.1 enum IMU::FailType `[private]`

Enumerator:

fNone

fNack

fBusErr

fArbLost

Definition at line 65 of file IMU.h.

```
                             {
        fNone              = 0,
        fNack              = 1,
        fBusErr            = 2,
        fArbLost           = 3
    } FailType;
```

5.6.3.2 enum IMU::PosType `[private]`

Enumerator:

- ***pStart***
- ***pRun***
- ***pWriteDn***
- ***pReadDn***
- ***pNack***
- ***pBusErr***
- ***pArbLost***
- ***pSetup***

Definition at line 54 of file IMU.h.

```
                         {
        pStart          = 0,
        pRun            = 1,
        pWriteDn        = 2,
        pReadDn         = 3,
        pNack           = 4,
        pBusErr         = 5,
        pArbLost        = 6,
        pSetup          = 7
    } PosType;
```

5.6.3.3 enum IMU::ProcessType `[private]`

Enumerator:

- ***ptTimer***
- ***ptI2CWrite***
- ***ptI2CRead***
- ***ptI2CNack***

Definition at line 78 of file IMU.h.

```
                  {
        ptTimer,
        ptI2CWrite,
        ptI2CRead,
        ptI2CNack
    } ProcessType;
```

5.6.3.4 enum IMU::StateType [private]

Enumerator:

sIdle

sConfigure

sConfigured

sWait

sReadGyro1

sReadAcc1

sReadGyro2

sReadAcc2

sErrRecover

Definition at line 42 of file IMU.h.

```
                             {
        sIdle               = 0,
        sConfigure          = 1,
        sConfigured         = 2,
        sWait               = 5,
        sReadGyro1          = 8,
        sReadAcc1           = 9,
        sReadGyro2          = 10,
        sReadAcc2           = 11,
        sErrRecover         = 12
    } StateType;
```

5.6.4 Constructor & Destructor Documentation

5.6.4.1 IMU::IMU (I2C_Master ∗ *pMas*)

Constructor for an auto-query channel Init fifos for dual channel

Definition at line 20 of file IMU.cpp.

References _aID, _bDataReady, _bDualChan, _bRun, _bUseGyro, _busyWaitTime, _ClkSel, _DLPF, _failType, _FullScale, _gID, _numChans, _pDBGPort, _pDBGPort2, _pMas, _pNextIMU, _previousState, _pTimer, _Rate, _State, fNone, I2C_Master::NotifyMe(), QueryChannels(), ResetFailStats(), and sIdle.

```
{
    _pNextIMU         = 0;
    _gID[0]           = 0;
    _aID[0]           = 0;
    _gID[1]           = 0;
    _aID[1]           = 0;
    _bDualChan        = false;
    _numChans         = 0;
    _pMas             = pMas;
    _DLPF            = 0x1;
    _FullScale       = 0x1;
    _ClkSel          = 0x1;
    _Rate            = 10;
    _State           = sIdle;
    _previousState   = sIdle;
    _failType        = fNone;
    _bRun            = false;
    _busyWaitTime    = 0;
    _bDataReady[0]   = false;
    _bDataReady[1]   = false;

    ResetFailStats();
    _pDBGPort        = 0;
    _pDBGPort2       = 0;
    _pNextIMU        = 0;
    _pTimer          = 0;
    _bUseGyro        = false;
    _pMas->NotifyMe(this);
    QueryChannels();
}
```

5.6.4.2 IMU::IMU (I2C_Master * *pMas,* uint8_t *gID,* uint8_t *aID*)

Constructor for a single IMU I2C Channel.

Definition at line 53 of file IMU.cpp.

References _aID, _bDualChan, _bRun, _bUseGyro, _ClkSel, _DLPF, _FullScale, _gID, _numChans, _pDBGPort, _pDBGPort2, _pMas, _pNextIMU, _pTimer, _Rate, _State, I2C_Master::NotifyMe(), ResetFailStats(), and sIdle.

```
{
    _pNextIMU    = 0;
    _gID[0]      = gID;
    _aID[0]      = aID;
    _bDualChan   = false;
    _numChans    = 1;
    _pMas        = pMas;
    _DLPF        = 0x1;
    _FullScale   = 0x1;
    _ClkSel      = 0x1;
    _Rate        = 10;
    _State       = sIdle;
    _bRun        = false;
    _pMas->NotifyMe(this);
    _pDBGPort    = 0;
    _pDBGPort2   = 0;
    _pTimer      = 0;
    ResetFailStats();
    _bUseGyro    = false;
}
```

5.6.4.3 IMU::IMU (I2C_Master ∗ *pMas,* uint8_t *gID,* uint8_t *aID,* uint8_t *gID2,* uint8_t *aID2*)

Constructor for a double IMU I2C Channel.

Definition at line 76 of file IMU.cpp.

References _aID, _bDualChan, _bRun, _bUseGyro, _ClkSel, _DLPF, _FullScale, _gID, _numChans, _pDBGPort, _pDBGPort2, _pMas, _pNextIMU, _pTimer, _Rate, _State, I2C_Master::NotifyMe(), ResetFailStats(), and sIdle.

```
{
    _pNextIMU    = 0;
    _gID[0]      = gID;
    _aID[0]      = aID;
    _gID[1]      = gID2;
    _aID[1]      = aID2;
    _bDualChan   = true;
    _numChans    = 2;
    _pMas        = pMas;
    _DLPF        = 0x1;
    _FullScale   = 0x1;
    _ClkSel      = 0x1;
    _Rate        = 10;
    _State       = sIdle;
    _bRun        = false;
    _pMas->NotifyMe(this);
    _pDBGPort    = 0;
    _pDBGPort2   = 0;
    _pTimer      = 0;
    ResetFailStats();
    _bUseGyro    = false;
}
```

5.6.5 Member Function Documentation

5.6.5.1 int IMU::BeginRead ( ) `[virtual]`

Implements IMUBase.

Definition at line 108 of file IMU.cpp.

References _pNextIMU, _State, IMUBase::BeginRead(), Run(), and sWait.

```
{
    if (_State == sWait) {
        Run();
    } else {
        if (_pNextIMU) {
            return _pNextIMU->BeginRead();
        }
    }
    return 0;
}
```

5.6.5.2 bool IMU::Busy ( ) [virtual]

Implements IMUBase.

Definition at line 277 of file IMU.cpp.

References _State, and sIdle.

```
{
    return _State != sIdle;
}
```

5.6.5.3 bool IMU::BusyTimeout ( ) [inline, protected]

Definition at line 195 of file IMU.h.

References _busyWaitTime.

Referenced by Run().

```
    {
        return ((millis() - _busyWaitTime) > 2);
    }
```

5.6.5.4 void IMU::ccx (uint8_t *id*, uint8_t *idx*) [virtual]

Timer Capture Compare - not used.

Implements TimerNotify.

Definition at line 752 of file IMU.cpp.

```
{
}
```

5.6.5.5 void IMU::CheckIDs (HardwareSerial ∗ *pSerial*) [virtual]

Implements IMUBase.

Definition at line 651 of file IMU.cpp.

References _aID, _gID, _numChans, _pMas, buffer, I2C_Master::CheckID(), Print::print(), and Wr().

```
{
    char buffer[50];
    for (int x=0;x<_numChans;x++) {
        int retc = _pMas->CheckID(_gID[x]);
        if (retc == 0) {
            sprintf(buffer,"Gyro%d (0x%x):Ack.\n",x,_gID[x]);
            pSerial->print(buffer);
        } else {
            sprintf(buffer,"Gyro%d (0x%x):NAck (%d).\n",x,_gID[x],retc);
            pSerial->print(buffer);
        }
        Wr(_gID[x], 0x3D, 0x8);
        retc = _pMas->CheckID(_aID[x]);
        if (retc == 0) {
            sprintf(buffer,"Acc%d (0x%x):Ack.\n",x,_aID[x]);
            pSerial->print(buffer);
        } else {
            sprintf(buffer,"Acc%d (0x%x):NAck (%d).\n",x,_aID[x],retc);
            pSerial->print(buffer);
        }
    }
}
```

5.6.5.6 int IMU::Configure (uint8_t *idx*) [protected]

Configure the Gyro and Accelerometer device. The input parameter selects the first or second channel. Build an array of RegWrite types so that I can check the return code of each write to insure they all pass.

Definition at line 803 of file IMU.cpp.

References _aID, _ClkSel, _DLPF, _FullScale, _gID, _Rate, ResetBusyTime(), and Wr().

Referenced by Setup().

```
{
    // Value for the sensor register
    uint16_t gval = 1000/_Rate;
    gval = gval - 1;

    int retc;

    RegWriteType    config[] = {
    //    // Turn on pass-through
        { _gID[idx], 0x3D, 0x0F },
    //
    //    // Init the Accelerometer.
        { _aID[idx], 0x20, 0x37},
        { _aID[idx], 0x21, 0x0},
        { _aID[idx], 0x22, 0x0},
        { _aID[idx], 0x23, 0x80 | 0x40},
        { _aID[idx], 0x24, 0x00},
    //
    //    // Set offsets to zero
        { _gID[idx], 0x0C, 0x00},
        { _gID[idx], 0x0D, 0x00},
        { _gID[idx], 0x0E, 0x00},
        { _gID[idx], 0x0F, 0x00},
        { _gID[idx], 0x10, 0x00},
        { _gID[idx], 0x11, 0x00},
    //
    //    // Configure registers.
        { _gID[idx], 0x12, 0xff},                    // Enable all outputs to to
       the fifo
        { _gID[idx], 0x13, 0x00},
    //    { _gID[idx], 0x14, _aID[idx] >> 1},            // Set slave address of
       ACC
        { _gID[idx], 0x15, gval},                    // Set sample rate
        { _gID[idx], 0x16, _DLPF | _FullScale << 3},
        { _gID[idx], 0x17, 0x00},
    //    { _gID[idx], 0x18, 0x80 | 0x28},              // Set burst address for
      Accelerometer, enable auto addr increment.
        { _gID[idx], 0x3E, _ClkSel},
    };

    uint8_t nItems = sizeof(config)/sizeof(RegWriteType);
    for (int idx = 0;idx <nItems;idx++) {
        retc = Wr(config[idx].ID,
            config[idx].Addr,
            config[idx].Data);
        ResetBusyTime();

        if (retc < 0) {
            return retc; // _configOkay will be false;
        }
    }

    return 0;
}
```

5.6.5.7 bool IMU::DataReady ( ) [virtual]

Implements IMUBase.

Definition at line 564 of file IMU.cpp.

References _bDataReady, _bDualChan, and _numChans.

```
{
    if (_numChans == 0) return true;

    bool bReady = false;
    cli();
    if (_bDualChan) {
        bReady =  _bDataReady[0] && _bDataReady[1];
    } else {
        bReady =  _bDataReady[0];
    }
    sei();
    return bReady;
}
```

5.6.5.8 void IMU::err (uint8_t *id*) [virtual]

Timer Error - ignored.

Implements TimerNotify.

Definition at line 738 of file IMU.cpp.

```
{
}
```

5.6.5.9 void IMU::FailRecovery ( )

Called after we handle a Nack, Bus Error or ArbLost Attempt to fix something, then return to the previous state and give it another go. We will basically keep doing this forever until the Manager says stop. All error or fail recovery goes through here.. I have finally gotten things cleaned up enough so that I have a central location for error attempts.

Definition at line 398 of file IMU.cpp.

References _failCount, _failType, _nackCount, _pMas, _previousState, fArbLost, fBusErr, fNack, fNone, Reset(), SetState(), StartTransaction(), and I2C_Master::WigglePin().

Referenced by I2CArbLost(), I2CBusError(), and I2CNack().

```
{
    switch(_failType) {
    case fNone:
        break;
    case fNack:
        if (_nackCount <7) {
            _pMas->WigglePin(10, 0,1);
        } else if (_nackCount < 10) {
            Reset();
            _pMas->WigglePin(10,0,1);
        }
        SetState(_previousState);
        break;
    case fBusErr:
        if (_failCount < 5) {
            Reset();
        }
        SetState(_previousState);
        break;
    case fArbLost:
        if (_failCount < 5) {
            Reset();
        }
        SetState(_previousState);
        break;
    }

    StartTransaction();
}
```

5.6.5.10 int IMU::ForceStartStop ( ) [virtual]

Implements IMUBase.

Definition at line 165 of file IMU.cpp.

References _pMas, and I2C_Master::ForceStartStop().

```
{
    return _pMas->ForceStartStop();
}
```

5.6.5.11 uint8_t ∗ IMU::GetPacketData (uint8_t ∗ *pData*) [virtual]

Retrieve packet data from the stored packets. If no packet data exists, this function will fill the data pointer with null data - this way any host software can continue, even if data is bad.

Implements IMUBase.

Definition at line 603 of file IMU.cpp.

References _bDataReady, _bDualChan, _dataBuffer, _failType, _numChans, _State, fNack, and sIdle.

```
{
    if (_numChans == 0) return pData;

    cli();

    *pData++ = 0xa5;
    *pData++ = 0x5a;
    if (_State == sIdle) {
        if (_failType == fNack) {
            memset(pData,'N',IMUPacket::PacketLen);
        } else {
            memset(pData,'I',IMUPacket::PacketLen);
        }
        pData += IMUPacket::PacketLen;
    } else if (_bDataReady[0] == true) {
        memcpy(pData,&_dataBuffer[0][0],IMUPacket::PacketLen);
        _bDataReady[0] = false;
        pData += IMUPacket::PacketLen;
    } else {
        memset(pData,0,IMUPacket::PacketLen);
        pData += IMUPacket::PacketLen;
    }

    if (_bDualChan) {
        *pData++ = 0xa5;
        *pData++ = 0x5a;
        if (_State == sIdle) {
            if (_failType == fNack) {
                memset(pData,'N',IMUPacket::PacketLen);
            } else {
                memset(pData,'I',IMUPacket::PacketLen);
            }
            pData += IMUPacket::PacketLen;
        } else if (_bDataReady[1] == true) {
            memcpy(pData,&_dataBuffer[1][0],IMUPacket::PacketLen);
            _bDataReady[1] = false;
            pData += IMUPacket::PacketLen;
        } else {
            memset(pData,0,IMUPacket::PacketLen);
            pData += IMUPacket::PacketLen;
        }
    }
    sei();
    return pData;
}
```

5.6.5.12 void IMU::I2CArbLost ( ) [virtual]

This occurs if the Arbitration is lost. If the I2C is in master mode, and it detects that it cannot control the state of the data line, i.e. it wants to set a HIGH but the line stays low, then this error occurs.

For a first pass, I am going to just try and return to Wait state - this way the timer will take back over and re-try some operation.

I temporarily set this to ErrRecover and then delay. If I am re-starting, then I will go right back, but I will at least trigger the Logic Analyzer

Implements I2CNotify.

Definition at line 534 of file IMU.cpp.

References _bFailDetected, _bRun, _failCount, _failType, _pDBGPort2, _previousState, FailRecovery(), fArbLost, pArbLost, pBusErr, ResetBusyTime(), sErrRecover, SetState(), and StartTransaction().

```
{
    if (_bRun == false) return;

    ResetBusyTime();
    {
        Mark marker(_pDBGPort2,pBusErr);
        _delay_us(3);
    }
    Mark marker(_pDBGPort2,pArbLost);
    _bFailDetected = true;
    ++_failCount;
    _failType = fArbLost;

    SetState(sErrRecover);
    _delay_us(5);
    if (_failCount > 10) {
        FailRecovery();
    } else {
        SetState(_previousState);
        StartTransaction();
    }
}
```

5.6.5.13 void IMU::I2CBusError ( ) [virtual]

Called by I2C Master when a Bus error occurs. This means some non-I2C compliant event occured. Normally, this is going to mean that some glitch occured on the I2C Bus.

For a first pass, I am going to just try and return to Wait state - this way the timer will take back over and re-try some operation.

I temporarily set this to ErrRecover and then delay. If I am re-starting, then I will go right back, but I will at least trigger the Logic Analyzer

Implements I2CNotify.

Definition at line 499 of file IMU.cpp.

References _bFailDetected, _bRun, _failCount, _failType, _pDBGPort2, _previousState, FailRecovery(), fBusErr, pBusErr, ResetBusyTime(), sErrRecover, SetState(), and StartTransaction().

```
{
    if (_bRun == false) return;

    ResetBusyTime();
    {
        Mark marker(_pDBGPort2,pBusErr);
        _delay_us(3);
    }
    Mark marker(_pDBGPort2,pBusErr);

    _bFailDetected = true;
    ++_failCount;
    _failType = fBusErr;

    SetState(sErrRecover);
    _delay_us(5);
    if (_failCount > 10) {
        FailRecovery();
    } else {
        SetState(_previousState);
        StartTransaction();
    }
}
```

5.6.5.14 void IMU::I2CNack ( ) [virtual]

Called by the I2C Master if we get a NAck. Current idea is the repeat the current command until it works since I am seeing that sometimes "Nacks" are temporary So a retry is best. Other types of failures may indicate a need for more desperate action - but those will probably be BusError or Arb Lost commands.

I temporarily set this to ErrRecover and then delay. If I am re-starting, then I will go right back, but I will at least trigger the Logic Analyzer

Re-start the same transaction.

Retry the current transaction until we are sure it won't work.

Keep failing.. just stop.

Implements I2CNotify.

Definition at line 463 of file IMU.cpp.

References _bRun, _failType, _nackCount, _pDBGPort2, _previousState, FailRecovery(), fNack, pBusErr, pNack, ResetBusyTime(), sErrRecover, SetState(), StartTransaction(), and Stop().

```
{
    if (_bRun == false) return;

    ResetBusyTime();
    {
        Mark marker(_pDBGPort2,pBusErr);
        _delay_us(3);
    }
    Mark marker(_pDBGPort2,pNack);

    ++_nackCount;

    SetState(sErrRecover);
    _delay_us(5);

    if (_nackCount < 5) {
        SetState(_previousState);
        StartTransaction();
    } else if (_nackCount < 10) {
        _failType = fNack;
        FailRecovery();
    } else {
        Stop();
    }
}
```

5.6.5.15 void IMU::I2CReadDone ( ) [virtual]

Called be the master when the read is complete. Requires registration Expected Context: Med Lvl I2C Int.

Implements I2CNotify.

Definition at line 446 of file IMU.cpp.

References _bRun, _pDBGPort2, pReadDn, ProcessTransaction(), ResetBusyTime(), and ResetFailStats().

```
{
    if (_bRun == false) return;

    Mark marker(_pDBGPort2,pReadDn);
    ResetBusyTime();

    ResetFailStats();
    ProcessTransaction();
}
```

5.6.5.16 void IMU::I2CWriteDone ( ) `[virtual]`

Called by the Master when the write is complete. Requires registration Expected Context: Med Lvl I2C Int.

Implements I2CNotify.

Definition at line 432 of file IMU.cpp.

References _bRun, _pDBGPort2, ProcessTransaction(), pWriteDn, ResetBusyTime(), and ResetFailStats().

```
{
    if (_bRun == false) return;

    Mark marker(_pDBGPort2,pWriteDn);
    ResetBusyTime();

    ResetFailStats();
    ProcessTransaction();
}
```

5.6.5.17 void IMU::MarkPos (PosType *p*) `[inline, protected]`

Definition at line 185 of file IMU.h.

References _pDBGPort2.

```
    {
        if (_pDBGPort2) _pDBGPort2->SetState((uint8_t) p);
    }
```

5.6.5.18 void IMU::NextIMU (IMUBase ∗ *pNext*) `[virtual]`

Implements IMUBase.

Definition at line 103 of file IMU.cpp.

References _pNextIMU.

```
{
    _pNextIMU = pNext;
}
```

5.6.5.19 void IMU::ovf (uint8_t *id*) `[virtual]`

Timer Overflow Interrupt. Overflow fires when the timer reaches the top period value. This is setup to fire when we get a timer tick, with a default rate of 500us IMUManager has only one Timer so the ID is not needed.

Implements TimerNotify.

Definition at line 746 of file IMU.cpp.

References Run().

```
{
    Run();
}
```

5.6.5.20 void IMU::ProcessTransaction ( ) [protected]

Called when an Asynchronous I2C operation succeeds. Switch on the state and perform the next appropriate action.

Start the next transaction.

Definition at line 348 of file IMU.cpp.

References _bDualChan, _bUseGyro, _pNextIMU, _State, IMUBase::BeginRead(), PushData(), SetState(), sReadAcc1, sReadAcc2, sReadGyro1, sReadGyro2, StartTransaction(), StoreAccData(), StoreGyroData(), and sWait.

Referenced by I2CReadDone(), and I2CWriteDone().

```
{
    switch(_State) {
        case sReadGyro1:
            StoreGyroData(1);
            SetState(sReadAcc1);
            break;
        case sReadAcc1:
            StoreAccData(1);
            PushData(1);
            if (_bDualChan) {
                if (_bUseGyro) {
                    SetState(sReadGyro2);
                } else {
                    SetState(sReadAcc2);
                }
            } else {
                SetState(sWait);
                if (_pNextIMU) {
                    _pNextIMU->BeginRead();
                }
            }
            break;
        case sReadGyro2:
            StoreGyroData(2);
            SetState(sReadAcc2);
            break;
        case sReadAcc2:
            StoreAccData(2);
            PushData(2);
            SetState(sWait);
            if (_pNextIMU) {
                _pNextIMU->BeginRead();
            }
            break;
        default:
            break;
    }

    StartTransaction();
}
```

5.6.5.21 void IMU::PushData (uint8_t *idx*) [protected]

Push the data in the temporary buffer onto the appropriate fifo.

Definition at line 594 of file IMU.cpp.

References _bDataReady.

Referenced by ProcessTransaction().

```
{
    _bDataReady[idx-1] = true;
}
```

5.6.5.22 void IMU::QueryChannels ( )

Definition at line 120 of file IMU.cpp.

References _aID, _bDualChan, _gID, _numChans, _pMas, and I2C_Master::CheckID().

Referenced by IMU().

```
{
    _numChans = 0;
    _bDualChan = 0;
    // Check the first, lower ID.
    int retc = _pMas->CheckID(0xD2);
    if (retc == 0) {
        _gID[_numChans] = 0xD2;
        _aID[_numChans] = 0x32; // Always high bit.
        _numChans++;
    }
    retc = _pMas->CheckID(0xD0);
    if (retc == 0) {
        _gID[_numChans] = 0xD0;
        _aID[_numChans] = 0x30; // Always low bit.
        _numChans++;
    }

    if (_numChans > 1) {
        _bDualChan = true;
    }
}
```

5.6.5.23 int IMU::Rd (uint8_t *ID*, uint8_t *addr*, uint8_t *cnt*, uint8_t ∗ *pData*) [protected]

Definition at line 766 of file IMU.cpp.

References _pMas, I2C_Master::ReadData(), and I2C_Master::WriteReadSync().

```
{
    // Only a single write, the address, then read data.
    int retc = _pMas->WriteReadSync(ID, &addr, 1, cnt);
    if ( retc < 0 ) {
        return retc;
    }
    return _pMas->ReadData(pData,cnt);
}
```

5.6.5.24 int IMU::RdAsync (uint8_t *ID*, uint8_t *addr*, uint8_t *cnt*) [protected]

Definition at line 786 of file IMU.cpp.

References _pMas, and I2C_Master::WriteRead().

Referenced by StartTransaction().

```
{
    // Only a single write, the address, then read data.
    return _pMas->WriteRead(ID, &addr, 1, cnt);
}
```

5.6.5.25 void IMU::ReadWord (uint16_t ∗ *pData*) [protected]

Definition at line 792 of file IMU.cpp.

