# FuBi: Automatic Function Exporting for Scripting and Networking

# Code Supplement

By Scott Bilas

## Abstract

In this paper I present and discuss code used in Dungeon Siege's FuBi. I had planned to write a fully functional sample application to demonstrate FuBi in action for scripting and networking (ready for GDC 2001 no less!), but I just don't have time given the title I'm working on and the looming ship date.

The next best thing is to just upload a bunch of the code that I've written for Dungeon Siege, and that's what this paper is about. Many of the important components of the system are included and discussed here. Unfortunately this means that this paper is far more complicated than my sample would have been, so hopefully you can follow it through all the extra clutter. If not, I'm always available for help via email. I'm putting this together in a big hurry as I find spare moments, so please bear with my poor writing and scatterbrained organization.

Important: this paper isn't going to make a whole lot of sense without reading the FuBi paper on my web site first. Or you could start with my gem "A Generic Function Binding Interface", also on my web site.

## Utilities

First we need to cover some of the utilities I use globally in this code. This pair of macros is meant to be used in an enum. Use the first to start an enum and set some special constants for the given start number, and the second to get the count and end enum (for iteration).

```
#define SET_BEGIN_ENUM( x, num )          \
    x##BEFORE_BEGIN = num - 1
#define SET_END_ENUM( x )                 \
    x##END,                               \
    x##BEGIN = x##BEFORE_BEGIN + 1,       \
    x##LAST  = x##END - 1,                \
    x##COUNT = x##END - x##BEGIN
```

A simple macro to wrap debug-only code:

```
#ifdef _DEBUG
#   define DEBUG_ONLY( f ) f
#else
#   define DEBUG_ONLY( f )
#endif
```

A nice helper to zero something out:

```
template <typename T> inline void ZeroObject( T& object )
    {  memset( &object, 0, sizeof( T ) );  }
```

This function checks to see if a pointer is pointing to “bad food”. Debug versions of memory managers typically use magic numbers like these for pre- and post-fill on memory blocks.

```
inline bool IsMemoryBadFood( DWORD d )
{
    return (    ( d == 0xDDDDDDDD )     // crt: dead land (deleted objects)
             || ( d == 0xCDCDCDCD )     // crt: clean land (new, uninit'd objects)
             || ( d == 0xFDFDFDFD )     // crt: no man's land (off the end)
             || ( d == 0xCCCCCCCC )     // vc++: stack objects init'd with this
             || ( d == 0xFEEEFEEE )     // ? nt internal ?
             || ( d == 0xBAADF00D ) );  // winnt: nt internal "not yours" filler
}
```

Here is a pair of classes that I use to wrap up the concept of a memory block – base pointer plus size. It’s safer than simply using a separate pointer and size parameter because the code can specifically look for this type being passed by value and adjust accordingly. Note that the class makes sure it doesn’t point to bad food as a simple safety, but doesn’t do anything more complicated than that (like calling *IsBad[Read|Write]Ptr*).

```
struct mem_ptr
{
    typedef void* MemType;

    void*  mem;         // pointer to start of memory
    size_t size;        // size of memory

    mem_ptr( void )               {  mem = NULL;  size = 0;
                                     DEBUG_ONLY( assert_valid() );  }
    mem_ptr( void* m, size_t s )  {  mem = m;  size = s;
                                     DEBUG_ONLY( assert_valid() );  }

    operator void* ( void ) const  {  return ( mem  );  }
    operator size_t( void ) const  {  return ( size );  }

    void assert_valid( void ) const
    {
        gpassert( !IsMemoryBadFood( (DWORD)mem ) );
    }
};

template <typename T> inline mem_ptr make_mem_ptr( T& obj )
    {  return ( mem_ptr( &obj, sizeof( T ) ) );  }

struct const_mem_ptr
{
    typedef const void* MemType;

    const void* mem;    // pointer to start of memory
    size_t      size;   // size of memory

    const_mem_ptr( void )                      {  mem = NULL;  size = 0;  }
    const_mem_ptr( const void* m, size_t s )   {  mem = m;  size = s;
                                                  DEBUG_ONLY( assert_valid() );  }
    const_mem_ptr( mem_ptr p )                 {  mem = p;  size = p;
                                                  DEBUG_ONLY( assert_valid() );  }

    operator const void* ( void ) const  {  return ( mem  );  }
```

```
    operator size_t      ( void ) const  {  return ( size );  }

    void assert_valid( void ) const
    {
        gpassert( !IsMemoryBadFood( (DWORD)mem ) );
        gpassert( size < 1024 * 1024 * 1024 );
    }
};

template <typename T> inline const_mem_ptr make_const_mem_ptr( const T& obj )
    {  return ( const_mem_ptr( &obj, sizeof( T ) ) );  }
```

Damn C++ stupidly long keywords…

```
#define rcast reinterpret_cast
#define dcast dynamic_cast
#define scast static_cast
#define ccast const_cast
```

These macros are used to automatically define operators for enum types. I like to use certain enums as integers or bitfields and this makes it easier.

```
#define MAKE_ENUM_BIT_OPERATORS( x ) \
    inline x operator | ( x a, x b ) \
        {  return ( scast <x> ( scast <int> ( a ) | scast <int> ( b ) ) ); } \
    inline x operator & ( x a, x b ) \
        {  return ( scast <x> ( scast <int> ( a ) & scast <int> ( b ) ) ); } \
    inline x operator |= ( x& a, x b ) \
        {  return ( a = a | b ); } \
    inline x operator &= ( x& a, x b ) \
        {  return ( a = a & b ); } \
    inline x NOT( x a ) /* can't figure out how to create an operator ~() */ \
        {  return ( scast <x> ( ~(scast <int> ( a )) ) ); }

#define MAKE_ENUM_MATH_OPERATORS( x ) \
    inline x operator ++ ( x& a )                       /* prefix increment */ \
        {  return ( a = scast <x> ( a + 1 ) );  } \
    inline x operator ++ ( x& a, int )                  /* postfix increment */ \
        {  x old = a;  ++a;  return ( old );  } \
    inline x operator -- ( x& a )                       /* prefix decrement */ \
        {  return ( a = scast <x> ( a - 1 ) );  } \
    inline x operator -- ( x& a, int )                  /* postfix decrement */ \
        {  x old = a;  --a;  return ( old );  } \
    inline x operator + ( x a, x b ) \
        {  return ( scast <x> ( scast <int> ( a ) + scast <int> ( b ) ) ); } \
    inline x operator - ( x a, x b ) \
        {  return ( scast <x> ( scast <int> ( a ) - scast <int> ( b ) ) ); } \
    inline x operator += ( x& a, x b ) \
        {  return ( a = scast <x> ( a + b ) ); } \
    inline x operator -= ( x& a, x b ) \
        {  return ( a = scast <x> ( a - b ) ); }
```

## Simple Macros

This is the basic FuBi export macro. As in the paper, simply *dllexport* your functions by putting this in front of the function prototype.

```
#define FEX __declspec ( dllexport )
```

In our function importer we will look for functions prefixed with *FUBI_* and treat them specially (this is the “protocol” I alluded to in the paper). These three macros set this up. Functions that we intend to be “reserved” we will wrap in the *FUBI_RESERVED* macro.

```
#define FEX_PREFIX FUBI_
#define FEX_PREFIX_TEXT #FEX_PREFIX
#define FUBI_RESERVED( NAME ) FEX_PREFIX##NAME
```

This macro is used to tag a class as a singleton, and classes that use this macro must have a static *GetSingleton()* function. See my gem in *Game Programming Gems I* called “An Automatic Singleton Utility” for an easy singleton class (also available on my web page). In the importer we will reserve a slot in the class’s type entry for a function pointer to its “get my singleton” method.

```
#define FUBI_SINGLETON_CLASS( T ) \
    FEX static T* __cdecl FUBI_RESERVED( GetClassSingleton )( void ) \
        {  return ( &GetSingleton() );  }
```

Simple classes that do not contain pointers or complex members are generally known as POD (plain old data) types. We can tag a class as a POD type by putting this macro inside of it. If a class exports this function, we will set the POD flag in the class and set its size as well. Any RPC (remote procedure call) code we have will need to pay attention to this flag and not pre/post-process accordingly.

```
#define FUBI_POD_CLASS( T ) \
    FEX static size_t __cdecl FUBI_RESERVED( PodGetSize )( void ) \
        {  return ( sizeof( T ) );  }
```

Here is how I tell a class that it is RPC capable via network cookie conversion. Put this at the bottom of your class declaration, passing in the name of the class and the names of functions that convert class pointers to/from network cookies. Note that this is unnecessary with POD types and singletons for obvious reasons. Typically this macro is used in classes that have ID’s that are globally consistent across machines. In the *TO_FUNC* function (which must take a *T** and return a *DWORD* which is the network cookie) simply take the pointer and ask the class for its ID. In the *FROM_FUNC* function (which must take a *DWORD* network cookie and return a *T**) look up the ID in the necessary database to resolve to a pointer.

```
#define FUBI_RPC_CLASS( T, TO_FUNC, FROM_FUNC ) \
    FEX static DWORD __cdecl FUBI_RESERVED( InstanceToNet )( T* instance ) \
        {  return ( TO_FUNC( instance ) );  } \
    FEX static T* __cdecl FUBI_RESERVED( NetToInstance )( DWORD cookie ) \
        {  return ( FROM_FUNC( cookie ) );  }
```

It just so happens that my scripting language (Siege)Skrit needed to know about inheritance of system types so it could have a pointer to a base and call a base function on it. This macro tells FuBi (a) when a class is derived from another and (b) what to offset the pointer by when casting to base. Detection of the function will give us the inheritance relationship, and calling the function will give us the offset.

```
#define FUBI_CLASS_INHERIT( T, BASE ) \
    FEX static int __cdecl FUBI_RESERVED( Inheritance )( BASE* ) \
            {  return ( (int)(BASE*)(T*)1 - (int)(T*)1 );  }
```

Now we need some constants. These are used internally in FuBi and in the network transport as virtual machine addresses. In Dungeon Siege, *RPC_TO_LOCAL* and *RPC_TO_SERVER* must resolve to actual machine ID's for performance reasons, so they are not exactly "const". Helper functions allow setting these after the network is up and running.

```
// $ this does not correspond to a DPID
const DWORD RPC_INVALID_ADDR        = (DWORD)( -1 );
// == DPNID_ALL_PLAYERS_GROUP
const DWORD RPC_TO_ALL              = (DWORD)(  0 );
// $ this does not correspond to a DPID
const DWORD RPC_TO_OTHERS           = (DWORD)( -2 );
// by default, each machine starts up as the server...
const DWORD RPC_TO_SERVER_DEFAULT   = (DWORD)( -3 );
// when joining an mp game, actual server changes the ID and gets the authority
const DWORD RPC_TO_LOCAL_DEFAULT    = (DWORD)( -3 );
// just a nice starting point for a non-dplay test app to assign machine id's
const DWORD RPC_TEST_ADDR_START     = (DWORD)(  1 );

// "constants" - treat as constant, modify ONLY by net pipe
extern const DWORD& RPC_TO_LOCAL;
extern const DWORD& RPC_TO_SERVER;
```

## Advanced Macros

The RPC macro is a little crazy, and nested to make it easier to manage. As with all parts of FuBi, this macro was vastly simpler when originally written, but the needs of the project changed that over time and these macros just got more and more things added.

