Bioband software overview

Medical Research Council

Jim Rayner
BeyondVoice Ltd

Purpose and scope

This document covers the software aspects of the Bioband project.

Its intention is to give an overview of the design and implementation decisions made during the production of the Bioband firmware and the corresponding software for the PC based data extraction and band test.

Background

The Bioband is intended to collect data on the activity of a human subject as part of a large-scale medical research initiative. The activity data will be collected in the form of a configurable number of three-axis accelerometer readings per second over a configurable period of days, usually a week or more.

The intended use case is the simple set up and capture of readings from a remote volunteer subject. Typically involving the band being posted to the subject, the subject, unsupervised, attaching the band to their body and leaving it in place for the data collection period. Once the data collection is complete, the subject takes off and returns the band, most likely by post, with the sample data being extracted on receipt by plugging the band into a PC. Both extraction and recharging of the band occurring as part of a quick and simple turnaround procedure, ready for the sending to the next volunteer.

The requirements for the Bioband project are formally captured in a document entitled ‘Needs and wishes for an open source waveform accelerometer’. That document lead to the specification of the key band criteria below.

The Bioband must:

- be able to collect three-axis accelerometer data of up to +/- 6g for at least 7 days at a sampling rate of at least 40Hz or more and a resolution of 12 bits.
- be small enough to be posted and attached unobtrusively to the human body, typically on the wrist.
- be able to ‘sleep’ for several days before starting data collection.
- be economical to manufacture in thousand-of quantities.
- store data in such a way as to make it straightforward to extract by research staff.
- store sufficient metadata to allow calibration factors to be checked and clock drift to be compensated for.

Design

To supply a lightweight battery efficient design the hardware has been based on the STM32F103 processor, for details see the hardware design document.

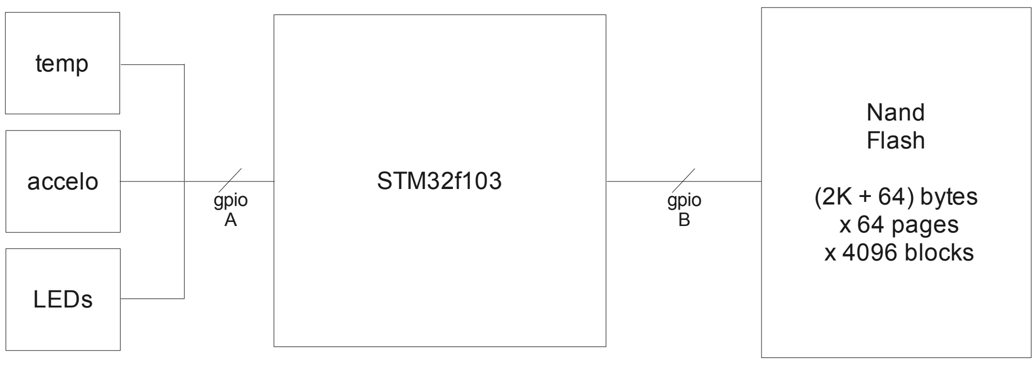
The STM32F103 processor is an inexpensive, low density ARM Cortex-M3 core based device. manufactured by ST Micro. The enclosed Cortex-M3 core provides a standardized microcontroller which goes beyond the classical CPU to provide the entire heart of a microcontroller (including the interrupt system, 24-bit timer and memory map). The Cortex M family is a cost effective and applicable to microcontroller applications. Of greatest importance to this project, it has a number of very low power consumption modes, making it ideal for long duration sampling using a physically small battery, important for a wrist or similar based device.

Main functionality:

- 32kB of flash memory (for the firmware)
- 10kB of SRAM (for the firmware data)
- 20 bytes of backup registers (BKP, backup domain)
- A number of general purpose I/O ports (GPIO)
- An analogue to digital converter (ADC)
- A realtime clock (RTC)
- Programmable interrupts
- A universal serial bus (USB) interface

Note that the firmware uses the STM standard peripheral and USB libraries to provide the support functionality for numerous areas, namely: the backup domain (bkp), analogue to digital conversion (adc), general purpose I/O (gpio), realtime time clock (rtc), clock crystal (rcc), power modes (pwr), serial peripheral interface (spi), serial interface (usart) and nested vector interrupt handling (misc). This simplifies access to these major components, easing the design of the firmware.

The main components of the design are as below



The temperature and accelerometer sensors as well as the LEDs are connected via the general purpose I/O port A (gpio A) to the processor. The Nand flash is connected via gpio B.

The Bioband requirements have been broken down, during the course of the project, into specific firmware design criteria, namely:

- sampling
- timestamping
- data format
- reliability

Note: this document does not detail every nuance of the source code, the details highlighted here are very much to give a starting point from which to understand the source code.

Sampling

The accelerometer chosen (LIS331) for the design can be configured to produce samples at a number of different sampling rates (50, 100, 400 or 1000 Hz). The firmware has been designed to enable configuration of the sampling rate. In a similar way, the accelerometer can also be configured with a maximum range +/-(2, 4 or 8g).

By allowing configuration rather than fixed values for rates and ranges, it may be possible to change the accelerometer in the future for a design with improved ranges or tolerances.

Timestamping

Each sample from the accelerometer shall interrupt the processor only when it is ready to be read. The crystal within the accelerometer, driving the sample rate, has a loose tolerance of +/-10%. To counter this, and accurately timestamp when the samples are recorded, the processor maintains an accurate real time clock and regularly timestamps the sample data. The use of additional timestamps means that complex data processing, to remove any sampling imperfections, can be done later offline on a PC rather than by using the limited resources of the band's processor to curve-fit the sample at the time of reception.

Within the code, the timestamps are referred to as ticks. The real time clock in the band runs off a 32khz (32768 hz) crystal, as such it is difficult to generate a precise 1 millisecond timestamp, so the code is designed to provide the next best thing, a so called tick, generated every 33 clocks, so by giving a known period of 33/32768, or 1.00708 milliseconds. It is a simple PC based conversion process to convert the ticks stored in the data to the required millisecond based timestamps.

Data Format

The Bioband is restricted in resources and cannot support a complex filesystem. Therefore data is stored raw, with no processing, in a very simple format.

The band needs to store a number of data elements at different times:

- accelerometer samples (16 bits per axis), at the approximate rate of the configured accelerometer (50, 100, 400 or 1000 Hz)
- current tick (32 bits) every few seconds, often enough to detect changes in the accelerometer tolerance
- temperature (16 bits) every few seconds, to detect small changes in temperature
- diagnostic data (data status, debug log) every few seconds, when required
- battery level (16 bits) every few minutes, a relatively slow changing quantity
- repetition of meta data (collection, configuration & calibration parameters) as required (every few minutes) to ensure safe recovery in case of memory corruption

The structure of the Nand flash memory (see Nand flash, page 13) in bytes, pages and blocks of pages gives a natural means to add the additional data to the samples in the necessary time domains:

- 6 bytes every sample (341 samples per page)
- 1 page (2048 bytes), every few seconds
- 1 block (64 pages) every few minutes

The actual timings depend on the sampling frequency, the example above is correct for 50 or 100 Hz sampling.

Reliability

The number one priority is that once a data collection has started it must continue free from interruption until either the desired collection period has been met or the battery has discharged to such a level that further collection is impossible.

This design criteria means plugging a cable into the Bioband connector, applying different voltages or environmental conditions to the connector, forcing the processor to reset (by use of the developer connections breakout board) must all have no detrimental effect on the outcome of the collection. Nothing that the volunteer subject should encounter whilst wearing the band should cause any long lasting/permanent loss of sample data.

Where an interruption is inevitable, ie the application of external reset, the interruption to the sampling should be kept to a minimum and the occurrence noted in the status of the page of sample data. This involves the use of the limited number of backup registers (bkp) in the processor to restore the previous program state upon the return from the processor reset. Also the consistency of the Nand flash must be checked in order to report on the potential or known data loss from the reset. Upon downloading the data, the sections of loss should be reported so that the data processing on the PC can take account of the abnormalities.

