IMS2 Developers Guide – Module Functions

Table of Contents

Introduction

Module functions are the bread and butter of IMS2. They're the things that make most of the cool stuff happen. In a way I misnamed them a bit, it would have been more descriptive to name them "module abilities", really, because that's what they do. Every module function adds a potential ability to a module. Potential because the module can have that ability, but it doesn't have to. If you were familiar with the original IMS, you should take a moment to process that new design paradigm and its implications. There are no rigid module types anymore. In fact, there's currently only one module class. Instead of having certain module types that require certain parameters and provide certain abilities, there are now module functions that do that, but that can be added to any module by config file. This gives module designers a lot of freedom, up to the point where a single module can be a fully functional vessel. This was not quite the intent of the architecture, rather it wanted to enable more flexibility in designing modules that make logical sense or to more accurately model actually existing modules, but it's a neat side effect.
This guide will work with the communications module function as an example. It is a good idea to fire up orbiter and take a look at how the finished BCM101_Antenna module behaves before you start working through this guide, and to put your nose into its config file for a bit. Doing so will also be the best possibility for you to understand the new paradigm and its capabilities. This module uses the same mesh as the one in the old IMS, yet it is thouroughly different. This is because this isn't just a module with one function... you'll find that the config file declares a whopping *six* module functions, five of them being instances of the one used as an example in this guide (the remaining one being for animations, which are also a module function in IMS2, giving spacecraft3-ish flexibility when it comes to designing animations). In other words, when you look at this code, keep in mind that it doesn't serve to control an entire module's worth of communication equipment... it serves to controll a single antenna. That module could have only one, or ten, or a hundred antennas... we wouldn't have to change the code a bit. It's all there in the config file.

How to use this guide

This guide will take you through the creation of IMS2_ModuleFunction_Comm and its associated classes. Not every line of code will be expounded upon here, instead the guide focuses on teaching you to work with the elementary interfaces and components that make up a module function. I'll be mentioning when I skip over parts of the code, and I encourage you to look it up in the source and try to understand what's going on based on what you just learned.

Over the course of this tutorial you will learn to:

- Write an IMS_ModuleFunctionData class to handle loading your module function from config file.
- Work with the interface provided by IMS_ModuleFunction_Base to give your modulefunction the abilities to be added to and removed from IMS vessels as well as saving and loading their state to a scenario file.
- Use the StateMachine class to more easily handle complex states.
- Use the Event Engine to communicate with other parts of a module.
- Write a graphical user interface so your module function can be controlled by a user through the IMS2 panel.

Many times, you won't need all of these things at once, but you'll always need some of them, so pick and choose.

Prerequisites

The guide is written with an intermediary level in object-oriented programming in mind. Specifically, you should be able to understand the basics of polymorphism, such as inheritance, method overriding and casting, and you should know the basic concept of a state machine. I will in general do little to explain basic programming techniques, so if you never heard of some the above things you might want to do some reading up on them first. I do write a little about the basic concept of events, and I talk a bit about void pointers and function pointers towards the end when they are needed, because these concepts seem frequently confusing to hobbyists of all levels.

Setting up the files you're going to need

The first thing I recommend you to do is to just quickly create the classes you're going to need and hook them up with IMS2. Otherwise things might get tedious because you'll be flying without proper intellisense support.
Create a new class in the ModuleData filter (folders in the project explorer), we'll call it IMS_ModuleFunctionData_Comm for this example (yours will of course be named differently), that inherits IMS_ModuleFunctionData_Base.
In the ModuleFuncionalities filter, you should create a new filter and under that create another class inheriting from IMS_ModuleFunction_Base. In case of this example, it's called IMS_ModuleFunction_Comm. Don't worry, we'll look at why we created those classes and inherited the classes we inherited later on. For now, it's important that we have the classes, and the

next important thing is to let IMS2 know about them.
In the Modules filter, open Moduletypes.h. There's the FUNCTIONTYPE enumerator, which defines an identifier for all known function types. Add a new identifier for your modulefunction to this enumerator. In this case, I added MTYPE_COMM:

```
enum FUNCTIONTYPE
{
	MTYPE_NONE,
	MTYPE_PRESSURISED,
	MTYPE_ANIM,
	MTYPE_COMM
};
```

Next, open the file IMS_ModuleFunctionFactory.cpp, in the Modules/Module functionalities filter. This is the class that creates all module functions at runtime, but it needs a few lines of code for every new module function added. The first thing it needs is to know the classes we just created. Add the headers of both your data class as well as your function class to the include list at the top of the file. Make sure the include for IMS_ModuleFunction_Factory.h is still at the end of the list, though.

We proceed to CreateNewModuleFunction() in the same file. This is the factory function that instantiates a new module function based on a ModuleData instance. We have to equip this function so it can produce instances of our new module function, so we add a new case, using the FUNCTIONTYPE identifier we just created. The added lines should look like this:

```
case MTYPE_COMM:
{
	IMS_ModuleFunctionData_Comm *data = (IMS_ModuleFunctionData_Comm*)_data;
	return new IMS_ModuleFunction_Comm(data, module);
}
```

The only thing we're doing here is to cast the passed pointer to the generic ModuleFunctionData instance to our new ModuleFunctionData_Comm class, and then create an instance of the new module function using the data and the module the function is in as arguments.

Next we go to CreateNewModuleFunctionData(). This is kind of the same thing, but here we create an instance of the ModuleFunctionData class, which is pretty painless. Just add a new case using your identifier and return a pointer to a new instance of the new ModuleFunctionData class:

```
case MTYPE_COMM:
	return new IMS_ModuleFunctionData_Comm();
```

We're almost done here. What's left is to connect the enum identifier to a string identifier with which the ModuleFunction will be identified in config files. For this we have to modify two more functions. First go to GetFunctionTypeString() and add a new case that returns a string when it receives your FUNCTIONTYPE identifier. The string you return should be unique in the whole function, otherwise there will be trouble. In the case of our example, this looks like this:

```
case MTYPE_COMM:
	retString = "COMMUNICATION";
	break;
```

And we also have to do it the other way round, so IMS2 can generate the FUNCTIONTYPE identifier from a string. Go to GetFunctionTypeFromString() and add another simple oneliner:

```
if (identifier == "COMMUNICATION") return MTYPE_COMM;
```

It should be self-explanatory that the string here has to match the string in GetFunctionTypeString() exactly, but I decided to mention it anyways.

This concludes the preparation. All further code will be written exclusively in your new classes, without having to descend into the belly of the beast. With this done, IMS2 has all it needs to identify your module function in a config file and instantiate it.

How to write your IMS_ModuleFunctionData class

The smartest thing to do once IMS2 is aware of our new toy is to write the ModuleFunctionData class. This class will load and contain all static data of your module function as defined in a config file. First because we already pass a pointer to this data to the constructor of the module function, so it makes sense to have this working first, and second because it's healthy to think about what data your module function is going to need before you get started.

To start with your ModuleFunctionData class, inherit IMS_ModuleFunctionData_Base and overload its interface. Your header will look something like this:

```
class IMS_ModuleFunctionData_Comm :
	public IMS_ModuleFunctionData_Base
{
public:
	friend class IMS_ModuleFunction_Comm;

	IMS_ModuleFunctionData_Comm();
	~IMS_ModuleFunctionData_Comm();

private:

	virtual bool processConfigLine(vector<string> &tokens);
	virtual bool validateData(string configfile);

};
```

So what is what here exactly? First off, we're inheriting IMS_ModuleFunctionData_Base. This base class will handle the tedious interfacing with the core, so your class will receive the parts of the config file that oncern it. From here, things get very straightforward. We befriend our future ModuleFunction class, so it can access the data contained in this class freely without the data having to be public knowledge. This isn't strictly necessary, but it saves us the hassle of writing getter methods for all the properties. We also declare a default constructor and destructor.

It gets more interesting in the private section. For starters, we overload processConfigLine(). This function will receive the data stored in the config file line by line, and the best thing is that it will only receive the data that is relevant to this module function.

The other thing we need to overload is validateData(). This function will get called after the loading process is finished, and will give you the opportunity to verify if your module function received all the data it expected from the config file. This gets pretty important for propper feedback when people start to write config files.

We'll look at a detailed implementation of these functions in a minute, but first we need to think about a few things. Namely, we need to think about what data we are actually going to need. We're writing a module function that wasn't around before, and the way we write this class defines how the module function has to be defined in a config file, so we need to know what data we need and what data we don't.

We'll be pretty modest for now: The communications module function will only need the name of a deployment animation, the name of a scanning animation and the name of a tracking animation. That's it, that's all we want. But we do need to store this data in our ModuleFunctionData class, so let's add a few private members, which will already conclude the entirety of our header file (apart from a slight addition when we'll tackle the GUI):

```
class IMS_ModuleFunctionData_Comm :
	public IMS_ModuleFunctionData_Base
{
public:
	friend class IMS_ModuleFunction_Comm;

	IMS_ModuleFunctionData_Comm();
	~IMS_ModuleFunctionData_Comm();

private:
	string trackinganimname = "";
	string deployanimname = "";
	string searchanimname = "";

	virtual bool processConfigLine(vector<string> &rokens);
	virtual bool validateData(string configfile);

};
```

The trackinganimname, deployanimname and searchanimname will store the respective animation names defined in the config file. It's time to move to the actual code.

The first thing to do in IMS_ModuleFunctionData_Comm.cpp is to put in the appropriate includes. Just to pre-empt confusion that might occur at this moment, yes, the includes are all in the .cpp file, not the header file. This has reasons: IMS2 is a very complex project, and having includes in header files that declare classes tends to produce unimpregnable dependency labyrinths in a project of this size. The iron include rule of IMS2 is therefore: No includes in headers with class or function declarations! There are headers that include other headers. In fact, we're including one of them here (Common.h). But the only function of these headers is to group includes to make it easier to include different parts of the project! They do not declare classes that are defined in other files. I.e. including them from anywhere will never result in a build log filled with unresolved symbols or double declarations. In the worst case build times might increase slightly, though not nearly as much as they are increased by a poorly structured include tree.
The includes we need for this one are very modest:

```
#include "Common.h"
#include "IMS_ModuleFunctionData_Base.h"
#include "IMS_ModuleFunctionData_Comm.h"
```

Common.h declares a few structs that are widely used in IMS2, but mostly just includes classes and functions from the standard library, the stl, and of course the orbiter api. I.e. things that can be reasonably expected to be needed by everyone.
The other two includes should be rather self-explanatory: There's the header of our base class, and the header of the class we're about to implement.

Constructor and destructor are straightforward too:

```
IMS_ModuleFunctionData_Comm::IMS_ModuleFunctionData_Comm()
{
	type = MTYPE_COMM;
}

IMS_ModuleFunctionData_Comm::~IMS_ModuleFunctionData_Comm()
{
}
```

In the constructor, all we do is to set the type to the identifier we added to ModuleTypes.h. The destructor is empty. It might well be that yours look a bit different, but for this example, this is all we need. Let's implement our actual data loading.

```
bool IMS_ModuleFunctionData_Comm::processConfigLine(vector<string> &tokens)
{
	if (tokens[0].compare("tracking_anim") == 0)
	{
		trackinganimname = tokens[1];
		return true;
	}
	else if (tokens[0].compare("deploy_anim") == 0)
	{
		deployanimname = tokens[1];
		return true;
	}
	else if (tokens[0].compare("search_anim") == 0)
	{
		searchanimname = tokens[1];
		return true;
	}

	return false;
}
```

This might require a bit of explanation. As mentioned, the processConfigLine() function will receive the data from the config file line by line. The lines themselves are already split into tokens which on the original line were separated by whitespace and the "=" sign. This means that the first token will be the parameter name (which will be normalised to be all lowercase!), followed by the value, or values, depending on what you need and how you are expecting the data to be noted in the config file (these will be unaltered). Basically, you're now saying how the config file needs to be written. In our case it's again very simple: We have the parameter, followed by "=", followed by the

name of the animation associated with that parameter. For example, the line:

```
Tracking_anim = track2
```

in the config file will be fed to this function as 2 tokens, tracking_anim and track2.

