EmbeddedSystemsBuildScripts

Release v1.0

Embedded Systems Department University Duisburg-Essen

Aug 09, 2021

FOR USERS

A collection of Bazel build scripts adding support for avr-gcc and unit testing with the Unity framework.

CHAPTER

ONE

BAZEL SETUP

This article explains how to setup Bazel, in order to work properly with the EmbeddedSystemsBuildScripts.

1.1 Install Bazelisk

Bazelisk is a bazel wrapper, which provides an easy way to switch between different bazel versions, without uninstalling your local bazel installation. In order to build with a specific bazel version, you need to supply a `.bazelversion` file, where the desired version is specified, in your project root. For more information take a look at the Github repository

1.1.1 Linux

The following manual explains how to install bazelisk on a ubuntu host. This should be the same on any other Debian based systems. Some things may differ if you're using a different Linux distribution. In that case please look up your errors and add them to the troubleshooting section.

1. Step: Install Go

The installation slightly differs between Ubuntu versions. Please take a look here. The first paragraph on Ubuntu 19.04(LTS) should be fine.

2. Step: Install Bazelisk

This chapter explains how to get and install bazelisk. However, you are also able to fetch a suited binary from the Github releases.

- run `go get github.com/bazelbuild/bazelisk` in your command line
- add to your PATH variable: `export PATH=$PATH:$(go env GOPATH)/bin`
- you may also want to simlink `bazelisk` to `bazel`, but that's not really necessary

1.1.2 MacOS

- install the homebrew package manager
- run `brew install bazelisk`

CHAPTER

TWO

AVRTOOLCHAIN

The AvrToolchain repository is an external dependency that is generated automatically by a `repository_rule` implemented in `@EmbeddedSystemsBuildScripts//Toolchains/Avr:avr.bzl`. It provides `cc_toolchains` for compiling code with the avr-gcc compiler, for different mcus. Most of the time you will want to enable the `--compile_mode=optimization` flag that already contains gcc flags we found useful for reducing code size.

2.1 Instantiate the AvrToolchain Repository

To depend on the `EmbeddedSystemsBuildScripts` add this to your `WORKSPACE` file:

```
load("@bazel_tools//tools/build_defs/repo:http.bzl", "http_archive")

http_archive(
  name = "EmbeddedSystemsBuildScripts",
  strip_prefix = "EmbeddedSystemsBuildScripts-{version}",
  urls = ["https://github.com/es-ude/EmbeddedSystemsBuildScripts/archive/{version}.tar.gz
↪"]
)
```

replace `{version}` with the actual version you want to use. Or use:

```
http_archive(
  name = "EmbeddedSystemsBuildScripts",
  strip_prefix = "EmbeddedSystemsBuildScripts-master",
  urls = ["https://github.com/es-ude/EmbeddedSystemsBuildScripts/archive/master.tar.gz"]
)
```

to depend on the current master branch. Now you can call the repository rule, that will create the necessary avr toolchains and platforms. Add:

```
load("@EmbeddedSystemsBuildScripts//Toolchains/Avr:avr.bzl", "avr_toolchain")

avr_toolchain()
```

to the `WORKSPACE` file. The `http_archive` rule has to be called before loading the `create_avr()` function.

2.2 On Platforms and Constraints

Our code has to be deployable on a range of 8-bit AVR platforms as well as the host platforms (this is where your bazel instance runs). Bazel's platforms and constraints mechanics allow to make build decisions depend on different constraints. The user can then specify a set of specific constraints to apply to the current build process with the help of the `platform` rule.

Constraints are basically just typed enumerations and platforms specify a set of constraints. The type of a `constraint_value` is called `constraint_setting`. For every `platform` at most one `constraint_value` for each `constraint_setting` may be specified (ie. your platform may not have arm and x64_86 as cpu architecture).

The scripts provided by us already take different constraints into account. This allows us to write scripts that will produce correct results without knowing the exact platform you want to build for.

