



mbeddr.core C User's Guide

Marcel Matzat¹, Bernhard Merkle², and Markus Voelter³

¹ itemis AG

² Sick AG

³ independent/itemis



Abstract. This document describes how to use mbeddr.core for C programming. It starts out with installation instructions for MPS, gcc/make and mbeddr.core. It then walks through a hello world example. The final section systematically discusses the differences or extensions of mbeddr C compared to regular C.

1 Installation

1.1 MPS

The mbeddr system is based on JetBrains MPS, an open source language workbench available from <http://www.jetbrains.com/mps/>. MPS is available for Windows, Mac and Linux, and we currently use version 2.0.x. Please make sure you install MPS into a folder that does not use blanks in any of its directory or file names (not even in the MPS 2.0 folder). This will simplify some of the command line work you may want to do.

After installing MPS using the platform-specific installer, please open the bin folder and edit the `mps.voptions` or `mps.exe.voptions` file (depending on your platform). To make MPS run smoothly, the `MaxPermSize` setting should be increased to `384m` or `512m`. It should look like this after the change:

```
-client
-Xss1024k
-ea
-Xmx1200m
-XX:MaxPermSize=512m
-XX:+HeapDumpOnOutOfMemoryError
-Dfile.encoding=UTF-8
```

On some 32bit Windows XP systems we had to reduce the `-Xmx1200m` setting to `768m` to get it to work.

1.2 GCC and make

The mbeddr toolkit relies on `gcc` and `make` for compilation (unless you use a different, target-specific build process). On Mac you should install XCode to get gcc,

make and the associated tools. On Linux, these tools should be installed by default. On Windows we recommend installing cygwin (<http://www.cygwin.com/>), a Unix-like environment for Windows. When selecting the packages to be installed, make sure `gcc` and `make` are included. Please also add `gcc` and `make` to your `PATH` variable in the mps startup file e.g. on Windows `mps.bat` should include the following at the very top of the file.

```
::rem mbeddr depends on Cygwin: gcc, make etc
set PATH=C:\ide\Cygwin\bin;%PATH%
```

1.3 mbeddr

You can get the mbeddr system either as a packaged download or via a git repository (currently only the download is available; it contains the sources as well). Get it from <http://mbeddr.wordpress.com/getit/>.

This document describes the mbeddr.core package. Save the ZIP file to a folder on your hard disk and unzip it. Once again, please make sure the path to the unzipped folder contains no blanks!

2 Important keyboard shortcuts in MPS and mbeddr

2.1 MPS in general

MPS is a projectional editor. It does not parse text and build an AST. Instead the AST is created directly by user editing actions, and what you see in terms of text (or other notations) is a projection. This has many advantages, but it also means that some of the well-known editing gestures we know from normal text editing don't work. So in this section we explain some keyboard shortcuts that are essential to work with MPS.

TODO(*Create Screencast and link to it.*)



Entering Things In MPS you can only enter those things that are available from the code completion menu. Using aliases and other "tricks", MPS manages to make this feel *almost* like text editing. Here are some hints though:

- As you start typing, the text you're entering remains red, with a light red background. This means the string you've entered has not yet bound.
- Entered text will bind if there is only one thing left in the code completion menu that starts with the substring you've already entered. An instance of the selected concept shows up and the red goes away.
- As long as text is still red, you can press **Ctrl-Space** to explicitly open the code completion menu, and you can select from those concepts that start with the substring you have entered so far.

- If you want to go back and enter something different from what the entered text already preselects, press **Ctrl-Space** again. This will show the whole code completion menu.
- Finally, if you're trying to enter something that does not bind at all because your entered prefix does not match any concept, there is not point in continuing to type. ~~Either you're trying to enter the wrong thing, or the language is broken.~~ There is no way of typing program text "all in red" — we have seen this, and it does not work.

Navigation Navigation in the source works as usual using the cursor keys or the mouse. References can be followed ("go to definition") either by **Ctrl-Click** or by using **Ctrl-B**.

Selection Selection is different. **Ctrl-Up/Down** can be used to select along the tree. For example consider a local variable declaration `int x = 2 + 3 * 4;` with the cursor at the 3. If you now press **Ctrl-Up**, the `3 * 4` will be selected because the `*` is the parent of the 3. Pressing **Ctrl-Up** again selects `2 + 3 * 4`, and the next **Ctrl-Up** selects the whole local variable declaration.

You can also select with **Shift-Up/Down**. This selects siblings in a list. For example, consider a statement list as in a function body ...

```
void aFunction() {
    int x;
    int y;
    int z;
}
```

... and imagine the cursor in the `x`. You can press **Ctrl-Up** once to select the whole `int x;` and then you can use **Shift-Down** to select the `y` and `z` siblings.