References _pMas, buffer, and I2C_Master::ReadData().

```
{
    static uint8_t buffer[2];
    _pMas->ReadData(&buffer[0],2);
    *pData = (buffer[0] << 8 | buffer[1]);
}
```

5.6.5.26 void IMU::Reset ( ) [virtual]

Implements IMUBase.

Definition at line 153 of file IMU.cpp.

References _pMas, I2C_Master::begin(), I2C_Master::end(), and I2C_Master::NotifyMe().

Referenced by FailRecovery(), and Run().

```
{
    _pMas->end();
    _pMas->begin(400e3);
    _pMas->NotifyMe(this);
}
```

5.6.5.27 void IMU::ResetBusyTime ( ) [inline, protected]

Definition at line 190 of file IMU.h.

References _busyWaitTime.

Referenced by Configure(), I2CArbLost(), I2CBusError(), I2CNack(), I2CReadDone(), I2CWriteDone(), and StartTransaction().

```
    {
        _busyWaitTime = millis();
    }
```

5.6.5.28 void IMU::ResetDevices ( ) [virtual]

Implements IMUBase.

Definition at line 160 of file IMU.cpp.

References _pMas, and I2C_Master::Stop().

```
{
    _pMas->Stop();
}
```

5.6.5.29 void IMU::ResetFailStats ( ) [inline, protected]

Definition at line 200 of file IMU.h.

References _bFailDetected, _failCount, _failType, _nackCount, and fNone.

Referenced by I2CReadDone(), I2CWriteDone(), IMU(), Setup(), and Start().

```
    {
        _bFailDetected      = false;
        _nackCount          = 0;
        _failCount          = 0;
        _failType           = fNone;
    }
```

5.6.5.30 void IMU::ResetTimer ( ) [virtual]

Reset the counter value. The master uses this to get all of the timers operating at the same time.

Implements IMUBase.

Definition at line 702 of file IMU.cpp.

References _pTimer, and TimerCntr::Counter().

Referenced by Start().

```
{
    if (_pTimer) _pTimer->Counter(0);
}
```

5.6.5.31 void IMU::Run ( ) [protected]

Called by the Timer function periodically. Many of the operations are chainged, meaning the completion of one I2C operation, indicated by an I2C Write or Read interrupt, starts the next operation in the chain. The chain checks the fifo lengths of both fifos, then reads the data if there is any. If there is no data yet, then the state machine enters the sWait state and the next timer will initiate the chain again. Expected Context: Low Lvl Timer

Do nothing while the I2C Master is busy, the master is in the process of some operation. If the master gets hung for some reason, then we will reset and set the state to the sWait state to just start over.

This block only does something in the wait state. Hopefully the rest of the logic, all driven by I2C timeouts, will keep things moving through the other states. Perhaps I should have some sort of state change timeout, as long as we are in the run mode.

Definition at line 290 of file IMU.cpp.

References _bRun, _bUseGyro, _pDBGPort2, _pMas, _State, I2C_Master::busy(), BusyTimeout(), pRun, Reset(), SetState(), sReadAcc1, sReadGyro1, StartTransaction(), and sWait.

Referenced by BeginRead(), and ovf().

```
{
    if (_bRun == false) return;

    Mark marker(_pDBGPort2,pRun);
    if (_pMas->busy() && BusyTimeout()) {
        Reset();
        SetState(sWait);
    }

    switch(_State) {
    case sWait:
        if (_bUseGyro) {
            SetState(sReadGyro1);
        } else {
            SetState(sReadAcc1);
        }
        StartTransaction();
        break;
    default:
        break;
    }
}
```

5.6.5.32 void IMU::SampleRate (uint16_t *rate*) [virtual]

Implements IMUBase.

Definition at line 170 of file IMU.cpp.

References _Rate, Print::print(), and SetTimerPeriod().

```
{
    // Range Limit the rate.
    if (rate < 10) {
        _Rate = 10;
    } else if (rate > 200) {
        _Rate = 200;
    } else {
        _Rate = rate;
    }

    uint16_t gval = 1000/rate;
    gval = gval - 1;

    if (pdbgserial) pdbgserial->print("Set Rate on IMU\n");

    SetTimerPeriod();
}
```

5.6.5.33 void IMU::SetDebugPort (DebugPort ∗ *pPort*)

Definition at line 143 of file IMU.cpp.

References _pDBGPort.

Referenced by main().

```
{
    _pDBGPort = pPort;
}
```

5.6.5.34 void IMU::SetDebugPort2 (DebugPort * *pPort*)

Definition at line 148 of file IMU.cpp.

References _pDBGPort2.

Referenced by main().

```
{
    _pDBGPort2 = pPort;
}
```

5.6.5.35 void IMU::SetState (StateType *s*) `[inline, protected]`

Definition at line 178 of file IMU.h.

References _pDBGPort, _previousState, and _State.

Referenced by FailRecovery(), I2CArbLost(), I2CBusError(), I2CNack(), ProcessTransaction(), Run(), Setup(), Start(), and Stop().

```
    {
        _previousState = _State;
        _State = s;
        if (_pDBGPort) _pDBGPort->SetState((uint8_t)_State);
    }
```

5.6.5.36 void IMU::SetTimer (TimerCntr * *pTimer*)

Set Timer object to use for the main timer tick. This timer needs to be fast enough to send off packets at the rate configured. So, if 200Hz is the rate, then this timer must run at 1 500us rate, etc. The default will be to run at 500us, then the timer can be slowed down if a slower rate is used, just to avoid as much overhead. The CPU Clock runs at 32Mhz, so the main timer clock is running at 32/64 or 500us period. Timer set to same interrupt level as the I2C so that those interrupts won't ever stomp on each other.

This will be 2us period

Definition at line 685 of file IMU.cpp.

References _pTimer, TimerCntr::CCEnable(), TimerCntr::ClkSel(), TimerCntr::EventSetup(), TimerCntr::IntLvlA(), TimerCntr::IntLvlB(), TimerCntr::Notify(), SetTimerPeriod(), and TimerCntr::WaveformGenMode().

```
{
    _pTimer = pTimer;

    _pTimer->ClkSel(TC_CLKSEL_DIV64_gc);
    SetTimerPeriod();
    _pTimer->CCEnable(0);
    _pTimer->WaveformGenMode(TC_WGMODE_NORMAL_gc);
    _pTimer->EventSetup(TC_EVACT_OFF_gc,TC_EVSEL_OFF_gc);
    _pTimer->IntLvlA(0,1);
    _pTimer->IntLvlB(0);
    _pTimer->Notify(this,0);
}
```

5.6.5.37 void IMU::SetTimerPeriod ( )

Definition at line 707 of file IMU.cpp.

References _pTimer, _Rate, and TimerCntr::Period().

Referenced by SampleRate(), and SetTimer().

```
{
    // Adjust the timer function to fire 5X faster
    // than the rate. At 200Hz, this will be 2Khz or
    // every 500us.
    // We set the timer to go off 5 times per IMU period.
    // This should range from 20ms for 10Hz, and 1 ms for 200
    // **** NoFifo
    // Set timer to fire at the rate.
    //unsigned long timerTicks = 100000/_Rate;
    unsigned long timerTicks = 500000/_Rate;
    if (timerTicks > 65000) {
        timerTicks = 65000;
    }
    if (_pTimer) _pTimer->Period(timerTicks);
}
```

5.6.5.38 int IMU::Setup ( ) [virtual]

Perform the configuration on the connected IMUs This should be done separately from the loop since this process takes time and we do not want the fifos to be too far out of step. Expected Context: Main

Implements IMUBase.

Definition at line 194 of file IMU.cpp.

References _bRun, _numChans, _pDBGPort2, _State, Configure(), Print::print(), pSetup, ResetFailStats(), sConfigure, sConfigured, SetState(), and sIdle.

Referenced by Start().

```
{
    if (_numChans == 0) return 0;
    _bRun = false;
    Mark marker(_pDBGPort2,pSetup);

    if (_State != sIdle) {
        if (pdbgserial)
            pdbgserial->print("IMU Already Running.\n");
        return 0; // Already running
    }

    ResetFailStats(); // inline in header
    SetState(sConfigure); // Inline in header

    // Start the process with Configure
    for (int x=0; x<_numChans;x++) {
        if (pdbgserial)
            pdbgserial->print("Configuring IMU\n");
        int retc = Configure(x);
        if (retc < 0 ) {
            // Try reset mechanisms - none of which have
            // been found that work properly yet.
            SetState(sIdle);
            if (pdbgserial)
                pdbgserial->print("IMU Configure Failed.");
            return retc;
        }
    }
    SetState(sConfigured);
    if (pdbgserial)
        pdbgserial->print("IMU Configured.\n");

    return 0;
}
```

5.6.5.39 int IMU::Start ( ) [virtual]

Start the reading process running. Run the Setup function which will do asynchronous writes to all of the registers. Setup the run variables and then set our state to reset the fifo lengths then get going. Generally the master will call all of the Setup functions first so the asynchronous Writes there won't be an issue. Expected Context: Main

Implements IMUBase.

Definition at line 239 of file IMU.cpp.

References _bDataReady, _bRun, _numChans, _pDBGPort2, _State, pWriteDn, ResetFailStats(), ResetTimer(), sConfigured, SetState(), Setup(), and sWait.

```
{
    if (_numChans == 0) return 0;

    Mark marker(_pDBGPort2,pWriteDn);

    int retc;
    if (_State != sConfigured) {
        retc = Setup();
        if (retc < 0) {
            return retc;
        }
    }

    cli();
    ResetTimer();
    SetState(sWait);
    ResetFailStats();
    _bDataReady[0] = false;
    _bDataReady[1] = false;
    _bRun = true;
    sei();
    return 0;
}
```

5.6.5.40 int IMU::StartTransaction ( ) [protected]

Definition at line 323 of file IMU.cpp.

References _aID, _gID, _State, RdAsync(), ResetBusyTime(), sReadAcc1, sReadAcc2, sReadGyro1, and sReadGyro2.

Referenced by FailRecovery(), I2CArbLost(), I2CBusError(), I2CNack(), ProcessTransaction(), and Run().

```
{
        int retc = 0;
        switch(_State) {
        case sReadGyro1:
            RdAsync(_gID[0], 0x1B, 8);
            break;
        case sReadAcc1:
            RdAsync(_aID[0], 0x80 | 0x28, 6);
            break;
        case sReadGyro2:
            RdAsync(_gID[1], 0x1B, 8);
            break;
        case sReadAcc2:
            RdAsync(_aID[1], 0x80 | 0x28, 6);
            break;
        default:
            break;
        }
        ResetBusyTime();
        return retc;
}
```

5.6.5.41 int IMU::Stop ( ) [virtual]

Force the state to Idle. We need to consider how this works with iterrupts, but this should only be called from main, below interrupt level, so I think it is okay.

Implements IMUBase.

Definition at line 268 of file IMU.cpp.

References _bRun, SetState(), and sIdle.

Referenced by I2CNack().

```
{
    cli();
    _bRun = false;
    SetState(sIdle);
    sei();
    return 0;
}
```

5.6.5.42 void IMU::StoreAccData (uint8_t *idx*) `[protected]`

Add the accelerometer data to the buffer I have the index there in case I need it later..

Definition at line 588 of file IMU.cpp.

References _dataBuffer, _pMas, and I2C_Master::ReadData().

Referenced by ProcessTransaction().

```
{
    _pMas->ReadData(&_dataBuffer[idx-1][8],6);
}
```

5.6.5.43 void IMU::StoreGyroData (uint8_t *idx*) `[protected]`

Store the gyro read data while I get the Accelerometer data I have the index there in case I need it later..

Definition at line 581 of file IMU.cpp.

References _dataBuffer, _pMas, and I2C_Master::ReadData().

Referenced by ProcessTransaction().

```
{
    _pMas->ReadData(&_dataBuffer[idx-1][0],8);
}
```

5.6.5.44 virtual void IMU::UseGyro (bool *bEnable*) `[inline, virtual]`

Implements IMUBase.

Definition at line 137 of file IMU.h.

References _bUseGyro.

```
{ _bUseGyro = bEnable; };
```

5.6.5.45 int IMU::Wr (uint8_t *ID*, uint8_t *addr*, uint8_t *data*) `[protected]`

Definition at line 758 of file IMU.cpp.

References _pMas, and I2C_Master::WriteSync().

Referenced by CheckIDs(), and Configure().

```
{
    static uint8_t  bytes[2];
    bytes[0] = addr;
    bytes[1] = data;
    return _pMas->WriteSync(ID, &bytes[0],2);
}
```

5.6.5.46 int IMU::WrAsync (uint8_t *ID*, uint8_t *addr*, uint8_t *data*) `[protected]`

Definition at line 776 of file IMU.cpp.

References _pMas, buffer, Print::print(), and I2C_Master::WriteRead().

```
{
    static uint8_t  bytes[2];
    bytes[0] = addr;
    bytes[1] = data;
    sprintf(buffer,"WrAsync to %d\n",ID);
    if (pdbgserial) pdbgserial->print(buffer);
    return _pMas->WriteRead(ID, &bytes[0],2,0);
}
```

5.6.6 Member Data Documentation

5.6.6.1 uint8_t IMU::_aID[2] `[private]`

Definition at line 94 of file IMU.h.

Referenced by CheckIDs(), Configure(), IMU(), QueryChannels(), and StartTransaction().

5.6.6.2 bool IMU::_bDataReady[2] `[private]`

Definition at line 104 of file IMU.h.

Referenced by DataReady(), GetPacketData(), IMU(), PushData(), and Start().

5.6.6.3 bool IMU::_bDualChan [private]

Definition at line 90 of file IMU.h.

Referenced by DataReady(), GetPacketData(), IMU(), ProcessTransaction(), and QueryChannels().

5.6.6.4 bool IMU::_bFailDetected [private]

Definition at line 110 of file IMU.h.

Referenced by I2CArbLost(), I2CBusError(), and ResetFailStats().

5.6.6.5 bool IMU::_bRun [private]

Definition at line 107 of file IMU.h.

Referenced by I2CArbLost(), I2CBusError(), I2CNack(), I2CReadDone(), I2CWriteDone(), IMU(), Run(), Setup(), Start(), and Stop().

5.6.6.6 bool IMU::_bUseGyro [private]

Definition at line 100 of file IMU.h.

Referenced by IMU(), ProcessTransaction(), Run(), and UseGyro().

5.6.6.7 unsigned int IMU::_busyWaitTime [private]

Definition at line 111 of file IMU.h.

Referenced by BusyTimeout(), IMU(), and ResetBusyTime().

5.6.6.8 uint8_t IMU::_ClkSel [private]

Definition at line 97 of file IMU.h.

Referenced by Configure(), and IMU().

5.6.6.9 bool IMU::_configOkay[2] `[private]`

Definition at line 92 of file IMU.h.

5.6.6.10 uint8_t IMU::_dataBuffer[2][20] `[private]`

This buffer is used to store data until we push it into the fifo.

Definition at line 103 of file IMU.h.

Referenced by GetPacketData(), StoreAccData(), and StoreGyroData().

5.6.6.11 uint8_t IMU::_DLPF `[private]`

Definition at line 95 of file IMU.h.

Referenced by Configure(), and IMU().

5.6.6.12 uint16_t IMU::_failCount `[private]`

Definition at line 108 of file IMU.h.

Referenced by FailRecovery(), I2CArbLost(), I2CBusError(), and ResetFailStats().

5.6.6.13 FailType IMU::_failType `[private]`

Definition at line 87 of file IMU.h.

Referenced by FailRecovery(), GetPacketData(), I2CArbLost(), I2CBusError(), I2CNack(), IMU(), and ResetFailStats().

5.6.6.14 uint8_t IMU::_FullScale `[private]`

Definition at line 96 of file IMU.h.

Referenced by Configure(), and IMU().

5.6.6.15 uint8_t IMU::_gID[2] `[private]`

Definition at line 93 of file IMU.h.

Referenced by CheckIDs(), Configure(), IMU(), QueryChannels(), and StartTransaction().

5.6.6.16 uint8_t IMU::_nackCount [private]

used for recover logic

Definition at line 109 of file IMU.h.

Referenced by FailRecovery(), I2CNack(), and ResetFailStats().

5.6.6.17 uint8_t IMU::_numChans [private]

Definition at line 91 of file IMU.h.

Referenced by CheckIDs(), DataReady(), GetPacketData(), IMU(), QueryChannels(), Setup(), and Start().

5.6.6.18 DebugPort* IMU::_pDBGPort [private]

Definition at line 112 of file IMU.h.

Referenced by IMU(), SetDebugPort(), and SetState().

5.6.6.19 DebugPort* IMU::_pDBGPort2 [private]

Definition at line 113 of file IMU.h.

Referenced by I2CArbLost(), I2CBusError(), I2CNack(), I2CReadDone(), I2CWriteDone(), IMU(), MarkPos(), Run(), SetDebugPort2(), Setup(), and Start().

5.6.6.20 I2C_Master* IMU::_pMas [private]

Definition at line 89 of file IMU.h.

Referenced by CheckIDs(), FailRecovery(), ForceStartStop(), IMU(), QueryChannels(), Rd(), RdAsync(), ReadWord(), Reset(), ResetDevices(), Run(), StoreAccData(), StoreGyroData(), Wr(), and WrAsync().

5.6.6.21 IMUBase* IMU::_pNextIMU [private]

Definition at line 114 of file IMU.h.

Referenced by BeginRead(), IMU(), NextIMU(), and ProcessTransaction().

5.6.6.22 StateType IMU::_previousState [private]

Definition at line 86 of file IMU.h.

Referenced by FailRecovery(), I2CArbLost(), I2CBusError(), I2CNack(), IMU(), and SetState().

5.6.6.23 TimerCntr* IMU::_pTimer [private]

Definition at line 106 of file IMU.h.

Referenced by IMU(), ResetTimer(), SetTimer(), and SetTimerPeriod().

5.6.6.24 uint16_t IMU::_Rate [private]

Definition at line 98 of file IMU.h.

Referenced by Configure(), IMU(), SampleRate(), and SetTimerPeriod().

5.6.6.25 StateType IMU::_State [private]

Definition at line 85 of file IMU.h.

Referenced by BeginRead(), Busy(), GetPacketData(), IMU(), ProcessTransaction(), Run(), SetState(), Setup(), Start(), and StartTransaction().

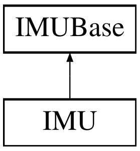
The documentation for this class was generated from the following files:

- IMU.h
- IMU.cpp

5.7 IMUBase Class Reference

```
#include <IMU.h>
```

Inheritance diagram for IMUBase:



Public Member Functions

- virtual void Reset ()=0
- virtual void SampleRate (uint16_t)=0
- virtual int Setup ()=0
- virtual int Start ()=0
- virtual int Stop ()=0
- virtual int ForceStartStop ()=0
- virtual bool Busy ()=0
- virtual void ResetTimer ()=0
- virtual void UseGyro (bool bEnable)=0
- virtual void NextIMU (IMUBase ∗)=0
- virtual int BeginRead ()=0
- virtual bool DataReady ()=0
- virtual uint8_t ∗ GetPacketData (uint8_t ∗pPacket)=0
- virtual void CheckIDs (HardwareSerial ∗pSerial)=0
- virtual void ResetDevices ()=0

5.7.1 Detailed Description

Definition at line 16 of file IMU.h.

5.7.2 Member Function Documentation

5.7.2.1 virtual int IMUBase::BeginRead ( ) `[pure virtual]`

Implemented in IMU.

Referenced by IMU::BeginRead(), and IMU::ProcessTransaction().

5.7.2.2 virtual bool IMUBase::Busy ( ) `[pure virtual]`

Implemented in IMU.

5.7.2.3 virtual void IMUBase::CheckIDs (HardwareSerial * *pSerial*) `[pure virtual]`

Implemented in IMU.

5.7.2.4 virtual bool IMUBase::DataReady ( ) `[pure virtual]`

Implemented in IMU.

5.7.2.5 virtual int IMUBase::ForceStartStop ( ) `[pure virtual]`

Implemented in IMU.

5.7.2.6 virtual uint8_t* IMUBase::GetPacketData (uint8_t * *pPacket*) `[pure virtual]`

Implemented in IMU.

5.7.2.7 virtual void IMUBase::NextIMU (IMUBase *) `[pure virtual]`

Implemented in IMU.

5.7.2.8 virtual void IMUBase::Reset ( ) `[pure virtual]`

Implemented in IMU.

5.7.2.9 virtual void IMUBase::ResetDevices ( ) `[pure virtual]`

Implemented in IMU.

5.7.2.10 virtual void IMUBase::ResetTimer ( ) `[pure virtual]`

Implemented in IMU.

5.7.2.11 virtual void IMUBase::SampleRate (uint16_t) [pure virtual]

Implemented in IMU.

5.7.2.12 virtual int IMUBase::Setup ( ) [pure virtual]

Implemented in IMU.

5.7.2.13 virtual int IMUBase::Start ( ) [pure virtual]

Implemented in IMU.

5.7.2.14 virtual int IMUBase::Stop ( ) [pure virtual]

Implemented in IMU.

5.7.2.15 virtual void IMUBase::UseGyro (bool *bEnable*) [pure virtual]

Implemented in IMU.

The documentation for this class was generated from the following file:

- IMU.h

5.8 Port Class Reference

```
#include <Port.h>
```

Public Member Functions

- Port (PORT_t ∗)
- ∼Port ()
- void Notify (PortNotify ∗pClient, uint8_t id)
- void int0 ()
- void int1 ()
- void SetDir (uint8_t dir)
- void SetPinsAsInput (uint8_t mask)
- void SetPinsAsOutput (uint8_t mask)

- void SetPinsHigh (uint8_t mask)
- void SetPinsLow (uint8_t mask)
- uint8_t GetPins ()
- void InterruptLevel (uint8_t num, uint8_t lvl)

  Interrupt control.

- void InterruptMask (uint8_t num, uint8_t mask)
- void PinControl (uint8_t mask, bool bSlewLimit, bool bInverted, PORT_OPC_t OutputConfig, PORT_ISC_t InputSense)

  Pin Control Register.

Protected Attributes

- PORT_t ∗ _pPort
- PortNotify ∗ _pNotifyClient
- uint8_t _pNotifyID

5.8.1 Detailed Description

Class to handle the setup and control of a port on the ATxmega. This class will setup the port using the PinControl method. It will also setup and receive interrupts on either int0 or int1. You may set an interrupt mask to determine which pins cause which interrupt. For example, it is possible to have pins 0 and 1 cause interrupts on ISR0, and pins 6 and 7 to interrupt on ISR1.

Definition at line 26 of file Port.h.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Port::Port (PORT_t ∗ *pPort*)

Definition at line 58 of file Port.cpp.

References _pPort, and SetPointer().

```
{
    _pPort = pPort;
    SetPointer(_pPort,this);
}
```

5.8.2.2 Port::∼Port ( )

Definition at line 64 of file Port.cpp.

References _pPort, and SetPointer().

```
{
    SetPointer(_pPort,0);
}
```

5.8.3 Member Function Documentation

5.8.3.1 uint8_t Port::GetPins ( )

Definition at line 116 of file Port.cpp.

References _pPort.

```
{
    return _pPort->IN;
}
```

5.8.3.2 void Port::int0 ( )

Definition at line 75 of file Port.cpp.

