mbeddr C User Guide

This document focuses on the C programmer who wants to exploit the benefits of the extensions to C provided by mbeddr out of the box. We assume that you have some knowledge of regular C (such asK&R C, ANSI C or C99). We also assume that you realize some of the shortfalls of C and are "open" to the improvements in mbeddr C. The main point of mbeddr C is the ability to extend C with domain-specific concepts such as state machines, components, or whatever you deem useful in your domain. We have also removed some of the "dangerous" features of C that are often prohibited from use in real world projects.

This document does not discuss how to develop new languages or extend existing languages. We refer to the *Extension Guide* instead. It is available from **http://mbeddr.com**.

Contents

1 Tutorial

This tutorial assumes that you have successfully run through the Hello World tutorial in Section **??**. If you haven't, you can use one of the Wizards in the **Code** menu to create a **Minimal Test Case**. From this, the starting point for our tutorial looks as follows:

- We have a default type size configuration.
- We have a build configuration with one **Executable**.
- We have an **ImplementationModule** with a **main** function.

1.1 Function Pointers

1.1.1 The Basic Program

As the first example, we will add a configurable event handler using function pointers. We create a new module **FunctionPointers** using the context menu **New -> c.m.core.module -> ImplementationModule** on the current model.

Inside it, we will add a **struct** called **DataItem** that contains two members. You create the **struct** by just typing **struct** inside the module. You add the members by simply starting to type the **int8** types.

```
struct DataItem {
  int8 id;
};
```

We then create two functions that are able to process the **DataItem**s. Here is one function that does nothing (intentionally). You enter this function by starting out with the **DataItem** type, then typing the name and then using the **(** to actually create the function (the thing has been a global variable up to this point!):

```
DataItem process_doNothing(DataItem e) {
  return e;
}
```

Other functions with the same signature may process the data in some specific way; We can generalize those into a function type using a **typedef**. Note that entering the function type **()=>()** is in fact a little bit cumbersome. The alias for entering it is **funtype**:

```
typedef (DataItem)=>(DataItem) as DataProcessorType;
```

We can now create a global variable that holds an instance of this type and that acts as a global event dispatcher. We also create a new, empty **test case** that we will use for making sure the program actually works. In the test we assign a reference to **process_doNothing** to that event handler.

```
DataProcessorType processor;
exported test case testProcessing {
  processor = :process_doNothing;
}
```

We can now write the first simple test:

```
exported test case testProcessing {
  processor = :process_doNothing;
  DataItem i1;
  i1.id = 42;
  DataItem i2 = processor(i1);
  assert(0) i2.id == 42;
}
```

Let us complete this into a runnable system. In the **Main** module we change our **main** function to run our new test. Note how we import the **FunctionPointers** module; we call the test case, which is visible because it is **exported**:

```
module Main imports FunctionPointers {
  exported int32 main(int32 argc, string*[] argv) {
    return test testProcessing;
  }
}
```

Looking at the build configuration we see an error that complains that the binary is inconsistent, because the **FunctionPointers** module is not included. We can fix this with a quick fix. This results in the following binary:

```
executable MbeddrTutorial isTest: true {
  used external libraries
  used mbeddr libraries
  included modules
```

```
    Main (mbeddr.tutorial.main.m1)
    FunctionPointers (mbeddr.tutorial.main.m1)
}
```

1.1.2 Building and Running

We can now build the system using **Ctrl-F9** or by selecting **Rebuild** from the context menu of the solution in the logical view on the left. If you have installed **gcc** correctly the binary should actually be compiled automatically. Here is the info message you should get in the MPS messages view:

```
make finished successfully for mbeddr.tutorial.main/mbeddr.tutorial.main.m1
```

Let us run this program from the command line. To get to the respective location in the file system, select the solution in the logical view, open the properties and copy the **Solution File** location. In my case it is:

```
/Users/markusvoelter/Documents/mbeddr/mbeddr.core/code/applications/tutorial/solutions/
    mbeddr.tutorial.main/mbeddr.tutorial.main.msd
```

We can open a console and **cd** to this directory, removing the last segment **mbeddr.tutorial.main.msd** from the path. In that directory we can **cd** to **source_gen**, and from there we can navigate down to the directory for the model: **cd mbeddr/tutorial/m1**

Using **ls** there, we can see the following:

```
$ ls -ll
total 104
-rw-r--r--  1 markusvoelter  staff   1189 Oct 30 21:11 FunctionPointers.c
-rw-r--r--  1 markusvoelter  staff    159 Oct 30 21:11 FunctionPointers.h
-rwxr-xr-x  1 markusvoelter  staff   9028 Oct 30 21:11 Main
-rw-r--r--  1 markusvoelter  staff    338 Oct 30 21:11 Main.c
-rw-r--r--  1 markusvoelter  staff    162 Oct 30 21:11 Main.h
-rw-r--r--  1 markusvoelter  staff    433 Oct 30 21:11 Makefile
drwxr-xr-x  4 markusvoelter  staff    136 Oct 30 21:11 bin
-rw-r--r--  1 markusvoelter  staff    943 Oct 30 21:11 module_dependencies.gv
-rw-r--r--  1 markusvoelter  staff  14069 Oct 30 21:11 trace.info
```

