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=======================xxxxxxxxxxxxxxxxxxxxxxx
```

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:

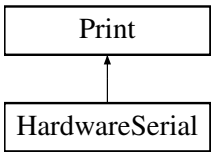
- fifo.h
- fifo.cpp

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.

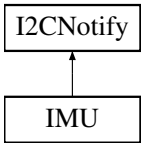
The documentation for this class was generated from the following file:

- 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.

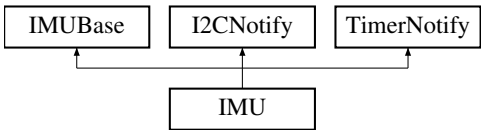
The documentation for this class was generated from the following file:

- 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().

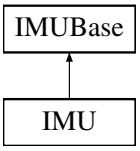
The documentation for this class was generated from the following files:

- 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().

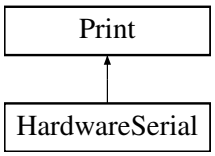
The documentation for this class was generated from the following file:

- 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 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().

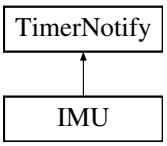
The documentation for this class was generated from the following files:

- 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

```
00001 #include <avr/io.h>
00002 #include <util/delay.h>
00003 #include <avr/interrupt.h>
00004 #include <stdint.h>
00005 #include <stdbool.h>
00006 #include <stdlib.h>
00007
00008
00010 #define AVR_ENTER_CRITICAL_REGION( ) uint8_t volatile saved_sreg = SREG; \
00011                                      cli();
00012
00016 #define AVR_LEAVE_CRITICAL_REGION( ) SREG = saved_sreg;
00017
00026 void CCPWrite( volatile uint8_t * address, uint8_t value )
00027 {
00028     AVR_ENTER_CRITICAL_REGION( );
00029
00030     volatile uint8_t * tmpAddr = address;
00031
00032     asm volatile(
00033         "movw r30,  %0"       "\n\t"
00034         "ldi  r16,  %2"       "\n\t"
00035         "out   %3, r16"       "\n\t"
00036         "st     Z,  %1"       "\n\t"
00037         :
00038         : "r" (tmpAddr), "r" (value), "M" (0xD8), "i" (&CCP)
00039         : "r16", "r30", "r31"
00040     );
00041
00042     AVR_LEAVE_CRITICAL_REGION( );
00043 }
```

```
00044
00045 /*********************************
00046 Function: init
00047 Purpose:
00048 A: Setup the clocking system.
00049 B: To Be Determined
00050
00051 We have an 8Mhz external oscillator. Set the PLL to multiply by
00052 16, to give a peripheral clock of 128Mhz.
00053 Set prescaler A to be 1, to get a 128Mhz peripheral clock.
00054 Set prescaler B to be 2 to get a 64Mhz 2X speed clock
00055 Set presclaer C to be 2 to get a 32Mhz CPU Clock
00056
00057 *********************************/
00058 void clksystem_init()
00059 {
00060     // This board does not have an external oscillator,
00061     // so we need to use the internal oscillator.
00062     // Enable the internal 32Mhz oscillator. Disables all the rest.
00063     OSC.CTRL = OSC_RC32MEN_bm | OSC_RC2MEN_bm | OSC_RC32KEN_bm;
00064
00065     //OSC.XOSCCTRL = 0; // we don't care, just set this to some value.
00066
00067     // Wait for the external oscilator. Don't wait forever though.
00068     int maxWait = 100; // Wait 1 second
00069     while (--maxWait && !(OSC.STATUS & (OSC_RC32MRDY_bm | OSC_RC32KRDY_bm | OSC_R
      C2MRDY_bm))) {
00070         _delay_loop_2(10);
00071     }
00072
00073     OSC.DFLLCTRL = 0;
00074     DFLLRC32M.CTRL = 0x1;
00075     DFLLRC2M.CTRL = 0x1;
00076
00077     // If the external oscilator is running, then we can switch the PLL
00078     // over to it and wait for the PLL to stabilize.
00079     if (OSC.STATUS & ( OSC_RC32MRDY_bm | OSC_RC2MRDY_bm)) {
00080         // Setup the PLL to use the internal 32Mhz oscillator and a
00081         // factor of 4 to get a 128Mhz PLL Clock.
00082
00083         // Make sure this is disabled before we try and configure it.
00084         OSC.CTRL &= ~OSC_PLLEN_bm;
00085
00086         OSC.PLLCTRL = OSC_PLLSRC_RC32M_gc | (16 << OSC_PLLFAC_gp);
00087
00088         OSC.CTRL |= OSC_PLLEN_bm;
00089
00090         //OSC.CTRL = CLK_PSADIV_16_gc;
00091         // Wait for OSC.STATUS to indicate that PLL is ready..
00092         while (!(OSC.STATUS & OSC_PLLRDY_bm)) {
00093         }
00094
00095         // Okay, the PLL is up on the new clock. Set the prescalers then
00096         // switch over to the PLL.
00097
00098         CCP = CCP_IOREG_gc;
00099         CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_2_2_gc;
00100         //CLK.PSCTRL = CLK_PSADIV_1_gc | CLK_PSBCDIV_1_1_gc;
00101
00102         // When all is done, make the PLL be the clock source for the system.
00103         CCP = CCP_IOREG_gc;
00104         CLK.CTRL = CLK_SCLKSEL_PLL_gc;
```

```
00105         //CLK.CTRL = CLK_SCLKSEL_RC32M_gc;
00106         //CLK.CTRL = CLK_SCLKSEL_RC2M_gc;
00107     }
00108 }
```

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

```
00001
00002 #ifndef _CLKSYSTEM_
00003 #   define _CLKSYSTEM_ 1
00004
00005
00006 int clksystem_init();
00007
00008
00009 #endif
```

6.5 CmdProcessor.cpp File Reference

```
#include <stdlib.h>

#include <string.h>

#include "CmdProcessor.h"
```

6.6 CmdProcessor.cpp

```
00001
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include "CmdProcessor.h"
00005
00011 CmdProcessor::CmdProcessor(HardwareSerial* pHW)
00012 {
00013     _pHW = pHW;
00014
00015     _pCmdString = (char*)malloc(128);
00016     _pCmd = 0;
00017     _pCmdTerm = (char*)malloc(3);
00018     strcpy(_pCmdTerm,"\n\r");
00019     _pCmdDelim = (char*)malloc(3);
00020     strcpy(_pCmdDelim," \t");
00021     resetCmd();
00022 }
00023
00025 CmdProcessor::~CmdProcessor()
```

```
00026 {
00027     if (_pHW) {
00028         _pHW->end();
00029     }
00030     free(_pCmdString);
00031     free(_pCmdDelim);
00032     free(_pCmdTerm);
00033 }
00034
00036 char* CmdProcessor::cmdTerm() { return _pCmdTerm; }
00037
00040 void CmdProcessor::cmdTerm(char* t)
00041 {
00042     free(_pCmdTerm);
00043     _pCmdTerm = (char*)malloc(strlen(t) + 1);
00044     strcpy(_pCmdTerm,t);
00045 }
00046
00048 const char* CmdProcessor::cmdDelim()
00049 {
00050     return _pCmdDelim;
00051 }
00052
00055 void CmdProcessor::cmdDelim(const char* d)
00056 {
00057     free(_pCmdDelim);
00058     _pCmdDelim = (char*)malloc(strlen(d) + 1);
00059     strcpy(_pCmdDelim,d);
00060 }
00061
00068 bool CmdProcessor::checkCommands()
00069 {
00070     while (_pHW->available() > 0) {
00071         unsigned char c = _pHW->read();
00072         if (strchr(_pCmdTerm,c) != 0) {
00073             if (_cmdPos > 0) {
00074                 // Done with this command.
00075                 _pCmdString[_cmdPos] = 0; // Null terminate command
00076                 processCmd();
00077                 return 1;
00078             } else {
00079                 _pHW->print("Ok\n");
00080             }
00081         } else {
00082             _pCmdString[_cmdPos++] = c;
00083         }
00084     }
00085     return 0;
00086 }
00087
00092 void CmdProcessor::processCmd()
00093 {
00094     // See if the command delimiter exists in the
00095     // command. if it does not, then the command
00096     // is the entire string.
00097     if (strpbrk(_pCmdString,_pCmdDelim)) {
00098         _pCmd = strtok(_pCmdString,_pCmdDelim);
00099         char* pTok = strtok(0,_pCmdDelim);
00100         int i = 0;
00101         while (i < 10 && pTok) {
00102             _pTokens[i++] = pTok;
00103             pTok = strtok(0,_pCmdDelim);
```

```
00104         }
00105         _paramCnt = i;
00106         _validCmd = true;
00107     } else {
00108         _pCmd = _pCmdString;
00109         _paramCnt = 0;
00110         _validCmd = true;
00111     }
00112 }
00113
00115 void CmdProcessor::resetCmd()
00116 {
00117     _cmdPos = 0;
00118     _validCmd = false;
00119     _paramCnt = 0;
00120 }
00121
00123 const char* CmdProcessor::getCmd()
00124 {
00125     return _pCmd;
00126 }
00127
00129 uint8_t CmdProcessor::paramCnt()
00130 {
00131     return _paramCnt;
00132 }
00133
00134
00144
00146 void CmdProcessor::getParam(uint8_t idx,uint16_t &p)
00147 {
00148     if (idx < _paramCnt) {
00149         p = atoi(_pTokens[idx]);
00150     }
00151 }
00152
00154 void CmdProcessor::getParam(uint8_t idx,uint8_t &p)
00155 {
00156     if (idx < _paramCnt) {
00157         p = atoi(_pTokens[idx]);
00158     }
00159 }
00160
00161 void CmdProcessor::getParam(uint8_t idx,int &p)
00162 {
00163     if (idx < _paramCnt) {
00164         p = atoi(_pTokens[idx]);
00165     }
00166 }
00167
00168 void CmdProcessor::getParam(uint8_t idx,long &l)
00169 {
00170     if (idx < _paramCnt) {
00171         l = atol(_pTokens[idx]);
00172     }
00173 }
00174
00176 void CmdProcessor::getParam(uint8_t idx,double &p)
00177 {
00178     if (idx < _paramCnt) {
00179         uint8_t nScans;
00180         nScans = sscanf(_pTokens[idx],"%lf", &p);
```

```
00181         //p = atof(_pTokens[idx]);
00182     }
00183 }
00184
00186 void CmdProcessor::getParam(uint8_t idx,char*& p, uint8_t maxlen)
00187 {
00188     if (idx < _paramCnt) {
00189         strncpy(p,_pTokens[idx],maxlen);
00190     }
00191 }
00192
```

6.7 CmdProcessor.h File Reference

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

Classes

- class CmdProcessor

6.8 CmdProcessor.h

```
00001 #ifndef CMDPROCESSOR_H
00002 #define CMDPROCESSOR_H
00003
00004 #include <avr/io.h>
00005 #include "HardwareSerial.h"
00006
00007 class CmdProcessor
00008 {
00009 protected:
00010     HardwareSerial* _pHW;
00011     char*           _pTokens[10];
00012     char*           _pCmd;
00013     char*           _pCmdString;
00014     uint8_t         _cmdPos;
00015     bool            _validCmd;
00016     char*           _pCmdTerm;
00017     char*           _pCmdDelim;
00018     uint8_t         _paramCnt;
00019
00020 public:
00021     CmdProcessor(HardwareSerial* pHW);
00022     ~CmdProcessor();
00023
00024     bool checkCommands();
00025     char* cmdTerm();
00026     void cmdTerm(char*);
00027     void resetCmd();
00028     const char* cmdDelim();
00029     void cmdDelim(const char*);
00030     const char* getCmd();
00031     uint8_t paramCnt();
00032     void getParam(uint8_t idx,uint8_t &p);
00033     void getParam(uint8_t idx,uint16_t &p);
```

```
00034     void getParam(uint8_t idx,long &l);
00035     void getParam(uint8_t idx,int &p);
00036     void getParam(uint8_t idx,double &f);
00037     void getParam(uint8_t idx,char*& p, uint8_t maxlen=128);
00038 protected:
00039     void processCmd();
00040
00041 };
00042
00043 #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

```
00001 __extension__ typedef int __guard __attribute__((mode (__DI__)));
00002
00003 extern "C" int __cxa_guard_acquire(__guard *);
00004 extern "C" void __cxa_guard_release (__guard *);
00005 extern "C" void __cxa_guard_abort (__guard *);
00006 extern "C" void __cxa_pure_virtual(void);
00007
```