References _pNotifyClient, _pNotifyID, _pPort, and PortNotify::PortISR0().

```
{
    if (_pNotifyClient) {
        _pNotifyClient->PortISR0(_pNotifyID);
    }
    _pPort->INTFLAGS = 0x1;
}
```

5.8.3.3 void Port::int1 ( )

Definition at line 83 of file Port.cpp.

References _pNotifyClient, _pNotifyID, _pPort, and PortNotify::PortISR1().

```
{
    if (_pNotifyClient) {
        _pNotifyClient->PortISR1(_pNotifyID);
    }
    _pPort->INTFLAGS = 0x2;
}
```

5.8.3.4 void Port::InterruptLevel (uint8_t *num*, uint8_t *lvl*)

Interrupt control.

Definition at line 121 of file Port.cpp.

References _pPort.

```
{
    if (num == 0) {
        _pPort->INTCTRL &= ~(0x3);
        _pPort->INTCTRL |= (lvl & 0x3);
    } else {
        _pPort->INTCTRL &= ~(0xC);
        _pPort->INTCTRL |= (lvl & 0x3) << 2;
    }
}
```

5.8.3.5 void Port::InterruptMask (uint8_t *num*, uint8_t *mask*)

Definition at line 132 of file Port.cpp.

References _pPort.

```
{
    if (num == 0) {
        _pPort->INT0MASK = mask;
    } else {
        _pPort->INT1MASK = mask;
    }
}
```

5.8.3.6 void Port::Notify (PortNotify ∗ *pClient*, uint8_t *id*)

Definition at line 69 of file Port.cpp.

References _pNotifyClient, and _pNotifyID.

```
{
    _pNotifyClient   = pClient;
    _pNotifyID       = id;
}
```

5.8.3.7 void Port::PinControl (uint8_t *mask*, bool *bSlewLimit*, bool *bInverted*, PORT_OPC_t *OutputConfig*, PORT_ISC_t *InputSense*)

Pin Control Register.

The MPCMASK is a neat feature. I set each of the bits of the mask high, then configure any of the PINxCTRL registers, and only the pins specified in the mask get configured. Also, they all get the same config, so it's faster. It does not matter if I am actually configuring pin 0 or not, even though I specify PN0CTRL.

Definition at line 141 of file Port.cpp.

References _pPort.

```
{
    PORTCFG.MPCMASK = mask;
    _pPort->PIN0CTRL =
        (bSlewLimit ? 0x80 : 0x0) |
        (bInverted ? 0x40 : 0x0)  |
        OutputConfig |
        InputSense
        ;
}
```

5.8.3.8 void Port::SetDir (uint8_t *dir*)

Definition at line 91 of file Port.cpp.

References _pPort.

```
{
    _pPort->DIR = dir;
}
```

5.8.3.9 void Port::SetPinsAsInput (uint8_t *mask*)

Definition at line 96 of file Port.cpp.

References _pPort.

```
{
    _pPort->DIRCLR = mask;
}
```

5.8.3.10 void Port::SetPinsAsOutput (uint8_t *mask*)

Definition at line 101 of file Port.cpp.

References _pPort.

```
{
    _pPort->DIRSET = mask;
}
```

5.8.3.11 void Port::SetPinsHigh (uint8_t *mask*)

Definition at line 106 of file Port.cpp.

References _pPort.

```
{
    _pPort->OUTSET = mask;
}
```

5.8.3.12 void Port::SetPinsLow (uint8_t *mask*)

Definition at line 111 of file Port.cpp.

References _pPort.

```
{
    _pPort->OUTCLR = mask;
}
```

5.8.4 Member Data Documentation

5.8.4.1 PortNotify∗ Port::_pNotifyClient [protected]

Definition at line 30 of file Port.h.

Referenced by int0(), int1(), and Notify().

5.8.4.2 uint8_t Port::_pNotifyID [protected]

Definition at line 31 of file Port.h.

Referenced by int0(), int1(), and Notify().

5.8.4.3 PORT_t∗ Port::_pPort [protected]

Definition at line 29 of file Port.h.

Referenced by GetPins(), int0(), int1(), InterruptLevel(), InterruptMask(), PinControl(), Port(), SetDir(), SetPinsAsInput(), SetPinsAsOutput(), SetPinsHigh(), SetPinsLow(), and ∼Port().

The documentation for this class was generated from the following files:

- Port.h
- Port.cpp

5.9 PortNotify Class Reference

```
#include <Port.h>
```

Public Member Functions

- virtual void PortISR0 (uint8_t id)=0
- virtual void PortISR1 (uint8_t id)=0

5.9.1 Detailed Description

Base class for any classes that require port change notification If a class requires notification of a port change event, then use this class as a base class. Call the Notify method of any Port class and pass in the ’this’ pointer of the class to notify. Also pass in an index value. The index value is needed becuase it is possible for a single class to request notifications from 2 or more Ports. In order to know which of the ports is sending the notification, use a unique index for both. The actual value of the index is not important.

Definition at line 13 of file Port.h.

5.9.2 Member Function Documentation

5.9.2.1 virtual void PortNotify::PortISR0 (uint8_t *id*) `[pure virtual]`

Referenced by Port::int0().

5.9.2.2 virtual void PortNotify::PortISR1 (uint8_t *id*) `[pure virtual]`

Referenced by Port::int1().

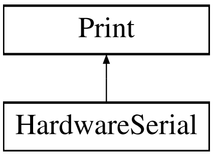
The documentation for this class was generated from the following file:

- Port.h

5.10 Print Class Reference

```
#include <Print.h>
```

Inheritance diagram for Print:



Public Member Functions

- virtual void write (uint8_t)=0
- virtual void write (const char ∗str)
- virtual void write (const uint8_t ∗buffer, size_t size)
- void print (const char[ ])
- void print (char, int=BYTE)
- void print (unsigned char, int=BYTE)
- void print (int, int=DEC)
- void print (unsigned int, int=DEC)
- void print (long, int=DEC)
- void print (unsigned long, int=DEC)
- void print (double, int=2)
- void println (const char[ ])
- void println (char, int=BYTE)
- void println (unsigned char, int=BYTE)
- void println (int, int=DEC)
- void println (unsigned int, int=DEC)
- void println (long, int=DEC)
- void println (unsigned long, int=DEC)
- void println (double, int=2)
- void println (void)

Private Member Functions

- void printNumber (unsigned long, uint8_t)
- void printFloat (double, uint8_t)

5.10.1 Detailed Description

Definition at line 32 of file Print.h.

5.10.2 Member Function Documentation

5.10.2.1 void Print::print (const char *str[ ]*)

Definition at line 45 of file Print.cpp.

References write().

Referenced by CmdProcessor::checkCommands(), IMU::CheckIDs(), FifoTest(), main(), print(), printFloat(), println(), printNumber(), IMU::SampleRate(), IMU::Setup(), and IMU::WrAsync().

```
{
  write(str);
}
```

5.10.2.2 void Print::print (unsigned char *b*, int *base* = BYTE)

Definition at line 55 of file Print.cpp.

References print().

```
{
  print((unsigned long) b, base);
}
```

5.10.2.3 void Print::print (int *n*, int *base* = DEC)

Definition at line 60 of file Print.cpp.

References print().

```
{
  print((long) n, base);
}
```

5.10.2.4 void Print::print (unsigned int *n*, int *base* = DEC)

Definition at line 65 of file Print.cpp.

References print().

```
{
  print((unsigned long) n, base);
}
```

5.10.2.5 void Print::print (long *n*, int *base* = DEC)

Definition at line 70 of file Print.cpp.

References print(), printNumber(), and write().

```
{
  if (base == 0) {
    write(n);
  } else if (base == 10) {
    if (n < 0) {
      print('-');
      n = -n;
    }
    printNumber(n, 10);
  } else {
    printNumber(n, base);
  }
}
```

5.10.2.6 void Print::print (char *c*, int *base* = BYTE)

Definition at line 50 of file Print.cpp.

References print().

```
{
  print((long) c, base);
}
```

5.10.2.7 void Print::print (unsigned long *n*, int *base* = DEC)

Definition at line 85 of file Print.cpp.

References printNumber(), and write().

```
{
  if (base == 0) write(n);
  else printNumber(n, base);
}
```

5.10.2.8 void Print::print (double *n*, int *digits* = 2)

Definition at line 91 of file Print.cpp.

References printFloat().

```
{
  printFloat(n, digits);
}
```

5.10.2.9 void Print::printFloat (double *number*, uint8_t *digits*) [private]

Definition at line 173 of file Print.cpp.

References print().

Referenced by print().

```
{
  // Handle negative numbers
  if (number < 0.0)
  {
     print('-');
     number = -number;
  }

  // Round correctly so that print(1.999, 2) prints as "2.00"
  double rounding = 0.5;
  for (uint8_t i=0; i<digits; ++i)
    rounding /= 10.0;

  number += rounding;

  // Extract the integer part of the number and print it
  unsigned long int_part = (unsigned long)number;
  double remainder = number - (double)int_part;
  print(int_part);

  // Print the decimal point, but only if there are digits beyond
  if (digits > 0)
    print(".");

  // Extract digits from the remainder one at a time
  while (digits-- > 0)
  {
    remainder *= 10.0;
    int toPrint = int(remainder);
    print(toPrint);
    remainder -= toPrint;
  }
}
```

5.10.2.10 void Print::println (long *n*, int *base* = DEC)

Definition at line 132 of file Print.cpp.

References print(), and println().

```
{
  print(n, base);
  println();
}
```

5.10.2.11 void Print::println (void)

Definition at line 96 of file Print.cpp.

References print().

Referenced by println().

```
{
  print('\r');
  print('\n');
}
```

5.10.2.12 void Print::println (unsigned char *b*, int *base* = **BYTE**)

Definition at line 114 of file Print.cpp.

References print(), and println().

```
{
  print(b, base);
  println();
}
```

5.10.2.13 void Print::println (double *n*, int *digits* = **2**)

Definition at line 144 of file Print.cpp.

References print(), and println().

```
{
  print(n, digits);
  println();
}
```

5.10.2.14 void Print::println (unsigned int *n*, int *base* = **DEC**)

Definition at line 126 of file Print.cpp.

References print(), and println().

```
{
  print(n, base);
  println();
}
```

5.10.2.15 void Print::println (unsigned long *n*, int *base* = `DEC`)

Definition at line 138 of file Print.cpp.

References print(), and println().

```
{
  print(n, base);
  println();
}
```

5.10.2.16 void Print::println (char *c*, int *base* = `BYTE`)

Definition at line 108 of file Print.cpp.

References print(), and println().

```
{
  print(c, base);
  println();
}
```

5.10.2.17 void Print::println (const char *c[ ]*)

Definition at line 102 of file Print.cpp.

References print(), and println().

```
{
  print(c);
  println();
}
```

5.10.2.18 void Print::println (int *n*, int *base* = `DEC`)

Definition at line 120 of file Print.cpp.

References print(), and println().

```
{
  print(n, base);
  println();
}
```

5.10.2.19 void Print::printNumber (unsigned long *n*, uint8_t *base*) [private]

Definition at line 152 of file Print.cpp.

References print().

Referenced by print().

```
{
  unsigned char buf[8 * sizeof(long)]; // Assumes 8-bit chars.
  unsigned long i = 0;

  if (n == 0) {
    print('0');
    return;
  }

  while (n > 0) {
    buf[i++] = n % base;
    n /= base;
  }

  for (; i > 0; i--)
    print((char) (buf[i - 1] < 10 ?
      '0' + buf[i - 1] :
      'A' + buf[i - 1] - 10));
}
```

5.10.2.20 virtual void Print::write (uint8_t) [pure virtual]

Implemented in HardwareSerial.

Referenced by print(), and write().

5.10.2.21 void Print::write (const char * *str*) [virtual]

Definition at line 32 of file Print.cpp.

References write().

```
{
  while (*str)
    write(*str++);
}
```

5.10.2.22 void Print::write (const uint8_t * *buffer*, size_t *size*) [virtual]

Definition at line 39 of file Print.cpp.

References write().

```
{
  while (size--)
    write(*buffer++);
}
```

The documentation for this class was generated from the following files:

- Print.h
- Print.cpp

5.11 IMU::regWrite Struct Reference

Public Attributes

- uint8_t ID
- uint8_t Addr
- uint8_t Data

5.11.1 Detailed Description

Definition at line 72 of file IMU.h.

5.11.2 Member Data Documentation

5.11.2.1 uint8_t IMU::regWrite::Addr

Definition at line 74 of file IMU.h.

5.11.2.2 uint8_t IMU::regWrite::Data

Definition at line 75 of file IMU.h.

5.11.2.3 uint8_t IMU::regWrite::ID

Definition at line 73 of file IMU.h.

The documentation for this struct was generated from the following file:

- IMU.h

5.12 ring_buffer Struct Reference

Public Attributes

- unsigned char buffer [RX_BUFFER_SIZE]
- int head
- int tail

5.12.1 Detailed Description

Definition at line 16 of file HardwareSerial.cpp.

5.12.2 Member Data Documentation

5.12.2.1 unsigned char ring_buffer::buffer[RX_BUFFER_SIZE]

Definition at line 17 of file HardwareSerial.cpp.

Referenced by HardwareSerial::read(), and store_char().

5.12.2.2 int ring_buffer::head

Definition at line 18 of file HardwareSerial.cpp.

Referenced by HardwareSerial::available(), HardwareSerial::flush(), HardwareSerial::read(), and store_char().

5.12.2.3 int ring_buffer::tail

Definition at line 19 of file HardwareSerial.cpp.

Referenced by HardwareSerial::available(), HardwareSerial::flush(), HardwareSerial::read(), and store_char().

The documentation for this struct was generated from the following file:

- HardwareSerial.cpp

5.13 TimerCntr Class Reference

```
#include <TimerCntr.h>
```

Public Member Functions

- TimerCntr (TC0_t ∗pTC)
- TimerCntr (TC1_t ∗pTC)
- ∼TimerCntr ()
- void Notify (TimerNotify ∗pClient, uint8_t id)
- void ovf ()
- void err ()
- void ccx (uint8_t idx)
- void ClkSel (TC_CLKSEL_t clksel)

    Set the clock source for this timer in CTRLA.

- void CCEnable (uint8_t mask)
- void WaveformGenMode (TC_WGMODE_t wgmode)
- void EventSetup (TC_EVACT_t act, TC_EVSEL_t src)
- void IntLvlA (uint8_t errlvl, uint8_t ovflvl)
- void IntLvlB (uint8_t val)
- void Counter (uint16_t newVal)
- uint16_t Counter ()
- void Period (uint16_t newPer)
- uint16_t Period ()
- void SetRate (uint32_t rateHz)
- void CCReg (uint8_t idx, uint16_t newVal)
- uint16_t CCReg (uint8_t idx)

Private Attributes

- TC0_t ∗ _pTC
- bool _bTC1
- TimerNotify ∗ _pNotifyClient
- uint8_t _pNotifyClientID

5.13.1 Detailed Description

Definition at line 13 of file TimerCntr.h.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 TimerCntr::TimerCntr (TC0_t ∗ *pTC*)

Definition at line 116 of file TimerCntr.cpp.

References _bTC1, _pNotifyClient, _pNotifyClientID, _pTC, and SetPointer().

```
{
    _pTC = pTC;
    _bTC1 = false;
    _pNotifyClient = 0;
    _pNotifyClientID = 0;
    SetPointer(pTC,this);
}
```

5.13.2.2 TimerCntr::TimerCntr (TC1_t ∗ *pTC*)

Definition at line 125 of file TimerCntr.cpp.

References _bTC1, _pNotifyClient, _pNotifyClientID, _pTC, and SetPointer().

```
{
    _pTC = (TC0_t*)pTC;
    _bTC1 = true;
    _pNotifyClient = 0;
    _pNotifyClientID = 0;
    SetPointer(pTC,this);
}
```

5.13.2.3 TimerCntr::∼TimerCntr ( )

Definition at line 134 of file TimerCntr.cpp.

References _bTC1, _pTC, and SetPointer().

```
{
    if (_bTC1) {
        SetPointer((TC1_t*)_pTC,0);
    } else {
        SetPointer(_pTC,0);
    }
}
```

5.13.3 Member Function Documentation

5.13.3.1 void TimerCntr::CCEnable (uint8_t *mask*)

Set the state of the 4 CCxEN bits in CTRLB. The 4 lower bits of mask will enable D, C, B or A if set to 1

Definition at line 150 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimer().

```
{
    _pTC->CTRLB = ((_pTC->CTRLB & 0x0F) | (mask << 4));
}
```

5.13.3.2 void TimerCntr::CCReg (uint8_t *idx*, uint16_t *newVal*)

Definition at line 200 of file TimerCntr.cpp.

References _bTC1, and _pTC.

```
{
    if (idx == 0) {
        _pTC->CCA = newVal;
    } else if (idx == 1) {
        _pTC->CCB = newVal;
    } else if (!_bTC1 && idx == 2) {
        _pTC->CCC = newVal;
    } else if (!_bTC1 && idx == 3) {
        _pTC->CCD = newVal;
    }
}
```

5.13.3.3 uint16_t TimerCntr::CCReg (uint8_t *idx*)

Definition at line 213 of file TimerCntr.cpp.

References _bTC1, and _pTC.

```
{
    if (idx == 0) {
        return _pTC->CCA;
    } else if (idx == 1) {
        return _pTC->CCB;
    } else if (!_bTC1 && idx == 2) {
        return _pTC->CCC;
    } else if (!_bTC1 && idx == 3) {
        return _pTC->CCD;
    }
    return 0;
}
```

5.13.3.4 void TimerCntr::ccx (uint8_t *idx*)

Definition at line 245 of file TimerCntr.cpp.

References _pNotifyClient, _pNotifyClientID, and TimerNotify::ccx().

```
{
    if (_pNotifyClient)
        _pNotifyClient->ccx(_pNotifyClientID,idx);
}
```

5.13.3.5 void TimerCntr::ClkSel (TC_CLKSEL_t *clksel*)

Set the clock source for this timer in CTRLA.

Definition at line 144 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimer().

```
{
    _pTC->CTRLA = clksel;
}
```

5.13.3.6 void TimerCntr::Counter (uint16_t *newVal*)

Definition at line 178 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::ResetTimer().

```
{
    _pTC->CNT = newVal;
}
```

5.13.3.7 uint16_t TimerCntr::Counter ( )

Definition at line 183 of file TimerCntr.cpp.

References _pTC.

```
{
    return _pTC->CNT;
}
```

5.13.3.8 void TimerCntr::err ( )

Definition at line 233 of file TimerCntr.cpp.

References _pNotifyClient, _pNotifyClientID, and TimerNotify::err().

```
{
    if (_pNotifyClient)
        _pNotifyClient->err(_pNotifyClientID);
}
```

5.13.3.9 void TimerCntr::EventSetup (TC_EVACT_t *act*, TC_EVSEL_t *src*)

Definition at line 162 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimer().

```
{
    _pTC->CTRLD = act | src;
}
```

5.13.3.10 void TimerCntr::IntLvlA (uint8_t *errlvl*, uint8_t *ovflvl*)

Definition at line 168 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimer().

```
{
    _pTC->INTCTRLA = (errlvl & 0x3) << 2 | (ovflvl & 0x3);
}
```

5.13.3.11 void TimerCntr::IntLvlB (uint8_t *val*)

Definition at line 173 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimer().

```
{
    _pTC->INTCTRLB = val;
}
```

5.13.3.12 void TimerCntr::Notify (TimerNotify ∗ *pClient,* uint8_t *id*)

Definition at line 227 of file TimerCntr.cpp.

References _pNotifyClient, and _pNotifyClientID.

Referenced by IMU::SetTimer().

```
{
    _pNotifyClient = pClient;
    _pNotifyClientID = id;
}
```

5.13.3.13 void TimerCntr::ovf ( )

Definition at line 239 of file TimerCntr.cpp.

References _pNotifyClient, _pNotifyClientID, and TimerNotify::ovf().

```
{
    if (_pNotifyClient)
        _pNotifyClient->ovf(_pNotifyClientID);
}
```

5.13.3.14 void TimerCntr::Period (uint16_t *newPer*)

Definition at line 189 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimerPeriod().

```
{
    _pTC->PER = newVal;
}
```

5.13.3.15 uint16_t TimerCntr::Period ( )

Definition at line 194 of file TimerCntr.cpp.

References _pTC.

```
{
    return _pTC->PER;
}
```

5.13.3.16 void TimerCntr::SetRate (uint32_t *rateHz*)

Definition at line 251 of file TimerCntr.cpp.

```
                                       {
    //add an auto prescaler using 32 bit array
}
```

5.13.3.17 void TimerCntr::WaveformGenMode (TC_WGMODE_t *wgmode*)

Set the Waveform generator mode. Normal disables waveform generation.

Definition at line 156 of file TimerCntr.cpp.

References _pTC.

Referenced by IMU::SetTimer().

```
{
    _pTC->CTRLB = ((_pTC->CTRLB & 0xF0) | wgmode);
}
```

5.13.4 Member Data Documentation

5.13.4.1 bool TimerCntr::_bTC1 `[private]`

Definition at line 19 of file TimerCntr.h.

Referenced by CCReg(), TimerCntr(), and ∼TimerCntr().

5.13.4.2 TimerNotify∗ TimerCntr::_pNotifyClient `[private]`

Definition at line 20 of file TimerCntr.h.

Referenced by ccx(), err(), Notify(), ovf(), and TimerCntr().

5.13.4.3 uint8_t TimerCntr::_pNotifyClientID [private]

Definition at line 21 of file TimerCntr.h.

Referenced by ccx(), err(), Notify(), ovf(), and TimerCntr().

5.13.4.4 TC0_t∗ TimerCntr::_pTC [private]

Definition at line 18 of file TimerCntr.h.

Referenced by CCEnable(), CCReg(), ClkSel(), Counter(), EventSetup(), IntLvlA(), IntLvlB(), Period(), TimerCntr(), WaveformGenMode(), and ∼TimerCntr().

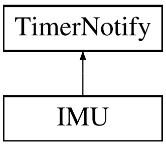
The documentation for this class was generated from the following files:

- TimerCntr.h
- TimerCntr.cpp

5.14 TimerNotify Class Reference

```
#include <TimerCntr.h>
```

Inheritance diagram for TimerNotify:



Public Member Functions

- virtual void err (uint8_t id)=0
- virtual void ovf (uint8_t id)=0
- virtual void ccx (uint8_t id, uint8_t idx)=0

5.14.1 Detailed Description

Definition at line 5 of file TimerCntr.h.

5.14.2 Member Function Documentation

5.14.2.1 virtual void TimerNotify::ccx (uint8_t *id,* uint8_t *idx*) [pure virtual]

Implemented in IMU.

Referenced by TimerCntr::ccx().

5.14.2.2 virtual void TimerNotify::err (uint8_t *id*) [pure virtual]

Implemented in IMU.

Referenced by TimerCntr::err().

5.14.2.3 virtual void TimerNotify::ovf (uint8_t *id*) [pure virtual]

Implemented in IMU.

Referenced by TimerCntr::ovf().

