# VALLEYFORGE TOOLCHAIN
Maintenance Manual

Zac Frank

January 24, 2012

## Abstract

The abstract goes here.

# Contents

Chapter 1

# Linux Environment

This section details the installation of the toolchain.

## 1.1 Installation

The following are assumed:

- The user has a firm grasp on C++ programming.
- The user has a rudimentary understanding of Bash scripting. (A good tutorial can be found at `http://linuxcommand.org/writing_shell_scripts.php`)
- The user knows how to use Git to update the remote repository (a tutorial can be found at `http://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-gu`
- The user has the latest version of Ubuntu installed. If not, the latest version can be downloaded from `http://ucmirror.canterbury.ac.nz/linux/ubuntu-releases/`. This image can be burned onto a disc, the computer booted from the disc, and on-screen instructions followed.

Once Ubuntu has been installed, we need to install the packages that do not come preinstalled with Ubuntu but are necessary for the toolchain to run. Most of these packages can be installed using the Debian Package repository. To install these packages, open the terminal, and enter:

```
sudo apt-get install byacc flex gcc-avr avr-libc
```

Next the toolchain will be installed.. Navigate to the directory where the toolchain will reside. In the command line, type

```
sudo git clone (PUT REPOSITORY URL HERE)
```

to download the toolchain to your working directory.

See the User Guide for instructions on how to use the toolchain.

# Chapter 2

# Directory Structure

The ValleyForge Folder contains eight subdirectories, each of which will be briefly explained, including each one's own subdirectories. More complex aspects are described in more detail in later sections.

## 2.1 bin

This directory contains the binaries that are produced when a component is built. Binaries are copied from their compile directory in tmp, and are named after the component. If an accompanying bootloader is also being built, this is also placed in this directory with the name `COMPONENTNAME_bootloader.hex`. The toolchain currently has no mechanism with which to delete binaries from this folder so at these stage they must be deleted manually (although when a component gets rebuilt the old binary is properly replaced by the new automatically).

## 2.2 bld

The bld subdirectory contains (almost) all the bash scripts which manipulate the toolchain. There are three scripts which provide the main interface to the user. They call other scripts in other subdirectories and are the only scripts necessary to be called directly.

- `VFstart`. This script should be the first called. It is used to create new components, edit, delete, create new files, change user info, e.t.c. A detailed description of this script can be found in section §4.1.
- `build`. This script is used to build components and bootloaders. When run without any arguments, it builds the active component. A detailed description can be found in section §4.2.
- `resettc`. This script resets the toolchain to its default values, i.e erases user info, default component. Does not erase the source directory.

Two other files are contained in this directory:

- `build_configs.cfg`. This file is used by many other scripts. Usually in the form of

```
source bld/build_configs.cfg
```

which runs the file as a script. However it only contains bash functions, each of which does nothing except assign various fields. Therefore when the user calls the function, a collection of variables become assigned, corresponding to the build configuration selected. Build configurations correspond to the different combinations of platform and micrprocessor, and each has its specific variables to be set (such as Makefile used, flash address to start from).

- `default_user_config.cfg`. A file which the user_config file gets restored to when resettc is called.

### 2.2.1 bld/common

Located here are small scripts that are used by many other scripts throught the bld subdirectories.

- `clear_bconf_vars`. A safety mechanism. Clears build configuration variables so that the wrong values are not used by accident.
- `def_colours`. Each script, when displaying text to the user, uses a colour scheme for different types of messages, defined in this file. Colours are explained in more detail in section §3.2.
- `load_build_configs`. This tests to see if a valid build configuration exists, and loads it (sets the appropriate variables.

### 2.2.2 bld/create

In this directory there are four scripts which are called when a new instance of something is created.

- `create_component`. Creates a new component. Brings up an interface prompting the user to enter component details, then creates a new subdirectory in src, and asks to create source files. A detailed description can be found in section §4.3.
- `create_files`. Creates new source files and places them in the specified source directory. A detalied description can be found in section §4.4.
- `create_library`. Similar to creating a new component, creates a new library. Also asks to create source files after setting up the library.
- `create_library_files`. Creates new library files from templates and places them in the library source path.

### 2.2.3 bld/delete

These scripts are used for deleting components and libraries.

- `delete_component`. Brings up a list of the available components. Deletes the source files and folder for the specified component. If the component is the active one, resets the active component.
- `delete_library`. Deletes a specified library from the toolchain.

### 2.2.4 bld/edit

- `edit_user_config`. Changes toolchain variables, such as user name and subsection.

### 2.2.5 bld/other

- `first_run`. This script is called when the toolchain is first run, and prompts the user to enter their information.

### 2.2.6 bld/preprocess

- `preprocess`. This script is used to change the contents of source files. It edits template files before moving them to the source directory, and it inserts variables into source files before compiling at build time. A more detailed description can be found in section §4.5.

## 2.3 doc

This is where the user Guide and this manual are stored, as well as the latex files for both. To edit these files, you need to have a latex editor (you can Texmaker through the Ubuntu Software Center). Please keep both files up-to-date as necessary.

## 2.4 lib

User defined libraries are stored here. They can be accessed using #include lib/LIBRARY_NAME in component source files. The contents of this folder are not under version control.

## 2.5 res

This folder stores all the toolchain HAL libraries, bootloaders, makefiles, freertos files, and basically any external resources needed to build programs. The various sub-directories:

### 2.5.1 res/arm

This contains a makefile and other files specific to the OMAP4430. I'm really not too sure what's going on here, so hopefully Paul will have filled this in before I leave here. A detailed description of what's going on can be found in section [].