Intentions Some editing functionalities are not available via "regular typing", but have to be performed via a quick fix. Quick fixes called intentions. The intentions menu can be shown by pressing **Alt-Enter**. For example, module contents in mbeddr can only be set to be **exported** by selecting *export* from the intentions menu. Explore the contents of the intentions menu from time to time to see what's possible. Surprises may lurk there :-)

Refactorings For many language constructs, refactorings are provided. Refactorings are more important in MPS than in normal text editors, because some (actually a few) editing operations are hard to do manually. Please explore the refactorings context menu, and take note when we explain refactorings in the user's guide. Unlike intentions, which cannot have a specific keyboard shortcut assigned, refactorings can, and we make use of this feature heavily. The next section introduces some of these.

2.2 mbeddr specific shortcuts

Documentation Many program elements can be documented. A documentation is basically free text associated with a program element. We distinguish documentation from commenting out code (explained below), and any program element whose concept implements the `IDocumentable` interface can have a documentation. Example of concepts that do include statements, functions, global variables or structs. A documentation is shown as a grey comment above the commented element (the documentation is really attached to the element, and not just written into a line above it — a subtle but important difference!)

```
// Here is some documentation for the function
int8_t main(string[ ] args, int8_t argc) {
    // ... and here is some doc for the report statement
    report(0) HelloWorldMessages.hello() on/if;
    return 0;
} main (function)
```



Documentation can be added using the *Add Documentation* intention, or by using **Ctrl-Alt-D** on the respective element.

Note that comments are only shown if they are turned on. You can use the context menu on any program element and select *ToggleDocs* to enable/disable display of comments. As soon as a comment is added, comment display is automatically turned on.

Commenting out Code Code that is commented out retains its syntax highlighting, but is shaded with a grey background.

```
// // Here is some documentation for the function
int8_t main(string[ ] args, int8_t argc) {
    // ... and here is some doc for the report statement
    report(0) HelloWorldMessages.hello() on/if;
    return 0;
}
```

Code can be commented out by pressing **Ctrl-Alt-C** (this is technically a refactoring, so this feature is also available from the refactorings context menu). This also works for lists of elements. Commented out code can be commented back in by pressing **Ctrl-Alt-C** on the comment itself (the `//`) or the commented element.

Commenting out code is a bit different than in regular, textual systems because code that is commented out is still "live": it is still stored as a tree, code completion still works in it, it may still be shown in *FindReferences*, and refactorings may affect the code. We are not sure if this is a desirable feature and we are looking for your feedback. Of course, the code is not executed. All commented program elements are removed in the first transformation phase.



Note: The current implementation of comments is still a little bit of a hack since we are waiting for some direct support by MPS. For example, errors should not be shown in commented code, and we are sure other quirks will arise as we continue using mbeddr.

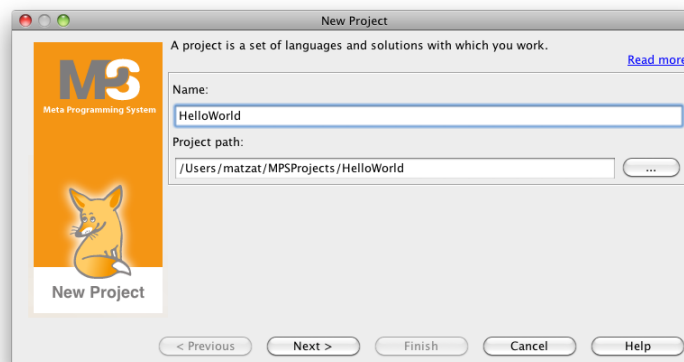
Not all program elements can be commented out (since special support by the language is necessary to make something commentable), only concepts that implement `ICommentable` can be commented. At this time, this is all statements and module contents.

3 Hello World Example


For this tutorial we assume that you know how to use the C programming language. We also assume that you have installed MPS, gcc/make and the mbeddr.core distribution. This has been discussed in the previous section.

3.1 Create new project

Start up MPS and create a new project. Call the project `HelloWorld` and store it in a directory without blanks in the path. Let the wizard create a solution, but no language.




We now have to make the project aware of the *mbeddr.core* languages installed via the distribution. Go to the *File* → *Settings* and select the *GlobalLibraries* in the IDE settings. Create a library called `mbeddr.core` that points to the root directory of the unzipped mbeddr installation.



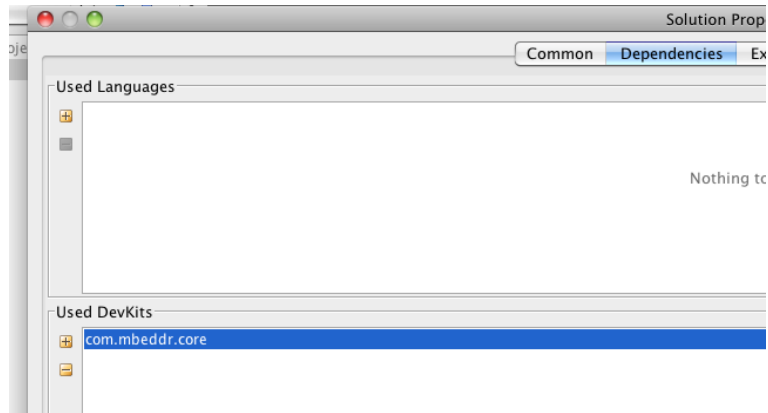
figures/SettingsGlobalLibraries.png

Notice that these are global settings and only have to be performed once before your first application project.