For example, I added the "embedded tag" work when I found out that detecting that a function is not RPC-capable is something that really needs to happen at startup time, rather than just-in-time. Because an exported function may be used for the scripting language or another purpose, you can't assume a function which would fail to work for an RPC is actually intended to be used as an RPC (and give a bogus error). The only way to know for sure is to check to see if an RPC macro is used inside the function, and towards this end I wrote a scanner that would search for a special embedded tag right inside a function. If this tag was found, it's safe to assume that the function is intended to be used for RPC's, and then we can verify it at startup time to make sure it's not passing weird pointers over the network etc.

Another enhancement came when Bart's network layer came online and added the concept of "retries" (Bartosz Kijanka is our lead engineer). Dungeon Siege networking provides the usual guaranteed delivery stuff but adds a new concept called "guaranteed execution". In our continuous world engine, just because a packet gets through doesn't mean it can actually execute just yet. The object that a message is expected to operate on may not have entered the world yet. This is caused by differences in where the client and server think the world frustum center is, and is usually caused by lag or client machines being significantly slower than the server. While the client is catching up, processing of those packets must be deferred. In order to make this work, certain RPC's are tagged as "retry" exports, which the receiver will periodically retry. The return value is a "retry cookie" that is used by the network transport to decide whether or not it succeeded or failed, or needs retrying. This cookie is detected by the FuBi importer and is handled specially by the dispatcher.

Anyway here are the macros. The last four are the important ones, and everything else is support. The *FUBI_RPC_CALL_IMPL* macro does all the real work. One of its critical roles is to look up the function that it's in. As in the paper, this is done by looking up *eip* in the function database to see which one we're in. Note that it is necessary to pass in the name of the function for RPC lookup because the linker will collapse multiple functions with identical code down to the same entry point. This is common with empty or other stub type functions. The name is necessary to differentiate the proper function.

```
// helper for param skipping - 4 bytes each
#define FUBI_PARAM_SKIP( COUNT ) \
    paramStart = rcast <BYTE*> ( paramStart ) + ((COUNT) * 4);
#define FUBI_NO_PARAM_SKIP \
    ;

// how far to search for RPC tag - this is actually 0x27 or so but give a
// little bit more for safety.
const int FUBI_RPC_MAX_SEARCH = 0x40;

// divider
#define FUBI_EMBED_TAG_MOV 0xB8      /* mov eax, [dword constant] */

// basic embedded tags
#define FUBI_EMBED_TAG_0 0x46        /* 'F' */
#define FUBI_EMBED_TAG_1 0x75        /* 'u' */
#define FUBI_EMBED_TAG_2 0x62        /* 'b' */
#define FUBI_EMBED_TAG_3 0x69        /* 'i' */

// special RPC tags
#define FUBI_RPC_TAG_0 0x46          /* 'F' */
#define FUBI_RPC_TAG_1 0x52          /* 'R' */
#define FUBI_RPC_TAG_2 0x70          /* 'p' */
#define FUBI_RPC_TAG_3 0x63          /* 'c' */

// the full tag
extern const BYTE FUBI_EMBEDDED_RPC_TAG[];

// helper for embedded tagging - include initial MOVs to keep the
// disassembler from getting confused. sequence is 0xB8 'Fubi' 0xB8 'abcd'
// in reverse byte order.
#define FUBI_EMBED_TAG( a, b, c, d )                                    \
    {                                                                   \
        __asm jmp $+15                                                  \
        __asm _emit FUBI_EMBED_TAG_MOV                                  \
        __asm _emit FUBI_EMBED_TAG_3                                    \
        __asm _emit FUBI_EMBED_TAG_1                                    \
        __asm _emit FUBI_EMBED_TAG_2                                    \
        __asm _emit FUBI_EMBED_TAG_0                                    \
        __asm _emit FUBI_EMBED_TAG_MOV                                  \
        __asm _emit d                                                   \
        __asm _emit c                                                   \
        __asm _emit b                                                   \
        __asm _emit a                                                   \
    }

// helper for "which function am i?" code - puts it into s_FunctionSpec
#define FUBI_FIND_FUNCTION( FUNC_NAME, RESOLVE_PROC, PARAM_SKIP )       \
    /* get the n'th parameter of the function */                        \
    void* paramStart;                                                   \
    {                                                                   \
        /* this is the start of the stack frame, skipping the old ebp */ \
        /* and return addr */                                           \
```

```
        __asm mov eax, ebp                                                  \
        __asm add eax, 8                                                    \
        __asm mov paramStart, eax                                           \
    }                                                                       \
    PARAM_SKIP;                                                             \
                                                                            \
    /* get the correct function spec and cache it */                        \
    static const FunctionSpec* s_FunctionSpec = RESOLVE_PROC( GetEIP(),     \
            FUNC_NAME, ELEMENT_COUNT( FUNC_NAME ) - 1 );                    \
    gpassert( s_FunctionSpec != NULL );

// special tag to say "this function is an RPC" - 'FRpc'
#define FUBI_RPC_TAG()                                                      \
    FUBI_EMBED_TAG( FUBI_RPC_TAG_0, FUBI_RPC_TAG_1,                         \
                    FUBI_RPC_TAG_2, FUBI_RPC_TAG_3 );                       \
    FuBi::AutoRpcTagBase FUBI_AutoRpcTagBase;                               \
    FuBi::AutoRpcTagBase* FUBI_AutoRpcTagBasePtr = &FUBI_AutoRpcTagBase

// implementation for an RPC call
#define FUBI_RPC_CALL_IMPL( FUNC_NAME, THIS_PARAM, RPC_ADDRESS, RETURN )    \
    FUBI_EMBED_TAG( FUBI_RPC_TAG_0, FUBI_RPC_TAG_1,                         \
                    FUBI_RPC_TAG_2, FUBI_RPC_TAG_3 );                       \
    FuBi::AutoRpcTag FUBI_AutoRpcTag( FUBI_AutoRpcTagBasePtr );             \
    {                                                                       \
        using namespace FuBi;                                               \
                                                                            \
        /* get function spec */                                             \
        FUBI_FIND_FUNCTION( FUNC_NAME, ResolveRpc, FUBI_NO_PARAM_SKIP );    \
                                                                            \
        /* send RPC */                                                      \
        DWORD rpcAddress = RPC_ADDRESS;                                     \
        if ( FuBi::RpcTestMacro( rpcAddress, s_FunctionSpec ) )             \
        {                                                                   \
            Cookie cookie = RPC_FAILURE;                                    \
            if ( s_FunctionSpec != NULL )                                   \
            {                                                               \
                cookie = SendRpc( s_FunctionSpec, (void*)(THIS_PARAM),      \
                                  paramStart, rpcAddress );                 \
            }                                                               \
            if ( rpcAddress != RPC_TO_ALL )                                 \
            {                                                               \
                RETURN;                                                     \
            }                                                               \
        }                                                                   \
    }

// return helpers
#define FUBI_RPC_RETURN \
    gpassert( cookie == RPC_SUCCESS ); return
#define FUBI_RPC_RETRY_RETURN \
    return ( cookie )

// non-retrying RPC calls
#define FUBI_RPC_CALL( FUNC_NAME, RPC_ADDRESS ) \
    FUBI_RPC_CALL_IMPL( #FUNC_NAME, NULL, RPC_ADDRESS, FUBI_RPC_RETURN )
#define FUBI_RPC_THIS_CALL( FUNC_NAME, RPC_ADDRESS ) \
    FUBI_RPC_CALL_IMPL( #FUNC_NAME, this, RPC_ADDRESS, FUBI_RPC_RETURN )

// retrying RPC calls
#define FUBI_RPC_CALL_RETRY( FUNC_NAME, RPC_ADDRESS ) \
    FUBI_RPC_CALL_IMPL( #FUNC_NAME, NULL, RPC_ADDRESS, FUBI_RPC_RETRY_RETURN )
#define FUBI_RPC_THIS_CALL_RETRY( FUNC_NAME, RPC_ADDRESS ) \
    FUBI_RPC_CALL_IMPL( #FUNC_NAME, this, RPC_ADDRESS, FUBI_RPC_RETRY_RETURN )
```

## Type System

This is the type system for FuBi. First we have the `eVarType` enum which is used as an ID for every type that FuBi knows about. All the basic C types are included by default in here, along with some basic mappings to common typedefs for convenience. Next we have a set of "special" types that are handled with custom code for scripting and networking. After that is a set of reserved ID's for enumerated types. It's convenient to separate these types from other user types because in most cases they can be treated simply as integers, with only occasionally needing to worry about their actual types for string conversions and type matching. And lastly we have a reserved range for all user-defined types. As we import symbols that we don't recognize, Note that this enum is treated as "open", and new ID's will be assigned in FuBi as it detects types.

```
enum eVarType
{
    VAR_UNKNOWN = -1,

    // standard types

    SET_BEGIN_ENUM( VAR_, 0 ),

    VAR_SCHAR,                      // signed char
    VAR_UCHAR,                      // unsigned char
    VAR_SHORT,                      // short
    VAR_USHORT,                     // unsigned short
    VAR_INT,                        // int
    VAR_UINT,                       // unsigned int
    VAR_INT64,                      // int64
    VAR_UINT64,                     // unsigned int64
    VAR_FLOAT,                      // float
    VAR_DOUBLE,                     // double
    VAR_VOID,                       // void
    VAR_BOOL,                       // bool

    SET_END_ENUM( VAR_ ),

    // mapped types - update these if we ever move to win64

#   ifdef _CHAR_UNSIGNED
    VAR_CHAR        = VAR_UCHAR,    // char
#   else
    VAR_CHAR        = VAR_SCHAR,    // char
#   endif

    // ansi/ms
    VAR_SSHORT      = VAR_SHORT,    // signed short
    VAR_SINT        = VAR_INT,      // signed int
    VAR_LONG        = VAR_INT,      // long
    VAR_SLONG       = VAR_LONG,     // signed long
    VAR_ULONG       = VAR_UINT,     // unsigned long
    VAR_INT8        = VAR_SCHAR,    // int8
    VAR_SINT8       = VAR_SCHAR,    // signed int8
    VAR_UINT8       = VAR_UCHAR,    // unsigned int8
    VAR_INT16       = VAR_SSHORT,   // int16
    VAR_SINT16      = VAR_SSHORT,   // signed int16
    VAR_UINT16      = VAR_USHORT,   // unsigned int16
    VAR_INT32       = VAR_SINT,     // int32
    VAR_SINT32      = VAR_SINT,     // signed int32
    VAR_UINT32      = VAR_UINT,     // unsigned int32
    VAR_SINT64      = VAR_INT64,    // signed int64
```

```
    VAR_LONGDOUBLE = VAR_DOUBLE,     // long double - same as double in
                                     // 32-bit code (no longer 80 bit)

    // windows
    VAR_BYTE       = VAR_UCHAR,      // Win32 "BYTE"
    VAR_WORD       = VAR_USHORT,     // Win32 "WORD"
    VAR_DWORD      = VAR_ULONG,      // Win32 "DWORD"

    // these require special processing

    SET_BEGIN_ENUM( VAR_SPECIAL_, 0x1000 ),

    VAR_MEM_PTR,                     // mem_ptr
    VAR_CONST_MEM_PTR,               // const_mem_ptr
    VAR_STRING,                      // std::string
    VAR_WSTRING,                     // std::wstring

    SET_END_ENUM( VAR_SPECIAL_ ),

    // enum - these are enums exported from the game

    VAR_ENUM = 0x2000,               // map types to these on the fly

    // user types - everything else

    VAR_USER = 0x3000,               // map types to these on the fly - they
                                     // correspond to class entries in SysExports

    // special tags
    VAR_ENUM_END = VAR_USER,
};

MAKE_ENUM_MATH_OPERATORS( eVarType );
```