### 2.5.2 res/avr

This contains all files specific to the avr. Mostly hal and FreeRTOS files are contained in this subdirectory. The files at the top level:

- `bconf_specific_config`. This is a script called when a new component is created. Some options are specific to certain build configurations, and therefore need not be set with each component.
- `bload_specific_config`. This is a script called when a bootloader is added to a component. Some options are specific to certain bootloaders, and do not need to be set with each bootloader. For instance, the AVR bootloader configuration needs to know which EEPROM address to store the shutdown state flag in, whereas the AVR32 microprocessor has no EEPROM to store such a flag.
- `Makefile`. This makefile is used to compile and link all avr components. It is copied to the tmp directory at build time, preprocessed to edit various fields to those needed for the built component, and then run to compile the component.

There are three folders:

- `bootloader`. This contains source files for the avr bootloader, including a generic file and respective modules. At time of writing (18.01.12) it has not yet been implemented to communicate with a programmer, nor does it write to flash memory. It currently flashes an LED (D3) and then runs the application when a button (B3) is pressed.
- `freertos`. The FreeRTOS contains mostly common files, plus a few port-specific files. The port specific files for FreeRTOS on the various AVR microprocessors are all contained in this subdirectory.
- `hal`. The ValleyForge HAL is split into common header files, and port specific implementation .cpp files. The implementation for AVR is contained in this directory. At time of writing (18.01.12) the following modules have been implemented:

  gpio tc watchdog mem

  Creating new HAL is detailed in section §5.

### 2.5.3 res/avr32

Contained here are the port specific files for the UC3 microprocessors (so far just the AT32UC3C0512C). While the avr compiler is installed directly onto the computer and can be accessed via the terminal, the avr32 compiler is located within the toolchain, and is called from there.

There will be one directory initally in the toolchain, but the avr32comp.tar file gets expanded into another whenever the avr32 compiler is used.

The folder:

- `freertos`. This contains the AVR32 port specific FreeRTOS files, which are transferred to the compile directory at build time.

The two files:

- `avr32comp.tar`. This is an uncompressed tarball, it contains the AVR32 utilities used for building components for the AT32UC3C0512C.
- `Makefile`. This is the makefile used whenever a component is compiled for AVR32. Like the AVR makefile, it gets preprocessed at build time, specific to the build configuration and files being compiled.

### 2.5.4 res/common

This contains files common to all architectures. It could probably contain freertos, although currently this is in its own subdirectory in main. Currently it contains the header files for the HAL. These get copied to the build directory at compile time, along with the architecture specific hal files from the respective directories.

### 2.5.5 res/freertos

Here are contained the common files for FreeRTOS which are copied to the build directory at compile time. As well as this, there is the folder FreeRTOSV7.1.0 which contains all the latest files from FreeRTOS, including all current official ports.

### 2.5.6 res/templates

All the template files which are used to outline the structure of a new component are stored here. They are fairly self-explanatory. The freertos template contains a simple task to build on. FreeRTOS is explained more clearly in section §7.

### 2.5.7 res/vendor

This is where external libraries should be stored. It's not really up and running properly at this stage, but the idea is that eventually we can update external libraries with git into this folder, then put our own versions of these libraries into lib, and git merge them whenever there are updates. At the moment this contains the Atmel Software Framework, which is a massive collection of libraries and utilities for both AVR and AVR32. For developing new functionality, it is often useful to look in there first.

## 2.6 src

Here is where the user's source code is placed, as well as other component specific files, such as the component configuration file, and the FreeRTOSconfig.h header file, which specifies FreeRTOS parameters, should the user be implementing FreeRTOS as part of their project. Currently, the code is all placed into one folder named after the component, as a subdirectory of src. However, future functionality could be added to support subdirectories of the component folders, in case the user wanted to include sub-modules in different folders for clarity. At compile time, the source files in the src/component folder get copied to the tmp directory and preprocessed to have all the correct information.

## 2.7 tmp

The tmp directory is used as a place to compile and link the components and bootloaders. At the start of every build session, it is cleared of all files and folders. Files get copied in from various sources, and preprocessed, then compiled and linked. The resulting hex file is copied into the bin directory. The directory is also cleared after building, but the files may be retained by adding the flag “-r” to the build script command (or “–retain”).

## 2.8 var

Currently all this contains is the user info in the form of config.cfg. It has a few fields which are user-specific. May be added to. Not really much else to say about it.

## 2.9 TODO

This is updated with everything that needs to be added or changed with the toolchain. When something is fixed, remove it from the TODO file, and mention it in the comment of your the git commit.

# Chapter 3

# Standard Bash Structures

Although an understanding of bash scripting is assumed, there are many ways to acheive certain functionalities in bash. However, we want a standard for how certain aspects are carried, to make the scripts easier to read and more easily maintainable by multiple engineers.

## 3.1 Script location

At the start of every script we need to figure out what the absolute path to the Toolchain directory is. This is the one from the create_files script:

```
# Determine what the absolute path to the root of the toolchain is.
SCRIPT=`readlink -f $0`
SCRIPTPATH=`dirname $SCRIPT`
TCPATH=$(echo $SCRIPTPATH | sed 's/\/bld.*//')
```

After this, the variable ${TCPATH} can be used anywhere in the script to point to a specific file or folder within the toolchain. The toolchain standard is to use this absolute path to files and folders, rather than a relative path (such as ../../bin/myfile.hex).