3.2 Project Structure and Settings

 An MPS project is a collection of languages and solutions. A *language* **defines** new language, e.g. an extension to C. A *solution* is an application project that **uses** existing languages. Solutions contain any number of models. Physically, models are XML files that store MPS programs. They are the relevant version control unit, and the fundamental unit of configuration. So, in the solution, create a new model with the name `main`: select *New* → *Model* from the solution's context menu.

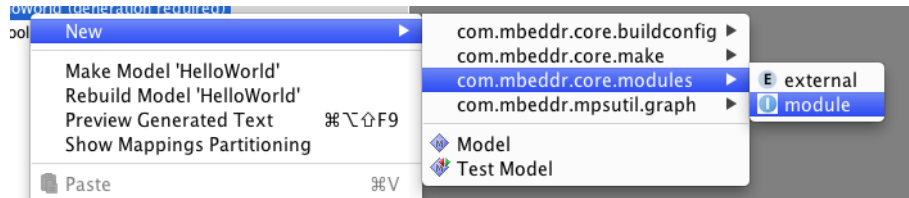
A model has to be configured with the languages that should be used to write the program in the model. In our case we need all the `mbeddr.core` languages. We have provided a *devkit* for these languages. A devkit is essentially a set of languages, used to simplify the import settings. As you create the model, the model properties dialog should open automatically. In the *Used Devkits* section, select the + button and add the `com.mbeddr.core` devkit.



This concludes the configuration and setup of your project. You can now start writing C code.

3.3 Create an empty Module

The first step is now to create an empty Module. The mbeddr.core C language does not use the artificial separation between `.h` and `.c` files. Instead mbeddr C uses a module concept. We then use code generation to create a corresponding `.h` and `.c` file but we don't expose this to the user. The top level concept in mbeddr C programs are *modules*. Modules act as namespaces and as the unit of visibility. A module can import another module. The importing module can then access *exported* contents of imported modules. So to get started, we create a new **implementation module** using the context menu as shown in the following picture:



Note: This operation, as well as almost all others, can be performed with the keyboard as well. Take a look at *File* → *Settings* → *Keymap* to find out or change keyboard mappings.

As a result, you will get an empty implementation module. As of yet it has no a name (name is red and underlined) and a placeholder `<...>` where top level C constructs such as functions, structs, or enums can be added.



As the next step, specify a name for the implementation module, e.g. `HelloWorld`.

```
module HelloWorld from HelloWorld.main imports nothing {  
    << ... >>  
}
```

The module name is still underlined in red because of a missing `TypeSizeConfiguration`. The `TypeSizeConfiguration` specifies the sizes of the primitive types (such as `int` or `long`) for the particular target platform. `mbeddr C` provides a default type size configuration, which can be added to a module with the intention *Create default type size configuration* (Fig. 2). For more details see chapter 4.5.

3.4 Writing the Program

Within the module you can now add module contents. Functions, among other things, are module contents. You can enter a main function the following ways:

- you can create a new function instance by typing "function" at the placeholder in the module, and then specify the name and arguments.
- you can also simply start typing the type of the function, "int" in this case, and then entering the name
- specifically for the main function, you can also just type "main" (it will set up the correct signature automatically)

At this point, we are ready to implement the Hello World program. Our aim is to simply output a log message and return 0. To add a return value, move into the function and type `return 0`.

```
module HelloWorld from HelloWorld.main imports nothing {  
    int32_t main() {  
        //print "Hello, World!";  
        return 0;  
    }  
}
```

To print the message we could use `printf` or some other `stdio` function. However in embedded systems there is often no `printf` or display available, so we use a special language extension for logging. It will be translated in a suitable way, depending on the available facilities for the target platform. Also, specific

log messages can be deactivated in which case they are statically removed from the program. So below our main function we create a new **message list** (just type **message** followed by **return**) and give it the name **log**.

Within the message list, hit **return** or type **message** to create a new message. Change the type from **ERROR** to **INFO** with the help of auto completion or just type **INFO** instead **ERROR**. Specify the name **hello**. Add a message property with type **string** and call it **world**. An empty message property will be created when you hit **return** within the brackets. Give the message text property the text **Hello**.



```
message list log {  
  INFO hello(string world) active: Hello  
}
```

Now you are ready to use the message list and its messages from your main function. Insert a **report()** statement, specify the message list **log** and select the message **hello**. Pass the string "World" as parameter.

```
module HelloWorld from HelloWorld.main imports nothing {  
  
  int32_t main(int8_t argc, string[ ] args) {  
    report(0) log.hello("World") on/if;  
    return 0;  
  } main (function)  
  
  message list log {  
    INFO hello(string world) active: Hello  
  }  
}
```

3.5 Build Configuration

We have to create one additional element: the **BuildConfiguration** specifies which modules should be compiled into an executable and will result in a **make** file.