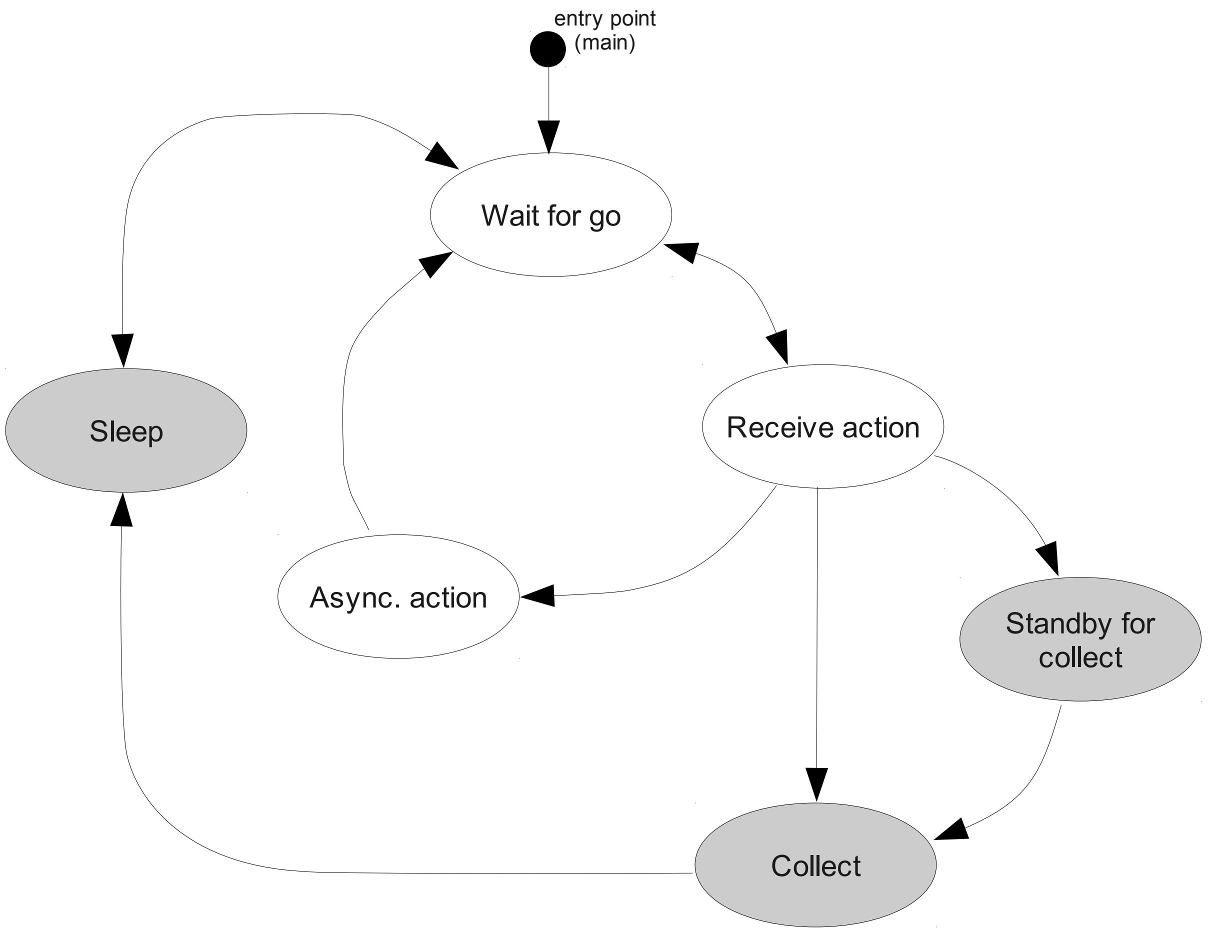
Implementation

The main constraint of the processor is the 32kB of flash for the software. This limit means the software has little room for additional features. The code is very simple, with one main and three interrupt driven entry points.

States

The main entry point, entered upon power reset (power connected for the first time or reapplied after battery drained) or external reset (by the JTAG programmer or reset button on the breakout board), initialises the hardware and then enters the state loop.

The core sequence of operation after main entry is to wait for a USB connection to be established to the device and receive the subsequent go byte, send some data to show the device is active (CWA), then receive one or more actions, which are either handled directly (such as send the meta data, usually the first request by the PC) or the action requires further processing, in which case it is either collect based or a full power asynchronous action (usually transferring data to the PC). The states are shown diagrammatically below:



Currently the asynchronous action is one of the following:

- read the raw file (flash memory) from the device
- read a specific page of memory from the device
- read the stored battery levels from the memory of the device
- read the stored temperature levels from the memory of the device
- read the bad blocks from the memory of the device (a diagnostic aid)
- read the debug buffer from the device

Strictly speaking the standby for collect and collect states are also asynchronous actions, but these differ since they are low power states and result in the device disconnecting from USB and eventually, after the action is complete, entering sleep state.

The wait for go, receive action and asynchronous action states are termed full power states, the device does not enter a low power mode during the operation of these states.

The standby for collection, collect and sleep states are low power mode states (shown in grey). They do the minimal set up work before stopping further execution and waiting for a software interrupt to re-awaken them (after the appropriate handler has dealt with the interrupt), whereupon they check if a state change is required or not.

Note that in the code the functions relating to the states are named as below:

state	function name (main.c)
Wait for go	waitForConfig
Receive action	configure
Async action	waitForEndOfRead
Standby for collect	waitForCollectionTime
Collect	collect
Sleep	sleep

Note: As currently all the asynchronous actions are read related, the related function name is waitForEndOfRead

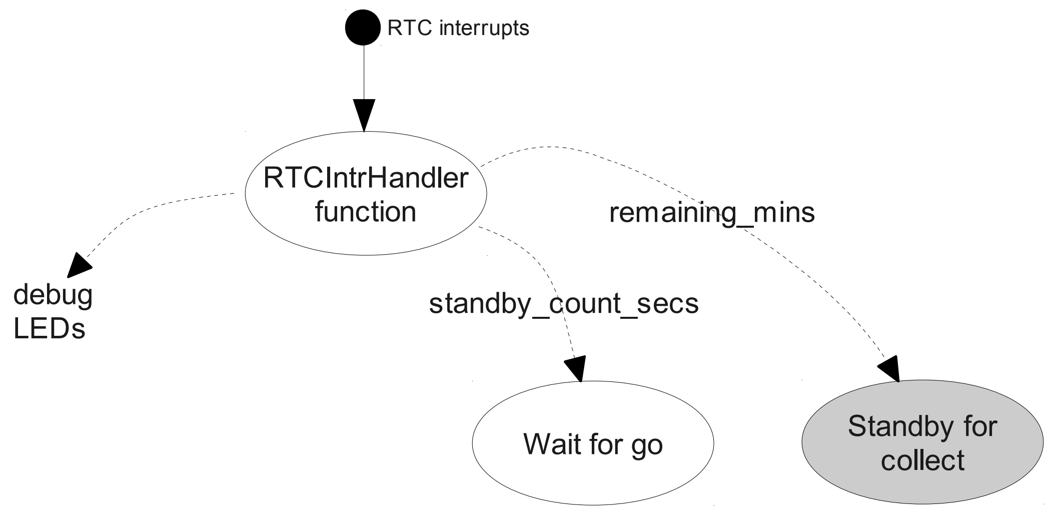
Interrupts

The Cortex processor has two operating modes: Thread mode and Handler mode. The CPU runs in Thread mode while it is executing in non-interrupt background mode and switches to the Handler mode when it is executing interrupts/exceptions.

Three interrupt handler mode entry points are used in the Bioband design:

- real time clock (interrupts every second or after a specific period)
- USB wakeup (USB lead inserted)
- accelerometer data ready

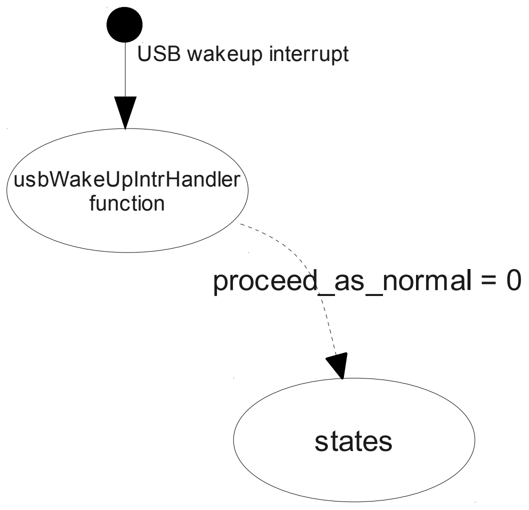
Real time clock interrupt



The real time clock interrupt handler (function RTCIntrHandler) is used to drive the periodic flashing of the diagnostic light emitting diodes (LEDs) and alter global time variables for the two main waiting states.

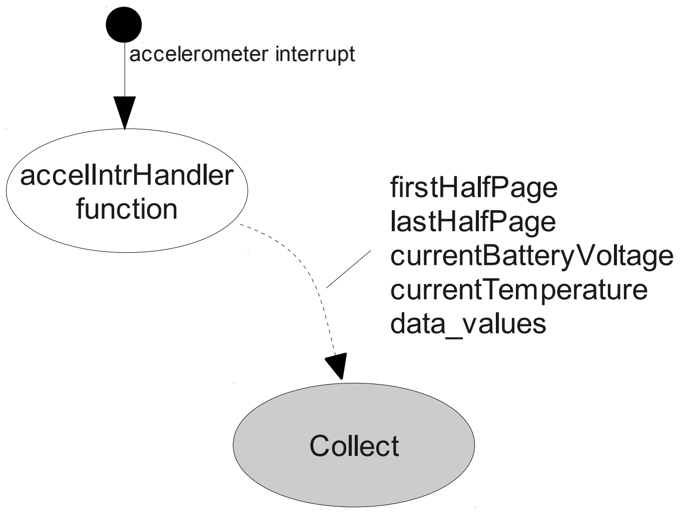
The LEDs are only enabled if the band is in debug mode due to the current drain on the device. The time variables are only altered when the state is “wait for configuration”, to ensure the device will return to the sleep state if there is no interaction in a 5 minute period, or during standby for collect, when the RTC interrupt is used to periodically check (every minute) if the collect should start.

