

# FuBi: Automatic Function Exporting for Scripting and Networking

By Scott Bilas

## Abstract

Much has been written about scripting engines – how to build the parser, how to design the virtual instruction set, and so on. And likewise, much has been written about networking – how to handle latency, how to pack up data efficiently, and how to pipe it to other machines. A critical piece tends to be missing, left as an exercise for the reader, and that is: how do you get your cool engine to talk to the game? That is, how does a scripting function call a C++ game function? How does a network remote procedure call actually invoke the C++ function when it reaches the other machine? What about more complicated issues such as passing pointers or strings over the network?

Typical solutions to this problem end up requiring nontrivial architectural changes to your game engine and lots of shoehorn code. This paper describes a novel method for exporting a game's functionality to dynamic systems such as scripting, networking, and databases that is almost completely unobtrusive to a game's systems. It can be bolted on to an existing game with very little effort. Exporting a game function of nearly any kind ultimately resolves down to inserting a single macro before the function declaration. Making the function networkable requires only a single additional macro inside of the function.

A prototype version of this system was published as the “Generic Function Binding Interface” gem in the book Game Programming Gems. Much has been learned since then. By the end of this paper, if you find yourself wondering whether or not this stuff really works, consider that an enhanced version of the design given here is essentially the foundation for much of *Dungeon Siege*, a game by Gas Powered Games. What started out as a 1000 line weeklong experiment (back when FuBi really just meant “function binder”) has turned into a 16K line gift that keeps on giving. This technique has completely changed the way that this author builds games.

## Audience

This paper is intended for engineers working “below the root” on back-end systems. Polygon pushers won't find much of interest here. While network and scripting engines are used as examples, they are just conveniently illustrative. This system is far more versatile than just these relatively simple needs, in that it is effectively a global type database, and as such is applicable to all kinds of fun things (more on this later).

All discussion and samples given here are very specific to a particular platform: Visual C++ 6.0 running on an x86 version of Win32. If you're working with a different platform and/or compiler, all is not lost. Everything should still be portable to most other platforms.

Here is what is platform-specific:

1. There's a little bit of assembly code in here that is obviously x86 specific. All x86 assembly code can be converted to any other instruction set, assuming the system uses a stack to pass parameters around.
2. How calling conventions work is specific to Visual C++ 6.0. Study your platform's calling conventions before you start porting. If you're using a system that passes parameters through register frames this task should become quite interesting.
3. The name mangling and unmangling is also specific to Visual C++ 6.0. I'm using DbgHelp.dll so I don't have to reverse-engineer the VC++ name-mangling format. In addition, this will provide forward compatibility with later versions of VC++. For other compilers, if you have source code (no problem with GCC) it should be relatively simple to figure out how names get mangled and unmangled.
4. Specific use is made of the way that Win32 image (DLL/EXE) exports work. Dynamic link libraries are hardly unique to Win32 – all FuBi needs is a table that maps exported function names to memory addresses. A linker map file will do just fine, although it will probably require post processing the EXE to bind the information in, which has obvious synchronization problems. If you choose to go with an EXE postprocessor, be sure to integrate it into your build scripts so synchronization is guaranteed.

Hopefully these issues will not be a problem for your chosen platform!

## The Problem

Here is the basic problem. We're trying to find a way for a system to call arbitrary engine functions. The mechanism must be convenient, efficient, and safe.

Conveniently, the compiler has to solve the exact same problem. It pushes some parameters onto the stack, calls the function, and then somebody pops the parameters off the stack, returning the results in a register, or perhaps on the stack somewhere. In order to be able to do this, the compiler requires quite a bit of information about the parameters being passed and the function being called. The actual address of the function is resolved at link time, but the compiler knows everything else in advance. So let's copy the compiler if we can.

Before we get started though, first we need to state the requirements of our psuedo-imaginary sample systems that require access to the engine's functions. This will help explain the motivation behind some of the techniques here. Why choose a scripting engine and a net pipe for our examples? They are two systems that can take immediate advantage of something like FuBi. They also explore many of the features that really give the system its kick.