The documentation for this class was generated from the following file:

- TimerCntr.h

6 File Documentation

6.1 clksystem.cpp File Reference

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
```

Defines

- #define AVR_ENTER_CRITICAL_REGION()

  This macro will protect the following code from interrupts.

- #define AVR_LEAVE_CRITICAL_REGION() SREG = saved_sreg;

  This macro must always be used in conjunction with AVR_ENTER_CRITICAL_REGION so the interrupts are enabled again.

Functions

- void CCPWrite (volatile uint8_t ∗address, uint8_t value)

  CCP write helper function written in assembly.

- void clksystem_init ()

6.1.1 Define Documentation

6.1.1.1 #define AVR_ENTER_CRITICAL_REGION( )

Value:

```
uint8_t volatile saved_sreg = SREG; \
                                   cli();
```

This macro will protect the following code from interrupts.

Definition at line 10 of file clksystem.cpp.

Referenced by CCPWrite().

6.1.1.2 #define AVR_LEAVE_CRITICAL_REGION( ) SREG = saved_sreg;

This macro must always be used in conjunction with AVR_ENTER_CRITICAL_REGION so the interrupts are enabled again.

Definition at line 16 of file clksystem.cpp.

Referenced by CCPWrite().

6.1.2 Function Documentation

6.1.2.1 void CCPWrite (volatile uint8_t ∗ *address*, uint8_t *value*)

CCP write helper function written in assembly.

This function is written in assembly because of the timecritial operation of writing to the registers.

Parameters

address	A pointer to the address to write to.
value	The value to put in to the register.

Definition at line 26 of file clksystem.cpp.

References AVR_ENTER_CRITICAL_REGION, and AVR_LEAVE_CRITICAL_REGION.

```
{
    AVR_ENTER_CRITICAL_REGION( );

    volatile uint8_t * tmpAddr = address;

    asm volatile(
        "movw r30,  %0"       "\n\t"
        "ldi  r16,  %2"       "\n\t"
        "out   %3, r16"       "\n\t"
        "st     Z,  %1"       "\n\t"
        :
        : "r" (tmpAddr), "r" (value), "M" (0xD8), "i" (&CCP)
        : "r16", "r30", "r31"
    );

    AVR_LEAVE_CRITICAL_REGION( );
}
```

6.1.2.2 void clksystem_init ( )

Definition at line 58 of file clksystem.cpp.

Referenced by init().

```
{
    // This board does not have an external oscillator,
    // so we need to use the internal oscillator.
    // Enable the internal 32Mhz oscillator. Disables all the rest.
    OSC.CTRL = OSC_RC32MEN_bm | OSC_RC2MEN_bm | OSC_RC32KEN_bm;

    //OSC.XOSCCTRL = 0; // we don't care, just set this to some value.

    // Wait for the external oscilator. Don't wait forever though.
    int maxWait = 100; // Wait 1 second
    while (--maxWait && !(OSC.STATUS & (OSC_RC32MRDY_bm | OSC_RC32KRDY_bm | OSC_R
      C2MRDY_bm))) {
        _delay_loop_2(10);
    }

    OSC.DFLLCTRL = 0;
    DFLLRC32M.CTRL = 0x1;
    DFLLRC2M.CTRL = 0x1;

    // If the external oscilator is running, then we can switch the PLL
    // over to it and wait for the PLL to stabilize.
    if (OSC.STATUS & ( OSC_RC32MRDY_bm | OSC_RC2MRDY_bm)) {
        // Setup the PLL to use the internal 32Mhz oscillator and a
        // factor of 4 to get a 128Mhz PLL Clock.
        // Make sure this is disabled before we try and configure it.
        OSC.CTRL &= ~OSC_PLLEN_bm;

        OSC.PLLCTRL = OSC_PLLSRC_RC32M_gc | (16 << OSC_PLLFAC_gp);

        OSC.CTRL |= OSC_PLLEN_bm;

        //OSC.CTRL = CLK_PSADIV_16_gc;
        // Wait for OSC.STATUS to indicate that PLL is ready..
        while (!(OSC.STATUS & OSC_PLLRDY_bm)) {
        }

        // Okay, the PLL is up on the new clock. Set the prescalers then
        // switch over to the PLL.

        CCP = CCP_IOREG_gc;
        CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_2_2_gc;
        //CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;

        // When all is done, make the PLL be the clock source for the system.
        CCP = CCP_IOREG_gc;
        CLK.CTRL = CLK_SCLKSEL_PLL_gc;
        //CLK.CTRL = CLK_SCLKSEL_RC32M_gc;
        //CLK.CTRL = CLK_SCLKSEL_RC2M_gc;
    }
}
```

6.2 clksystem.cpp

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>


#define AVR_ENTER_CRITICAL_REGION( ) uint8_t volatile saved_sreg = SREG; \
                                     cli();

#define AVR_LEAVE_CRITICAL_REGION( ) SREG = saved_sreg;

void CCPWrite( volatile uint8_t * address, uint8_t value )
{
    AVR_ENTER_CRITICAL_REGION( );

    volatile uint8_t * tmpAddr = address;

    asm volatile(
        "movw r30,  %0"       "\n\t"
        "ldi  r16,  %2"       "\n\t"
        "out   %3, r16"       "\n\t"
        "st     Z,  %1"       "\n\t"
        :
        : "r" (tmpAddr), "r" (value), "M" (0xD8), "i" (&CCP)
        : "r16", "r30", "r31"
    );

    AVR_LEAVE_CRITICAL_REGION( );
}
```

```

/**********************************
Function: init
Purpose:
A: Setup the clocking system.
B: To Be Determined

We have an 8Mhz external oscillator. Set the PLL to multiply by
16, to give a peripheral clock of 128Mhz.
Set prescaler A to be 1, to get a 128Mhz peripheral clock.
Set prescaler B to be 2 to get a 64Mhz 2X speed clock
Set presclaer C to be 2 to get a 32Mhz CPU Clock

**********************************/
void clksystem_init()
{
    // This board does not have an external oscillator,
    // so we need to use the internal oscillator.
    // Enable the internal 32Mhz oscillator. Disables all the rest.
    OSC.CTRL = OSC_RC32MEN_bm | OSC_RC2MEN_bm | OSC_RC32KEN_bm;

    //OSC.XOSCCTRL = 0; // we don't care, just set this to some value.

    // Wait for the external oscilator. Don't wait forever though.
    int maxWait = 100; // Wait 1 second
    while (--maxWait && !(OSC.STATUS & (OSC_RC32MRDY_bm | OSC_RC32KRDY_bm | OSC_RC2MRDY_bm))) {
        _delay_loop_2(10);
    }

    OSC.DFLLCTRL = 0;
    DFLLRC32M.CTRL = 0x1;
    DFLLRC2M.CTRL = 0x1;

    // If the external oscilator is running, then we can switch the PLL
    // over to it and wait for the PLL to stabilize.
    if (OSC.STATUS & ( OSC_RC32MRDY_bm | OSC_RC2MRDY_bm)) {
        // Setup the PLL to use the internal 32Mhz oscillator and a
        // factor of 4 to get a 128Mhz PLL Clock.

        // Make sure this is disabled before we try and configure it.
        OSC.CTRL &= ~OSC_PLLEN_bm;

        OSC.PLLCTRL = OSC_PLLSRC_RC32M_gc | (16 << OSC_PLLFAC_gp);

        OSC.CTRL |= OSC_PLLEN_bm;

        //OSC.CTRL = CLK_PSADIV_16_gc;
        // Wait for OSC.STATUS to indicate that PLL is ready..
        while (!(OSC.STATUS & OSC_PLLRDY_bm)) {
        }

        // Okay, the PLL is up on the new clock. Set the prescalers then
        // switch over to the PLL.

        CCP = CCP_IOREG_gc;
        CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_2_2_gc;
        //CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;

        // When all is done, make the PLL be the clock source for the system.
        CCP = CCP_IOREG_gc;
        CLK.CTRL = CLK_SCLKSEL_PLL_gc;
        //CLK.CTRL = CLK_SCLKSEL_RC32M_gc;
        //CLK.CTRL = CLK_SCLKSEL_RC2M_gc;
    }
}
```

6.3 clksystem.h File Reference

Functions

- int clksystem_init ()

6.3.1 Function Documentation

6.3.1.1 int clksystem_init ( )

Definition at line 58 of file clksystem.cpp.

Referenced by init().

```
{
    // This board does not have an external oscillator,
    // so we need to use the internal oscillator.
    // Enable the internal 32Mhz oscillator. Disables all the rest.
    OSC.CTRL = OSC_RC32MEN_bm | OSC_RC2MEN_bm | OSC_RC32KEN_bm;

    //OSC.XOSCCTRL = 0; // we don’t care, just set this to some value.

    // Wait for the external oscilator. Don’t wait forever though.
    int maxWait = 100; // Wait 1 second
    while (--maxWait && !(OSC.STATUS & (OSC_RC32MRDY_bm | OSC_RC32KRDY_bm | OSC_R
      C2MRDY_bm))) {
        _delay_loop_2(10);
    }

    OSC.DFLLCTRL = 0;
    DFLLRC32M.CTRL = 0x1;
    DFLLRC2M.CTRL = 0x1;

    // If the external oscilator is running, then we can switch the PLL
    // over to it and wait for the PLL to stabilize.
    if (OSC.STATUS & ( OSC_RC32MRDY_bm | OSC_RC2MRDY_bm)) {
        // Setup the PLL to use the internal 32Mhz oscillator and a
        // factor of 4 to get a 128Mhz PLL Clock.

        // Make sure this is disabled before we try and configure it.
        OSC.CTRL &= ~OSC_PLLEN_bm;

        OSC.PLLCTRL = OSC_PLLSRC_RC32M_gc | (16 << OSC_PLLFAC_gp);

        OSC.CTRL |= OSC_PLLEN_bm;

        //OSC.CTRL = CLK_PSADIV_16_gc;
        // Wait for OSC.STATUS to indicate that PLL is ready..
        while (!(OSC.STATUS & OSC_PLLRDY_bm)) {
        }
        // Okay, the PLL is up on the new clock. Set the prescalers then
        // switch over to the PLL.

        CCP = CCP_IOREG_gc;
        CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_2_2_gc;
        //CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;

        // When all is done, make the PLL be the clock source for the system.
        CCP = CCP_IOREG_gc;
        CLK.CTRL = CLK_SCLKSEL_PLL_gc;
        //CLK.CTRL = CLK_SCLKSEL_RC32M_gc;
        //CLK.CTRL = CLK_SCLKSEL_RC2M_gc;
    }
}
```

6.4 clksystem.h

```

#ifndef _CLKSYSTEM_
#   define _CLKSYSTEM_ 1


int clksystem_init();


#endif
```

6.5 CmdProcessor.cpp File Reference

#include <stdlib.h>

#include <string.h>

#include "CmdProcessor.h"

6.6 CmdProcessor.cpp

```

#include <stdlib.h>
#include <string.h>
#include "CmdProcessor.h"

CmdProcessor::CmdProcessor(HardwareSerial* pHW)
{
    _pHW = pHW;

    _pCmdString = (char*)malloc(128);
    _pCmd = 0;
    _pCmdTerm = (char*)malloc(3);
    strcpy(_pCmdTerm,"\n\r");
    _pCmdDelim = (char*)malloc(3);
    strcpy(_pCmdDelim," \t");
    resetCmd();
}

CmdProcessor::~CmdProcessor()
```

```
{
    if (_pHW) {
        _pHW->end();
    }
    free(_pCmdString);
    free(_pCmdDelim);
    free(_pCmdTerm);
}

char* CmdProcessor::cmdTerm() { return _pCmdTerm; }

void CmdProcessor::cmdTerm(char* t)
{
    free(_pCmdTerm);
    _pCmdTerm = (char*)malloc(strlen(t) + 1);
    strcpy(_pCmdTerm,t);
}

const char* CmdProcessor::cmdDelim()
{
    return _pCmdDelim;
}

void CmdProcessor::cmdDelim(const char* d)
{
    free(_pCmdDelim);
    _pCmdDelim = (char*)malloc(strlen(d) + 1);
    strcpy(_pCmdDelim,d);
}

bool CmdProcessor::checkCommands()
{
    while (_pHW->available() > 0) {
        unsigned char c = _pHW->read();
        if (strchr(_pCmdTerm,c) != 0) {
            if (_cmdPos > 0) {
                // Done with this command.
                _pCmdString[_cmdPos] = 0; // Null terminate command
                processCmd();
                return 1;
            } else {
                _pHW->print("Ok\n");
            }
        } else {
            _pCmdString[_cmdPos++] = c;
        }
    }
    return 0;
}

void CmdProcessor::processCmd()
{
    // See if the command delimiter exists in the
    // command. if it does not, then the command
    // is the entire string.
    if (strpbrk(_pCmdString,_pCmdDelim)) {
        _pCmd = strtok(_pCmdString,_pCmdDelim);
        char* pTok = strtok(0,_pCmdDelim);
        int i = 0;
        while (i < 10 && pTok) {
            _pTokens[i++] = pTok;
            pTok = strtok(0,_pCmdDelim);
        }
        _paramCnt = i;
        _validCmd = true;
    } else {
        _pCmd = _pCmdString;
        _paramCnt = 0;
        _validCmd = true;
    }
}

void CmdProcessor::resetCmd()
{
    _cmdPos = 0;
    _validCmd = false;
    _paramCnt = 0;
}

const char* CmdProcessor::getCmd()
{
    return _pCmd;
}

uint8_t CmdProcessor::paramCnt()
{
    return _paramCnt;
}



void CmdProcessor::getParam(uint8_t idx,uint16_t &p)
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}

void CmdProcessor::getParam(uint8_t idx,uint8_t &p)
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}

void CmdProcessor::getParam(uint8_t idx,int &p)
{
    if (idx < _paramCnt) {
        p = atoi(_pTokens[idx]);
    }
}

void CmdProcessor::getParam(uint8_t idx,long &l)
{
    if (idx < _paramCnt) {
        l = atol(_pTokens[idx]);
    }
}

void CmdProcessor::getParam(uint8_t idx,double &p)
{
    if (idx < _paramCnt) {
        uint8_t nScans;
        nScans = sscanf(_pTokens[idx],"%lf", &p);
        //p = atof(_pTokens[idx]);
    }
}

void CmdProcessor::getParam(uint8_t idx,char*& p, uint8_t maxlen)
{
    if (idx < _paramCnt) {
        strncpy(p,_pTokens[idx],maxlen);
    }
}

```

6.7 CmdProcessor.h File Reference

```
#include <avr/io.h>
#include "HardwareSerial.h"
```

Classes

- class CmdProcessor

6.8 CmdProcessor.h

```
#ifndef CMDPROCESSOR_H
#define CMDPROCESSOR_H

#include <avr/io.h>
#include "HardwareSerial.h"

class CmdProcessor
{
protected:
    HardwareSerial* _pHW;
    char*           _pTokens[10];
    char*           _pCmd;
    char*           _pCmdString;
    uint8_t         _cmdPos;
    bool            _validCmd;
    char*           _pCmdTerm;
    char*           _pCmdDelim;
    uint8_t         _paramCnt;

public:
    CmdProcessor(HardwareSerial* pHW);
    ~CmdProcessor();

    bool checkCommands();
    char* cmdTerm();
    void cmdTerm(char*);
    void resetCmd();
    const char* cmdDelim();
    void cmdDelim(const char*);
    const char* getCmd();
    uint8_t paramCnt();
    void getParam(uint8_t idx,uint8_t &p);
    void getParam(uint8_t idx,uint16_t &p);
    void getParam(uint8_t idx,long &l);
    void getParam(uint8_t idx,int &p);
    void getParam(uint8_t idx,double &f);
    void getParam(uint8_t idx,char*& p, uint8_t maxlen=128);
protected:
    void processCmd();

};

#endif
```

6.9 cpp_hacks.cpp File Reference

```
#include "cpp_hacks.h"
```

Functions

- int __cxa_guard_acquire (__guard *g)
- void __cxa_guard_release (__guard *g)
- void __cxa_guard_abort (__guard *)
- void __cxa_pure_virtual (void)

6.9.1 Function Documentation

6.9.1.1 void __cxa_guard_abort (__guard *)

Definition at line 5 of file cpp_hacks.cpp.

```
{};
```

6.9.1.2 int __cxa_guard_acquire (__guard * *g*)

Definition at line 3 of file cpp_hacks.cpp.

```
{return !*(char *)(g);};
```

6.9.1.3 void __cxa_guard_release (__guard * *g*)

Definition at line 4 of file cpp_hacks.cpp.

```
{*(char *)g = 1;};
```

6.9.1.4 void __cxa_pure_virtual (void)

Definition at line 7 of file cpp_hacks.cpp.

```
{};
```

6.10 cpp_hacks.cpp

```
00001 #include "cpp_hacks.h"
00002 
00003 int __cxa_guard_acquire(__guard *g) {return !*(char *)(g);};
00004 void __cxa_guard_release (__guard *g) {*(char *)g = 1;};
00005 void __cxa_guard_abort (__guard *) {};
00006 
00007 void __cxa_pure_virtual(void) {};
00008 
```

6.11 cpp_hacks.h File Reference

Functions

- __extension__ typedef int __guard __attribute__ ((mode(__DI__)))
- int __cxa_guard_acquire (__guard *)
- void __cxa_guard_release (__guard *)
- void __cxa_guard_abort (__guard *)
- void __cxa_pure_virtual (void)

6.11.1 Function Documentation

6.11.1.1 __extension__ typedef int __guard __attribute__ ((mode(__DI__)))

6.11.1.2 void __cxa_guard_abort (__guard *)

Definition at line 5 of file cpp_hacks.cpp.

```
{};
```

6.11.1.3 int __cxa_guard_acquire (__guard *)

Definition at line 3 of file cpp_hacks.cpp.

```
{return !*(char *)(g);};
```

6.11.1.4 void __cxa_guard_release (__guard *)

Definition at line 4 of file cpp_hacks.cpp.

```
{*(char *)g = 1;};
```

6.11.1.5 void __cxa_pure_virtual (void)

Definition at line 7 of file cpp_hacks.cpp.

```
{};
```

6.12 cpp_hacks.h

```
__extension__ typedef int __guard __attribute__((mode (__DI__)));

extern "C" int __cxa_guard_acquire(__guard *);
extern "C" void __cxa_guard_release (__guard *);
extern "C" void __cxa_guard_abort (__guard *);
extern "C" void __cxa_pure_virtual(void);

```

6.13 Documentation.html File Reference

6.14 Documentation.html

```

```

6.15 fifo.cpp File Reference

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "NewDel.h"
#include "fifo.h"
```

Functions

- void FifoTest (HardwareSerial &ds)

6.15.1 Function Documentation

6.15.1.1 void FifoTest (HardwareSerial & *ds*)

Test function to validate that the fifo works as expected. It is always a good idea to validate your data structures. Languages like Python or Perl have such validation built in, while C++ does not. This function can be used in a special build to run through some test cases and make sure that the fifo behaves as we exect, especially in the edge cases, such as when it gets full, wraps around, etc.

Definition at line 110 of file fifo.cpp.

References buffer, Fifo::clear(), Fifo::count(), Fifo::pop(), Print::print(), and Fifo::push().

```
{
    Fifo f1(20);
    Fifo::FifoType x;
    char    buffer[128];

    for (x=1;x<25;x++) {
        f1.push(&x);
        sprintf(buffer,"Fifo push %d count %d\n",x,f1.count());
        ds.print(buffer);
    }
    f1.clear();
    sprintf(buffer,"Fifo count %d\n",f1.count());
    ds.print(buffer);

    for (x=0;x<10;x++) {
        f1.push(&x);
    }
    sprintf(buffer,"Fifo count %d. Expected 10\n",f1.count());
    ds.print(buffer);

    Fifo::FifoType y;
    for (x=1;x<15;x++) {
        f1.pop(&y);
        sprintf(buffer,"Fifo pop %d count %d\n",y,f1.count());
        ds.print(buffer);
    }
}
```

6.16 fifo.cpp

```
#include <stdio.h>
#include <string.h>
#include <math.h>

#include "NewDel.h"
#include "fifo.h"

Fifo::Fifo(uint8_t size)
{
    _size = size;
    _pdata = (FifoType*)malloc(_size * sizeof(FifoType));
    clear();
}

```

```
void Fifo::clear()
{
    _start = _end = _pdata;
}



uint8_t Fifo::count()
{
    if (_end == _start) return 0;
    if (_end > _start) {
        return _end - _start;
    }
    return _size - (_start - _end);
}

bool Fifo::full()
{
    return (_start - _end)  == 1 || (_end - _start) == _size;
}

bool Fifo::empty()
{
    return (_start == _end);
}

int8_t Fifo::push(FifoType* d)
{
    if (full()) return -1;
    *(_end++) = *d;

    // Wrap the end back to the beginning.
    if ((_end - _pdata) > _size) {
        _end = _pdata;
    }

        return 0;
}

int8_t Fifo::pop(FifoType* pD)
{
    if (empty()) {
        return -1; // Nothing else to do
    }
    *pD = *(_start++);
    if ((_start - _pdata) > _size) {
        _start = _pdata;
    }
    return 0;
}

void FifoTest(HardwareSerial& ds)
{
    Fifo f1(20);
    Fifo::FifoType x;
    char    buffer[128];

    for (x=1;x<25;x++) {
        f1.push(&x);
        sprintf(buffer,"Fifo push %d count %d\n",x,f1.count());
        ds.print(buffer);
    }
    f1.clear();
    sprintf(buffer,"Fifo count %d\n",f1.count());
    ds.print(buffer);

    for (x=0;x<10;x++) {
        f1.push(&x);
    }
    sprintf(buffer,"Fifo count %d. Expected 10\n",f1.count());
    ds.print(buffer);

    Fifo::FifoType y;
    for (x=1;x<15;x++) {
        f1.pop(&y);
        sprintf(buffer,"Fifo pop %d count %d\n",y,f1.count());
        ds.print(buffer);
    }
}

```

6.17 fifo.h File Reference

`#include <inttypes.h>`

`#include "HardwareSerial.h"`

Classes

- class Fifo

  Fifo Class for unsigned 8 bit values.

Functions

- void FifoTest (HardwareSerial &ds)

6.17.1 Function Documentation

6.17.1.1 void FifoTest (HardwareSerial & *ds*)

Test function to validate that the fifo works as expected. It is always a good idea to validate your data structures. Languages like Python or Perl have such validation built in, while C++ does not. This function can be used in a special build to run through some test cases and make sure that the fifo behaves as we exect, especially in the edge cases, such as when it gets full, wraps around, etc.

Definition at line 110 of file fifo.cpp.

References buffer, Fifo::clear(), Fifo::count(), Fifo::pop(), Print::print(), and Fifo::push().

```
{
    Fifo f1(20);
    Fifo::FifoType x;
    char    buffer[128];

    for (x=1;x<25;x++) {
        f1.push(&x);
        sprintf(buffer,"Fifo push %d count %d\n",x,f1.count());
        ds.print(buffer);
    }
    f1.clear();
    sprintf(buffer,"Fifo count %d\n",f1.count());
    ds.print(buffer);

    for (x=0;x<10;x++) {
        f1.push(&x);
    }
    sprintf(buffer,"Fifo count %d. Expected 10\n",f1.count());
    ds.print(buffer);

    Fifo::FifoType y;
    for (x=1;x<15;x++) {
        f1.pop(&y);
        sprintf(buffer,"Fifo pop %d count %d\n",y,f1.count());
        ds.print(buffer);
    }
}
```

6.18 fifo.h

```

#ifndef _fifo_h
#define _fifo_h

#include <inttypes.h>
#include "HardwareSerial.h"

class Fifo
{
public:

    typedef uint8_t  FifoType;
private:

    FifoType*       _pdata;
    FifoType*       _start;
    FifoType*       _end;
    uint8_t         _size;

public:
    Fifo(uint8_t size);


    int8_t push(FifoType*);
    int8_t pop(FifoType* pData);
    uint8_t count();
    bool full();
    bool empty();
    void clear();
};

extern void FifoTest(HardwareSerial& ds);

#endif
```

6.19 GyroAcc.cpp File Reference

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <string.h>
#include "NewDel.h"
#include "clksystem.h"
#include "time.h"
#include "HardwareSerial.h"
#include "GyroCmdProcessor.h"
#include "I2C_Master.h"
#include "IMU.h"
#include "IMUManager.h"
#include "TimerCntr.h"
#include "Port.h"
#include "MyDriver.h"
#include "Revision.h"
```

Functions

- void timer_init ()

  Declared in TimerCore.cpp.