6.13 Documentation.html File Reference

6.14 Documentation.html

```
00001
```

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

```
00001 #include <stdio.h>
00002 #include <string.h>
00003 #include <math.h>
00004
00005 #include "NewDel.h"
00006 #include "fifo.h"
00007
00014 Fifo::Fifo(uint8_t size)
00015 {
00016     _size = size;
00017     _pdata = (FifoType*)malloc(_size * sizeof(FifoType));
00018     clear();
00019 }
00020
```

```
00022 void Fifo::clear()
00023 {
00024     _start = _end = _pdata;
00025 }
00026
00033
00040
00050 uint8_t Fifo::count()
00051 {
00052     if (_end == _start) return 0;
00053     if (_end > _start) {
00054         return _end - _start;
00055     }
00056     return _size - (_start - _end);
00057 }
00058
00060 bool Fifo::full()
00061 {
00062     return (_start - _end)  == 1 || (_end - _start) == _size;
00063 }
00064
00066 bool Fifo::empty()
00067 {
00068     return (_start == _end);
00069 }
00070
00074 int8_t Fifo::push(FifoType* d)
00075 {
00076     if (full()) return -1;
00077     *(_end++) = *d;
00078
00079     // Wrap the end back to the beginning.
00080     if ((_end - _pdata) > _size) {
00081         _end = _pdata;
00082     }
00083
00084         return 0;
00085 }
00086
00092 int8_t Fifo::pop(FifoType* pD)
00093 {
00094     if (empty()) {
00095         return -1; // Nothing else to do
00096     }
00097     *pD = *(_start++);
00098     if ((_start - _pdata) > _size) {
00099         _start = _pdata;
00100     }
00101     return 0;
00102 }
00103
00110 void FifoTest(HardwareSerial& ds)
00111 {
00112     Fifo f1(20);
00113     Fifo::FifoType x;
00114     char    buffer[128];
00115
00116     for (x=1;x<25;x++) {
00117         f1.push(&x);
00118         sprintf(buffer,"Fifo push %d count %d\n",x,f1.count());
00119         ds.print(buffer);
00120     }
```

```
00121     f1.clear();
00122     sprintf(buffer,"Fifo count %d\n",f1.count());
00123     ds.print(buffer);
00124
00125     for (x=0;x<10;x++) {
00126         f1.push(&x);
00127     }
00128     sprintf(buffer,"Fifo count %d. Expected 10\n",f1.count());
00129     ds.print(buffer);
00130
00131     Fifo::FifoType y;
00132     for (x=1;x<15;x++) {
00133         f1.pop(&y);
00134         sprintf(buffer,"Fifo pop %d count %d\n",y,f1.count());
00135         ds.print(buffer);
00136     }
00137 }
00138
```

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

```
00001
00002 #ifndef _fifo_h
00003 #define _fifo_h
00004
00005 #include <inttypes.h>
00006 #include "HardwareSerial.h"
00007
00016 class Fifo
00017 {
00018 public:
00019
00020     typedef uint8_t  FifoType;
00021 private:
00022
00023     FifoType*       _pdata;
00024     FifoType*       _start;
00025     FifoType*       _end;
00026     uint8_t         _size;
00027
00028 public:
00029     Fifo(uint8_t size);
00030
00031
00032     int8_t push(FifoType*);
00033     int8_t pop(FifoType* pData);
00034     uint8_t count();
00035     bool full();
00036     bool empty();
00037     void clear();
00038 };
00039
00040 extern void FifoTest(HardwareSerial& ds);
00041
00042 #endif
```

```
00043
00044
```

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.
```

```
00054     // Specify the USART, PORT, rxPin and txPin and the baud rate.
00055     HardwareSerial cmdSerial(&USARTD0, &PORTD, PIN2_bm, PIN3_bm);
00056     cmdSerial.begin(115200);
00057     //cmdSerial.begin(285000);
00058     //cmdSerial.begin(921600);
00059 
00060     CCP = CCP_IOREG_gc;
00061     MCU.MCUCR = MCU_JTAGD_bm;
00062     DebugPort dbgPort(&PORTB);
00063     dbgPort.PinMask(0xF0,4);
00064     dbgPort.SetState(0xf);
00065     dbgPort.SetState(0);
00066     pdbgport = &dbgPort;
00067 
00068     I2C_Master  *pMaster[4];
00069     I2C_Master hand(&TWIC);
00070     I2C_Master single(&TWID);
00071     I2C_Master pair1(&TWIE);
00072     I2C_Master pair2(&TWIF);
00073 
00074     pMaster[0] = &hand;
00075     pMaster[1] = &single; // Pinkie
00076     pMaster[2] = &pair1;  // Ring + Middle
00077     pMaster[3] = &pair2;  // Index + Thumb
00078 
00079     for (int x=0;x<4;x++) {
00080         if (pMaster[x]) {
00081             pMaster[x]->begin(400e3);
00082         }
00083     }
00084 
00085     // Give the hardware time to stabilize..
00086     _delay_ms(1000);
00087 
00088     // When constructed without a list of ID's, the IMU will query known
00089     // ID's 0xD0 and 0xD2 for the Gyro, and then 0x30 and 0x32 respectively
00090     // for the ACC.
00091     IMU     hand_imu(&hand);
00092     IMU     single_imu(&single);
00093     IMU     pair1_imu(&pair1);
00094     IMU     pair2_imu(&pair2);
00095 
00096     // These all share the debug port, so they must operate
00097     // in sequence. If I went back to a parallel operation, then
00098     // this would have to change.
00099     hand_imu.SetDebugPort(&dbgPort);
00100 
00101     single_imu.SetDebugPort(&dbgPort);
00102 
00103     pair1_imu.SetDebugPort(&dbgPort);
00104 
00105     pair2_imu.SetDebugPort(&dbgPort);
00106 
00107 #ifdef USE_DEBUGPORT_2
00108     hand_imu.SetDebugPort2(&dbgPort2);
00109     single_imu.SetDebugPort2(&dbgPort2);
00110     pair1_imu.SetDebugPort2(&dbgPort2);
00111     pair2_imu.SetDebugPort2(&dbgPort2);
00112 #endif
00113 
00116     TimerCntr   tcA(&TCC1);
00117 
```

```
00118     IMUManager imumgr(&cmdSerial);
00119     imumgr.SetBlueLed(&PORTJ, PIN7_bm);
00120     imumgr.LedOff();
00121     imumgr.SetTimer(&tcA);
00122     imumgr.AddIMU(&hand_imu);
00123     imumgr.AddIMU(&single_imu);
00124     imumgr.AddIMU(&pair1_imu);
00125     imumgr.AddIMU(&pair2_imu);
00126     imumgr.SampleRate(10);
00127
00128     GyroCmdProcessor cmdproc(&cmdSerial,&pMaster[0],&imumgr);
00129
00130     sprintf(buffer,"Welcome to Gyro Glove.\nRev %d.%d.%d Date: %02d/%02d/%04d Bui
      lt at: %02d:%02d\n",
00131         RevMajor, RevMinor, RevInc,
00132         DateMonth, DateDay, DateYear,
00133         TimeHour, TimeMin
00134         );
00135     cmdSerial.print(buffer);
00136
00138     TimerCntr mdtc(&TCD0);
00139     MyDriver md;
00140     md.SetTimer(&mdtc);
00141
00142     cmdSerial.print("Starting endless loop\n");
00143
00144     while(1) {
00145         cmdproc.Loop();
00146         imumgr.Loop();
00147     }
00148
00149     return 0;
00150 }
00151
00152
```

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

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <inttypes.h>
00005 #include <avr/io.h>
00006 #include <avr/interrupt.h>
00007
00008 #include "HardwareSerial.h"
00009
00010 // Define constants and variables for buffering incoming serial data.  We're
00011 // using a ring buffer (I think), in which rx_buffer_head is the index of the
00012 // location to which to write the next incoming character and rx_buffer_tail
00013 // is the index of the location from which to read.
00014 #define RX_BUFFER_SIZE 128
00015
00016 struct ring_buffer {
00017     unsigned char buffer[RX_BUFFER_SIZE];
00018     int head;
00019     int tail;
00020 };
00021
```

```
00022 // Generate all of the ISR handlers.. hook them up to a class if/when a class
00023 // is instantiated for a particular USART.
00024 #define SERIAL_ISR_DEF(usart) \
00025 static HardwareSerial*  usart##cp = 0;\
00026 ISR(usart##_RXC_vect) {\
00027     if (usart##cp) usart##cp->rxc();\
00028 }\
00029 ISR(usart##_DRE_vect) {\
00030     if (usart##cp) usart##cp->dre();\
00031 }\
00032 ISR(usart##_TXC_vect) {\
00033     if (usart##cp) usart##cp->txc();\
00034 }
00035
00036 SERIAL_ISR_DEF(USARTC0);
00037 SERIAL_ISR_DEF(USARTC1);
00038 SERIAL_ISR_DEF(USARTD0);
00039 SERIAL_ISR_DEF(USARTD1);
00040 SERIAL_ISR_DEF(USARTE0);
00041 SERIAL_ISR_DEF(USARTE1);
00042 SERIAL_ISR_DEF(USARTF0);
00043 #if defined(USARTF1_RXC_vect)
00044 SERIAL_ISR_DEF(USARTF1);
00045 #endif
00046
00047 inline void store_char(unsigned char c, ring_buffer *rx_buffer)
00048 {
00049     int i = (rx_buffer->head + 1) % RX_BUFFER_SIZE;
00050
00051     // if we should be storing the received character into the location
00052     // just before the tail (meaning that the head would advance to the
00053     // current location of the tail), we're about to overflow the buffer
00054     // and so we don't write the character or advance the head.
00055     if (i != rx_buffer->tail) {
00056         rx_buffer->buffer[rx_buffer->head] = c;
00057         rx_buffer->head = i;
00058     }
00059 }
00060
00061 static void SetPointer(USART_t* usart,HardwareSerial* p)
00062 {
00063     // Register this object with the appropriate
00064     // pointer so that the ISR routines can call p
00065     // class.
00066     if(usart == &USARTC0) {
00067         USARTC0cp = p;
00068     } else if (usart == &USARTC1) {
00069         USARTC1cp = p;
00070     } else if (usart ==  &USARTD0) {
00071         USARTD0cp = p;
00072     } else if (usart ==  &USARTD1) {
00073         USARTD1cp = p;
00074     } else if (usart ==  &USARTE0) {
00075         USARTE0cp = p;
00076     } else if (usart ==  &USARTE1) {
00077         USARTE1cp = p;
00078     } else if (usart ==  &USARTF0) {
00079         USARTF0cp = p;
00080 #if defined(USARTF1_RXC_vect)
00081     } else if (usart ==  &USARTF1) {
00082         USARTF1cp = p;
00083 #endif
```

```
00084     }
00085 }
00086
00093
00094 void HardwareSerial::rxc()
00095 {
00096     unsigned char c = _usart->DATA;
00097     store_char(c,_rx_buffer);
00098 }
00099
00100 void HardwareSerial::dre()
00101 {
00102 }
00103
00104 void HardwareSerial::txc()
00105 {
00106 }
00107
00109
00110 // Constructors ////////////////////////////////////////////////////////////
00111
00112 HardwareSerial::HardwareSerial(
00113     USART_t     *usart,
00114     PORT_t      *port,
00115     uint8_t      in_bm,
00116     uint8_t      out_bm)
00117 {
00118     _rx_buffer = (ring_buffer*)malloc(RX_BUFFER_SIZE+2*sizeof(int));
00119     _usart     = usart;
00120     _port      = port;
00121     _in_bm     = in_bm;
00122     _out_bm    = out_bm;
00123     _bsel      = 0;
00124     _bscale    = 0;
00125     _baudrate  = 9600;
00126     _bEn       = true;
00127     SetPointer(_usart,this);
00128 }
00129
00130 HardwareSerial::~HardwareSerial()
00131 {
00132     end();
00133     free(_rx_buffer);
00134     _rx_buffer = 0;
00135     SetPointer(_usart,0);
00136 }
00137
00138 // Public Methods //////////////////////////////////////////////////////////
00139
00140 void HardwareSerial::begin(long baud,int8_t bscale)
00141 {
00142     uint16_t BSEL;
00143     _bscale = bscale;
00144     _baudrate = baud;
00145
00146     float fPER = F_CPU;
00147     float fBaud = baud;
00148
00149     _port->DIRCLR = _in_bm;  // input
00150     _port->DIRSET = _out_bm; // output
00151
00152     // set the baud rate
```

```
00153     if (bscale >= 0) {
00154         BSEL = fPER/((1 << bscale) * 16 * baud) - 1;
00155         //BSEL = F_CPU / 16 / baud - 1;
00156     } else {
00157         bscale = -1 * bscale;
00158         BSEL = (1 << bscale) * (fPER/(16.0 * fBaud) - 1);
00159     }
00160 
00161     _usart->BAUDCTRLA = (uint8_t)BSEL;
00162     _usart->BAUDCTRLB = ((bscale & 0xf) << 4) | ((BSEL & 0xf00) >> 8);
00163 
00164     // enable Rx and Tx
00165     _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
00166     // enable interrupt
00167     _usart->CTRLA = USART_RXCINTLVL_HI_gc;
00168 
00169     // Char size, parity and stop bits: 8N1
00170     _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
00171 }
00172 
00173 void HardwareSerial::begin2x(long baud,int8_t bscale)
00174 {
00175     uint16_t baud_setting;
00176     _bscale = bscale;
00177     _baudrate = baud;
00178 
00179     // TODO: Serial. Fix serial double clock.
00180     long fPER = F_CPU * 4;
00181 
00182     _port->DIRCLR = _in_bm;  // input
00183     _port->DIRSET = _out_bm; // output
00184 
00185     // set the baud rate using the 2X calculations
00186     _usart->CTRLB |= 1 << 1; // the last 1 was the _u2x value
00187     baud_setting = fPER / 8 / baud - 1;
00188 
00189     _usart->BAUDCTRLA = (uint8_t)baud_setting;
00190     _usart->BAUDCTRLB = baud_setting >> 8;
00191 
00192 
00193     // enable Rx and Tx
00194     _usart->CTRLB |= USART_RXEN_bm | USART_TXEN_bm;
00195     // enable interrupt
00196     _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
      ;
00197 
00198     // Char size, parity and stop bits: 8N1
00199     _usart->CTRLC = USART_CHSIZE_8BIT_gc | USART_PMODE_DISABLED_gc;
00200     SetPointer(_usart,this);
00201 }
00202 
00203 void HardwareSerial::end()
00204 {
00205     SetPointer(_usart,(HardwareSerial*)0);
00206 
00207     // disable Rx and Tx
00208     _usart->CTRLB &= ~(USART_RXEN_bm | USART_TXEN_bm);
00209     // disable interrupt
00210     _usart->CTRLA = (_usart->CTRLA & ~USART_RXCINTLVL_gm) | USART_RXCINTLVL_LO_gc
      ;
00211 }
00212 
```

```
00213 uint8_t HardwareSerial::available(void)
00214 {
00215     return (RX_BUFFER_SIZE + _rx_buffer->head - _rx_buffer->tail) %
      RX_BUFFER_SIZE;
00216 }
00217
00218 int HardwareSerial::read(void)
00219 {
00220     // if the head isn't ahead of the tail, we don't have any characters
00221     if (_rx_buffer->head == _rx_buffer->tail) {
00222         return -1;
00223     } else {
00224         unsigned char c = _rx_buffer->buffer[_rx_buffer->tail];
00225         _rx_buffer->tail = (_rx_buffer->tail + 1) % RX_BUFFER_SIZE;
00226         return c;
00227     }
00228 }
00229
00230 void HardwareSerial::flush()
00231 {
00232     // don't reverse this or there may be problems if the RX interrupt
00233     // occurs after reading the value of rx_buffer_head but before writing
00234     // the value to rx_buffer_tail; the previous value of rx_buffer_head
00235     // may be written to rx_buffer_tail, making it appear as if the buffer
00236     // were full, not empty.
00237     _rx_buffer->head = _rx_buffer->tail;
00238 }
00239
00240 void HardwareSerial::write(uint8_t c)
00241 {
00242     if (_bEn) {
00243         while ( !(_usart->STATUS & USART_DREIF_bm) );
00244         _usart->DATA = c;
00245     }
00246 }
00247
00248 void HardwareSerial::enable(bool bEn)
00249 {
00250     _bEn = bEn;
00251 }
00252
00253
```