## 3.2 Colours

The ValleyForge Toolchain regularly prints messages to the terminal. In order to make these messages clear and their type discernable, different types of messages are given different colours. All scripts, after determining the location of the toolchain, have the code:

```
source $TCPATH/bld/common/def_colours
```

which loads the constants defined in the colours script. These colours are then used as such:

```
echo -e -n "${GREEN}Enter basenames for the files to be created: (
    Space separated) ${NO_COLOUR}"
```

Note that the ${NO_COLOUR} tag is always used at the end of a sentence, to clear the applied colour. This is important, as any messages that do not come from the toolchain will all be in the default terminal text colour.

The colours used are

- `Green`. This is used for prompts, when the user is required to input a parameter or answer a question.
- `Cyan`. Toolchain status messages use this colour, to inform the user what is happening in the background.
- `Yellow`. Warning messages use yellow to warn that something is not as expected, but could well be on purpose so is not considered an error.
- `Red`. Red is for error messages.

For vital information, use the colour tags with BOLD at the front, such as `BOLD_RED` for fatal error messages.

## 3.3 Script Parameters

When a script is , it often has some parameters passed to it to determine its course of action. To demonstrate the method of passing these parameters and showing the user how to manipulate them, an example using the create_files script is provided.

```
PROGNAME=${0##*/}
SHORTOPTS="hbn:c:t:"
LONGOPTS="help,batch,name:,component:,type:"

# Use 'getopt' to parse the command line options.
ARGS=$(getopt -s bash --options $SHORTOPTS --longoptions $LONGOPTS --
    name $PROGNAME -- "$@")
eval set -- "$ARGS"

# Handle the parsed parameters.
while true; do
        # Select the appropriate behaviour for each parameter.
        case $1 in
                -h|--help)
                        # Just print the usage message and then exit.
                        usage
                        exit 0
                        ;;
                -b|--batch)
                        # Select batch mode.
                        BATCHMODE=1
                        ;;
                -n|--name)
                        # Specify the name of the files to create.
                        shift
                        FILENAMES="$FILENAMES$1␣" # NOTE - The space
                            is intentional!
                        ;;
                -c|--component)
                        # Specify the name of the component to add
                            files to.
                        shift
                        COMPONENT="$1"
                        ;;
                -t|--type)
                        # Specify the type of files to create.
                        shift
                        TYPE="$1"
                        ;;
                --)
                        # We're done parsing options.  Anything else
                            must be parameters.
                        shift
                        FILENAMES="$FILENAMES$*␣" # NOTE - The space
                            in intentional!
                        break
                        ;;
                *)
                        # Anything else must be parameters.
                        shift
                        FILENAMES="$FILENAMES$*␣" # NOTE - The space
                            in intentional!
                        break
                        ;;
        esac

        # Advance on to the next parameter.
        shift
done
```

To add a new parameter, for example "size", we would first add into long and short opts:

```
SHORTOPTS="hbn:c:t:s:"
LONGOPTS="help,batch,name:,component:,type:,size:"
```

The ":" character indicates that it expects something after the option is called. If it was a flag which didn't need a parameter, but was simply a flag in and of itself, it would not need this ":".

Next, add it to parameter handling section:

```
-t|--type)
                        # Specify the type of files to create.
                        shift
                        TYPE="$1"
                        ;;
-s|--size)
```

```
                        # Specify the size of files to create
                        shift
                        SIZE="$1"
                        ;;
```

and add it to the usage menu in the "usage" function:

```
Options:
       -h --help                      Show this message.
       -b --batch                     Operate in 'batch mode',
          wherein there are no interactive prompts.
       -n --name <File Name>          Specify the name of the files
           to add to the current project.
       -c --component                 Specify the name of the
          component to add the files to.  Defaults to the 'active
          component'.
       -t --type <File Type>          Specify the kind of files to
          create (either 'C' or 'CPP').
       -p --platform                  Specify the platform for
          which the files will be created (BareMetal/freertos)
       -s --size                               Specify the size of
          the files to create.
EOF
```

You can now use the variable ${SIZE} in the rest of the script.

## 3.4 User Prompts

Often a script will ask for user input before it can continue. There are four main types of prompts:

- `Given Words, First Letter Underlined`. This is used in the VFstart script near the beginning, where preset and unchanging options are given. The user hits the key corresponding to the underlined letter in each option.

- `Enumerated List`. A list of options is given with a number before each option. The user enters a number and presses the Enter key. This is useful because more parameters can be added, or entirely new ones added through an external variable, very easily and elegantly.

- `Text Input`. Here the user enters some text, followed by the Enter key, which is then used by the toolchain. An example where this is used is when the user is asked to enter the name for their new component.

- `Yes/No Prompts`. "Y" or "N" key is pressed (Either capital or lower-case).

### 3.4.1 Given Words

An example from the script VFstart:

```
while :
        do
        # Print the submenu prompt to the user.
        echo -e "${GREEN}Please select an option:\n${NO_COLOUR}"
        echo -en "\tCreate ${UNDERLINED}C${NO_COLOUR}OMPONENT    
            Create ${UNDERLINED}F${NO_COLOUR}ILES    Create ${
            UNDERLINED}L${NO_COLOUR}IBRARY    ${UNDERLINED}B${
            NO_COLOUR}ACK"

        # Read a single character input from the user and select the
            appropriate response.
        read -s -n 1
        echo -e "\n" # NOTE - This is required since read won't add a
             newline after reading a single character.
        case "$REPLY" in
                "C" | "c" )
                        # Run the create component script.
                        bash $TCPATH/bld/create/create_component
                        ;;

                "F" | "f" )
                        # Run the create files script.
                        bash $TCPATH/bld/create/create_files
                        ;;

                "L" | "l" )
                        # Run the create library script.
                        bash $TCPATH/bld/create/create_library
                        ;;

                "B" | "b" | "Q" | "q" | "X" | "x" )
                        # Go back on level.  Force redrawing the top
                            level menu, even if we aren't in block
                            mode.
                        REDRAW_TOP=1
                        break
                        ;;

                *)
                        # Any other option is invalid.  We print a
                            message to that effect and try again.
                        echo -e "${RED}Invalid choice. Try again or 
                            press Q to quit.\n${NO_COLOUR}"
                        continue
                        ;;
        esac

        # We're done here.
        break
done
```

The options for the read function here dictate that it should only wait for one character from the user before it parses it.