- void init ()
- void processCmd (CmdProcessor &cmdproc)
- int main ()

Variables

- HardwareSerial ∗ pdbgserial = 0
- DebugPort ∗ pdbgport = 0

6.19.1 Function Documentation

6.19.1.1 void init ( )

Definition at line 23 of file GyroAcc.cpp.

References clksystem_init(), and timer_init().

Referenced by main().

```
            {

    clksystem_init();
    timer_init();
}
```

6.19.1.2 int main ( )

Create timers for each of the IMU Masters. I use the TCx1 timers which are less capable, but it hardly matters as these are just setting a timer callback.

Give the rate a default value. Start low

This bit just toggles the light

Definition at line 34 of file GyroAcc.cpp.

References I2C_Master::begin(), HardwareSerial::begin(), buffer, HardwareSerial::enable(), init(), pdbgport, Print::print(), IMU::SetDebugPort(), and IMU::SetDebugPort2().

```
{
    char    buffer[100];
    init();

    PMIC.CTRL |= PMIC_HILVLEN_bm | PMIC_LOLVLEN_bm | PMIC_MEDLVLEN_bm;
    sei();

#ifdef USE_DEBUGPORT_2

    DebugPort dbgPort2(&PORTE);
    dbgPort2.PinMask(0xF0,5);
    dbgPort2.SetState(0);
#endif


    HardwareSerial dbgserial(&USARTF1, &PORTF, PIN6_bm, PIN7_bm);
    dbgserial.begin(115200);
    pdbgserial = &dbgserial;
    pdbgserial->enable(false);

    // Instantiate a commad processor.
    // Specify the USART, PORT, rxPin and txPin and the baud rate.
    HardwareSerial cmdSerial(&USARTD0, &PORTD, PIN2_bm, PIN3_bm);
    cmdSerial.begin(115200);
    //cmdSerial.begin(285000);
    //cmdSerial.begin(921600);

    CCP = CCP_IOREG_gc;
    MCU.MCUCR = MCU_JTAGD_bm;
    DebugPort dbgPort(&PORTB);
    dbgPort.PinMask(0xF0,4);
    dbgPort.SetState(0xf);
    dbgPort.SetState(0);
    pdbgport = &dbgPort;

    I2C_Master  *pMaster[4];
    I2C_Master hand(&TWIC);
    I2C_Master single(&TWID);
    I2C_Master pair1(&TWIE);
    I2C_Master pair2(&TWIF);

    pMaster[0] = &hand;
    pMaster[1] = &single; // Pinkie
    pMaster[2] = &pair1;  // Ring + Middle
    pMaster[3] = &pair2;  // Index + Thumb

    for (int x=0;x<4;x++) {
        if (pMaster[x]) {
            pMaster[x]->begin(400e3);
        }
    }

    // Give the hardware time to stabilize..
    _delay_ms(1000);

    // When constructed without a list of ID's, the IMU will query known
    // ID's 0xD0 and 0xD2 for the Gyro, and then 0x30 and 0x32 respectively
    // for the ACC.
    IMU     hand_imu(&hand);
    IMU     single_imu(&single);
    IMU     pair1_imu(&pair1);
    IMU     pair2_imu(&pair2);

    // These all share the debug port, so they must operate
    // in sequence. If I went back to a parallel operation, then
    // this would have to change.
    hand_imu.SetDebugPort(&dbgPort);

    single_imu.SetDebugPort(&dbgPort);

    pair1_imu.SetDebugPort(&dbgPort);

    pair2_imu.SetDebugPort(&dbgPort);

#ifdef USE_DEBUGPORT_2

    hand_imu.SetDebugPort2(&dbgPort2);
    single_imu.SetDebugPort2(&dbgPort2);
    pair1_imu.SetDebugPort2(&dbgPort2);
    pair2_imu.SetDebugPort2(&dbgPort2);
#endif


    TimerCntr   tcA(&TCC1);

    IMUManager imumgr(&cmdSerial);
    imumgr.SetBlueLed(&PORTJ, PIN7_bm);
    imumgr.LedOff();
    imumgr.SetTimer(&tcA);
    imumgr.AddIMU(&hand_imu);
    imumgr.AddIMU(&single_imu);
    imumgr.AddIMU(&pair1_imu);
    imumgr.AddIMU(&pair2_imu);
    imumgr.SampleRate(10);

    GyroCmdProcessor cmdproc(&cmdSerial,&pMaster[0],&imumgr);

    sprintf(buffer,"Welcome to Gyro Glove.\nRev %d.%d.%d Date: %02d/%02d/%04d Bui
      lt at: %02d:%02d\n",
        RevMajor, RevMinor, RevInc,
        DateMonth, DateDay, DateYear,
        TimeHour, TimeMin
        );
    cmdSerial.print(buffer);

    TimerCntr mdtc(&TCD0);
    MyDriver md;
    md.SetTimer(&mdtc);

    cmdSerial.print("Starting endless loop\n");

    while(1) {
        cmdproc.Loop();
        imumgr.Loop();
    }

    return 0;
}
```

6.19.1.3 void processCmd (CmdProcessor & *cmdproc*)

6.19.1.4 void timer_init ( )

Declared in TimerCore.cpp.

Referenced by init().

6.19.2 Variable Documentation

6.19.2.1 DebugPort∗ pdbgport = 0

Definition at line 32 of file GyroAcc.cpp.

Referenced by main().

6.19.2.2 HardwareSerial∗ pdbgserial = 0

Definition at line 31 of file GyroAcc.cpp.

6.20 GyroAcc.cpp

```
00001
00002 #include <avr/io.h>
00003 #include <avr/interrupt.h>
00004 #include <util/delay.h>
00005 #include <string.h>
00006 #include "NewDel.h"
00007 #include "clksystem.h"
00008 #include "time.h"
00009
00010 #include "HardwareSerial.h"
00011 #include "GyroCmdProcessor.h"
00012 #include "I2C_Master.h"
00013 #include "IMU.h"
00014 #include "IMUManager.h"
00015 #include "TimerCntr.h"
00016 #include "Port.h"
00017 #include "MyDriver.h"
00018 #include "Revision.h"
00019
00021 extern void timer_init();
00022
00023 void init() {
00024
00025     clksystem_init();
00026     timer_init();
00027 }
00028
00029 void processCmd(CmdProcessor& cmdproc);
00030
00031 HardwareSerial* pdbgserial = 0;
00032 DebugPort*      pdbgport = 0;
00033
00034 int main()
00035 {
00036     char    buffer[100];
00037     init();
00038
00039     PMIC.CTRL |= PMIC_HILVLEN_bm | PMIC_LOLVLEN_bm | PMIC_MEDLVLEN_bm;
00040     sei();
00041
00042 #ifdef USE_DEBUGPORT_2
00043     DebugPort dbgPort2(&PORTE);
00044     dbgPort2.PinMask(0xF0,5);
00045     dbgPort2.SetState(0);
00046 #endif
00047
00048     HardwareSerial dbgserial(&USARTF1, &PORTF, PIN6_bm, PIN7_bm);
00049     dbgserial.begin(115200);
00050     pdbgserial = &dbgserial;
00051     pdbgserial->enable(false);
00052
00053     // Instantiate a commad processor.
```

```
    // Specify the USART, PORT, rxPin and txPin and the baud rate.
    HardwareSerial cmdSerial(&USARTD0, &PORTD, PIN2_bm, PIN3_bm);
    cmdSerial.begin(115200);
    //cmdSerial.begin(285000);
    //cmdSerial.begin(921600);

    CCP = CCP_IOREG_gc;
    MCU.MCUCR = MCU_JTAGD_bm;
    DebugPort dbgPort(&PORTB);
    dbgPort.PinMask(0xF0,4);
    dbgPort.SetState(0xf);
    dbgPort.SetState(0);
    pdbgport = &dbgPort;

    I2C_Master  *pMaster[4];
    I2C_Master hand(&TWIC);
    I2C_Master single(&TWID);
    I2C_Master pair1(&TWIE);
    I2C_Master pair2(&TWIF);

    pMaster[0] = &hand;
    pMaster[1] = &single; // Pinkie
    pMaster[2] = &pair1;  // Ring + Middle
    pMaster[3] = &pair2;  // Index + Thumb

    for (int x=0;x<4;x++) {
        if (pMaster[x]) {
            pMaster[x]->begin(400e3);
        }
    }

    // Give the hardware time to stabilize..
    _delay_ms(1000);

    // When constructed without a list of ID's, the IMU will query known
    // ID's 0xD0 and 0xD2 for the Gyro, and then 0x30 and 0x32 respectively
    // for the ACC.
    IMU     hand_imu(&hand);
    IMU     single_imu(&single);
    IMU     pair1_imu(&pair1);
    IMU     pair2_imu(&pair2);

    // These all share the debug port, so they must operate
    // in sequence. If I went back to a parallel operation, then
    // this would have to change.
    hand_imu.SetDebugPort(&dbgPort);

    single_imu.SetDebugPort(&dbgPort);

    pair1_imu.SetDebugPort(&dbgPort);

    pair2_imu.SetDebugPort(&dbgPort);

#ifdef USE_DEBUGPORT_2
    hand_imu.SetDebugPort2(&dbgPort2);
    single_imu.SetDebugPort2(&dbgPort2);
    pair1_imu.SetDebugPort2(&dbgPort2);
    pair2_imu.SetDebugPort2(&dbgPort2);
#endif

    TimerCntr   tcA(&TCC1);

    IMUManager imumgr(&cmdSerial);
    imumgr.SetBlueLed(&PORTJ, PIN7_bm);
    imumgr.LedOff();
    imumgr.SetTimer(&tcA);
    imumgr.AddIMU(&hand_imu);
    imumgr.AddIMU(&single_imu);
    imumgr.AddIMU(&pair1_imu);
    imumgr.AddIMU(&pair2_imu);
    imumgr.SampleRate(10);

    GyroCmdProcessor cmdproc(&cmdSerial,&pMaster[0],&imumgr);

    sprintf(buffer,"Welcome to Gyro Glove.\nRev %d.%d.%d Date: %02d/%02d/%04d Built at: %02d:%02d\n",
        RevMajor, RevMinor, RevInc,
        DateMonth, DateDay, DateYear,
        TimeHour, TimeMin
        );
    cmdSerial.print(buffer);

    TimerCntr mdtc(&TCD0);
    MyDriver md;
    md.SetTimer(&mdtc);

    cmdSerial.print("Starting endless loop\n");

    while(1) {
        cmdproc.Loop();
        imumgr.Loop();
    }

    return 0;
}


```

6.21 HardwareSerial.cpp File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "HardwareSerial.h"
```

Classes

- struct ring_buffer

Defines

- #define RX_BUFFER_SIZE 128
- #define SERIAL_ISR_DEF(usart)

Functions

- SERIAL_ISR_DEF (USARTC0)
- SERIAL_ISR_DEF (USARTC1)
- SERIAL_ISR_DEF (USARTD0)
- SERIAL_ISR_DEF (USARTD1)
- SERIAL_ISR_DEF (USARTE0)
- SERIAL_ISR_DEF (USARTE1)
- SERIAL_ISR_DEF (USARTF0)
- void store_char (unsigned char c, ring_buffer ∗rx_buffer)
- static void SetPointer (USART_t ∗usart, HardwareSerial ∗p)

6.21.1 Define Documentation

6.21.1.1 #define RX_BUFFER_SIZE 128

Definition at line 14 of file HardwareSerial.cpp.

Referenced by HardwareSerial::available(), HardwareSerial::HardwareSerial(), HardwareSerial::read(), and store_char().

6.21.1.2 #define SERIAL_ISR_DEF(*usart*)

Value:

```
static HardwareSerial*  usart##cp = 0;\
ISR(usart##_RXC_vect) {\
    if (usart##cp) usart##cp->rxc();\
}\
ISR(usart##_DRE_vect) {\
    if (usart##cp) usart##cp->dre();\
}\
ISR(usart##_TXC_vect) {\
    if (usart##cp) usart##cp->txc();\
}
```

Definition at line 24 of file HardwareSerial.cpp.

6.21.2 Function Documentation

6.21.2.1 SERIAL_ISR_DEF (USARTC0)

6.21.2.2 SERIAL_ISR_DEF (USARTC1)

6.21.2.3 SERIAL_ISR_DEF (USARTD1)

6.21.2.4 SERIAL_ISR_DEF (USARTF0)

6.21.2.5 SERIAL_ISR_DEF (USARTE1)

6.21.2.6 SERIAL_ISR_DEF (USARTE0)

6.21.2.7 SERIAL_ISR_DEF (USARTD0)

6.21.2.8 static void SetPointer (USART_t ∗ *usart*, HardwareSerial ∗ *p*) [static]

Definition at line 61 of file HardwareSerial.cpp.

Referenced by HardwareSerial::begin2x(), HardwareSerial::end(), HardwareSerial::HardwareSerial(), TimerCntr::TimerCntr(), HardwareSerial::∼HardwareSerial(), and TimerCntr::∼TimerCntr().

```
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if(usart == &USARTC0) {
        USARTC0cp = p;
    } else if (usart == &USARTC1) {
        USARTC1cp = p;
    } else if (usart ==  &USARTD0) {
        USARTD0cp = p;
    } else if (usart ==  &USARTD1) {
        USARTD1cp = p;
    } else if (usart ==  &USARTE0) {
        USARTE0cp = p;
    } else if (usart ==  &USARTE1) {
        USARTE1cp = p;
    } else if (usart ==  &USARTF0) {
        USARTF0cp = p;
#if defined(USARTF1_RXC_vect)
    } else if (usart ==  &USARTF1) {
        USARTF1cp = p;
#endif
    }
}
```

6.21.2.9 void store_char (unsigned char *c*, ring_buffer ∗ *rx_buffer*) [inline]

Definition at line 47 of file HardwareSerial.cpp.

References ring_buffer::buffer, ring_buffer::head, RX_BUFFER_SIZE, and ring_buffer::tail.

Referenced by HardwareSerial::rxc().

```
{
    int i = (rx_buffer->head + 1) % RX_BUFFER_SIZE;

    // if we should be storing the received character into the location
    // just before the tail (meaning that the head would advance to the
    // current location of the tail), we're about to overflow the buffer
    // and so we don't write the character or advance the head.
    if (i != rx_buffer->tail) {
        rx_buffer->buffer[rx_buffer->head] = c;
        rx_buffer->head = i;
    }
}
```

6.22 HardwareSerial.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#include "HardwareSerial.h"

// Define constants and variables for buffering incoming serial data.  We're
// using a ring buffer (I think), in which rx_buffer_head is the index of the
// location to which to write the next incoming character and rx_buffer_tail
// is the index of the location from which to read.
#define RX_BUFFER_SIZE 128

struct ring_buffer {
    unsigned char buffer[RX_BUFFER_SIZE];
    int head;
    int tail;
};

```

```
// Generate all of the ISR handlers.. hook them up to a class if/when a class
// is instantiated for a particular USART.
#define SERIAL_ISR_DEF(usart) \
static HardwareSerial*  usart##cp = 0;\
ISR(usart##_RXC_vect) {\
    if (usart##cp) usart##cp->rxc();\
}\
ISR(usart##_DRE_vect) {\
    if (usart##cp) usart##cp->dre();\
}\
ISR(usart##_TXC_vect) {\
    if (usart##cp) usart##cp->txc();\
}

SERIAL_ISR_DEF(USARTC0);
SERIAL_ISR_DEF(USARTC1);
SERIAL_ISR_DEF(USARTD0);
SERIAL_ISR_DEF(USARTD1);
SERIAL_ISR_DEF(USARTE0);
SERIAL_ISR_DEF(USARTE1);
SERIAL_ISR_DEF(USARTF0);
#if defined(USARTF1_RXC_vect)
SERIAL_ISR_DEF(USARTF1);
#endif

inline void store_char(unsigned char c, ring_buffer *rx_buffer)
{
    int i = (rx_buffer->head + 1) % RX_BUFFER_SIZE;

    // if we should be storing the received character into the location
    // just before the tail (meaning that the head would advance to the
    // current location of the tail), we're about to overflow the buffer
    // and so we don't write the character or advance the head.
    if (i != rx_buffer->tail) {
        rx_buffer->buffer[rx_buffer->head] = c;
        rx_buffer->head = i;
    }
}

static void SetPointer(USART_t* usart,HardwareSerial* p)
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if(usart == &USARTC0) {
        USARTC0cp = p;
    } else if (usart == &USARTC1) {
        USARTC1cp = p;
    } else if (usart ==  &USARTD0) {
        USARTD0cp = p;
    } else if (usart ==  &USARTD1) {
        USARTD1cp = p;
    } else if (usart ==  &USARTE0) {
        USARTE0cp = p;
    } else if (usart ==  &USARTE1) {
        USARTE1cp = p;
    } else if (usart ==  &USARTF0) {
        USARTF0cp = p;
#if defined(USARTF1_RXC_vect)
    } else if (usart ==  &USARTF1) {
        USARTF1cp = p;
#endif
```

```
    }
}


void HardwareSerial::rxc()
{
    unsigned char c = _usart->DATA;
    store_char(c,_rx_buffer);
}

void HardwareSerial::dre()
{
}

void HardwareSerial::txc()
{
}


// Constructors ////////////////////////////////////////////////////////////////

HardwareSerial::HardwareSerial(
    USART_t     *usart,
    PORT_t      *port,
    uint8_t      in_bm,
    uint8_t      out_bm)
{
    _rx_buffer = (ring_buffer*)malloc(RX_BUFFER_SIZE+2*sizeof(int));
    _usart     = usart;
    _port      = port;
    _in_bm     = in_bm;
    _out_bm    = out_bm;
    _bsel      = 0;
    _bscale    = 0;
    _baudrate  = 9600;
    _bEn       = true;
    SetPointer(_usart,this);
}

HardwareSerial::~HardwareSerial()
{
    end();
    free(_rx_buffer);
    _rx_buffer = 0;
    SetPointer(_usart,0);
}

// Public Methods //////////////////////////////////////////////////////////////

void HardwareSerial::begin(long baud,int8_t bscale)
{
    uint16_t BSEL;
    _bscale = bscale;
    _baudrate = baud;

    float fPER = F_CPU;
    float fBaud = baud;

    _port->DIRCLR = _in_bm;  // input
    _port->DIRSET = _out_bm; // output

    // set the baud rate
    if (bscale >= 0) {
        BSEL = fPER/((1 << bscale) * 16 * baud) - 1;
        //BSEL = F_CPU / 16 / baud - 1;
    } else {
        bscale = -1 * bscale;
        BSEL = (1 << bscale) * (fPER/(16.0 * fBaud) - 1);
    }

    _usart->BAUDCTRLA = (uint8_t)BSEL;
    _usart->BAUDCTRLB = ((bscale & 0xf) << 4) | ((BSEL & 0xf00) >> 8);

    // enable Rx and Tx
    _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
    // enable interrupt
    _usart->CTRLA = USART_RXCINTLVL_HI_gc;

    // Char size, parity and stop bits: 8N1
    _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
}

void HardwareSerial::begin2x(long baud,int8_t bscale)
{
    uint16_t baud_setting;
    _bscale = bscale;
    _baudrate = baud;

    // TODO: Serial. Fix serial double clock.
    long fPER = F_CPU * 4;

    _port->DIRCLR = _in_bm;  // input
    _port->DIRSET = _out_bm; // output

    // set the baud rate using the 2X calculations
    _usart->CTRLB |= 1 << 1; // the last 1 was the _u2x value
    baud_setting = fPER / 8 / baud - 1;

    _usart->BAUDCTRLA = (uint8_t)baud_setting;
    _usart->BAUDCTRLB = baud_setting >> 8;


    // enable Rx and Tx
    _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
    // enable interrupt
    _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
;

    // Char size, parity and stop bits: 8N1
    _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
    SetPointer(_usart,this);
}

void HardwareSerial::end()
{
    SetPointer(_usart,(HardwareSerial*)0);

    // disable Rx and Tx
    _usart->CTRLB &= ~(USART_RXEN_bm | USART_TXEN_bm);
    // disable interrupt
    _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
;
}

```

```
uint8_t HardwareSerial::available(void)
{
    return (RX_BUFFER_SIZE + _rx_buffer->head - _rx_buffer->tail) % RX_BUFFER_SIZE;
}

int HardwareSerial::read(void)
{
    // if the head isn't ahead of the tail, we don't have any characters
    if (_rx_buffer->head == _rx_buffer->tail) {
        return -1;
    } else {
        unsigned char c = _rx_buffer->buffer[_rx_buffer->tail];
        _rx_buffer->tail = (_rx_buffer->tail + 1) % RX_BUFFER_SIZE;
        return c;
    }
}

void HardwareSerial::flush()
{
    // don't reverse this or there may be problems if the RX interrupt
    // occurs after reading the value of rx_buffer_head but before writing
    // the value to rx_buffer_tail; the previous value of rx_buffer_head
    // may be written to rx_buffer_tail, making it appear as if the buffer
    // were full, not empty.
    _rx_buffer->head = _rx_buffer->tail;
}

void HardwareSerial::write(uint8_t c)
{
    if (_bEn) {
        while ( !(_usart->STATUS & USART_DREIF_bm) );
        _usart->DATA = c;
    }
}

void HardwareSerial::enable(bool bEn)
{
    _bEn = bEn;
}


```

6.23 HardwareSerial.h File Reference

```
#include <avr/io.h>
#include <inttypes.h>
#include "Print.h"
```

Classes

- class HardwareSerial

  HardwareSerial implementation.