In the main model (outside our implementation module), create a new instance of **BuildConfiguration**. It will contain some useful defaults, e.g. the **gcc** compiler and its options. At the placeholder, create a new **Program** and call it **HelloWorld**. In the program's body, add a reference to the **HelloWorld** implementation module we've created before. The code should look like this:

```
Build Configuration  
  env desktop  
  compiler: gcc  
  compiler options: -std=c99
```

```
program HelloWorld isTest: false {  
  HelloWorld  
}
```

Note: Compilation and build is highly specific to the target platform. We currently support the default gcc/make based approach, but we expect to add different approaches over time. It is likely that you will have to add your own approach for your platform.

3.6 Building and Executing the program

Press **Ctrl-F9** (or **Cmd-F9** on the Mac) to rebuild the program. In the `HelloWorld/solutions/HelloWorld/source_gen/HelloWorld/main` directory you should now have at least the following files (there may be others, but those are not important now):

```
Makefile  
HelloWorld.c  
HelloWorld.h
```

To compile the files, open a command prompt (must be a cygwin prompt on Windows!) in this directory and type **make**. The output should look something like this:

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

This rebuilds the executable file `HelloWorld.exe` or `HelloWorld`. Running it shows this:

```
\$ ./HelloWorld.exe  
hello: Hello @HelloWorld:main:0  
world = World
```

Note how the log statement outputs the location of the log statement in the program (report statement number 0 in function `main` in module `HelloWorld`; take a look back at the source code: the index of the statement (here: 0) is also output in the program source). This source location can be output because the `report` statement is translated statically, and during the translation, the transformation of course knows its location in the code.

3.7 Command-Line Build

TODO(*Here we had the ant-based translation example before. I liked it. Can we add it back?*)

This concludes our hello world example. In the next section we will examine important differences between mbeddr C and regular C.

4 Differences to regular C

This section describes the differences between *mbeddr C* and regular C 99. All examples shown in this chapter can be found in the *HelloWorld* project that is available for download together with the *mbeddr.core* distribution.

4.1 Preprocessor

mbeddr C does not support the preprocessor. Instead we provide first class concepts for the various use cases of the C preprocessor. This avoids some of the chaos that can be created by misusing the preprocessor and provides much better analyzability. We will provide examples later. The first example is the module system explained in the next section.

4.2 Modules

While we *generate* header files, we don't *expose* them to the programmer in MPS. Instead, we have defined modules as the top-level concept. Modules also act as a kind of namespace. Module contents can be exported, in which case, if that module is imported by another module, the exported contents can be used by the importing module.

We distinguish between *implementation modules* which contain actual implementation code, and *external modules* which act as proxies for pre-existing header files that we want to be able to use from within mbeddr C programs.

Implementation modules The following example shows a implementation module (`ImplementationModule`) with an exported function. You can toggle the *exported* flag with the intention *export* or *export: remove*. The second module (`ModuleUsingTheExportedFunction`) imports the `ImplementationModule` with the `imports` keyword in the module header. An importing module can access all exported contents defined in imported modules.

```
module ImplementationModule from HelloWorld.ImplementationModules
  imports nothing {

    exported int32_t add(int32_t i, int32_t j) {
      return i + j;
    } add (function)
  }
```

```

module ModuleUsingTheExportedFunction from HelloWorld.ImplementationModules
  imports ImplementationModule {

  int32_t main(int8_t argc, string[ ] args) {
    int32_t result = ImplementationModule::add(10, 15);
    return 0;
  } main (function)
}

```

External modules mbeddr C code has to be able to access existing code. We go about this the following way:



- We identify existing external header files and the corresponding object or library files
- We create an *external module* to represent those; the external module specifies the `.h` file and the object/library files it represents
- In the external module we add the those contents of the existing `.h` file we want to make visible to the mbeddr C program
- We can now import the external module into any implementation module from which we want to be able to call into the external code
- The generator generates the necessary `#include` statements, and the corresponding build configuration.

Note: In the future we will provide a mechanism to import existing header files into an external module. As of now, the relevant signatures etc. have to be typed in manually.

The following code shows the external module `STDIO`. In the `resources` section, you have to provide the path to the resources associated with this external module. You can add `headers` and `linkables` here. Since gcc knows what to link when `<stdio.h>` is included, we don't have to specify a linkable here.

```

external module STDIO resources header : <stdio.h>
// external module contents are exported by default
{
  void printf(string format, ...);
}

```

To call methods from external modules, you have to import the external module into your implementation module: `imports STDIO`. You can add now call the `printf` function defined in the external module.

```

module MainApp from HelloWorld.ExternalModules imports STDIO {

  int32_t main() {

```

```

    STDIO::printf("Dies ist ein stdout.printf Text: %s\n", "Noch einer");
    return 0;
} main (function)
}

```

4.3 Build configuration

The **BuildConfiguration** specifies which modules should be compiled into an executable. It will be generated into a **make** file that performs the compilation.