All we have to do is compare the first token with the parameter names we expect, and if we have a match, store the value in the appropriate property. As mentioned, there are three possible parameters we need for this module function, which are tracking_anim, deploy_anim and search_anim. If a line contains any of these parameters, we store the value, and return true. If it's a parameter we don't know, we return false. Those return values are basically a convienience in case anybody ever wants to inherit from your module function to build on it. In that case, he will be able to call your code during loading and determine whether the parameter was already consumed by you, resulting in him not having to read them a second time.
Note that there is also the option to throw a runtime_error here if you see that a parameter has an invalid value. The base class calling the function will catch the exception, log its message as a warning to the Orbiter log together with the name of the offending config file, and continue.

But there is the very real possibility that our module function in the end will not be able to actually work if there were wrong or missing parameters. That's what the validateData() function is for:

```
bool IMS_ModuleFunctionData_Comm::validateData(string configfile)
{
	//there must be at least one set of controls defined, otherwise the whole
	//modulefunction makes no sense!
	if (deployanimname == "" && searchanimname == "" && trackinganimname == "")
	{
		Helpers::writeToLog(string(configfile +
					": must have at least one set of controls!"), L_ERROR);
		return false;
	}
	return true;
}
```

This will be called after all the lines concerning this module function have been processed. It's the place where you should evaluate the data you loaded, and decide whether or not your module function will be able to work as intended with the data you have.
In the case of our example, we want at least one of the three possible animations to be passed. The reason for this is flexibility: Somebody wants to make an antenna that cannot be deployed and retracted, but is able to track a target? He should be able to do that. There's an antenna that only needs to deploy and doesn't have any further capabilities? We don't want to make that impossible (in fact, the BCM101_Antenna.cfg has such an antenna.
On the other hand, the module function doesn't make sense if we can't do *anything* with it. Ergo, we want a module creator to supply at least one animation, and we're good.
If we didn't get that, we write an error to the log, including the name of the faulty config file, and we return false.
You shouldn't take that false on the light shoulder. Returning false from this function means your Module Function cannot work, and it will cause the calling class to throw and crash the simulation on purpose to prevent difficult to interpret behavior down the road.

And that is that... we determined what data we need, we loaded it, and finally we validated it. There's nothing more to do data-wise. Let's start with the actual module function!

The ModuleFunction interface

Again, the first thing we should be doing is overloading the necessary interface from the base class and defining the constructor. Don't worry, it's not very large. Go to the the header file of your ModuleFunction class, in this example IMS_ModuleFunction_Comm.h, and overload the abstract methods from the base class:

```
class IMS_ModuleFunction_Comm :
	public IMS_ModuleFunction_Base
{


public:

	IMS_ModuleFunction_Comm(IMS_ModuleFunctionData_Comm *_data, IMS_Module *_module,
				bool creategui = true);
	~IMS_ModuleFunction_Comm();

	//overloads from IMS_ModuleFunction_Base
	virtual void PostLoad();
	virtual void SaveState(FILEHANDLE scn);
	virtual void AddFunctionToVessel(IMS2 *vessel);
	virtual void RemoveFunctionFromVessel(IMS2 *vessel){};
	virtual void PreStep(double simdt, IMS2 *vessel);

protected:

	virtual void processScenarioLine(string line);
	virtual bool ProcessEvent(Event_Base *e);

private:
	IMS_ModuleFunctionData_Comm *data;
};
```

Let's take a quick look here at what is what:

PostLoad() will get called by the module containing the module function after everything concerning the module function, including a scenario state, has been loaded, but before the ModuleFunction is added to the vessel. Use it to initialise variables and processes that need data that is not known at compile time.

SaveState() is pretty self-explanatory: It is called when orbiter saves the scenario state, with a handle to the file. This is where we'll save all the dynamic properties of the module function to a scenario.

AddFunctionToVessel() is called whenever the module function is added to a vessel. This is at simulation start, when assimilating the module containing the module function, or when splitting of the module (since it has to be added to the new vessel it is now a part of). If your module function needs to add stuff to the core vessel, this is where you do it.

RemoveFunctionFromVessel(), in contrast, is called whenever the module function is removed from a vessel. This happens on vessel deletion, or in case of the containing module being split off. Whatever you add to a vessel in AddFunctionToVessel(), you have to remove again here!

PreStep() is called at the beginning of every frame, before orbiter applies the vessel state to the simulation. Basically, this is your controll loop, where you check states and process recurring tasks.

processScenarioLine() is similar to processConfigLine() in the module data class we just wrote. When the state is loaded from scenario, it will receive the relevant entries line by line. The only difference here is that this function receives the entry as an entire string and not as a set of tokens, to give you more freedom in your processing.

Finally, ProcessEvent() is an overload from EventHandler, not from IMS_ModuleFunction_Base. We'll look at it more indept when we cover event handling. For now, it's enoguh to know that by inheriting IMS_ModuleFunction_Base, you already registered your module function with the relevant event pipes, so you won't have to bother about that.

the data member will store a pointer to an instance of the data class we created beforehand, but that much should be obvious.

Next, I recommend you put function stubs for all these methods in your .cpp. Once you've done that, the whole thing should build, and you can for example test if your module data class actually works as intended. For convienience's sake, here's the cpp with the function stubs for IMS_ModuleFunction_Comm. Use it as a starting point when writing your own module functions. It will also serve to take a look at the includes we need up to here (continued on next page):

```
#include "GuiIncludes.h"
#include "ModuleFunctionIncludes.h"
#include "StateMachineIncludes.h"
#include "IMS_ModuleFunctionData_Comm.h"
#include "IMS_ModuleFunction_Comm.h"

IMS_ModuleFunction_Comm::IMS_ModuleFunction_Comm(IMS_ModuleFunctionData_Comm         *_data,
IMS_Module *_module, bool creategui)
	: IMS_ModuleFunction_Base(_data, _module, MTYPE_COMM), data(_data)
{
}

IMS_ModuleFunction_Comm::~IMS_ModuleFunction_Comm()
{
}

void IMS_ModuleFunction_Comm::PostLoad()
{
}

void IMS_ModuleFunction_Comm::PreStep(double simdt, IMS2 *vessel)
{
}

void IMS_ModuleFunction_Comm::AddFunctionToVessel(IMS2 *vessel)
{
}

bool IMS_ModuleFunction_Comm::ProcessEvent(Event_Base *e)
{
	return false;
```

```
}
void IMS_ModuleFunction_Comm::SaveState(FILEHANDLE scn)
{
}
void IMS_ModuleFunction_Comm::processScenarioLine(string line)
{
}
```

Pretty quaint for now, but we're going to put lots of code in here, as well as several more methods, don't you worry! For now, let's take a look at those includes:

GuiIncludes includes almost all of the GUI, as well as Common.h, the header file that includes all the basic libraries and declares the structs most widely used in IMS2. You'll have to include this even if you don't need a GUI for your module function, because the base class needs to know the classes. After all, it doesn't know whether your module function has a GUI or not, but needs to be prepared if it does.

ModuleFunctionIncludes takes care of the dependencies of IMS_ModuleFunction_Base as well as IMS_ModuleFunctionData_Base, among which is also the EventHandler. In other words, it includes everything you need to build a module function appart from your own header files.

StateMachineIncludes includes the state machine. This isn't mandatory, as it is very well possible that you don't need a state machine for your module function. However, in many cases (such as this example) the state machine will be an essential part of your module function.

IMS_ModuleFunctionData_Comm.h and IMS_ModuleFunction_Comm.h finally are the header files of our own classes.

If you did everything up to here correctly, your code should now build. If you make a config file for a module that uses your module function, you should be able to run orbiter and for example check whether your data class works as intended. But of course, your module function won't do anything yet. So, good job so far... we can finally start writing our actual module function!

Setting up the state machine

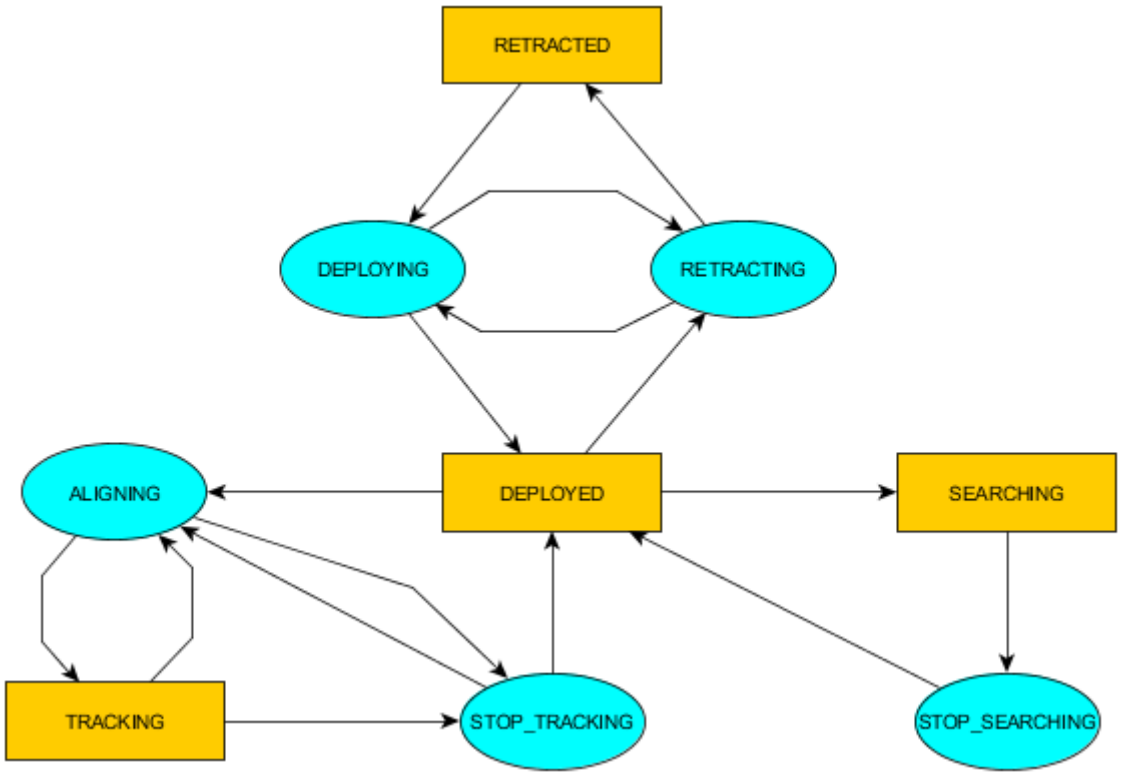
IMS2 has a very capable state machine built in to make things easier for you. Managing states will be an important task for many module functions, and some of them can become pretty complex. I'd consider the current example to be of moderate complexity, but this is already a level at which you do not want to manage your states with simple if-else statements, and even staticly defined states would become cumbersome. Let's take a look at what we got:

As we defined in IMS_ModuleFunction_Data_Comm, the module function has three animations: Deploy, scan, and track. Sounds simple enough. Believe it or not, these three simple animations result in 9 possible states the module function can currently be in.
First of all, it can of course be retracted, which it will usually be at the beginning. When the antenna deploys, it will take a time until it is fully deployed – the time the deploying animation needs to play out. During this time the antenna isn't retracted nor deployed. It is in an intermediate state, which we'll call "deploying". The same is true if the antenna was deployed and is currently in the process of retracting, so we'll need another state for that.