6.24 HardwareSerial.h

```
#ifndef HardwareSerial_h
#define HardwareSerial_h

#include <avr/io.h>
#include <inttypes.h>

#include "Print.h"

struct ring_buffer;


class HardwareSerial : public Print
{
  protected:
    ring_buffer *_rx_buffer;
    USART_t     *_usart;
    PORT_t      *_port;
    uint8_t     _in_bm;
    uint8_t     _out_bm;
    uint8_t     _bsel;
    int8_t     _bscale;
    long        _baudrate;
    bool        _bEn;
  public:
    HardwareSerial(
        USART_t     *usart,
        PORT_t      *port,
        uint8_t     in_bm,
        uint8_t     out_bm);
    ~HardwareSerial();
    void begin(long baudrate,int8_t bscale = 0);
    void begin2x(long baudrate,int8_t bscale = 0);
    void end();
    uint8_t available(void);
    int read(void);
    void flush(void);
    virtual void write(uint8_t);
    using Print::write; // pull in write(str) and write(buf, size) from Print
    void rxc();
    void dre();
    void txc();
    void enable(bool bEn);
};


#endif
```

6.25 I2C_Master.h File Reference

```
#include <stdlib.h>
#include <inttypes.h>
#include <avr/io.h>
#include "I2CTransaction.h"
#include "I2CTransaction.h"
```

Classes

- class I2CNotify
- class I2C_Master

6.26 I2C_Master.h

```

#ifndef I2C_Master_h
#define I2C_Master_h

#include <stdlib.h>
#include <inttypes.h>
#include <avr/io.h>

#include "I2CTransaction.h"

// Derive a class from this class
// in order to give the class the ability
// to be notified when a write or read is done.
class I2CNotify
{
public:
    virtual void I2CWriteDone() = 0;
    virtual void I2CReadDone() = 0;
    virtual void I2CBusError() = 0;
    virtual void I2CArbLost() = 0;
    virtual void I2CNack() = 0;
};


class I2C_Master
{
public:
    typedef enum {
        sIdle,
        sBusy,
        sError,
        sArb,
        sIDScan,
        sIDCheck
    } DriverState;

    typedef enum {
        rOk,
        rFail,
        rArbLost,
        rBussErr,
        rNack,
        rBufferOverrun,
        rUnknown,
        rTimeout
    } DriverResult;

private:
    TWI_t*          _twi;
    PORT_t*         _twiPort;
    bool            _bEnabled;
    DriverState     _State;
    DriverResult    _Result;
    void*           _pReserved;
    I2CNotify*      _pNotifyClient;

    // Transaction Data
    uint8_t         _DeviceID;
    uint8_t         _nBytesWritten;
    uint8_t         _nWriteBytes;
    uint8_t         _nReadBytes;
    uint8_t         _nBytesRead;

    uint8_t*        _WriteData;
    uint8_t         _wrBufferLen;
    uint8_t*        _ReadData;
    uint8_t         _rdBufferLen;

    //I2CTransaction* _pTransaction;

    // For ID Scanning
    uint8_t         _idScanCurrent;
    uint8_t         _IDList[128];
    bool            _ScanComplete;

public:

    typedef enum ErrorType {
        eNone           = 0,
        eDisabled       = -1,
        eBusy           = -2,
        eNack           = -3,
        eArbLost        = -4,
        eBusErr         = -5,
        eTimeout        = -6,
        eSDAStuck       = -7,
        eSCLStuck       = -8,
        eUnknown        = -9
    } ErrorType;

    I2C_Master(TWI_t*  twi);
    ~I2C_Master();

    //
    void begin(uint32_t freq);
    void end();

    // Perform a write. Return status indicates result
    // < 0 indicates an error: -1 == NACK -1 = Some other error???
    // > 0 indicates # of bytes written with valid ACK.
    ErrorType Write(uint8_t ID, uint8_t* Data, uint8_t nBytes);
    ErrorType WriteSync(uint8_t ID, uint8_t* Data, uint8_t nBytes);

    // Perform a read. Return status indicates result.
    // maxcnt indidcates how many bytes to attempt to read. Will read until
    // a NACK occurs or maxcnt bytes are read.
    // RETC < 0: -1 -> NACK of ID.
    // RETC > 0: number of bytes read (before NACK from Slave
    ErrorType Read(uint8_t ID, uint8_t nBytes);
    ErrorType ReadSync(uint8_t ID, uint8_t nBytes);

    ErrorType WriteRead(uint8_t ID,
                  uint8_t* wrData,
                  uint8_t nWriteBytes,
                  uint8_t nReadBytes);
    ErrorType WriteReadSync(uint8_t ID,
                  uint8_t* wrData,
                  uint8_t nWriteBytes,
                  uint8_t nReadBytes);


    void master_int();
    void slave_int();

    void WriteHandler();
    void ReadHandler();
    void ArbHandler();
    void ErrorHandler();

    void MasterFinished();
    int testack(uint8_t ID);
    void dumpregs();

    I2C_Master::DriverResult Result();
    I2C_Master::DriverState State();
    uint8_t ReadData(uint8_t* pData, uint8_t maxcnt);
    uint8_t ReadData(uint8_t index);
    uint8_t nReadBytes();

    ErrorType CheckID(uint8_t ID);
    void Stop();      // Use I2C Master for stop.
    ErrorType ForceStartStop();
    ErrorType WigglePin(uint8_t cnt, uint8_t pinSel, uint8_t otherState);
    void CleanRegs();

    void loop(); // Called periodically.

    bool busy(); // In the process of transacting...
    void* isReserved();
    bool Reserve(void*);
    void NotifyMe(I2CNotify* pMe);

    inline bool IsIdle()
    {
        return (_twi->MASTER.STATUS & TWI_MASTER_BUSSTATE_gm)
            == TWI_MASTER_BUSSTATE_IDLE_gc;
    }

protected:
    uint8_t busState();
    void showstate();


};


#endif

```

6.27 IMU.cpp File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <inttypes.h>
#include <util/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "HardwareSerial.h"
#include "IMUPacketFifo.h"
#include "TimerCore.h"
#include "IMU.h"
#include "Mark.h"
```

Variables

- HardwareSerial ∗ pdbgserial
- static char buffer [128]

6.27.1 Variable Documentation

6.27.1.1 char buffer[128] `[static]`

Definition at line 16 of file IMU.cpp.

Referenced by IMU::CheckIDs(), FifoTest(), main(), IMU::ReadWord(), and IMU::WrAsync().

6.27.1.2 HardwareSerial∗ pdbgserial

Definition at line 31 of file GyroAcc.cpp.

6.28 IMU.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include <util/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "HardwareSerial.h"
#include "IMUPacketFifo.h"
#include "TimerCore.h"

#include "IMU.h"
#include "Mark.h"

```

```
extern HardwareSerial* pdbgserial;
static char buffer[128];

IMU::IMU(I2C_Master* pMas)
{
    _pNextIMU       = 0;
    _gID[0]         = 0;
    _aID[0]         = 0;
    _gID[1]         = 0;
    _aID[1]         = 0;
    _bDualChan      = false;
    _numChans       = 0;
    _pMas           = pMas;
    _DLPF           = 0x1;
    _FullScale      = 0x1;
    _ClkSel         = 0x1;
    _Rate           = 10;
    _State          = sIdle;
    _previousState  = sIdle;
    _failType       = fNone;
    _bRun           = false;
    _busyWaitTime   = 0;
    _bDataReady[0]  = false;
    _bDataReady[1]  = false;

    ResetFailStats();
    _pDBGPort       = 0;
    _pDBGPort2      = 0;
    _pNextIMU       = 0;
    _pTimer         = 0;
    _bUseGyro       = false;
    _pMas->NotifyMe(this);
    QueryChannels();
}

IMU::IMU(I2C_Master* pMas, uint8_t gID, uint8_t aID)
{
    _pNextIMU   = 0;
    _gID[0]     = gID;
    _aID[0]     = aID;
    _bDualChan  = false;
    _numChans   = 1;
    _pMas       = pMas;
    _DLPF       = 0x1;
    _FullScale  = 0x1;
    _ClkSel     = 0x1;
    _Rate       = 10;
    _State      = sIdle;
    _bRun       = false;
    _pMas->NotifyMe(this);
    _pDBGPort   = 0;
    _pDBGPort2  = 0;
    _pTimer     = 0;
    ResetFailStats();
    _bUseGyro   = false;
}

IMU::IMU(I2C_Master* pMas,
        uint8_t gID, uint8_t aID,
        uint8_t gID2, uint8_t aID2
    )
{
    _pNextIMU   = 0;
    _gID[0]     = gID;
    _aID[0]     = aID;
    _gID[1]     = gID2;
    _aID[1]     = aID2;
    _bDualChan  = true;
    _numChans   = 2;
    _pMas       = pMas;
    _DLPF       = 0x1;
    _FullScale  = 0x1;
    _ClkSel     = 0x1;
    _Rate       = 10;
    _State      = sIdle;
    _bRun       = false;
    _pMas->NotifyMe(this);
    _pDBGPort   = 0;
    _pDBGPort2  = 0;
    _pTimer     = 0;
    ResetFailStats();
    _bUseGyro   = false;
}

void IMU::NextIMU(IMUBase* pNext)
{
    _pNextIMU = pNext;
}

int IMU::BeginRead()
{
    if (_State == sWait) {
        Run();
    } else {
        if (_pNextIMU) {
            return _pNextIMU->BeginRead();
        }
    }
    return 0;
}

void IMU::QueryChannels()
{
    _numChans = 0;
    _bDualChan = 0;
    // Check the first, lower ID.
    int retc = _pMas->CheckID(0xD2);
    if (retc == 0) {
        _gID[_numChans] = 0xD2;
        _aID[_numChans] = 0x32; // Always high bit.
        _numChans++;
    }
    retc = _pMas->CheckID(0xD0);
    if (retc == 0) {
        _gID[_numChans] = 0xD0;
        _aID[_numChans] = 0x30; // Always low bit.
        _numChans++;
    }

    if (_numChans > 1) {
        _bDualChan = true;
    }
}

```

```
void IMU::SetDebugPort(DebugPort* pPort)
{
    _pDBGPort = pPort;
}

void IMU::SetDebugPort2(DebugPort* pPort)
{
    _pDBGPort2 = pPort;
}

void IMU::Reset()
{
    _pMas->end();
    _pMas->begin(400e3);
    _pMas->NotifyMe(this);
}

void IMU::ResetDevices()
{
    _pMas->Stop();
}

int IMU::ForceStartStop()
{
    return _pMas->ForceStartStop();
}

void IMU::SampleRate(uint16_t rate)
{
    // Range Limit the rate.
    if (rate < 10) {
        _Rate = 10;
    } else if (rate > 200) {
        _Rate = 200;
    } else {
        _Rate = rate;
    }

    uint16_t gval = 1000/rate;
    gval = gval - 1;

    if (pdbgserial) pdbgserial->print("Set Rate on IMU\n");

    SetTimerPeriod();
}

int IMU::Setup()
{
    if (_numChans == 0) return 0;

    _bRun = false;
    Mark marker(_pDBGPort2,pSetup);

    if (_State != sIdle) {
        if (pdbgserial)
            pdbgserial->print("IMU Already Running.\n");
        return 0; // Already running
    }

    ResetFailStats(); // inline in header
    SetState(sConfigure); // Inline in header

    // Start the process with Configure
    for (int x=0; x<_numChans;x++) {
        if (pdbgserial)
            pdbgserial->print("Configuring IMU\n");
        int retc = Configure(x);
        if (retc < 0 ) {
            // Try reset mechanisms - none of which have
            // been found that work properly yet.
            SetState(sIdle);
            if (pdbgserial)
                pdbgserial->print("IMU Configure Failed.");
            return retc;
        }
    }
    SetState(sConfigured);
    if (pdbgserial)
        pdbgserial->print("IMU Configured.\n");

    return 0;
}

int IMU::Start()
{
    if (_numChans == 0) return 0;

    Mark marker(_pDBGPort2,pWriteDn);

    int retc;
    if (_State != sConfigured) {
        retc = Setup();
        if (retc < 0) {
            return retc;
        }
    }

    cli();
    ResetTimer();
    SetState(sWait);
    ResetFailStats();
    _bDataReady[0] = false;
    _bDataReady[1] = false;
    _bRun = true;
    sei();
    return 0;
}

int IMU::Stop()
{
    cli();
    _bRun = false;
    SetState(sIdle);
    sei();
    return 0;
}

bool IMU::Busy()
{
    return _State != sIdle;
}

void IMU::Run()
{
    if (_bRun == false) return;

    Mark marker(_pDBGPort2,pRun);

    if (_pMas->busy() && BusyTimeout()) {
        Reset();
        SetState(sWait);
    }

    switch(_State) {
    case sWait:
        if (_bUseGyro) {
            SetState(sReadGyro1);
        } else {
            SetState(sReadAcc1);
        }
        StartTransaction();
        break;
    default:
        break;
    }
}

int IMU::StartTransaction()
{
    int retc = 0;
    switch(_State) {
    case sReadGyro1:
        RdAsync(_gID[0], 0x1B, 8);
        break;
    case sReadAcc1:
        RdAsync(_aID[0], 0x80 | 0x28, 6);
        break;
    case sReadGyro2:
        RdAsync(_gID[1], 0x1B, 8);
        break;
    case sReadAcc2:
        RdAsync(_aID[1], 0x80 | 0x28, 6);
        break;
    default:
        break;
    }
    ResetBusyTime();
    return retc;
}

void IMU::ProcessTransaction()
{
    switch(_State) {
        case sReadGyro1:
            StoreGyroData(1);
            SetState(sReadAcc1);
            break;
        case sReadAcc1:
            StoreAccData(1);
            PushData(1);
            if (_bDualChan) {
                if (_bUseGyro) {
                    SetState(sReadGyro2);
                } else {
                    SetState(sReadAcc2);
                }
            } else {
                SetState(sWait);
                if (_pNextIMU) {
                    _pNextIMU->BeginRead();
                }
            }
            break;
        case sReadGyro2:
            StoreGyroData(2);
            SetState(sReadAcc2);
            break;
        case sReadAcc2:
            StoreAccData(2);
            PushData(2);
            SetState(sWait);
            if (_pNextIMU) {
                _pNextIMU->BeginRead();
            }
            break;
        default:
            break;
    }

    StartTransaction();
}

void IMU::FailRecovery()
{
    switch(_failType) {
    case fNone:
        break;
    case fNack:
        if (_nackCount <7) {
            _pMas->WigglePin(10, 0,1);
        } else if (_nackCount < 10) {
            Reset();
            _pMas->WigglePin(10,0,1);
        }
        SetState(_previousState);
        break;
    case fBusErr:
        if (_failCount < 5) {
            Reset();
        }
        SetState(_previousState);
        break;
    case fArbLost:
        if (_failCount < 5) {
            Reset();
        }
        SetState(_previousState);
        break;
    }

    StartTransaction();
}

void IMU::I2CWriteDone()
{
    if (_bRun == false) return;

    Mark marker(_pDBGPort2,pWriteDn);
    ResetBusyTime();

    ResetFailStats();
    ProcessTransaction();
}

void IMU::I2CReadDone()
{
    if (_bRun == false) return;

    Mark marker(_pDBGPort2,pReadDn);
    ResetBusyTime();

    ResetFailStats();
    ProcessTransaction();
}

void IMU::I2CNack()
{
    if (_bRun == false) return;

    ResetBusyTime();
    {
        Mark marker(_pDBGPort2,pBusErr);
        _delay_us(3);
    }
    Mark marker(_pDBGPort2,pNack);

    ++_nackCount;

    SetState(sErrRecover);
    _delay_us(5);

    if (_nackCount < 5) {
        SetState(_previousState);
        StartTransaction();
    } else if (_nackCount < 10) {
        _failType = fNack;
        FailRecovery();
    } else {
        Stop();
    }
}

void IMU::I2CBusError()
{
    if (_bRun == false) return;

    ResetBusyTime();
    {
        Mark marker(_pDBGPort2,pBusErr);
        _delay_us(3);
    }
    Mark marker(_pDBGPort2,pBusErr);

    _bFailDetected = true;
    ++_failCount;
    _failType = fBusErr;

    SetState(sErrRecover);
    _delay_us(5);
    if (_failCount > 10) {
        FailRecovery();
    } else {
        SetState(_previousState);
        StartTransaction();
    }
}

void IMU::I2CArbLost()
{
    if (_bRun == false) return;

    ResetBusyTime();
    {
        Mark marker(_pDBGPort2,pBusErr);
        _delay_us(3);
    }
    Mark marker(_pDBGPort2,pArbLost);
    _bFailDetected = true;
    ++_failCount;
    _failType = fArbLost;

    SetState(sErrRecover);
    _delay_us(5);
    if (_failCount > 10) {
        FailRecovery();
    } else {
        SetState(_previousState);
        StartTransaction();
    }
}

bool IMU::DataReady()
{
    if (_numChans == 0) return true;

    bool bReady = false;
    cli();
    if (_bDualChan) {
        bReady =  _bDataReady[0] && _bDataReady[1];
    } else {
        bReady =  _bDataReady[0];
    }
    sei();
    return bReady;
}

void IMU::StoreGyroData(uint8_t idx)
{
    _pMas->ReadData(&_dataBuffer[idx-1][0],8);
}

void IMU::StoreAccData(uint8_t idx)
{
    _pMas->ReadData(&_dataBuffer[idx-1][8],6);
}

void IMU::PushData(uint8_t idx)
{
    _bDataReady[idx-1] = true;
}

uint8_t* IMU::GetPacketData(uint8_t* pData)
```

```
{
    if (_numChans == 0) return pData;

    cli();

    *pData++ = 0xa5;
    *pData++ = 0x5a;
    if (_State == sIdle) {
        if (_failType == fNack) {
            memset(pData,'N',IMUPacket::PacketLen);
        } else {
            memset(pData,'I',IMUPacket::PacketLen);
        }
        pData += IMUPacket::PacketLen;
    } else if (_bDataReady[0] == true) {
        memcpy(pData,&_dataBuffer[0][0],IMUPacket::PacketLen);
        _bDataReady[0] = false;
        pData += IMUPacket::PacketLen;
    } else {
        memset(pData,0,IMUPacket::PacketLen);
        pData += IMUPacket::PacketLen;
    }

    if (_bDualChan) {
        *pData++ = 0xa5;
        *pData++ = 0x5a;
        if (_State == sIdle) {
            if (_failType == fNack) {
                memset(pData,'N',IMUPacket::PacketLen);
            } else {
                memset(pData,'I',IMUPacket::PacketLen);
            }
            pData += IMUPacket::PacketLen;
        } else if (_bDataReady[1] == true) {
            memcpy(pData,&_dataBuffer[1][0],IMUPacket::PacketLen);
            _bDataReady[1] = false;
            pData += IMUPacket::PacketLen;
        } else {
            memset(pData,0,IMUPacket::PacketLen);
            pData += IMUPacket::PacketLen;
        }
    }
    sei();
    return pData;
}

// Diagnostic Routines
void IMU::CheckIDs(HardwareSerial* pSerial)
{
    char buffer[50];
    for (int x=0;x<_numChans;x++) {
        int retc = _pMas->CheckID(_gID[x]);
        if (retc == 0) {
            sprintf(buffer,"Gyro%d (0x%x):Ack.\n",x,_gID[x]);
            pSerial->print(buffer);
        } else {
            sprintf(buffer,"Gyro%d (0x%x):NAck (%d).\n",x,_gID[x],retc);
            pSerial->print(buffer);
        }
        Wr(_gID[x], 0x3D, 0x8);
        retc = _pMas->CheckID(_aID[x]);
        if (retc == 0) {
            sprintf(buffer,"Acc%d (0x%x):Ack.\n",x,_aID[x]);
            pSerial->print(buffer);
        } else {
            sprintf(buffer,"Acc%d (0x%x):NAck (%d).\n",x,_aID[x],retc);
            pSerial->print(buffer);
        }
    }
}

void IMU::SetTimer(TimerCntr* pTimer)
{
    _pTimer = pTimer;

    _pTimer->ClkSel(TC_CLKSEL_DIV64_gc);
    SetTimerPeriod();
    _pTimer->CCEnable(0);
    _pTimer->WaveformGenMode(TC_WGMODE_NORMAL_gc);
    _pTimer->EventSetup(TC_EVACT_OFF_gc,TC_EVSEL_OFF_gc);
    _pTimer->IntLvlA(0,1);
    _pTimer->IntLvlB(0);
    _pTimer->Notify(this,0);
}

void IMU::ResetTimer()
{
    if (_pTimer) _pTimer->Counter(0);
}

void IMU::SetTimerPeriod()
{
    // Adjust the timer function to fire 5X faster
    // than the rate. At 200Hz, this will be 2Khz or
    // every 500us.
    // We set the timer to go off 5 times per IMU period.
    // This should range from 20ms for 10Hz, and 1 ms for 200
    // **** NoFifo
    // Set timer to fire at the rate.
    //unsigned long timerTicks = 100000/_Rate;
    unsigned long timerTicks = 500000/_Rate;
    if (timerTicks > 65000) {
        timerTicks = 65000;
    }
    if (_pTimer) _pTimer->Period(timerTicks);
}


void IMU::err(uint8_t id)
{
}

void IMU::ovf(uint8_t id)
{
    Run();
}

void IMU::ccx(uint8_t id,uint8_t idx)
{
}


int IMU::Wr(uint8_t ID, uint8_t addr, uint8_t data)
{
    static uint8_t  bytes[2];
    bytes[0] = addr;
    bytes[1] = data;
    return _pMas->WriteSync(ID, &bytes[0],2);
}

int IMU::Rd(uint8_t ID, uint8_t addr, uint8_t cnt, uint8_t* pData)
{
    // Only a single write, the address, then read data.
    int retc = _pMas->WriteReadSync(ID, &addr, 1, cnt);
    if ( retc < 0 ) {
        return retc;
    }
    return _pMas->ReadData(pData,cnt);
}

int IMU::WrAsync(uint8_t ID, uint8_t addr, uint8_t data)
{
    static uint8_t  bytes[2];
    bytes[0] = addr;
    bytes[1] = data;
    sprintf(buffer,"WrAsync to %d\n",ID);
    if (pdbgserial) pdbgserial->print(buffer);
    return _pMas->WriteRead(ID, &bytes[0],2,0);
}

int IMU::RdAsync(uint8_t ID, uint8_t addr, uint8_t cnt)
{
    // Only a single write, the address, then read data.
    return _pMas->WriteRead(ID, &addr, 1, cnt);
}

void IMU::ReadWord(uint16_t* pData)
{
    static uint8_t buffer[2];
    _pMas->ReadData(&buffer[0],2);
    *pData = (buffer[0] << 8 | buffer[1]);
}

int IMU::Configure(uint8_t idx)
{
    // Value for the sensor register
    uint16_t gval = 1000/_Rate;
    gval = gval - 1;

    int retc;

    RegWriteType    config[] = {
    //    // Turn on pass-through
        { _gID[idx], 0x3D, 0x0F },
    //
    //    // Init the Accelerometer.
        { _aID[idx], 0x20, 0x37},
        { _aID[idx], 0x21, 0x0},
        { _aID[idx], 0x22, 0x0},
        { _aID[idx], 0x23, 0x80 | 0x40},
        { _aID[idx], 0x24, 0x00},
    //
    //    // Set offsets to zero
        { _gID[idx], 0x0C, 0x00},
        { _gID[idx], 0x0D, 0x00},
        { _gID[idx], 0x0E, 0x00},
        { _gID[idx], 0x0F, 0x00},
        { _gID[idx], 0x10, 0x00},
        { _gID[idx], 0x11, 0x00},
    //
    //    // Configure registers.
        { _gID[idx], 0x12, 0xff},                     // Enable all outputs to to the fifo
        { _gID[idx], 0x13, 0x00},
    //    { _gID[idx], 0x14, _aID[idx] >> 1},             // Set slave address of ACC
        { _gID[idx], 0x15, gval},                     // Set sample rate
        { _gID[idx], 0x16, _DLPF | _FullScale << 3},
        { _gID[idx], 0x17, 0x00},
    //    { _gID[idx], 0x18, 0x80 | 0x28},              // Set burst address for Accelerometer, enable auto addr increment.
        { _gID[idx], 0x3E, _ClkSel},
    };

    uint8_t nItems = sizeof(config)/sizeof(RegWriteType);
    for (int idx = 0;idx <nItems;idx++) {
        retc = Wr(config[idx].ID,
            config[idx].Addr,
            config[idx].Data);
        ResetBusyTime();

        if (retc < 0) {
            return retc; // _configOkay will be false;
        }
    }

    return 0;
}