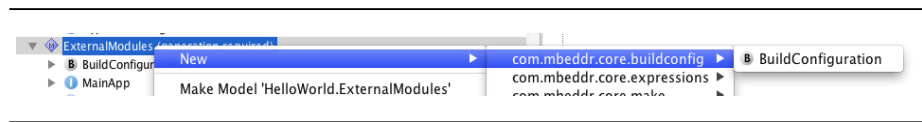


Fig. 1. Add a BuildConfiguration Model

The next example shows the default build configuration when you add a new **BuildConfiguration** with the context menu (see Fig. 1).

```

Build Configuration
  env desktop
  compiler: gcc
  compiler options: -std=c99

<< ... >>

```

The build configuration specifies how the make process is executed. The **env** parameter lets you select the target environment. This determines how certain language constructs are translated to C. At the moment you can only select desktop. More to follow. **compiler** is the command that invokes your C compiler. By default it is gcc. **compiler options** let you specify additional command line options for the compiler.

The main part of the build configuration supports the definition of binaries. Binaries are either executables or libraries.

Executables An executable binds together a set of modules references in the program, and compiles it into an executable. Exactly one module in a each executable has to have a main function.

The build configuration results in a **make** file which is automatically run as part of the MPS build, resulting in the corresponding executable binaries. The generated code, the make file and the executables can be found in the **source_gen** folder of the respective solution.

Below is the build configuration of the `ExternalModules` example. It defines one executable `Application`. It consist of the modules `MainApp` and `STDIO`.

```
Build Configuration
  env desktop
  compiler: gcc
  compiler options: -std=c99

executable Application isTest: false {
  MainApp
  STDIO
}
```

Libraries **TODO()**

4.4 Unit tests

Unit Tests are supported as first class citizens by mbeddr C. A `Test Case` implements `IModuleContent`, so it can be used in implementation modules alongside with functions, structs or global variables. To assert the corecctness of a result you have to use the `assert` statement followed by an boolean expression (note that `assert` really can just be used inside test cases). A `fail` statement is also available.

```
module AddTest from HelloWorld.UnitTests imports nothing {

  exported test case testAddInt {
    assert(0) 1 + 2 == 3;
    assert(1) -1 + 1 == 1;
  } testAddInt(test case)

  exported test case testAddFloat {
    float f1 = 5.0;
    float f2 = 10.5;
    assert(0) f1 + f2 == 15.5;
  } testAddFloat(test case)
}
```

The next code shows a main function that executes the test cases imported from the `AddTest` module. The `test` expression supports invocations of test cases; it also evaluates to the number of failed assertions. By returning this value from `main`, we get an exit code `!= 0` in the case a test failed.

```
module TestSuite from HelloWorld.UnitTests imports AddTest {
  int32_t main() {
    return test testAddInt, testAddFloat;
  } main (function)
}
```

In the build configuration, the `isTest: true` flag can be set to true; this adds a `test` target to the make file, so you can call `make test` on the command line in the source folder and run the tests.

The example above contains a failing assertion `assert(1) -1 + 1 == 1;`. Below is the console output after running `make test` in the generated source folder for the solution:

```
runningTest: running test @AddTest:test_testAddInt:0
FAILED: ***FAILED*** @AddTest:test_testAddInt:2
    testID = 1
runningTest: running test @AddTest:test_testAddFloat:0
make: *** [test] Error 1
```

If you change the assertion to `assert(1) -1 + 1 == 0;` you will get the following errors):

```
runningTest: running test @AddTest:test_testAddInt:0
runningTest: running test @AddTest:test_testAddFloat:0
```

4.5 Primitive Numeric Datatypes

The standard C data types (`int`, `long`, etc.) have different sizes on different platforms. This makes them non-portable. C99 provides another set of primitive data types with clearly defined sizes (`int8_t`, `int16_t`). In mbeddr C you *have* to use the C99 types, resulting in more portable programs. To be able to work with existing header files, the system has to know how the C99 types relate to the standard primitive types. This is the purpose of the `TypeSizeConfiguration`. It establishes a mapping between the C99 types and the standard primitive types.

The `TypeSizeConfiguration` mentioned above can be added with the *Create default type size configuration* (Fig. 2), or by creating one through the *New* menu on models. Every model has to contain exactly one type size configuration. To create the default, you can use an intention on the `TypeSizeConfiguration` itself.

