Build Environment of EMSBench

Florian Kluge
University of Augsburg

October 2, 2015

Abstract

This document provides a technical description of the EMSBench implementation. Its aim is to give an understanding about how the build process is structured and to ease porting the embedded code to new platforms.

Contents

1 Directory Structure

The directory structure of EMSBench is depicted in figure 1. The root directory contains two python scripts for building the embedded engine management software (EMS) and trace generator (tg). Relevant library functions are located in the `/builder/` directory. The `/data/` directory contains driving cycles and car data that can be used during generation of the trace generator. The preprocessor for the trace generator is located in `/tgpp/`. It is intended to be executed on the host platform that is used to build the embedded code.

The platform-independent code for EMS and tg is located in the corresponding subdirectories inside `/embedded/`. The `/embedded/arch/default/` directory acts as a blueprint for the implementation of a platform-specific hardware abstraction layer (HAL). The required interfaces for the application-specific HALs are specified inside the `include/` directory. An actual implementation may use macros instead of functions. HAL implementations are located in the `hal-$APP/` subdirectories, more generic HAL code may be located in `hal/`. If your platform requires additional code/libraries, these should be placed in the `bsp/` subdirectory.

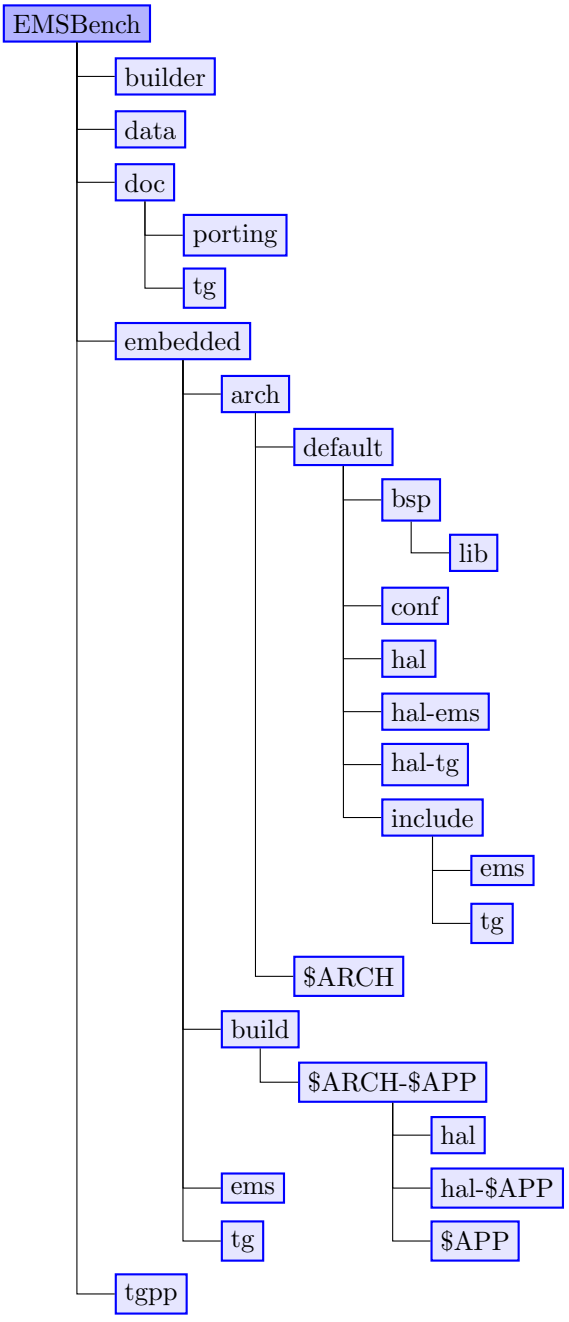

Figure 1: Directory structure of EMSBench

2 Build Environment

The whole build process of the EMSBench applications is managed by the `build.py` script. Concerning the embedded applications (`tg` and `ems`), a number of `make` configuration files are used. This section describes the configuration files and the variables defined therein.

2.1 Platform-Independant Configuration

The EMSBench `embedded` directory contains the following platform-independant `make` configuration files:

`conf/build.mk` manages the overall build process

`$APP/files.mk` lists the (platform-independant) source files of application `$APP`.

2.1.1 `build.mk`

The whole build process of an embedded application is controlled by the file `embedded/conf/build.mk`. This file contains the rules to build the target ELF binary as well as the relevant dependency and object files. Therefore, it requires a number of variables that must be provided by the platform-specific configuration files (see sect. 2.2):

The following variables must be defined in the actual `Makefile` that includes the `build.mk` file (this is automatically done by the Python scripts):

`BASE` Path to the `embedded` directory, relative to the including `Makefile`.

`ARCH` Platform architecture name.

`APP` Name of the application that shall be built (currently, either `ems` or `tg`).

`SUPP_SRC` Additional source code files that reside in the build directory (optional, needed for trace generator).

From the actual application, the following variables are imported:

`APP_SRC` lists the application's source files

Finally, a number of variables is required from the platform-specific configuration files:

`HAL_C_SRC` **and** `HAL_S_SRC` Generic HAL C and assembler sources.

`HAL_SUPP_S_SRC` Supplementary assembler sources. These files are compiled, but not included in the link process. They should be linked by platform-specific linker flags, e.g. `-msys-crt0='hal/crt0.o'` on the NIOS platform.

`HAP_C_SRC` **and** `HAP_S_SRC` Application-specific HAL C and assembler sources.

2.1.2 `$APP/files.mk`

The `files.mk` file for each application defines the following variables:

`THE_APP_SRC` lists all source files of the application. Source files must reside directly inside the application directory, the build process does not support subdirectories.

2.2 Platform-Specific Configuration Files

Inside each `arch/$ARCH` directory, a number of `make` configuration files provides platform-dependant build information:

`conf/toolchain.mk` defines the toolchain variables (`CC`, `LD`). If you need to add some more flags, also do it in this files.

`conf/upload.mk` provides a rule to upload the `$(TARGET).elf` to the hardware.

`hal/files.mk` lists all source files of the platform's generic HAL. Use `HAL_C_SRC` for C and `HAL_S_SRC` for assembler sources. Assembler files that are not directly linked must be referenced in `HAL_SUPP_S_SRC`.

`hal-$APP/files.mk` lists all source files of the application-specific HAL. Use `HAP_C_SRC` for C and `HAP_S_SRC` for assembler sources.

2.3 Building

Use the `build-ems.py` resp. `build-tg.py` scripts in the root directory.

2.4 The Application `Makefile`

The actual Makefiles for the `tg` and `ems` applications are generated by the Python build scripts. They define some of the variables mentioned above and the include the `build.mk` file that manges the build process.

3 Porting

To port EMSbench to a new platform `$NPF`, take the following steps:

1. Create copy of the `embedded/arch/default/` directory, or create the `embedded/arch/$NPF` structure from scratch.
2. Copy the include files from the `default` directory. Adjust appropriately, if you want to use macros instead of function calls in places.
3. Implement the remaining functions in the appropriate `hal*/` directories. Make sure to follow the callback functions in your interrupt handlers.
4. If your platform needs a board support package (BSP), place it inside the `bsp` directory. Also, make an approriate `Makefile` available.
5. Write the configuration files for `make` according to sect. 2.2.
6. If necessary, add further compilation parameters to the relevant `make` flags (e.g. `CFLAGS += ...`).
7. Add your platform to the `PLATFORMS` array in `/builder/data.py`. If your platform requires building of an additional BSP, set the second parameter to `True`. The build process will use the `Makefile` in the platform's `bsp/` subdirectory.