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

```
00001 #ifndef HardwareSerial_h
00002 #define HardwareSerial_h
00003
00004 #include <avr/io.h>
00005 #include <inttypes.h>
00006
00007 #include "Print.h"
00008
00009 struct ring_buffer;
00010
00012
00023 class HardwareSerial : public Print
00024 {
00025   protected:
00026     ring_buffer *_rx_buffer;
00027     USART_t     *_usart;
00028     PORT_t      *_port;
00029     uint8_t     _in_bm;
00030     uint8_t     _out_bm;
00031     uint8_t     _bsel;
00032     int8_t     _bscale;
00033     long        _baudrate;
00034     bool        _bEn;
00035   public:
00036     HardwareSerial(
00037         USART_t     *usart,
00038         PORT_t      *port,
00039         uint8_t     in_bm,
00040         uint8_t     out_bm);
00041     ~HardwareSerial();
00042     void begin(long baudrate,int8_t bscale = 0);
00043     void begin2x(long baudrate,int8_t bscale = 0);
00044     void end();
00045     uint8_t available(void);
00046     int read(void);
00047     void flush(void);
00048     virtual void write(uint8_t);
00049     using Print::write; // pull in write(str) and write(buf, size) from Print
00050     void rxc();
00051     void dre();
00052     void txc();
00053     void enable(bool bEn);
00054 };
00055
00056
00057 #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

```
00001
00002 #ifndef I2C_Master_h
00003 #define I2C_Master_h
00004
00005 #include <stdlib.h>
00006 #include <inttypes.h>
00007 #include <avr/io.h>
00008
00009 #include "I2CTransaction.h"
00010
00011 // Derive a class from this class
00012 // in order to give the class the ability
00013 // to be notified when a write or read is done.
00014 class I2CNotify
00015 {
00016 public:
00017     virtual void I2CWriteDone() = 0;
00018     virtual void I2CReadDone() = 0;
00019     virtual void I2CBusError() = 0;
00020     virtual void I2CArbLost() = 0;
00021     virtual void I2CNack() = 0;
00022 };
00023
00024
00025 class I2C_Master
00026 {
00027 public:
00028     typedef enum {
00029         sIdle,
00030         sBusy,
00031         sError,
00032         sArb,
00033         sIDScan,
00034         sIDCheck
00035     } DriverState;
00036
00037     typedef enum {
00038         rOk,
00039         rFail,
00040         rArbLost,
00041         rBussErr,
00042         rNack,
00043         rBufferOverrun,
00044         rUnknown,
00045         rTimeout
00046     } DriverResult;
00047
00048 private:
00049     TWI_t*          _twi;
00050     PORT_t*         _twiPort;
00051     bool            _bEnabled;
00052     DriverState     _State;
00053     DriverResult    _Result;
```

```
00054     void*           _pReserved;
00055     I2CNotify*      _pNotifyClient;
00056
00057     // Transaction Data
00058     uint8_t         _DeviceID;
00059     uint8_t         _nBytesWritten;
00060     uint8_t         _nWriteBytes;
00061     uint8_t         _nReadBytes;
00062     uint8_t         _nBytesRead;
00063
00064     uint8_t*        _WriteData;
00065     uint8_t         _wrBufferLen;
00066     uint8_t*        _ReadData;
00067     uint8_t         _rdBufferLen;
00068
00069     //I2CTransaction* _pTransaction;
00070
00071     // For ID Scanning
00072     uint8_t         _idScanCurrent;
00073     uint8_t         _IDList[128];
00074     bool            _ScanComplete;
00075
00076 public:
00077
00078     typedef enum ErrorType {
00079         eNone           = 0,
00080         eDisabled       = -1,
00081         eBusy           = -2,
00082         eNack           = -3,
00083         eArbLost        = -4,
00084         eBusErr         = -5,
00085         eTimeout        = -6,
00086         eSDAStuck       = -7,
00087         eSCLStuck       = -8,
00088         eUnknown        = -9
00089     } ErrorType;
00090
00091     I2C_Master(TWI_t*  twi);
00092     ~I2C_Master();
00093
00094     //
00095     void begin(uint32_t freq);
00096     void end();
00097
00098     // Perform a write. Return status indicates result
00099     // < 0 indicates an error: -1 == NACK -1 = Some other error???
00100     // > 0 indicates # of bytes written with valid ACK.
00101     ErrorType Write(uint8_t ID, uint8_t* Data, uint8_t nBytes);
00102     ErrorType WriteSync(uint8_t ID, uint8_t* Data, uint8_t nBytes);
00103
00104     // Perform a read. Return status indicates result.
00105     // maxcnt indidcates how many bytes to attempt to read. Will read until
00106     // a NACK occurs or maxcnt bytes are read.
00107     // RETC < 0: -1 -> NACK of ID.
00108     // RETC > 0: number of bytes read (before NACK from Slave
00109     ErrorType Read(uint8_t ID, uint8_t nBytes);
00110     ErrorType ReadSync(uint8_t ID, uint8_t nBytes);
00111
00112     ErrorType WriteRead(uint8_t ID,
00113                         uint8_t* wrData,
00114                         uint8_t nWriteBytes,
00115                         uint8_t nReadBytes);
```

```
00116     ErrorType WriteReadSync(uint8_t ID,
00117                   uint8_t* wrData,
00118                   uint8_t nWriteBytes,
00119                   uint8_t nReadBytes);
00120
00121
00122     void master_int();
00123     void slave_int();
00124
00125     void WriteHandler();
00126     void ReadHandler();
00127     void ArbHandler();
00128     void ErrorHandler();
00129
00130     void MasterFinished();
00131     int testack(uint8_t ID);
00132     void dumpregs();
00133
00134     I2C_Master::DriverResult Result();
00135     I2C_Master::DriverState State();
00136     uint8_t ReadData(uint8_t* pData, uint8_t maxcnt);
00137     uint8_t ReadData(uint8_t index);
00138     uint8_t nReadBytes();
00139
00140     ErrorType CheckID(uint8_t ID);
00141     void Stop();      // Use I2C Master for stop.
00142     ErrorType ForceStartStop();
00143     ErrorType WigglePin(uint8_t cnt, uint8_t pinSel, uint8_t otherState);
00144     void CleanRegs();
00145
00146     void loop(); // Called periodically.
00147
00148     bool busy(); // In the process of transacting...
00149     void* isReserved();
00150     bool Reserve(void*);
00151     void NotifyMe(I2CNotify* pMe);
00152
00153     inline bool IsIdle()
00154     {
00155         return (_twi->MASTER.STATUS & TWI_MASTER_BUSSTATE_gm)
00156             == TWI_MASTER_BUSSTATE_IDLE_gc;
00157     }
00158
00159 protected:
00160     uint8_t busState();
00161     void showstate();
00162
00163
00164 };
00165
00166
00167 #endif
00168
```

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

```
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 "IMUPacketFifo.h"
00010 #include "TimerCore.h"
00011
00012 #include "IMU.h"
00013 #include "Mark.h"
00014
```

```
00015 extern HardwareSerial* pdbgserial;
00016 static char buffer[128];
00017
00020 IMU::IMU(I2C_Master* pMas)
00021 {
00022     _pNextIMU       = 0;
00023     _gID[0]         = 0;
00024     _aID[0]         = 0;
00025     _gID[1]         = 0;
00026     _aID[1]         = 0;
00027     _bDualChan      = false;
00028     _numChans       = 0;
00029     _pMas           = pMas;
00030     _DLPF           = 0x1;
00031     _FullScale      = 0x1;
00032     _ClkSel         = 0x1;
00033     _Rate           = 10;
00034     _State          = sIdle;
00035     _previousState  = sIdle;
00036     _failType       = fNone;
00037     _bRun           = false;
00038     _busyWaitTime   = 0;
00039     _bDataReady[0]  = false;
00040     _bDataReady[1]  = false;
00041
00042     ResetFailStats();
00043     _pDBGPort       = 0;
00044     _pDBGPort2      = 0;
00045     _pNextIMU       = 0;
00046     _pTimer         = 0;
00047     _bUseGyro       = false;
00048     _pMas->NotifyMe(this);
00049     QueryChannels();
00050 }
00051
00053 IMU::IMU(I2C_Master* pMas, uint8_t gID, uint8_t aID)
00054 {
00055     _pNextIMU   = 0;
00056     _gID[0]     = gID;
00057     _aID[0]     = aID;
00058     _bDualChan  = false;
00059     _numChans   = 1;
00060     _pMas       = pMas;
00061     _DLPF       = 0x1;
00062     _FullScale  = 0x1;
00063     _ClkSel     = 0x1;
00064     _Rate       = 10;
00065     _State      = sIdle;
00066     _bRun       = false;
00067     _pMas->NotifyMe(this);
00068     _pDBGPort   = 0;
00069     _pDBGPort2  = 0;
00070     _pTimer     = 0;
00071     ResetFailStats();
00072     _bUseGyro   = false;
00073 }
00074
00076 IMU::IMU(I2C_Master* pMas,
00077         uint8_t gID, uint8_t aID,
00078         uint8_t gID2, uint8_t aID2
00079     )
00080 {
```