USB wakeup interrupt



The USB wakeup interrupt handler (function usbWakeUpIntrHandler), in any other state than collect or standby for collect, simply resets a global variable (proceed_as_normal) to indicate that a USB lead has been inserted. This variable is checked in various state loops and the resetting of the variable acts as a breakout from the current state, of particular importance being the sleep state.

Accelerometer interrupt



The accelerometer interrupt handler (function accelIntrHandler) is active only during the collect state. Upon entry the accelerometer sample is retrieved and stored in the data array (data_values) and a check made whether the number of samples is enough to trigger the writing of a half page of the stored samples to the nand flash memory (global variables firstHalfPage and lastHalfPage). Due to the settling time for battery and temperature measurements and to keep the power use low, these also need for these to be enabled/disabled at appropriate sampling trigger points depending on the sampling frequency.

Debug

The lack of memory (32KB for the program) limits the debugging that can be added to the code and the long running of the device during a typical collection phase (over a week) limits how the device can be debugged and the amount of debug data that can be captured and sensibly filtered.

Currently the code uses a combination of LED colours (when in debug mode) and the output of simple string tokens via either serial communications or stored in the Nand flash along with the sample data.

The LED colours are configured with the following meanings:

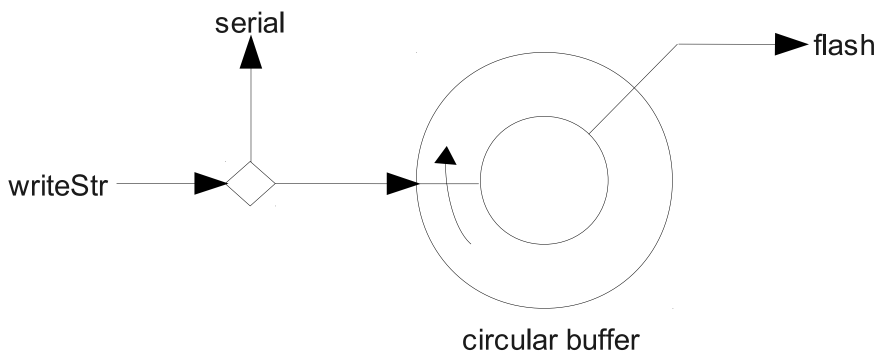
- fast flashing blue led – the Bioband is carrying out data transfer or data collection
- flashing green led – the Bioband is in the configuration (receive action) state
- flashing red led – the Bioband is awaiting for a connection to be established to the device
- flashing purplish blue (red & blue LEDs) – the Bioband is in the standby for collect state
- constant white light – the device has hit an exception, programming error

Through out the code, points of interest are logged using the following api:

```
writeStr(const char* dbg_txt)
writeHex8(uint8_t n)
writeHex16(uint16_t n)
writeHex32(uint32_t n)
```

The debug data is either transferred to a connected PC via serial or held in a circular buffer and later transferred to the debug field in the spare area of the Nand flash page.

The circular buffer is read and used to fill the debug field upon the current sample page being written to Nand flash. A circular buffer is used as there may be a difference between the rates of production and consumption. The circular buffer allows the flow to be controlled or in the extreme case, debug data to be ignored, if the buffer is full due to production outweighing consumption (a tilda ~ character is used to mark in the debug data when overflow has occurred).



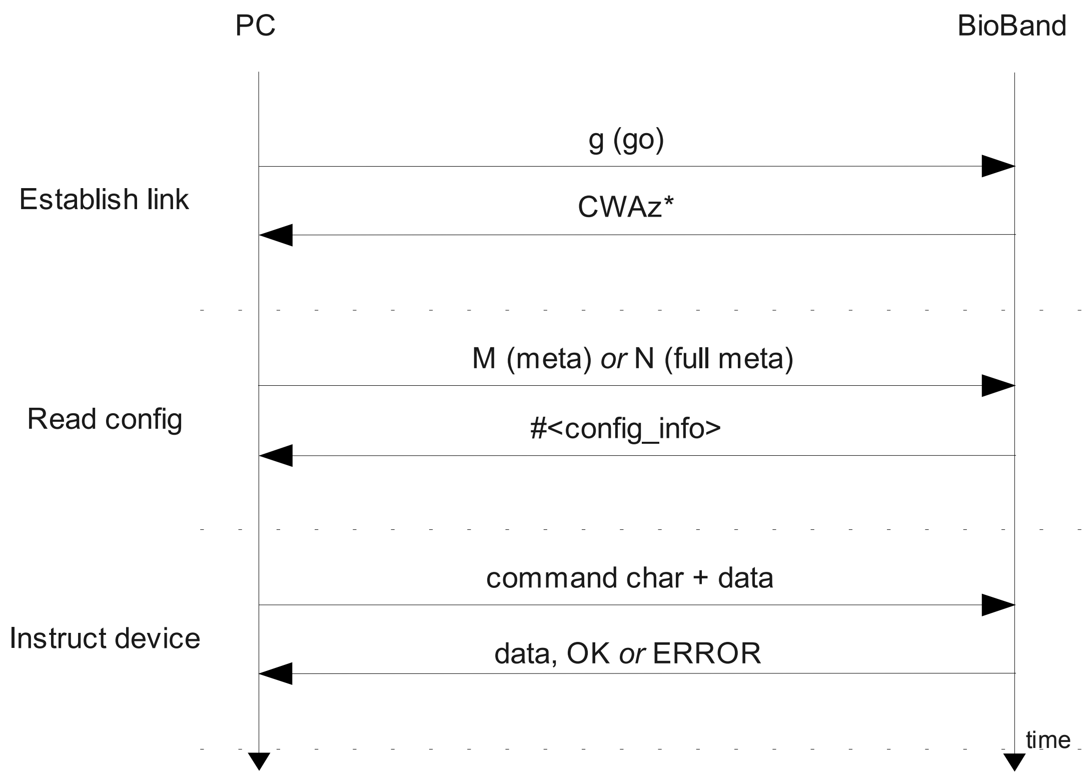
The debug buffer is held in the processor's memory, as such it does not retain debug data across a processor reset. A useful future optimisation may be to store the debug codes to flash as they occur rather than when a page of samples is ready to be written.

USB protocol

The USB interface on the processor supports up to eight endpoints, which are user configurable as endpoints for control, interrupt, bulk or isochronous pipes. The endpoint packet buffers are stored in 512 bytes of memory in the processor. When the device is initialised, the application software divides the memory into a series of buffers. In the Bioband case, three endpoints are used, one control pipe and a raw bulk transfer in both directions. The raw bulk protocol used is an adhoc structure of 64 bytes, usually the leading byte representing the command being requested followed by any data associated with the command.

The protocol, governed by the PC side bioband software, uses three steps:

- Establish the link with the device, a simple g character is sent, with a “CWAz” (or “CWA1” if in non debug mode) sequence of characters expected in return.
- Read the configuration of the device, by passing a M or N character depending on whether the details of the last page / timestamp is required (I.e potentially taking longer to extract from the flash memory of the device). The configuration data is passed back with an initial # character symbolising that the sequence of data is the configuration data.
- Finally, instruct the device as to which action you wish it to perform. Depending on the instruction, the byte stream returned may contain data (in potentially several transmissions) or a simple OK or ERROR string of characters for a simple set instruction result.



Note *: CWA is the old name for the Bioband

The full list of commands and their responses are detailed below:

Description	PC → Bioband		Bioband → PC	
	Lead byte	Data	Lead byte	Data
establish link between PC & band	g			CWA1 *or* CWAz
read meta data from band	M		#	config data
read full meta data from band	N		#	config data
configure collection parameters	#	collection data	#	config data
start collection	s			OK *or* ERROR
read battery levels data	L			battery data*
read temperature levels data	t			temperature data*
read bad blocks data	B			bad blocks data*
set Bioband identity	I	identity data		OK
return to establish link state	r			no response**
get Bioband version information	V			version data
get Bioband device time	T			device time
erase all stored data	X			OK
set Bioband LED colour	l	colour setting		OK
instruct Bioband to go to sleep	z			no response**
read a specific page of flash	p			page data
read the next page of flash	q			page data
read raw Bioband data from flash	R			raw data*
wipe the backup domain data	W			OK
read the Bioband debug buffer	b			debug data*
set the accelerometer configuration	A	accelerometer config		OK
read the accelerometer configuration	a			accelerometer config
read the accelerometer config from flash	f			accelerometer config
set the calibration data	K	calibration data		OK
set the first download timestamp	J	timestamp		OK *or* ERROR
get the first download timestamp	i			timestamp
check if complete capture in flash	C			boolean

Note *: asynchronous actions, may require 1 or more transfers of data, the end of the data transfer is simply signalled by ending with a “Done” character sequence