Next we have a `VarTypeSpec`, which is a bridge class to the save game and object management systems in Dungeon Siege. FuBi creates one of these for a user-defined type if it's detected to be a POD type. The implementations of these functions are trivial (just setting members).

```
struct VarTypeSpec
{
    const char*    m_InternalName;      // internal name of type
    const char*    m_ExternalName;      // external name (presented to users)
    int            m_SizeBytes;         // size in bytes
    Trait::eFlags  m_Flags;             // flags for this type
    ToStringProc   m_ToStringProc;      // convert this type to a string
    FromStringProc m_FromStringProc;    // convert this type from a string
    CopyVarProc    m_CopyVarProc;       // make a copy of the type

    VarTypeSpec( void )  {  ::ZeroObject( *this );  }
    VarTypeSpec( const ClassSpec* spec );
    VarTypeSpec( const EnumSpec* spec );
    VarTypeSpec( const char* iname, const char* ename, int size,
                 Trait::eFlags flags, ToStringProc to,
                 FromStringProc from, CopyVarProc copy );
};
```

This next class is used to represent an individual type. If you've ever written a compiler, the first thing you'll notice is how naïve it is. A type that can be *const* and a pointer and that's about it? When I originally designed this system, I had intended it to simply distribute network calls and permit script callbacks. Had I known the degree to which the engine would use it, I would

have used a better type system capable of handling complex types. Also note that I tended to use structs rather than classes for the types here. Originally they were extremely simple nested structs used and managed exclusively by the export system. Since then they have grown to fully “self-aware” types, and should have had their members protected from direct access. This would have saved me some trouble when I made updates to the meanings of these variables. FuBi v2 will fix these oversights.

```
struct TypeSpec
{
    enum eFlags
    {
        FLAG_NONE            =      0,
        FLAG_CONST           = 1 << 0,      // const variable
        FLAG_POINTER         = 1 << 1,      // pointer-to-type
        FLAG_POINTER_POINTER = 1 << 2,      // pointer-to-pointer-to-type (special)
        FLAG_REFERENCE       = 1 << 3,      // reference-to-type
        FLAG_HANDLE          = 1 << 4,      // ResHandle <type>
    };

    enum eCompare
    {
        COMPARE_EQUAL,                  // types are exactly equal
        COMPARE_PROMOTABLE,             // types can be promoted to match
        COMPARE_COMPATIBLE,             // types can be converted to match
        COMPARE_INCOMPATIBLE,           // not compatible without explicit cast
    };

    eVarType m_Type;
    eFlags   m_Flags;

    TypeSpec( void )
        : m_Type( VAR_UNKNOWN ), m_Flags( FLAG_NONE )  {  }
    TypeSpec( eVarType type )
        : m_Type( type ),        m_Flags( FLAG_NONE )  {  }
    TypeSpec( eVarType type, eFlags flags )
        : m_Type( type ),        m_Flags( flags )  {  }

    bool operator == ( eVarType type ) const
        {  return ( (m_Type == type) && (m_Flags == FLAG_NONE) );  }
    bool operator != ( eVarType type ) const
        {  return ( (m_Type != type) || (m_Flags != FLAG_NONE) );  }
    bool operator == ( TypeSpec type ) const
        {  return ( (m_Type == type.m_Type) && (m_Flags == type.m_Flags) );  }
    bool operator != ( TypeSpec type ) const
        {  return ( (m_Type != type.m_Type) || (m_Flags != type.m_Flags) );  }

    bool     IsSimple         ( void ) const;
    bool     IsSimpleInteger  ( void ) const;
    bool     IsSimpleFloat    ( void ) const;
    bool     IsPointerClass   ( void ) const;
    bool     IsPassByValue    ( void ) const;
    bool     IsCString        ( void ) const;
    bool     IsCStringW       ( void ) const;
    bool     IsString         ( void ) const;
    bool     IsStringW        ( void ) const;
    bool     IsAString        ( void ) const;
    bool     IsAStringW       ( void ) const;
    void     SetCString       ( void );
    void     SetString        ( void );
    bool     IsFuBiCookie     ( void ) const;
    bool     IsSingleton      ( void ) const;
    bool     ContentsAreSimple( void ) const;
```

```
    bool     CanRPC           ( DEBUG_ONLY( gpstring* cantRpcReason ) ) const;
    bool     CanSkrit         ( void ) const;
    int      GetSizeBytes     ( void ) const;
    TypeSpec GetBaseType      ( void ) const;
};
```

A *TypeSpec* is simply a dumb type container. Many times we'll want to attach some extra information to a type, such as a name, default value, and possible constraints. This new type is called a *ParamSpec*. Note that I don't bother to show the constraint system here, because it never really got used for much (time constraints again) and didn't get a chance for the design to be really tested well. Also note that I use elements of *ReportSys* here – it's too large a system to include in this doc but it's nothing special, just your standard boring stream abstraction (*iostreams* did not meet my needs).

```
struct ParamSpec
{
    struct Extra
    {
        gpstring           m_Name;           // name of parameter
        gpstring*          m_DefaultValue;   // def value to use (NULL if none)
        bool               m_DefaultIsCode;  // true if def value is actually code
        bool               m_NoXfer;         // don't bother persisting this
#       if !GP_RETAIL
        my ConstraintSpec* m_ConstraintSpec; // constraint func applied to param
#       endif // !GP_RETAIL

        Extra( void );
       ~Extra( void );

        Extra& operator = ( const Extra& other );
    };

    TypeSpec m_Type;
    Extra*   m_Extra;

    ParamSpec( const ParamSpec& other );
    ParamSpec( const TypeSpec& type, const char* name );
    ParamSpec( const TypeSpec& type )
        : m_Type( type ), m_Extra( NULL )  {  }
   ~ParamSpec( void )
        {  delete ( m_Extra );  }

    ParamSpec& operator = ( const ParamSpec& other );

    bool operator == ( TypeSpec type ) const
        {  return ( m_Type == type );  }

    // extra mod
    Extra* GetExtra( void )
        {  BuildExtra();  return ( m_Extra );  }
    void BuildExtra( void )
        {  if ( m_Extra == NULL )  {  m_Extra = new Extra;  }  }

    // attributes
    void SetName( const gpstring& name )
        {  GetExtra()->m_Name = name;  }
    const gpstring& GetName( void ) const
        {  gpassert( m_Extra != NULL );  return ( m_Extra->m_Name );  }
    gpstring GetDefaultValue  ( void ) const;
    void SetDefaultValue( const gpstring& defValue, bool isCode = false );
    void SetNoDefaultValue( void );
```

```
    // utility
    DWORD CalcDigest( void ) const;

    // constraints
#   if !GP_RETAIL
    void                  SetConstraint( ConstraintSpec* spec );
    const ConstraintSpec* GetConstraint( void ) const;
    void                  GenerateDocs ( ReportSys::ContextRef context = NULL,
                                         eDocsLevel level = DOCS_NORMAL ) const;
#   endif // !GP_RETAIL
};
```

Now we're getting to the really serious stuff. Here is the *FunctionSpec* class, which contains all the information necessary to type-check and call an exported function.

```
struct FunctionSpec
{
    enum eFlags
    {
        FLAG_NONE               =       0,

        // $ keep these in sync with FunctionSpecFlags in FuBiDefs.h

        // call types
        FLAG_CALL_CDECL         = 1 <<  0,   // __cdecl
        FLAG_CALL_FASTCALL      = 1 <<  1,   // __fastcall
        FLAG_CALL_STDCALL       = 1 <<  2,   // __stdcall
        FLAG_CALL_THISCALL      = 1 <<  3,   // __thiscall
        FLAG_CALL_MASK          =     0xF,   // use to mask call convention

        // function traits
        FLAG_MEMBER             = 1 <<  4,   // member function of a class
        FLAG_STATIC_MEMBER      = 1 <<  5,   // static member function of a class
        FLAG_CONST              = 1 <<  6,   // member function is const
        FLAG_VARARG             = 1 <<  7,   // variable arg function

        // permissions
        FLAG_HIDDEN             = 1 <<  8,   // hidden from ordinary queries
        FLAG_DOCO               = 1 <<  9,   // doco function - not critical
        FLAG_CHECK_SERVER_ONLY  = 1 << 10,   // only can execute on server
        FLAG_DEV_ONLY           = 1 << 11,   // only allow this in dev builds
        FLAG_RETRY              = 1 << 12,   // this is a "retryable" function

        // misc
        FLAG_POSTPROCESSED      = 1 << 13,   // already postproc'd for doco etc.
        FLAG_SIMPLE_ARGS        = 1 << 14,   // args do not contain ptrs etc.
        FLAG_CAN_RPC            = 1 << 15,   // can be called via RPC
        FLAG_CAN_SKRIT          = 1 << 16,   // can be called from Skrit
        FLAG_SKRIT_IMPORT       = 1 << 17,   // meant for skrit importing
        FLAG_EVENT              = 1 << 18,   // this is a skrit event
        FLAG_TRAIT              = 1 << 19,   // trait-related function

        // membership
        FLAG_MEMBER_OF_ALL      = 1 << 20,   // enabled for everybody
        FLAG_MEMBER_OF_GAME     = 1 << 21,   // just enabled for game
        FLAG_MEMBER_OF_CONSOLE  = 1 << 22,   // just enabled for the dev console
        FLAG_MEMBER_OF_EDITOR   = 1 << 23,   // just enabled for SiegeEditor
        FLAG_MEMBER_OF_TRIAL    = 1 << 24,   // trial function, don't rely on it yet
    };

    static eFlags ToFlags( FunctionSpecFlags::ePermissions bit );
    static eFlags ToFlags( FunctionSpecFlags::eMembership  bit );
```

```
    static bool   TestMembership( eFlags flags,
                                  FunctionSpecFlags::eMembership bit );

    static const eFlags DEFAULT_MEMBERSHIP;

    typedef std::vector <ParamSpec> ParamSpecs;

    gpstring    m_Name;                 // name of this function
    ClassSpec*  m_Parent;               // member of a class, or NULL if global
    eFlags      m_Flags;                // calling convention, modifiers, traits
    DWORD       m_FunctionPtr;          // memory address of start of function
    UINT        m_SerialID;             // serial number based on EXE import order
    ParamSpecs  m_ParamSpecs;           // param types and names this func takes
    UINT        m_ParamSizeBytes;       // total size of parameter set, or if
                                        // vararg then it's size of known params
    TypeSpec    m_ReturnType;           // return type of function
    DWORD       m_Digest;               // digest of this func - use as a checksum

#   if !GP_RETAIL
    const char* m_Docs;                 // docs for this function
    const char* m_MangledName;          // original mangled name of the function
    gpstring    m_UnmangledName;        // after we unmangle it - ready to parse
#   endif // !GP_RETAIL

    FunctionSpec( void );

    bool     IsValidGet                 ( void ) const;
    bool     IsValidSet                 ( void ) const;
    void     PostProcess                ( void );
    gpstring BuildQualifiedName         ( void ) const;
    gpstring BuildQualifiedNameAndParams( const void* param, const void* object,
                                          bool isRpcPacked,
                                          bool allowBinaryOut = false ) const;

#   if !GP_RETAIL
    void AssertValid ( void ) const;
    void GenerateDocs( ReportSys::ContextRef context = NULL,
                       eDocsLevel level = DOCS_NORMAL ) const;
#   endif // !GP_RETAIL
};

MAKE_ENUM_BIT_OPERATORS( FunctionSpec::eFlags );
```