```
00081     _pNextIMU   = 0;
00082     _gID[0]     = gID;
00083     _aID[0]     = aID;
00084     _gID[1]     = gID2;
00085     _aID[1]     = aID2;
00086     _bDualChan  = true;
00087     _numChans   = 2;
00088     _pMas       = pMas;
00089     _DLPF       = 0x1;
00090     _FullScale  = 0x1;
00091     _ClkSel     = 0x1;
00092     _Rate       = 10;
00093     _State      = sIdle;
00094     _bRun       = false;
00095     _pMas->NotifyMe(this);
00096     _pDBGPort   = 0;
00097     _pDBGPort2  = 0;
00098     _pTimer     = 0;
00099     ResetFailStats();
00100     _bUseGyro   = false;
00101 }
00102
00103 void IMU::NextIMU(IMUBase* pNext)
00104 {
00105     _pNextIMU = pNext;
00106 }
00107
00108 int IMU::BeginRead()
00109 {
00110     if (_State == sWait) {
00111         Run();
00112     } else {
00113         if (_pNextIMU) {
00114             return _pNextIMU->BeginRead();
00115         }
00116     }
00117     return 0;
00118 }
00119
00120 void IMU::QueryChannels()
00121 {
00122     _numChans = 0;
00123     _bDualChan = 0;
00124     // Check the first, lower ID.
00125     int retc = _pMas->CheckID(0xD2);
00126     if (retc == 0) {
00127         _gID[_numChans] = 0xD2;
00128         _aID[_numChans] = 0x32; // Always high bit.
00129         _numChans++;
00130     }
00131     retc = _pMas->CheckID(0xD0);
00132     if (retc == 0) {
00133         _gID[_numChans] = 0xD0;
00134         _aID[_numChans] = 0x30; // Always low bit.
00135         _numChans++;
00136     }
00137
00138     if (_numChans > 1) {
00139         _bDualChan = true;
00140     }
00141 }
00142
```

```
00143 void IMU::SetDebugPort(DebugPort* pPort)
00144 {
00145     _pDBGPort = pPort;
00146 }
00147
00148 void IMU::SetDebugPort2(DebugPort* pPort)
00149 {
00150     _pDBGPort2 = pPort;
00151 }
00152
00153 void IMU::Reset()
00154 {
00155     _pMas->end();
00156     _pMas->begin(400e3);
00157     _pMas->NotifyMe(this);
00158 }
00159
00160 void IMU::ResetDevices()
00161 {
00162     _pMas->Stop();
00163 }
00164
00165 int IMU::ForceStartStop()
00166 {
00167     return _pMas->ForceStartStop();
00168 }
00169
00170 void IMU::SampleRate(uint16_t rate)
00171 {
00172     // Range Limit the rate.
00173     if (rate < 10) {
00174         _Rate = 10;
00175     } else if (rate > 200) {
00176         _Rate = 200;
00177     } else {
00178         _Rate = rate;
00179     }
00180
00181     uint16_t gval = 1000/rate;
00182     gval = gval - 1;
00183
00184     if (pdbgserial) pdbgserial->print("Set Rate on IMU\n");
00185
00186     SetTimerPeriod();
00187 }
00188
00194 int IMU::Setup()
00195 {
00196     if (_numChans == 0) return 0;
00197
00198     _bRun = false;
00199     Mark marker(_pDBGPort2,pSetup);
00200
00201     if (_State != sIdle) {
00202         if (pdbgserial)
00203             pdbgserial->print("IMU Already Running.\n");
00204         return 0; // Already running
00205     }
00206
00207     ResetFailStats(); // inline in header
00208     SetState(sConfigure); // Inline in header
00209
```

```
00210     // Start the process with Configure
00211     for (int x=0; x<_numChans;x++) {
00212         if (pdbgserial)
00213             pdbgserial->print("Configuring IMU\n");
00214         int retc = Configure(x);
00215         if (retc < 0 ) {
00216             // Try reset mechanisms - none of which have
00217             // been found that work properly yet.
00218             SetState(sIdle);
00219             if (pdbgserial)
00220                 pdbgserial->print("IMU Configure Failed.");
00221             return retc;
00222         }
00223     }
00224     SetState(sConfigured);
00225     if (pdbgserial)
00226         pdbgserial->print("IMU Configured.\n");
00227 
00228     return 0;
00229 }
00230 
00239 int IMU::Start()
00240 {
00241     if (_numChans == 0) return 0;
00242 
00243     Mark marker(_pDBGPort2,pWriteDn);
00244 
00245     int retc;
00246     if (_State != sConfigured) {
00247         retc = Setup();
00248         if (retc < 0) {
00249             return retc;
00250         }
00251     }
00252 
00253     cli();
00254     ResetTimer();
00255     SetState(sWait);
00256     ResetFailStats();
00257     _bDataReady[0] = false;
00258     _bDataReady[1] = false;
00259     _bRun = true;
00260     sei();
00261     return 0;
00262 }
00263 
00268 int IMU::Stop()
00269 {
00270     cli();
00271     _bRun = false;
00272     SetState(sIdle);
00273     sei();
00274     return 0;
00275 }
00276 
00277 bool IMU::Busy()
00278 {
00279     return _State != sIdle;
00280 }
00281 
00290 void IMU::Run()
00291 {
```

```
00292     if (_bRun == false) return;
00293
00294     Mark marker(_pDBGPort2,pRun);
00295
00300     if (_pMas->busy() && BusyTimeout()) {
00301         Reset();
00302         SetState(sWait);
00303     }
00304
00309     switch(_State) {
00310     case sWait:
00311         if (_bUseGyro) {
00312             SetState(sReadGyro1);
00313         } else {
00314             SetState(sReadAcc1);
00315         }
00316         StartTransaction();
00317         break;
00318     default:
00319         break;
00320     }
00321 }
00322
00323 int IMU::StartTransaction()
00324 {
00325     int retc = 0;
00326     switch(_State) {
00327     case sReadGyro1:
00328         RdAsync(_gID[0], 0x1B, 8);
00329         break;
00330     case sReadAcc1:
00331         RdAsync(_aID[0], 0x80 | 0x28, 6);
00332         break;
00333     case sReadGyro2:
00334         RdAsync(_gID[1], 0x1B, 8);
00335         break;
00336     case sReadAcc2:
00337         RdAsync(_aID[1], 0x80 | 0x28, 6);
00338         break;
00339     default:
00340         break;
00341     }
00342     ResetBusyTime();
00343     return retc;
00344 }
00345
00348 void IMU::ProcessTransaction()
00349 {
00350     switch(_State) {
00351         case sReadGyro1:
00352             StoreGyroData(1);
00353             SetState(sReadAcc1);
00354             break;
00355         case sReadAcc1:
00356             StoreAccData(1);
00357             PushData(1);
00358             if (_bDualChan) {
00359                 if (_bUseGyro) {
00360                     SetState(sReadGyro2);
00361                 } else {
00362                     SetState(sReadAcc2);
00363                 }
```

```
00364             } else {
00365                 SetState(sWait);
00366                 if (_pNextIMU) {
00367                     _pNextIMU->BeginRead();
00368                 }
00369             }
00370             break;
00371         case sReadGyro2:
00372             StoreGyroData(2);
00373             SetState(sReadAcc2);
00374             break;
00375         case sReadAcc2:
00376             StoreAccData(2);
00377             PushData(2);
00378             SetState(sWait);
00379             if (_pNextIMU) {
00380                 _pNextIMU->BeginRead();
00381             }
00382             break;
00383         default:
00384             break;
00385     }
00386 
00388     StartTransaction();
00389 }
00390 
00398 void IMU::FailRecovery()
00399 {
00400     switch(_failType) {
00401     case fNone:
00402         break;
00403     case fNack:
00404         if (_nackCount <7) {
00405             _pMas->WigglePin(10, 0,1);
00406         } else if (_nackCount < 10) {
00407             Reset();
00408             _pMas->WigglePin(10,0,1);
00409         }
00410         SetState(_previousState);
00411         break;
00412     case fBusErr:
00413         if (_failCount < 5) {
00414             Reset();
00415         }
00416         SetState(_previousState);
00417         break;
00418     case fArbLost:
00419         if (_failCount < 5) {
00420             Reset();
00421         }
00422         SetState(_previousState);
00423         break;
00424     }
00425 
00426     StartTransaction();
00427 }
00428 
00432 void IMU::I2CWriteDone()
00433 {
00434     if (_bRun == false) return;
00435 
00436     Mark marker(_pDBGPort2,pWriteDn);
```

```
00437     ResetBusyTime();
00438
00439     ResetFailStats();
00440     ProcessTransaction();
00441 }
00442
00446 void IMU::I2CReadDone()
00447 {
00448     if (_bRun == false) return;
00449
00450     Mark marker(_pDBGPort2,pReadDn);
00451     ResetBusyTime();
00452
00453     ResetFailStats();
00454     ProcessTransaction();
00455 }
00456
00463 void IMU::I2CNack()
00464 {
00465     if (_bRun == false) return;
00466
00467     ResetBusyTime();
00468     {
00469         Mark marker(_pDBGPort2,pBusErr);
00470         _delay_us(3);
00471     }
00472     Mark marker(_pDBGPort2,pNack);
00473
00474     ++_nackCount;
00475
00479     SetState(sErrRecover);
00480     _delay_us(5);
00481
00483     if (_nackCount < 5) {
00485         SetState(_previousState);
00486         StartTransaction();
00487     } else if (_nackCount < 10) {
00488         _failType = fNack;
00489         FailRecovery();
00490     } else {
00492         Stop();
00493     }
00494 }
00495
00499 void IMU::I2CBusError()
00500 {
00501     if (_bRun == false) return;
00502
00503     ResetBusyTime();
00504     {
00505         Mark marker(_pDBGPort2,pBusErr);
00506         _delay_us(3);
00507     }
00508     Mark marker(_pDBGPort2,pBusErr);
00512
00513     _bFailDetected = true;
00514     ++_failCount;
00515     _failType = fBusErr;
00516
00520     SetState(sErrRecover);
00521     _delay_us(5);
00522     if (_failCount > 10) {
```

```
00523         FailRecovery();
00524     } else {
00525         SetState(_previousState);
00526         StartTransaction();
00527     }
00528 }
00529
00534 void IMU::I2CArbLost()
00535 {
00536     if (_bRun == false) return;
00537
00538     ResetBusyTime();
00539     {
00540         Mark marker(_pDBGPort2,pBusErr);
00541         _delay_us(3);
00542     }
00543     Mark marker(_pDBGPort2,pArbLost);
00547     _bFailDetected = true;
00548     ++_failCount;
00549     _failType = fArbLost;
00550
00554     SetState(sErrRecover);
00555     _delay_us(5);
00556     if (_failCount > 10) {
00557         FailRecovery();
00558     } else {
00559         SetState(_previousState);
00560         StartTransaction();
00561     }
00562 }
00563
00564 bool IMU::DataReady()
00565 {
00566     if (_numChans == 0) return true;
00567
00568     bool bReady = false;
00569     cli();
00570     if (_bDualChan) {
00571         bReady =  _bDataReady[0] && _bDataReady[1];
00572     } else {
00573         bReady =  _bDataReady[0];
00574     }
00575     sei();
00576     return bReady;
00577 }
00578
00581 void IMU::StoreGyroData(uint8_t idx)
00582 {
00583     _pMas->ReadData(&_dataBuffer[idx-1][0],8);
00584 }
00585
00588 void IMU::StoreAccData(uint8_t idx)
00589 {
00590     _pMas->ReadData(&_dataBuffer[idx-1][8],6);
00591 }
00592
00594 void IMU::PushData(uint8_t idx)
00595 {
00596     _bDataReady[idx-1] = true;
00597 }
00598
00603 uint8_t* IMU::GetPacketData(uint8_t* pData)
```

```
00604 {
00605     if (_numChans == 0) return pData;
00606 
00607     cli();
00608 
00609     *pData++ = 0xa5;
00610     *pData++ = 0x5a;
00611     if (_State == sIdle) {
00612         if (_failType == fNack) {
00613             memset(pData,'N',IMUPacket::PacketLen);
00614         } else {
00615             memset(pData,'I',IMUPacket::PacketLen);
00616         }
00617         pData += IMUPacket::PacketLen;
00618     } else if (_bDataReady[0] == true) {
00619         memcpy(pData,&_dataBuffer[0][0],IMUPacket::PacketLen);
00620         _bDataReady[0] = false;
00621         pData += IMUPacket::PacketLen;
00622     } else {
00623         memset(pData,0,IMUPacket::PacketLen);
00624         pData += IMUPacket::PacketLen;
00625     }
00626 
00627     if (_bDualChan) {
00628         *pData++ = 0xa5;
00629         *pData++ = 0x5a;
00630         if (_State == sIdle) {
00631             if (_failType == fNack) {
00632                 memset(pData,'N',IMUPacket::PacketLen);
00633             } else {
00634                 memset(pData,'I',IMUPacket::PacketLen);
00635             }
00636             pData += IMUPacket::PacketLen;
00637         } else if (_bDataReady[1] == true) {
00638             memcpy(pData,&_dataBuffer[1][0],IMUPacket::PacketLen);
00639             _bDataReady[1] = false;
00640             pData += IMUPacket::PacketLen;
00641         } else {
00642             memset(pData,0,IMUPacket::PacketLen);
00643             pData += IMUPacket::PacketLen;
00644         }
00645     }
00646     sei();
00647     return pData;
00648 }
00649 
00650 // Diagnostic Routines
00651 void IMU::CheckIDs(HardwareSerial* pSerial)
00652 {
00653     char buffer[50];
00654     for (int x=0;x<_numChans;x++) {
00655         int retc = _pMas->CheckID(_gID[x]);
00656         if (retc == 0) {
00657             sprintf(buffer,"Gyro%d (0x%x):Ack.\n",x,_gID[x]);
00658             pSerial->print(buffer);
00659         } else {
00660             sprintf(buffer,"Gyro%d (0x%x):NAck (%d).\n",x,_gID[x],retc);
00661             pSerial->print(buffer);
00662         }
00663         Wr(_gID[x], 0x3D, 0x8);
00664         retc = _pMas->CheckID(_aID[x]);
00665         if (retc == 0) {
```