So, just to deploy and retract the thing, we already end up with 4 states: Retracting, Deploying, Deployed and Retracted.
If somebody wants to track a target, the antenna needs time to align itself, a state which we'll call "aligning". When it is aiming at the target it is in a different state again, "tracking". And finally, if we don't want the antenna to track anymore, it has to make its way back to its original alignement, or we'll get happy meshgroups flying around wildly when retracting it. We'll call this state "stop-tracking".
And then of course it can also be "scanning", and when it should stop with that, it again has to revert to its original position, which we name "stop-scanning". If you kept count, we now have the 9 possible states we need to adequately describe what our module function is currently doing. It's a smart move to gather these up and put them into an enumeration in the module function's header file, or our code will get very difficult to read indeed. The enum in this case looks like this (scanning here is named searching, because my brain must have farted at the time):

```
enum
{
	COMMSTATE_RETRACTED = 0,
	COMMSTATE_DEPLOYED = 1,
	COMMSTATE_DEPLOYING = 2,
	COMMSTATE_RETRACTING = 3,
	COMMSTATE_SEARCHING = 4,
	COMMSTATE_STOP_SEARCHING = 5,
	COMMSTATE_ALIGNING = 6,
	COMMSTATE_TRACKING = 7,
	COMMSTATE_STOP_TRACKING = 8
};
```

(note: It is not neccessary to pass out values manually, but I usually do it because it helps with debuging).
But the tricky thing of a state machine is of course to define how these states are connected. For example, if we are currently in COMMSTATE_TRACKING, the antenna of course cannot change its state to COMMSTATE_RETRACTED from one moment to another. It has to pass other states on the way there, in this case STOP_TRACKING, DEPLOYED and RETRACTING to finally arrive at the desired state of RETRACTED. This is always the fun part of state machines, where hilaritiy ensues while you are trying to get your head out of the labyrinth of nested if-statements you just built around yourself.
Fortunately, StateMachine takes most of that hassle of your hands by letting you literally create a map of your states and then decides which states it needs to pass by running a pathfinding algorithm on it. This makes things a lot easier, but not *too* easy. Let's take a moment and draw up a map of the above states and how they connect to each other:



Huh... that looks more complicated than expected, doesn't it? First of all, let's look at the arrows. They always only point in one direction. If there is a two-way connection between two states, then it's symbolised by two arrows. Why is that? Simply because you should think of a connection as being one way. For example, if the antenna is deploying, it will eventually enter the deployed state. Once it's there, it cannot go back to deploying state, although that's where it came from. But the way back to retracted leads only over the retracting state, and the way from retracted to deployed leads only over the deploying state. It's a one-way track, and the way back looks different than the way you came. That should be easy enough to understand for anyone that has ever navigated a decently sized city by car.

But sometimes, there is a connection one way, and one back the other way. It's still important that you think of them as two different connections, but they effectively enable direct transition between two states in either direction. For example, if the antenna is currently deploying, and the user changes his mind, there is nothing stopping him from switching to retracting state before it reaches deployed state. The animation will simply change direction. The reverse is also true, so if someone thinks it's big fun he can switch between these 2 states to and fro without ever reaching either retracted or deployed state. But people that take long-term joy in things like that usually don't play orbiter, so let's drop that train of thought.

Instead, let's look at the other pecularity of that diagram: There's two different kinds of boxes, rectangular ones and eliptic ones. These two types of boxes are stable states (rectangular) and intermediate states (eliptical). So what the hell are those?
Stable states are, as the name implies, stable – they don't change unless acted upon by an outside force (usually the user). Without outside pressure, a stable state will *never* change. But if we want to change it, we can change it at *any* time. There are no preconditions that have to be met for the state to change.

Contrast that to an intermediate state: An intermediate state will eventually reach its *end*, and when

it does so, has no choice but to evolve into the next state down the line. But until the conditions are met for the state to be considered "complete", it *cannot* change, even if the user wants it to. He'll just have to wait for nature to take its course.
To put it into a more practical example, Retracted is a stable state. As long as the user doesn't have anything else to do with the antenna, it will just stay in retracted state, indefinitely if it has to. But the user could at any time decide to deploy it, and the antenna will immediately leave the retracted state and move towards state the user wishes. There's nothing stopping it, it doesn't have to finish its tea first or anything.

So let's suppose the user wants it to track a target. The state immediately jumps from retracted to deploying, an intermediate state... and remains stuck there until that state is considered complete! No matter how impatient the user, it cannot move on to deployed state unless it got its condition fulfilled. In this case, the condition is for the deployment animation to finish. So the user cannot make it leave its current state at will. But once the animation has finished, the state *has* to change to Deployed, it cannot decide to just keep sitting in the intermediate state. When it's done, it's done.

But the user wanted the antenna to track a target, remember? So the state, now being in the stable deployed state, will immediately evolve further to the aligning state, where it will again remain until it gets the message that it is alligned with the target, at which point it will remain in the stable tracking state. A state it will remain in until it gets outside input to move elsewhere.

So, to summarize one more time: Stable states can change any time, but don't have to, intermediate states can only change if there is a fulfilled condition, but then they absolutely must.
There is *one* exception to this rule: Directly connected intermediate states can change into one another at any time. This sounds weird at first, but it's not that strange. Consider the two-way connection between the Deploying and the Retracting states above, both intermediate. As mentioned before, the user can change his mind and start retracting while the antenna is still deploying and vice versa. Or he can decide to track another target when the antenna has stoped tracking and is on its way to its origin, then it doesn't have to reset all the way before beginning to align with its new target. Instead, it can move from STOP_TRACKING immediately back to ALIGNING.

These three rules are essentially the whole logic behind StateMachine. So with that out of the way, how to use it in code?
First off, we need to add a StateMachine instance as a member of our class, and we need to initialise the machine with the possible states, and connect those states. In other words, we have to explain the map we drew above to it. The first we do in the header, the later we do in the constructor of our class.

Modify the private section of the header file to look like this:

```
private:
	IMS_ModuleFunctionData_Comm *data;
	StateMachine state;
```

Next, we load the state machine with all possible states. We do this with StateMachine::AddState(), in the constructor of our module function. But there's a catch: When writing the data class, we said that a valid module function only needs one of the three possible animations. Depending on which animations were defined in the config file, we don't necessarily need all the states. Since StateMachine receives the valid states at runtime, this isn't more bothersome than adding a few if-statements to the code. In the end, our state definitions look like this:

```
//check which actions are actually supported by the config
bool hasdeployment = data->deployanimname != "";
bool hasscanning = data->searchanimname != "";
bool hastracking = data->trackinganimname != "";


state.AddState(COMMSTATE_DEPLOYED, "deployed");

if (hasdeployment)
{
	state.AddState(COMMSTATE_RETRACTED, "retracted");
	state.AddState(COMMSTATE_DEPLOYING, "deploying", false);
	state.AddState(COMMSTATE_RETRACTING, "retracting", false);
}

if (hasscanning)
{
	state.AddState(COMMSTATE_SEARCHING, "scanning");
	state.AddState(COMMSTATE_STOP_SEARCHING, "resetting", false);
}

if (hastracking)
{
	state.AddState(COMMSTATE_ALIGNING, "aligning", false);
	state.AddState(COMMSTATE_TRACKING, "tracking");
	state.AddState(COMMSTATE_STOP_TRACKING, "resetting", false);
}
```

Let's have a closer look at AddState(): It takes three arguments, two of which are optional. The first and only mandatory is an integer identifying the state. As can be expected, you are not allowed to add the same state identifier twice, and as mentioned, it's best if you define these identifiers in an enumeration as we did here.
The second argument is a string describing the state in plain text. You can or cannot add a description to a state. Since this module function will need to communicate its state to the user, this is a very convienient way to store a description along with the identifier without having to resolve the identifier yourself every time. If noone ever needs to read the state as plain text, you don't have to pass this argument. The default is an empty string.
Last but not least, we have a boolean flag describing whether the state is stable or not. The default is true, so you only have to use this if you want to declare the state as an intermediate state.

So now that we have all the states added, we have to connect them. This takes a bit more code, since there are more connections than states, but it's still on a one-line-per-item basis. As before, we'll have to check which states are actually present (continued on next page):

```
if (hasdeployment)
{
	state.ConnectStateTo(COMMSTATE_RETRACTED, COMMSTATE_DEPLOYING);
	state.ConnectStateTo(COMMSTATE_DEPLOYING, COMMSTATE_DEPLOYED);
	state.ConnectStateTo(COMMSTATE_DEPLOYED, COMMSTATE_RETRACTING);
	state.ConnectStateTo(COMMSTATE_RETRACTING, COMMSTATE_RETRACTED);
	state.ConnectStateTo(COMMSTATE_RETRACTING, COMMSTATE_DEPLOYING);
	state.ConnectStateTo(COMMSTATE_DEPLOYING, COMMSTATE_RETRACTING);
}

if (hasscanning)
{
	state.ConnectStateTo(COMMSTATE_DEPLOYED, COMMSTATE_SEARCHING);
	state.ConnectStateTo(COMMSTATE_SEARCHING, COMMSTATE_STOP_SEARCHING);
	state.ConnectStateTo(COMMSTATE_STOP_SEARCHING, COMMSTATE_DEPLOYED);
```

```
}

if (hastracking)
{
	state.ConnectStateTo(COMMSTATE_DEPLOYED, COMMSTATE_ALIGNING);
	state.ConnectStateTo(COMMSTATE_ALIGNING, COMMSTATE_TRACKING);
	state.ConnectStateTo(COMMSTATE_TRACKING, COMMSTATE_ALIGNING);
	state.ConnectStateTo(COMMSTATE_TRACKING, COMMSTATE_STOP_TRACKING);
	state.ConnectStateTo(COMMSTATE_ALIGNING, COMMSTATE_STOP_TRACKING);
	state.ConnectStateTo(COMMSTATE_STOP_TRACKING, COMMSTATE_DEPLOYED);
	state.ConnectStateTo(COMMSTATE_STOP_TRACKING, COMMSTATE_ALIGNING);
}
```

This represents our drawn map to the point: We define a number of stable and intermediate states, connected together by unidirectional connections. As you can see, we define two separate connections for connections that go both ways. In case it isn't obvious, the direction of the arrow in the diagram goes from the first argument of ConnectStateTo() to the second.
We're almost ready to use the statemachine now... what it still needs to know is the initial state. If you try to use it without having defined the initial state, you will trigger an assert.

We do however have to wait a few moments with setting the initial state, because we don't know yet in what state the machine was saved. And before you ask, yes, it can be initialised only once. But we can put a default initialisation into PostLoad(), which will be executed after a scenario was loaded:

```
void IMS_ModuleFunction_Comm::PostLoad()
{
	if (!state.IsInitialised())
	{
		//setting the initial state of the statemachine, as it wasn't loaded from
		//scenario
		if (data->deployanimname != "")
		{
			state.SetInitialState(COMMSTATE_RETRACTED);
		}
		else
		{
			//if deployment is not supported, the antenna is
			//actually considered permanently deployed
			state.SetInitialState(COMMSTATE_DEPLOYED);
		}
	}
}
```

The first thing we check here is whether the state machine has already been initialised. This would be the case if a state was loaded from scenario, at least once we finish the code. Next we have two possibilites: Either the module function has a deployment animation, in which case the default initial state is retracted, or it doesn't have a deployment animation, in which case the default initial state is deployed. Because obviously we couldn't do anything with it if it was retracted without the possibility to deploy it.

Now, what have we done all this for? Pretty simple, really: Instead of constantly checking what state you are in and what states you are allowed to go to from there, you can leave it to the state machine. You tell it which state you want to reach, and it will look for the best path there itself. How exactly to do that, and how to controll state changes, is covered in the next chapter.

Controlling your state with external inputs and events

You need a means to determine what state your module function should be moving towards. In our example, The triggers for changing stable states will come from the outside, from a UI. But we don't have to bother with writing a menu for it just yet. What we have to do is give the module function the ability to receive external commands. Preferably, we do that by public functions. While we are intending to call those from the GUI, having public calls to commands in your module function offers more flexibility, in case something else should ever need to take command of them (for example an AI module, though I'm not planning on writing one... but I have made it a design paradigm to leave as many possibilities open as possible).
What commands does the comm module function actually need? Well, basically we have three different animations, but for the purpose of this tutorial we'll only look at one. The rest works according to the same principles, and you can look them up in the source code if you want to.
For now, let's go back to our header file and declare a public function to controll our deployment animation, which looks pretty simple:

```
	void CommandDeploy();
```

Let's implement CommandDeploy() to get an idea of what will be happening here. It is a rather short, unassuming function, but it's packed with tons of new stuff you have to know:

```
void IMS_ModuleFunction_Comm::CommandDeploy()
{
	if (data->deployanimname == "") return;

	if (state.GetState() != COMMSTATE_RETRACTED)
	{
		//if the state is not currently retracted, clicking the deployment box is
		//interpreted as an order to retract
		state.SetTargetState(COMMSTATE_RETRACTED);
	}
	else
	{
		//if it is retracted, we interpret it as an order to deploy
		state.SetTargetState(COMMSTATE_DEPLOYED);
	}
	//if the current state is stable, advance it.
	//if the current state is an Intermediate one, the state
	//will be advanced by an event at the proper time
	state.AdvanceStateSecure();
}
```

The first thing we do is to check if the module function instance actually has a deployment animation. If not, we just won't bother doing anything.