Integral Types The following integral types are not allowed in implementation modules, and can only be used in external modules for compatibility: `char`, `short`, `int`, `long`, `long long`, as well as their unsigned counterparts. The following list shows the default mapping of the C99 types:

- `int8_t` → `char`
- `int16_t` → `short`

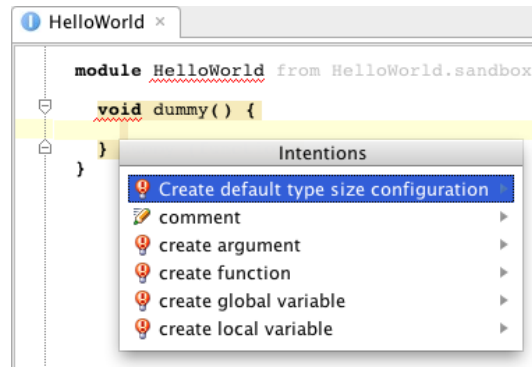


Fig. 2. Create default `TypeSizeConfiguration`

- `int32_t` → `int`
- `int64_t` → `long long`
- `uint8_t` → `unsigned char`
- `uint16_t` → `unsigned short`
- `uint32_t` → `unsigned int`
- `uint64_t` → `unsigned long long`

Floating Point Types **TODO()**

4.6 Booleans

We have introduced a specific `boolean` datatype, including the `true` and `false` literals. Integers cannot be used interchangeably with Boolean values. We do provide a (red, ugly) cast operator between integers and booleans for reasons of interop with legacy code. The following example shows the usage of the Boolean data type.

```
module BooleanDatatype from HelloWorld.BooleanDatatype imports nothing {  
  exported test case booleanTest {  
    boolean b = false;  
    assert(0) b == false;  
    if ( !b ) { b = true; } if  
    assert(1) b == true;  
    assert(2) int2bool<1> == true;  
  } booleanTest(test case)  
}
```

4.7 Literals

mbeddr C supports special literals for hex, octal and binary numbers. The type of the literal is the smallest possible signed integer type (`int8_t`, ..., `int64_t`) that can represent the number.

```
module LiteralsApp from HelloWorld.Literals imports nothing {

  exported test case testLiterals {
    int32_t intFromHex = hex<aff12>;
    assert(0) intFromHex == 720658;

    int32_t intFromOct = oct<334477>;
    assert(1) intFromOct == 112959;

    int32_t intFromBin = bin<100110011>;
    assert(2) intFromBin == 307;
  } testLiterals(test case)
}
```

4.8 Pointers

C supports two styles of specifying pointer types: `int *pointer2int` and `int* pointer2int`. In mbeddr C, only the latter is supported: pointer-ness is a characteristic of a type, not of a variable.

Pointer Arithmetics For pointer arithmetics you have to use an explicit type conversion `pointer2int` and `int2pointer`. For more details, look at the following example. You also see the usage of pointer dereference (`*xp`) and assigning an address with `&`.

```
module BasicPointer from HelloWorld.Pointer imports stdlib {

  exported test case testBasicPointer {

    int32_t x = 10;
    int32_t* xp = &x;
    assert(0) *xp == 10;

    int32_t[ ] anArray = {4, 5};
    int32_t* ap = anArray;
    assert(1) *ap == 4;

    // pointer arithmetic
    ap = int2pointer<pointer2int<ap> + 1>;
    assert(2) *ap == 5;

  } testBasicPointer(test case)
  ...
}
```

Memory allocation works the same way as in regular C except that you need an external module to call functions such as `malloc` from `stdlib`. The next

example shows how to do this. Note that `size_t` is a primitive type, built into mbeddr. It's size is also defined in a `TypeSizeConfiguration`.

```
external module stdlib resources header : <stdlib.h>
{
  void* malloc(size_t size);
  void free(void* pointer);
}
```

You have to include the external module `stdlib` in your implementation module with `imports stdlib`. You can then call `malloc` or `free`:

```
module BasicPointer from HelloWorld.Pointer imports stdlib {
  ...
  exported test case mallocTest {
    int8_t* mem = ((int8_t*) stdlib::malloc(sizeof int8_t));
    *mem = 10;
    assert(0) *mem == 10;
    stdlib::free(mem);
  } mallocTest(test case)
}
```

Function Pointers In regular C, you define a function pointer type like this: `int (*pt2Function) (int, int)`. The first part is the return type, followed by the name and a comma separated argument type list. The pointer asterisk is added before the name. This is a rather ugly notation; we've cleaned it up in mbeddr C.

In mbeddr, we have introduced the notion of function types and function references. These are syntactically different from pointers (of course they are mapped to function pointers in the generated C code). We have also introduced lambdas (i.e. closures without their own state).

For function types you first define the argument list and then the return type, separated by `=>` (a little bit like Scala). Here is an example: `(int32_t, int32_t)=>(int32_t)` You can enter a function type by using the `funtype` alias (see Fig. 3). Function types are types, so they can be used in function signatures, local variables or `typedefs`, just like any other type (see example *HelloWorld.Pointer.FunctionPointerAsTypes*).