Let's say the scripting engine is your standard p-code interpreter that uses a virtual stack to manipulate expressions. This is a common design because it is easy to implement and in many ways is very similar to how C code operates, and as such is familiar ground for us. The scripting engine will support some basic types: `bool`, `float`, `int`, `string`, and `void`. For fun, let's have it support pointers to user-defined types as well. The basic requirement of our scripting engine is that it can call engine functions that use these types, automatically passing

parameters properly, and receiving them back. The `string` type will work with either `const char*` or `std::string` as parameters or return values.

The basic requirement of our net pipe is that we can automatically do remote procedure calls (RPC's) without a lot of hassle. This means that the system should automatically intercept an ordinary function call meant to be an RPC, and then pack up and send all necessary parameters to the other machine. And as long as it's doing this, it should also process any pointers it finds in the parameter list so that they make sense to the other machine. Strings or special "plain old data" types and memory blocks being passed to a function can be tacked on to the end of the parameter list and passed over the network without worries. General pointers, on the other hand, can't be passed over without converting them first to a "network cookie" that the other machine can resolve back to a real pointer. Our function binding system should include a variety of methods to support this parameter pre- and post-processing.

FuBi must do all of these things and yet maintain convenience, efficiency, and safety. It must be convenient in that it doesn't make the engineer jump through hoops to export a function or the game's types. It must be efficient by not requiring a lot of work in order to make these function calls, or to pack parameters up for the network. And it must be type safe, so that when we call a function, the types passed in will match what that function is expecting so we aren't surprised by the results.

## One Possible Solution

Enough talk, let's get to work! What we need is a system for calling functions generically. For this we'll need a table of all our exported functions that includes at minimum (a) the name of the function and (b) a pointer to it. How about this:

```
void Baz( void );
void Tel( void );
// ...

struct Function
{
    typedef void (*Proc)( void );

    const char* m_Name;
    Proc        m_Proc;
};

Function g_Functions[] =
{
    { "Baz", Baz, },
    { "Tel", Tel, },
    // ...
};
```

It really can't get much simpler. The `Function` structure represents an individual function export, and includes the name of the function and a pointer to it. The `g_Functions` variable is a table of all available functions to be called. Our example function exports are of course `Baz` and `Tel`. When our scripting engine finds a call to function `Baz` in a script, it can search through the

`g_Functions` table to resolve function calls by name and then call the procedure directly once found. Hopefully this lookup would be done through an index for speed. Our net pipe can use a faster and smaller 2-byte serial ID to resolve the function pointer across the network. We can simply use the index of the entry within the table as the serial ID. Easy and simple.

This solution would work well, and it is very efficient and safe. Unfortunately, it is so inconvenient to export functions that it totally cripples the design for any project with more than a handful of functions to export. First, all functions must be identical in signature – they must all take no parameters and return `void`. To work around this, we could change the `Function:Proc` type so that the functions could at least return a value and take some parameters. Perhaps we could make it take a `float`, an `int`, and a `const char*`, and return an `int`, and hope that the functions would be able to pick what they wanted and use them, but this is just not acceptable. Considering the large and varied function sets required of modern games, it's unlikely that we'll be able to find a lowest common denominator parameter set that will work for all our exported functions.

A popular way to work around this problem is to use a callback casting technique. Change `Function:Proc` so that it takes, say, three `long` parameters and then cast the real types to these before calling the function. Within the function we “crack” them to get the real types back again. In situations where more than three parameters are required, we can make one of them a pointer to a `struct` that contains the rest of the parameters. This technique is fairly efficient, and is commonly used by commercial API's for generic callbacks – for example, in Windows we call this a “window procedure”. However, while it's flexible and fast enough for our needs, it's obviously very unsafe and inconvenient, considering all the casting that is going on here.

There are many problems with this approach. You can't simply call one of these functions; you have to turn all your parameters into `longs` first. And you can't simply write one of these functions, you have to accept a bunch of `longs` and crack them to get at your real parameters. Exporting a function is now a time-consuming process. You also have to write up documentation on the exact protocol clients must use to call each function, something that normally is made obvious by the function prototype. What a pain! In addition, because the types being passed in are not the real types, this makes it impossible to figure out which of the generic parameters are pointers, which makes passing the parameters over the network for RPC's very difficult.

Now the painful process of converting to and from `longs` can certainly be helped with a bunch of clever macros and templates. Or perhaps we could store all our parameters in a special “frame” object that automates the packing and unpacking and maybe even adds a measure of type safety by storing types along with the data. Maybe we could receive an `argc` and `argv` like `main()` does and extract arbitrary parameters from the incoming strings. None of these solutions are very appealing because they all have one thing in common: they are all inefficient and inconvenient. They would work, but we can do better.