Most functions in Dungeon Siege are member functions or enclosed in a namespace, and so they end up being owned by a class, which is represented by a *ClassSpec*:

```
struct ClassSpec
{
    // flags
    enum eFlags
    {
        FLAG_NONE           =      0,
        FLAG_MANAGED        = 1 << 0,   // this is a "managed" ResHandleMgr class
        FLAG_SINGLETON      = 1 << 1,   // "singleton" class - derive from Singleton
        FLAG_CANRPC         = 1 << 2,   // class ptrs can be sent over the net
        FLAG_PROPCANSKRIT   = 1 << 3,   // my properties are available to Skrit
        FLAG_POSTPROCESSED  = 1 << 4,   // already postprocessed for doco, vars...
        FLAG_HIDDEN         = 1 << 5,   // this class is just a placeholder, ignore
        FLAG_HAS_STATICS    = 1 << 6,   // this class has visible static methods or
                                        // it's a namespace with global functions
        FLAG_POD            = 1 << 7,   // plain old data
        FLAG_POINTERCLASS   = 1 << 8,   // pointer class - probably a handle type,
                                        // can never be instantiated
```

```
    };

    // collections
    typedef std::map <gpstring, VariableSpec, istring_less> VariableMap;
    typedef std::pair <VariableMap::iterator, bool> VariableMapInsertRet;
    typedef std::pair <const ClassSpec*, int> ParentSpec;
    typedef std::vector <ParentSpec> ParentColl;

    // procedures
    typedef bool  (__cdecl *DoesHandleMgrExistProc     )( void  );
    typedef UINT  (__cdecl *HandleAddRefProc           )( DWORD );
    typedef UINT  (__cdecl *HandleReleaseProc          )( DWORD );
    typedef bool  (__cdecl *HandleIsValidProc          )( DWORD );
    typedef void* (__cdecl *HandleGetProc              )( DWORD );
    typedef void* (__cdecl *GetClassSingletonProc      )( void  );
    typedef DWORD (__cdecl *InstanceToNetProc          )( void* );
    typedef void* (__cdecl *NetToInstanceProc          )( DWORD, FuBiCookie* );
    typedef void  (__cdecl *GetHeaderSpecProc          )( ClassHeaderSpec& );

    // spec
    gpstring               m_Name;                     // class name
    eVarType               m_Type;                     // VAR_USER-based index
    eFlags                 m_Flags;                    // flags for this class
    FunctionSpec::eFlags   m_Membership;               // membership flags
    ClassHeaderSpec*       m_HeaderSpec;               // table spec for schema
    VarTypeSpec*           m_VarTypeSpec;              // traits
    ParentColl             m_ParentClasses;            // derived from what?

#   if !GP_RETAIL
    const char*            m_Docs;                     // doco for entire class
#   endif // !GP_RETAIL

    // members
    FunctionByNameIndex    m_Functions;                // static/nonstatic methods
    VariableMap            m_Variables;                // set/get pairs

    // callbacks
    DoesHandleMgrExistProc m_DoesHandleMgrExistProc;   // check for mgr existence
    HandleAddRefProc       m_HandleAddRefProc;         // add ref to a handle
    HandleReleaseProc      m_HandleReleaseProc;        // remove ref from a handle
    HandleIsValidProc      m_HandleIsValidProc;        // check valid handle
    HandleGetProc          m_HandleGetProc;            // deref a handle
    GetClassSingletonProc  m_GetClassSingletonProc;    // look up the singleton
    InstanceToNetProc      m_InstanceToNetProc;        // convert ptr->net cookie
    NetToInstanceProc      m_NetToInstanceProc;        // convert net cookie->ptr

    ClassSpec( void );
    ClassSpec( const ClassSpec& other );
   ~ClassSpec( void );

    ClassSpec& operator = ( const ClassSpec& other );

    bool IsDerivedFrom       ( eVarType type ) const;
    int  GetDerivedBaseOffset( eVarType base ) const;
    void SetPod              ( size_t size );

    bool  AddMemberFunction( FunctionSpec* function );
    void  PostProcess      ( void );
    DWORD AddExtraDigest   ( DWORD digest ) const;

#   if !GP_RETAIL
    void AssertValid ( void ) const;
    void GenerateDocs( ReportSys::ContextRef context = NULL ) const;
#   endif // !GP_RETAIL
```

```
private:
    // special: will auto-create spec but only on internal class request
    VarTypeSpec* GetVarTypeSpec( void ) const;
};

MAKE_ENUM_BIT_OPERATORS( ClassSpec::eFlags );
```