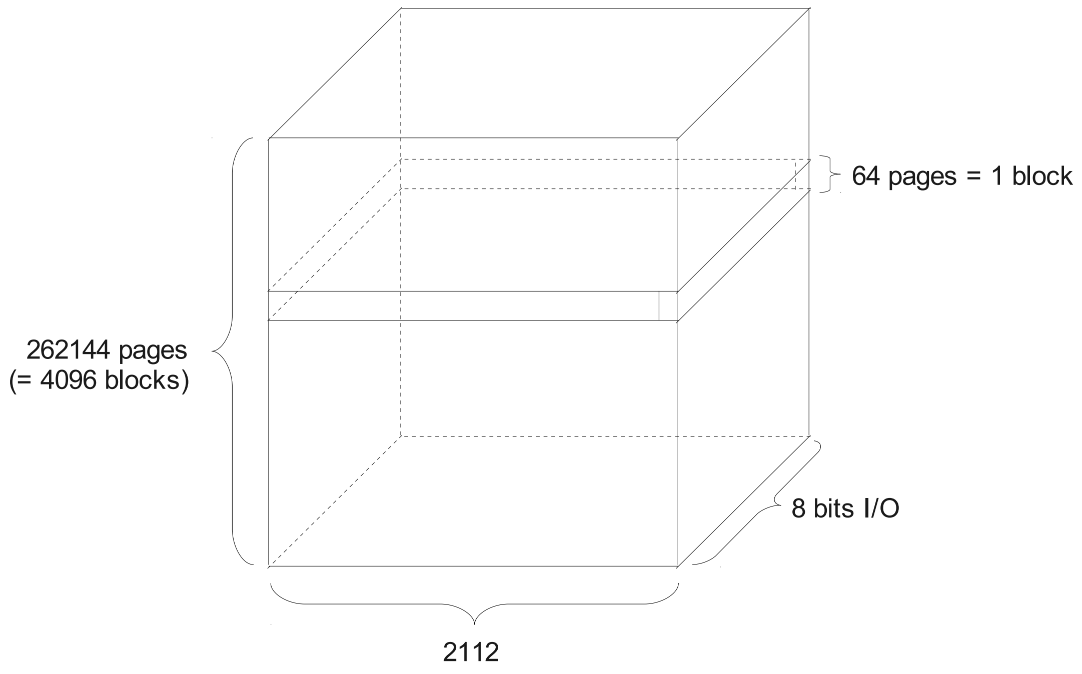
Note **: No response is given for “return to establish link” state and “instruct Bioband to go to sleep” instructions due to their nature, they alter the operation of the USB link and so a response cannot be given.

Nand flash

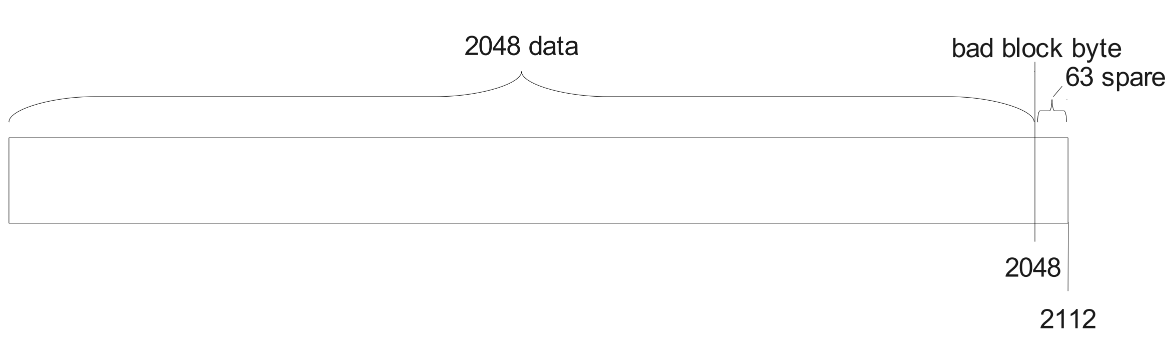
This Bioband data must be stored in a non-volatile memory, so that it is retained even if the battery powering the CWA becomes exhausted during or after data collection. There is only one realistic choice for such storage in the volume required in this application: NAND Flash memory. This type of memory is commonly used as storage in solid-state laptop hard drives, music players, mobile phones and other portable devices, and is thus cheap and readily available. It consumes no power when not being accessed, making it ideal for battery-powered use.

Nand flash is typically made up of blocks, each block containing a set number of pages, each page containing a set number of bytes.

Below is the flash composition of the HY27UF084G2 chip, used in the Bioband design:



Each page consists of 2112 bytes, split into 2048 for data and 64 so called spare:



The first byte of the spare area is reserved for the bad block marker. If the marker is not 0xFF then the block containing the page is deemed to be faulty and is ignored by the firmware.

Most NAND devices are shipped from the factory with some blocks marked as bad. By allowing some bad blocks, the manufacturers achieve far higher yields than would be possible than if all blocks had to be verified good. This significantly reduces NAND flash costs and only slightly decreases the storage capacity of the parts.

Nand does not act like conventional memory. Initially or when a block is erased, all the pages within the block contain binary logic ones. Writing to the nand changes ones to zeros. Once changed, you cannot change zeros to ones. Access is also different, reading / writing is performed on a page basis and erasing on a block basis.

For this project, the Nand flash has been harnessed at the most basic level, no use of the page wide error correcting checksum (ECC), only the sample data is stored with a simple two byte checksum, leaving the spare area to store additional Bioband meta data.

Note the flash handling code does not implement any wear levelling algorithm. The battery is likely to degrade after 300 – 400 recharge cycles, a long time before the need for wear levelling becomes a potential issue (3000+ cycles).

As the flash spare area is not used to hold the conventional higher level error correction data superfluous in this role, we have used the space to store meta data rather than interlace it within the samples and sample checksum held in the main 2048 bytes.

For the Bioband, the spare area of the flash is used to hold meta data (layout specified in header file shared.h):

field	bytes
bad block	1
page status	1
current tick	4
temperature level	2
battery level	2
band identity	8
subject identity	8
test identity	8
centre identity	2
calibration	12
max samples	4
start epoc	4
accelerometer config	1
version	1
<debug>	6

page 0

field	bytes
bad block	1
page status	1
current tick	4
temperature level	2
downloaded timestamp	4
end tick	4
end sample number	4
action timestamp	4
<debug>	40

page 1

field	bytes
bad block	1
page status	1
current tick	4
temperature level	2
<debug>	56

pages 2 - 63

The page 0 meta data is duplicated in the page 0 of every block in the highly unlikely case that one or more of the initially written blocks are effected by corruption.

In the case of an incomplete collection (for example if the battery died) then the end tick and end sample number details in page 1 will still be all ones (I.e. 0xFFFF), a useful indicator for incorrect completion. Note that the page 1 details are not replicated across every block.

Details from the circular debug buffer (page 10) fill the remaining space (see <debug> fields above). If there is no debug details then a raw page block index is written to the area as a simple marker to aid debugging of the raw data file when transferred to or present on the PC.

Test

No simulated hardware environment exists for the Bioband design, so all the implementation test has been carried out directly on the hardware, an additional time factor added to all the testing work.

The following testing process has been used for the Bioband, namely:

- Simple unit dev tests – use of the MRC test program to drive the different paths in the state machine.
- Long term dev tests – the use of 1000Hz sampling has proved useful to quickly build up a large number of samples and test the handling of flash memory handling.
- Acceptance and System real life tests – the real life usage, as yet, difficult to reproduce in the development environment.

Outstanding issues/defects

As of 28th September 2011, these are the known issues with the Bioband firmware/software

Description	Comment
A download cannot be restarted from the point reached if the physical connection is broken (e.g due to faulty connection/connector) during the download.	Could be solved by improving the connector and/or enabling a start index for the read (so can resume from point reached).
Should not be possible to easily change the bioband tag id once it is set	If the tag id already exists then ask the user for confirmation.
Every time a 'new' bioband is plugged into a Windows XP based PC it requests the Windows driver to be installed again	Each Bioband has a different serial number, this appears to cause problems with Windows XP.
On Windows, with debug on, a lot of USB errors are reported once the band enters sampling mode or sleep mode	The driver does not know to disconnect.
If you accidently leave the JTAG openocd program running when the band enters sleep mode then cannot re-JTAG using openocd again until the battery runs down or the battery is shorted.	For now, do an immediate Ctrl-C of openocd after flash. Issue can only be fixed by new release of hardware breakout board.
Temperature level samples take a short while to level out, very first samples appear to be 2-3 degrees higher than the longer term average.	Maybe not unreasonable.
New or fully discharged Biobands appear not to respond to USB comms until a short while after they had been connected to the power	In theory, USB comms should work as soon as power is available but this appears not to be occurring.
Iteraction with some of the Biobands appears to be easily affected by changing the angle of the connector with relation to the Bioband, for example when resting the newly connected Bioband on a table surface etc.	Could be solved by improving the connector.
The connector does not react well to long term submersion in water.	Noted here for completeness even though not software related.
A small number of Biobands in every test have suddenly stopped collecting data for no logged reason and with the battery level being acceptable (over 4V).	The Biobands appear to be susceptible to some form of environmental effect on the connector although the form of reset or other effect is still to be caught.
USB data rate is not as fast as it could be	USB 2.0 operation should be possible but so far performance is more like USB 1.1

Build procedure

System requirements

The band code needs to be cross compiled on a Linux based PC in order to run on the STM32 ARM Cortex-M3 processor.