## The Function Database

It's just not going to be possible to come up with a standard function pointer type that we can use to call exported functions. This effectively means that we can't have the scripting engine or net pipe make function calls using the normal C++ methods we're used to. At compile time, we just don't know what parameters the function we're trying to call is going to take. We found out in the last section that we can't place draconian restrictions on the functions that are to be exported (which would allow standard C++ function call methods), and so we have no choice but to figure out how to build a completely general-purpose "blind" function calling system.

So how does a blind function call work? We need to do the same things the C++ compiler does when it generates machine code: push the parameters, call the function using its address, and then retrieve the return value to pass back to the original caller. The parameters may come from our script engine's virtual stack, or it may come from a network packet if sent via RPC. These new requirements make the design of the function export table a little more complex – it may be more accurate to call it a function database.

Each entry in the function database will contain all the information that is necessary in order to generically call an individual function, which is about the same as what the C++ compiler needs when it calls a function. It needs to know the function's name, its address, the types of each of the parameters and the return value if any, and the "calling convention" (how we call the function). Note that, for the compiler, the function address is not resolved until link time, but this will not be a problem for us. The reasons for this will be obvious later on. Here is the new and improved function database:

```
// function specification
struct Function
{
    // simple variable spec
    enum eVarType
    {
        VAR_VOID, VAR_BOOL, VAR_INT,
        VAR_FLOAT, VAR_STRING, VAR_USER,
    };

    // possible calling conventions
    enum eCallType
    {
        CALL_CDECL, CALL_STDCALL, CALL_THISCALL,
        CALL_THISCALLVARARG,
    };

    typedef std::vector <eVarType> ParamVec;

    std::string m_Name;
    void* m_Proc;
    eVarType m_ReturnType;
    ParamVec m_ParamTypes;
    eCallType m_CallType;
};

typedef std::vector <Function> FunctionVec;
```

```
// the global set of specifications for exported functions
FunctionVec g_Functions;
```

Again, the function index (or primary key) within the database can be used as the serial ID. So as not to be distracted, let's assume for the moment that we have a way to fill `g_Functions` with specifications for all our exported functions (we'll get to how this works later, don't worry). Now how can we use this information to actually call functions? First we must learn about calling conventions.

## Calling Conventions

You can check your compiler's documentation to see how its calling conventions work. On Visual C++ for x86 Win32, all function calls have certain things in common:

1. The stack grows downward, and all parameters are pushed from right to left. In effect, parameters go from left to right on the stack for increasing memory addresses.
2. The stack pointer (`esp`) always points to the lowest memory address of the stack, which unfortunately has the name of "top". It must be `DWORD` (4-byte) aligned, so each parameter pushed must be likewise aligned to a `DWORD`. The `push` instruction decrements `esp` first, then stores the data. The `pop` instruction loads data first, then increments `esp`.
3. Parameters passed by value are pushed on the stack in their entirety. Any `doubles` (8-byte) and user-defined types are just copied right onto the stack. The memory addresses contained by references and pointers are pushed on the stack directly. Note that to the compiler's code generator there is no difference between a reference and a pointer, so we can treat them as effectively the same thing.
4. Simple non-float return values like integers and pointers are stored in the `eax` register. 8-byte structures are returned in `edx` and `eax` as a pair. Floats and doubles are returned through the FPU in `ST0`. Return values for user-defined types have their addresses pushed onto the stack last, but will also be returned in `eax`.
5. When calling a nonstatic member function of a class, the `this` pointer is stored in the `ecx` register. The `this` pointer will also be returned in `eax`, not that we'll need it.

Disclaimer: I came up with the above rules empirically – by looking at disassembly listings of various function calls and seeing how things worked. Some of these rules may not be absolute, though in the year and a half or so of FuBi's existence I've found no exceptions.

Here are the four calling conventions that we'll be supporting:

- `cdecl`

The caller is responsible for popping its own arguments off the stack after the call completes. This is the default calling convention for static and global functions in C and C++. In addition, this convention is required for variable argument functions because the called function doesn't have the information it needs to pop the correct number of arguments.