Finally, this is the big nasty type manager class. The name *SysExports* comes from FuBi's original purpose, which was a simple system function binder, or an advanced version of Gabriel Knight 3's *SysExports*/*Generics* system. A lot has changed since then, but the name remains, oh well. It's also a big monolithic class, which I tend to like, but this one is just too big, which I don't tend to like. Well anyway here it is:

```
class SysExports : public Singleton <SysExports>
{
public:
    SET_NO_INHERITED( SysExports );

// Types.

    // update this as sysexports changes seriously
    enum  {  VERSION = MAKEVERSION( 1, 0, 0 )  };

    enum eOptions
    {
        // disable the "found doco for nonexistent export" warning
        OPTION_NO_EXTRA_DOC_WARNING     = 1 << 0,
        // allow rpc's to be used without requiring a sync call
        OPTION_NO_REQUIRE_SYNC          = 1 << 1,
        // don't check for all-all nesting in sent rpc's
        OPTION_NO_CHECK_ALL_RPC_NESTING = 1 << 2,
        // relax certain super-strict warnings
        OPTION_NO_STRICT                = 1 << 3,

        // ...

        OPTION_NONE = 0,
    };

    struct SyncSpec
    {
        DWORD m_Size;                                   // size of this structure
        DWORD m_Version;
        DWORD m_Digest;

        SyncSpec( void )
        {
            ::ZeroObject( *this );
            m_Size = sizeof( *this );
        }

        SyncSpec( DWORD digest )
        {
            m_Size    = sizeof( *this );
            m_Version = VERSION;
            m_Digest  = digest;
        }

        bool operator == ( const SyncSpec& other ) const
        {
            return ( ::memcmp( this, &other, sizeof( *this ) ) == 0 );
```

```
        }
    };

// Setup.

    // ctor/dtor
    SysExports( void );
   ~SysExports( void );

    // local options
    void SetOptions( eOptions options, bool set = true )
        {  m_Options = (eOptions)(set ? (m_Options | options)
                                      : (m_Options & ~options));  }
    void ClearOptions( eOptions options )
        {  SetOptions( options, false );  }
    void ToggleOptions( eOptions options )
        {  m_Options = (eOptions)(m_Options ^ options);  }
    bool TestOptions( eOptions options ) const
        {  return ( (m_Options & options) != 0 );  }

    // synchronizing for rpc's
    bool Synchronize( const SyncSpec& spec );
    void SetSynchronize( bool set = true )
        {  m_RpcSyncComplete = set;  }
    void ClearSynchronize( void )
        {  SetSynchronize( false );  }

    // spec building
    SyncSpec MakeSynchronizeSpec( void ) const;

    // callback registration
    void RegisterSendRpcVerifyCb( SendRpcVerifyCb cb )
        {  m_SendRpcVerifyCb = cb;  }
    void RegisterSendRpcBroadcastCb( SendRpcBroadcastCb cb )
        {  m_SendRpcBroadcastCb = cb;  }
    void RegisterDispatchRpcVerifyCb( DispatchRpcVerifyCb cb )
        {  m_DispatchRpcVerifyCb = cb;  }

    // exception list for nested rpc warning
#   if GP_DEBUG
    void RegisterNestedRpcException( const char* caller, const char* called );
#   endif // GP_DEBUG

// Importing.

    bool ImportFunction( const char* mangledName, DWORD address,
                         SignatureParser* parser );
    bool ImportBindings( HMODULE module = NULL );

// Type info.

    const FunctionSpec*  FindFunctionByAddrExact(
        DWORD ptr, const char* funcName, int funcNameLen ) const;
    const FunctionSpec*  FindFunctionByAddrNear(
        DWORD ptr, const char* funcName, int funcNameLen ) const;
    const FunctionSpec*  FindFunctionBySerialID(
        UINT serialID ) const;
    const FunctionIndex* FindFunctionsByQualifiedName(
        const char* funcName ) const;
    int                  FindFunctionsByQualifiedName(
        FunctionIndexColl& coll, const char* funcName ) const;
    UINT                 FindEventSerialByName(
        const char* eventName ) const;
```

```
    int GetFunctionSize( const FunctionSpec* spec );
    int GetFunctionCount( void ) const
        {  return ( scast <int> ( m_FunctionsBySerialID.size() ) );  }

    const char* FindTypeName      ( eVarType type ) const;
    const char* FindNiceTypeName  ( const FuBi::TypeSpec& spec ) const;
    gpstring    MakeFullTypeName  ( const FuBi::TypeSpec& spec ) const;
    eVarType    FindType          ( const char* name ) const;
    eVarType    FindOrCreateType  ( const char* name );
    eVarType    FindTypeInternalOk( const char* name ) const;

    ClassSpec*       GetClass ( const gpstring& name );
    const ClassSpec* FindClass( eVarType type ) const;
    ClassSpec*       FindClass( eVarType type );
    const ClassSpec* FindClass( const char* name ) const;

    const ClassSpec* FindEventNamespace( void ) const;
    const ClassSpec* FindSkritObject   ( void ) const;
    const ClassSpec* FindFuBiCookie    ( void ) const;

    EnumSpec*       GetEnum         ( const gpstring& name );
    const EnumSpec* FindEnum        ( eVarType type ) const;
    const EnumSpec* FindEnumConstant( const char* name, DWORD& value,
                                      bool fastOnly = false ) const;

    const VarTypeSpec* GetVarTypeSpec     ( eVarType type ) const;
    int                GetVarTypeSizeBytes( eVarType type ) const;
    bool               IsSimpleInteger    ( eVarType type ) const;
    bool               IsSimpleFloat      ( eVarType type ) const;
    bool               IsPod              ( eVarType type ) const;

    bool IsKeyword           ( const gpstring& name ) const;
    bool IsDerivative        ( eVarType derived, eVarType base ) const;
    int  GetDerivedBaseOffset( eVarType derived, eVarType base ) const;

    bool ToString  ( eVarType type, string& out, const void* obj,
                     eXfer xfer = XFER_NORMAL,
                     eXMode xmode = XMODE_DEFAULT ) const;
    bool ToString  ( const TypeSpec& type, gpstring& out, const void* obj,
                     eXfer xfer = XFER_NORMAL,
                     eXMode xmode = XMODE_DEFAULT ) const;
    bool FromString( eVarType type, const char* str, void* obj,
                     eXMode xmode = XMODE_DEFAULT ) const;
    bool FromString( const TypeSpec& type, const char* str, void* obj,
                     eXMode xmode = XMODE_DEFAULT ) const;
    bool CopyVar   ( eVarType type, void* dst, const void* src,
                     eXMode xmode = XMODE_DEFAULT ) const;
    bool CopyVar   ( const TypeSpec& type, void* dst, const void* src,
                     eXMode xmode = XMODE_DEFAULT ) const;
    int  CompareVar( eVarType type, const void* l, const void* r ) const;
    int  CompareVar( const TypeSpec& type, const void* l, const void* r ) const;

// Iterators.

    ClassIndex::const_iterator GetClassBegin( void ) const
        {  return ( m_ClassVarTypeIndex.begin() );  }
    ClassIndex::const_iterator GetClassEnd( void ) const
        {  return ( m_ClassVarTypeIndex.end() );  }

    EnumIndex::const_iterator GetEnumBegin( void ) const
        {  return ( m_EnumVarTypeIndex.begin() );  }
    EnumIndex::const_iterator GetEnumEnd( void ) const
        {  return ( m_EnumVarTypeIndex.end() );  }
```

```
    FunctionByNameIndex::const_iterator GetGlobalsBegin( void ) const
        {  return ( m_GlobalFunctions.begin() );  }
    FunctionByNameIndex::const_iterator GetGlobalsEnd( void ) const
        {  return ( m_GlobalFunctions.end() );  }

// Global functions.

    inline const FunctionIndex* FindGlobalFunction( const char* name ) const;

// Remote procedure call methods.

    const FunctionSpec* ResolveRpc     ( DWORD EIP, const char* funcName,
                                         int funcNameLen ) const;
    Cookie              SendRpc        ( const FunctionSpec* spec, void* thisParam,
                                         const void* firstParam, DWORD toAddr );
    Cookie              DispatchNextRpc( DWORD callerAddr );
    bool                RpcTestMacro   ( DWORD addr, const FunctionSpec* spec );
    bool                IsDispatching  ( void ) const
                                         {  return ( ms_IsDispatching );  }
    bool                ClearDispatcher( void );
    void                EnterRpc       ( const bool* oldDispatching = NULL );
    void                LeaveRpc       ( void );

// Callback methods.

    const FunctionSpec* ResolveEvent( DWORD EIP, const char* funcName,
                                      int funcNameLen ) const;
    const FunctionSpec* ResolveCall ( DWORD EIP, const char* funcName,
                                      int funcNameLen ) const;

// Misc.

    DWORD GetDigest( void ) const;

#   if !GP_RETAIL
    void GenerateDocs( ReportSys::ContextRef context = NULL ) const;
#   endif // !GP_RETAIL

// Debugging.

#   if !GP_RETAIL
    void DumpStatistics( ReportSys::ContextRef ctx ) const;
    void DumpStatistics( void ) const  {  DumpStatistics( NULL );  }
#   endif // !GP_RETAIL

#   if GP_DEBUG
    void CheckSanity( void ) const;
#   endif // GP_DEBUG

private:

// Maps.

    // these are containers that own their contents

    // $ note: lesson learned after hours of debugging - the release linker will
    //   fold identical functions together under the same entry point, while
    //   giving each a separate export entry. this means that many functions
    //   will share the same address! so the FunctionByAddrMap must be multimap.

    typedef std::map    <gpstring, ClassSpec*, istring_less>        ClassByNameMap;
    typedef std::map    <const char*, EnumSpec*, istring_less>    EnumByNameMap;
    typedef std::pair   <eVarType, DWORD>                         EnumConstant;
    typedef std::map    <const char*, EnumConstant, istring_less> StringToEnumMap;
```

```
    typedef std::multimap <DWORD, FunctionSpec>           FunctionByAddrMap;
    typedef std::pair   <ClassByNameMap::iterator, bool>  ClassByNameMapInsertRet;
    typedef std::pair   <EnumByNameMap::iterator, bool>   EnumByNameMapInsertRet;
    typedef std::set    <gpstring, istring_less>                 StringSet;
    typedef std::vector <VarTypeSpec>                            VarTypeSpecColl;

#   if GP_DEBUG
    typedef std::pair     <UINT /*caller*/, UINT /*called*/>  SerialIdPair;
    typedef std::vector   <SerialIdPair>                      SerialIdPairColl;
    typedef std::vector   <UINT>                              SerialIdColl;
#   endif // GP_DEBUG

    const FunctionSpec* FindFunctionByAddrHelper( const char* funcName,
        int funcNameLen, FunctionByAddrMap::const_iterator found ) const;
    RpcVerifyColl&      GetRpcVerifyColl( void );

    ClassByNameMap    m_Classes;                    // all classes
    EnumByNameMap     m_Enums;                      // all enums
    FunctionByAddrMap m_Functions;                  // all functions

// Indexes.

    // these are containers that point to elements of maps

    ClassIndex          m_ClassVarTypeIndex;        // map eVarType - VAR_USER
                                                    // --> const ClassSpec*
    EnumIndex           m_EnumVarTypeIndex;         // map eVarType - VAR_ENUM
                                                    // --> const EnumSpec*
    EnumIndex           m_IrregularEnums;           // exported irregular enums
    StringToEnumMap     m_EnumConstants;            // all exported enum constants
    FunctionByNameIndex m_GlobalFunctions;          // all global functions
    FunctionIndex       m_FunctionsBySerialID;      // index == serial ID
    VarTypeSpecColl     m_VarTypeSpecs;             // built-in type specs

#   if GP_DEBUG
    SerialIdPairColl    m_NestedRpcExceptionColl;   // exceptions to nested rpc's
    SerialIdColl        m_NestedRpcXCallers;        // match on any called
    SerialIdColl        m_NestedRpcXCalleds;        // match on any caller
#   endif // GP_DEBUG

// RPC support.

    typedef kerneltool::Critical Critical;
    typedef std::list <gpstring> StringVec;
    typedef std::list <gpwstring> WStringVec;
    typedef std::vector <std::pair <DWORD, RpcVerifyColl> > RpcColl;

    mutable Critical    m_SendCritical;             // for serializing RPC sends
    StringVec           m_RpcStrings;               // temp store strings from net
    WStringVec          m_WRpcStrings;              // temp store wstrings from net
    RpcColl             m_RpcColl;                  // per-thread map of call stack
    SendRpcVerifyCb     m_SendRpcVerifyCb;          // verify cb: SendRpc()
    SendRpcBroadcastCb  m_SendRpcBroadcastCb;       // rebroadcasting callback
    DispatchRpcVerifyCb m_DispatchRpcVerifyCb;      // verify cb: DispatchNextRpc()

// Other.

    eOptions  m_Options;            // general options
    StringSet m_Keywords;           // keywords exported for warnings
    DWORD     m_Digest;             // this is a digest of all of FuBi's exports
    bool      m_RenameComplete;     // renaming is complete
    bool      m_ImportComplete;     // import is complete
    bool      m_RpcSyncComplete;    // rpc synching has been done
```

```
    DECL_THREAD static bool ms_IsDispatching;

    SET_NO_COPYING( SysExports );
};

MAKE_ENUM_BIT_OPERATORS( SysExports::eOptions );

#define gFuBiSysExports FuBi::SysExports::GetSingleton()
```

In order to reduce dependency on *SysExports*, and remove the need to *#include* its header file, all of its functions that are called by the macros are wrapped in helper functions that are implemented in a C++ file to reroute to the *SysExports* singleton. These functions are left out here because all they do is call the equivalent named *SysExports* function, which isn't very interesting. Generally you can take *FuBi::FuncName()* to mean *gSysExports.FuncName()*.

## Exports Implementation

The first critical requirement of *SysExports* is to be able to import the functions that are FEX'd from the executable. In emails sent to me regarding FuBi, getting this to work right seems to be the major headache so here is the (somewhat messy and verbose) implementation. The *ImportBindings* function iterates over the entries in the export table and passes each along to *ImportFunction*, which does all the real work.