Values of type funtype are either references to functions or lambdas. In regular C, you have use the address operator to obtain a function pointer (`&function`). In mbeddr C, you use the `:` operator (as in `:someFunction`) to distinguish function references from regular pointer stuff. Of course the type and values have to be compatible; for function types this means that the signature must be the same. The following example shows the use of function references:

```

int32_t main() {
    fun
    return 0;
} main (function)

```

Fig. 3. Add a function pointer with code completion

```

module FunctionPointer from HelloWorld.Pointer imports nothing {

  int32_t add(int32_t a, int32_t b) {
    return a + b;
  } add (function)

  int32_t minus(int32_t a, int32_t b) {
    return a - b;
  } minus (function)

  exported test case testFunctionPointer {
    // function pointer signature
    (int32_t, int32_t)=>(int32_t) pt2Function;

    // assign "add"
    pt2Function = :add;
    assert(0) pt2Function(20, 10) == 30;

    // assign "minus"
    pt2Function = :minus;
    assert(1) pt2Function(20, 10) == 10;
  } testFunctionPointer(test case)
}

```

Function types are types, mentioned already above. This behavior is illustrated in the next example. The typedef `typedef (int3_t, int32_t)=>(int32_t) as ftype;` defines a new function type. The type `ftype` is the first parameter in the `doOperation` function. You can easily call the function `doOperation(:add, 20, 10)` and put any suitable function reference as the first parameter.

```

module FunctionPointerAsTypes from HelloWorld.Pointer imports nothing {

  typedef (int32_t, int32_t)=>(int32_t) as ftype;

  int32_t add(int32_t a, int32_t b) {
    return a + b;
  } add (function)

  exported test case testFunctionPointer {
    // call "add"
    assert(0) doOperation(:add, 20, 10) == 30;
  } testFunctionPointer(test case)

  int32_t doOperation(ftype operation, int32_t firstOp, int32_t secondOp) {
    return operation(firstOp, secondOp);
  } doOperation (function)
}

```

Lambdas are also supported. The syntax for a lambda is [arg1, arg2, ...|an-expression-using-args] following is an example:

```
module Lambdas from HelloWorld.Pointer imports nothing {

  typedef (int32_t, int32_t)=>(int32_t) as ftype;

  exported test case testFunctionPointer {
    assert(0) doOperation([a, b|a + b;], 20, 10) == 30;
  } testFunctionPointer(test case)

  int32_t doOperation(ftype operation, int32_t firstOp, int32_t secondOp) {
    return operation(firstOp, secondOp);
  } doOperation (function)
}
```

4.9 Enumerations

The mbeddr C language also provides enumeration support, comparable to to C99. There is one difference compared to regular C99. In mbeddr C an enumeration is not an integer type. This means, you can't do any arithmetic operations with enumerations.

Note: We may add a way to cast enums to ints later if it turns out that "enum arithmetics" are necessary

```
module EnumerationApp from HelloWorld.Enumerations imports nothing {

  enum SEASON { SPRING; SUMMER; AUTUMN; WINTER; }

  exported test case testEnumeration {
    SEASON season = SPRING;
    assert(0) season != WINTER;
    season = WINTER;
    assert(1) season == WINTER;
  } testEnumeration(test case)
}
```

4.10 Goto

We have no goto.

Note: This may change :-)

4.11 Switch statement

In the **switch** statement, we don't use the annoying fall through semantics. Only one **case** within the **switch** will ever be executed, since we automatically generate a **break** statement into the generated C code. You can also add an **default** statement which will be executed if no other case match.

The next example shows a **switch** statement with integers and enumeration as the switched expression.

```
module SwitchStatement from HelloWorld.SwitchStatement imports nothing {

  var int32_t globalState;

  enum DAY { MONDAY; THUESDAY; WEDNESDAY; }

  exported test case testSwitchCase {
    globalState = -1;

    // Switch with int
    callSwitich(0);
    assert(0) globalState == 20;

    callSwitich(1);
    assert(1) globalState == 0;

    callSwitich(2);
    assert(2) globalState == 10;

    // Switch with day
    callSwitchWithEnumeration(MONDAY);
    assert(3) globalState == 1;

    callSwitchWithEnumeration(WEDNESDAY);
    assert(4) globalState == 3;

    callSwitchWithEnumeration(THUESDAY);
    assert(5) globalState == 2;
  } testSwitchCase(test case)

  void callSwitich(int32_t state) {
    switch ( state ) {
      case 1: { globalState = 0; break; }
      case 2: { globalState = 10; break; }
      default: { globalState = 20; break; }
    } switch
  } callSwitich (function)

  void callSwitchWithEnumeration(DAY day) {
    switch ( day ) {
      case MONDAY: { globalState = 1; break; }
      case THUESDAY: { globalState = 2; break; }
      case WEDNESDAY: { globalState = 3; break; }
    } switch
  } callSwitchWithEnumeration (function)
}
```

4.12 Variables

Global variables Global variables start with the keyword `var`. In every other respect they are identical to regular C.

```
module GlobalVariables from HelloWorld.Variables imports nothing {

    var int32_t globalInt32;

    exported test case testGlobalVariables {
        setGlobalVar(10);
        assert(0) globalInt32 == 10;
        setGlobalVar(20);
        assert(1) globalInt32 == 20;
        return;
    } testGlobalVariables(test case)

    void setGlobalVar(int32_t globalVarValue) {
        globalInt32 = globalVarValue;
    } setGlobalVar (function)
}
```

Local variables At this point a local variable declaration can only declare one variable at a time; otherwise it is just like in C.