The example commands and output is from a cross compile on a Linux Ubuntu Long Term Support (10.04) distribution, other distributions can be used but be aware that you will need to alter the commands to those of the distribution used where appropriate, not recommended unless experienced with Linux across different distributions.

JTAG programming dongle

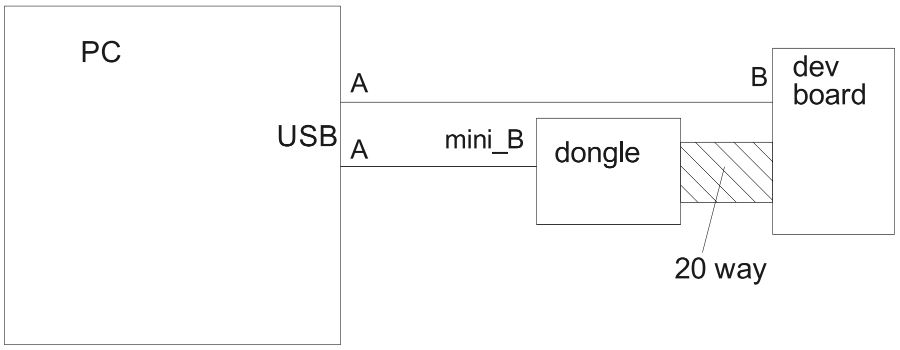
The Bioband is programmed using the JTAG interface. The JTAG interface allows compiled code to be downloaded into the STM microcontroller in the band. This requires the Bioband to be plugged into a development board (so called dev board or breakout board) and a JTAG programming dongle be used to talk to the JTAG interface from the PC.

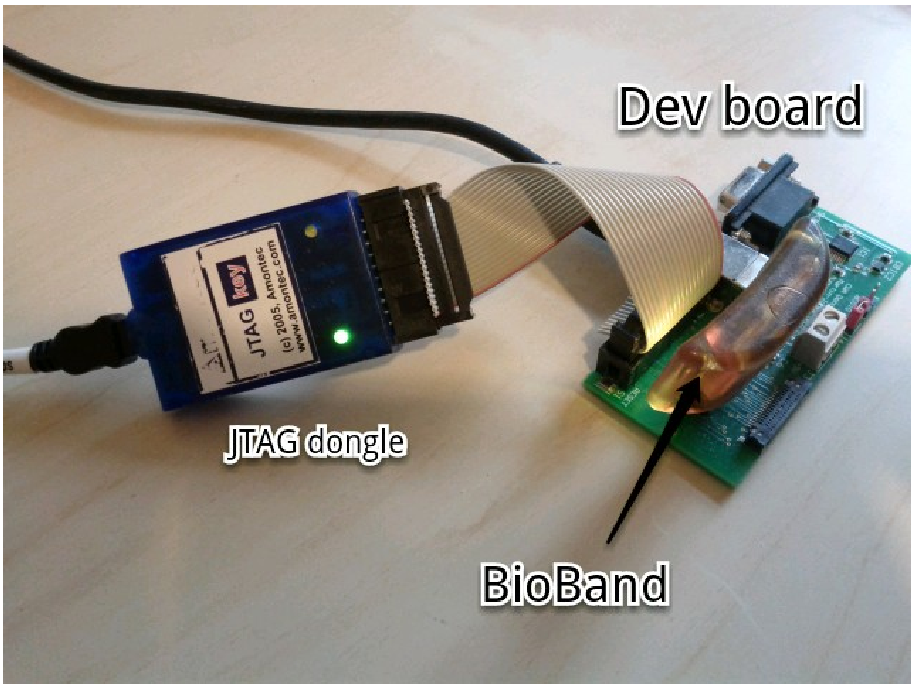
For this project the Amontec JTAGkey programming dongle has been used (www.amontec.com), although other devices should provide the same function. The use of this device was a quick solution when combined with the open source openocd program for controlling the JTAG process.

The dongle requires a standard USB A to mini USB B connector lead.

The development board requires a USB A to USB B connector lead, often found for connecting a PC to a peripheral such as a printer.

Between the dongle and the dev board is a 20 way strip cable.





Using Openocd

Assuming a dongle / dev board set up as previously described, the openocd software for driving the dongle can be tested.

Firstly install the required software packages (as noted before, these commands assume the use of Ubuntu LTS 10.04 or similar Linux distribution)

```
sudo apt-get install libnet-telnet-perl openocd
```

If you haven't already, install the Bioband software in a folder of your choosing, for example `/home/$USER/Bioband2`, where $USER is your username on the PC

Change directory to where the configuration file for openocd is stored

```
cd /home/$USER/Bioband2/software/Bioband/band/gcc
```

Within a Linux terminal do a test openocd invocation with a band connected

```
openocd
```

If the dongle is not found or there are errors try installing libftdi

i.e `sudo apt-get install libftdi1`

Also ensure in the openocd.cfg file in the current directory that

```
ft2232_device_desc "Amontec JTAGkey A"
```

or

```
ft2232_device_desc "Amontec JTAGkey-2"
```

is specified.

Select the correct description depending on the type of Amontec key being used (refer to sticker on the device)

You should now have a working development dongle and openocd

Depending on the version installed you should get output similar to the following (although the version of openocd is now likely to be 0.3.1)

```
Open On-Chip Debugger 0.2.0 (2009-12-11-10:48) svn:2826
$URL: http://svn.berlios.de/svnroot/repos/openocd/tags/openocd-
0.2.0/src/openocd.c $
For bug reports, read http://svn.berlios.de/svnroot/repos/openocd/trunk/BUGS
500 kHz
jtag_nsrst_delay: 100
jtag_ntrst_delay: 100
Info : JTAG tap: stm32.cpu tap/device found: 0x3ba00477 (mfg: 0x23b, part:
0xba00, ver: 0x3)
Info : JTAG Tap/device matched
Info : JTAG tap: stm32.bs tap/device found: 0x16410041 (mfg: 0x020, part:
0x6410, ver: 0x1)
Info : JTAG Tap/device matched
```

Hopefully the output should then remain static.

If there are then intermittant lines on the screen saying something similar to ..

```
Warn : Timeout (1000ms) waiting for ACK = OK/FAULT in SWJDP transaction
```

this means it has detected the device but the Bioband is asleep.

Press the Ctrl and C keys together to break out of the openocd program.

Toolchain

The band code needs to be cross compiled on a Linux based PC in order to run on the STM32 ARM Cortex-M3 processor. This section describes how to set up the cross compiler on a new/clean PC, the example commands and output is from a cross compile on a Linux Ubuntu (10.04) distribution.

Note: creating the toolchain from scratch is a long winded process and it's likely that I've not covered every nuance below.

During the compilation, do not proceed onto the next stage if the current stage you are executing fails to build as expected. In such circumstances it is best to delete the build directory and start again, be mindful of spelling mistakes and missing command parameters.

First off all set up the required base packages.

In a terminal

```
sudo apt-get install build-essential flex bison libgmp3-dev libmpfr-dev autoconf
sudo apt-get install texinfo libncurses5-dev libexpat1 libexpat1-dev


export TOOLPATH=/usr/local/cross-cortex-m3
sudo mkdir $TOOLPATH
sudo chown $USER:$USER $TOOLPATH
```

where $USER is your username. It is important to change the ownership since that ensures you don't need to sudo in order to make install, which would require setting up the root user with the same environment.

Add this to your ~/.bashrc file

```
export TOOLPATH=/usr/local/cross-cortex-m3
export PATH=${TOOLPATH}/bin:$PATH
```

It may also be worth setting up an alias to the Bioband directory

```
alias Bioband='cd /home/$USER/Bioband2/software/Bioband'
```

Building the toolchain in your home directory …

If you have a multicore processor you can speed up the compile by specifying the number of processors available on the make command line after the -j parameter (see example use of two processors (`make -j 2`)). If you PC has a single processor or you don't know the number of cores then omit the -j parameter.

