An MEL/C Hello World

Version 1.0, Jan 23, 2011

Markus Völter
(voelter@acm.org)

Introduction

This is a quick hello world tutorial for the MED/C hosted at *mbeddr.com*. You need to install MPS 1.5.1 from

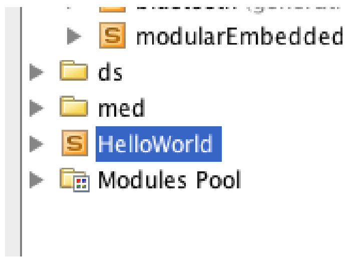
http://www.jetbrains.com/mps/download/index.html

and check out (or download) the MEL/C from *mbeddr.com* or the code hosting site at

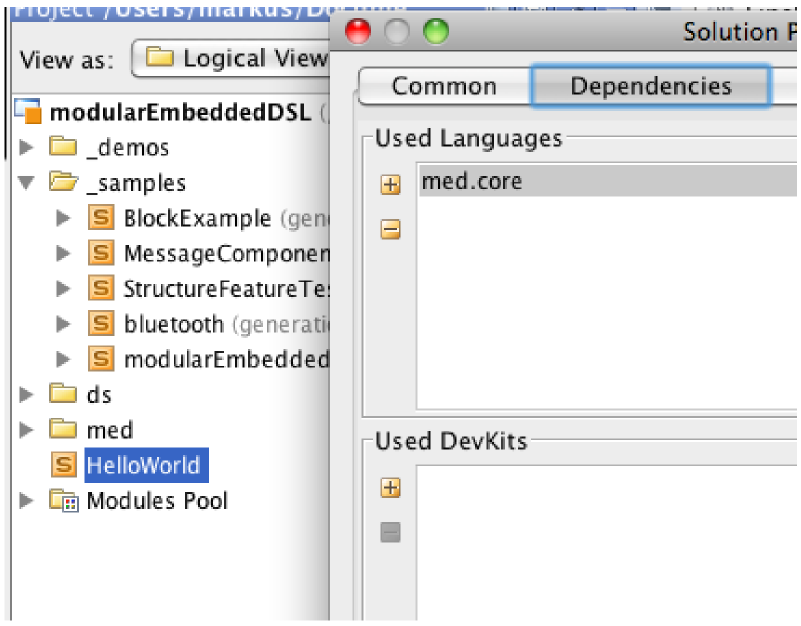
http://code.google.com/p/mpscmindstorms/

Step 1: A function

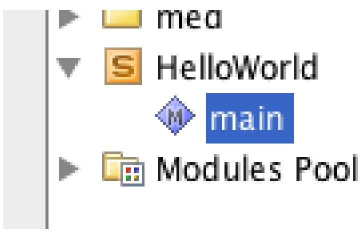
After downloading/checking out the MEL source code, create a new solution in the current project; we call it *HelloWorld*



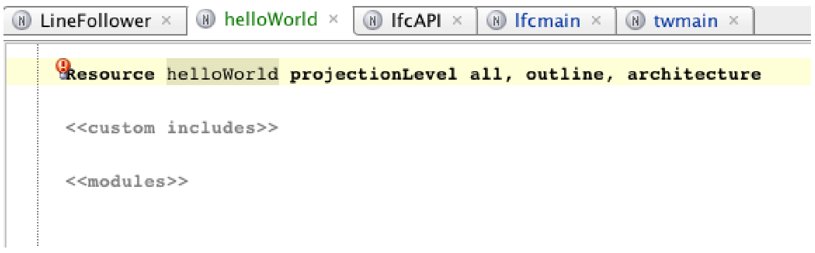
Then use the solution properties to define a dependency on the *med.core* language.