```
00666             sprintf(buffer,"Acc%d (0x%x):Ack.\n",x,_aID[x]);
00667             pSerial->print(buffer);
00668         } else {
00669             sprintf(buffer,"Acc%d (0x%x):NAck (%d).\n",x,_aID[x],retc);
00670             pSerial->print(buffer);
00671         }
00672     }
00673 }
00674
00685 void IMU::SetTimer(TimerCntr* pTimer)
00686 {
00687     _pTimer = pTimer;
00688
00690     _pTimer->ClkSel(TC_CLKSEL_DIV64_gc);
00691     SetTimerPeriod();
00692     _pTimer->CCEnable(0);
00693     _pTimer->WaveformGenMode(TC_WGMODE_NORMAL_gc);
00694     _pTimer->EventSetup(TC_EVACT_OFF_gc,TC_EVSEL_OFF_gc);
00695     _pTimer->IntLvlA(0,1);
00696     _pTimer->IntLvlB(0);
00697     _pTimer->Notify(this,0);
00698 }
00699
00702 void IMU::ResetTimer()
00703 {
00704     if (_pTimer) _pTimer->Counter(0);
00705 }
00706
00707 void IMU::SetTimerPeriod()
00708 {
00709     // Adjust the timer function to fire 5X faster
00710     // than the rate. At 200Hz, this will be 2Khz or
00711     // every 500us.
00712     // We set the timer to go off 5 times per IMU period.
00713     // This should range from 20ms for 10Hz, and 1 ms for 200
00714     // **** NoFifo
00715     // Set timer to fire at the rate.
00716     //unsigned long timerTicks = 100000/_Rate;
00717     unsigned long timerTicks = 500000/_Rate;
00718     if (timerTicks > 65000) {
00719         timerTicks = 65000;
00720     }
00721     if (_pTimer) _pTimer->Period(timerTicks);
00722 }
00723
00736
00738 void IMU::err(uint8_t id)
00739 {
00740 }
00741
00746 void IMU::ovf(uint8_t id)
00747 {
00748     Run();
00749 }
00750
00752 void IMU::ccx(uint8_t id,uint8_t idx)
00753 {
00754 }
00755
00757
00758 int IMU::Wr(uint8_t ID, uint8_t addr, uint8_t data)
00759 {
```

```
00760     static uint8_t  bytes[2];
00761     bytes[0] = addr;
00762     bytes[1] = data;
00763     return _pMas->WriteSync(ID, &bytes[0],2);
00764 }
00765
00766 int IMU::Rd(uint8_t ID, uint8_t addr, uint8_t cnt, uint8_t* pData)
00767 {
00768     // Only a single write, the address, then read data.
00769     int retc = _pMas->WriteReadSync(ID, &addr, 1, cnt);
00770     if ( retc < 0 ) {
00771         return retc;
00772     }
00773     return _pMas->ReadData(pData,cnt);
00774 }
00775
00776 int IMU::WrAsync(uint8_t ID, uint8_t addr, uint8_t data)
00777 {
00778     static uint8_t  bytes[2];
00779     bytes[0] = addr;
00780     bytes[1] = data;
00781     sprintf(buffer,"WrAsync to %d\n",ID);
00782     if (pdbgserial) pdbgserial->print(buffer);
00783     return _pMas->WriteRead(ID, &bytes[0],2,0);
00784 }
00785
00786 int IMU::RdAsync(uint8_t ID, uint8_t addr, uint8_t cnt)
00787 {
00788     // Only a single write, the address, then read data.
00789     return _pMas->WriteRead(ID, &addr, 1, cnt);
00790 }
00791
00792 void IMU::ReadWord(uint16_t* pData)
00793 {
00794     static uint8_t buffer[2];
00795     _pMas->ReadData(&buffer[0],2);
00796     *pData = (buffer[0] << 8 | buffer[1]);
00797 }
00798
00803 int IMU::Configure(uint8_t idx)
00804 {
00805     // Value for the sensor register
00806     uint16_t gval = 1000/_Rate;
00807     gval = gval - 1;
00808
00809     int retc;
00810
00811     RegWriteType    config[] = {
00812     //    // Turn on pass-through
00813         { _gID[idx], 0x3D, 0x0F },
00814     //
00815     //    // Init the Accelerometer.
00816         { _aID[idx], 0x20, 0x37},
00817         { _aID[idx], 0x21, 0x0},
00818         { _aID[idx], 0x22, 0x0},
00819         { _aID[idx], 0x23, 0x80 | 0x40},
00820         { _aID[idx], 0x24, 0x00},
00821     //
00822     //    // Set offsets to zero
00823         { _gID[idx], 0x0C, 0x00},
00824         { _gID[idx], 0x0D, 0x00},
00825         { _gID[idx], 0x0E, 0x00},
```

```
00826         { _gID[idx], 0x0F, 0x00},
00827         { _gID[idx], 0x10, 0x00},
00828         { _gID[idx], 0x11, 0x00},
00829     //
00830     //    // Configure registers.
00831         { _gID[idx], 0x12, 0xff},                    // Enable all outputs to to
      the fifo
00832         { _gID[idx], 0x13, 0x00},
00833     //    { _gID[idx], 0x14, _aID[idx] >> 1},             // Set slave address of
      ACC
00834         { _gID[idx], 0x15, gval},                    // Set sample rate
00835         { _gID[idx], 0x16, _DLPF | _FullScale << 3},
00836         { _gID[idx], 0x17, 0x00},
00837     //    { _gID[idx], 0x18, 0x80 | 0x28},              // Set burst address for
     Accelerometer, enable auto addr increment.
00838         { _gID[idx], 0x3E, _ClkSel},
00839     };
00840
00841     uint8_t nItems = sizeof(config)/sizeof(RegWriteType);
00842     for (int idx = 0;idx <nItems;idx++) {
00843         retc = Wr(config[idx].ID,
00844             config[idx].Addr,
00845             config[idx].Data);
00846         ResetBusyTime();
00847
00848         if (retc < 0) {
00849             return retc; // _configOkay will be false;
00850         }
00851     }
00852
00853     return 0;
00854 }
00855
00856
00857
```

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

```
00001
00002 #ifndef IMU_h
00003 #define IMU_h
00004
00005 #include <stdlib.h>
00006 #include <inttypes.h>
00007 #include <avr/io.h>
00008 #include <util/delay.h>
00009 #include "I2C_Master.h"
00010 #include "HardwareSerial.h"
00011 #include "IMUPacketFifo.h"
00012 #include "TimerCntr.h"
00013 #include "TimerCore.h"
00014 #include "DebugPort.h"
00015
00016 class IMUBase
00017 {
00018 public:
00019     virtual void Reset() = 0;
00020     virtual void SampleRate(uint16_t) = 0;
00021
00022     virtual int Setup() = 0;
00023     virtual int Start() = 0;
00024     virtual int Stop() = 0;
00025     virtual int ForceStartStop() = 0;
00026     virtual bool Busy() = 0;
00027     virtual void ResetTimer() = 0;
00028     virtual void UseGyro(bool bEnable) = 0;
00029     virtual void NextIMU(IMUBase*) = 0;
00030     virtual int BeginRead() = 0;
00031
00032     virtual bool DataReady() = 0;
00033     virtual uint8_t* GetPacketData(uint8_t* pPacket) = 0;
00034
00035     // Diags
00036     virtual void CheckIDs(HardwareSerial* pSerial) = 0;
00037     virtual void ResetDevices() = 0;
00038 };
00039
00040 class IMU : public IMUBase, public I2CNotify, public TimerNotify
00041 {
00042     typedef enum StateType {
00043         sIdle           = 0,
00044         sConfigure      = 1,
00045         sConfigured     = 2,
00046         sWait           = 5,
00047         sReadGyro1      = 8,
00048         sReadAcc1       = 9,
00049         sReadGyro2      = 10,
00050         sReadAcc2       = 11,
00051         sErrRecover     = 12
00052     } StateType;
```

```
00053
00054     typedef enum PosType {
00055         pStart          = 0,
00056         pRun            = 1,
00057         pWriteDn        = 2,
00058         pReadDn         = 3,
00059         pNack           = 4,
00060         pBusErr         = 5,
00061         pArbLost        = 6,
00062         pSetup          = 7
00063     } PosType;
00064
00065     typedef enum FailType {
00066         fNone           = 0,
00067         fNack           = 1,
00068         fBusErr         = 2,
00069         fArbLost        = 3
00070     } FailType;
00071
00072     typedef struct regWrite {
00073         uint8_t     ID;
00074         uint8_t     Addr;
00075         uint8_t     Data;
00076     } RegWriteType;
00077
00078     typedef enum  {
00079         ptTimer,
00080         ptI2CWrite,
00081         ptI2CRead,
00082         ptI2CNack
00083     } ProcessType;
00084
00085     StateType       _State;
00086     StateType       _previousState; // Used for error recovery.
00087     FailType        _failType;
00088
00089     I2C_Master*     _pMas;
00090     bool            _bDualChan;
00091     uint8_t         _numChans;
00092     bool            _configOkay[2];
00093     uint8_t         _gID[2];
00094     uint8_t         _aID[2];
00095     uint8_t         _DLPF;
00096     uint8_t         _FullScale;
00097     uint8_t         _ClkSel;
00098     uint16_t        _Rate;
00099
00100     bool            _bUseGyro;
00101
00103     uint8_t         _dataBuffer[2][20];
00104     bool            _bDataReady[2];
00105
00106     TimerCntr*      _pTimer;
00107     bool            _bRun;
00108     uint16_t        _failCount;
00109     uint8_t         _nackCount;
00110     bool            _bFailDetected;
00111     unsigned int    _busyWaitTime;
00112     DebugPort*      _pDBGPort;
00113     DebugPort*      _pDBGPort2;
00114     IMUBase*        _pNextIMU;
00115
```

```
00116 public:
00117     IMU(I2C_Master* pMas);
00118     IMU(I2C_Master* pMas, uint8_t gID, uint8_t aID);
00119     IMU(I2C_Master* pMas, uint8_t gID, uint8_t aID,
00120         uint8_t gID2, uint8_t aID2
00121         );
00122
00123
00124     void QueryChannels();
00125     void SetDebugPort(DebugPort* pPort);
00126     void SetDebugPort2(DebugPort* pPort);
00127
00128     virtual void Reset();
00129     virtual void SampleRate(uint16_t);
00130
00131     virtual int Setup();
00132     virtual int Start();
00133     virtual int Stop();
00134     virtual int ForceStartStop();
00135     virtual bool Busy();
00136     virtual void ResetTimer();
00137     virtual void UseGyro(bool bEnable) { _bUseGyro = bEnable; };
00138     virtual void NextIMU(IMUBase* pNext);
00139     virtual int BeginRead();
00140
00141     virtual bool DataReady();
00142     virtual uint8_t* GetPacketData(uint8_t*);
00143
00144     // Diags
00145     virtual void CheckIDs(HardwareSerial* pSerial);
00146     virtual void ResetDevices();
00147
00148     // Timer Notification
00149     void SetTimer(TimerCntr* pTimer);
00150     void SetTimerPeriod();
00151     virtual void err(uint8_t id);
00152     virtual void ovf(uint8_t id);
00153     virtual void ccx(uint8_t id, uint8_t idx);
00154
00155     // I2C
00156     virtual void I2CWriteDone();
00157     virtual void I2CReadDone();
00158     virtual void I2CBusError();
00159     virtual void I2CArbLost();
00160     virtual void I2CNack();
00161
00162     void FailRecovery();
00163
00164 protected:
00165     void Run();
00166     int StartTransaction();
00167     void ProcessTransaction();
00168     int Configure(uint8_t idx);
00169     int Wr(uint8_t ID, uint8_t addr, uint8_t data);
00170     int Rd(uint8_t ID, uint8_t addr, uint8_t cnt, uint8_t* pData);
00171     int WrAsync(uint8_t ID, uint8_t addr, uint8_t data);
00172     int RdAsync(uint8_t ID, uint8_t addr, uint8_t cnt);
00173     void ReadWord(uint16_t *pData);
00174     void StoreGyroData(uint8_t idx);
00175     void StoreAccData(uint8_t idx);
00176     void PushData(uint8_t idx);
00177
```