- `stdcall`

The called function cleans up the stack. This is the standard convention used for Win32 API calls, probably because it is slightly more efficient in terms of client code size.

- `thiscall`

This is a C++ nonstatic member function call. It is exactly the same as `stdcall` except additionally store `this` in `ecx`.

- `thiscallvararg`

This special calling convention is only used with variable-argument nonstatic member function calls in C++. It is exactly the same as `cdecl` except (a) additionally store `this` in `ecx` and (b) `push ecx` on the stack right before the function call.

There is one other calling convention (`fastcall`), but supporting it is probably not worth the time to implement. The `fastcall` convention is meant for performance, and given the relative overhead of any script engine or network RPC, this performance gain would be lost instantly. It would be far simpler to just require that any exported functions use another calling convention.

Time to write some assembly code. Say our generic sample system decides on a function to call and it has the parameters ready. We will have the function address, a pointer to the parameters (organized as required by all calling conventions), the size of the parameters, and possibly a pointer to the object if it's a member function. That's all we need! So let's write some utility functions that take care of calling the function properly given all of these parameters and the calling convention. We'll also need a function to retrieve a floating-point value from the FPU's `ST0` register (as is convention) to be stored in a generic return value. These will work:

```
DWORD Call_cdecl( const void* args, size_t sz, DWORD func )
{
    DWORD rc;                // here's our return value...
    __asm
    {
        mov     ecx, sz        // get size of buffer
        mov     esi, args      // get buffer
        sub     esp, ecx       // allocate stack space
        mov     edi, esp       // start of destination stack frame
        shr     ecx, 2         // make it dwords
        rep     movsd         // copy params to real stack
        call   [func]         // call the function
        mov     rc, eax        // save the return value
    }
}
```

```

        add    esp, sz        // restore the stack pointer
    }
    return ( rc );
}

DWORD Call_stdcall( const void* args, size_t sz, DWORD func )
{
    DWORD rc;                // here's our return value...
    __asm
    {
        mov    ecx, sz        // get size of buffer
        mov    esi, args      // get buffer
        sub    esp, ecx       // allocate stack space
        mov    edi, esp       // start of destination stack frame
        shr    ecx, 2         // make it dwords
        rep    movsd          // copy it
        call   [func]         // call the function
        mov    rc, eax        // save the return value
    }
    return ( rc );
}

DWORD Call_thiscall( const void* args, size_t sz,
                    void* object, DWORD func )
{
    DWORD rc;                // here's our return value...
    __asm
    {
        mov    ecx, sz        // get size of buffer
        mov    esi, args      // get buffer
        sub    esp, ecx       // allocate stack space
        mov    edi, esp       // start of destination stack frame
        shr    ecx, 2         // make it dwords
        rep    movsd          // copy it
        mov    ecx, object     // set "this"
        call   [func]         // call the function

        mov    rc, eax        // save the return value
    }
    return ( rc );
}

DWORD Call_thiscallvararg( const void* args, size_t sz,
                          void* object, DWORD func )
{
    DWORD rc;                // here's our return value...
    __asm
    {
        mov    ecx, sz        // get size of buffer
        mov    esi, args      // get buffer
        sub    esp, ecx       // allocate stack space
        mov    edi, esp       // start of destination stack frame
        shr    ecx, 2         // make it dwords
        rep    movsd          // copy it
        mov    ecx, object     // set "this"
        push   ecx            // push it on the stack
        call   [func]         // call the function
    }
}

```

```

        mov    rc,  eax    // save the return value
        mov    eax,  sz    // ready to restore stack pointer
        add    eax,  4     // pushed ecx too
        add    esp,  eax   // restore the stack pointer
    }
    return ( rc );
}

__declspec ( naked ) DWORD GetST0( void )
{
    DWORD f;                // temp var
    __asm
    {
        fstp dword ptr [f]    // pop ST0 into f
        mov  eax, dword ptr [f] // copy into eax
        ret                                // done
    }
}

```

Now we can truly call a function in an almost completely generic way. This is assuming that we have the function database all set up, of course. Now it's just a matter of our client systems looking up the functions they need to call in the database and passing in properly organized parameters.