### 3.4.2 Enumerated List

This is a compact method and is shown below in the create_component script for choosing a platform:

```
# Create a menu of choices and have the user select one.
select PLATFORM in $PLATFORMS
        do
                # Check if the selected platform is actually valid.
                PLATFORM=$(echo "$PLATFORMS" | grep -w -o "$PLATFORM"
                   )
                if [ -z "$PLATFORM" ]; then
                        # The selected platform was not in the list
                           of platforms, so the user is apparently a
                           moron.
                        echo -e "${RED}Invalid choice.  Try again.\n$
                           {NO_COLOUR}"
                else
                        # A legitimate option was selected, so we can
                            go now.
                        echo -e "${CYAN}Selected platform $PLATFORM.\
                           n${NO_COLOUR}"
                        break
                fi
        done
```

In this case, the variable ${PLATFORMS} is a set of words separated by spaces. The list of words is enumerated, and the option selected is assigned to the variable ${PLATFORM}.

### 3.4.3 Text Input

An example is given below:

```
while :
do
        # We will need to prompt the user for the name of the
           component to create.
        echo -e -n "${GREEN}Enter a name for the component to be
           created: (No spaces) ${NO_COLOUR}"
        read
        echo -e ""
        # Check the user actually entered something.
        if [ -z "$REPLY" ]; then
                # We'll just prompt again.
                continue
        # Check to see if this name is already taken.
        elif [ -d "$TCPATH/src/$REPLY" ]; then
```

```
                # The name provided is already taken.  Prompt the
                    user to choose an available name.
                echo -e "${RED}The component '$REPLY' already exists.
                    Please choose another name.\n${NO_COLOUR}"
        else
        # The name is probably legit, so we move on.
                COMPONENT=$REPLY
                break
        fi
done
```

### 3.4.4 Yes/No Prompts

```
echo -e -n "${GREEN}Do you wish to make this the current component? (
    Y/N)${NO_COLOUR}"
read -n 1
echo -e "\n" # NOTE - This is because the read command won't put a
    newline after it reads a character.

# If they responded YES, then set the current component.
if [[ $REPLY =~ ^[Yy]$ ]]; then
        # We want to set this as the current component.
        sed -i "s^\(tc_curr *= *\).*^\1$COMPONENT^" $TCPATH/
            $USER_CONFIG_FILE
fi
```

## 3.5 Style

Try to keep the style of coding similar to that already established. Comments should appear above the relevant line, rather than to the right. Comments should be frequent and verbose.
Variables should be in all caps, with underscores to separate words in one variable.
Functions are lower case, with underscores to separate words.
Tabbing should follow standard coding practice.

# Chapter 4

# Script Descriptions

In this section, some of the more essential and complex scripts are described in detail, with an idea of how they work, and what aspect of the overall picture they relate to.

## 4.1 Script: VFstart

Location: `ValleyForge/bld/VFstart`

### 4.1.1 Summary

VFstart is the main interface to the toolchain when not building a component.

### 4.1.2 Prologue

It starts off, like all the scripts, by finding the file location of the toolchain. It then defines some constants, in this case locations of other relevant files, and imports the colour scheme.

### 4.1.3 Functions

`Usage` – This function is simply a print to terminal. It shows the user the usage of the script, particularly the parameters that can be passed to it and what they do.

### 4.1.4 Script Proper

#### Resetting Fields

Resetting fields is a safety measure to ensure that no wrong parameters are passed around, and an error can be picked up easily if the variables are empty. Simply entering:

```
VARIABLE=
```

clears the variable, meaning that if it is not reassigned, an empty value will be assigned to it. It is easy to check for a variable being empty, whereas if it has the wrong value assigned to it, this is not as easy to pick up.

#### Parsing Parameters

The script then parses command line parameters, as described in section §3.3. The parameters for VFstart are:

- `help`. A standard option for any script, this prints out the usage text to the terminal
- `block`. If the user specifies the parameter "Block Mode", then once the user has performed one action, it will prompt for another, rather than the default action of exiting the interface once an action (such as creating a new component) has been performed.

#### Loop

The while loop serves to enable the block mode, where once one task is completed, another can begin. The variable "REDRAW_TOP" is used to bring the user back to the starting interface if called for at certain points during the script. The script checks the state of the variable and goes back to the starting interface if set. It is reset when the while loop starts again.

#### User Config File

The toolchain then does some basic checks to discern the nature of the environment. It checks to see if a user config file is present, creates one if there isn't, and asks the user to fill out their info if it hasn't been set. Currently, all that this does is put their name at the start of source files when they are created, but future functionality could be added (like company, aspect working on).

#### Active Component

The script checks the user config file stored in `var/config.cfg` to set what the field `tc_curr` is set to. If it does not correspond to a folder in the src directory, the user is warned and advised to set a component to be active, and told how to create new components.

#### Opening Prompt

The script gives the user four options, and the user selects one by pressing the key corresponding to the underlined letter in the word.

- **Create**. To create a new component, library, or files. See section §4.3.
- **Edit**. To edit user info, active a component, or edit toolchain settings.
- **Delete**. To delete components or libraries.
- **Quit**. Yep.

