Module Messaging Extended Library, version 2.0

Source code: https://github.com/ADSWNJ/ModuleMessagingExt and https://github.com/ADSWNJ/MMExt2

Intro

For a long time, we (i.e. Szymon and Andrew) have championed the ability for Orbiter modules (e.g. vessels, MFDs) to be able to communicate with each other in a way that makes the simulation experience more seamless. For example, would it not be natural to allow TransX to talk to LaunchMFD to synchronize the launch parameters, or for Glideslope 2 to be able to look at BaseSync's determination of the best orbit to do a de-orbit burn? The Module Messaging series of library utilities made this possible.

History

The original release of this library was in 2014, as Module Messaging. This implemented the core capability to send and receive Booleans, Integers, Doubles, VECTOR3, MATRIX3, and MATRIX4. This release was enhanced in 2016 with the release of Module Messaging Extended 1.0 and then 1.1, which added logic to support variable passing per vessel in a simulation (e.g. for different burn data for a fleet of vessels), and the initial ability to support the sending and receiving of arbitrary data structures by reference (e.g. to expose a larger set of parameters without having to constantly call dozens of put or get functions just to synchronize data). We were worried about causing unnecessary crashes to desktops by having modules referencing different versions of structures or different compiler implementations, or for a receiving module to be able to overwrite a sending module's data, so we added tons of safeguards for this. And so, we had a good library, and it has been widely implemented now for some of the core addons in the simulation, including TransX, LaunchMFD, BaseSyncMFD, Glideslope 2 and RV Orientation.

In 2017, we released the second version of Module Messaging Extended. This was driven by three goals: (1) stop the Error 126 errors and the dozens of Orbiter Forum threads asking why a particular MFD cannot load (due to a missing dependency), (2) provide a simple and an advanced code experience to interact with Module Messaging according to what the developer prefers, and (3) provide backwards compatibility for any existing ModuleMessagingExt 1.1 module clients. We have achieved all these goals in this release.

How Do I Use This Utility?

There are two 'facades' or interfaces, to Module Messaging Extended – basic and advanced. The Basic Interface supports Put, Get, and Delete functions for int, bool, double, VECTOR3, MATRIX3, MATRIX4, and std::string, for your current vessel. The Advanced Interface supports all the basic functions, plus two structure pass-by-reference functions, plus the ability to put, get, and delete for alternative vessels. You can use both interfaces interchangeably in your code (i.e. the data you can put or get in the Basic

Interface can also be accessed from the Advanced Interface).

The Basic Interface

1. From the place where you need to get data from MMExt2 (e.g. in your class header file):

```
#include "MMExt2_Basic.hpp"
```

2. In your class header, declare the interface, e.g. mm, with your choice of module name. (Replace the red text with your preference).

```
MMExt2::Basic mm;
```

3. In your class constructor, initialize the interface:

```
bool ret = mm.Init(”name”);
```

… where:
”name” is your module’s name (i.e. replace with your choice)
ret is true if the module finds MMExt2.dll, false if not (e.g. so you can put a message in the Orbiter.log, or notify the user on your MFD.)

4. In your code – add Put, Delete, and Get calls like this:

```
bool ret;
ret = mm.Put(”var”, val);
ret = mm.Delete(”var”);
ret = mm.Get(”other_module”, ”var”, &val);
```

… where:
”var” is your specific variable name (e.g. “Target”, “Burn_Time”, etc.)
&val is the address for the returned value for the variable name (as an int, bool, double, VECTOR3, MATRIX3, MATRIX4, or std::string), so long as the function returns true.
”other_module” is the name of the other MFD sending you the data,
ret is true if the function was successful

5. Usage notes for Put and Delete:
    a. Pick descriptive names for your variables, to make it clear what the data is.
    b. Do not duplicate names for different variable types, even though technically it is possible.
    c. All data is passed by value – i.e. when you put the data, MMExt2 takes a copy. If you want pass-by-reference semantics, then look at the structure-passing in the advanced interface.

d. Delete is only called from the module that Puts the data. I.e. if you are just getting the data, you never call Delete, as you are not the owner of that data.
e. The Delete function removes all data types for your module name and with that variable name. I.e. consider it a wildcard delete for int, bool, double, VECTOR3, MATRIX3, MATRIX4, and std::string.
f. Delete will not remove any data in your code – it just removes the copy in MMExt2, such that all further calls to Get will return `false`.

6. Usage notes for Get:
    a. Look at the documentation or the source code for the other module to determine what data variables and data types it is sending.
    b. When the Get completes, you will have a private copy of the data returned into your code. You need to repeat the Get call to get refreshed data – e.g. every second, every simulation step, or manually on request from your user. If you want a reference to a live variable in the other module, then have a look at the structure-passing capabilities in the advanced interface.

The Advanced Interface

This is an extension of the Basic Interface, so we will only point out the things that are different.

1. Include, declare, and initialize a MMExt2::Advanced interface with these commands in the appropriate places (replaing the red text with your choices):

```
#include "MMExt2_Advanced.hpp"
MMExt2::Basic mm;
bool ret = mm.Init(”name”);
```

2. Put, Delete, and Get calls:

```
ret = mm.Put(”var”, val, vessel);
ret = mm.Delete(”var”, vessel);
ret = mm.Get(”other_module”, ”var”, &val, vessel);
```

… where:
`vessel` is a `*VESSEL` pointer to the vessel definition. It is an optional parameter, and defaults to the current focus vessel if you leave it out.