In the terminal enter the following

```
mkdir ~/toolchain
cd ~/toolchain

wget http://ftp.gnu.org/gnu/binutils/binutils-2.19.tar.bz2
tar -xvjf binutils-2.19.tar.bz2
```

```
wget http://fun-tech.se/toolchain/gcc/binutils-2.19_tc-arm.c.patch
patch binutils-2.19/gas/config/tc-arm.c binutils-2.19_tc-arm.c.patch

cd binutils-2.19
mkdir build
cd build
../configure --target=arm-none-eabi  \
             --prefix=$TOOLPATH  \
             --enable-interwork  \
             --enable-multilib  \
             --with-gnu-as  \
             --with-gnu-ld  \
             --disable-nls
make -j 2
make install

cd ~/toolchain
wget ftp://ftp.sunet.se/pub/gnu/gcc/releases/gcc-4.3.4/gcc-4.3.4.tar.bz2
tar -xvjf gcc-4.3.4.tar.bz2
cd gcc-4.3.4
mkdir build
cd build
../configure --target=arm-none-eabi  \
             --prefix=$TOOLPATH  \
             --enable-interwork  \
             --enable-multilib  \
             --enable-languages="c,c++"  \
             --with-newlib  \
             --without-headers  \
             --disable-shared  \
             --with-gnu-as  \
             --with-gnu-ld
make -j 2 all-gcc
make install-gcc

cd ~/toolchain
wget ftp://sources.redhat.com/pub/newlib/newlib-1.17.0.tar.gz
tar -xvzf newlib-1.17.0.tar.gz

wget http://fun-tech.se/toolchain/gcc/newlib-1.17.0-missing-makeinfo.patch
patch newlib-1.17.0/configure newlib-1.17.0-missing-makeinfo.patch

cd newlib-1.17.0
mkdir build
cd build
../configure --target=arm-none-eabi  \
             --prefix=$TOOLPATH  \
             --enable-interwork  \
             --disable-newlib-supplied-syscalls  \
             --with-gnu-ld  \
             --with-gnu-as  \
             --disable-shared

make -j 2 CFLAGS_FOR_TARGET="-ffunction-sections \
                        -fdata-sections \
                        -DPREFER_SIZE_OVER_SPEED \
                        -D__OPTIMIZE_SIZE__ \
                        -Os \
                        -fomit-frame-pointer \
                        -mcpu=cortex-m3 \
                        -mthumb \
                        -D__thumb2__ \
                        -D__BUFSIZ__=256" \
                        CCASFLAGS="-mcpu=cortex-m3 -mthumb -D__thumb2__"
make install


cd ~/toolchain
cd gcc-4.3.4/build
make -j 2 CFLAGS="-mcpu=cortex-m3 -mthumb" \
     CXXFLAGS="-mcpu=cortex-m3 -mthumb" \
     LIBCXXFLAGS="-mcpu=cortex-m3 -mthumb" \
     all
make install

cd ~/toolchain
wget http://ftp.gnu.org/gnu/gdb/gdb-7.0.tar.bz2
tar -xvjf gdb-7.0.tar.bz2
cd gdb-7.0
mkdir build
cd build
../configure --target=arm-none-eabi \
                      --prefix=$TOOLPATH  \
                      --enable-languages=c,c++ \
                      --enable-thumb \
                      --enable-interwork \
                      --enable-multilib \
                      --enable-tui \
                      --with-newlib \
                      --disable-werror \
                      --disable-libada \
                      --disable-libssp
make -j 2
make install
```

Flashing the band

With openocd running in a terminal (see page 17)

Open another terminal

Change to the same Bioband/band/gcc directory as openocd is running from, compile the code by executing the following

```
make clean
make
```

You should end up with some output similar to the following, only the last few lines shown for brevity

```
 <snip>
 arm-none-eabi-gcc -I./ -I../inc
-I../../../Libraries/stm32f10x_StdPeriph_Driver/inc
-I../../../Libraries/CMSIS/Core/CM3 -I../../../Libraries/toolchain_USB-FS-
Device_Driver/inc -c -fno-common -O2 -g -mcpu=cortex-m3 -mthumb -Dstm32f10x_LD
-DUSE_STDPERIPH_DRIVER -DUSE_toolchain10B_EVAL -DHSE_VALUE=16000000
-DMRC_CWA ../src/main.c
 arm-none-eabi-ld -v -Tlinker.cmd -nostartfiles -o main.out  stm32f10x_rcc.o
stm32f10x_gpio.o stm32f10x_spi.o stm32f10x_rtc.o stm32f10x_bkp.o stm32f10x_pwr.o
stm32f10x_exti.o stm32f10x_adc.o stm32f10x_usart.o stm32f10x_tim.o misc.o
system_stm32f10x.o core_cm3.o startup_stm32f10x_ld.o stm32f10x_it.o usb_core.o
usb_init.o usb_int.o usb_mem.o usb_regs.o usb_sil.o hw_config.o usb_desc.o
usb_endp.o usb_istr.o usb_prop.o usb_pwr.o nand_cwa.o accel.o tempsensor.o
debug.o shared.o main.o
 GNU ld (GNU Binutils) 2.19
 ...copying
 arm-none-eabi-objcopy -Obinary main.out main.bin
 arm-none-eabi-objdump -S main.out > main.list
```

The important part is the main.bin which is transferred to/flashed onto the connected Bioband by executing ..

```
make flash
```

If there is an error finding the script do_flash.pl, make sure that the Bioband/scripts directory is on your $PATH by typing `env` into the terminal.

If the scripts directory is missing, type

```
export PATH=/home/$USER/Bioband2/scripts:$PATH
```

substituting your script directory path instead of the example given

Upon the make flash, the openocd terminal should give additional output similar to the following

```
Info : accepting 'telnet' connection from 0
Info : JTAG tap: stm32.cpu tap/device found: 0x3ba00477 (mfg: 0x23b, part:
0xba00, ver: 0x3)
Info : JTAG Tap/device matched
Info : JTAG tap: stm32.bs tap/device found: 0x16410041 (mfg: 0x020, part:
0x6410, ver: 0x1)
Info : JTAG Tap/device matched
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08007dac
Info : device id = 0x20016410
Info : flash size = 32kbytes
flash 'stm32x' found at 0x08000000
stm32x mass erase complete
Warn : not enough working area available(requested 16384, free 16336)
wrote  33572 byte from file
/data/projects/mrc/svn/mrc/software/trunk/dev_board/Bioband2.1/Band/gcc/main.bin
to flash bank 0 at offset 0x00000000 in 2.291996s (14.304194 kb/s)
Info : JTAG tap: stm32.cpu tap/device found: 0x3ba00477 (mfg: 0x23b, part:
0xba00, ver: 0x3)
Info : JTAG Tap/device matched
Info : JTAG tap: stm32.bs tap/device found: 0x16410041 (mfg: 0x020, part:
0x6410, ver: 0x1)
Info : JTAG Tap/device matched
Info : dropped 'telnet' connection - error -400
```

Don't worry about the '`not enough working area available`' message

One very important note - after the 'telnet' connection message at the end has been displayed, press the Ctrl and C buttons whilst the focus is on the terminal with openocd running

This step is needed since there is an issue if the Bioband goes to sleep (after 5 minutes of inactivity, in order to save battery) whilst openocd is running, the device can become unusable until the battery goes flat and is recharged.

Resetting the Bioband

When flashing a Bioband, it is useful to reset the bkp domain on the device and set the band id.

To do this use the mrc program after flashing, see building the PC side code (page 25)

```
./mrc -nobkp
```

You will then need to reset the band (reset button on the breakout board)

Then if flashing for the first time, you will need to assign an identity to the band

```
./mrc -id black1
```

In this example, assigning black1. A short collection is required to save the id into flash.

It is preferable for these ids to be unique, as they are used in part as the USB serial number

This step can also be carried out using the demo program (page 26).

Instructions for PC software

There are two demonstration programs (demo, mrc) that can be used to access data on the Bioband. Both are located within the Bioband/PC directory tree of the installed code.

Building the PC side code

On Linux:

Change directory to the Linux directory for the particular PC side code, i.e PC/MRC/Linux or PC/Demo/Linux.

Then type

```
make
```

In order to talk to the Bioband using ./demo or ./mrc on certain flavours of Linux without being superuser you need to alter the following udev file 50-udev-default.rules in /lib/udev/rules.d

Search for the usb_device line and change the MODE from "0664" to "0666" as shown below

```
    SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", MODE="0666"
```

On Windows:

You need to ensure that a copy of Windows Visual Studio is installed, a free copy of Visual Studio 2008 Express is available if you search on the Microsoft website

Double click on MRC.vcproj or Demo.vcproj files in PC/MRC/Windows or PC/Demo/Windows

This will open Visual Studio, then select the Build

You will need to install the Windows USB driver in order to communicate with the Bioband, there is a user_guide.pdf file that explains the steps in the Windows directory

Automating the PC code

On Linux:

There is a simple shell script (Bioband.sh) that will invoke the Demo program. It is easy to engineer the calling of this script upon a new Bioband being detected by the USB system by configuring a udev rule and placing that rule in /etc/udev/rules.d

For example ..

```
# udev config for Bioband
ACTION=="add",SUBSYSTEM=="usb",ATTRS{idVendor}=="0483",ATTRS{idProduct}=="6000",
ATTRS{serial}=="?*",RUN+="/home/mrc/Bioband2/udev/Bioband.sh '$attr{serial}'"
```

Once added with pathname to the location of Bioband.sh, refresh the udev rules by doing the following

```
sudo reload udev
```