```

6.29 IMU.h File Reference

```
#include <stdlib.h>
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>
#include "I2C_Master.h"
#include "HardwareSerial.h"
#include "IMUPacketFifo.h"
#include "TimerCntr.h"
#include "TimerCore.h"
#include "DebugPort.h"
```

Classes

- class IMUBase
- class IMU
- struct IMU::regWrite

6.30 IMU.h

```

#ifndef IMU_h
#define IMU_h

#include <stdlib.h>
#include <inttypes.h>
#include <avr/io.h>
#include <util/delay.h>
#include "I2C_Master.h"
#include "HardwareSerial.h"
#include "IMUPacketFifo.h"
#include "TimerCntr.h"
#include "TimerCore.h"
#include "DebugPort.h"

class IMUBase
{
public:
    virtual void Reset() = 0;
    virtual void SampleRate(uint16_t) = 0;

    virtual int Setup() = 0;
    virtual int Start() = 0;
    virtual int Stop() = 0;
    virtual int ForceStartStop() = 0;
    virtual bool Busy() = 0;
    virtual void ResetTimer() = 0;
    virtual void UseGyro(bool bEnable) = 0;
    virtual void NextIMU(IMUBase*) = 0;
    virtual int BeginRead() = 0;

    virtual bool DataReady() = 0;
    virtual uint8_t* GetPacketData(uint8_t* pPacket) = 0;

    // Diags
    virtual void CheckIDs(HardwareSerial* pSerial) = 0;
    virtual void ResetDevices() = 0;
};

class IMU : public IMUBase, public I2CNotify, public TimerNotify
{
    typedef enum StateType {
        sIdle           = 0,
        sConfigure      = 1,
        sConfigured     = 2,
        sWait           = 5,
        sReadGyro1      = 8,
        sReadAcc1       = 9,
        sReadGyro2      = 10,
        sReadAcc2       = 11,
        sErrRecover     = 12
    } StateType;

    typedef enum PosType {
        pStart          = 0,
        pRun            = 1,
        pWriteDn        = 2,
        pReadDn         = 3,
        pNack           = 4,
        pBusErr         = 5,
        pArbLost        = 6,
        pSetup          = 7
    } PosType;

    typedef enum FailType {
        fNone           = 0,
        fNack           = 1,
        fBusErr         = 2,
        fArbLost        = 3
    } FailType;

    typedef struct regWrite {
        uint8_t     ID;
        uint8_t     Addr;
        uint8_t     Data;
    } RegWriteType;

    typedef enum  {
        ptTimer,
        ptI2CWrite,
        ptI2CRead,
        ptI2CNack
    } ProcessType;

    StateType       _State;
    StateType       _previousState; // Used for error recovery.
    FailType        _failType;

    I2C_Master*     _pMas;
    bool            _bDualChan;
    uint8_t         _numChans;
    bool            _configOkay[2];
    uint8_t         _gID[2];
    uint8_t         _aID[2];
    uint8_t         _DLPF;
    uint8_t         _FullScale;
    uint8_t         _ClkSel;
    uint16_t        _Rate;

    bool            _bUseGyro;

    uint8_t         _dataBuffer[2][20];
    bool            _bDataReady[2];

    TimerCntr*      _pTimer;
    bool            _bRun;
    uint16_t        _failCount;
    uint8_t         _nackCount;
    bool            _bFailDetected;
    unsigned int    _busyWaitTime;
    DebugPort*      _pDBGPort;
    DebugPort*      _pDBGPort2;
    IMUBase*        _pNextIMU;

```

```
public:
    IMU(I2C_Master* pMas);
    IMU(I2C_Master* pMas, uint8_t gID, uint8_t aID);
    IMU(I2C_Master* pMas, uint8_t gID, uint8_t aID,
        uint8_t gID2, uint8_t aID2
        );


    void QueryChannels();
    void SetDebugPort(DebugPort* pPort);
    void SetDebugPort2(DebugPort* pPort);

    virtual void Reset();
    virtual void SampleRate(uint16_t);

    virtual int Setup();
    virtual int Start();
    virtual int Stop();
    virtual int ForceStartStop();
    virtual bool Busy();
    virtual void ResetTimer();
    virtual void UseGyro(bool bEnable) { _bUseGyro = bEnable; };
    virtual void NextIMU(IMUBase* pNext);
    virtual int BeginRead();

    virtual bool DataReady();
    virtual uint8_t* GetPacketData(uint8_t*);

    // Diags
    virtual void CheckIDs(HardwareSerial* pSerial);
    virtual void ResetDevices();

    // Timer Notification
    void SetTimer(TimerCntr* pTimer);
    void SetTimerPeriod();
    virtual void err(uint8_t id);
    virtual void ovf(uint8_t id);
    virtual void ccx(uint8_t id, uint8_t idx);

    // I2C
    virtual void I2CWriteDone();
    virtual void I2CReadDone();
    virtual void I2CBusError();
    virtual void I2CArbLost();
    virtual void I2CNack();

    void FailRecovery();

protected:
    void Run();
    int StartTransaction();
    void ProcessTransaction();
    int Configure(uint8_t idx);
    int Wr(uint8_t ID, uint8_t addr, uint8_t data);
    int Rd(uint8_t ID, uint8_t addr, uint8_t cnt, uint8_t* pData);
    int WrAsync(uint8_t ID, uint8_t addr, uint8_t data);
    int RdAsync(uint8_t ID, uint8_t addr, uint8_t cnt);
    void ReadWord(uint16_t *pData);
    void StoreGyroData(uint8_t idx);
    void StoreAccData(uint8_t idx);
    void PushData(uint8_t idx);

    inline void SetState(StateType s)
    {
        _previousState = _State;
        _State = s;
        if (_pDBGPort) _pDBGPort->SetState((uint8_t)_State);
    }

    inline void MarkPos(PosType p)
    {
        if (_pDBGPort2) _pDBGPort2->SetState((uint8_t) p);
    }

    inline void ResetBusyTime()
    {
        _busyWaitTime = millis();
    }

    inline bool BusyTimeout()
    {
        return ((millis() - _busyWaitTime) > 2);
    }

    inline void ResetFailStats()
    {
        _bFailDetected      = false;
        _nackCount          = 0;
        _failCount          = 0;
        _failType           = fNone;
    }

};


#endif

```

6.31 IMUManager.cpp File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
#include <util/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "HardwareSerial.h"
#include "TimerCore.h"
#include "TimerCntr.h"
#include "IMU.h"
#include "IMUManager.h"
```

Defines

- #define ALL_IMU(func)
- #define ALL_IMUP(func, param)
- #define ALL_IMURET(func)
- #define ALL_IMUBOOL(func)

Variables

- HardwareSerial ∗ pdbgserial
- static uint8_t buffer [255]

6.31.1 Define Documentation

6.31.1.1 #define ALL_IMU(*func*)

Value:

```
for (int x=0;x<4;x++) { \
        if (_pIMU[x]) {          \
            _pIMU[x]->func();    \
        }    \
    }
```

Definition at line 18 of file IMUManager.cpp.

6.31.1.2 #define ALL_IMUBOOL(*func*)

Value:

```
bool bReady = true; \
    for (int x = 0;x<4;x++) {    \
        if (_pIMU[x]) { \
            if (!_pIMU[x]->func()) { \
                bReady = false; \
            }    \
        }    \
    }    \
    return bReady;
```

Definition at line 44 of file IMUManager.cpp.

6.31.1.3 #define ALL_IMUP(*func,* *param*)

Value:

```
for (int x=0;x<4;x++) { \
        if (_pIMU[x]) {          \
            _pIMU[x]->func(param);    \
        }    \
    }
```

Definition at line 25 of file IMUManager.cpp.

6.31.1.4 #define ALL_IMURET(*func*)

Value:

```
int retc =0; \
    for (int x=0;x<4;x++) { \
        if (_pIMU[x]) {          \
            retc = _pIMU[x]->func();    \
            if (retc < 0) { \
                return retc;\
            }\
        }    \
    }\
    return retc;
```

Definition at line 32 of file IMUManager.cpp.

6.31.2 Variable Documentation

6.31.2.1 uint8_t buffer[255] `[static]`

Definition at line 16 of file IMUManager.cpp.

6.31.2.2 HardwareSerial∗ pdbgserial

Definition at line 31 of file GyroAcc.cpp.

6.32 IMUManager.cpp

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <inttypes.h>
00005 #include <util/delay.h>
00006 #include <avr/io.h>
00007 #include <avr/interrupt.h>
00008 #include "HardwareSerial.h"
00009 #include "TimerCore.h"
00010 #include "TimerCntr.h"
00011
00012 #include "IMU.h"
00013 #include "IMUManager.h"
00014
00015 extern HardwareSerial* pdbgserial;
00016 static uint8_t  buffer[255];
00017
00018 #define ALL_IMU(func) \
    for (int x=0;x<4;x++) { \
        if (_pIMU[x]) {         \
            _pIMU[x]->func();   \
        }   \
    }

#define ALL_IMUP(func,param) \
    for (int x=0;x<4;x++) { \
        if (_pIMU[x]) {         \
            _pIMU[x]->func(param);   \
        }   \
    }

#define ALL_IMURET(func) \
    int retc =0; \
    for (int x=0;x<4;x++) { \
        if (_pIMU[x]) {         \
            retc = _pIMU[x]->func();   \
            if (retc < 0) { \
                return retc;\
            }\
        }   \
    }\
    return retc;

#define ALL_IMUBOOL(func) \
    bool bReady = true; \
    for (int x = 0;x<4;x++) {   \
        if (_pIMU[x]) { \
            if (!_pIMU[x]->func()) { \
                bReady = false; \
            }   \
        }   \
    }   \
    return bReady;

IMUManager::IMUManager(HardwareSerial* pSerial)
{
    int x;
    for (x =0;x<4;x++) {
        _pIMU[x] = 0;
    }

    _nIMUs          = 0;
    _State          = sIdle;
    _nStreamWDCounter = 0;
    _packetId       = 0;
    _pBlueLed       = 0;
    _bLedState      = false;

    _pTimer         = 0;
    _pSerial        = pSerial;
    _pDBGPort       = 0;
    _sampleRate     = 10;
    _maxMillisPerPacket = 1000;
    _lastSendMillis = 0;
}

void IMUManager::SetBlueLed(PORT_t* port, uint8_t Pin)
{
    _pBlueLed   = port;
    _LedPin     = Pin;

    _pBlueLed->DIRSET = _LedPin;
    _pBlueLed->OUTSET = _LedPin;
    _bLedState        = false;
}

void IMUManager::LedOn()
{
    _pBlueLed->OUTCLR   = _LedPin;
    _bLedState          = true;
}

void IMUManager::LedOff()
{
    _pBlueLed->OUTSET   = _LedPin;
    _bLedState          = false;
}

void IMUManager::PacketLedIndicator()
{
    if ((_packetId % 6) == 0) {
        ToggleLed();
    }
}

void IMUManager::ToggleLed()
{
    if ( _bLedState) {
        _bLedState = false;
        _pBlueLed->OUTCLR = _LedPin;
    } else {
        _bLedState = true;
        _pBlueLed->OUTSET = _LedPin;
    }
}

void IMUManager::ShowLedStart()
{
    _pBlueLed->OUTCLR = _LedPin;
    _delay_ms(10*17); // One Frame
    _pBlueLed->OUTSET = _LedPin;
    _delay_ms(5*17); // One Frame
    _pBlueLed->OUTCLR = _LedPin;
    _delay_ms(10*17); // One Frame
    _pBlueLed->OUTSET = _LedPin;
    _delay_ms(17); // One Frame
    _bLedState        = false;
}

void IMUManager::ShowLedStop()
{
    _pBlueLed->OUTCLR = _LedPin;
    _delay_ms(5*17); // One Frame
    _pBlueLed->OUTSET = _LedPin;
    _delay_ms(5*17); // One Frame
    _pBlueLed->OUTCLR = _LedPin;
    _delay_ms(5*17); // One Frame
    _pBlueLed->OUTSET = _LedPin;
    _delay_ms(5*17); // One Frame
    _pBlueLed->OUTCLR = _LedPin;
    _bLedState        = false;
}
```

```

void IMUManager::SampleRate(uint16_t rate)
{
    // Range Limit the rate.
    if (rate < 10) {
        _sampleRate = 10;
    } else if (rate > 200) {
        _sampleRate = 200;
    } else {
        _sampleRate = rate;
    }

    ALL_IMUP(SampleRate,_sampleRate);


    if (pdbgserial) {
        char buffer[50];
        sprintf(buffer,"Sample rate set to %d\n", _sampleRate);
        pdbgserial->print(buffer);
    }

    // Note: I had this set to 2X the sample rate (in milliseconds),
    // but that was a problem. The IMU's wait until there is 2X the
    // amount of data in the fifo, so that takes 2X the time, and we
    // would time out first time around.. bad idea.
    _maxMillisPerPacket = 3 * 1000/_sampleRate;

    SetTimerPeriod();
}

int IMUManager::AddIMU(IMUBase* pIMU)
{
    for (int x=0;x<4;x++) {
        if (_pIMU[x] == 0) {
            // Add into first empty spot
            _pIMU[x] = pIMU;
            if (x > 0) {
                _pIMU[x-1]->NextIMU(pIMU);
            }
            _nIMUs++;
            return x;
        }
    }

    // Seems we are full!!
    return -1;
}

int IMUManager::Setup()
{
    ALL_IMURET(Setup);
}

int IMUManager::Start()
{
    ALL_IMURET(Start);
}

void IMUManager::Stop()
{
    ALL_IMU(Stop);
    _State = sIdle;
    ShowLedStop();
    LedOff();
}

void IMUManager::Reset()
{
    ALL_IMU(Reset);

    _State = sIdle;
    _packetId       = 0;
}

void IMUManager::ResetDevices()
{
    ALL_IMU(ResetDevices);
}

void IMUManager::ForceStartStop()
{
    ALL_IMU(ForceStartStop);
}

// Diagnostic Routines
void IMUManager::CheckIDs(HardwareSerial* pSerial,int idx)
{
    if (idx < 0) {
        ALL_IMUP(CheckIDs,pSerial);
    } else {
        if (_pIMU[idx]) {
            _pIMU[idx]->CheckIDs(pSerial);
        }
    }
}

bool IMUManager::DataReady()
{
    ALL_IMUBOOL(DataReady)
}

void IMUManager::IMUUseGyro(bool bEn)
{
    ALL_IMUP(UseGyro,bEn);
}

int IMUManager::StreamStart(bool bUseGyro)
{
    if (_State != sIdle ) {
        Stop();
    }

    IMUUseGyro(bUseGyro);

    int retc = Setup();
    if (retc < 0) {
        return retc;
    }

    retc = Start();

    if (retc < 0) {
        return retc;
    }

    ShowLedStart();
    _LedCounter = 0;
    _packetId       = 0;
    _nStreamWDCounter = 20;

    SetState(sDataWait); // Jump ahead a state.
    ResetDataReadyTO();
    return 0;
}

void IMUManager::StreamWatchdog()
{
    _nStreamWDCounter = 20;
}

//
//  Iterate over the embedded IMU objects, retrieve results
//  as needed. Do this in a 2-pass fashion, so that we do
//  IMU 1 on each interface first, then IMU2 on any interfaces
//  that have 2 IMU devices on them. After this is all done,
//  we should have all of the required IMU data, then we can
//  initiate a packet send to the host with as much as six
//  IMU's worth of data!
//
int IMUManager::Loop()
{
    switch(_State) {
    case sIdle:
        break;
    case sDataWait:
        if (DataReady()) {
            ResetDataReadyTO();
            _State = sDataReady;
        } else if (DataReadyTimeout()) {
            ResetDataReadyTO();
            _State = sDataTimeout;
        }
        break;
    case sDataReady:
        PacketLedIndicator();
        if (_nStreamWDCounter == 0) {
            DiscardData();
            _State = sDataWait;
        } else {
            --_nStreamWDCounter;
            SendPacket(false);
            _State = sDataWait;
        }
        break;
    case sDataTimeout:
        PacketLedIndicator();
        if (_nStreamWDCounter == 0) {
            DiscardData();
            _State = sDataWait;
        } else {
            --_nStreamWDCounter;
            SendPacket(true);
            _State = sDataWait;
        }
        break;
    }

    return 0;
}

void IMUManager::Run()
{
    if (_State != sIdle) {
        // Start the IMU's going one at a time...
        if (_pIMU[0]) {
            _pIMU[0]->BeginRead();
        }
    }
}

void IMUManager::SendPacket(bool bTimeout)
{
    uint8_t*    pPacket = &_dataPacket[0];
    if (true || !bTimeout) {
        for (int x = 0;x<4;x++) {
            if (_pIMU[x]) {
                // This puts the data at the pointer,
                // then returns the end of the data.
                // This might be 2*14 or 1*14
                pPacket = _pIMU[x]->GetPacketData(pPacket);
            }
        }
    }
    // Packet format:
    // SNP header
    // byte: length of packet
    // byte: packet type (0xB7)
    // byte(s): length bytes
    // bytes(2): 2 byte CRC
    // string: END
    // newline
    uint8_t size = pPacket - &_dataPacket[0];
    buffer[0] = 'S';
    buffer[1] = 'N';
    buffer[2] = 'P';
    buffer[3] = 0xB7;
    buffer[4] = _packetId++;
    buffer[5] = size;
    memcpy(&buffer[6],&_dataPacket[0],size);
    // Compute CRC -- someday
    uint16_t crc = 0xaf5a;
    uint8_t crchi = (crc >> 8) & 0xff;
    uint8_t crclo = crc & 0xff;
    buffer[6+size]   = _nStreamWDCounter;
    buffer[6+size+1]   = crchi;
    buffer[6+size+2] = crclo;
    sprintf((char*)&buffer[6+size+3],"END\n");
    _pSerial->write(&buffer[0],6+size+3+4);
}

void IMUManager::DiscardData()
{
    uint8_t*    pPacket = &_dataPacket[0];
    for (int x = 0;x<4;x++) {
        if (_pIMU[x]) {
            // This puts the data at the pointer,
            // then returns the end of the data.
            // This might be 2*14 or 1*14
            pPacket = _pIMU[x]->GetPacketData(pPacket);
        }
    }
}

void IMUManager::SetTimer(TimerCntr* pTimer)
{
    _pTimer = pTimer;

    _pTimer->ClkSel(TC_CLKSEL_DIV64_gc);
    SetTimerPeriod();
    _pTimer->CCEnable(0);
    _pTimer->WaveformGenMode(TC_WGMODE_NORMAL_gc);
    _pTimer->EventSetup(TC_EVACT_OFF_gc,TC_EVSEL_OFF_gc);
    _pTimer->IntLvlA(0,1);
    _pTimer->IntLvlB(0);
    _pTimer->Notify(this,0);
}

void IMUManager::SetTimerPeriod()
{
    // Adjust the timer function to fire 5X faster
    // than the rate. At 200Hz, this will be 2Khz or
    // every 500us.
    // We set the timer to go off 5 times per IMU period.
    // This should range from 20ms for 10Hz, and 1 ms for 200
    // **** NoFifo
    // Set timer to fire at the rate.
    //unsigned long timerTicks = 100000/_Rate;
    unsigned int timerTicks = 500000/_sampleRate;
    if (timerTicks > 65000) {
        timerTicks = 65000;
    }
    // This is a special case. If I set the rate to 180,
    // then this will assume I am trying to sync with the camera
    // which has a frame rate of 59.94. 3X this is 179.82.
    // Setting this timer ticks value will put our IMU rate
    // close to 3x the frame rate of the camera, which is what
    // we want.
    if (_sampleRate == 180) {
        timerTicks = 2780;
    }
    if (pdbgserial) {
        char buffer[50];
        sprintf(buffer,"Timer Period:%u\n",timerTicks);
        pdbgserial->print(buffer);
    }
    if (_pTimer) _pTimer->Period(timerTicks);
}


void IMUManager::err(uint8_t id)
{
}

void IMUManager::ovf(uint8_t id)
{
    Run();
}

void IMUManager::ccx(uint8_t id,uint8_t idx)
{
```

```
00505 }
00506
00508
```

6.33 NewDel.cpp File Reference

```
#include "newdel.h"
```

Functions

- void ∗ operator new (size_t size)
- void operator delete (void ∗ptr)
- void ∗ operator new[ ] (size_t size)
- void operator delete[ ] (void ∗ptr)

6.33.1 Function Documentation

6.33.1.1 void operator delete (void ∗ *ptr*)

Definition at line 8 of file NewDel.cpp.

```
{
  free(ptr);
}
```

6.33.1.2 void operator delete[ ] (void ∗ *ptr*)

Definition at line 18 of file NewDel.cpp.

```
{
    free(ptr);
}
```

6.33.1.3 void∗ operator new (size_t *size*)

Definition at line 3 of file NewDel.cpp.

```
{
  return malloc(size);
}
```

6.33.1.4 void∗ operator new[] (size_t *size*)

Definition at line 13 of file NewDel.cpp.

```
{
    return malloc(size);
}
```

6.34 NewDel.cpp

```
00001 #include "newdel.h"
00002 
00003 void * operator new(size_t size)
00004 {
00005   return malloc(size);
00006 }
00007 
00008 void operator delete(void * ptr)
00009 {
00010   free(ptr);
00011 }
00012 
00013 void * operator new[](size_t size)
00014 {
00015     return malloc(size);
00016 }
00017 
00018 void operator delete[](void * ptr)
00019 {
00020     free(ptr);
00021 }
```

6.35 NewDel.h File Reference

```
#include <stdlib.h>
#include <stdlib.h>
```

Functions

- void ∗ operator new (size_t size)
- void operator delete (void ∗ptr)

6.35.1 Function Documentation

6.35.1.1 void operator delete (void ∗ *ptr*)

Definition at line 8 of file NewDel.cpp.

```
{
  free(ptr);
}
```

6.35.1.2 void∗ operator new (size_t *size*)

Definition at line 3 of file NewDel.cpp.

```
{
  return malloc(size);
}
```

6.36 NewDel.h

```
#include <stdlib.h>

void * operator new(size_t size);
void operator delete(void * ptr);

```

6.37 Port.cpp File Reference

```
#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>
#include "Port.h"
#include "HardwareSerial.h"
```

Defines

- #define PORT_ISR_DEF(port)

Functions

- PORT_ISR_DEF (PORTA)
- PORT_ISR_DEF (PORTB)
- PORT_ISR_DEF (PORTC)
- PORT_ISR_DEF (PORTD)
- PORT_ISR_DEF (PORTE)
- PORT_ISR_DEF (PORTF)
- static void SetPointer (PORT_t ∗port, Port ∗p)

Variables

- HardwareSerial ∗ pdbgserial

6.37.1 Define Documentation

6.37.1.1 #define PORT_ISR_DEF(*port*)

Value:

```
static Port*  port##cp = 0;\
ISR(port##_INT0_vect) {\
    if (port##cp) port##cp->int0();\
}\
ISR(port##_INT1_vect) {\
    if (port##cp) port##cp->int1();\
}
```

Definition at line 14 of file Port.cpp.

6.37.2 Function Documentation

6.37.2.1 PORT_ISR_DEF (PORTA)

6.37.2.2 PORT_ISR_DEF (PORTB)

6.37.2.3 PORT_ISR_DEF (PORTF)

6.37.2.4 PORT_ISR_DEF (PORTE)

6.37.2.5 PORT_ISR_DEF (PORTD)

6.37.2.6 PORT_ISR_DEF (PORTC)

6.37.2.7 static void SetPointer (PORT_t * *port*, Port * *p*) [static]

Definition at line 33 of file Port.cpp.