The next condition contains a new call, StateMachine::GetState(). This simply returns the identifier of the state the state machine is currently in. The overall logic of the command goes like this: If the antenna is currently retracted, the user will want to deploy it. If it is in *any other state*, we assume that the user wants to retract it. This results in two very simple conditionals that would be extremely complicated were it not for the state machine: We simply call StateMachine::SetTargetState() and tell it what state the state machine has to be moving towards.

And that's it! If the state isn't deployed, we don't even care about what the exact state is currently. We just tell the machine that we want the antenna to eventually end up retracted, and it already has all the data it needs to decide itself what the best course of action is to get it there.

The last thing we do, no matter which one of the two conditions apply, is call StateMachine::AdvanceStateSecure(). Do you remember what I explained about stable and intermiediate states? How stable states can't change without outside input, and intermediate states can't change until certain conditions are fulfilled? AdvanceStateSecure() is that outside input that kicks the machine into gear and out of the current state *if* it is in a stable state. But only if it is in a stable state, and if it has a valid target state. If the state is intermediate, it will not be advanced. As a result, you could pretty much call AdvanceStateSecure() whenever you want... it cannot throw the state machine out of synch.
The remaining two commands, CommandSearch() and CommandTrack(), operate practically the same. I'd suggest you take a look at them in the source code at this point, because I won't be going into them here. Don't look at SetTarget too much though. That one is somewhat different, since it involves an Orbiter input callback, which will be covered near the end of this guide.

By now you might be protesting that we didn't actually do anything. A command has been issued, and a state has changed, but the program didn't do a single thing. Isn't there supposed to be an animation going on by now?

Well, yes... But the beauty of a state machine is that you can centralise the execution of tasks. We don't want to evaluate the state and decide what to do in every single method where we change a state, that would kind of defeat the purpose. Instead, we'll check every frame if the state has changed, and *then* take action. Because that way, we only have to worry about the current state, not where the state came from or where it is going. Just what it changed to.
Checking for something every frame sounds an awful lot like a controll loop, and if you remember the interface, we already have a method hooked into orbiters controll loop. So we'll put all our state evaluation into PreStep(). For now, we'll just deal with the state changes that can come from our first command:

```
IMS_ModuleFunction_Base::PreStep(simdt, vessel);
{
	IMS_ModuleFunction_Base::PreStep(simdt, vessel);

	//check if the state has changed during the last frame,
	//and whether we reached the target state
	if (state.StateChanged())
	{
		switch (state.GetStateAndAdvance())
		{
		case COMMSTATE_DEPLOYING:
			addEvent(new StartAnimationEvent(data->deployanimname, 1.0));
			break;
		case COMMSTATE_RETRACTING:
			addEvent(new StartAnimationEvent(data->deployanimname, -1.0));
			break;
		}
	}
}
```

Again there's some new things here. Firts of all, we have to call the PreStep on the base class, which

handles sending events. Don't forgett this, or nothing will be happening in your module function!
Next we have a call to StateMachine::StateChanged(). This queries the state machine if the state has changed *since the last call to StateChanged().* This is important to keep in mind, because it implies two things: First, you absolutely have to call it every frame, or the reporting won't be reliable. Second, if you call it a second time in the same loop, it will simply return false (since the state won't have time to change in the meantime). So if you need this information anywhere else for some reason, put the result in a variable.
For our current purposes, and for any purposes I can think of for that matter, we just need it right here, right now.
The next call is to StateMachine::GetStateAndAdvance(). This is a shorthand, and the result is identical to doing this:

```
switch (state.GetState())
{
	//do stuff
}
state.AdvanceStateSecure();
```

In other words, it takes it off your hands to advance the state at the end of the method. Not much, but there you go. Just keep in mind that the state might not be the same anymore after you call GetStateAndAdvance, while calling GetState several times will always return the same result.

The cases of the switch statement should be self-explanatory: We check what state we just arrived at, and take appropriate action. The action we take, is to add an event. Let's go to the next chapter and see how that works exactly.

Working with events

In this chapter, you'll learn how to use the event engine built into IMS2. We won't go into any depth on how it works, just how to use it. If you want to know more about the architecture, read the description in EventHandler.h. We also don't cover how to register your event handler or setting up channels of communications, since that has already been taken care of by IMS_ModuleFunction_Base and the core, respectively.
One thing you should know, however, is that the events you send out from your module function will not leave the context of the module. In other words, you can only communicate with other module functions that are within the module containing the current instance of your module function. There are however certain events that will be routed to the vessel core by the module. It's not really up to the module function to decide whether an event is appropriate to send to the core, while the module is in a much better position to decide that. For clarities sake events that get rerouted are listed below, although we won't need them for this tutorial.

For now, events routed to the core from module functions are these:

- MassHasChangedEvent
- ConsumableAddedEvent

But what are events, anyways? Events are a means by which individual parts of a program can communicate among each other without knowing of each others existance, ergo they also don't depend on each others existance. In a dynamic aggregation model like IMS2, where you never know what parts will actually be around at runtime, this is a highly desirable feature. To show you what I mean, let's look at that line we left unexplained at the end of the last chapter:

```
addEvent(new StartAnimationEvent(data->deployanimname, 1.0));
```

EventHandler::addEvent() simply adds a new event to the handlers event cue. You will notice that what we're passing to it is actually a pointer to a newly created event, but you don't have to worry about deallocation. While it is generally true that the one allocating an object should also be the one destroying it, this way of doing it simply offered the simplest synthax without requiring a maintenance-heavy factory function. So you don't have to manage the created object (in fact, when you create the event like this, right in the function call, you can't retain a pointer to it anyways). The eventhandler will take full ownership of the event and will dealocate it once it is no longer needed. This makes an event pretty much a fire-and-forget affair: You add it, and you're done with it!

What we add here is, of course, a very specific event. An event that is intended to start an animation. The argument we pass it is the name of the deployment animation we read in from the config file, followed by a speed at which the animation should play out. Usually, you'll just want to pass 1.0 (or -1.0 to reverse an animation) for this. We won't cover the intricacies of how animations work here, that is outside the scope.
But wait, you might say, we don't *have* an animation! We can't possibly start an animation that doesn't exist, now can we? Technically that would be correct... but remember what I wrote above? That communication by events doesn't really depend on the existance of the other components? We can send that event without an animation existing. The event will wander around fruitlessly and finally be destroyed without having actually achieved anything, but your program won't crash. But of course you are right in that the purpose here should be that an animation actually starts, so an animation we need. But not here!

Animations are actually another module function. That is, a part of IMS that exists on the very same level as the class we are writing currently. Whether or not the animation exists therefore is not your responsibility. It is the responsibility of the one writing the config file that uses your module function, because there he will have to define the animations independantly from the definition of the antenna. Take a look at BCM101_Antenna.cfg to see what that looks like.

For our purposes here, we'll just assume the animation has been correctly defined. If it wasn't, the program won't crash, and overall won't even be affected... just the specific instance of your module function that needs that non-existing animation won't work right (there will be a log message from the animation module function that somebody tried to start a non-existing animation, so the process isn't entirely without feedback).

So what happens once we added the event? It will be sent to all module functions inside the current module (in fact, your own module function who sent it might receive it too, but that's not a problem. We just won't look at it). The animation module function will process it, set the animation in motion, and will send its own event back that will tell us whether or not it was succesfull, which again will be sent to all module functions inside this module. What is currently more interesting to us, however, is that the animation will also send out an event when it finishes.

We'll ambush that event when it comes through processEvent()! (continued on next page)

```
bool IMS_ModuleFunction_Comm::ProcessEvent(Event_Base *e)
{
	if (*e == ANIMATIONFINISHEDEVENT)
	{
		AnimationFinishedEvent *anim = (AnimationFinishedEvent*)e;
		string id = anim->GetAnimationId();
		if (id == data->deployanimname)
		{
			state.AdvanceState();
		}
	}
	return false;
}
```

All events that are sent out from the module to its module functions will pass through this method. All you have to do is to see if any of them interest you, and act accordingly.

In this case, we first make a quick check if the event is an AnimationFinishedEvent by comparing it to its general identifier. Every event has such an identifier, which is great for quick recognition. The convention here is that the identifier has the same name as the event class, but written in all caps. They are declared in EventTypes.h if you want an overview.

If the event coming through is indeed an AnimationFinishedEvent, we have to check if it's actually one we're waiting for, since there might be a lot of animations going on, and a lot of AnimationFinishedEvents flying around. Convieniently, all animation related events carry the string identifier of the concerned animation with them (we already used it when sending the StartAnimationEvent, remember?). But to get at it, we'll have to cast the Event_Base we received into an AnimationFinishedEvent first. Since we have already checked if the event is an AnimationFinishedEvent, this is perfectly safe.
Once we have our event in the proper type, we retrieve the id of the animation that sent it, and if it is the same as the name of our deployment animation, we know that the deployment animation has finished! We don't actually know in which direction, but thanks to our state machine, we don't need to. All we need to do is to advance the state.
You might notice that we're not doing that with AdvanceStateSecure() this time, but instead simply with AdvanceState(). The reason for this is that AdvanceStateSecure() wouldn't actually do anything here.

Remember what an intermediate state is? It's a state than can only change if a precondition has been met. If the deployment animation has just finished, we know that a moment before it had to be running, from which we can safely assume that our module function is in either DEPLOYING or RETRACTING state (unless we royally screwed up something somewhere). Both of these are intermediate states. AdvanceStateSecure(), as mentioned, is specifically designed to not touch these. AdvanceState() isn't, and is therefore inherently dangerous to use. It will advance the state of the statemachine no matter what, except if the statemachine has nowhere to go to (i.e. has reached its target state and hasn't had a new one set yet).
But AdvanceState() is the only thing that can force an intermediate state to change. And since we just asserted that the running deployment animation has finished, which is the precondition for the DEPLOYING or RETRACTING state to evolve into either DEPLOED or RETRACTED, we have to give the statemachine that nudge to let it know that we're serious: "Yes, it's ok, the condition has been met, get on with it already!" Needless to say, you should use that call carefully. Usually it should only be triggered by events, or maybe timers if you have to measure something internally (allthough the best way to set a timer in IMS2 is, you might have guessed it, by using an event).

Let's catch a few more events here to make the deployment functionality complete:

```
	else if (*e == ANIMATIONFAILEDEVENT)
	{
		AnimationFailedEvent *anim = (AnimationFailedEvent*)e;
		string id = anim->GetAnimationId();
		int curstate = state.GetState();

		if (id == data->deployanimname)
		{
			//we'll have to know what direction the animation was supposed to move
			if (anim->GetSpeed() < 0 && curstate == COMMSTATE_RETRACTING)
			{
				state.SetTargetState(COMMSTATE_DEPLOYED);
			}
			else if (anim->GetSpeed() > 0 && curstate == COMMSTATE_DEPLOYING)
			{
				state.SetTargetState(COMMSTATE_RETRACTED);
			}
		}
	}
	else if (*e == ANIMATIONSTARTEDEVENT)
	{
		AnimationStartedEvent *ev = (AnimationStartedEvent*)e;
		string animid = ev->GetAnimationId();
		int curstate = state.GetState();
		if (animid == data->deployanimname)
		{
			if (ev->GetSpeed() > 0 && curstate == COMMSTATE_RETRACTED)
			{
				state.SetTargetState(COMMSTATE_DEPLOYED);
				state.AdvanceStateSecure();
			}
			else if (ev->GetSpeed() < 0 && curstate == COMMSTATE_DEPLOYED)
			{
				state.SetTargetState(COMMSTATE_RETRACTED);
				state.AdvanceStateSecure();
			}
		}
	}
```

We listen for both failure and sucess events of our deployment animation. As before, when we catch an event of the right type, we cast it to the proper type and see if the id matches our animation.
The reason why we listen for the ANIMATIONFAILEDEVENT is pretty obvious: Our state machine has a target state set, and evidently running through deployment or retraction lies on its path. But if the animation failed to start, it will never finish. In other words, the antenna will be caught in its current intermediate state and never be able to leave it, no matter what we do, because it can only leave it if an animation has finished.