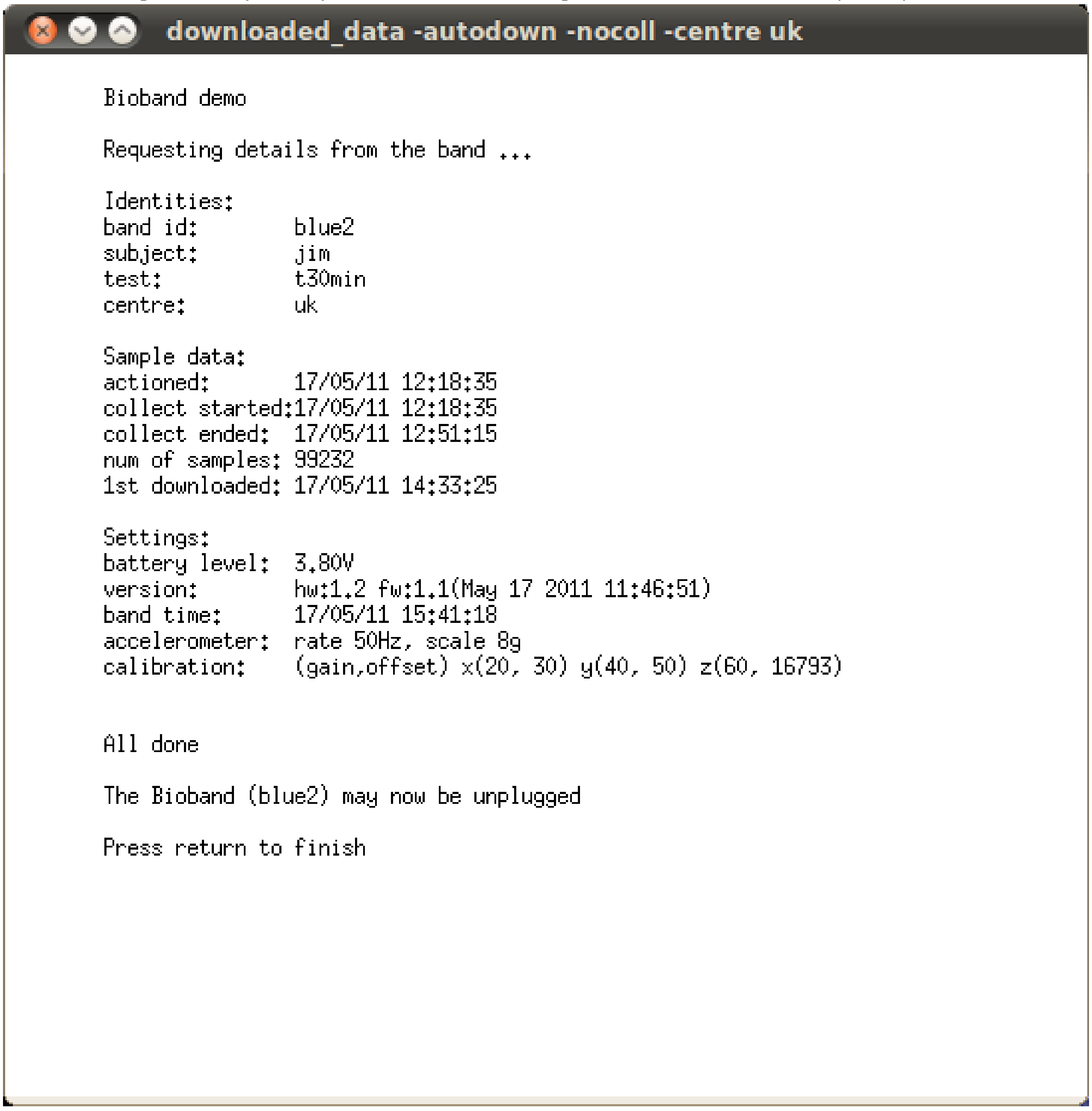
On Windows:

At the current time, no in-depth work has been spent on automating the calling of the MRC or Demo programs by Windows detecting a USB event. If the Bioband work is taken forward this is a likely work area.

The Demo program

The demo program is designed to show how a simple workflow of data download and program for the next collection cycle might be encoded using the Bioband API methods

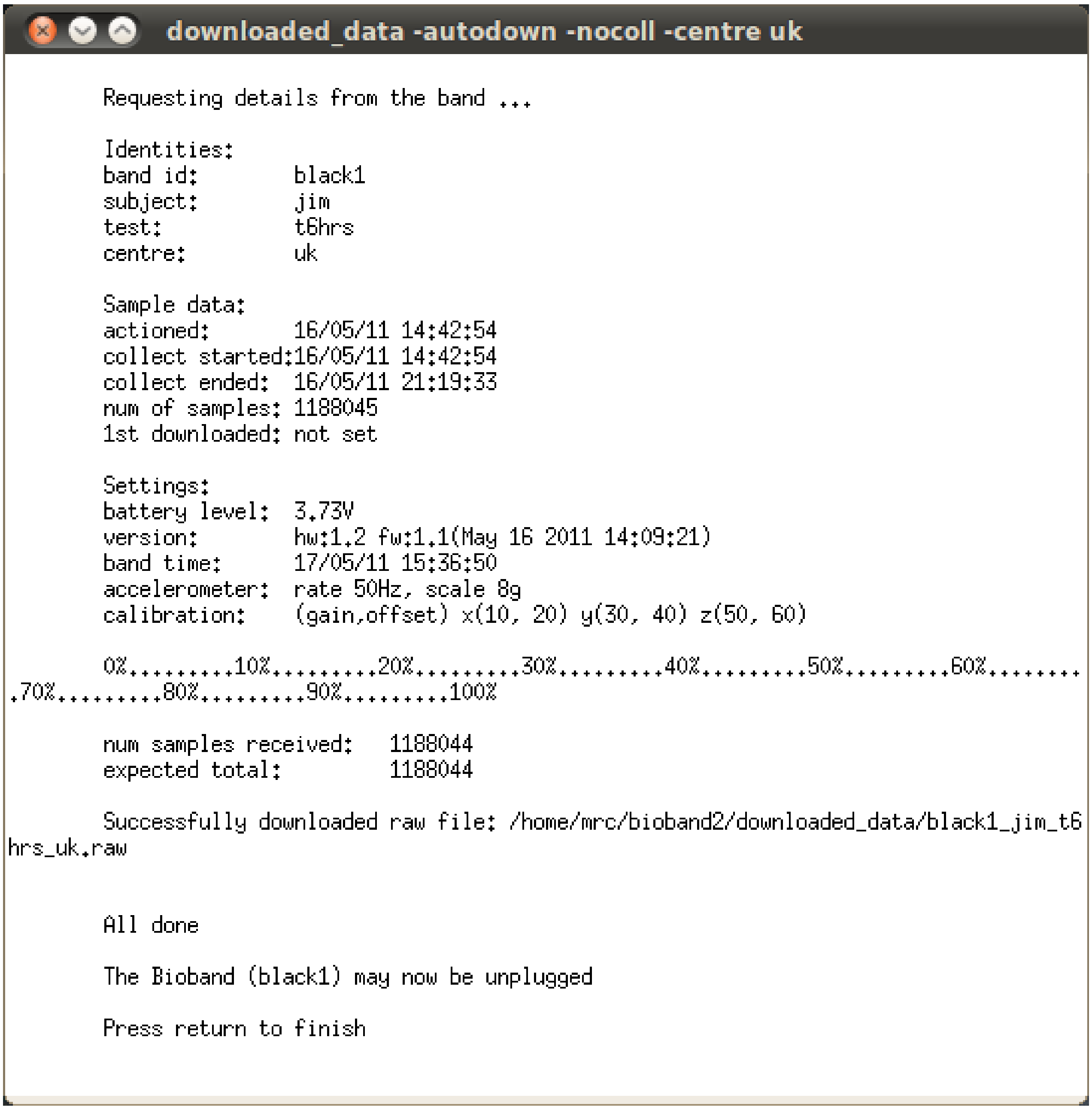
The following is example output from demo running in an xterm on Ubuntu (Linux):



```
Bioband demo

Requesting details from the band ...

Identities:
band id:        blue2
subject:        jim
test:           t30min
centre:         uk

Sample data:
actioned:       17/05/11 12:18:35
collect started:17/05/11 12:18:35
collect ended:  17/05/11 12:51:15
num of samples: 99232
1st downloaded: 17/05/11 14:33:25

Settings:
battery level:  3.80V
version:        hw:1.2 fw:1.1(May 17 2011 11:46:51)
band time:      17/05/11 15:41:18
accelerometer:  rate 50Hz, scale 8g
calibration:    (gain,offset) x(20, 30) y(40, 50) z(60, 16793)


All done

The Bioband (blue2) may now be unplugged

Press return to finish
```

