

CJP's Orbiter SDK extensions

Last update: April 25, 2010

Author: CJP

Introduction

Orbiter contains several built-in MFDs (Multi Functional Display modes), and it also provides a plug-in mechanism for adding new MFDs. However, Orbiter's built-in MFDs use some functionality that is not available through its plug-in programming interface. An example of this is the use of pop-up menus for selecting a reference or a target object for the MFD.

This document describes some extensions to the Orbiter SDK, which allow MFD plug-in developers to use the same plug-in functionality as the built-in MFDs.

The programming interface

The following functions are available:

```
void oapiCelBodyPopup(  
    void *userPtr, char *title, menuClbk clbk, DWORD flags);  
  
void oapiTargetPopup(  
    void *userPtr, char *title, menuClbk clbk, OBJHANDLE refObj, DWORD flags);  
  
void oapiSetHUDReference(OBJHANDLE refObj);
```

The function **oapiCelBodyPopup** shows a pop-up menu listing all celestial bodies in the simulation. This is typically used for selecting a reference object.

The function **oapiTargetPopup** shows a pop-up menu listing all celestial bodies and vessels orbiting around a certain celestial body. This is typically used for selecting a target object.

The function **oapiSetHUDReference** sets the reference object for the HUD (Head Up Display). It also sets the HUD to Orbit mode.

The functions receive the following arguments:

- **userPtr**: a pointer of arbitrary type that will be passed to the callback function. If you don't know how to use this, or you don't care, you can use NULL. However, it is often useful to let this argument point to your MFD instance.
- **title**: this is the title that will appear on top of the pop-up menu.
- **clbk**: this is the callback function that will be called when the user selects one of the menu options. See below.
- **flags**: certain flags that affect the behavior of the menu. See below.
- **refObj**: the reference object. **oapiTargetPopup** will show all objects that orbit around this object. **oapiSetHUDReference** uses this as the new reference object for the HUD.

The following flags are known:

- **MENU_HIDE_SUN** (value 0x1): When set, the sun is not listed in the pop-up menus.

As you see, so far, only one flag is known.

The callback function prototype is the following:

```
bool menuCallback(  
    void *menuPtr, unsigned int menuIndex, char *itemName, void *userPtr);
```

The return value should indicate whether the program accepts the user's choice (**true**) or rejects it (**false**). When the user presses the *Escape* key while in the menu, the menu will be closed without any callback calls.

The callback function receives the following arguments:

- **menuPtr**: Probably(!) a pointer to the menu object. Can be NULL in a rare case (see the source code).
- **menuIndex**: The index of the selected menu item in the menu.
- **itemName**: The name of the selected menu item.
- **userPtr**: The same pointer that has been passed to the popup menu function.

Known bugs / issues

- These functions only work properly for Orbiter 2006-P1. For other versions, the reverse engineering that made these functions possible needs to be done again. The functions check whether they deal with version 2006-P1: if not, the pop-up menu functions will simply provide an **oapiOpenInputBox** to the user, and a call to **oapiSetHUDReference** will simply be ignored.
- Somehow, calling one of the pop-up functions multiple times can cause problems. Orbiter has to do some things between two calls to make it work. When you get a menu that is longer / wider than expected, containing duplicate entries and/or empty space, you can check the following things in your code:
 - Make sure these functions are only called once from your MFD's event handlers
 - If the pop-up function is called as a result of a mouse click on an MFD button, make sure you only respond to button down events, and not to other events. You should have a piece of code like this in **ConsumeButton**:

```
if(!(event & PANEL_MOUSE_LBDOWN)) return false;
```

How it was done

These functions were created by partially reverse-engineering Orbiter.exe. This section describes how the reverse engineering was done.

Assumptions

First of all, the pop-up menus have a very similar look-and-feel as the input box created by **oapiOpenInputBox**. They are also created by the same programmer, so it makes sense to assume they have a similar programming interface (with a function to initialize the thing, and a callback for when the user finishes a choice).