So now that we know that the animation didn't start, we use an additional call of the AnimationFailedEvent: We get the speed at which the animation was supposed to launch, because this also gives us the direction. A negative value indicates that the animation was supposed to start in reverse. So if it was started in reverse, it failed to retract, and if it was started normally, it failed to deploy. The only thing we do is to set a new target state for the state machine: If it failed to deploy, the state should go back to retracted, and if it failed to retract, the state should go back to deployed. This action relies on us having properly maped our states. If there wouldn't be a direct

connection back and forth between DEPLOYING and RETRACTING state, this would actually not be helpful, because the state machine would have no way of getting back to its state without waiting for the animation to finish, which of course it won't in this case.
The reason why an animation fails, by the way, is usually because a precondition isn't fulfilled. If you take a look at the working BCM101_Antenna in orbiter, you will notice that there the small antennas are covered by the retracted dishes. I.e. if they were to deploy with the dishes folded up, they would clip through them. For such situations it is possible to declare a dependency for an animation in the config file, that certain animations in this module must have certain states or the animation can't be triggered. Instead, it will send the AnimationFailedEvent.

Next, we catch the AnimationStartedEvent, and the reason for this is a bit less obvious. We already check if the animation failed, so if it didn't, it must have sucessfully started, right? While that statement is perfectly correct, we're assuming that an animation can only ever be triggered by one module function instance. But that is not necessarily the case! For example, if you look at the original deployment animation of BCM101_Antenna with SC3, or the deployment animation it had in IMS1, the whole thing unfolds itself on a single command. The deployment animations which are seperate in our current config were merged together into one animation.
The keyphrase here is "in our config". What happens if someone writes another config, and decides to keep the old mode of deplyoment, and all antennas in the module use the same deployment animation? In that case, if another modulefunction instance deploys, visually all others are deploying too. But since the command was only explicitly issued to one instance over CommandDeploy(), the rest doesn't actually know that they are deploying. They still think they are retracted.
But if we catch the AnimationStartedEvent and see that it came from "our" animation, we can take action and adjust our state, even if this instance isn't the one that actually got the command. In other words, our state machine will now transition to deploying or retracting state, even though CommandDeploy() was never called in this instance. This gives a lot of additional flexibility to config writers.

State and Events recap

Now that we have looked at the state machine, the event handling and how they interact, let's sum this up by a short summary of what goes on in what order in that code we wrote.
After defining all the states and mapping them out, pretty much nothing happens. The antenna is in RETRACTED state and doesn't move.
The whole rube goldberg machine starts ticking when something calls CommandDeploy(). Within the function itself, DEPLOYED is set as the target state. The state machine will search the best way there, and will conclude that its path consists of RETRACTED (the current state) and DEPLOYING (an intermediate state), which will bring it to DEPLOYED.
Also in the CommandDeploy() function we tell the state machine to advance the state if it is safe to do so. The machine will conclude that it is currently in RETRACTED state, which is stable, and that it is therefore perfectly fine to move on. It evolves into DEPLOYING state.

When we come through PostStep() the next time, we notice that the state has changed during the last frame. We check what the new state is, and on learning that it is DEPLOYING, we send an event to start the deployment animation.
If everything went right, the AnimationStartedEvent will fire in processEvents, but since this is the module function instance that started the animation, there won't be any effects from that. If it would

have been another instance, this one would now set its target state to DEPLOYED and pretty much go through all the steps described above.

For a while, nothing will happen. The state doesn't change, so we pretty much skip over PostStep(), and no relevant events are coming in, so the state remains unchanged some more. Until the AnimationFinishedEvent hits. As soon as we process that event we force the state machine to move on a step. The next state on the statemachines chosen path is DEPLOYED, so it will evolve to that. DEPLOYED is a stable state, so it will immediately attempt to evolve the state again... except it then notices that this is the target state, and it settles down until another target state is set and the whole game begins anew.
Of course things will get more complicated when the other animations and commands start to chime in, but the flow remains exactly the same, so if you understood this, you should have no trouble building on it.

Saving and loading the state of your module function to scenario

Needless to say, a module function should be able to resume where it left off when resuming a scenario. This isn't such a big deal. We already overloaded the necessary functions and created stubs for them , so all that's left is the code. But first we have to decide what we want to save. If we have a state machine, we pretty much have to save the state the machine is currently in, and the target state if there is one. For the complete version of IMS_ModuleFunction_Comm, we also have to save the current target for the tracking animation, but because we haven't bothered with that so far, we'll leave it out in this tutorial. As always, consult the complete code in the IMS2 repository if you're interested in it.
Appart from that, there's really nothing else to save here. The animations will save their own state, and will resume by themselves if they were running while the scenario was saved. Since all actions in our module function depend on either a state change or an event received by an animation, everything will work just fine if we just remember the position and target of the state machine.
The resulting code looks like this:

```
void IMS_ModuleFunction_Comm::SaveState(FILEHANDLE scn)
{
	int curstate, tgtstate;
	curstate = state.GetState();
	tgtstate = state.GetTargetState();
	oapiWriteScenario_int(scn, "STATE", curstate);
	if (tgtstate != curstate)
	{
		oapiWriteScenario_int(scn, "TGTSTATE", state.GetTargetState());
	}
}
```

Note that we only save the target state if it is different from the current state. If it isn't we don't care about it.

And that's that already. The function will be called whenever orbiter saves state, and the appropriate scenario block in which your data will be saved has already been opened by the mdoule, and will be closed by it. Nothing to see here, move along!

Loading the state works very similar to the data class we wrote. There's the processScenarioLine() method which will receive your saved data line by line. Only difference here is that it will receive the line as a whole string, since it doesn't make much sense to impose a strict format for scenario

entries. Depending on how you format your entires, you'll need to unpack that data differently, of course. For this example, I used the pretty standard approach that is also used in the orbiter examples, and it'll probably be best suited for most tasks.

```
void IMS_ModuleFunction_Comm::processScenarioLine(string line)
{
	if (line.substr(0, 5) == "STATE")
	{
		int curstate = atoi(line.substr(6).data());
		state.SetInitialState(curstate);
	}
	else if (line.substr(0, 8) == "TGTSTATE")
	{
		int tgtstate = atoi(line.substr(9).data());
		if (tgtstate != state.GetState())
		{
			state.SetTargetState(tgtstate);
		}
	}
}
```

As you can see, we immiediately initialise the state machine if it saved the state. This happens before PostLoad() is called, so once the code arrives there, it will see that the state machine is already initialised and won't do so any more. The machine ends up in the same state as it left, with the same target state, and since the animations do the same by themselves, there will be no discontinuity. If an animation was running when the scenario was saved, it will be running now and send its event when it finishes as if nothing had happened in the meantime.

Creating a menu

We're almost done here, except there's a pretty big, and technically unrelated chunk left. It seems prudent however to also explain how to write a menu so a user can controll your module function. This will require us to create a new class, I'm afraid.
For the example, we create a new class called "GUI_ModuleFunction_Comm. It inherits, unsurprisingly, from GUI_ModuleFunction_Base, which is required so your menu can be integrated into the module's controll page without any hassle. We'll need to make a few adjustments to the module function class to enable this, but first let's explore what we're dealing with here. The GUI isn't exactly a simple beast, all things considered. A GUI toolkit never is, although for some irrational reason we always think that it really should be.

Let's take a look at the minimal draft of what our header file should look like (cont. on next page):

```
class GUI_ModuleFunction_Comm :
	public GUI_ModuleFunction_Base
{
public:
	GUI_ModuleFunction_Comm(IMS_ModuleFunction_Comm *_function, GUIplugin *gui, bool
				hasdeployment, bool hasscanning, bool hastracking);
	~GUI_ModuleFunction_Comm();

	virtual int ProcessChildren(GUI_MOUSE_EVENT _event, int _x, int _y);
```

```
protected:

	IMS_ModuleFunction_Comm *function;

	bool updatenextframe = false;

	virtual bool updateMe();

private:
	int getHeight(bool hasdeployment, bool hasscanning, bool hastracking);

};
```

This is just about the bare minimum you can get away with. As always, I'll shortly explain what is what before we pile more on top of it.

First there's a constructor, that receives a pointer to the module function instance it is representing, then a weird thing called a GUIplugin which we'll look at later (simplified, it's the parent the menu will be drawn on), and then some boolean flags which are specific to the purposes of our example module function. They will tell the gui for which possible options it has to provide controls, depending on which animations were declared in the config file.

Next is a thouroughly ordinary destructor, followed by an overload from the very bottom of the tree of GUI elements, that we'll use to actually process the user input.

There's also a protected overload updateMe(), which will enable us to request a redraw from Orbiter, so we can actually change visuals even when the element wasn't clicked (calling it "animations" seems overly grand. Technically you could implement complex 2-d animations for your module function menu this way, but technically you could also re-implement Tetris or Nethack inside it, neither of which I would consider a very good idea).

There's two members, one of which we'll use in combination with the above method, the other is our pointer to the module function instance, because we'll have to talk to it if we want to get things moving.

Finally there's the private function getHeight(). This may or may not be necessary. It returns the height of the menu in pixel. We'll look at it when we start implementing things.

We'll go right on to add more stuff to this class, but first let's stop and understand what it is we actually have here. If you're on your toes, you will have noticed that I talked about method overloads from GUI_element before. Indeed, our GUI class inherits GUI_ModuleFunction_Base, which inherits GUI_Page, which in turn inherits GUI_BaseElement. It helps to keep that in mind, but not that much if I don't explain what a GUI element is, I guess.

In short, a GUI element is anything drawn in any menu (except for text, though text often is wrapped by GUI elements). A button is a GUI element. A checkbox is too. A list box, a scrollbar, even the very background they are drawn on, are all GUI elements. They form a big happy tree that describes all controls, their positions and looks, their text, even the very menu structure.
In short, a GUI is nothing more than an amalgamation of elements with different abilities, but the same purpose: To draw something to the screen, and to give a shout if they are clicked on by the mouse. That's the basic functionality.

So what you are effectively writing now is a new GUI element. Just another branch in the tree. The difference is that you are not writing a generalised element with the intent to be used as controls by other menus. Instead, you are writing a highly specialised GUI element that is intended to only controll this one module function. The purpose, though, remains the same: To draw to the screen and to process the input of its children.

Second up the inheritance tree of the GUI framework is GUI_page. GUI_page is a GUI element who's major function is to group children. While really any GUI element could do that, the page is the one specialised for the task, and is therefore always the go-to point when implementing a complete menu. All specific menus currently present in IMS2 are derived from the page, be it the main root menu (a page containing a bunch of buttons) or the assembly- and disassembly menus (a page containing a bunch of buttons and listboxes).

All these elements, including the menus, have several things in common:
They have a rect, defining their size and position *relative to their parent.*
They have a style that defines their appearance.
They have a parent, a GUI element that contains them. This does not apply to root elements that are directly linked into the panel, but at the level we're working on here we're a good way removed from those.