It has been invoked using the -autodown and -nocoll options

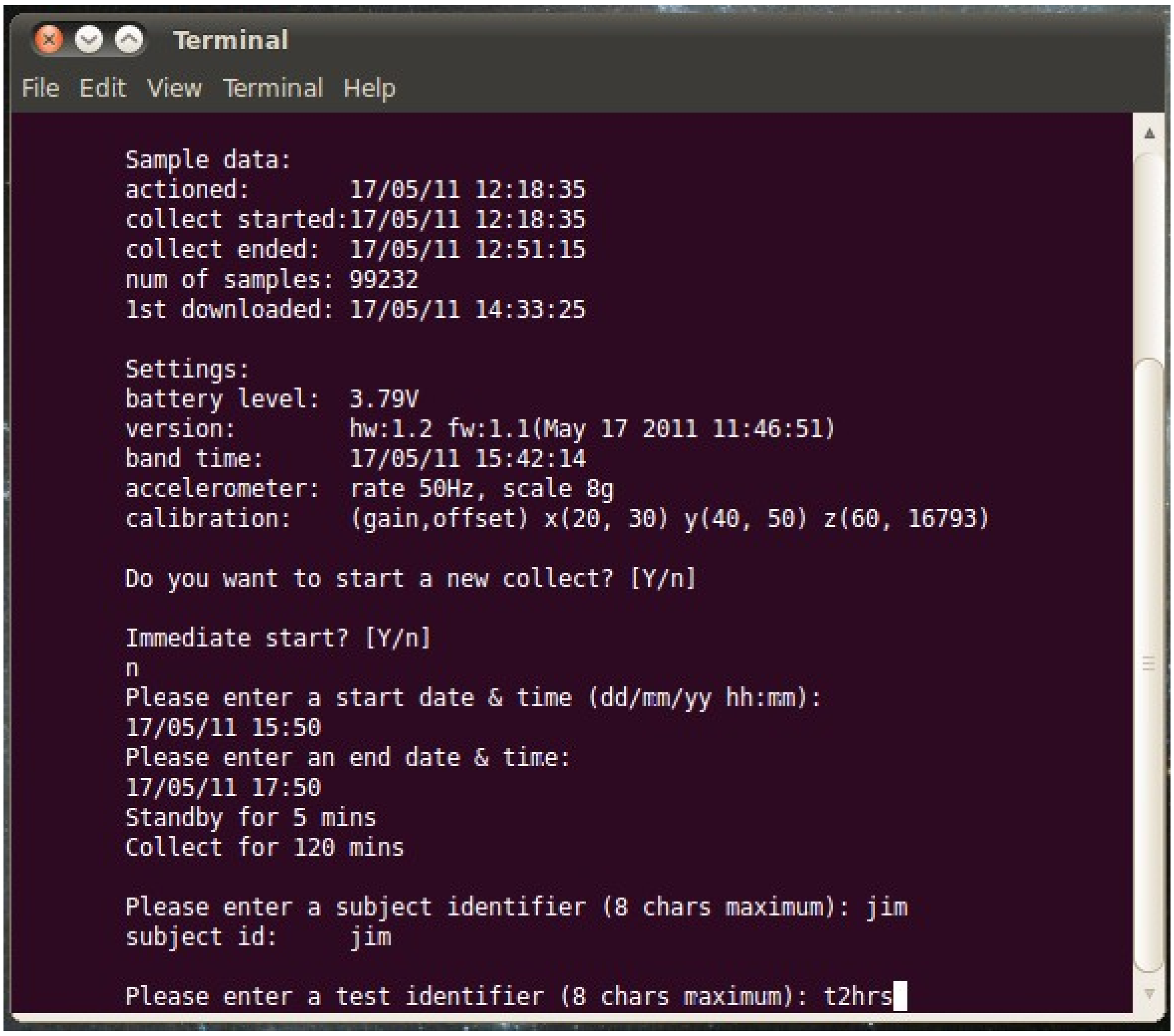
The demo code can be configured to manually question or automatically download a raw file from a band when it is connected (only if the raw file hasn't been downloaded before, given by the 1st downloaded time).



```
        Requesting details from the band ...

        Identities:
        band id:        black1
        subject:        jim
        test:           t6hrs
        centre:         uk

        Sample data:
        actioned:       16/05/11 14:42:54
        collect started:16/05/11 14:42:54
        collect ended:  16/05/11 21:19:33
        num of samples: 1188045
        1st downloaded: not set

        Settings:
        battery level:  3.73V
        version:        hw:1.2 fw:1.1(May 16 2011 14:09:21)
        band time:      17/05/11 15:36:50
        accelerometer:  rate 50Hz, scale 8g
        calibration:    (gain,offset) x(10, 20) y(30, 40) z(50, 60)

        0%.........10%.........20%.........30%.........40%.........50%.........60%........
.70%.........80%.........90%.........100%

        num samples received:   1188044
        expected total:         1188044

        Successfully downloaded raw file: /home/mrc/bioband2/downloaded_data/black1_jim_t6
hrs_uk.raw


        All done

        The Bioband (black1) may now be unplugged

        Press return to finish
```

Example output showing the successful download of a short raw file from the Bioband, the automatic naming of the raw file and placing into a specified download directory.

The demo code can also be configured to ask questions relating to starting a new collection, in case a quick turnaround of the band is needed



```
        Sample data:
        actioned:       17/05/11 12:18:35
        collect started:17/05/11 12:18:35
        collect ended:  17/05/11 12:51:15
        num of samples: 99232
        1st downloaded: 17/05/11 14:33:25

        Settings:
        battery level:  3.79V
        version:        hw:1.2 fw:1.1(May 17 2011 11:46:51)
        band time:      17/05/11 15:42:14
        accelerometer:  rate 50Hz, scale 8g
        calibration:    (gain,offset) x(20, 30) y(40, 50) z(60, 16793)

        Do you want to start a new collect? [Y/n]

        Immediate start? [Y/n]
        n
        Please enter a start date & time (dd/mm/yy hh:mm):
        17/05/11 15:50
        Please enter an end date & time:
        17/05/11 17:50
        Standby for 5 mins
        Collect for 120 mins

        Please enter a subject identifier (8 chars maximum): jim
        subject id:     jim

        Please enter a test identifier (8 chars maximum): t2hrs
```

The example above, being manually run (./demo) in a normal terminal, shows a collection being configured not for an immediate start but for a point of time in the future. The user is asked to enter subject and test identifiers, also centre id if not specified already on the command line.

These demo program actions are configured by the optional command line parameters

-snum <serial number>	specific Bioband serial number to connect to
-autodown	always download if data not already downloaded
-downdir <dir_path>	directory to place the download files in
-nocoll	don't ask collect question
-centre <id>	set the centre id to use for any sample collection

The MRC program

The mrc program is the original code used for development and testing of the band, given its use, it has grown to provide a range of requests, mainly for development, using the Bioband APIs to manipulate the device at a individual element level compared to Demo

The main useful commands are:

```
./mrc -p 5 -l 5 -sj jim -tst test1 -cen uk
```

This would start a collect for 5 minutes after a pause for 5 minutes, labelling the data with subject(jim), test(test1) and centre id(uk)

```
./mrc -uraw
```

This can be used to download the raw file from the band into the current directory, naming the file according to the band, subject, test and centre ids.

Alternatively if a specific filename is required

```
./mrc -raw collection2.raw
```

The raw options can be used with rsum, rbl, rtl & rcsv options if wish to produce particular detail files at the time of download from the Bioband.

```
./mrc -rsum -rbl black1_battery -raw collection2.raw
```

Offline working

The mrc program can also be used, when not connected to a band, to extract details from a raw file

For example

```
./mrc -rsum -rbl black1_battery -rtl black1_battery -rcsv black1_battery -fraw
black1_jim_t6hrs_uk.raw
```

The above command will output a summary, showing what is in raw file black1_jim_t6hrs_uk.raw. It will also output battery, temperature and csv file details into separate files as named.

Note the -fraw, -raw or -uraw commands must come at the end of the command line since the invocation of the command causes the processing of the raw file

The full set of mrc commands are shown below:

```
./mrc ?
MRC
Version: May 16 2011 14:37:28

Cmd line params & default values:
Raw file command options
        -raw [filename] read raw image of data from band to file or screen
        -uraw read raw image from band and produce a uniquely named raw file
        -fraw <filename> read raw image of band data from file
        -rbl <filename> store raw battery levels to file
        -rtl <filename> store raw temperature levels to file
        -rdbg <filename> store raw debug to file
        -rcsv <filename> store csv output to file
        -uall create uniquely named files for bl,tl,dbg & csv from band
Band sampling command options
        -l collection time (in mins)
        -p standby time (in mins)
        -z set mode (0 = real, 1 = debug)
        -sj set subject id (max 8 chars)
        -tst set test id (max 8 chars)
        -cen set centre id (max 2 chars)
Other band commands
        -id set band id (max 8 chars)
        -bl read all the stored battery level measurements
        -tl read all the stored temperature level measurements
        -gdt current device time
        -gfdt get first download time
        -ef erase flash (remember this wipes id and subject as well)
        -gi get band id
        -gsj get subject id
        -gt get test id
        -gcen get centre id
        -gsb get stored samples size in bytes
        -gsp get stored samples size in pages
        -rp <page number (between 1 and value given by -gsp)> read page number
        -gbl get current battery level
        -gv get the versions
        -sac <rate 50,100,400,1000> <scale 2,4,8> set accelerometer config
        -rac read current accelerometer data rate and g scale
        -raf read accelerometer data rate and g scale for previous run
        -scal set calibration data, 2 * unsigned values (gain & offset) for each axis
        -gts go to sleep
Diag commands
        -bb read bad block info
        -nobkp reset bkp domain on band
        -bug read dbg info from band
```

Information sources

The following link and resources were used during the design and implementation of this project

Accelerometer LIS331DLH
http://www.st.com/internet/analog/product/218132.jsp

Nand flash HY27UF084G2B
http://www.hynix.co.kr/datasheet/eng/nand/details/large_11_HY27UF084G2B.jsp

Cortex m3 reference manual
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337g/DDI0337G_cortex_m3_r2p0_trm.pdf

STM32 resource page (click on the Resources tab)
http://www.st.com/stonline/stappl/st/mcu/subclass/1169.jsp

Openocd
http://developer.berlios.de/projects/openocd/

Helpful STM32 cross toolchain notes
http://fun-tech.se/stm32/gcc/index.php