MIML Tutorial

Mike Hooper

This document is designed so users can get a brief overview of why miml, and how the language works. Users should be able to understand miml but not be at a point to modify the language or code generator. For modifying the code generator and the miml files, please reference the code generator documentation.

Team Elderberry

Ron Astin, Chris Glasser, Jordan Hewitt, Mike Hooper, Josef Mihalits, and Clark Wachsmuth

2/13/2013

Overview

This document is divided into 3 main sections. The first section is setup. This shows the user what is needed on any system to run miml properly. The second section debriefs the user on how to run a miml file through the code generator. The last section is dedicated to understanding how miml works. This section attempts to breakdown two key miml files in a way that a user can understand.

Setup

Below are the prerequisites to being able to run the miml code generator.

Python3

Users need to have python 3 installed on their system, and also have it resolvable in the user's path (ie. `/usr/bin`) To check whether or not you have python 3, use the which command. It should appear as below:

```
> which python3
/usr/bin/python3
```

If you don't have python 3 installed on your system run the command:

```
> sudo apt-get install python3
[sudo] password for <user>:
```

pyYAML

Download pyYAML from the link below:

http://pyyaml.org/download/pyyaml/PyYAML-3.10.tar.gz

After navigating to the directory where the file downloaded to, and after extracting the file.

```
>sudo python3 setup.py install
[sudo] password for <user>:
```

Running MIML through the Code Generator

First, you need to navigate to where the miml file you wish to use is located.

For example, we will use the Main.miml file from the rock paper scissors demo.

```
> pwd
<location of repo on local system>/elderberry/demos/rockpaper
```

The code generator and cg.conf is located in the root structure of the repository. The generator is dependent on cg.conf and needs this file to run. You must copy cg.conf to the folder you are working in.

To run the generator:

```
> cp ../../cg.conf ./
> ../../codeGen.py Main.miml
```

The code generator will generate fcfmain.c, fcfmain.h, and miml.mk

The make file is important because it allows the linker to work, and the code generator to run via make.

The header file allows the user modules to compile

The source file contains all the generated functions and the initialize and finalize functions.

Understanding Miml

This section we will be using two files to demonstrate how miml works. This is a very generic overview of the language and how it functions. In this section we will be focusing on two files and how they relate to each other. The first file is SensorFusion.miml and the second is Main.miml Both files are located in `<repository on local system>/miml_examples`

SensorFusion.miml

```
%YAML 1.2
---
include: sensorfusion.h
object: sensorfusion.o
init: sensorFusionInit();
senders:
  sendSFPositionData:
  - [source, int32_t]
  - [x, int]
  - [y, int]
  - [z, int]
  sendSFLogMessage:
  - [source, int32_t]
  - [buffer, char*]
receivers:
  getIMUPositionData:
  - [source, int32_t]
  - [x, int]
  - [y, int]
  - [z, int]
```

The first two lines are for formatting and explanation.
`---` represents the start of the functioning content of the file.

Colons denote hash key values. For example sendSFPositionData: is the key value for all the dashes below it. Ordering is not preserved because this is a hash not an array.

Dashes are sequenced lists or arrays. Here ordering matters. Arrays are defined as [param_name, type]. This allocates the variable or parameter with the indicated type.

Only types used in miml (excluding parameter types) are scalars, hashes, and arrays.

Main.miml

```
%YAML 1.2
---
sources:
- [LOG, Logger.miml]
- [IMU, IMU.miml]
- [SF, SensorFusion.miml]
- [CTL, Control.miml]
messages:
  # IMU Messages
  IMU.sendIMUData:
    - SF.getIMUPositionData
    - LOG.getPositionMessage
  IMU.sendGPSData:
    - SF.getIMUPositionData
    - LOG.getPositionMessage
  IMU.sendALTData:
    - SF.getIMUPositionData
    - LOG.getPositionMessage
  IMU.sendIMULogMessage:
    - LOG.getLogMessage
  # Sensor Fusion Messages
  SF.sendSFPositionData:
    - CTL.getSFPositionData
    - LOG.getPositionMessage
  SF.sendSFLogMessage:
    - LOG.getLogMessage
  # Control Messages
  CTL.sendRollControlAdjustment:
    # I did not define the next module in this example but this
    # would be the receiver for altering roll control hardware.
    - LOG.getPositionMessage
  CTL.sendCTLLogMessage:
    - LOG.getLogMessage
```

# denotes comments, these will not be parsed.

Sources is a hash key of lists of sequences. These sequences are tokens. These tokens reference files listed as the second argument of the sequence. For example SF is the token for SensorFusion.miml

Messages is a hash of hashes containing lists. The key messages references different token.

The messages map contains keys of senders and values which are sequences of receivers. Sequence order specifies call order when the sender functions are generated. All data in the messages map are formatted as two part strings:

"token"."function"

Where "token" refers to a defined source token, and function refers to a defined sender or receiver within that source. During validation the code generator checks to ensure that message keys are defined as senders while sequenced values are defined as receivers.

Conclusion

When a function is defined in miml, it can be called in any piece of your code with the proper include statements without any declaration or prototypes. This is because the code generator will have its own source and header file defining the function. So if the header file is included in your source code, it will resolve with no error.