Importantly, we see a **Makefile**, so we can call **make**. This will build the binary:

```
$ make
rm -rf ./bin
mkdir -p ./bin
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o Main bin/Main.o bin/FunctionPointers.o
```

This results in an executable **MbeddrTutorial**. We can run it by calling **./MbeddrTutorial** or by calling **make test**. The output should look as follows:

```
$ make test
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
```

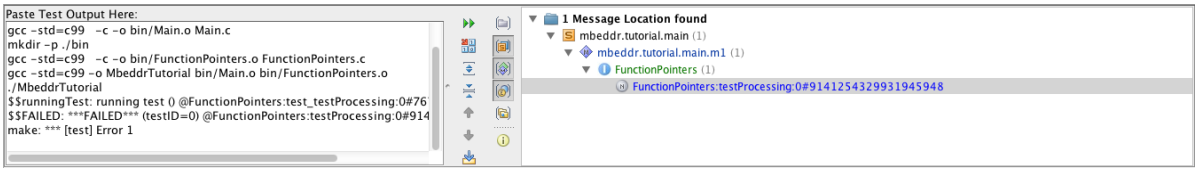
So the test succeeded, everything seems fine. Let us try to introduce an error by somehow breaking the assertion.

```
assert(0) i2.id == 0;
```

After regenerating the code (**Ctrl-F9** or **Rebuild**) we can call **make test** again and we get:

```
$ make test
mkdir -p ./bin
gcc -std=c99   -c -o bin/Main.o Main.c
mkdir -p ./bin
gcc -std=c99   -c -o bin/FunctionPointers.o FunctionPointers.c
gcc -std=c99 -o MbeddrTutorial bin/Main.o bin/FunctionPointers.o
./MbeddrTutorial
$$runningTest: running test () @FunctionPointers:test_testProcessing:0#767515563077315487
$$FAILED: ***FAILED*** (testID=0) @FunctionPointers:testProcessing:0#9141254329931945948
make: *** [test] Error 1
```

As you can see the test fails. It says that in the **FunctionPointers** module, **testProcessing** test case, assert number **0**. You can either navigate to the offending **assert** manually. Alternatively you can copy the console output into the clipboard and then paste it into the text box into the window that opens from the **Analyze -> mbeddr Analyze Error Output** menu:



Double clicking on the respective line opens the editor on the offending **assert** statement.

1.2 Debugging

Let us go back to the definition of **DataItem** and make it more useful. Our application is supposed to work with tracking data (captured from a bike computer or a flight logger). We change the **struct** as follows (note that renaming the thing is trivial — you just change the name!):

```
struct Trackpoint {
  int8 id;        // sequence ID of the track point
  int8 timestamp; // timestamp as taken from GPS time
  int8 x;         // longtitude, simplified as a number
  int8 y;         // latitude, simplified as a number
  int8 alt;       // altitude as of the GPS
  int8 speed;     // current speed, if available
};
```

We also create a second processing function that nulls the **alt** member (as a means of getting the "ground track" as projected on a map):

```
Trackpoint process_nullifyAlt(Trackpoint e) {
  e.alt = 0;
  return e;
}
```

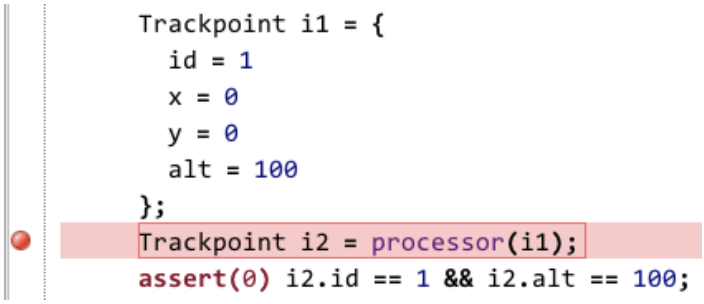
We can now enhance our test case the following way. Notice how this nicely tests the "redirection" of function pointers; we invoke the same **processor**, but since it has been assigned another function reference, it now does something different.

```
exported test case testProcessing {
  Trackpoint i1 = {
    id = 1
    x = 0
    y = 0
    alt = 100
  };

  processor = :process_doNothing;
  Trackpoint i2 = processor(i1);
  assert(0) i2.id == 1 && i2.alt == 100;

  processor = :process_nullifyAlt;
  Trackpoint i3 = processor(i1);
  assert(1) i3.id == 1 && i3.alt == 0;
}
```

This is an interesting example for debugging. We start by setting a breakpoint on the **Trackpoint i2 = processor(i1);** line. You can set a breakpoint by clicking into the gutter.



1.3 Physical Units

Bibliography