```
00178     inline void SetState(StateType s)
00179     {
00180         _previousState = _State;
00181         _State = s;
00182         if (_pDBGPort) _pDBGPort->SetState((uint8_t)_State);
00183     }
00184 
00185     inline void MarkPos(PosType p)
00186     {
00187         if (_pDBGPort2) _pDBGPort2->SetState((uint8_t) p);
00188     }
00189 
00190     inline void ResetBusyTime()
00191     {
00192         _busyWaitTime = millis();
00193     }
00194 
00195     inline bool BusyTimeout()
00196     {
00197         return ((millis() - _busyWaitTime) > 2);
00198     }
00199 
00200     inline void ResetFailStats()
00201     {
00202         _bFailDetected      = false;
00203         _nackCount          = 0;
00204         _failCount          = 0;
00205         _failType           = fNone;
00206     }
00207 
00208 };
00209 
00210 
00211 #endif
00212 
```

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) \
```

```
00019     for (int x=0;x<4;x++) { \
00020         if (_pIMU[x]) {         \
00021             _pIMU[x]->func();   \
00022         }   \
00023     }
00024
00025 #define ALL_IMUP(func,param) \
00026     for (int x=0;x<4;x++) { \
00027         if (_pIMU[x]) {         \
00028             _pIMU[x]->func(param);   \
00029         }   \
00030     }
00031
00032 #define ALL_IMURET(func) \
00033     int retc =0; \
00034     for (int x=0;x<4;x++) { \
00035         if (_pIMU[x]) {         \
00036             retc = _pIMU[x]->func();   \
00037             if (retc < 0) { \
00038                 return retc;\
00039             }\
00040         }   \
00041     }\
00042     return retc;
00043
00044 #define ALL_IMUBOOL(func) \
00045     bool bReady = true; \
00046     for (int x = 0;x<4;x++) {   \
00047         if (_pIMU[x]) { \
00048             if (!_pIMU[x]->func()) { \
00049                 bReady = false; \
00050             }   \
00051         }   \
00052     }   \
00053     return bReady;
00054
00055 IMUManager::IMUManager(HardwareSerial* pSerial)
00056 {
00057     int x;
00058     for (x =0;x<4;x++) {
00059         _pIMU[x] = 0;
00060     }
00061
00062     _nIMUs          = 0;
00063     _State          = sIdle;
00064     _nStreamWDCounter = 0;
00065     _packetId       = 0;
00066     _pBlueLed       = 0;
00067     _bLedState      = false;
00068
00069     _pTimer         = 0;
00070     _pSerial        = pSerial;
00071     _pDBGPort       = 0;
00072     _sampleRate     = 10;
00073     _maxMillisPerPacket = 1000;
00074     _lastSendMillis = 0;
00075 }
00076
00077 void IMUManager::SetBlueLed(PORT_t* port, uint8_t Pin)
00078 {
00079     _pBlueLed   = port;
00080     _LedPin     = Pin;
```

```
00081
00082     _pBlueLed->DIRSET = _LedPin;
00083     _pBlueLed->OUTSET = _LedPin;
00084     _bLedState        = false;
00085 }
00086
00087 void IMUManager::LedOn()
00088 {
00089     _pBlueLed->OUTCLR   = _LedPin;
00090     _bLedState          = true;
00091 }
00092
00093 void IMUManager::LedOff()
00094 {
00095     _pBlueLed->OUTSET   = _LedPin;
00096     _bLedState          = false;
00097 }
00098
00104 void IMUManager::PacketLedIndicator()
00105 {
00106     if ((_packetId % 6) == 0) {
00107         ToggleLed();
00108     }
00109 }
00110
00112 void IMUManager::ToggleLed()
00113 {
00114     if ( _bLedState) {
00115         _bLedState = false;
00116         _pBlueLed->OUTCLR = _LedPin;
00117     } else {
00118         _bLedState = true;
00119         _pBlueLed->OUTSET = _LedPin;
00120     }
00121 }
00122
00124 void IMUManager::ShowLedStart()
00125 {
00126     _pBlueLed->OUTCLR = _LedPin;
00127     _delay_ms(10*17); // One Frame
00128     _pBlueLed->OUTSET = _LedPin;
00129     _delay_ms(5*17); // One Frame
00130     _pBlueLed->OUTCLR = _LedPin;
00131     _delay_ms(10*17); // One Frame
00132     _pBlueLed->OUTSET = _LedPin;
00133     _delay_ms(17); // One Frame
00134     _bLedState        = false;
00135 }
00136
00138 void IMUManager::ShowLedStop()
00139 {
00140     _pBlueLed->OUTCLR = _LedPin;
00141     _delay_ms(5*17); // One Frame
00142     _pBlueLed->OUTSET = _LedPin;
00143     _delay_ms(5*17); // One Frame
00144     _pBlueLed->OUTCLR = _LedPin;
00145     _delay_ms(5*17); // One Frame
00146     _pBlueLed->OUTSET = _LedPin;
00147     _delay_ms(5*17); // One Frame
00148     _pBlueLed->OUTCLR = _LedPin;
00149     _bLedState        = false;
00150 }
```

```
00151
00152 void IMUManager::SampleRate(uint16_t rate)
00153 {
00154     // Range Limit the rate.
00155     if (rate < 10) {
00156         _sampleRate = 10;
00157     } else if (rate > 200) {
00158         _sampleRate = 200;
00159     } else {
00160         _sampleRate = rate;
00161     }
00162
00163     ALL_IMUP(SampleRate,_sampleRate);
00164
00165
00166     if (pdbgserial) {
00167         char buffer[50];
00168         sprintf(buffer,"Sample rate set to %d\n", _sampleRate);
00169         pdbgserial->print(buffer);
00170     }
00171
00172     // Note: I had this set to 2X the sample rate (in milliseconds),
00173     // but that was a problem. The IMU's wait until there is 2X the
00174     // amount of data in the fifo, so that takes 2X the time, and we
00175     // would time out first time around.. bad idea.
00176     _maxMillisPerPacket = 3 * 1000/_sampleRate;
00177
00178     SetTimerPeriod();
00179 }
00180
00181 int IMUManager::AddIMU(IMUBase* pIMU)
00182 {
00183     for (int x=0;x<4;x++) {
00184         if (_pIMU[x] == 0) {
00185             // Add into first empty spot
00186             _pIMU[x] = pIMU;
00187             if (x > 0) {
00188                 _pIMU[x-1]->NextIMU(pIMU);
00189             }
00190             _nIMUs++;
00191             return x;
00192         }
00193     }
00194
00195     // Seems we are full!!
00196     return -1;
00197 }
00198
00199 int IMUManager::Setup()
00200 {
00201     ALL_IMURET(Setup);
00202 }
00203
00204 int IMUManager::Start()
00205 {
00206     ALL_IMURET(Start);
00207 }
00208
00209 void IMUManager::Stop()
00210 {
00211     ALL_IMU(Stop);
00212     _State = sIdle;
```

```
00213     ShowLedStop();
00214     LedOff();
00215 }
00216
00217 void IMUManager::Reset()
00218 {
00219     ALL_IMU(Reset);
00220
00221     _State = sIdle;
00222     _packetId       = 0;
00223 }
00224
00225 void IMUManager::ResetDevices()
00226 {
00227     ALL_IMU(ResetDevices);
00228 }
00229
00230 void IMUManager::ForceStartStop()
00231 {
00232     ALL_IMU(ForceStartStop);
00233 }
00234
00235 // Diagnostic Routines
00236 void IMUManager::CheckIDs(HardwareSerial* pSerial,int idx)
00237 {
00238     if (idx < 0) {
00239         ALL_IMUP(CheckIDs,pSerial);
00240     } else {
00241         if (_pIMU[idx]) {
00242             _pIMU[idx]->CheckIDs(pSerial);
00243         }
00244     }
00245 }
00246
00248 bool IMUManager::DataReady()
00249 {
00250     ALL_IMUBOOL(DataReady)
00251 }
00252
00253 void IMUManager::IMUUseGyro(bool bEn)
00254 {
00255     ALL_IMUP(UseGyro,bEn);
00256 }
00257
00264 int IMUManager::StreamStart(bool bUseGyro)
00265 {
00266     if (_State != sIdle ) {
00267         Stop();
00268     }
00269
00270     IMUUseGyro(bUseGyro);
00271
00272     int retc = Setup();
00273     if (retc < 0) {
00274         return retc;
00275     }
00276
00277     retc = Start();
00278
00279     if (retc < 0) {
00280         return retc;
00281     }
```

```
00282
00283     ShowLedStart();
00284     _LedCounter = 0;
00285     _packetId       = 0;
00286     _nStreamWDCounter = 20;
00287
00288     SetState(sDataWait); // Jump ahead a state.
00289     ResetDataReadyTO();
00290     return 0;
00291 }
00292
00293 void IMUManager::StreamWatchdog()
00294 {
00295     _nStreamWDCounter = 20;
00296 }
00297
00298 //
00299 //  Iterate over the embedded IMU objects, retrieve results
00300 //  as needed. Do this in a 2-pass fashion, so that we do
00301 //  IMU 1 on each interface first, then IMU2 on any interfaces
00302 //  that have 2 IMU devices on them. After this is all done,
00303 //  we should have all of the required IMU data, then we can
00304 //  initiate a packet send to the host with as much as six
00305 //  IMU's worth of data!
00306 //
00307 int IMUManager::Loop()
00308 {
00309     switch(_State) {
00310     case sIdle:
00311         break;
00312     case sDataWait:
00313         if (DataReady()) {
00314             ResetDataReadyTO();
00315             _State = sDataReady;
00316         } else if (DataReadyTimeout()) {
00317             ResetDataReadyTO();
00318             _State = sDataTimeout;
00319         }
00320         break;
00321     case sDataReady:
00322         PacketLedIndicator();
00323         if (_nStreamWDCounter == 0) {
00324             DiscardData();
00325             _State = sDataWait;
00326         } else {
00327             --_nStreamWDCounter;
00328             SendPacket(false);
00329             _State = sDataWait;
00330         }
00331         break;
00332     case sDataTimeout:
00333         PacketLedIndicator();
00334         if (_nStreamWDCounter == 0) {
00335             DiscardData();
00336             _State = sDataWait;
00337         } else {
00338             --_nStreamWDCounter;
00339             SendPacket(true);
00340             _State = sDataWait;
00341         }
00342         break;
00343     }
```

```
00344
00345     return 0;
00346 }
00347
00348 void IMUManager::Run()
00349 {
00350     if (_State != sIdle) {
00351         // Start the IMU's going one at a time...
00352         if (_pIMU[0]) {
00353             _pIMU[0]->BeginRead();
00354         }
00355     }
00356 }
00357
00366 void IMUManager::SendPacket(bool bTimeout)
00367 {
00368     uint8_t*    pPacket = &_dataPacket[0];
00369     if (true || !bTimeout) {
00370         for (int x = 0;x<4;x++) {
00371             if (_pIMU[x]) {
00372                 // This puts the data at the pointer,
00373                 // then returns the end of the data.
00374                 // This might be 2*14 or 1*14
00375                 pPacket = _pIMU[x]->GetPacketData(pPacket);
00376             }
00377         }
00378     }
00379     // Packet format:
00380     // SNP header
00381     // byte: length of packet
00382     // byte: packet type (0xB7)
00383     // byte(s): length bytes
00384     // bytes(2): 2 byte CRC
00385     // string: END
00386     // newline
00387     uint8_t size = pPacket - &_dataPacket[0];
00388     buffer[0] = 'S';
00389     buffer[1] = 'N';
00390     buffer[2] = 'P';
00391     buffer[3] = 0xB7;
00392     buffer[4] = _packetId++;
00393     buffer[5] = size;
00394     memcpy(&buffer[6],&_dataPacket[0],size);
00395     // Compute CRC -- someday
00396     uint16_t crc = 0xaf5a;
00397     uint8_t crchi = (crc >> 8) & 0xff;
00398     uint8_t crclo = crc & 0xff;
00399     buffer[6+size]   = _nStreamWDCounter;
00400     buffer[6+size+1]   = crchi;
00401     buffer[6+size+2] = crclo;
00402     sprintf((char*)&buffer[6+size+3],"END\n");
00403     _pSerial->write(&buffer[0],6+size+3+4);
00404 }
00405
00406 void IMUManager::DiscardData()
00407 {
00408     uint8_t*    pPacket = &_dataPacket[0];
00409     for (int x = 0;x<4;x++) {
00410         if (_pIMU[x]) {
00411             // This puts the data at the pointer,
00412             // then returns the end of the data.
00413             // This might be 2*14 or 1*14
```

```
00414             pPacket = _pIMU[x]->GetPacketData(pPacket);
00415         }
00416     }
00417 }
00418
00429 void IMUManager::SetTimer(TimerCntr* pTimer)
00430 {
00431     _pTimer = pTimer;
00432
00434     _pTimer->ClkSel(TC_CLKSEL_DIV64_gc);
00435     SetTimerPeriod();
00436     _pTimer->CCEnable(0);
00437     _pTimer->WaveformGenMode(TC_WGMODE_NORMAL_gc);
00438     _pTimer->EventSetup(TC_EVACT_OFF_gc,TC_EVSEL_OFF_gc);
00439     _pTimer->IntLvlA(0,1);
00440     _pTimer->IntLvlB(0);
00441     _pTimer->Notify(this,0);
00442 }
00443
00444 void IMUManager::SetTimerPeriod()
00445 {
00446     // Adjust the timer function to fire 5X faster
00447     // than the rate. At 200Hz, this will be 2Khz or
00448     // every 500us.
00449     // We set the timer to go off 5 times per IMU period.
00450     // This should range from 20ms for 10Hz, and 1 ms for 200
00451     // **** NoFifo
00452     // Set timer to fire at the rate.
00453     //unsigned long timerTicks = 100000/_Rate;
00454     unsigned int timerTicks = 500000/_sampleRate;
00455     if (timerTicks > 65000) {
00456         timerTicks = 65000;
00457     }
00458     // This is a special case. If I set the rate to 180,
00459     // then this will assume I am trying to sync with the camera
00460     // which has a frame rate of 59.94. 3X this is 179.82.
00461     // Setting this timer ticks value will put our IMU rate
00462     // close to 3x the frame rate of the camera, which is what
00463     // we want.
00464     if (_sampleRate == 180) {
00465         timerTicks = 2780;
00466     }
00467     if (pdbgserial) {
00468         char buffer[50];
00469         sprintf(buffer,"Timer Period:%u\n",timerTicks);
00470         pdbgserial->print(buffer);
00471     }
00472     if (_pTimer) _pTimer->Period(timerTicks);
00473 }
00474
00487
00489 void IMUManager::err(uint8_t id)
00490 {
00491 }
00492
00497 void IMUManager::ovf(uint8_t id)
00498 {
00499     Run();
00500 }
00501
00503 void IMUManager::ccx(uint8_t id,uint8_t idx)
00504 {
```

```
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