Note that the *SignatureParser* is a simple scanner/parser built with MKS Lex & Yacc. All it does is parse the undecorated names and store the results in a *FunctionSpec*.

```
bool SysExports :: ImportFunction(
    const char* mangledName, DWORD address, SignatureParser* parser )
{
    // $$$ this entire function should be !GP_RETAIL

    gpassert( !m_ImportComplete );

    typedef DWORD (WINAPI* UnDecorateSymbolNameProc)(LPCSTR, LPSTR, DWORD, DWORD);
    int mangledLen = ::strlen( mangledName );

    // make sure we've got our dll
    static DbgHelpDll s_DbgHelpDll;
    if ( !s_DbgHelpDll.Load() )
    {
        gpfatal( "FuBi: unable to parse export table because no DBGHELP.DLL "
                 "support, app must fatal\n" );
    }

// De-mangle the name.

    // $ note that we could decode the mangled name directly by writing a
    //   parser specifically for that. however the format is specific to the
    //   compiler and version, plus there are no publicly available docs on
    //   it. rather than reverse engineer the format by exporting every possible
    //   function variant, just let DbgHelp to do the work for us. to keep from
    //   being dependent on a debug support DLL and exposing our functions to
    //   the world through the export table, a postprocessor will strip the
    //   export table and store it in the resources instead.

    // de-mangle into readable text for our cheesy little parser
    char unmangledName[ 0x400 ];
    if ( s_DbgHelpDll.UnDecorateSymbolName(
```

```
        mangledName,
        unmangledName,
        ELEMENT_COUNT( unmangledName ),
        UNDNAME_COMPLETE | UNDNAME_32_BIT_DECODE ) == 0 )
{
    return ( false );       // $ error is in ::GetLastError();
}

// check for special exports that we should ignore
static const char* s_IgnoreFuncs[] =
{
    "CoverageAddAnnotation",
    "CoverageClearData",
    "CoverageDisableRecordingData",
    "CoverageIsRecordingData",
    "CoverageIsRunning",
    "CoverageSaveData",
    "CoverageStartRecordingData",
    "CoverageStopRecordingData",
    "PurelockIsRunning",
    "PurelockPrintf",
    "PurePrintf",
    "PurifyAllHandlesInuse",
    "PurifyAllInuse",
    "PurifyAllLeaks",
    "PurifyAssertIsReadable",
    "PurifyAssertIsWritable",
    "PurifyBlue",
    "PurifyClearInuse",
    "PurifyClearLeaks",
    "PurifyDescribe",
    "PurifyGetPoolId",
    "PurifyGetUserData",
    "PurifyGreen",
    "PurifyHeapValidate",
    "PurifyIsInitialized",
    "PurifyIsReadable",
    "PurifyIsRunning",
    "PurifyIsWritable",
    "PurifyMapPool",
    "PurifyMarkAsInitialized",
    "PurifyMarkAsUninitialized",
    "PurifyMarkForNoTrap",
    "PurifyMarkForTrap",
    "PurifyNewHandlesInuse",
    "PurifyNewInuse",
    "PurifyNewLeaks",
    "PurifyPrintf",
    "PurifyRed",
    "PurifySetLateDetectScanCounter",
    "PurifySetLateDetectScanInterval",
    "PurifySetPoolId",
    "PurifySetUserData",
    "PurifyWhatColors",
    "PurifyYellow",
    "QuantifyAddAnnotation",
    "QuantifyClearData",
    "QuantifyDisableRecordingData",
    "QuantifyIsRecordingData",
    "QuantifyIsRunning",
    "QuantifySaveData",
    "QuantifyStartRecordingData",
    "QuantifyStopRecordingData",
};
```

```
    // c++ exports always start with ?
    if ( *mangledName != '?' )
    {
        const char** i, ** ibegin = s_IgnoreFuncs, ** iend
            = ARRAY_END( s_IgnoreFuncs );
        for ( i = ibegin ; i != iend ; ++i )
        {
            if ( same_with_case( unmangledName, *i ) )
            {
                return ( true );
            }
        }
    }

// Build function spec.

    // add generic spec to map
    FunctionByAddrMap::iterator newFunc
        = m_Functions.insert( std::make_pair( address, FunctionSpec() ) );
    FunctionSpec& spec = newFunc->second;

    // initialize
    spec.m_FunctionPtr = address;
    spec.m_SerialID    = m_FunctionsBySerialID.size();

#   if !GP_RETAIL
    spec.m_MangledName   = mangledName;
    spec.m_UnmangledName = unmangledName;
#   endif // !GP_RETAIL

    // add it
    m_FunctionsBySerialID.push_back( &spec );

    // this should be SMALL - plus it must be in a word anyway for RPC's
    gpassert( m_FunctionsBySerialID.size() <= std::numeric_limits <WORD>::max() );

    // parse it
    SignatureParser::eParse parse = parser->Parse( unmangledName, spec );
    if ( parse != SignatureParser::PARSE_OK )
    {
        if ( parse == SignatureParser::PARSE_ERROR )
        {
            gperrorf(( "Error parsing function '%s'\n", unmangledName ));
        }
        goto no_func;
    }

    // check for dup name at same address - this is illegal because it will
    // cause name resolution problems for rpc and skrit callbacks
#   if GP_DEBUG
    {
        FunctionByAddrMap::const_iterator i,
            begin = m_Functions.lower_bound( address ),
            end   = m_Functions.upper_bound( address );
        for ( i = begin ; i != end ; ++i )
        {
            if ( i != newFunc )
            {
                assert( !newFunc->second.m_Name.same_no_case( i->second.m_Name ) );
            }
        }
    }
#   endif // GP_DEBUG
```

```
// Add it to database.

    if ( (spec.m_Parent == NULL) && !(spec.m_Flags & FunctionSpec::FLAG_TRAIT) )
    {
        // global function
        if ( !AddFunctionToIndex( m_GlobalFunctions, &spec ) )
        {
            goto no_func;
        }
    }
    else
    {
        // special: reassign traits to a particular class
        if ( spec.m_Parent == NULL )
        {
            gpassert( spec.m_Flags & FunctionSpec::FLAG_TRAIT );
            spec.m_Parent = FindClass( parser->GetTraitType().m_Type );
        }

        // trait or member function
        if ( !spec.m_Parent->AddMemberFunction( &spec ) )
        {
            goto no_func;
        }
    }

// Reassign serial ID if canonical RPC.

    // check for canonical function
    {
        for (  CanonicalRpcSpec* i = CanonicalRpcSpec::ms_Root
             ; i != NULL
             ; i = i->m_Next )
        {
            // get class if any
            const ClassSpec* classSpec = NULL;
            if ( i->m_ClassName != NULL )
            {
                classSpec = FindClass( i->m_ClassName );
            }

            // match!
            if (    (spec.m_Parent == classSpec)
                && spec.m_Name.same_with_case( i->m_FuncName ) )
            {
                // remove old
                m_FunctionsBySerialID.pop_back();

                // this will assert if we've got a naming collision - overloaded
                // functions cannot be used for canonical rpc's, sorry
                gpassert( m_FunctionsBySerialID[ i->m_Index ] == NULL );

                // reassign and move
                spec.m_SerialID = i->m_Index;
                m_FunctionsBySerialID[ spec.m_SerialID ] = &spec;
            }
        }
    }

// Update digest.

    // $$$ skrits will be affected by default params! also include param defaults
    //     in the checksum (requires doco access)
```

```
    // set individual function digest
    spec.m_Digest = GetCRC32( 0, mangledName, mangledLen + 1 );

    // only do this for non-doco functions
    if ( !(spec.m_Flags & FunctionSpec::FLAG_DOCO) )
    {
        // use the mangled name - it contains the full signature!
        m_Digest = GetCRC32( m_Digest, mangledName, mangledLen + 1 );
    }

    return ( true );

// Non-function case.

no_func:

    // undo addition of function into database
    m_Functions.erase( newFunc );
    m_FunctionsBySerialID.pop_back();

    // not a real function
    return ( false );
}

bool SysExports :: ImportBindings( HMODULE module )
{
    // cannot add to existing import set - i'm not set up for that.
    gpassert( !m_ImportComplete );

    // build parser
    SignatureParser parser;

    // $$$ first attempt to import from the resources, ELSE do all of this export
    // $$$ table stuff... store the digest in with the resource data

    // set to local image by default
    if ( module == NULL )
    {
        module = ::GetModuleHandle( NULL );
    }

// Prep the import.

    // preallocate our canonical rpc's
    m_FunctionsBySerialID.resize( CanonicalRpcSpec::ms_Count, NULL );

// Find the export table.

    // get headers
    const BYTE* imageBase = rcast <const BYTE*> ( module );
    const IMAGE_DOS_HEADER* dosHeader = rcast <const IMAGE_DOS_HEADER*> ( module );
    const IMAGE_NT_HEADERS* winHeader = rcast <const IMAGE_NT_HEADERS*> (
        imageBase + dosHeader->e_lfanew );

    // find the export data directory
    const IMAGE_DATA_DIRECTORY& exportDataDir
        = winHeader->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_EXPORT ];
    DWORD exportRva  = exportDataDir.VirtualAddress;
    DWORD exportSize = exportDataDir.Size;

    // see if it exists
    if ( exportRva == 0 )
    {
```

```
        return ( false );   // nope
    }

    // find the export directory
    DWORD exportBegin = exportRva, exportEnd = exportBegin + exportSize;
    const IMAGE_EXPORT_DIRECTORY* exportDir
        = rcast <const IMAGE_EXPORT_DIRECTORY*> ( imageBase + exportBegin );

    // find the subtables
    const DWORD* funcTable
        = rcast <const DWORD*> ( imageBase + exportDir->AddressOfFunctions    );
    const DWORD* nameTable
        = rcast <const DWORD*> ( imageBase + exportDir->AddressOfNames        );
    const WORD * ordinalTable
        = rcast <const WORD *> ( imageBase + exportDir->AddressOfNameOrdinals );

// Iterate over all the exported functions.

    const DWORD* nameTableBegin = nameTable,
               * nameTableEnd   = nameTableBegin + exportDir->NumberOfNames,
               * nameTableIter;
    const WORD * ordinalTableIter = ordinalTable;

    bool success = true;
    for (  nameTableIter = nameTableBegin
         ; nameTableIter != nameTableEnd
         ; ++nameTableIter, ++ordinalTableIter )
    {
        const char* functionName
            = rcast <const char*> ( imageBase + *nameTableIter );
        DWORD functionAddress = funcTable[ *ordinalTableIter ];

        // $ special: skip if it's just a string
        if ( same_with_case( "??_C@_", functionName, 6 ) )
        {
            continue;
        }

        // this will be false if it's a forwarding export (unlikely but
        // whatever...)
        if ( (functionAddress < exportBegin) || (functionAddress >= exportEnd) )
        {
            // get address to function call
            functionAddress += rcast <DWORD> ( imageBase );

            // $ NOTE $ this code is VERY VC++ version specific. it knows about
            //          how the compiler will export a function via a relative
            //          jump (probably for easy relocations). if this pattern
            //          changes in the future, this code will need to be
            //          updated.

            // it probably points to a jump indirection - redirect to actual
            // function
            if ( *rcast <const BYTE*> ( functionAddress ) == 0xE9 /*jmp*/ )
            {
                // cool, figure out relative jump address (function + sizeof op)
                ++functionAddress;
                DWORD offset = *rcast <const DWORD*> ( functionAddress );
                functionAddress += 4 + offset;
            }

            // attempt to import it
            if ( !ImportFunction( functionName, functionAddress, &parser ) )
            {
```

```
                success = false;
            }
        }
    }

// Postprocess.

    const ClassSpec* eventSpec = FindEventNamespace();
    eVarType skritObjectType = FindSkritObject()->m_Type;

    // postprocess global functions
    PostProcess( m_GlobalFunctions );

    // fix up classes
    {
        ClassByNameMap::iterator i,
            begin = m_Classes.begin(), end = m_Classes.end();
        for ( i = begin ; i != end ; ++i )
        {
            ClassSpec& spec = *i->second;

            gpassert( !(spec.m_Flags & ClassSpec::FLAG_POSTPROCESSED) );
            spec.PostProcess();
            spec.m_Flags |= ClassSpec::FLAG_POSTPROCESSED;

            m_Digest = spec.AddExtraDigest( m_Digest );
        }
    }

    // postprocess our enums
    {
        EnumIndex::iterator i,
            begin = m_EnumVarTypeIndex.begin(), end = m_EnumVarTypeIndex.end();
        for ( i = begin ; i != end ; ++i )
        {
            (*i)->PostProcess();
        }

        begin = m_IrregularEnums.begin(), end = m_IrregularEnums.end();
        for ( i = begin ; i != end ; ++i )
        {
            (*i)->PostProcess();
        }
    }

    // postprocess SKRIT_IMPORT and DECLARE_EVENT functions
    {
        FunctionByAddrMap::iterator i,
            begin = m_Functions.begin(), end = m_Functions.end();
        for ( i = begin ; i != end ; ++i )
        {
            FunctionSpec& spec = i->second;
            if (   (spec.m_ParamSpecs.size() >= 2)      // object + function name
                && (spec.m_ParamSpecs[ 0 ].m_Type.m_Type == skritObjectType)
                && (spec.m_ParamSpecs[ 1 ].m_Type.IsCString()) )
            {
                spec.m_Flags |= FunctionSpec::FLAG_HIDDEN
                              | FunctionSpec::FLAG_SKRIT_IMPORT;
            }

            if ( spec.m_Parent == eventSpec )
            {
                spec.m_Flags |= FunctionSpec::FLAG_EVENT;
#               if ( !GP_RETAIL )
```

```
                {
                    if ( !spec.m_Name.same_no_case( "On", 2 ) )
                    {
                        gpwarningf(( "FuBi consistency warning for event function "
                                     "'%s' - events should be of the form "
                                     "'OnEvent'\n",
                                     spec.m_Name.c_str() ));
                    }
                }
#               endif // !GP_RETAIL
            }
        }
    }

    // final renaming
    ReplaceNameInIndex( m_Classes );
    ReplaceNameInIndex( m_Enums );
    m_RenameComplete = true;

    // check for overloads in event namespace - not supported
#   if ( !GP_RETAIL )
    {
        FunctionByNameIndex::const_iterator
                i,
                begin = eventSpec->m_Functions.begin(),
                end   = eventSpec->m_Functions.end();
        for ( i = begin ; i != end ; ++i )
        {
            // check for overloads
            if ( i->second.size() != 1 )
            {
                gperrorf(( "%s namespace contains overloaded function '%s' - not "
                           "supported by FuBi/Skrit\n",
                               eventSpec->m_Name.c_str(), i->first.c_str() ));
                success = false;
            }
            const FunctionSpec* funcSpec = *i->second.begin();

            // check for vararg
            if ( funcSpec->m_Flags & FunctionSpec::FLAG_VARARG )
            {
                gperrorf(( "%s namespace cannot contain functions using a "
                           "variable argument list ('%s')\n",
                               eventSpec->m_Name.c_str(), i->first.c_str() ));
                success = false;
            }

            // first param must be skrit object
            if ( funcSpec->m_ParamSpecs.empty() )
            {
                gperrorf(( "Event function '%s' must at least take a Skrit object "
                           "as its first param\n",
                               i->first.c_str() ));
                success = false;
            }
            else
            {
                const TypeSpec& firstParam
                    = funcSpec->m_ParamSpecs.begin()->m_Type;
                if (   (firstParam.m_Type != skritObjectType)
                    || !(firstParam.m_Flags & (TypeSpec::FLAG_POINTER |
                                               TypeSpec::FLAG_HANDLE |
                                               TypeSpec::FLAG_REFERENCE)) )
                {
```

```
                    gperrorf(( "Event function '%s' must at least take a Skrit "
                               "object as its first param\n",
                                   i->first.c_str() ));
                    success = false;
                }
            }
        }
    }
#   endif // !GP_RETAIL

    // check to make sure that all canonical rpc's are accounted for
#   if ( GP_DEBUG )
    {
        FunctionIndex::const_iterator begin = m_FunctionsBySerialID.begin(),
                                      end   = begin + CanonicalRpcSpec::ms_Count;
        gpassert( std::find( begin, end, (FunctionSpec*)NULL ) == end );
    }
#   endif // GP_DEBUG

    // check to make sure we've used all our replacements
#   if ( GP_DEBUG )
    {
        if ( !TestOptions( OPTION_NO_STRICT ) )
        {
            NameReplaceSpec::ReplaceColl coll;
            NameReplaceSpec::GetUnusedReplacements( coll );

            if ( !coll.empty() )
            {
                ReportSys::AutoReport autoReport( &gErrorContext );
                gperrorf(( "Found unused global FuBi name replacement specs:\n" ));
                ReportSys::AutoIndent autoIndent( &gErrorContext );

                NameReplaceSpec::ReplaceColl::const_iterator i,
                    begin = coll.begin(), end = coll.end();
                for ( i = begin ; i != end ; ++i )
                {
                    gperrorf(( "%s -> %s\n", i->first, i->second ));
                }
            }
        }
    }
#   endif // GP_DEBUG

// Done.

    // import all done - no more data
    m_ImportComplete = true;

    // verify that everything ok
    GPDEBUG_ONLY( CheckSanity() );

    // done
    return ( success );
}
```

