VALLEYFORGE TOOLCHAIN
Maintenance Manual

Zac Frank

January 19, 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 here[].
- `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 [].
- `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 §2.2.
- `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 template files. A detailed description can be found in section [].
- `create_files`. Creates new template files and places them in the specified source directory. A detalied description can be found in section [].
- `create_library`. Similar to creating a new component, creates a new library. Also asks to create template 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 [].

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 [].

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[].

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 creat_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 VFstart

Location: `ValleyForge/bld/VFstart`

VFstart is the main interface to the toolchain when not building a component.
It starts off, like all the scripts, by finding the flie location of the toolchain. It then defines some constants, in this case locations of other relevant files, and imports the colour scheme. Functions come next, in this case only one, the usage function, which is the message that gets printed to the terminal if the user runs "`VFstart -h`" or "`VFstart --help`".

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.

The script then parses command line parameters, as described in section §2.3.

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. 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.

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.