```
00001 #include <stdlib.h>
00002
00003 void * operator new(size_t size);
00004 void operator delete(void * ptr);
00005
```

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

```
00001
00002
00003 #include <avr/io.h>
00004 #include <inttypes.h>
00005 #include <avr/interrupt.h>
00006
00007 #include "Port.h"
00008 #include "HardwareSerial.h"
00009
00010 extern HardwareSerial* pdbgserial;
00011
00012 // Generate all of the ISR handlers.. hook them up to a class if/when a class
00013 // is instantiated for a particular USART.
00014 #define PORT_ISR_DEF(port) \
00015 static Port*  port##cp = 0;\
00016 ISR(port##_INT0_vect) {\
```

```
00017     if (port##cp) port##cp->int0();\
00018 }\
00019 ISR(port##_INT1_vect) {\
00020     if (port##cp) port##cp->int1();\
00021 }
00022
00023 PORT_ISR_DEF(PORTA);
00024 PORT_ISR_DEF(PORTB);
00025 PORT_ISR_DEF(PORTC);
00026 PORT_ISR_DEF(PORTD);
00027 PORT_ISR_DEF(PORTE);
00028 PORT_ISR_DEF(PORTF);
00029 #if defined (__AVR_ATxmega128A1__)
00030 PORT_ISR_DEF(PORTH);
00031 #endif
00032
00033 static void SetPointer(PORT_t* port,Port* p)
00034 {
00035     // Register this object with the appropriate
00036     // pointer so that the ISR routines can call p
00037     // class.
00038     if(port == &PORTA) {
00039         PORTAcp = p;
00040     } else if (port == &PORTB) {
00041         PORTBcp = p;
00042     } else if (port ==  &PORTC) {
00043         PORTCcp = p;
00044     } else if (port ==  &PORTD) {
00045         PORTDcp = p;
00046     } else if (port ==  &PORTE) {
00047         PORTEcp = p;
00048     } else if (port ==  &PORTF) {
00049         PORTFcp = p;
00050 #if defined (__AVR_ATxmega128A1__)
00051     } else if (port ==  &PORTH) {
00052         PORTHcp = p;
00053 #endif
00054     }
00055 }
00056
00057
00058 Port::Port(PORT_t* pPort)
00059 {
00060     _pPort = pPort;
00061     SetPointer(_pPort,this);
00062 }
00063
00064 Port::~Port()
00065 {
00066     SetPointer(_pPort,0);
00067 }
00068
00069 void Port::Notify(PortNotify* pClient,uint8_t id)
00070 {
00071     _pNotifyClient  = pClient;
00072     _pNotifyID      = id;
00073 }
00074
00075 void Port::int0()
00076 {
00077     if (_pNotifyClient) {
00078         _pNotifyClient->PortISR0(_pNotifyID);
```

```
00079     }
00080     _pPort->INTFLAGS = 0x1;
00081 }
00082
00083 void Port::int1()
00084 {
00085     if (_pNotifyClient) {
00086         _pNotifyClient->PortISR1(_pNotifyID);
00087     }
00088     _pPort->INTFLAGS = 0x2;
00089 }
00090
00091 void Port::SetDir(uint8_t dir)
00092 {
00093     _pPort->DIR = dir;
00094 }
00095
00096 void Port::SetPinsAsInput(uint8_t mask)
00097 {
00098     _pPort->DIRCLR = mask;
00099 }
00100
00101 void Port::SetPinsAsOutput(uint8_t mask)
00102 {
00103     _pPort->DIRSET = mask;
00104 }
00105
00106 void Port::SetPinsHigh(uint8_t mask)
00107 {
00108     _pPort->OUTSET = mask;
00109 }
00110
00111 void Port::SetPinsLow(uint8_t mask)
00112 {
00113     _pPort->OUTCLR = mask;
00114 }
00115
00116 uint8_t Port::GetPins()
00117 {
00118     return _pPort->IN;
00119 }
00120
00121 void Port::InterruptLevel(uint8_t num, uint8_t lvl)
00122 {
00123     if (num == 0) {
00124         _pPort->INTCTRL &= ~(0x3);
00125         _pPort->INTCTRL |= (lvl & 0x3);
00126     } else {
00127         _pPort->INTCTRL &= ~(0xC);
00128         _pPort->INTCTRL |= (lvl & 0x3) << 2;
00129     }
00130 }
00131
00132 void Port::InterruptMask(uint8_t num, uint8_t mask)
00133 {
00134     if (num == 0) {
00135         _pPort->INT0MASK = mask;
00136     } else {
00137         _pPort->INT1MASK = mask;
00138     }
00139 }
00140
```

```
00141 void Port::PinControl(uint8_t mask,
00142         bool bSlewLimit,
00143         bool bInverted,
00144         PORT_OPC_t OutputConfig,
00145         PORT_ISC_t InputSense
00146         )
00147 {
00154     PORTCFG.MPCMASK = mask;
00155     _pPort->PIN0CTRL =
00156         (bSlewLimit ? 0x80 : 0x0) |
00157         (bInverted ? 0x40 : 0x0)  |
00158         OutputConfig |
00159         InputSense
00160         ;
00161 }
00162
00163
```

6.39 Port.h File Reference

Classes

- class PortNotify
- class Port

6.40 Port.h

```
00001
00002 #ifndef Port_h
00003 #define Port_h
00004
00013 class PortNotify
00014 {
00015 public:
00016     virtual void PortISR0(uint8_t id) = 0;
00017     virtual void PortISR1(uint8_t id) = 0;
00018 };
00019
00026 class Port
00027 {
00028 protected:
00029     PORT_t*     _pPort;
00030     PortNotify* _pNotifyClient;
00031     uint8_t     _pNotifyID;
00032
00033 public:
00034
00035     Port(PORT_t*);
00036     ~Port();
00037
00038     void Notify(PortNotify* pClient,uint8_t id);
00039     void int0();
00040     void int1();
00041
00042     void SetDir(uint8_t dir);
00043     void SetPinsAsInput(uint8_t mask);
00044     void SetPinsAsOutput(uint8_t mask);
00045     void SetPinsHigh(uint8_t mask);
```

```
00046     void SetPinsLow(uint8_t mask);
00047     uint8_t GetPins();
00048
00050     void InterruptLevel(uint8_t num, uint8_t lvl);
00051     void InterruptMask(uint8_t num, uint8_t mask);
00052
00054     void PinControl(uint8_t mask,
00055         bool bSlewLimit,
00056         bool bInverted,
00057         PORT_OPC_t OutputConfig,
00058         PORT_ISC_t InputSense
00059         );
00060 };
00061
00062
00063 #endif
```

6.41 Print.cpp File Reference

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

6.42 Print.cpp

```
00001 /*
00002  Print.cpp - Base class that provides print() and println()
00003  Copyright (c) 2008 David A. Mellis.  All right reserved.
00004
00005  This library is free software; you can redistribute it and/or
00006  modify it under the terms of the GNU Lesser General Public
00007  License as published by the Free Software Foundation; either
00008  version 2.1 of the License, or (at your option) any later version.
00009
00010  This library is distributed in the hope that it will be useful,
00011  but WITHOUT ANY WARRANTY; without even the implied warranty of
00012  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
00013  Lesser General Public License for more details.
00014
00015  You should have received a copy of the GNU Lesser General Public
00016  License along with this library; if not, write to the Free Software
00017  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
00018
00019  Modified 23 November 2006 by David A. Mellis
00020  */
00021
00022 #include <stdio.h>
00023 #include <string.h>
00024 #include <math.h>
00025 //#include "wiring.h"
00026
00027 #include "Print.h"
00028
00029 // Public Methods //////////////////////////////////////////////////////////////
00030
```

```
00031 /* default implementation: may be overridden */
00032 void Print::write(const char *str)
00033 {
00034   while (*str)
00035     write(*str++);
00036 }
00037
00038 /* default implementation: may be overridden */
00039 void Print::write(const uint8_t *buffer, size_t size)
00040 {
00041   while (size--)
00042     write(*buffer++);
00043 }
00044
00045 void Print::print(const char str[])
00046 {
00047   write(str);
00048 }
00049
00050 void Print::print(char c, int base)
00051 {
00052   print((long) c, base);
00053 }
00054
00055 void Print::print(unsigned char b, int base)
00056 {
00057   print((unsigned long) b, base);
00058 }
00059
00060 void Print::print(int n, int base)
00061 {
00062   print((long) n, base);
00063 }
00064
00065 void Print::print(unsigned int n, int base)
00066 {
00067   print((unsigned long) n, base);
00068 }
00069
00070 void Print::print(long n, int base)
00071 {
00072   if (base == 0) {
00073     write(n);
00074   } else if (base == 10) {
00075     if (n < 0) {
00076       print('-');
00077       n = -n;
00078     }
00079     printNumber(n, 10);
00080   } else {
00081     printNumber(n, base);
00082   }
00083 }
00084
00085 void Print::print(unsigned long n, int base)
00086 {
00087   if (base == 0) write(n);
00088   else printNumber(n, base);
00089 }
00090
00091 void Print::print(double n, int digits)
00092 {
```

```
00093   printFloat(n, digits);
00094 }
00095
00096 void Print::println(void)
00097 {
00098   print('\r');
00099   print('\n');
00100 }
00101
00102 void Print::println(const char c[])
00103 {
00104   print(c);
00105   println();
00106 }
00107
00108 void Print::println(char c, int base)
00109 {
00110   print(c, base);
00111   println();
00112 }
00113
00114 void Print::println(unsigned char b, int base)
00115 {
00116   print(b, base);
00117   println();
00118 }
00119
00120 void Print::println(int n, int base)
00121 {
00122   print(n, base);
00123   println();
00124 }
00125
00126 void Print::println(unsigned int n, int base)
00127 {
00128   print(n, base);
00129   println();
00130 }
00131
00132 void Print::println(long n, int base)
00133 {
00134   print(n, base);
00135   println();
00136 }
00137
00138 void Print::println(unsigned long n, int base)
00139 {
00140   print(n, base);
00141   println();
00142 }
00143
00144 void Print::println(double n, int digits)
00145 {
00146   print(n, digits);
00147   println();
00148 }
00149
00150 // Private Methods /////////////////////////////////////////////////////////////
00151
00152 void Print::printNumber(unsigned long n, uint8_t base)
00153 {
00154   unsigned char buf[8 * sizeof(long)]; // Assumes 8-bit chars.
```

```
00155   unsigned long i = 0;
00156
00157   if (n == 0) {
00158     print('0');
00159     return;
00160   }
00161
00162   while (n > 0) {
00163     buf[i++] = n % base;
00164     n /= base;
00165   }
00166
00167   for (; i > 0; i--)
00168     print((char) (buf[i - 1] < 10 ?
00169       '0' + buf[i - 1] :
00170       'A' + buf[i - 1] - 10));
00171 }
00172
00173 void Print::printFloat(double number, uint8_t digits)
00174 {
00175   // Handle negative numbers
00176   if (number < 0.0)
00177   {
00178      print('-');
00179      number = -number;
00180   }
00181
00182   // Round correctly so that print(1.999, 2) prints as "2.00"
00183   double rounding = 0.5;
00184   for (uint8_t i=0; i<digits; ++i)
00185     rounding /= 10.0;
00186
00187   number += rounding;
00188
00189   // Extract the integer part of the number and print it
00190   unsigned long int_part = (unsigned long)number;
00191   double remainder = number - (double)int_part;
00192   print(int_part);
00193
00194   // Print the decimal point, but only if there are digits beyond
00195   if (digits > 0)
00196     print(".");
00197
00198   // Extract digits from the remainder one at a time
00199   while (digits-- > 0)
00200   {
00201     remainder *= 10.0;
00202     int toPrint = int(remainder);
00203     print(toPrint);
00204     remainder -= toPrint;
00205   }
00206 }
```

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