Here are a few of the database lookup functions that I use all over the place. I'm including these for fun, but maybe it will be useful. Took a little while to get some of them right.

```
const FunctionSpec* SysExports :: FindFunctionByAddrNear( DWORD ptr,
    const char* funcName, int funcNameLen ) const
{
    gpassert(   (funcName != NULL)
```

```
            && (::strlen( funcName ) == (size_t)funcNameLen) );

    // same as FindFunctionExact() but meant for finding what function we're
    //  currently executing in.

    FunctionByAddrMap::const_iterator found = m_Functions.lower_bound( ptr );

    // three conditions from lower_bound:
    //
    // 1. found == end() - found points to the iterator AFTER the one we want
    // 2. found points to the iterator AFTER the one we want
    // 3. found == begin() - ptr is not pointing to a valid function

    if ( found == m_Functions.begin() )
    {
        // invalid function
        return ( NULL );
    }
    else
    {
        // we're at least one past what we want, but it's a multimap - go to
        // the front
        --found;
        DWORD addr = found->first;
        while ( (found != m_Functions.begin()) && (found->first == addr) )
        {
            // seek backwards
            --found;
        }

        // now we're one before what we want
        ++found;

        // all done
        return ( FindFunctionByAddrHelper( funcName, funcNameLen, found ) );
    }
}

const FunctionIndex* SysExports :: FindFunctionsByQualifiedName(
    const char* funcName ) const
{
    const FunctionIndex* funcSpec = NULL;

    const char* start = ::strstr( funcName, "::" );
    if ( start != NULL )
    {
        const ClassSpec* classSpec = FindClass( gpstring( funcName, start ) );
        if ( classSpec != NULL )
        {
            FunctionByNameIndex::const_iterator found
                = classSpec->m_Functions.find( start + 2 );
            if ( found != classSpec->m_Functions.end() )
            {
                funcSpec = &found->second;
            }
        }
    }
    else
    {
        funcSpec = FindGlobalFunction( funcName );
    }

    return ( funcSpec );
}
```

```
int SysExports :: FindFunctionsByQualifiedName( FunctionIndexColl& coll,
    const char* funcName ) const
{
    int oldSize = coll.size();

    const char* start = ::strstr( funcName, "::" );
    if ( start != NULL )
    {
        const ClassSpec* classSpec = FindClass( gpstring( funcName, start ) );
        if ( classSpec != NULL )
        {
            const char* memberName = start + 2;
            if ( stringtool::HasDosWildcards( memberName ) )
            {
                FunctionByNameIndex::const_iterator i,
                    ibegin = classSpec->m_Functions.begin(),
                    iend   = classSpec->m_Functions.end();
                for ( i = ibegin ; i != iend ; ++i )
                {
                    if ( stringtool::IsDosWildcardMatch( memberName, i->first ) )
                    {
                        coll.push_back( &i->second );
                    }
                }
            }
            else
            {
                FunctionByNameIndex::const_iterator found
                    = classSpec->m_Functions.find( memberName );
                if ( found != classSpec->m_Functions.end() )
                {
                    coll.push_back( &found->second );
                }
            }
        }
    }
    else
    {
        const FunctionIndex* found = FindGlobalFunction( funcName );
        if ( found != NULL )
        {
            coll.push_back( found );
        }
    }

    return ( coll.size() - oldSize );
}

UINT SysExports :: FindEventSerialByName( const char* eventName ) const
{
    UINT serialId = INVALID_FUNCTION_SERIAL;

    const ClassSpec* classSpec = FindEventNamespace();
    if ( classSpec != NULL )
    {
        FunctionByNameIndex::const_iterator found
            = classSpec->m_Functions.find( eventName );
        if ( found != classSpec->m_Functions.end() )
        {
            gpassert( !found->second.empty() );
            serialId = found->second[ 0 ]->m_SerialID;
        }
    }
```

```
    return ( serialId );
}

const FunctionSpec* SysExports :: ResolveRpc( DWORD EIP, const char* funcName,
    int funcNameLen ) const
{
    // in here for dev modes we attempt to limp along by returning null, which
    // will abort the RPC, rather than just crashing.

    // find the function
    const FunctionSpec* spec = FindFunctionByAddrNear(
        EIP, funcName, funcNameLen );

#   if !GP_RETAIL

    // verify it even exists
    if ( spec == NULL )
    {
        gperrorf(( "RPC: unable to find FuBi function from EIP = 0x%08X "
                   "(should be '%s')\n",
                    EIP, funcName ));
        return ( NULL );
    }

    // verify it can be rpc'd
    if (   !(spec->m_Flags & FunctionSpec::FLAG_CAN_RPC)
        || (spec->m_Flags & FunctionSpec::FLAG_VARARG) )
    {
        gperrorf(( "RPC: function '%s' cannot be called remotely (check its "
                   "signature)\n",
                    spec->BuildQualifiedName().c_str() ));
        return ( NULL );
    }

#   endif // !GP_RETAIL

    return ( spec );
}

const FunctionSpec* SysExports :: FindFunctionByAddrHelper(
        const char* funcName, int funcNameLen,
        FunctionByAddrMap::const_iterator found ) const
{
    // is it the only one at that address? we get to cheat if it is. most
    //  functions will be like this so we'll keep the efficiency most of the
    //  time.
    DWORD ptr = found->first;
    FunctionByAddrMap::const_iterator next = found;
    ++next;
    if ( (next == m_Functions.end()) || (next->first != ptr) )
    {
        const FunctionSpec* spec = &found->second;

#       if !GP_RETAIL
        // verify it's the right one
        if ( !same_with_case_fast( spec->m_Name.c_str(), spec->m_Name.length(),
                                   funcName, funcNameLen ) )
        {
            gperrorf(( "FuBi call function lookup mismatch from address = "
                       "0x%08X - found '%s' (should be '%s')\n",
                        ptr, spec->m_Name.c_str(), funcName ));
            return ( NULL );
        }
```

```
#       endif // !GP_RETAIL

        return ( spec );
    }

    // do linear search to find our named function - necessary to get the
    //  proper typing on the function.
    for ( ; (found != m_Functions.end()) && (found->first == ptr) ; ++found )
    {
        const FunctionSpec* spec = &found->second;
        if ( same_with_case_fast( spec->m_Name.c_str(), spec->m_Name.length(),
                                  funcName, funcNameLen ) )
        {
            return ( spec );
        }
    }

    gperrorf(( "FuBi call function lookup failed from address = 0x%08X "
               "(function is '%s')\n", ptr, funcName ));
    return ( NULL );
}
```

## RPC Implementation

In the *FUBI_RPC_CALL_IMPL* macro I use the *AutoRpcTag* and derivative classes – these are not very important, so ignore them. They simply handle automatic clearing of the RPC dispatch state in the case of an early exit. The *RpcTestMacro* function, on the other hand, is important. It determines what to do upon entering the RPC function. The function is fairly self-explanatory except perhaps the nested dispatch detection.