So, obviously, our menu has to contain a bunch of things too if we want to control anything. In the case of our example, I've decided for checkboxes to trigger the individual animations, because it seemed to me that they could best represent the state complexity of the thing. So to control the three animations, we'll need 3 checkboxes. We also need a label to describe the state the module function is currently in, a label to display the target of the tracking animation, as well as a button to set that target. The first thing we do to get them is to add pointers to them as members of our menu. I've put them in the protected scope, just in case somebody will be using the class for a more specific implementation of an antenna:

```
        GUI_CheckBox *deploybox = NULL;
        GUI_CheckBox *searchbox = NULL;
        GUI_CheckBox *trackbox = NULL;
        GUI_LabelValuePair *statedescription;
        GUI_LabelValuePair *targetdescription = NULL;
        GUI_DynamicButton *settargetbtn = NULL;
```

We'll need these pointers to talk to and identify our controlls, so it's about time to look at that constructor and see how to add them to our menu. We'll look a bit at the synthax, what is what, as well as the used elements in a bit more detail (continued on the next two pages):

```
GUI_ModuleFunction_Comm::GUI_ModuleFunction_Comm(IMS_ModuleFunction_Comm         *_function,
GUIplugin *gui, bool hasdeployment, bool hasscanning, bool hastracking)
       : GUI_ModuleFunction_Base(getHeight(hasdeployment, hasscanning, hastracking), gui,
              gui->GetStyle(STYLE_BUTTON)), function(_function)
{
       //create the required controls
       gui->CreateLabel(function->GetData()->GetName(), _R(10, 10, width - 10, 35), id);

       //the statedescription will always be there
       statedescription = gui->CreateLabelValuePair("Status:", "", _R(120, 40, 400, 65), id,
                                  -1, STYLE_DEFAULT, GUI_SMALL_ERROR_FONT);

       int curypos = 40;
       if (hasdeployment)
       {
              deploybox = gui->CreateCheckBox("deploy", _R(15, curypos, 120, curypos + 25),
                                           id);
              curypos += 30;
       }

       if (hasscanning)
       {
              searchbox = gui->CreateCheckBox("scan", _R(15, curypos, 120, curypos + 25),
                                           id);
              curypos += 30;
       }

       if (hastracking)
       {
              //now things are getting a bit more complicated, because tracking controll
              //needs 3 gui elements to work,
              //resulting in three possible layouts depending on what other controls are
              //supported

              trackbox = gui->CreateCheckBox("track", _R(15, curypos, 120, curypos + 25),
                                           id);

              if (curypos == 40)
              {
                     //tracking is all there is. draw the button and the indicator on the
                     //line below, side by side
                     //the gui will look like this:
                     //trackbox          statedescription
                     //targetbtn         targetdescription
                     settargetbtn = gui->CreateDynamicButton("set target", _R(15, 70, 95,
                                         95), id);
                     targetdescription = gui->CreateLabelValuePair("Target:", "None",
                                         _R(120, 70, 400, 95), id, -1, STYLE_DEFAULT,
                                         GUI_SMALL_ERROR_FONT);
              }
              else if (curypos == 70)
              {
                     //the upper line is taken by either the deploybox or the searchbox. set
                     //the gui up like this:
                     //whateverbox       statedescription
                     //trackbox
                     //targetbtn                 targetdescription
                     settargetbtn = gui->CreateDynamicButton("set target", _R(120, 100, 200,
                                                  125), id);
```

```
                targetdescription = gui->CreateLabelValuePair("Target:", "None",
                                    _R(120, 100, 400, 125), id, -1, STYLE_DEFAULT,
                                    GUI_SMALL_ERROR_FONT);
        }
        else
        {
                //all operations are supported, set the GUI up like this:
                //deploybox             statedescription
                //searchbox             targetdescription
                //trackbox              targetbtn
                targetdescription = gui->CreateLabelValuePair("Target:", "None",
                                    _R(120, 70, 400, 95), id, -1, STYLE_DEFAULT,
                                    GUI_SMALL_ERROR_FONT);
                settargetbtn = gui->CreateDynamicButton("set target", _R(120, 100, 200,
                                                        125), id);
        }
    }
}
```

This seems overly complicated, and in a way it is, because it's a dynamic layout. I.e. different elements are created at different positions depending on what controlls are needed, which depends on which animations the author of the config file defined. I'll therefore keep the explanations to what's really new and don't go through the entire code.
And the first thing that might need some explanation is the call to the constructor of the base class. Specifically, you might notice that I'm calling the method getHeight() inside the constructor call. This is because, as mentioned, the size of this element is only known at runtime. If the menu you're writing for your module function always has the same height, you can just pass a fixed value here, no problem. But in this case, the height needs to be calculated, and we can't do it afterwards (GUI elements are not resizable, unfortunately). Calling the base constructor will already have allocated the video memory and drawn its background into it, so we need to know the height now, before this whole thing actually exists.

We also pass a default style for our element. In this case the button style, because it has smaller corner radii than the default window style. But these identifiers are very prone to change in the future, so this kind of thing isn't that important yet.

In the first executed line inside the constructor, we create a label that will display the name of the module function from the config file as a title. Let me just repeat that line so you don't have to flip pages during the following explanations:

```
gui->CreateLabel(function->GetData()->GetName(), _R(10, 10, width - 10, 35), id);
```

First, let's determine what is actually called here: gui is a GUIplugin instance, in this case it's actually a member of the module containing the module function containing this menu (confused yet?). We'll see more on that when we integrate the whole menu into our module function class.
I won't go into too much depth about GUIplugin and GUImanager. All you need to know for this task is that a GUIplugin is a self-contained and self-managed GUI tree. Self-contained means it can be pluged into any existing GUI tree without it really becoming a part of that tree, and thus can just as easily be removed again. Self-managed means that it will handle creation, storage and destruction of all its children.
You should, in fact, never instantiate a GUI element on your own. Always use constructors of a GUIentity (the class the plugin derives from), otherwise you'll have to bother with some secondary operations to get your element properly registered and placed in the tree. In case you're wondering where our menu as a whole is registered in that tree, that's done by our base class.

So what we're doing at the beginning of that line is to simply call a factory function on the module's GUI plugin that will result in us getting a label. A label is just a dumb static text in a certain font and style. Because this is a heading, we're using the defaults for those, so you don't see them in this line.
What we do see is that CreateLabel() gets passed the name stored in our module functions' data instance (which was loaded by IMS_ModuleFunctionData_Base, so we didn't have to do it ourselves), followed by a rectangle that defines the elements position inside the parent. In this case it goes from pixel 10/10 in the upper-left to 10 pixels left of the right edge and down to pixel 35. If you did the arithmetic in your head, you now know that the element is 25 pixels high, and has an unknown width depending on how wide the GUI_ModuleFunction_Comm element actually is. How wide is that exactly? Well, it's inherited by the parent element. I can't tell you quite exactly either, since that width also depends on the margin of the style of the parent. Just know that that you don't have to bother about the exact width as long as you define your rectangles horizontal boundaries parent-relative as is shown here.

Why did we put it over almost the whole width of our menu? Well, it's a heading, and the default style and font settings of the GUI_Label class means the text will be written centered both vertically and horizontally on the element, so this is just an easy way to get the element centered and make sure it'll have enough space.
Finally, we pass it the id of our own element as parent.

The result of this whole operation will be the creation of a label that is already inserted into the GUI-tree as a child of our menu. We could retain the pointer to the created element, it is returned by the call. But in this case, we don't need it. That label is only there to display text. It cannot be clicked, the name cannot be changed, and it is already integrated into the memory managment of the plugin. We can forgett about it, it will just be there, doing its job.

Let's look at the next line:

```
      statedescription = gui->CreateLabelValuePair("Status:", "", _R(120, 40, 400, 65), id,
                                 -1, STYLE_DEFAULT, GUI_SMALL_ERROR_FONT);
```

Here we create a "LabelValuePair", which is like a label, except it will display some dynamic text to the right of the label. The label itself can't be changed (as is also the case for GUI_Label), but the "value" text can be changed at a whim. So this is a nice element if you need to display something along the lines of "Altitude: 1000m", where "Altitude:" would be the label that never changes, and "1000m" behind it could change to show whatever the current altitude is. It's a very convienient element for any kind of labeled listing.
In this case, the resulting element will show the current state of the module function, next to the label "Status: ". I won't go into the rectangle and the parent id anymore, but next is a -1, which tells the plugin that it should generate a unique identifier for this element on its own. We didn't see this in the previous call because it's actually the default value, but since we pass some non-default ones down the line this time, we have to put it here. And it's helpful that you understand what it's doing.
Every GUI element is identified inside its GUIentity (such as a plugin) by a unique integer value. The value has to be unique, otherwise there would be ambiguity about which element was actually clicked.
Common sense would be to have an enum with these identifiers, so you could more easily identify them by yourself, and indeed I do this for all fixed elements in the vessel's GUI. The problem is, what you're doing here isn't fixed. The whole GUI you're writing here might be there, or it might not, depending on whether the users vessel has a module that uses your module function. There might be one onstance of it in a modules plugin, or there might be 10. If those 10 instances all have

children with the same identifiers, you're screwed! So it's important that whenever you work on a module function GUI, you stick to dynamic id generation.
Anyways, there's something else here that warrants attention. For the first time we see a style and a font passed to a factory function.
The identifiers for these are declared in GUI_Common.h if you want to see what's available. I can pretty much guarantee that these are going to see massive changes though, so it's not too imnportant for now. Important is that you understand why you would pass a non-default font and style to an element.
In this case it's pretty simple: The default style and font of a LabelValuePair is for a heading, which this won't be, so we pass it the "default style" (that is, the style that defines the default skin style of the entire GUI... not the default style of a LabelValuePair). For the font, however, we do not just pass it the default small font. We pass it the error font, which is distinguished by having a red highlight color. In other words, if text gets highlighted in this font, it appears in red, which is usually associated with something bad. We'll need this for the situation when an antenna isn't able to track its target, in which case we can just tell the element to switch to highlighted text, and voila! Instant error message, without lifting a finger.
Also, we are keeping a pointer to this one. After all, we'll need to change its text every once in a while.

The rest of the code basically does the same thing, just with different positions in different cases. You might find it helpfull if you need a dynamic layout yourself, but there's not much new in it once you realise what's going on in these two lines. Two other element types enter the picture, checkboxes and a button, both of which should be rather obvious in what they do. If you ever need to know what exactly an element does, you should refer to the appropriate element class in the documentation, which also contains explanations on all available factory functions in a GUI entity.

If we'd already integrated this thing into our module function class, we'd already be able to see it on the screen at this time, and it would already be a bit functional. At least the checkboxes would check and uncheck if they are clicked. But of course all this isn't of much use if we can't tell it what to do when things are clicked.

Handling mouse events and redraws

Remember how we overloaded that ProcessChildren() function in the header? The base implementation of this method does nothing else than to send an event to all its children, and return the childs id if one was clicked, or -1 otherwise. Once the return value reaches the lowest level, it will trigger a redraw of the panel area if a child was clicked, so visual changes like a checkbox being checked or a listbox being scrolled take immediate effect.
By overriding this method, we get a neat point where we can look at that return value and decide to do stuff if necessary. The code for this is comfortably straightforward, but let's make sure we understand everything.

```
int GUI_ModuleFunction_Comm::ProcessChildren(GUI_MOUSE_EVENT _event, int _x, int _y)
{
	//send the event to the children, so they can check if they were clicked
	int eventId = GUI_BaseElement::ProcessChildren(_event, _x, _y);

	//no children of this element have been clicked
	if (eventId == -1) return -1;

	if (deploybox && eventId == deploybox->GetId())
	{
		function->CommandDeploy();
	}
	else if (searchbox && eventId == searchbox->GetId())
	{
		function->CommandSearch();
	}
	else if (trackbox && eventId == trackbox->GetId())
	{
		function->CommandTrack();
	}
	else if(settargetbtn && eventId == settargetbtn->GetId())
	{
		function->CommandSetTarget();
	}

	return eventId;
}
```

The first thing we do is actually call the base method. This will send the mouse event to all the children, and return us the id of the clicked element if one was clicked. If the return is -1, nothing was clicked on, so we can return that right away without giving it any further thought.

But if something was clicked, we need to find out what it was. For that we can simply query the ids of our child elements and compare them to the id we got back from the base method. There's only four elements that really interest us, and what they do is again very simple since we did such a good job when writing the module function. They simply call the appropriate command on it.
Just for repetition: We did not actually implement all of these during this tutorial. In fact, we only looked at CommandDeploy(), but the rest are pretty much the same thing in different colors. The only exception here is CommandSetTarget, but we'll look at this later on, since it deals with another topic (text input via keyboard, if you must know already).

That's really all there is to this. If the player clicks on deploybox, the CommandDeploy() on the module function instance is called, and you already know what happens from there.

But there is one missing piece here, which is "animation" (I still refuse to apply that term without quotation marks to what I'm going to do here).