We can now create a new model in the solution; we call it *main* and also create a depenency to *med.core* as the used language. This is how it looks afterwards:

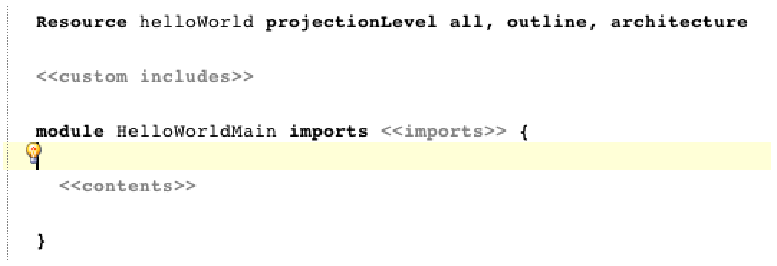


In this model we can now create a new *Resource*. Resources are the top-level elements of MEL programs. Use the context menu on the model, and select *New->med.core->Resource*. You can name the Resource *helloWorld*.



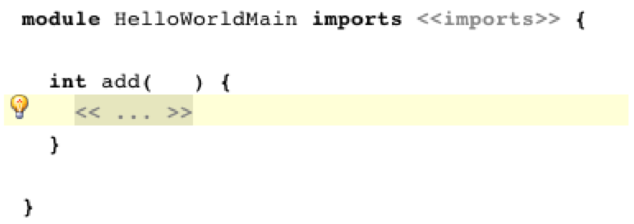
```
Resource helloWorld projectionLevel all, outline, architecture

  <<custom includes>>

  <<modules>>
```

Resources contain any number of *modules*; modules are a bit like namespaces, but currently not nested. So in the modules slot of the Resource press Ctrl-Space and select *module*. Call the module *HelloWorldMain*.



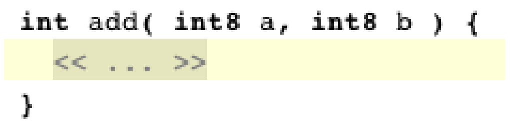
```
Resource helloWorld projectionLevel all, outline, architecture

<<custom includes>>

module HelloWorldMain imports <<imports>> {

  <<contents>>

}
```

You are now ready to write some actual code. Let's start with a function that adds two numbers. In the contents type the word *procedure* (or select it from the code completion menu). Call the procedure *add*. To change the type, move the cursor over the *void* return type and press Ctrl-Space twice. This gives you a list of all available types. Select *int*.



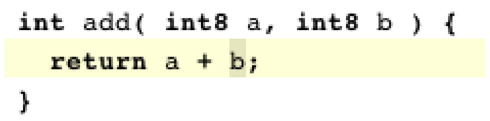
```
module HelloWorldMain imports <<imports>> {

  int add(   ) {
    << ... >>
  }

}
```

We can now add the two arguments. Position the cursor between the parentheses, and press Enter. This will automatically create a new parameter with an *int8* type, which is ok for our purpose. Call the parameter *a*, then press comma. You'll get another parameter, call it *b*.



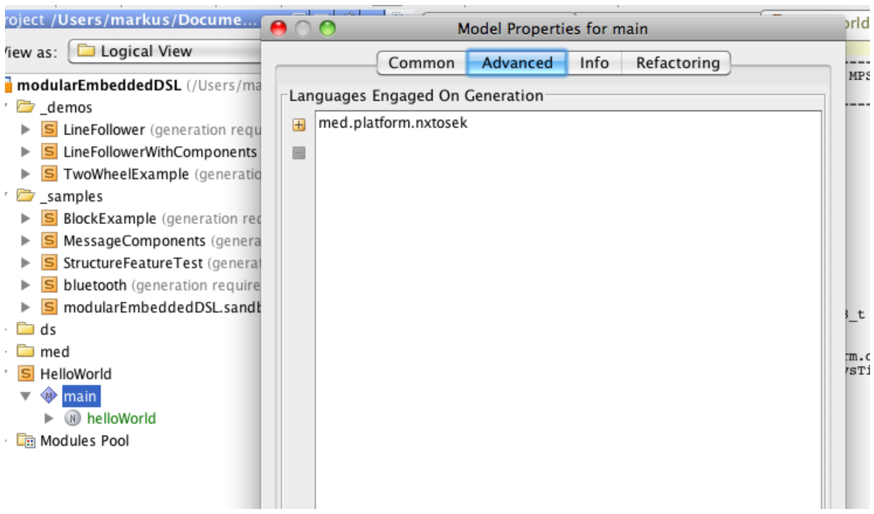
```
int add( int8 a, int8 b ) {
  << ... >>
}
```