The next step was to realize that the pop-up menus used by the built-in MFDs have fixed titles, like “Orbit MFD: target”. Like in **oapiOpenInputBox**, this title string is probably passed to the menu initialization function as a character pointer argument. Also, the fixed title is probably a string literal in the MFD code that calls the initialization function.

Getting started

Now, string literals are stored as statically allocated data in the .exe file. The pop-up menu titles can easily be found in Orbiter.exe by searching for them with a hex editor. The actual pointer values that will point to these strings are easily calculated, as windows simply loads the entire .exe file into the address space of the process. With a PE file viewer or a debugger it's easy to find out the address offset of Orbiter.exe, and together with the positions of the strings in the .exe file, we now have the pointer values of the title strings. They can be checked in the debugger.

The next step is to find out on which places these strings are used. I used a disassembler for this, but you can also search for the pointer values in the .exe file with a hex editor. If you choose the last option, you have to be aware that the x86 is a little endian platform, so the least significant byte comes first. If everything is OK so far, each string should be used in exactly one place.

The disassembler already showed me the start addresses of the assembly instructions around the location where the string pointer was found. If you used a hex editor you don't have this information, but you can probably figure it out once you know the pointer is the argument of a push instruction. It makes sense to assume it's a push instruction, as push is the thing that places function argument values onto the stack. My disassembler already showed me it's a push instruction in all cases I investigated.

Diving into the code

With the start address of the instruction, Orbiter.exe can now be started from inside a debugger, and we can set a breakpoint at that address. In Visual Studio, the disassembly window nicely confirms again it truly is a push instruction. With the breakpoint set, we can check whether it breaks at the moments we expect it to break. For me it did.

Now, the assembly code can be investigated in more detail. This can be done in the disassembly window of Visual Studio, or in any other disassembly tool. The code clearly shows the typical pattern of a function call: a certain number of push instructions, followed by a call instruction. One of the push

instructions is the one that pushes the title. The call instruction is the most interesting one, as it tells us the address of the function that triggers the menu.

Now it turns out that there are three different functions: one at location 0x00431050, one at location 0x00431190, and one at location 0x004994D0. Also, by further investigation of the assembly code in the first two functions, it turns out that these two actually call the third function. So, the third one is the real function, and the first two are 'utility' functions that make life easier for the programmer in some way.

The rest of the analysis is about determining the properties of these functions. So far, this has only been done for the first two functions, and as a result these are the only ones that are currently available in this API.

Detailed analysis

The first thing to find out is the number of arguments, and the return value. In the calling code, we can see three push commands before 0x00431050 is called, and four push commands before 0x00431190 is called. In the code of the functions themselves, we can check whether these are actually used (and of course they are). Hint: the ESP register points to the current position of the stack. The four bytes on that position are the return pointer for the function call. The stack grows downward, so all recently pushed values are found on higher addresses, while later values (such as local variables of the function) are found on lower addresses.

Next to values on the stack, there's something else that could influence the function: register values. Some calling conventions are known to use registers as a fast alternative to the stack for passing function arguments. Now, by looking at the function code, we can already see that some registers are not used for argument passing, as their values are simply overwritten. However, the ECX register is explicitly used in both functions 0x00431050 and 0x00431190.

By browsing the Internet for calling conventions used by microsoft compilers, it becomes clear that this is consistent with the *thiscall* calling convention. This convention is used for calling methods in C++: the **this** pointer is placed in ECX, and all other arguments are placed on the stack.

Now, it would be interesting to know what kind of object ECX points to, but this turns out to be relatively unimportant¹. The functions 0x00431050 and 0x00431190 don't de-reference ECX: they only pass it to the function 0x004994D0. 0x004994D0 also follows the *thiscall* convention, but its **this** pointer points to another object. Later in the reverse engineering process, it is confirmed that passing NULL as ECX value to 0x00431050 and 0x00431190 doesn't cause a crash, so it's safe to assume that this pointer is not dereferenced *anywhere*.