Referenced by Port::Port(), and Port::~Port().

```
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if(port == &PORTA) {
        PORTAcp = p;
    } else if (port == &PORTB) {
        PORTBcp = p;
    } else if (port ==  &PORTC) {
        PORTCcp = p;
    } else if (port ==  &PORTD) {
        PORTDcp = p;
    } else if (port ==  &PORTE) {
        PORTEcp = p;
    } else if (port ==  &PORTF) {
        PORTFcp = p;
#if defined (__AVR_ATxmega128A1__)
    } else if (port ==  &PORTH) {
        PORTHcp = p;
#endif
    }
}
```

6.37.3 Variable Documentation

6.37.3.1 HardwareSerial* pdbgserial

Definition at line 31 of file GyroAcc.cpp.

6.38 Port.cpp

```


#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>

#include "Port.h"
#include "HardwareSerial.h"

extern HardwareSerial* pdbgserial;

// Generate all of the ISR handlers.. hook them up to a class if/when a class
// is instantiated for a particular USART.
#define PORT_ISR_DEF(port) \
static Port*  port##cp = 0;\
ISR(port##_INT0_vect) {\
    if (port##cp) port##cp->int0();\
}\
ISR(port##_INT1_vect) {\
    if (port##cp) port##cp->int1();\
}

PORT_ISR_DEF(PORTA);
PORT_ISR_DEF(PORTB);
PORT_ISR_DEF(PORTC);
PORT_ISR_DEF(PORTD);
PORT_ISR_DEF(PORTE);
PORT_ISR_DEF(PORTF);
#if defined (__AVR_ATxmega128A1__)
PORT_ISR_DEF(PORTH);
#endif

static void SetPointer(PORT_t* port,Port* p)
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if(port == &PORTA) {
        PORTAcp = p;
    } else if (port == &PORTB) {
        PORTBcp = p;
    } else if (port ==  &PORTC) {
        PORTCcp = p;
    } else if (port ==  &PORTD) {
        PORTDcp = p;
    } else if (port ==  &PORTE) {
        PORTEcp = p;
    } else if (port ==  &PORTF) {
        PORTFcp = p;
#if defined (__AVR_ATxmega128A1__)
    } else if (port ==  &PORTH) {
        PORTHcp = p;
#endif
    }
}


Port::Port(PORT_t* pPort)
{
    _pPort = pPort;
    SetPointer(_pPort,this);
}

Port::~Port()
{
    SetPointer(_pPort,0);
}

void Port::Notify(PortNotify* pClient,uint8_t id)
{
    _pNotifyClient  = pClient;
    _pNotifyID      = id;
}

void Port::int0()
{
    if (_pNotifyClient) {
        _pNotifyClient->PortISR0(_pNotifyID);
    }
    _pPort->INTFLAGS = 0x1;
}

void Port::int1()
{
    if (_pNotifyClient) {
        _pNotifyClient->PortISR1(_pNotifyID);
    }
    _pPort->INTFLAGS = 0x2;
}

void Port::SetDir(uint8_t dir)
{
    _pPort->DIR = dir;
}

void Port::SetPinsAsInput(uint8_t mask)
{
    _pPort->DIRCLR = mask;
}

void Port::SetPinsAsOutput(uint8_t mask)
{
    _pPort->DIRSET = mask;
}

void Port::SetPinsHigh(uint8_t mask)
{
    _pPort->OUTSET = mask;
}

void Port::SetPinsLow(uint8_t mask)
{
    _pPort->OUTCLR = mask;
}

uint8_t Port::GetPins()
{
    return _pPort->IN;
}

void Port::InterruptLevel(uint8_t num, uint8_t lvl)
{
    if (num == 0) {
        _pPort->INTCTRL &= ~(0x3);
        _pPort->INTCTRL |= (lvl & 0x3);
    } else {
        _pPort->INTCTRL &= ~(0xC);
        _pPort->INTCTRL |= (lvl & 0x3) << 2;
    }
}

void Port::InterruptMask(uint8_t num, uint8_t mask)
{
    if (num == 0) {
        _pPort->INT0MASK = mask;
    } else {
        _pPort->INT1MASK = mask;
    }
}

```

```
void Port::PinControl(uint8_t mask,
        bool bSlewLimit,
        bool bInverted,
        PORT_OPC_t OutputConfig,
        PORT_ISC_t InputSense
        )
{
    PORTCFG.MPCMASK = mask;
    _pPort->PIN0CTRL =
        (bSlewLimit ? 0x80 : 0x0) |
        (bInverted ? 0x40 : 0x0)  |
        OutputConfig |
        InputSense
        ;
}


```

6.39 Port.h File Reference

Classes

- class PortNotify
- class Port

6.40 Port.h

```

#ifndef Port_h
#define Port_h

class PortNotify
{
public:
    virtual void PortISR0(uint8_t id) = 0;
    virtual void PortISR1(uint8_t id) = 0;
};

class Port
{
protected:
    PORT_t*     _pPort;
    PortNotify* _pNotifyClient;
    uint8_t     _pNotifyID;

public:

    Port(PORT_t*);
    ~Port();

    void Notify(PortNotify* pClient,uint8_t id);
    void int0();
    void int1();

    void SetDir(uint8_t dir);
    void SetPinsAsInput(uint8_t mask);
    void SetPinsAsOutput(uint8_t mask);
    void SetPinsHigh(uint8_t mask);
    void SetPinsLow(uint8_t mask);
    uint8_t GetPins();

    void InterruptLevel(uint8_t num, uint8_t lvl);
    void InterruptMask(uint8_t num, uint8_t mask);

    void PinControl(uint8_t mask,
        bool bSlewLimit,
        bool bInverted,
        PORT_OPC_t OutputConfig,
        PORT_ISC_t InputSense
        );
};


#endif
```

6.41 Print.cpp File Reference

`#include <stdio.h>`

`#include <string.h>`

`#include <math.h>`

`#include "Print.h"`

6.42 Print.cpp

```
/*
 Print.cpp - Base class that provides print() and println()
 Copyright (c) 2008 David A. Mellis.  All right reserved.

 This library is free software; you can redistribute it and/or
 modify it under the terms of the GNU Lesser General Public
 License as published by the Free Software Foundation; either
 version 2.1 of the License, or (at your option) any later version.

 This library is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 Lesser General Public License for more details.

 You should have received a copy of the GNU Lesser General Public
 License along with this library; if not, write to the Free Software
 Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA

 Modified 23 November 2006 by David A. Mellis
 */

#include <stdio.h>
#include <string.h>
#include <math.h>
//#include "wiring.h"

#include "Print.h"

// Public Methods //////////////////////////////////////////////////////////////

```

```
/* default implementation: may be overridden */
void Print::write(const char *str)
{
  while (*str)
    write(*str++);
}

/* default implementation: may be overridden */
void Print::write(const uint8_t *buffer, size_t size)
{
  while (size--)
    write(*buffer++);
}

void Print::print(const char str[])
{
  write(str);
}

void Print::print(char c, int base)
{
  print((long) c, base);
}

void Print::print(unsigned char b, int base)
{
  print((unsigned long) b, base);
}

void Print::print(int n, int base)
{
  print((long) n, base);
}

void Print::print(unsigned int n, int base)
{
  print((unsigned long) n, base);
}

void Print::print(long n, int base)
{
  if (base == 0) {
    write(n);
  } else if (base == 10) {
    if (n < 0) {
      print('-');
      n = -n;
    }
    printNumber(n, 10);
  } else {
    printNumber(n, base);
  }
}

void Print::print(unsigned long n, int base)
{
  if (base == 0) write(n);
  else printNumber(n, base);
}

void Print::print(double n, int digits)
{
  printFloat(n, digits);
}

void Print::println(void)
{
  print('\r');
  print('\n');
}

void Print::println(const char c[])
{
  print(c);
  println();
}

void Print::println(char c, int base)
{
  print(c, base);
  println();
}

void Print::println(unsigned char b, int base)
{
  print(b, base);
  println();
}

void Print::println(int n, int base)
{
  print(n, base);
  println();
}

void Print::println(unsigned int n, int base)
{
  print(n, base);
  println();
}

void Print::println(long n, int base)
{
  print(n, base);
  println();
}

void Print::println(unsigned long n, int base)
{
  print(n, base);
  println();
}

void Print::println(double n, int digits)
{
  print(n, digits);
  println();
}

// Private Methods /////////////////////////////////////////////////////////////

void Print::printNumber(unsigned long n, uint8_t base)
{
  unsigned char buf[8 * sizeof(long)]; // Assumes 8-bit chars.
  unsigned long i = 0;

  if (n == 0) {
    print('0');
    return;
  }

  while (n > 0) {
    buf[i++] = n % base;
    n /= base;
  }

  for (; i > 0; i--)
    print((char) (buf[i - 1] < 10 ?
      '0' + buf[i - 1] :
      'A' + buf[i - 1] - 10));
}

void Print::printFloat(double number, uint8_t digits)
{
  // Handle negative numbers
  if (number < 0.0)
  {
     print('-');
     number = -number;
  }

  // Round correctly so that print(1.999, 2) prints as "2.00"
  double rounding = 0.5;
  for (uint8_t i=0; i<digits; ++i)
    rounding /= 10.0;

  number += rounding;

  // Extract the integer part of the number and print it
  unsigned long int_part = (unsigned long)number;
  double remainder = number - (double)int_part;
  print(int_part);

  // Print the decimal point, but only if there are digits beyond
  if (digits > 0)
    print(".");

  // Extract digits from the remainder one at a time
  while (digits-- > 0)
  {
    remainder *= 10.0;
    int toPrint = int(remainder);
    print(toPrint);
    remainder -= toPrint;
  }
}
```

6.43 Print.h File Reference

```
#include <inttypes.h>
#include <stdio.h>
#include <stdio.h>
```

Classes

- class Print

Defines

- #define DEC 10
- #define HEX 16
- #define OCT 8
- #define BIN 2
- #define BYTE 0

6.43.1 Define Documentation

6.43.1.1 #define BIN 2

Definition at line 29 of file Print.h.

6.43.1.2 #define BYTE 0

Definition at line 30 of file Print.h.

6.43.1.3 #define DEC 10

Definition at line 26 of file Print.h.

6.43.1.4 #define HEX 16

Definition at line 27 of file Print.h.

6.43.1.5 #define OCT 8

Definition at line 28 of file Print.h.

6.44 Print.h

```
/*
  Print.h - Base class that provides print() and println()
  Copyright (c) 2008 David A. Mellis.  All right reserved.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  This library is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
  Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with this library; if not, write to the Free Software
  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
*/

#ifndef Print_h
#define Print_h

#include <inttypes.h>
#include <stdio.h> // for size_t

#define DEC 10
#define HEX 16
#define OCT 8
#define BIN 2
#define BYTE 0

class Print
{
  private:
    void printNumber(unsigned long, uint8_t);
    void printFloat(double, uint8_t);
  public:
    virtual void write(uint8_t) = 0;
    virtual void write(const char *str);
    virtual void write(const uint8_t *buffer, size_t size);

    void print(const char[]);
    void print(char, int = BYTE);
    void print(unsigned char, int = BYTE);
    void print(int, int = DEC);
    void print(unsigned int, int = DEC);
    void print(long, int = DEC);
    void print(unsigned long, int = DEC);
    void print(double, int = 2);

    void println(const char[]);
    void println(char, int = BYTE);
    void println(unsigned char, int = BYTE);
    void println(int, int = DEC);
    void println(unsigned int, int = DEC);
    void println(long, int = DEC);
    void println(unsigned long, int = DEC);
    void println(double, int = 2);
    void println(void);
};
```

```

#endif
```

6.45 TimerCntr.cpp File Reference

```
#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>
#include "TimerCore.h"
#include "TimerCntr.h"
```

Defines

- #define TC0_ISR_DEF(tc)
- #define TC1_ISR_DEF(tc)

Functions

- TC0_ISR_DEF (TCC0)
- TC0_ISR_DEF (TCD0)
- TC1_ISR_DEF (TCC1)
- TC1_ISR_DEF (TCD1)
- static void SetPointer (TC0_t ∗tc, TimerCntr ∗pTC)
- static void SetPointer (TC1_t ∗tc, TimerCntr ∗pTC)

6.45.1 Define Documentation

6.45.1.1 #define TC0_ISR_DEF(*tc*)

Value:

```
static TimerCntr*  tc##cp = 0;\
ISR(tc##_ERR_vect) {\
    if (tc##cp) tc##cp->err();\
}\
ISR(tc##_OVF_vect) {\
    if (tc##cp) tc##cp->ovf();\
}\
ISR(tc##_CCA_vect) {\
    if (tc##cp) tc##cp->ccx(0);\
}\
ISR(tc##_CCB_vect) {\
    if (tc##cp) tc##cp->ccx(1);\
}\
ISR(tc##_CCC_vect) {\
    if (tc##cp) tc##cp->ccx(2);\
}\
ISR(tc##_CCD_vect) {\
    if (tc##cp) tc##cp->ccx(3);\
}
```

Definition at line 12 of file TimerCntr.cpp.

6.45.1.2 #define TC1_ISR_DEF(*tc*)

Value:

```
static TimerCntr*  tc##cp = 0;\
ISR(tc##_ERR_vect) {\
    if (tc##cp) tc##cp->err();\
}\
ISR(tc##_OVF_vect) {\
    if (tc##cp) tc##cp->ovf();\
}\
ISR(tc##_CCA_vect) {\
    if (tc##cp) tc##cp->ccx(0);\
}\
ISR(tc##_CCB_vect) {\
    if (tc##cp) tc##cp->ccx(1);\
}
```

Definition at line 47 of file TimerCntr.cpp.

6.45.2 Function Documentation

6.45.2.1 static void SetPointer (TC0_t ∗ *tc*, TimerCntr ∗ *pTC*) [static]

Definition at line 74 of file TimerCntr.cpp.

```
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if (tc == &TCC0) {
        TCC0cp = pTC;
    } else if (tc == &TCD0) {
        TCD0cp = pTC;
#if not defined(TIMERCORE_USE_TCE0)
    } else if (tc == &TCE0) {
        TCE0cp = pTC;
#endif
#if not defined(TIMERCORE_USE_TCF0)
    } else if (tc == &TCF0) {
        TCF0cp = pTC;
#endif
    }
}
```

6.45.2.2 static void SetPointer (TC1_t ∗ *tc*, TimerCntr ∗ *pTC*) [static]

Definition at line 94 of file TimerCntr.cpp.

```
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if (tc == &TCC1) {
        TCC1cp = pTC;
    } else if (tc == &TCD1) {
        TCD1cp = pTC;
#if not defined(TIMERCORE_USE_TCE1)
    } else if (tc == &TCE1) {
        TCE1cp = pTC;
#endif
#if not defined(TIMERCORE_USE_TCF1)
#if defined(TCF1)
    } else if (tc == &TCF1) {
        TCF1cp = pTC;
#endif
#endif
    }
}
```

6.45.2.3 TC0_ISR_DEF (TCD0)

6.45.2.4 TC0_ISR_DEF (TCC0)

6.45.2.5 TC1_ISR_DEF (TCC1)

6.45.2.6 TC1_ISR_DEF (TCD1)

6.46 TimerCntr.cpp

```


#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>

```

```
#include "TimerCore.h"
#include "TimerCntr.h"

// Generate all of the ISR handlers.. hook them up to a class if/when a class
// is instantiated for a particular USART.
#define TC0_ISR_DEF(tc) \
static TimerCntr*  tc##cp = 0;\
ISR(tc##_ERR_vect) {\
    if (tc##cp) tc##cp->err();\
}\
ISR(tc##_OVF_vect) {\
    if (tc##cp) tc##cp->ovf();\
}\
ISR(tc##_CCA_vect) {\
    if (tc##cp) tc##cp->ccx(0);\
}\
ISR(tc##_CCB_vect) {\
    if (tc##cp) tc##cp->ccx(1);\
}\
ISR(tc##_CCC_vect) {\
    if (tc##cp) tc##cp->ccx(2);\
}\
ISR(tc##_CCD_vect) {\
    if (tc##cp) tc##cp->ccx(3);\
}

TC0_ISR_DEF(TCC0);
TC0_ISR_DEF(TCD0);

// The TimerCore.cpp will use TCE0 and TCE1 if
// TCF1 is not defined.. Hence, I need to NOT define
// It here.
#if not defined(TIMERCORE_USE_TCE0)
TC0_ISR_DEF(TCE0);
#endif

#if not defined(TIMERCORE_USE_TCF0)
TC0_ISR_DEF(TCF0);
#endif

#define TC1_ISR_DEF(tc) \
static TimerCntr*  tc##cp = 0;\
ISR(tc##_ERR_vect) {\
    if (tc##cp) tc##cp->err();\
}\
ISR(tc##_OVF_vect) {\
    if (tc##cp) tc##cp->ovf();\
}\
ISR(tc##_CCA_vect) {\
    if (tc##cp) tc##cp->ccx(0);\
}\
ISR(tc##_CCB_vect) {\
    if (tc##cp) tc##cp->ccx(1);\
}

TC1_ISR_DEF(TCC1);
TC1_ISR_DEF(TCD1);
#if not defined(TIMERCORE_USE_TCE1)
TC1_ISR_DEF(TCE1);
#endif

#if not defined(TIMERCORE_USE_TCF1)
```

```
#ifdef TCF1
TC1_ISR_DEF(TCF1);
#endif
#endif

static void SetPointer(TC0_t* tc,TimerCntr* pTC)
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if (tc == &TCC0) {
        TCC0cp = pTC;
    } else if (tc == &TCD0) {
        TCD0cp = pTC;
#if not defined(TIMERCORE_USE_TCE0)
    } else if (tc == &TCE0) {
        TCE0cp = pTC;
#endif
#if not defined(TIMERCORE_USE_TCF0)
    } else if (tc == &TCF0) {
        TCF0cp = pTC;
#endif
    }
}

static void SetPointer(TC1_t* tc,TimerCntr* pTC)
{
    // Register this object with the appropriate
    // pointer so that the ISR routines can call p
    // class.
    if (tc == &TCC1) {
        TCC1cp = pTC;
    } else if (tc == &TCD1) {
        TCD1cp = pTC;
#if not defined(TIMERCORE_USE_TCE1)
    } else if (tc == &TCE1) {
        TCE1cp = pTC;
#endif
#if not defined(TIMERCORE_USE_TCF1)
#if defined(TCF1)
    } else if (tc == &TCF1) {
        TCF1cp = pTC;
#endif
#endif
    }
}

TimerCntr::TimerCntr(TC0_t*  pTC)
{
    _pTC = pTC;
    _bTC1 = false;
    _pNotifyClient = 0;
    _pNotifyClientID = 0;
    SetPointer(pTC,this);
}

TimerCntr::TimerCntr(TC1_t*  pTC)
{
    _pTC = (TC0_t*)pTC;
    _bTC1 = true;
    _pNotifyClient = 0;
    _pNotifyClientID = 0;
    SetPointer(pTC,this);
}

TimerCntr::~TimerCntr()
{
    if (_bTC1) {
        SetPointer((TC1_t*)_pTC,0);
    } else {
        SetPointer(_pTC,0);
    }
}


void TimerCntr::ClkSel(TC_CLKSEL_t clksel)
{
    _pTC->CTRLA = clksel;
}


void TimerCntr::CCEnable(uint8_t mask)
{
    _pTC->CTRLB = ((_pTC->CTRLB & 0x0F) | (mask << 4));
}


void TimerCntr::WaveformGenMode(TC_WGMODE_t wgmode)
{
    _pTC->CTRLB = ((_pTC->CTRLB & 0xF0) | wgmode);
}


void TimerCntr::EventSetup(TC_EVACT_t act, TC_EVSEL_t src)
{
    _pTC->CTRLD = act | src;
}


void TimerCntr::IntLvlA(uint8_t errlvl, uint8_t ovflvl)
{
    _pTC->INTCTRLA = (errlvl & 0x3) << 2 | (ovflvl & 0x3);
}

void TimerCntr::IntLvlB(uint8_t val)
{
    _pTC->INTCTRLB = val;
}

void TimerCntr::Counter(uint16_t newVal)
{
    _pTC->CNT = newVal;
}

uint16_t TimerCntr::Counter()
{
    return _pTC->CNT;
}


void TimerCntr::Period(uint16_t newVal)
{
    _pTC->PER = newVal;
}
```

```

uint16_t TimerCntr::Period()
{
    return _pTC->PER;
}


void TimerCntr::CCReg(uint8_t idx, uint16_t newVal)
{
    if (idx == 0) {
        _pTC->CCA = newVal;
    } else if (idx == 1) {
        _pTC->CCB = newVal;
    } else if (!_bTC1 && idx == 2) {
        _pTC->CCC = newVal;
    } else if (!_bTC1 && idx == 3) {
        _pTC->CCD = newVal;
    }
}

uint16_t TimerCntr::CCReg(uint8_t idx)
{
    if (idx == 0) {
        return _pTC->CCA;
    } else if (idx == 1) {
        return _pTC->CCB;
    } else if (!_bTC1 && idx == 2) {
        return _pTC->CCC;
    } else if (!_bTC1 && idx == 3) {
        return _pTC->CCD;
    }
    return 0;
}

void TimerCntr::Notify(TimerNotify* pClient,uint8_t id)
{
    _pNotifyClient = pClient;
    _pNotifyClientID = id;
}

void TimerCntr::err()
{
    if (_pNotifyClient)
        _pNotifyClient->err(_pNotifyClientID);
}

void TimerCntr::ovf()
{
    if (_pNotifyClient)
        _pNotifyClient->ovf(_pNotifyClientID);
}

void TimerCntr::ccx(uint8_t idx)
{
    if (_pNotifyClient)
        _pNotifyClient->ccx(_pNotifyClientID,idx);
}

void TimerCntr::SetRate(uint32_t rateHz){
    //add an auto prescaler using 32 bit array
}

```

6.47 TimerCntr.h File Reference

Classes

- class TimerNotify
- class TimerCntr

6.48 TimerCntr.h

```

#ifndef TimerCntr_h
#define TimerCntr_h

class TimerNotify
{
public:
    virtual void err(uint8_t id) = 0;
    virtual void ovf(uint8_t id) = 0;
    virtual void ccx(uint8_t id,uint8_t idx) = 0;
};

class TimerCntr
{
    // Since TC0 is a super-set, use this for the
    // default, but then we will override this in the
    // concrete classes.
    TC0_t*          _pTC;
    bool            _bTC1;
    TimerNotify*    _pNotifyClient;
    uint8_t         _pNotifyClientID;

public:

    TimerCntr(TC0_t* pTC);
    TimerCntr(TC1_t* pTC);
    ~TimerCntr();

    void Notify(TimerNotify* pClient, uint8_t id);

    void ovf();
    void err();
    void ccx(uint8_t idx);

    void ClkSel(TC_CLKSEL_t clksel);

    void CCEnable(uint8_t mask);

    void WaveformGenMode(TC_WGMODE_t wgmode);

    void EventSetup(TC_EVACT_t act, TC_EVSEL_t src);

    void IntLvlA(uint8_t errlvl, uint8_t ovflvl);

    void IntLvlB(uint8_t val);

    void Counter(uint16_t newVal);
    uint16_t Counter();

    void Period(uint16_t newPer);
    uint16_t Period();

    void SetRate(uint32_t rateHz);

    void CCReg(uint8_t idx, uint16_t newVal);
    uint16_t CCReg(uint8_t idx);

};


#endif
```

Index