## Calling the Function

Before anybody starts calling any functions, we need to figure out *which* ones to call. This is the job of the client subsystem – it will query the function database and look up the particular function it's trying to call based on some key. For the scripting engine, this key will be the function name. Say it's parsing code like this:

```
Baz( 1, 2.5, "Hello" );
```

The parser will recognize a function call and three parameters – one integer constant, one float constant, and a string literal. The compiler or interpreter will then look up “Baz” in the function database. If it finds the entry, then it must make sure the parameters are the proper types, and output p-code to convert them if necessary, giving an error or warning if there are any problems. The `Baz` function can be resolved to its serial ID (index or primary key within the database) for storage efficiency. Then, when executing the script code, the virtual machine's “call” instruction can look up the function by its serial ID again and retrieve the function specification. Our parameters are already on the VM's virtual stack, which we have thoughtfully made `DWORD`-aligned and pushed from left to right (going up in memory), so the virtual stack matches exactly what the calling conventions require. Then it is a simple matter to just take the address of the start of the parameters and pass it into the proper utility function (`Call_cdecl()` etc). At runtime, the virtual machine will call an exported function in a highly efficient way.

For our net pipe, looking up the `Function` instance is a little more complicated. Earlier we decided to make the code automatically intercept function calls meant for the local machine and route them through the network instead. We can do this by inserting a little bit of code

inside the function that we are making networkable. Get the current instruction pointer (`eip`) and use that as a fuzzy key into the function database. In other words, look in `g_Functions` for the entry with the highest `m_Proc` value that is less than the current CPU instruction pointer (`eip`). Here is an example:

```

__declspec ( naked ) DWORD GetEIP( void )
{
    __asm
    {
        mov eax, dword ptr [esp]
        ret
    }
}

const int LOCAL_MACHINE_ID = -1;

#define FUBI_RPC_CALL( addr ) \
    { \
        static const Function* s_Function = FindFunction( GetEIP() ); \
        if ( addr != LOCAL_MACHINE_ID ) \
        { \
            RouteFunction( addr, s_Function, (BYTE*)&addr + 4 ); \
            return; \
        } \
    }

void NetBaz( DWORD remoteAddress, int i, float f, const char* s )
{
    FUBI_RPC_CALL( remoteAddress );

    printf( "i = %d, f = %f, s = %s\n", i, f, s );
}

```

In the above code, we have made a `NetBaz` function that is capable of being called remotely, depending on the `remoteAddress` passed in, which may be set to the constant `LOCAL_MACHINE_ID` if meant to run locally. Converting a function to be networkable is as simple as inserting the `FUBI_RPC_CALL` macro at the top of it! This macro relies on three functions. First, it needs to get the instruction pointer (`eip`) for the query, which the `GetEIP()` function provides. Second, the fuzzy lookup in the function database using this instruction pointer is done by a function called `FindFunction()` (easy to write, not provided here). And finally, assuming that the function call is meant to go over the network, we call a third function called `RouteFunction()` (this is part of the net pipe) which simply packs up the parameters it has been passed based on the function specification and sends them over the network along with the serial ID. Note that the function spec is only looked up once because it is cached into a `static`. It will not change during the session and as such requires no further lookups, so this is an easy and useful optimization to do. The resulting RPC is highly efficient and speedy – the parameters can be `memcpy`'d out to the outgoing packet buffer.

On the other end, a `DispatchFunction()` method would look something like this:

```

void DispatchFunction( int serialID, const void* params )
{
    const Function* function = g_Functions[ serialID ];
    if ( function->m_CallType == Function::CALL_CDECL )
    {
        Call_cdecl( params, function->m_ParamTypes.size() * 4,
                    function->m_Proc );
    }
    // ...other calling convention handlers here...
}

```

For illustrative purposes, this function is vastly simplified from the full and safe implementation it should be. It does not do any parameter processing to handle strings and other pointers being passed across the network. Real implementations of both `RouteFunction()` and `DispatchFunction()` would analyze the `m_ParamTypes` of the function specification and post process the data. Strings and other simple memory block types could get tacked on to the end of the parameter stream. Pointers to user-defined types could be converted into network cookies and back via published callbacks installed into FuBi. In the dispatcher, the embedded pointers in the parameter stream would be reset to point to the tacked on data already retrieved in the packet.