If you ever took the actual BCM101_Antenna module for a spin in orbiter (which I hope you did), you will have noticed that actions that are either pending or in transition have their checkboxes blinking. Keep in mind that every change visible to your eye while playing required Orbiter itself to redraw the respective panel area, and that the only time it can do that until now is if a valid id was returned in ProcessChildren... in other words, we can't tell orbiter to redraw if the user refuses to click on anything. Since that's a rather depressing perspective (imagine a nice text pop-up saying "please continue clicking every second to see what is happening"), there's of course another way to do it. Enter the updateMe() method.

This method is again kind of a controll loop – it gets called every frame. If it returns true, the *entire* panel area will be redrawn. However, it is important that you understand that panel redraws are *expensive*, even with all the essential visuals pre-rendered, and drawing lots of text is ridculously so. So if you just had the bright idea to overload this method and return true without significant reason just to make your code less complicated, you either forget the idea again or I'll hit you on the head with a stone weighting a kilogramm for every frame per second you just sacrificed.

With that out of the way, let's learn how to do it properly. For that purpose I have added a short enumeration to the header which we'll use to control the states of our blinkenlights... blinkenboxes?

```
enum CHECKBOXSTATE
{
	OFF,
	ON,
	BLINKING
};
```

We also need to add the following members to the protected scope in the header, to store the current state of our checkboxes:

```
	CHECKBOXSTATE deployboxstate = OFF;
	CHECKBOXSTATE searchboxstate = OFF;
	CHECKBOXSTATE trackboxstate = OFF;
```

And the method itself, finally, looks like this:

```
bool GUI_ModuleFunction_Comm::updateMe()
{
	bool needsupdate = false;

	if (updatenextframe)
	{
		//the module has reached a stable state, update the gui visuals immediately
		updatenextframe = false;
		needsupdate = true;
	}

	//if any of the checkboxes are blinking, update them every second
	if (GUImanager::CheckSynchronisedInterval(1000))
	{
		if (deployboxstate == BLINKING)
		{
			deploybox->ToggleChecked();
			needsupdate = true;
		}
		if (searchboxstate == BLINKING)
		{
			searchbox->ToggleChecked();
			needsupdate = true;
		}
		if (trackboxstate == BLINKING)
		{
			trackbox->ToggleChecked();
			needsupdate = true;
		}
	}
	return needsupdate;
}
```

First of, we check if any parts of the code requested a redraw (we'll see those parts in a minute), and if yes, we note that we'll need a redraw no matter what happens next, but we don't return yet. We want to get our boxes blinking, after all.
So the next thing we do is to look whether the blinking interval has elapsed (in this case 1000 miliseconds). You might notice that we use some external function provided by GUImanager to do this, and again it's important you understand what's going on here.
The static method we call here doesn't as much measure an interval, as it will tell if an interval has recured measured against a fixed point in time. To understand why this is important, recall my warning about making too many redraws from a few paragraphs ago.
We have three checkboxes here... what would happen if they all were blinking on and off every second, but according to an interval that started when they were clicked? That's right, we get 3 panel area redraws per second. But it gets worse... Take the BCM101_Antenna module as an example. The module control page of that thing has a glorious 13 checkboxes, which potentially could all be blinking at the same time. And the whole panel area has to be redrawn every time one of them blinks on or off. 13 redraws per second for nothing, except for a lightshow that is more distracting than helpfull.
Using CheckSynchronisedInterval() ensures that all these will blink on and off in the exact same frame... reducing the number of redraws to exactly 1 per second, no matter how many of them are there. In fact, no matter how many of them are anywhere in the entire simulation. Every element redrawing at 1 second intervals will blink on and of in perfect synchronicity.

But of course they're not blinking just because we're redrawing the panel. That's why we can see that they are blinking. But they still have to do the blinking. For that purpose we do nothing more than just toggle the respective box's checked state, a method the checkbox convieniently provides.
So once the state of a checkbox is set to BLINKING, they will now start turning on and off every second.
But we don't change their state to blinking yet, so we're not done. The way we do this, however, is so straightforward that it won't require much explanation. We simply add a few public calls so our containing module function can change the state of the checkboxes when appropriate. I'll just show one of them here, because the others look exactly the same (if you want to go and look them up, they're called "SetSearchBoxState" and "SetTrackBoxState" respectively) :

```
void GUI_ModuleFunction_Comm::SetDeployBoxState(CHECKBOXSTATE newstate)
{
	if (newstate != deployboxstate)
	{
		//if a checkbox is set to on or off, this means that
		//the module has reached a stable state. We have to redraw the gui visuals
		//to reflect that.
		if (newstate != BLINKING)
		{
			updatenextframe = true;
		}

		deployboxstate = newstate;
		switch (deployboxstate)
		{
		case OFF:
			deploybox->SetChecked(false);
			break;
		case ON:
			deploybox->SetChecked(true);
			break;
		}
	}
}
```

There's not much to explain here: First we check if the state actually changed, and if it did (and didn't change to BLINKING) we set the flag for updateMe() to tell orbiter to redraw the panel area on the next step, so changes will become apparent immediately. Then we check what the new state is and set the checked status of the checkbox accordingly, and we're done.

But blinking checkboxes aren't the only thing that can change on the display. We're also doing a text output describing the current state of the module function. This is even simpler:

```
void GUI_ModuleFunction_Comm::SetStateDescription(string state, bool hilighted)
{
	if (state != statedescription->GetValue())
	{
		statedescription->SetValue(state, hilighted);
		updatenextframe = true;
	}
}
```

Nothing to see here, move al... wait, what's this hilighted?
Remember how we set a non-default font for the statedescription label? If we tell it to print its value highlighted, the text will have another color. That's exactly what we're doing here. If the module function thinks something is wrong, it can pass true for hilighted, and the statedescription element will print its value in the hilighted font style, in this case red.

Embedding your menu into your module function and making the two talk to each other

We now have a menu that would be largely functional, if only it would show up on the IMS2 panel in the simulation! It currently doesn't show up because our module function doesn't tell the module that it has a menu, so let's learn how to do that.
The first thing we have to do is revisit the header file of IMS_ModuleFunction_Comm (not the GUI's header, the module function's header, in case that wasn't clear), add a member that will store a pointer to our GUI and override another tiny method.

Add this member to the private scope:

```
GUI_ModuleFunction_Comm *menu;
```

And this method overload to the public scope:

```
virtual GUI_ModuleFunction_Base *GetGui() { return menu };
```

Of course you'll also have to include the headers of GUI_ModuleFunction_Base and your new gui class to the cpp file of your module function class.

Why we need the member shouldn't need any clarification. And the overlride is obviously there to return the member we just added. A module will use the GetGui() call on all its module functions to see if they have a menu to add to the module control page. The base implementation of the function just returns NULL, so if you don't have a menu, you don't need to override GetGui(). And if you do, all you have to do is return your menu instead of NULL. The module will handle the rest.

But of course our menu hasn't actually been initialised. Let's go back to the constructor of

IMS_ModuleFunction_Comm and change that by adding the following lines after the declaration of our boolean flags that check which animations are present:

```
if (creategui)
{
	//add our GUI to the module's GUI page
	menu = new GUI_ModuleFunction_Comm(this, module->GetGui(), hasdeployment,
					hasscanning, hastracking);
}
```

There are two things here that warrant an explanation. The first is why we're checking if creategui is true. This is a little bit of future proofing. Imagine somebody someday inherits your class to make a more specific implementation, maybe to simulate properties of a real-life antenna. Probably he's going to write up his own GUI for that. But he might need a way to call your constructor without ending up creating your GUI, since he won't use it. The way we did this now, he can do so conveniantly.
The other point is module->GetGui(). module is the pointer to the module containing this module function. The GetGui() call will return its GUIplugin, which we used to create our GUI elements in the constructor of GUI_ModuleFunction_Comm. Every module that has module functions with a GUI has its GUIplugin initialised and ready at this point. But how could it know that our module function has a GUI at this point, since we didn't even create it yet?
Let's go back to the start of this entire tutorial, when we wrote the header for IMS_ModuleFunctionData_Comm. Somewhere I mentioned that there would be a tiny addition to this header once we tackle the GUI. Well, the time has come, and the slight addition to said header is indeed only one line, again a method overrride that just returns something else than the base implementation. For readability's sake I'll repost the whole, now complete header of the data class, I'm sure you'll spot the addition:

```
class IMS_ModuleFunctionData_Comm :
	public IMS_ModuleFunctionData_Base
{
public:
	friend class IMS_ModuleFunction_Comm;

	IMS_ModuleFunctionData_Comm();
	~IMS_ModuleFunctionData_Comm();

	bool HasGui() { return true; };

private:
	string trackinganimname = "";
	string deployanimname = "";
	string searchanimname = "";

	void processConfigLine(vector<string> &rokens);
	bool validateData(string configfile);

};
```

Obviously, the method in question is HasGui(), and all it does is return true instead of false, quite similar to the GetGui method we just made a minute ago. This enables a module to know whether or not it will require a GUIplugin before creating its module functions, and in turn means it can have the plugin ready if it does, or not waste the memory for one if it doesn't.

So... that's about all there is to integrating your menu into IMS2's menu tree. You don't have to worry about deallocation either, since as mentioned, your entire menu is indistinguishable from any other GUI element to the plugin. It will delete it on its destruction like everything else.

So are we done, then? Hardly. We might now see our menu, and if everything is as it should be we can even controll our modulefunction perfectly fine with it, but we don't get any feedback yet. No blinkenboxes, no status updates, nothing. Let's revisit our PreStep() method and change that. Again, I will only post the code relevant to deployment and retracting (retractment?) to not let things get untidy in here. You're welcome to look at the actual code to see that all the rest looks very similar:

```
void IMS_ModuleFunction_Comm::PreStep(double simdt, IMS2 *vessel)
{
	//don't forget to call the prestep function of the base class!
	IMS_ModuleFunction_Base::PreStep(simdt, vessel);


	//check if the state has changed during the last frame,
	//and whether we reached the target state
	if (state.StateChanged())
	{
		//state has changed, update the state description on the gui
		//debug
		string statedesc = state.GetStateDescription();

		menu->SetStateDescription(state.GetStateDescription());

		switch (state.GetStateAndAdvance())
		{
		case COMMSTATE_DEPLOYING:
			addEvent(new StartAnimationEvent(data->deployanimname, 1.0));
			menu->SetDeployBoxState(BLINKING);
			break;
		case COMMSTATE_RETRACTING:
			addEvent(new StartAnimationEvent(data->deployanimname, -1.0));
			menu->SetDeployBoxState(BLINKING);
			break;
		case COMMSTATE_DEPLOYED:
			menu->SetSearchBoxState(OFF);
			menu->SetTrackBoxState(OFF);
			menu->SetDeployBoxState(ON);
			break;
		case COMMSTATE_RETRACTED:
			menu->SetSearchBoxState(OFF);
			menu->SetTrackBoxState(OFF);
			menu->SetDeployBoxState(OFF);
			break;
		}
	}
}
```

Notice what's new? Appart from only launching events, we also update our GUI. The first thing we do when the state changed is update the state description on the menu, by simply passing the description we assigned our states when initialising the state machine to the SetStateDescription method we wrote for our GUI. Hoorray for convienience!

And then we set the state of the respective checkboxes depending on which state we're in. If we're deploying or retracting, the deploybox state starts blinking. If we finished deploying or retracting,

the deploybox lights up permanently or turns itself off, depending on what the state is. But we also do set the other deployboxes in this case. Why would we do that?
Well, imagine the antenna was scanning, and the user ordered it to retract. In that case, the searchbox is blinking while the antenna stops what it's doing and returns to its original position. In other words, it transitions from scanning state to deployed state. Once there, the scanning box obviously needs to stop blinking.
But we don't have any means to verify which state we actually came from at this point. We could make tedious queries as to which boxes are currently blinking and shut them off, but there's really no need for this. Once we reach deployed state, we can be sure that all other movements have finished. So we just shut them all off. This way we'll always get the right result when we arrive at deployed state, no matter which state we came from.