Move the cursor into the body and type return. It binds to a *return* statement and moves the cursor into the expression field. Type *a+b* there.



```
int add( int8 a, int8 b ) {
  return a + b;
}
```

Before we can compile the program, we have to determine whether we want to run it on OSEK or on Win32. Let's choose Osek. Go the the model properties, select the Advanced tab, and add *med.platform.nxtosek* to the *Languages Engaged on Generation*:


roject /Users/markus/Docume...
View as: Logical View
modularEmbeddedDSL (/Users/ma
_demos
LineFollower (generation requ
LineFollowerWithComponents
TwoWheelExample (generatio
_samples
BlockExample (generation re
MessageComponents (genera
StructureFeatureTest (genera
bluetooth (generation require
modularEmbeddedDSL.sandb
ds
med
HelloWorld
main
helloWorld
Modules Pool
Model Properties for main
Common
Advanced
Info
Refactoring

Languages Engaged On Generation

med.platform.nxtosek

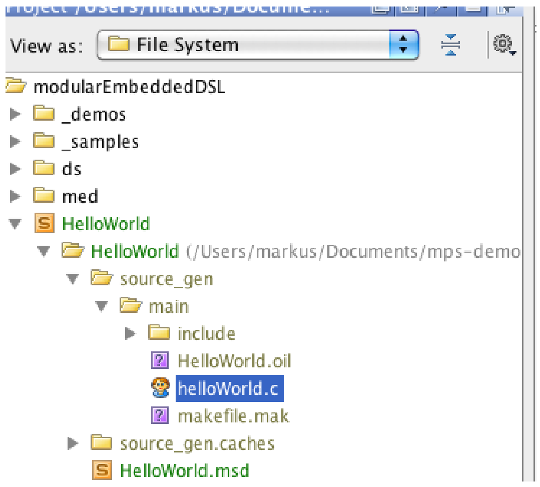
Back in the code, you can now press Ctrl-F9 (Cmd-F9 on the Mac) to generate the C code. You can also select the context menu on the model and select *Generate Files*.

This is the C code you will get (you can find it in the file system, in the *source-gen* directory of your project/solution.) Notice how some of the Osek boilerplate has been added.

```
/*
---------------------------------------------------------------
   header generated from modularEmbeddedDSL via MPS
   resource helloWorld
---------------------------------------------------------------
*/

// the header of this file
#include "include/helloWorld.h"

// used resources

// custom includes
#include "kernel.h"
#include "kernel_id.h"
#include "stdint.h"
#include "stdint.h"


int helloWorld_HelloWorldMain_add(int8_t a, int8_t b) {
    return (a + b);
}
DeclareCounter(SysTimerCnt); // added by
platform.osek:addCounterTrigger
void user_1ms_isr_type2(void) { SignalCounter(SysTimerCnt); } //
added by platform.osek:addCounterTrigger
```

To be able to compile it, we are still missing a make file. So let's go back to our model, and create a new root object of type *System*.

Call the System *HelloWorld* as well (name doesn't matter, actually). In the resources slot, press Ctrl-Space and select our *HelloWorld* resource. If you regenerate the files, you will see more generated code:



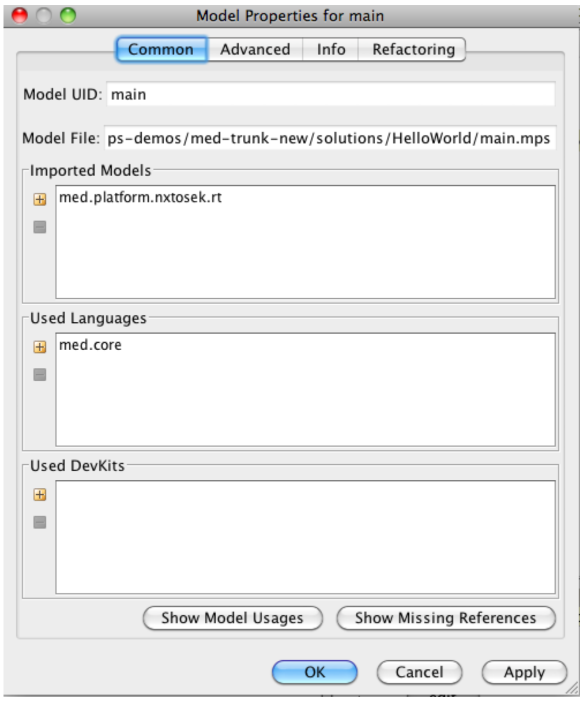
```
modularEmbeddedDSL
  _demos
  _samples
  ds
  med
  HelloWorld
    HelloWorld (/Users/markus/Documents/mps-demo
      source_gen
        main
          include
          HelloWorld.oil
          helloWorld.c
          makefile.mak
      source_gen.caches
      HelloWorld.msd
```

Take a look at the *oil* file and the make file. The system should now be compilable with the Lejos-OSEK tool chain for Lego Mindstorms at:

http://lejos-osek.sourceforge.net/

Step 2: Calling an external function to output „Hello World"

Let us now write a real hello world that outputs that string onto the lego mindstorms device. Go to the model properties of your model and create a dependency to the *med.platform.nxtosek.rt* model:


Model Properties for main

Common | Advanced | Info | Refactoring

Model UID: main

Model File: ps-demos/med-trunk-new/solutions/HelloWorld/main.mps

Imported Models: med.platform.nxtosek.rt

Used Languages: med.core

Used DevKits:

Back in your module, you can now press Ctrl-Space in the imports slot, and select *ECAPI*. This is an MPS-ified version of (part of) the lego mindstorms API. In our add operation, you can now call ECAPI functions, for example like this:

```
int add( int8 a, int8 b ) {
  display_clear( );
  display_string("Hello, World");
  return a + b;
}
```

Notice how this demonstrates how to call into existing non-MPS-ed libraries. We refer to the header, and the generated make file will link the respective implementations automatically.

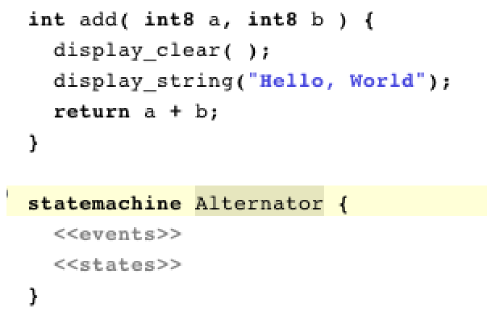
Step 3: C Extensions

The whole point of the MEL is that C can be mixed with other abstractions. The available abstractions are described in the various papers I wrote, such as this one:

http://www.voelter.de/papers/Voelter-EmbeddedSystemsDevelopmentWithProjectionalLanguageWorkbenches.pdf

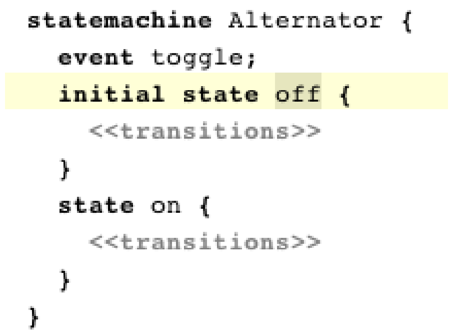
Let's add a simple thing so you get a feel for how to use the extensions:

In the used languages section of the model properties of your main model, add the *med.statemachine* language. As a consequence, you can now instantiate state machines in modules. Position the cursor after the closing brace of the procedure you had created before, and press Enter. Type *state machine* then. This creates a new state machine. Name it *Alternator*.



```
int add( int8 a, int8 b ) {
  display_clear( );
  display_string("Hello, World");
  return a + b;
}

statemachine Alternator {
  <<events>>
  <<states>>
}
```