### Create Menu

The script gives the user four options:

- **Create Component**. This runs the create_component script located in bld/create. A description of the script can be found in section §4.3.
- **Create Files**. This runs the create_files script located in bld/create. A description of the script can be found in section §4.4.
- **Create Library**. This runs the create_library script located in bld/create.
- **Back**. Yep.

### Delete Menu

The script gives the user three options:

- **Delete Component**. This runs the delete_component script located in bld/delete.
- **Create Library**. This runs the delete_library script located in bld/delete.
- **Back**...

### Edit Menu

This menu gives the user four options:

- **Activate Component**. This lists the existing components (determined by folders existing in the `src` directory) and lets the user choose one to be the active component.
- **Reconfigure Active Component**. This runs the create_component script with the name of the active component passed as a parameter.
- **Edit User Config**. This runs the script `edit_user_config`, located in `bld/edit`.
- **Back**...

### Epilogue

The running of the script is checked for errors. If there are any errors and the script is in Block mode, then the script will ask for user confirmation before proceeding, because the top of the while loop clears the screen, erasing the error messages that the user would want to see.

The script checks the REDRAW and BLOCKMODE variables and reruns the while loop or exits the script as desired.

## 4.2 Script: build

The build script gets run whenever the user wants to compile their component into a binary to upload to their device.

### 4.2.1 Summary

Whenever the user wants to compile their component, they compile it using the build script. It is set up such that one only has to run this script, perhaps with a few parameters, to successfully build the component. With no parameters passed, it simply builds the actice component. The paramaters that can be passed to this script currently are:

- `help`. This prints the usage message, informing the user how to use the script.
- `all`. This builds all the components, one after the other. If one component fails to build, the others will still be attempted.
- `retain`. Often, to debug preprocessing errors, or inclusion errors, it is useful to look at the code that is actually being compiled, as it may differ from how it is found in the source folders. The retain functionality does not delete the files from the tmp directory, so that the user may look at them. They do, however get deleted straight away next time the build script is run.
- `loader`. This builds the corresponding bootloader as well, if there is one. It compiles in a subdirectory of `tmp_src` called bootloader, and the resulting hex file is called COMPONENT_NAME_bootloader.hex and also placed in the bin directory.
- `name`. This will build the specified component only.
- `nohal`. While hal aspects are not yet complete, it can be useful to build components without any use of the hal. This makes sure the hal files are not put into the build directory.

### 4.2.2 Prologue

As per usual, the script first finds out where it is located, and the absolute path of the toolchain, defines a few constants (mainly file locations), and sources the colour scheme (see §3.2), as well as the preprocessing functions (see more about the preprocessing script in section §4.5)

### 4.2.3 Functions

the build script has quite a few functions...

#### makeavr

This function is responsible for compiling and linking avr files into a hex binary input.
First it copies the makefile to be used from its source, in this case `res/avr`.

It then checks to see if a linker script has been specified for this particular build configuration. If so, it copies it from the location specified in the build configuration to the `tmp_src/component` directory, used for compiling files.
The build script has already copied all relevant source files to the build directory from various sources. Now the makeavr function creates a list of all the source files to be built, and their type. Next, the function edits the makefile, replacing strings such as "BUILD_INSERTS_C_FILES_HERE" with the relevant entries. This is contrast to the replacements that follow, which erase thing to the right of the "=" and therefore can't be used to add to a line. If a variable is empty, the string is simply replaced with nothing.
The makefile is then run. All makefile output is printed to the screen. If the makefile returns with an error, the whole script exits and doesn't attempt to do anymore.

#### make_avr_bootloader

This function does the same as the above in section §**??**:makeavr except for the avr_bootloader. Different make functions are used for each because the makefiles are different and need different fields replaced, however, it probably wouldn't be too hard to combine all these make functions together.

#### makeavr32

Same as above, see section §4.2.3:makeavr.

#### makearm

Same as above, see section §4.2.3:makeavr.

#### inflate_avr32cc

The AVR32 compiler is kept in a tarball, and is inflated whenever required. This function simply inflates all relevant binaries.

#### get_rtos_files

This function retrieves the relevant rtos files for respective build configurations and places them in the build directory.

#### get_suitable_libs

If the user has specified a library to be included in the component, this function retrieves the relevant files and copies them into the build directory. In the tmp directory these files are placed in a subdirectory called "lib".

#### get_hal

Retrieves the relevant hal files and places them in the build directory (`tmp/tmp_src/hal`). It first checks to see that all necessary parameters are present in order to implement hal features. Then it copies the base hal files, including semaphores, a hal layout, and a target config file, which are always implemented when the hal is used.
It then checks which hal elements are specified to be included in the corresponding build configuration, and copies these over too. They are comprised of a header file and a target specific c++ file. All hal files are preprocessed to ensure the correct values for the build configuration.

#### usage

This prints a message to the terminal outlining how to use the script, which parameters can be passed to it. To see the parameters, see the build summary in section §4.2.

### 4.2.4 Script Proper

First everything in the `tmp/tmp_src` directory is deleted. The parameters are then processed. How this process works and can be altered is shown in section §3.3.
If the "all" parameter is not specified, the config file is checked to see what the active component is. The variable "`NAME`" is set to have the name of the component.
If the "all" paramter is specified, then the variable "`NAME`" is set as the names of all components, spaced apart.
The `build_configs.cfg` is sourced. Note that at this stage no code is run here, since it is comprised entirely of functions that need to be called.
A for loop is run, iterating through all components to be built. So usually just once.
The component config file is located. If it is not found, the user is prompted to create one. Else it is sourced, i.e its variable values read in. Any build config variables are cleared by running the `build_config_vars` script. If you add any variables to any of the build configurations, make sure this script clears them, as otherwise values may be retained and lead to errors.
The build script then finds and copies all the user source files into the build directory `tmp/tmp_src`, if there are none it stops building the component. All these files are then preprocessed. See a detailed description of the preprocess script in section §4.5. The two functions, `get_suitable_libs` (see section §4.2.3) and `get_hal` (see section §4.2.3) are then run.