Meaning of the arguments

Now, back to the function arguments on the stack. 0x00431050 is the most simple case, as it has only three arguments. As you might have guessed, 0x00431050 corresponds to the **oapiCelBodyPopup**

¹ If you really want to know: some runtime debugging of function 0x00431050 suggest that its **this** pointer (= ECX value) points to an object of the same or similar type as the **Instrument_User** class in mfdapi.h.

function, and 0x00431190 corresponds to the **oapiTargetPopup** function, and requires an extra argument for the reference object. The other three arguments are identical.

We already know the purpose of the last² argument of 0x00431050: it's the title. The first one also seems simple: it is always set to a constant 32-bit value, either 1 or 0. It could be a boolean, or, more generally speaking, a bitmask of flags. Later, its meaning was discovered by experimenting (the meaning is that of the **flags** variable), but for now it's sufficient to know that 0 and 1 are valid argument values.

This leaves the middle argument of 0x00431050 as the last unknown. Because of the assumed similarity with **oapiOpenInputBox**, we know there still needs to be a callback function pointer, so this is probably it. By setting a breakpoint at 0x00431050, and triggering a break, we can find out that its value is actually a pointer to a location within the Orbiter.exe image. We can go to that location in the disassembly window, and investigate the assembly code there, and set a breakpoint. By setting a breakpoint there, it was confirmed that this function was called at exactly the moment we expected: when the user selects an option in the pop-up menu.

The callback function

Now, an interesting question is how many arguments the callback function should have, and what its return value should be. Especially the behavior on the stack is important, because when a called function handles the stack in a different way than the calling function expects, this can potentially lead to crashes. As long as we don't crash the thing, we can make our own callback function, give it to the popup menu, and see what values we get back.

The number of arguments was determined by breaking the program in the known callback function of a built-in MFD³, and going one step back in the call stack. You can do this manually by looking at the values on the stack: the value where ESP points to, is the return pointer to the calling function. By going to that address in the disassembly window, you can see the assembly code that calls the callback function. This is again a typical function call: four push instructions, followed by a call instruction. So there are four 32-bit arguments to the callback function.

For the return value, I simply assumed it to be boolean, just like in the **oapiOpenInputBox** case.

2 Here, “last” is the argument that is pushed on the stack last. This is not necessarily the same as the last in the C++ source code (in fact, I don't know what the convention is here).

3 We know it because the middle argument of 0x00431050 points to it. Remember?

Using the functions

Now it's time to write some code. We know which function should be called, and we know what kind of argument values it expects. This is the code for calling 0x00431050:

```
typedef bool (*menuClbk)(void *, unsigned int, char *, void *);

void *InputBox_userPtr = NULL;
menuClbk InputBox_clbk = NULL;
bool InputBoxCallback(void *id, char *str, void *usrdata);

void oapiCelBodyPopup(void *userPtr, char *title, menuClbk clbk, DWORD flags)
{
    DWORD testProcAddr = 0x00431050;

    //Check for exact orbiter version,
    //and provide alternative if not
    if(!isOrbiter2006p1())
    {
        InputBox_userPtr = userPtr;
        InputBox_clbk = clbk;
        oapiOpenInputBox(title, InputBoxCallback, "", 20, NULL);
        return;
    }

    /*
    In the actual function 0x00431050, ecx (the this pointer)
    points to an InstrUser object. However, it is not
    dereferenced, so we can put any user-defined pointer
    in it.
    */
    __asm mov ecx, [userPtr]
    __asm push [flags]
    __asm push [clbk]
    __asm push [title]
    __asm call testProcAddr
}
```

Some comments on this:

- A special function **isOrbiter2006p1** was developed to test whether we're actually dealing with the exact Orbiter.exe version for which the reverse engineering was done. If not, a safe fall-back, based on **oapiOpenInputBox**, is provided.
- The actual function call is written in assembly. This is done to deal with a **thiscall** function, without the need to do extremely complicated and very dirty C++ hacks.
- The **this** pointer of the called function is abused to give the **userPtr** to the pop-up menu.