Now that we have the `Function` instance and our parameter buffer, we can use `m_CallType` to choose and call the correct blind call utility function from the previous section. Pass in the parameter buffer, its size, optionally the object if a member function call, and `m_Proc`. Then we can either use the return value directly or call `GetST0()` to get it if `m_ReturnType` is a `float` or `double`. And that's all there is to calling a function generically!

## Completing the Solution

Up until now we've been assuming that the `g_Functions` database has already been set up. Let's finally go back and fill in this gaping hole so we will have a complete solution. One simple way to fill the database is the hard coded method:

```

float Baz( int, const char* );
int Tel( void );

void SetupFunctionExports( void )
{
    {
        Function function;

        function.m_Name      = "Baz";
        function.m_Proc      = Baz;
        function.m_ReturnType = VAR_FLOAT;
        function.m_ParamTypes . push_back( VAR_INT );
        function.m_ParamTypes . push_back( VAR_STRING );
        function.m_CallType  = CALL_CDECL;

        g_Functions.push_back( function );
    }
}

```

```

Function function;

function.m_Name      = "Tel";
function.m_Proc      = Tel;
function.m_ReturnType = VAR_INT;
function.m_CallType  = CALL_CDECL;

g_Functions.push_back( function );
}
}

```

This example is not exactly optimized, but it demonstrates what we need done. We can improve it a lot of ways using helper functions, templates, and macros to make it easier to add new functions to the table. We can break out the setup routine so that different parts of the game may install their own exported functions into the database. These are all nice improvements, but overall this hard coded method will always be completely unsafe and inconvenient. In order to add a new function to the table, someone has to write some code that specifies its types, name, and calling convention and takes its address. These things aren't always obvious and are easy to get wrong (*stdcall* instead of *cdecl*?), so this process is prone to error, not to mention time-consuming. Independently making changes to the declarations of exported functions without updating the table could introduce some nasty and hard to debug problems. It's going to be too much work to keep the function specifications in sync with the actual function prototypes.

We need a way to build this table automatically in order to eliminate these problems. This has been the unattainable goal so far with our early attempts in the beginning of this paper. Now that we have this generic function database system, some options open up. It turns out that the C++ compiler already has all the information we need. While parsing the function's prototype, the compiler builds an internal representation of the function very similar to our `Function` entry. Unfortunately, we don't have access to this information from within the code. We could probably find a way to use the PDB (debug symbols database) to query for what we need, but we can't ship debug symbols with the game. Besides, we wouldn't have an easy way to tell which functions are for export and which aren't.

Speaking of exports, why not use DLL exports? The `__declspec( dllexport )` tag tells the linker to store the names and addresses of our functions in an export table. The export table is a standard component of the Win32 Portable Executable file format. This is normally used by DLL's so that other modules can import their functions, but we can use this on our game's EXE in a way that Microsoft's engineers probably never intended. FuBi can walk its own export table upon startup, which will give it the names and addresses of all of the exported functions in the game. Now this is the automation we're looking for!

Now what we need to do is get the types of the parameters and return value, and the calling convention of each of these functions. We could have a separate tool scan the codebase and extract this information directly from the source code, compile it into a database, and have the game read that at startup. Not a bad solution, but it would be a lot of trouble and would introduce potential synchronization problems between the database and the EXE. We're so close to a fully automated solution, we shouldn't stop there. Fortunately, we don't have to. C++

name mangling fills in this final critical link and gives us our complete end-to-end automated solution.

Name mangling is a C++ feature typically implemented to support type safety and overloaded name resolution. Mangled names are encoded with all the information we require – the types of each parameter and return value, whether or not the function is a class method, whether it's static or not, what its calling convention is, etc. Everything we need is there and more. All we need to do is unmangle the names into a form we can understand and then use that to build the `Function` entry to add to `g_Functions`. Then we're done! Not surprisingly, the name-mangling format is completely implementation specific, undocumented, and even changes from release to release of Visual C++. It would be a Bad Idea to attempt to reverse-engineer it. Happily, this is also completely unnecessary – Microsoft exported a function from `DbgHelp.dll` that does exactly what we need: `UnDecorateSymbolName()`. Here are the various forms of our `Baz()` function from the last sample, if it were to be DLL-exported, and subsequently imported and unmangled by FuBi at runtime:

```
// function declaration
float Baz( int, const char* );

// mangled
?Baz@@@YAMHPBD@Z

// unmangled
float __cdecl Baz(int,char const *)
```

The final unmangled text can be easily parsed and converted to a `Function` entry for addition to our `g_Functions` database. We finally made it!