The actual make function is then run. These functions are described above from section §4.2.3. A check is then done, and the bootloader compiled if desired. This is done pretty much the same way as the application. The build directory is then cleared (or not if directed not to) and the script finishes.

## 4.3 Script: Create_component

This script is used to create new components for the user's version of the toolchain.

### 4.3.1 Summary

This script is generally called from within other scripts rather than being called by itself, although it can be called by itself, with no parameters. It is responsible for creating new components, including building a source directory, making the config file, setting up the bootloader, and calling the `create_files` (section §4.4) script, if requested to create source files. The script has the following available parameters:

- `help`. Runs the usage function, showing the usage of the script.
- `batch`. Runs the script in batch mode. This means there are no prompts, and all parameters must be specified in the script parameter set.
- `component`. For batch mode, specifies the name of the component.
- `subsystem`. For batch mode, specifies the name of the subsystem.
- `target`. For batch mode, specifies the name of the target.
- `platform`. For batch mode, specifies the platform for the component.
- `loader`. For batch mode, specifies the bootloader for the component.

### 4.3.2 Prologue

Like the others, we first find out our location, define some constants, and source the colours and preprocessing scripts.

### 4.3.3 Functions

#### choose_target

This searches the "`build_configs.cfg`" file for application build configurations, extracts the target name from each, and lists them in a list to select (selecting method can be found in section §3.4.2). The variable "`TARGET`" is assigned the name of the target that the user chooses.

#### choose_platform

This searches the "`build_configs.cfg`" file for application build configurations for the specified target. It then lists the platforms available for the target, the user selects one, and it is recorded in the variable "`PLATFORM`". The "cut" function takes the names of all the functions in the `build_configs.cfg` file, and cuts everything else away leaving only the name of the platform.

#### choose_bootloader

This searches the "`build_configs.cfg`" file for bootloader build configurations for the specified target. If none are available, it skips onwards, else the user may choose one from an enumerated list.

#### run_bconf_specific_config

Some build configurations require extra setup specific to them, so for build configurations where this is the case, an extra script kept in the res directory is run. For example for the avr bootloader, an extra script is run asking the user to specify which pins are to be used by the bootloader for the leds or for the input pin.
This function checks to see if the build configuration specifies any extra script to run, and runs it if that is the case.

#### run_bload_specific_config

Same as above, except for the bootloader rather than the application. The two functions are split because there are separate configurations for application and bootloader build configurations.

#### usage

### 4.3.4 Script Proper

Fields are reset, and parameters parsed (see section §3.3 for how this is done).
The user is asked to provide a component name. This name is tested for legitimacy, then the same is done for the subsystem name.
The user then chooses the target, platform, and bootloader. Then the component's source directory is created, and the component config file placed in this directory.
The user is asked whether they want to make this new component the active component. If so, the value in the `config.cfg` file in `var` is edited.
Next they are asked whether they want to create files for the new component. If so, the `create/create_files` script is run (see section §4.4).

## 4.4 Script: Create Files

This takes some template source files, copies and edits them, and places them into a component's source directory.

### 4.4.1 Summary

The `create_files` script uses the template files located in `res/templates` (see section §2.5.6). When the user specifies new files to be created, it edits values, such as subsystem, user info, component name, target name, and lays out a style to follow. It creates a header file and a source file (can be c or c++). The parameters are

- `help`. Runs the usage function, showing the usage of the script.
- `batch`. Runs the script in batch mode. This means there are no prompts, and all parameters must be specified in the script parameter set.
- `name`. For batch mode, specifies the name of files.
- `component`. For batch mode, specifies the name of files.
- `type`. For batch mode, specifies the of file (C or C++).

### 4.4.2 Prologue

Starts off finding its own file location and that of the toolchain. it then defines some constants, in this case paths to the template files, as well as the user config file and the `build_configs.cfg` build configurations file. It sources the colours and preprocessing functions.

### 4.4.3 Functions

#### Usage

This prints a message to the terminal which tells the user how to use the script, listing its parameters can giving a summary of its purpose and usage.

### 4.4.4 Script Proper

A few fields are reset, and then the parameters are parsed (see section §3.3).
Assuming we aren't in batch mode, the user is prompted for a name for the pair of files. If a component has been specified, its config file will be read.
Else the user config file is read, the name of the active component extraced, and the active component used to create files for.
The user chooses the source file type (C or C++ so far) and the script does a few error checks.
Then the script reads in the build configurations, and tries to find a matching one. If one is found, the appropriate files are preprocessed using the `preprocess_template` script. There are four different main templates covering the possibilities of FreeRTOS or Bare Metal, and C or C++.
If the component uses FreeRTOS, and no `FreeRTOSConfig.h` file exists in the source directory, then it is copied from its location in res. FreeRTOS requires this header file to run.

## 4.5 Script: preprocess

This script is used to edit generic files to change them to application specific instances.

### 4.5.1 Summary

This script, like `build_configs.cfg`, has no script proper, and is just some functions. Each function corresponds to a different file being processed. In each file are different fields to be edited.