Now, with this code, we can experiment. The details about the callback function (meaning of the argument values, and effect of the return type) were discovered. Also, the difference between **flags = 0** and **flags = 1** was discovered. A similar function, calling 0x00431190, was developed, and it turned out to work fine when a simple OBJHANDLE was given as the extra argument.

Setting the HUD reference body

The function of the built-in Orbit MFD to set the HUD reference body was a bit tougher to get, because, unlike the pop-up functions, it doesn't use any easy-to-find string literals. It became possible for me as soon as I realized that all button functions of a single MFD class are likely to be called from a single function. In other words, I expected a piece of code like this pseudo-code:

```
OrbitMFD::buttonEvent(buttonID)
{
    switch(buttonID)
    {
        case REF:
            refFunc();
            break;
        case TGT:
            tgtFunc();
            break;
        ...
        case HUD:
            hudFunc();
            break;
    }
}
```

Now, the trick is that we already know locations inside `refFunc()` and `tgtFunc()` where we can set breakpoints, because of the pop-up functions we found earlier. By breaking inside one of these functions, and reading return addresses from the stack, the described function can be found. We can set a breakpoint on the start of each function we find on the stack, and then see which of the breakpoints are triggered by a click on the HUD button.

It turns out that the above pseudo-code was correct. The deepest functions in the stack are shared between the different buttons of Orbit MFD, but in one function, each button triggers a different execution path. This happens through a conditional jump-instruction, which is probably part of the compiled switch statement.

In this 'execution-path-splitting'-function, the only thing that is done in the execution path of the HUD button, is a call to another function. So, this other function has to contain all the functionality we're looking for. Luckily, this other function contains only about 20 assembly instructions, which makes it feasible to analyze the code manually.

The lucky result of this analysis is that the function only uses the following data objects:

- An object that can be obtained through a statically allocated pointer (so the pointer is available at a fixed memory location).
- A piece of data that is relative to the *this* pointer given to the function. By comparing it with some OBJHANDLE values of celestial bodies, this turned out to be an OBJHANDLE to the new reference object.

Both are available to us, even when we do not have a pointer to a real Orbit MFD object. The rest of the job was simply a matter of imitating the behavior of this function with a self-made piece of code.

What still needs to be done

For now, the function 0x004994D0 is the holy grail. It should provide all the functionality of the other two pop-up functions, and more. The assembly code of 0x00431050 and 0x00431190 should provide examples of how 0x004994D0 can be called. Other locations where 0x004994D0 is used directly are for the docking MFD target and map MFD target: they should provide clues on how to use the additional functionality.

With the techniques that are described here, a lot of other functions can also be uncovered. At a more ambitious level, we could even start a sort of documentation project for the contents of the .exe file, with a database of addresses, function prototypes, high-level code of what it does, & so on. However, it would be nicer if the next version of Orbiter has all functionality used by the built-in MFDs exposed through a public API. It would be even better if (more and more parts of) new Orbiter versions become open source, so that dirty hacks like this won't be necessary anymore.

Final words

I sincerely hope that this hacking is appreciated by Martin Schweiger, and I hope it doesn't make him think he needs to encrypt his binary-only releases. The only reason I did it was to make something useful available to the Orbiter community that was not available otherwise. As mr. Schweiger releases Orbiter as freeware, I believe we basically work on the same goal.

I learned a lot by doing this, and I kind of enjoyed it in a masochistic way, but I think it would have been a lot easier if the source code were available. I hope my results demonstrate that hiding the source code can make life a lot more difficult, but in the long term it won't make a difference. I even believe that encryption of binaries and DRM will not stop people who are really willing to spend time on hacking a system.

I don't think this is a bad thing: in general, openness about technology gives more freedom to consumers. On the other hand, it might be dangerous for those who base their business model on hiding information. Doing business based on hiding information might work for a while, just like doing politics based on hiding information might work for a while, but in the long term it is not sustainable. I hope that the availability of powerful hacking techniques makes people respond in a positive way, by transforming the industry to become based on openness, instead of a negative way, by protecting secrets with technology and laws, which in the end will hurt the consumers most.