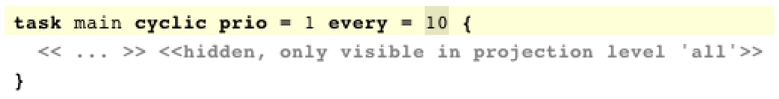
Let's create an event *toggle*. Press Enter in the *events* slot and name the new event *toggle*. Create two states *on* and *off* by pressing enter and entering the name. Make the *off* state the initial state by pressing Alt-Enter and selecting *statemachine: make initial*.



```
statemachine Alternator {
  event toggle;
  initial state off {
    <<transitions>>
  }
  state on {
    <<transitions>>
  }
}
```

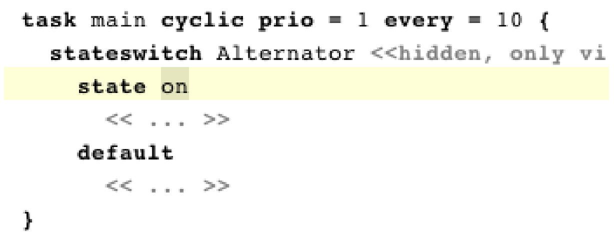
Let us then add transitions to each state that go to the other if the event is received. You do it by pressing Enter in the *transitions* slot, selecting (Ctrl-Space) the source event (*toggle*) and selecting (Ctrl-Space) the target state (*on* or *off*).

Let us now create a task. Create a dependency in the model's properties to the *med.tasks* language. You can also simply press Ctrl-L to import a new language. Once you did that, press Enter behind the closing parens of the state machine or the procedure, and type *task*. Name in *main*. Behind the name press Ctrl-Enter and select cyclic. In the every slot put 10 or something.



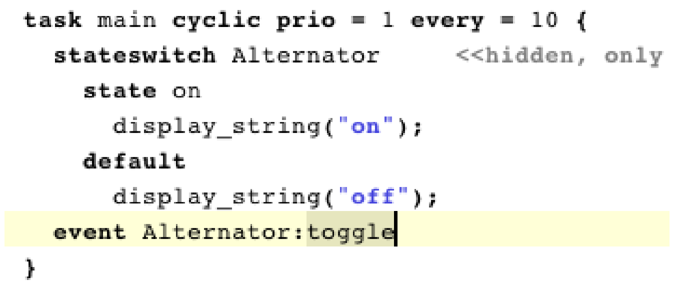
```
task main cyclic prio = 1 every = 10 {
  << ... >> <<hidden, only visible in projection level 'all'>>
}
```

We want the cyclic task to print something if the state machine is in the *on* state. Otherwise, print nothing. In the body of the task type *stateswitch*. Select our *Alternator* state machine as the target machine. Press Enter in the cases slot. Select *on* in the state field of the case.



```
task main cyclic prio = 1 every = 10 {
  stateswitch Alternator <<hidden, only vi
    state on
      << ... >>
    default
      << ... >>
}
```

In the body of the on state case, print *on* to the display; Display *off* in the default case. Finally, add a *event* statement at the end of the complete task to “toggle” the state machine.



```
task main cyclic prio = 1 every = 10 {
  stateswitch Alternator     <<hidden, only
    state on
      display_string("on");
    default
      display_string("off");
  event Alternator:toggle
}
```

Note: instead of using the *stateswitch* in the task, we could have used entry actions in the state of the statemachine. But I wanted to demonstrate the *stateswitch*.

We can now regenerate the model (Ctrl-F9, or from the context menu). Take a look at the generated code to understand what we have done in terms of C code.

Step 4: Your own extension

Please watch the screencasts at

http://mbeddr.wordpress.com/about

to learn how to build your own extensions. Read this tutorial

http://code.google.com/p/mps-lwc11/wiki/GettingStarted

to better understand the details behind it.