The functions go through the various files, replacing strings such as "TOOLCHAIN_INSERTS_C_FILES_HERE" with the relevant entries, or replacing everything to the right of a specified "=". If a variable is empty, the string is simply replaced with nothing.
The only parameter passed to any of the functions is the path of the file to be preprocessed.

Each function uses sed to find and replace. Sed is hard to use, just find examples in these scripts and modify them if you are not familiar with sed.

## 4.6 Script: buildconfigs.cfg

The build configs are simply functions that are called, assigning values to specified fields. The variables that are mostly assigned are:

- `BLOAD_NAME or BCONF_NAME`: The string that is the display name for this build configuration
- `C_COMPILER`: The compiler used to compile C files, i.e what is placed in the makefile in this space.
- `P_COMPILER`: The compiler used to compile C++ files, i.e what is placed in the makefile in this space.
- `OBJCOPY`: The program used to convert output files.
- `OBJDUMP`: The program used to inspect object files and their properties.
- `SIZE`: The program used to measure the size of output files.
- `MAKEFUNCTION`: The function within build used to build this configuration.
- `MAKEFILE`: The path to the makefile used to build this configuration.
- `SOURCEPATH`: The path to the source files for this configuration (not valid for all configurations, mainly just for bootloader and FreeRTOS).
- `MCU_CODE`: The mcu code that the compiler receives to know which target to compile for.
- `IOHEADER`: The name of the io file to include.
- `STDINTHEADER`: The name of the stdint file to include.
- `BOOTSTART`: Only valid for bootloader configs, the start address of the bootloader.
- `PORT_COUNT`: All the gpio ports on the microprocessor being used.

- `BLOAD_SPECIFIC_CONFIG or BCONF_SPECIFIC_CONFIG`: The location of the script to be executed when a new instance of a component with this build configuration is created.
- `ACTIVE_MODULE`: For bootloaders, the module (isp,can,i2c) used to transfer data during bootloading.
- `MAX_EEPROM_ADDRESS`: Each target has a limited amount of EEPROM memory, this indicates how large it is and its max address.
- `LINKSCRIPT`: If the build configuration needs a special linker script, this shows its path.
- `FREERTOS_CONFIG_FILE`: The path to the FreeRTOSConfig.h file for this configuration.
- `CFLAGS`: Flags to add to the C compiler.
- `PFLAGS`: Flags to add to the C++ compiler.
- `AFLAGS`: Flags to add to the Assembler compiler.
- `HAL_HEADER_PATH`: Path to the HAL headers.
- `HAL_SOURCE_PATH`: Path to the HAL C++ files specific to this build configuration.
- `HAL_EN_LIST`: A list of the HAL modules to be used in this build configuration.
- `BCONF_SPECIFIC_RTOS_PATH`: When FreeRTOS is being used, the path to the port-specific FreeRTOS files.
- `BCONF_COMMON_RTOS_PATH`: When FreeRTOS is being used, the path to the commmon FreeRTOS files.

# Chapter 5

# HAL

The ValleyForge Hardware Abstraction library is implemented to give a common interface to various microprocessors and make code portable between devices.
At this stage four modules' interface structure has been completed: `gpio.h`, `mem.h`, `tc.h`, and `watchdog.h`. These have been implemented for one microprocessory only, the ATmega2560.
As well as this, there is a general HAL file which contains global interrupt enable/disbable, which links to the `semaphore.c/h` files, and a `target_config.h` header file, which has a lot of "#define"s outlining different kinds of constants for each target.
The HAL modules are implemented in C++.
Although only the ATmega2560 modules have been implemented, the AVR architectures will all use the same implementation file, by using the AVR preprocessor to select which parts of code are relevant to the specific microcontroller.

## 5.1 Modules

The currently implemented modules:

- `Gpio`. The gpio module uses a two-dimensional array of binary semaphores to hold assign GPIO pin usage. A user uses the `pin_grab` function to be given a class instance of a pin, which they can then manipulate until they relinquish hold on it again.
- `Tc`. The Timer/Counter Module does not use semaphores, due to the scarcity of timers and then need to use one timer for multiple applications. The user is given an instance of a timer class and is able to use this to manipulate the timers. Due to timers varying greatly between different microprocessors, this module is large and messy.
- `Mem`. The memory module writes to EEPROM. For the AVR architectures it is really no more than a layer between the `avr/eeprom` module and the user.
- `Watchdog`. This implements simple watchdog timers.

## 5.2 Size and Use

At this stage, the HAL is around 16kB, making it too large to fit into the bootloader section. Currently the smallest flash size of any of the micrcontrollers is 64kB (the ATmega64M1), meaning it can be used for any of the specified ValleyForge microprocessors. However, safety mechanisms in the code, as well as multiple layers of abstraction mean that it does not run super efficiently.

# Chapter 6

# Bootloader

The bootloader is used to transfer code to the microprocessor without use of an ISP programmer. Currently an AVR bootloader has been started, and its functionality is to be able to run a bootloader program, then switch to the application code when a button is pushed. There should be, floating around somewhere, a guide for creating bootloaders for AVR. It is very useful. Some aspects to look out for when programming the bootloader:

## 6.1 Boot Address

The boot address given in the datasheets is a **word** address, not a **byte** address. However the address entered into the makefile via the `build_configs.cfg` file is a **byte** address, and therefore the address needs to be converted. Take this into account if you want to change the boot size and therefore need a new boot address, or a re-implementing the bootloader. Make sure all the fuses are correct. Things that can happen with an incorrect boot address:

- The bootloader program may not run at all and go straight to the application code. This usually happens when the boot address given is lower than the actual. It means the program counter No-Ops all the way round to the starting address again, without ever running the bootloader code.
- Interrupts may not work. When you write a bootloader, it creates a new interrupt table at the bootloader start address. This means you will have two interrupt vector tables in your microprocessor. I had one instance where I set the bootloader address too high. The bootloader code still ran, but because the interrupt vector table was not where the microprocessor expected it to be, my interrupts did not fire.