So now our procedure for filling the `g_Functions` database is:

1. Iterate over all entries in the EXE's export table, and retrieve each function's address and mangled name.
2. Unmangle each name to get a function prototype in text form.
3. Parse the function prototype to retrieve name, class, type and calling convention information.
4. Store the results in a new entry within `g_Functions`.
5. Repeat for each export entry.

The export table is a custom binary format that's documented on the Microsoft Developer Network Library (<http://msdn.microsoft.com>). Search for the `.edata` section within the library entry for the "Microsoft Portable Executable and Common Object File Format Specification". This will discuss in detail the structure of the binary data and can be used to build the export table iterator.

There is one minor issue with the export table: its entries don't actually point to your exported functions. Instead, they point to a jump table, which in turn points to the actual functions. This isn't important if you're only interested in binding to functions and calling them generically, as is required by a scripting engine. However, if you need to be able to do a reverse lookup and use the instruction pointer from within the called function to find its `Function` instance (required for RPC's as described earlier), you'll need to get the actual address of the function for comparison, not the address of the entry in the jump table. This is easy enough: dereference the address given by the DLL export entry to find the jump table entry. The first byte will be `0xE9` (`jmp`), followed by a 4-byte offset to the actual entry point of your function. We can emulate the `jmp` function to find the target address. Take the address given by the DLL export entry, add 5 for the full `jmp` instruction, add the 4-byte offset, and this will be the address of the entry point of your function. This can then be used for reverse lookup to find the `Function` instance from within `g_Functions` as described previously.

Now we're more or less done. There are a few minor wrinkles to worry about though. First of all, this technique does not work with virtual functions. Unfortunately there is no way to get access to the exact order and makeup of a v-table without using the PDB file. So when a script calls an exported virtual function, it will not actually be able to call the proper virtual function based on the dynamic type of the object. Even though we know the function is virtual and can get at the object's v-table, we can't do anything useful with it. Instead, that specific function will be called, regardless of the object's dynamic type. This can be worked around by exporting a separate wrapper function that takes a pointer to the object followed by the parameters and calls the virtual function. This will let the C++ compiler do the work of looking up the proper function using the v-table. It's not pretty, but the scripting language can hide this pretty easily and make it appear that the function is really being called virtually, like magic.

Compatibility across EXE's is another concern. Depending on how you map serial ID's to functions, you may have a synchronization problem across EXE's, if a later version of the game has a new function in the middle which offsets all the rest. A simple way to cut down on problems from this would be to use a CRC of the mangled name as the serial ID, though it will make the database lookup by serial ID more expensive (though this is probably not a large concern).

## End Game

We now have everything we need to call functions in a completely generic way. In order to publish a function in the system and allow other subsystems such as scripting and network RPC's to bind to it, we simply tag it with `__declspec( dllexport )`. This verbose tag is best wrapped in a macro to reduce clutter. Here is are two sample functions: one plain, and one set up for RPC's.

```
#define FUBI_EXPORT __declspec( dllexport )

FUBI_EXPORT void Baz( int i, const char* str )
{
    // ...
}
```

```

FUBI_EXPORT void Tel( DWORD remoteAddress, int i, const char* str )
{
    FUBI_RPC_CALL( remoteAddress );

    // ...
}

```

At runtime FuBi will iterate over the Win32 export table, and extract name, type and calling convention information from each entry. Other subsystems can look up functions by memory address, name, or serial ID and call them generically using one of the blind call utility functions.

By now you may be wondering, “why go to all this trouble?” It may seem like quite a bit more work to implement than necessary, like it’s complicating what originally may have seemed like a simple problem to solve. For smaller projects with small export sets it probably is. FuBi is really meant for larger projects, which are changing constantly. Exporting functions to the system is so easy that it actually changes the development process. It doesn’t take that long to write a system like this – the original FuBi prototype took about a week, and it was usable enough with no further modifications. A game with FuBi, a primitive command language, and a console for command entry can very quickly turn into a powerful self-aware system. A function like this can be written in five seconds:

```

FUBI_EXPORT void TestMySpecialFunction( void )
{
    CallSomeFunkyStuff();
    DrawSomeFunkyDebugStuff();
}

```

This avoids the tedious process of writing a test function, binding it to a debug key or menu item, and then executing it. Instead, just write a pile of functions, `FUBI_EXPORT` them, and then call them up from the console. Create functions to tweak variables or dump debug data. It’s so easy to export functions that development functions will proliferate quickly. All of this is on top of the actual purpose for which FuBi exists: to bind functions for scripting and networking.