At this point (assuming we implemented this for all states, not just those concerning deployment) we'd already have a pretty neat arrangement: The user clearly sees the current state explained in text, all checkboxes refering to a state that was reached are lit, and checkboxes refering to orders currently being executed are blinking. But we can make it *better*! With very little addition, we can also get the checkboxes blinking that represent *pending* orders, that is, orders that are not currently being worked on, but that will be worked on down the line. So if the user clicks on the deploy checkbox of a scanning antenna, the searchbox will start blinking to signify that the antenna is stopping to scan, and the deploybox will start blinking to signify that once that is done, it will also retract. Let's have another look at CommandDeploy() in order to facilitate this:

```
void IMS_ModuleFunction_Comm::CommandDeploy()
{
	if (data->deployanimname == "") return;

	if (state.GetState() != COMMSTATE_RETRACTED)
	{
		state.SetTargetState(COMMSTATE_RETRACTED);
	}
	else
	{
		state.SetTargetState(COMMSTATE_DEPLOYED);
	}

	menu->SetDeployBoxState(BLINKING);
	state.AdvanceStateSecure();
}
```

What's changed? Only that we now set the deploybox to blinking as soon as the command is triggered. Because we handle everything that will happen later with the checkbox when the state changes, that's quite enough. With that, we're pretty much done here.

Handling keyboard input with Orbiter Input Callbacks

There's one last thing I'd like to look at, although it's really more of an oapi thing than an IMS thing. Still, chances are you'll need it, and it's in everyones interest that you do it right. The word is about input callbacks, which are the nice keyboard prompts you get when for example clicking on the target button in IMFD or the likes. They're an input box provided by orbiter, because otherwise it would be very difficult to intercept key strokes and send them to something other than the

simulation core (although that did get a lot easier in orbiter 2016).
Using these things is pretty easy, but can be confusing to people that are unfamiliar with function pointers and/or void pointers.

The place to start with this is not in the module function class as such, nor in the GUI... first, we want to write our callback function, and that needs to be outside any class. This is because a function pointer and a method pointer are two quite different things, for perfectly logical reasons that none the less sound ridiculously contrieved when you hear them, so I won't expound on it.
Be that as it may, we need a function callback outside the class, but the best place to put it is still in your mdoule function header/source file, just outside the class scope. Add the following declaration to the top of IMS_ModuleFunction_Comm.h (doesn't matter that much where, as long as it's after the includes and before the class declaration):

```
bool SetTargetInputCallback(void *id, char *str, void *usrdata);
```

The important thing about this function isn't the name. That matters exactly zero. What matters is the form: It returns a boolean, and it takes a void*, a char* and another void*, in this order. *This* is the important stuff. That thing could look like this:

```
bool TheGreatGatsbyPeedHisPants(void *Daisy, char *Tom, void *Pammy);
```

And it would *still work*!

The implementation, in our case, looks like this:

```
bool SetTargetInputCallback(void *id, char *str, void *usrdata)
{
	if (strlen(str) == 0)
	{
		return false;
	}

	IMS_ModuleFunction_Comm *comm = (IMS_ModuleFunction_Comm*)usrdata;
	comm->SetTarget(string(str));
	return true;
}
```

It doesn't actually matter where in the cpp file it is located, but if you're a good chap you'll have it either at the end or the beginning, not confusingly amidst methods that belong to the class scope.
Let's have a look at what's going on here.
The first thing you have to realise is that this function will be called after the input box poped up and you hit enter. The id doesn't really interest us. It's useful under certain circumstances which should never arise for us. str contains the text entered by the user when hitting enter, so that's our actual input.
And usrdata is a pointer to an arbitrary object that we can pass along. This is the one that causes most confusion to for the uninitiated, and to understand it you must know what a void pointer is. In short, it's a pointer that doesn't have a type. Just a memory address without any information about what is actually located at this address. Why is this useful? It basically allows us to pass a pointer to literally *whatever we want* to the function, because any pointer of any type can be cast to a void pointer. The caveat being, of course, that the function that receives the void pointer has to know what hides behind it, or it can't do anything with it. This makes void pointers a powerful tool for functions that are themselves passed as pointers.

You see, when we tell orbiter to create the input box to set the target, we'll tell it to invoke exactly this function by passing it as a pointer (you'll see how that works in a minute, it'll be fairly trivial to understand once you got these basics). We'll also send a void pointer to the module function that spawned the input box along as usrdata, because the input box cannot be reasonably expected to know about the types we're using. So if I don't screw up my code completely, I can be assured that this function will only ever get called by an input box spawned by a IMS_ModuleFunction_Comm to set the target. And thereby I can also be assured that usrdata to this function will *always* hide an object of type IMS_ModuleFunction_Comm. As a result, we can cast the void pointer back to something useful with impunity, here done at the end of the function. And why do we need this? Well, it would be kind of nice if whatever the user entered finds its way back to the module function that spawned the input box, right? By casting the void pointer back, the callback can pass the entered string back to the module function via a public method, which we'll look at shortly.
At the end, we return true, which closes the input box. Returning false will immediately respawn the input box, which we here only do when the user has entered nothing at all.

Let's set this thing up with our module function! You might remember that we had a "set target" button in the GUI, and you might also remember that really all we want to do in the GUI is to call a method on the module function, not process things right then and there. If your memory is, like, *incredible*, you might also remember that we did call CommandSetTarget() on the module function when the target button was clicked in the menu.

Let's actually put that function in (the declaration goes of course in the public scope, this gets called from our menu after all):

```
void IMS_ModuleFunction_Comm::CommandSetTarget()
{
	if (data->trackinganimname == "") return;

	//open an input callback for the user to set the target
	oapiOpenInputBox("enter target", SetTargetInputCallback, NULL, 15, this);
}
```

After a short sanity check to see whether this module function even has a tracking animation (true, the button that calls this method shouldn't exist if it doesn't, but sanity is sanity... This is a public method after all) we tell orbiter to open an input box, and that call should now no longer be a complete mystery. What we pass here is a title for the input box, immediately followed by a pointer to our callback. Function pointers are passed by just passing the function name without (). The messy syntax is all on the receivers end, which in this case would be our dear Doctor. Since we here pass a pointer to our function to the input box, we can be sure that this specific input box will always call that function and not something else.
Next we pass a NULL. We could actually pass a char* containing an initial text for the input box (that is, an initial *input*, not a title. We already passed that!), but we don't want that here. Then we're passing a number that tells the input box how many characters wide it should be. And finally, we pass a pointer to our very own module function instance. This is of course the usrdata, which we don't even have to cast, because any pointer can be cast to void* implicitly.
There, done! That was the syntactically complex part.

But we're still short one method, namely the one that will receive the input from the callback. Again, the declaration goes into the public scope, since the callback is entirely outside the class scope of our module function:

```
void IMS_ModuleFunction_Comm::SetTarget(string target)
{
	if (data->trackinganimname == "") return;

	OBJHANDLE checktarget = oapiGetObjectByName((char*)target.data());
	int currentstate = state.GetState();

	if (checktarget == NULL)
	{
		//if an invalid target was entered, display an error message
		menu->SetTargetDescription("invalid target!", true);
		//if the antenna is currently tracking or aligning with a previous target,
		//stop it and bring it back to origin
		if (currentstate == COMMSTATE_TRACKING ||
			currentstate == COMMSTATE_ALIGNING)
		{
			state.SetTargetState(COMMSTATE_DEPLOYED);
		}
		targetname = "";
	}
	else
	{
		//a valid target was entered, update state, animation and gui
		menu->SetTargetDescription(target);
		targetname = target;
		if (currentstate == COMMSTATE_TRACKING ||
			currentstate == COMMSTATE_ALIGNING)
		{
			addEvent(new ModifyTrackingAnimationEvent(data->trackinganimname, 0.0,
				checktarget));
			state.SetTargetState(COMMSTATE_TRACKING);
		}
	}
	state.AdvanceStateSecure();
}
```

Huh, that's probably a bit longer than expected. What's going on? well, mostly really just input validation, a thing every developer with a modicum of experience knows to be 1) really annoying, 2) more tedious than it has any right to be, and 3) absolutely indispensible. We won't look too much at it, since most of it is very specific to our use case (like verifying if an object by the entered name actually exists in the scenario). But there's still a few things worth mentioning.
First off, what happens in case of an invalid input? We set the target description to something that will let the user know that he made a typo or two. The true we pass along with the message when calling SetTargetDescription() will cause the LabelValuePair to draw in highlighted font, which in our case is red as you might recall, so the user will immediately see that something is wrong.

But we don't just have to flash an error. We also have to stop tracking if the antenna is currently tracking, because obviously we can't track an invalid target. So we tell the antenna to get back into deployed state right away. Everything else, like stopping and starting animations, is at this point safely handled by PreStep(), thanks Jove for that!
We do something similar if the target is valid... First we tell the tracking animation that it has a new target, and then we tell the antenna to get into tracking state... if it is already in tracking state? Why is that necessary? Well, it is necessary because the state must indeed change. When you're tracking earth, and suddenly the target changes to mars, then you're off target, so you have to align again first. Which in turn means that the tracking box in the menu should start blinking, etc. The trick here is the way the state machine looks for a path, which I've mentioned before. It will not just remain in

the current state if the same target state is set. It will search the shortest way from the current state back to itself, which in this case will go back to ALIGNING, which will cause the antenna to home in on its new target, and from there to TRACKING, and will therefore trigger all the appropriate state changes that will keep the animation, our module function and our menu in synch.

And that's pretty much that. There's not much more to learn in this example, congratulations. Now go and code me some cool module functions, please?

Stuff you might have to deal with that wasn't explained here

Unfortunately, there's a chance that you'll need to do some stuff that is outside the scope of this guide. I'll just list the ones I expect you to run into with a few hints at what to do, so you're not completely shot.

I need a GUI element that doesn't exist!

The GUI isn't finished. There's all sorts of common elements still missing. The way we created the menu here is of only limited use for writing a completely new GUI element, since this will involve using the drawing engine directly, as well as blitting and some rather esoteric and not too thouroughly documented knowledge about how styles and fonts work.
In general, I'd say that it won't be worth it if you learn to do all that stuff unless you really want to. If you just want a new GUI element to implement your menu in an orderly fashion, drop me a note. Usually it isn't a very large effort for somebody who's familiar with the intricate details of the toolkit.

I need the core to do something special!

This will occur more often than you might think. Basically, this example doesn't touch the core at all, because it adds nothing to the actual VESSEL instance (the animations are their own module function, and do indeed have a special method in the core to support them properly). But oftentimes adding something to the VESSEL instance is exactly what the module function is there for. It is still worth thinking about whether core support is really necessary. The module function gives you the possibility to add and remove stuff from the vessel without problem. But often it will be required that the core *manages* something. For example, when a Tank module function gets added, the IMS2 core needs to be made aware of it and be able to track its state, and connect it to thrusters.
Once you realise that you need core support, the first thing to do is consult with me. Not because I am protective of the core like a mother bear, but simply so I know what's needed and what's being worked on. Things have a tendency to get made twice if nobody keeps the oversight. Chances are I can give you some suggestions as to how to best implement the needed functionality.
There's one rule that you can keep in mind though: Simple wrappers that just call underlying methods of the vessel with maybe some twists should be put into IMSGetSet.cpp. Any interconnected functionality that needs more than two or three methods should be implemented as a manager class that the core vessel will retain an instance to, rather than having the mehods directly implemented into the IMS2 class. Finally, methods that don't really have to do with the core but will be needed by some of its methods, like algorithms for complicated calculations and such, should not be part of the core at all, but be put into either the IMS_algorithms namespace (if simple functions) or a class in that namespace (if it's something more complex involving multiple methods).

I need a special event!

Sure, go ahead. There's not much you can screw up with that, as events are the very incarnation of modular compartmentalisation. Take a look at how the existing events are constructed, and at their documentation, throw me a question if you're unsure. I can't think of any part of the code where you could mess up less, just as long as you don't tinker with the existing ones, and especially not with the event engine itself. Once an event is in broad use in the code, changes to it should be done only for very good reasons, by people who know exactly what they're doing to them, and with rigorous testing afterwards. But creating new events? You literally can't break anything by doing that.