## 6.2 Fuses

Make sure that the microprocessor's fuses are set to the correct values. This can not be done through a bootloader, it must be done using an ISP, JTAG, or HVPP programmer. The BOOTRST fuse is of course, important, but one must also make sure that the boot size is correct.

Be careful, also, to make sure the fuses aren't set to some strange clock speed. This bricks the micro. It can be fixed using HVPP, but is a right pain.

## 6.3 Registers

When the bootloader program has finished, make sure that before it starts running the application code, it resets hardware registers back to their original state. This can be done manually, or by resetting the micro (using the watchdog timer) and then going straight into the application code.

## 6.4 Merging Hex Files

The application and bootloader are built seperately, so to merge them into one hex file, one simply deletes the last line of one hex file and puts the other after it in a text editor. There are also scripts online that will do it for you.

# Chapter 7

# FreeRTOS

FreeRTOS is a small operating system for microcontrollers, that has been ported to work with all the Atmel processors being used for the ValleyForge project. Its website can be found at `http://www.freertos.org/`

## 7.1 Structure

FreeRTOS is split into two main sets of files: Common and Port-Specific. The common files contain all port-independent elements, those describing tasks and task handling.
There are three source files that form the FreeRTOS base: tasks.c, queue.c, and list.c. On top of this, there are a bunch of common header files that are included by these source files.
There are four port-specific files for each port. The port source file, port.c, a memory allocation file (this is not directly port-specific, there are three different types, and one uses the one best suited to one's microcontroller), and two header files, portmacro.h and projdefs.h.

## 7.2 Current Status

FreeRTOS has been made to run on all Atmel devices covered in the ValleyForge toolchain. Each was tested by running a single task that flashed an LED. There are many different options and variations within FreeRTOS, and not all have been tested or implemented.
Some playing around with clock settings and such is required. This should be editable in FreeRTOSConfig.h, which is placed in the user's source directory when they create a component that uses FreeRTOS.
The ATmega2560 port is different from the other AVR ports, as it needs to address a larger flash space. This has some strange effect. In any case, I had to download the port from AVR freaks, which used an old version. I finally managed to get the port working with the new version as it does now, but if the user tries to implement any other FreeRTOS functionality on the ATmega2560 and it doesn't work it may have something to do with this.

# Chapter 8

# Programming with the STK600

For debugging purposes, it may be that you will be using the STK600. I have decided to try and impart the wisdom I have learnt about this fickle device here.

## 8.1 Clock and Voltage

The clock speed and Target Voltage of the STK600 can be set using avrdude, or AVR Studio. Most devices operate between 3 and 5.5V, which can be supplied through the USB.
If the status LED is blinking orange, this means that you may have the wrong combination of routing and socket boards mounted. If it is blinking red, officially this means it is drawing too much current and that you have a short circuit on your board, but actually it just seems to do this at its whim, and still works, so I wouldn't read too much into it.

## 8.2 Programming AVR Devices

To upload a hex file using avrdude with the STK600, I:

Attached the 6-pin header for ISP programming, then type into the terminal:

```
avrdude -c stk600 -p m64 -P usb -F -v -U myfile.hex
```

The "-F" flag near the middle is the "Force" flag. This was because there was no mcu code for the ATmega64M1, so I used that given for the ATmega64, and used the force flag to program the device despite its warning that it didn't have the right identification code.
As stated in the bootloader description (section §6), make sure that if you are loading a bootloader on, that the linker has been given the correct bootloader start address.
Sometimes, having certain ports attached to LEDs or switches can cause errors when programming the micros using the ISP programmer.

## 8.3 Programming AVR32 Devices

Avrdude does not provide any support for AVR32 devices. For this reason I used avr32program. This used to come with the AVR32 Toolchain, but for some reason no longer does. It does come with the latest version of AVR32 Studio for Linux however, which is where I got it from.

This is how I loaded code:

Attach the JTAG connection using the 10-pin header.

```
avr32program -c USB:0048395C7318 --part UC3C0512C program -
    finternal@0x80000000,512Kb -v -e -F elf myfile.elf
```

While avrdude automatically erases the flash. before writing unless told not to, avr32program does not erase unless told to. Therefore we need the “-e” flag to tell it to erase the the flash. This is where I had the most trouble with having things connected to ports when the chip was starting up/being written to. Port B in particular was not very favourable.

## 8.4 Other Observations

Occaisionally the device will throw a fit and randomly flash LEDs that it supposed to be giving a solid signal to. Resetting the STK600 fixes this.
Generally ignore the status LED.
Don't ignore the VTarget LED. This indicates whether there is voltage to the device.
Be careful, make sure you know what you're doing before changing fuse values.
Make sure you take anti-static precautions when changing routing/socket boards.

# Chapter 9

# FAQ

Of course we don't really have any frequently asked questions at this stage, but I'll just post up a few things here that I might expect. I'm guessing throughout 2012 I may have to update this section, depending on how much people will be asking questions, but anyway:

## 9.1 I want to add a new HAL module. How do I do this?

The first thing you'll want to do is decide what functions are going to be available to the user. What will they require? Is it possible to implement these things on all the microprocessors? If not, which ones will it work on
Next, create your header file. A good example of the type of header file one could create would be gpio.h, found in `res/common/hal`.