We already ship some platform definitions for platforms that we use internally. You can see a list of these definitions by running:

```
$ bazel query `(kind:platform, @AvrToolchain//platforms:*)
```

To compile for one of these platforms use e.g.:

```
$ bazel build //:myTarget --platforms @AvrToolchain//platforms:Motherboard
```

By default, we compile with the feature named `gnu99`, that adds `--std=gnu99` to the build command. However, if you want to build with avr-g++ `-std=gnu99` is an invalid flag and can be disabled by adding the build flag `--feature=-gnu99`.

2.2.1 How to define your own platforms

To define your own avr based platform you will need to specify at least the mcu. Run:

```
bazel query 'kind(constraint_value, @AvrToolchain//platforms/mcu:*)'
```

to retrieve a list of available mcus. Additionally there is the `@AvrToolchain//platforms:avr_common` platform that serves as a parent for all other avr based platforms. E.g. a new platform definition could look like this:

```
platform(
  name = "MyPlatform",
  constraint_values = [
    "@AvrToolchain//platforms/mcu:atmega16",
    "@AvrToolchain//platforms/cpu_frequency:8mhz",
  ],
  parents = ["@AvrToolchain//platforms:avr_common"],
)
```

To see a list of available constraint settings run:

```
$ bazel query 'kind(constraint_setting, @AvrToolchain//platforms/...)'
```

and to see a list of available values for the setting <my_setting> you can run:

```
$ bazel query 'attr(constraint_setting, <my_setting>, @AvrToolchain//...)'
```

CHAPTER

THREE

UNIT TESTING

3.1 Basic Setup

In order to make unit testing work, the `WORKSPACE` file must contain the external dependency `Unity`:

```
http_archive(
    name = "Unity",
    build_file = "@EmbeddedSystemsBuildScripts//:BUILD.Unity",
    strip_prefix = "Unity-master",
    urls = ["https://github.com/ThrowTheSwitch/Unity/archive/master.tar.gz"],
)
```

We would advise to use the `BazelCProjectCreator` for creating a project. This python script creates the complete project, including a unit test. However, if you want to include unit tests in your current project, we would advise you to create a folder called `test`. This folder should contain *.c files with unit tests and a `BUILD` file.

Content of a .c test file

```
#include "unity.h"

void test_shouldFail(void)
{
    TEST_FAIL();
}
```

Content of the `BUILD` file:

```
load("@EmbeddedSystemsBuildScripts//Unity:unity.bzl", "unity_test")
```

Each file that contains unit tests can be compiled and executed by using the `unity_test` macro, i.e.:

```
unity_test(
    cexception = False,
    file_name = "first_Test.c",
    deps = [
        "//:Library",
        "//My_Project:HdrOnlyLib",
    ]
)
```

The tests can be be run by executing `bazel test test:first_Test` from the project root in the command line. Alternatively, all available tests can be run with `bazel test test:all`.

3.2 CException

In the example unit test listed above, `cexception` is set to False. If you want to include CException as an external dependency in your project, you need to add the following to your `WORKSPACE` file:

```
http_archive(
    name = "CException",
    build_file = "@EmbeddedSystemsBuildScripts//:BUILD.CException",
    strip_prefix = "CException-master",
    urls = ["https://github.com/ThrowTheSwitch/CException/archive/master.tar.gz"],
)
```

Additionally, you may set the `cexception` attribute to True (default value is True).

3.3 Mocking

We currently make use of `CMock` for creating mocks. CMock can be included as an external dependency by adding the following to the `WORKSPACE` file:

```
http_archive(
    name = "CMock",
    build_file = "@EmbeddedSystemsBuildScripts//:BUILD.CMock",
    strip_prefix = "CMock-master",
    urls = ["https://github.com/ThrowTheSwitch/CMock/archive/   master.tar.gz"],
)
```

Mocks are created in the `BUILD` file of the test folder. In order to do that, load the macro `mock()`, by adding it to the load statement, i.e.:

```
mock(
    name = "mock_MyHeader",
    srcs = ["//MyProject:MyHeader.h"],
    deps = ["//MyProject:MyHeaderLibrary"],
)
```

In order to use the mock in a unit test, the mock has to be in the dependencies of the unit test at the first position, i.e.:

```
unity_test(
    cexception = False,
    file_name = "my_Test.c",
    deps = [
        "mock_MyHeader",
        "//MyProject:MyHeaderLibrary",
    ],
)
```