4.13 Arrays

Array brackets must shown up after the type, not the variable name. The following example shows the usage of arrays in mbeddr C. mbeddr C also supports multi dimensional arrays and the usage is equivalent to regular C.

```
module ArrayApplication from HelloWorld.Arrays imports nothing {

    exported test case arrayTest {
        int32_t[3] array = {1, 2, 3};

        assert(0) array[0] == 1;

        int8_t[2][2] array2 = {{1, 2}, {3, 4}};
        assert(1) array2[1][1] == 4;
    } arrayTest(test case)
}
```

4.14 Reporting

Reporting or logging is provided as a special concept. Its designed as a platform-independent reporting system. With the current generator and the `desktop` setting in the build configuration, `report` statements are generated into a `printf`. For other target platforms, other translations will be supported in the future, for example, by storing the message into some kind of error memory.

If you want to use reporting in your module, you first have to define a `message list` in a module (see Fig. 4.14). Inside, you can add `MessageDefinitions` with three different severities:

- ERROR (default)
- INFO
- WARN

Every message definition has a name (acts as an identifier to reference a message in a report statement), a severity, a string message and any number of additional arguments. Currently, only integer values and strings are allowed.

A `report` statement references a message from a message list and supplies values for all arguments defined by the message. The following example shows an example.

```
module Reporting from HelloWorld.Reporting imports nothing {  
  
  message list demo {  
    INFO programStarted() & "Program has just started running"  
    ERROR noArgumentPassedIn(int16_t actualArgCount) active:  
      No argument has been passed in, although an arg is expected  
  }  
  
  int32_t main(int8_t argc, string[ ] argv) {  
    report(0) demo.programStarted() on argv == 0;  
    report(1) demo.noArgumentPassedIn(argv[0]) on argc == 0;  
    return 0;  
  } main (function)  
}
```

Note how the first report statement outputs the message in all cases. The second one only outputs the message if a condition is met.

Report statements can be disabled; this removes all the code from the program, so no overhead is entailed. Intentions on the message definition support enabling and disabling messages. It is also possible to enable/disable groups of messages by using intentions on the message list.

Note: At this time there is no way of enabling/disabling messages at runtime. This will be added in the future.



5 Command Line Generation

mbeddr C models can be generated to C code from the command line using ant. The HelloWorld project comes with an example ant file: in the project root directory, you can find a `build.xml` ant file:

```
<project name="HelloWorld" default="build">

  <property file="build.properties"/>

  <taskdef resource="jetbrains/mps/build/ant/antlib.xml"
    classpath="\${mps.home}/languages/generate.ant.task.jar"/>


  <target name="build">
    <mps.generate loglevel="info" fork="true" failonerror="true">
      <jvmargs id="myargs">
        <arg value="-Xmx512m"/>
      </jvmargs>
      <project file="\${mbeddr.home}/code/applications/HelloWorld/HelloWorld.mpr"/>
      <library name="mbeddr" dir="\${mbeddr.home}/code/languages"/>
    </mps.generate>
  </target>
</project>
```

It uses the `mps.generate` task provided with MPS. All the code is boilerplate, except these two lines:

```
<project file="\${mbeddr.home}/code/applications/HelloWorld/HelloWorld.mpr"/>
<library name="mbeddr" dir="\${mbeddr.home}/code/languages"/>
```

The first line specifies the project whose contents should be generated. We point to the `HelloWorld.mpr` project in our case. If you only want to generate parts of a project (only some solutions or models), take a look at this article: <http://confluence.jetbrains.net/display/MPSD2/HowTo+-+MPS+and+ant>

The second line points to the directory that contains all the languages used by the to-be-generated project.

To make it work, you have to 

To make it work, you also have to provide a `build.properties` file to define two path variables:

```
mps.home=/some/path/to/MPD2.0/
mbeddr.home=/the/path/to/mbeddr/
```

Assuming you have installed `ant`, you can simply type `ant` at the command prompt in the directory that contains the `build.xml` file. Unfortunately, generation takes quite some time to execute (50 seconds on my machine). However,

most if the time is startup and loading all the languages, so having a bigger program won't make much of a difference. The output should look like this:

```
L:\lwes-assembla\mbeddr\code\applications\HelloWorld>ant
Buildfile: build.xml

build:
[mps.generate] Build number MPS Build.MPS-20.7460
[mps.generate] Loaded project MPSProject file: L:\lwes-assembla\mbeddr\code\applications\HelloWorld\HelloWorld.mpr
[mps.generate] Per-root generation set to false
[mps.generate] Generating:
[mps.generate]     MPSProject file: L:\lwes-assembla\mbeddr\code\applications\HelloWorld\HelloWorld.mpr

BUILD SUCCESSFUL
Total time: 57 seconds
```

You can now run `make` to build the executable.

6 Debugging

Note: The debugger is not yet finished. Will be available in early 2012.