```
00001 /*
00002   Print.h - Base class that provides print() and println()
00003   Copyright (c) 2008 David A. Mellis.  All right reserved.
00004
00005   This library is free software; you can redistribute it and/or
00006   modify it under the terms of the GNU Lesser General Public
00007   License as published by the Free Software Foundation; either
00008   version 2.1 of the License, or (at your option) any later version.
00009
00010   This library is distributed in the hope that it will be useful,
00011   but WITHOUT ANY WARRANTY; without even the implied warranty of
00012   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
00013   Lesser General Public License for more details.
00014
00015   You should have received a copy of the GNU Lesser General Public
00016   License along with this library; if not, write to the Free Software
00017   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
00018 */
00019
00020 #ifndef Print_h
00021 #define Print_h
00022
00023 #include <inttypes.h>
00024 #include <stdio.h> // for size_t
00025
00026 #define DEC 10
00027 #define HEX 16
00028 #define OCT 8
00029 #define BIN 2
00030 #define BYTE 0
00031
00032 class Print
00033 {
00034   private:
00035     void printNumber(unsigned long, uint8_t);
00036     void printFloat(double, uint8_t);
00037   public:
00038     virtual void write(uint8_t) = 0;
00039     virtual void write(const char *str);
00040     virtual void write(const uint8_t *buffer, size_t size);
00041
00042     void print(const char[]);
00043     void print(char, int = BYTE);
00044     void print(unsigned char, int = BYTE);
00045     void print(int, int = DEC);
00046     void print(unsigned int, int = DEC);
00047     void print(long, int = DEC);
00048     void print(unsigned long, int = DEC);
00049     void print(double, int = 2);
00050
00051     void println(const char[]);
00052     void println(char, int = BYTE);
00053     void println(unsigned char, int = BYTE);
00054     void println(int, int = DEC);
00055     void println(unsigned int, int = DEC);
00056     void println(long, int = DEC);
00057     void println(unsigned long, int = DEC);
00058     void println(double, int = 2);
00059     void println(void);
00060 };
```

```
00061
00062 #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

```
00001
00002
00003 #include <avr/io.h>
00004 #include <inttypes.h>
00005 #include <avr/interrupt.h>
00006
```

```
00007 #include "TimerCore.h"
00008 #include "TimerCntr.h"
00009
00010 // Generate all of the ISR handlers.. hook them up to a class if/when a class
00011 // is instantiated for a particular USART.
00012 #define TC0_ISR_DEF(tc) \
00013 static TimerCntr*  tc##cp = 0;\
00014 ISR(tc##_ERR_vect) {\
00015     if (tc##cp) tc##cp->err();\
00016 }\
00017 ISR(tc##_OVF_vect) {\
00018     if (tc##cp) tc##cp->ovf();\
00019 }\
00020 ISR(tc##_CCA_vect) {\
00021     if (tc##cp) tc##cp->ccx(0);\
00022 }\
00023 ISR(tc##_CCB_vect) {\
00024     if (tc##cp) tc##cp->ccx(1);\
00025 }\
00026 ISR(tc##_CCC_vect) {\
00027     if (tc##cp) tc##cp->ccx(2);\
00028 }\
00029 ISR(tc##_CCD_vect) {\
00030     if (tc##cp) tc##cp->ccx(3);\
00031 }
00032
00033 TC0_ISR_DEF(TCC0);
00034 TC0_ISR_DEF(TCD0);
00035
00036 // The TimerCore.cpp will use TCE0 and TCE1 if
00037 // TCF1 is not defined.. Hence, I need to NOT define
00038 // It here.
00039 #if not defined(TIMERCORE_USE_TCE0)
00040 TC0_ISR_DEF(TCE0);
00041 #endif
00042
00043 #if not defined(TIMERCORE_USE_TCF0)
00044 TC0_ISR_DEF(TCF0);
00045 #endif
00046
00047 #define TC1_ISR_DEF(tc) \
00048 static TimerCntr*  tc##cp = 0;\
00049 ISR(tc##_ERR_vect) {\
00050     if (tc##cp) tc##cp->err();\
00051 }\
00052 ISR(tc##_OVF_vect) {\
00053     if (tc##cp) tc##cp->ovf();\
00054 }\
00055 ISR(tc##_CCA_vect) {\
00056     if (tc##cp) tc##cp->ccx(0);\
00057 }\
00058 ISR(tc##_CCB_vect) {\
00059     if (tc##cp) tc##cp->ccx(1);\
00060 }
00061
00062 TC1_ISR_DEF(TCC1);
00063 TC1_ISR_DEF(TCD1);
00064 #if not defined(TIMERCORE_USE_TCE1)
00065 TC1_ISR_DEF(TCE1);
00066 #endif
00067
00068 #if not defined(TIMERCORE_USE_TCF1)
```

```
00069 #ifdef TCF1
00070 TC1_ISR_DEF(TCF1);
00071 #endif
00072 #endif
00073
00074 static void SetPointer(TC0_t* tc,TimerCntr* pTC)
00075 {
00076     // Register this object with the appropriate
00077     // pointer so that the ISR routines can call p
00078     // class.
00079     if (tc == &TCC0) {
00080         TCC0cp = pTC;
00081     } else if (tc == &TCD0) {
00082         TCD0cp = pTC;
00083 #if not defined(TIMERCORE_USE_TCE0)
00084     } else if (tc == &TCE0) {
00085         TCE0cp = pTC;
00086 #endif
00087 #if not defined(TIMERCORE_USE_TCF0)
00088     } else if (tc == &TCF0) {
00089         TCF0cp = pTC;
00090 #endif
00091     }
00092 }
00093
00094 static void SetPointer(TC1_t* tc,TimerCntr* pTC)
00095 {
00096     // Register this object with the appropriate
00097     // pointer so that the ISR routines can call p
00098     // class.
00099     if (tc == &TCC1) {
00100         TCC1cp = pTC;
00101     } else if (tc == &TCD1) {
00102         TCD1cp = pTC;
00103 #if not defined(TIMERCORE_USE_TCE1)
00104     } else if (tc == &TCE1) {
00105         TCE1cp = pTC;
00106 #endif
00107 #if not defined(TIMERCORE_USE_TCF1)
00108 #if defined(TCF1)
00109     } else if (tc == &TCF1) {
00110         TCF1cp = pTC;
00111 #endif
00112 #endif
00113     }
00114 }
00115
00116 TimerCntr::TimerCntr(TC0_t*  pTC)
00117 {
00118     _pTC = pTC;
00119     _bTC1 = false;
00120     _pNotifyClient = 0;
00121     _pNotifyClientID = 0;
00122     SetPointer(pTC,this);
00123 }
00124
00125 TimerCntr::TimerCntr(TC1_t*  pTC)
00126 {
00127     _pTC = (TC0_t*)pTC;
00128     _bTC1 = true;
00129     _pNotifyClient = 0;
00130     _pNotifyClientID = 0;
```

```
00131     SetPointer(pTC,this);
00132 }
00133
00134 TimerCntr::~TimerCntr()
00135 {
00136     if (_bTC1) {
00137         SetPointer((TC1_t*)_pTC,0);
00138     } else {
00139         SetPointer(_pTC,0);
00140     }
00141 }
00142
00143
00144 void TimerCntr::ClkSel(TC_CLKSEL_t clksel)
00145 {
00146     _pTC->CTRLA = clksel;
00147 }
00148
00149
00150 void TimerCntr::CCEnable(uint8_t mask)
00151 {
00152     _pTC->CTRLB = ((_pTC->CTRLB & 0x0F) | (mask << 4));
00153 }
00154
00155
00156 void TimerCntr::WaveformGenMode(TC_WGMODE_t wgmode)
00157 {
00158     _pTC->CTRLB = ((_pTC->CTRLB & 0xF0) | wgmode);
00159 }
00160
00161
00162 void TimerCntr::EventSetup(TC_EVACT_t act, TC_EVSEL_t src)
00163 {
00164     _pTC->CTRLD = act | src;
00165 }
00166
00167
00168 void TimerCntr::IntLvlA(uint8_t errlvl, uint8_t ovflvl)
00169 {
00170     _pTC->INTCTRLA = (errlvl & 0x3) << 2 | (ovflvl & 0x3);
00171 }
00172
00173 void TimerCntr::IntLvlB(uint8_t val)
00174 {
00175     _pTC->INTCTRLB = val;
00176 }
00177
00178 void TimerCntr::Counter(uint16_t newVal)
00179 {
00180     _pTC->CNT = newVal;
00181 }
00182
00183 uint16_t TimerCntr::Counter()
00184 {
00185     return _pTC->CNT;
00186 }
00187
00188
00189 void TimerCntr::Period(uint16_t newVal)
00190 {
00191     _pTC->PER = newVal;
00192 }
```

```
00193
00194 uint16_t TimerCntr::Period()
00195 {
00196     return _pTC->PER;
00197 }
00198
00199
00200 void TimerCntr::CCReg(uint8_t idx, uint16_t newVal)
00201 {
00202     if (idx == 0) {
00203         _pTC->CCA = newVal;
00204     } else if (idx == 1) {
00205         _pTC->CCB = newVal;
00206     } else if (!_bTC1 && idx == 2) {
00207         _pTC->CCC = newVal;
00208     } else if (!_bTC1 && idx == 3) {
00209         _pTC->CCD = newVal;
00210     }
00211 }
00212
00213 uint16_t TimerCntr::CCReg(uint8_t idx)
00214 {
00215     if (idx == 0) {
00216         return _pTC->CCA;
00217     } else if (idx == 1) {
00218         return _pTC->CCB;
00219     } else if (!_bTC1 && idx == 2) {
00220         return _pTC->CCC;
00221     } else if (!_bTC1 && idx == 3) {
00222         return _pTC->CCD;
00223     }
00224     return 0;
00225 }
00226
00227 void TimerCntr::Notify(TimerNotify* pClient,uint8_t id)
00228 {
00229     _pNotifyClient = pClient;
00230     _pNotifyClientID = id;
00231 }
00232
00233 void TimerCntr::err()
00234 {
00235     if (_pNotifyClient)
00236         _pNotifyClient->err(_pNotifyClientID);
00237 }
00238
00239 void TimerCntr::ovf()
00240 {
00241     if (_pNotifyClient)
00242         _pNotifyClient->ovf(_pNotifyClientID);
00243 }
00244
00245 void TimerCntr::ccx(uint8_t idx)
00246 {
00247     if (_pNotifyClient)
00248         _pNotifyClient->ccx(_pNotifyClientID,idx);
00249 }
00250
00251 void TimerCntr::SetRate(uint32_t rateHz){
00252     //add an auto prescaler using 32 bit array
00253 }
00254
```

```
00255
00256
```

6.47 TimerCntr.h File Reference

Classes

- class TimerNotify
- class TimerCntr

6.48 TimerCntr.h

```
00001
00002 #ifndef TimerCntr_h
00003 #define TimerCntr_h
00004
00005 class TimerNotify
00006 {
00007 public:
00008     virtual void err(uint8_t id) = 0;
00009     virtual void ovf(uint8_t id) = 0;
00010     virtual void ccx(uint8_t id,uint8_t idx) = 0;
00011 };
00012
00013 class TimerCntr
00014 {
00015     // Since TC0 is a super-set, use this for the
00016     // default, but then we will override this in the
00017     // concrete classes.
00018     TC0_t*          _pTC;
00019     bool            _bTC1;
00020     TimerNotify*    _pNotifyClient;
00021     uint8_t         _pNotifyClientID;
00022
00023 public:
00024
00025     TimerCntr(TC0_t* pTC);
00026     TimerCntr(TC1_t* pTC);
00027     ~TimerCntr();
00028
00029     void Notify(TimerNotify* pClient, uint8_t id);
00030
00031     void ovf();
00032     void err();
00033     void ccx(uint8_t idx);
00034
00036     void ClkSel(TC_CLKSEL_t clksel);
00037
00041     void CCEnable(uint8_t mask);
00042
00045     void WaveformGenMode(TC_WGMODE_t wgmode);
00046
00047     void EventSetup(TC_EVACT_t act, TC_EVSEL_t src);
00048
00049     void IntLvlA(uint8_t errlvl, uint8_t ovflvl);
00050
00051     void IntLvlB(uint8_t val);
00052
```

```
00053     void Counter(uint16_t newVal);
00054     uint16_t Counter();
00055
00056     void Period(uint16_t newPer);
00057     uint16_t Period();
00058
00059     void SetRate(uint32_t rateHz);
00060
00061     void CCReg(uint8_t idx, uint16_t newVal);
00062     uint16_t CCReg(uint8_t idx);
00063
00064 };
00065
00066
00067 #endif
00068
00069
```

Index