## Further Work

This paper, due to limitations on length, has only focused on the function binding aspects of FuBi, but in actual implementation there is so much more. The FuBi in Dungeon Siege has evolved rapidly; today it is a general purpose high-performance runtime symbol engine. It gives any module that it is linked to a measure of “self-awareness” via knowledge of its own data structures and functions. It has found uses in content management, network security, object persistence, statistics storage, and even a new form of runtime assertion (automatic code flow assertions).

As “exercises for the reader”, here are some areas that can be easily enhanced to add a lot of value to FuBi without much effort. Some of these things directly benefit scripting engines, and others benefit networking.

- Bind arbitrary information to functions

This is FuBi in reverse. Add documentation, security and usage flags, function set memberships, and other traits on a per-function basis. This can be done by exporting a separate function with the same name as the function, but adding a special prefix that FuBi can watch for and extract into a separate collection.

Many useful things can be done through this technique. In Dungeon Siege, this is how pointers become network cookies and back – these are tagged through specially prefixed FuBi exports that are collected separately.

- Add knowledge of singletons

When calling member functions of singleton objects over the network, there's no need to use a network cookie if everybody only has one of them. Tell FuBi about your singletons and it can bypass the conversion and more or less treat these functions as globals (though it still must use `thiscall` of course).

- Analyze RPC flow

RPC's can be logged very easily with FuBi, because it can dump the names and parameters of every function call. It knows the types of the parameters it is dumping and can do string conversions easily using this information so the log entries are completely readable. The analyzer could also watch for nested RPC's very easily, which would help track potential performance hotspots. Dump the results to an XML file for analysis by external tools.

- Tell FuBi about your Plain Ol' Data (POD) types

Install the names of your POD types into FuBi so that its RPC packer can treat POD parameters you are passing by value or reference as memory blocks. This is very useful for, say, passing a `vector_3` over the network, or a `RECT`, etc. Also allow passing of blocks of memory over the network. Create a simple `struct` that contains just a `void*` and a `size_t`. FuBi can detect this type being passed and do the necessary appending of data to the end of the packet.

This technique of creating special types that FuBi can handle with custom code is very powerful. It can be used in many other ways. Essentially what we're talking about here is a general-purpose automated parameter transform utility. Let client code install "drivers" for its special types and see what is possible!

- Support adding other DLL's on the fly

There is nothing about this system that says it has to work with a single EXE. You could have FuBi import functions and types from multiple DLL's, even swapping them in and out on the fly. Then you wouldn't even have to restart the game to insert new functions into the system.

Note that this will require a slight change to our `FUBI_RPC_CALL` macro. Rebinding functions will invalidate our static caches, so to counter this, keep a list of the addresses of these statics. When the caches become invalid due to rebinding, loop through the list and `NULL` them out again.

- It's not just for game engine exports!

Why limit this to just game code? This system can work with any DLL that has exports, even system DLL's. The problem with system DLL's is that they are usually standard C exports and do not provide the name-mangling that we require. However, as they are constant, you could create a tool to scan `windows.h` and other header files and extract the unmangled function prototypes, converting them into a type library. Then FuBi could use this to build its parameter and calling convention specs. This would add system call support to your scripting language. Perhaps you could teach FuBi to speak COM, in which case you can use existing type libraries and IDL files to build the function database.

The future is bright for FuBi. More debugging help, more database queries, more tool support is all on the way. Game development is on a collision course with enterprise technologies such as databases, the web, XML, and some advanced dev environments that are coming soon. With complete knowledge of the game's types, FuBi will be able to act as a communications channel between these powerful external tools and the game's dynamic systems.

And of course it will still continue to bind functions.

## References

Sample source code is provided on my website. This includes a fully commented, working demo of a simple command processor using FuBi exports and a sample RPC system built on top of FuBi. This system will meet the requirements stated in this paper.