3. Usage notes:
    a. The vessel parameter allows you to access data for other vessels in the scenario. For example – you may be sharing burn or targeting data across a fleet of vessels, or accessing information from a target vessel for a rendezvous. The vessel must exist in the scenario when you reference it, so be aware of the creation and destruction events on vessels if you want to use this functionality.

b. In addition to the standard data types in the Basic Interface, the Advanced Interface can also pass pointers to data structures. As this is a complex topic, we present it below in a separate section.

The Advanced Interface Structure Handling

Until now, we have worked with a set of simple Orbiter data types. We also handle std::string, with some careful work in the implementation to avoid compiler-specific dependencies with std::string binary formats.

When you pass data by reference in a structure, you get all the advantages of direct access to the internal data in the other module, but you are also exposed to many compiler-specific implementation details that can cause problems for data interchange. The structure handling code in MMExt2 tries as hard as possible to shield you from problems, but the key message is not to expose anything that uses the standard template library (e.g. no strings, vectors, maps). We want to avoid is a crash to desktop caused by modules accidentally referencing bad data structures, or making false assumptions about application binary implementations across versions of the compiler.

We implement five safety-checks to try to avoid these problems:

1. We insist that the class or struct you wish to pass is derived from one of two root structures we implement in Module Messaging. This allows us to cast pointers to the root class, and to implement the remaining checks.

2. We insist on a version number when you instantiate your class, and we check for that version number in the get module at runtime. This makes sure that the getting code is defining that same version of the export/import data structure. (The code putting the external structure should give you a header file defining the implementation.)

3. We implement run-time size checks for the structure, in case either the compiler implements the size differently, or the coder of the module modifies the structure without changing the version number. If the size is incorrect, we do not allow the pointer to be passed to the receiving module.

4. We insist on the receiving pointer being a const pointer. This prevents the receiving side from accidentally or intentionally trying to change the sending side's data structure. If you want to do such things, establish a two-way interchange and a message protocol of your own between the two modules, and have each module expose their sending side as separate structures.

5. The last check is to recommend that the structure packing is explicitly defined via #pragma pack directives. This removes yet another way for the various compiler versions to trip you up by

putting miscellaneous whitespace in your structure for byte-alignment reasons.

OK – with all those caveats, here's the code you need to publish (put) an MMStruct-derived structure from your code :

1. Make a header file describing your external-facing structure – e.g. MyModule_ExportStruct.hpp, using this template:

```
// Module Messaging Export Structure for MY MODULE
#pragma once
#include "MMExt2_Advanced.hpp"
#pragma pack(push)
#pragma pack(8)
#define MYSTRUCT_VER 1
struct ExportStruct : public MMExt2::MMStruct {
   ExportStruct() : MMExt2::MMStruct(MYSTRUCT_VER,
                                     sizeof(MyStruct)) {};

   // … add your elements here …
   // … e.g. bool valid;

};
#pragma pack(pop)
```

2. Make sure that this export structure derives from public MMExt2::MMStruct, and has no standard template library components inside.

3. In your class header, declare the structure in a place that will not go out of scope whilst the vessel is active. Sample code:

```
struct ExportStruct mm_export;
```

4. In your code, publish the structure via your Module Messaging Extended Advanced Interface::

```
ret = mm.PutMMStruct("var", &mm_export, vessel);
```

5. Note – once you have published a structure, there is no Delete function, as you may have other modules directly accessing your private memory location via your published pointer. Consider using a Boolean flag in the structure to indicate if the data is valid.

On the other side, to access (get) the structure, do this:

1. Include the header from the other module (e.g. they should put it in an accessible place suck as Orbitersdk\include):

```
#include "MyModule_ExportStruct.hpp"
```

2. In your class header, define a constant pointer to the export structure (note: it must be constant, to guarantee that the client does not write to the structure):

```
const ExportStruct *pExportStruct;
```

3. In your code, establish a connection to the structure via your Module Messaging Extended Advanced Interface:

```
ret = mm.GetMMStruct("mod", "var", &pExportStruct, vessel);
```

4. Assuming you have a successful return code, you may now access the live data in the export structure without any further Module Messaging commands – e.g.

```
if (pExportStruct->valid) // do something…
```

Compatibility with previous ModuleMessagingExt modules

MMExt2 is backwards-compatible with ModuleMessagingExt 1.1 modules. By installing ModuleMessagingExt 2.0, any older client will access MMExt2.dll indirectly via an updated ModuleMessagingExt.dll. Clients of the older ModuleMessagingExt 1.1 will only be able to access the following data types: int, bool, double, VECTOR3, MATRIX3, MATRIX4, and EnjoLib::ModuleMessagingExtBase derived structures. The new MMExt2::MMStruct-derived structures and the std::string data type are only available via the new MMExt2 interface.

A note from ADSWNJ

I want to recognize the partnership I have had for many years now with Szymon "Enjo" Ender, on this module and on many other MFDs in the Orbiter simulation. This library is the result of many emails and many hours of coding and testing between us.

- Andrew Stokes, December 2017