Because any function may or may not decide to RPC, FuBi implicitly encourages a distributed networking model where RPC functions are sprinkled all over the code. It's so convenient to make a function into an RPC (just FEX it and add a macro to its implementation) that hidden performance problems may result on occasion. If one RPC function calls another RPC function, then two separate network calls are made, which unnecessarily uses up bandwidth, and causes duplicate work on the clients. In a deterministic system, it is generally possible to call the initial top-level RPC, and then have all further work performed locally.

For example, say you have two network calls *SetHealth()* to set the "hit points" and then *SetSleepState()* to set a flag as to whether or not the object is asleep. Also say that *SetHealth* automatically puts an object to sleep if the health drops below a certain percentage. So calling *SetHealth* would also sometimes call *SetSleepState*, using up extra bandwidth and causing an extra *SetSleepState* call on the client. This can easily be avoided by having *SetHealth* call a non-RPC function *LocalSetSleepState* function (which, incidentally, *SetSleepState* could call as well). Then the client and server both perform the exact same work locally, avoiding the unnecessary RPC. Cases like these aren't obvious, but *RpcTestMacro* can find them automatically, and that is what the extra debug code checks for. If it detects a nested RPC it will warn, unless that condition has been specifically excepted.

```
bool SysExports :: RpcTestMacro( DWORD addr, const FunctionSpec* spec )
{
    // get local thread's rpc call spec - one must have been added by auto tag ctor
    RpcVerifyColl& coll = GetRpcVerifyColl();
    assert( !coll.empty() );
```

```
    // check to see if we've got a nested "all" dispatch
#   if GP_DEBUG
    if (   !TestOptions( OPTION_NO_CHECK_ALL_RPC_NESTING )
        && ((addr == RPC_TO_ALL) || (addr == RPC_TO_OTHERS))
        && (spec != NULL) )
    {
        RpcVerifyColl::const_iterator i, begin = coll.begin(), end = coll.end();
        for ( i = begin ; i != end ; ++i )
        {
            if ( (i->m_Address == RPC_TO_ALL) || (i->m_Address == RPC_TO_OTHERS) )
            {
                const FunctionSpec* caller = i->m_Function;
                const FunctionSpec* called = spec;

                // check against exception list
                if (   !std::binary_search( m_NestedRpcExceptionColl.begin(),
                                           m_NestedRpcExceptionColl.end(),
                                           SerialIdPair( caller->m_SerialID,
                                                         called->m_SerialID ) )
                    && !std::binary_search( m_NestedRpcXCallers.begin(),
                                           m_NestedRpcXCallers.end(),
                                           caller->m_SerialID )
                    && !std::binary_search( m_NestedRpcXCallers.begin(),
                                           m_NestedRpcXCallers.end(),
                                           called->m_SerialID ) )
                {
                    ReportSys::ContextRef ctx
                         = AssertData::IsAssertOccurring()
                         ? &gWarningContext : &gPerfContext;
                    ReportSys::OutputF( ctx, "PERFORMANCE WARNING: Nested 'all' "
                                             "RPC send detected!\n"
                                             "\tCaller: '%s'\n"
                                             "\tCalled: '%s'\n",
                                        caller->BuildQualifiedName().c_str(),
                                        called->BuildQualifiedName().c_str() );
                }
            }
        }
    }
#   endif // GP_DEBUG

    // set address for call on stack
    coll.rbegin()->m_Address = addr;

    // if we're in an rpc, run it local - this will reset dispatch flag
    bool old = ms_IsDispatching;
    ms_IsDispatching = false;
    if ( old )
    {
        return ( false );
    }

    // if it's supposed to go local, run it local
    if ( addr == RPC_TO_LOCAL )
    {
        return ( false );
    }

    // if it's for the server which is local, run it local
    if ( (addr == RPC_TO_SERVER) && IsServerLocal() )
    {
        return ( false );
    }
```

```
    // ok, do the rpc
    return ( true );
}
```

This is a good place to point out that you should come up with a naming convention for your RPC's. Bart came up with a good convention for Dungeon Siege. We prefix each RPC function with S (server-only), RS (RPC to server), or RC (RPC to clients). Generally, server-side logic functions such as AI scripts will call non-RPC 'S' functions. The 'S' functions will then call the RPC'd 'RC' functions, which will route to all affected clients using *RPC_TO_ALL*. If the clients ever have to talk to the server, they do so through an RPC'd 'RS' method using *RPC_TO_SERVER*. Generally 'RC' functions are off-limits to call from script or non-server code. Here is an example:

```
FEX void RSChat( const char* str )
{
    FUBI_RPC_CALL( RSChat, RPC_TO_SERVER );
    SChat( str );
}

FEX void RCChat( const char* str );
{
    FUBI_RPC_CALL( RCChat, RPC_TO_ALL );
    Chat( str );
}

void SChat( const char* str )
{
    RCChat( str );

    // perform other server duties, such as logging...
}

void Chat( const char* str )
{
    printf( "Chat: %s\n", str );
}
```

Note that *SChat* and *RSChat* can probably be simplified by merging, as can *RCChat* and *Chat*. This is often the case with most RPC's in Dungeon Siege because they are so simple. In other RPC's, though, there is a lot more logic that must run in each of these functions that is distinct, and so they must remain separate. For example, object creation in Dungeon Siege is totally screwy, and as such I need this convention to maintain my sanity.

Anyway here is the really fun stuff – the dispatcher and network routers. Notice that these functions are practically mirror images of each other. Most of this code goes into the packing and unpacking of parameters. Some pointers get converted to cookies, and others get the data they are pointing to tacked on as a rider.

```
Cookie SysExports :: SendRpc( const FunctionSpec* spec, void* thisParam,
    const void* firstParam, DWORD toAddr )
{
    struct AutoClear
    {
        bool& m_B;
```

```
        AutoClear( bool& b ) : m_B( b )  {  gpassert( !m_B );  m_B = true;  }
       ~AutoClear( void )  {  gpassert( m_B );  m_B = false;  }
    };

    AutoClear autoClear( ms_IsSending );

    gpassert( spec != NULL );
    gpassert( !::IsMemoryBadFood( (DWORD)thisParam ) );
    gpassert( !::IsMemoryBadFood( (DWORD)firstParam ) );

    // serialize RPC requests
    Critical::Lock locker( m_SendCritical );

// Preprocess.

    // get local thread's rpc call spec - one must have been added by auto tag ctor
    RpcVerifyColl& coll = GetRpcVerifyColl();
    gpassert( !coll.empty() );

    // set spec for call on stack
    RpcVerifySpec& verifySpec = *coll.rbegin();
    verifySpec.m_Function = spec;
    verifySpec.m_Params = firstParam;
    verifySpec.m_This = thisParam;
    gpassert( verifySpec.m_Address == toAddr );

    // verify that we are ready for non-canonical rpc's
#   if ( GP_DEBUG )
    {
        // we must have either done a sync or are issuing an rpc send for a
        // canonical rpc function.
        gpassert(   m_RpcSyncComplete
                 || ((int)spec->m_SerialID < CanonicalRpcSpec::ms_Count)
                 || TestOptions( OPTION_NO_REQUIRE_SYNC ) );
    }
#   endif // GP_DEBUG

    // verify that we're using the proper macro - it needs a "THIS_" if we're
    // calling a nonstatic method. note that if we're calling it on a singleton
    // we don't *need* the "this" pointer.
#   if ( GP_DEBUG )
    {
        if ( spec->m_Flags & FunctionSpec::FLAG_CALL_THISCALL )
        {
            const ClassSpec* classSpec = spec->m_Parent;
            gpassertf(   (classSpec != NULL)
                      && (classSpec->m_Flags & ClassSpec::FLAG_CANRPC),
                      ( "Attempted to RPC-send from a non-RPCable function '%s'!",
                        spec->BuildQualifiedName().c_str() ));
            if ( !(classSpec->m_Flags & ClassSpec::FLAG_SINGLETON) )
            {
                gpassertf( thisParam != NULL,
                           ( "Incorrect macro usage for FuBi RPC's for function "
                             "'%s'! Did you forget the '_THIS'?",
                             spec->BuildQualifiedName().c_str() ));
            }
        }
    }
#   endif // GP_DEBUG

    // verify we have a transport - if we don't, then silently abort (common in
    // single player games)
    if( !NetSendTransport::DoesSingletonExist() )
    {
```

```
        return ( RPC_SUCCESS );
    }

    // ask callback to reprocess the RPC stack for addressing on a broadcast
    AddressColl addresses;
    if ( ((toAddr == RPC_TO_ALL) || (toAddr == RPC_TO_OTHERS))
        && m_SendRpcBroadcastCb )
    {
        if ( !m_SendRpcBroadcastCb( verifySpec, addresses ) )
        {
            // leave it alone
            addresses.clear();
        }
        else if ( addresses.size() == 1 )
        {
            // only a single shot
            toAddr = addresses.front();

            // is this just for myself?
            if ( toAddr == RPC_TO_LOCAL )
            {
                // $ success! abort! early bailout
                return ( RPC_SUCCESS );
            }

            // take it
            verifySpec.m_Address = toAddr;
            addresses.clear();
        }
        else if ( addresses.empty() )
        {
            // $ broadcaster removed all addresses, no need to continue!
            // early bailout!
            return ( RPC_SUCCESS );
        }
    }

    // ask callback to verify the RPC stack
    if ( m_SendRpcVerifyCb && !AssertData::IsAssertOccurring() )
    {
        if ( !m_SendRpcVerifyCb( coll, addresses ) )
        {
            // client doesn't want to allow this
            return ( RPC_FAILURE_IGNORE );
        }
    }

    // begin the rpc
    bool isRetrying = (spec->m_Flags & FunctionSpec::FLAG_RETRY) != 0;
    Cookie cookie = addresses.empty()
                  ? gFuBiNetSendTransport.BeginRPC( toAddr, isRetrying )
                  : gFuBiNetSendTransport.BeginMultiRPC(
                        addresses.begin(), addresses.size(), isRetrying );

// Pack parameters.

    // pack function serial ID
    gFuBiNetSendTransport.Store( scast <WORD> ( spec->m_SerialID ) );

    // pack parameters
    UINT paramIter = gFuBiNetSendTransport.Store(
        firstParam, spec->m_ParamSizeBytes );

    // pack riders and special data
```

```
if ( !(spec->m_Flags & FunctionSpec::FLAG_SIMPLE_ARGS) )
{
    FunctionSpec::ParamSpecs::const_iterator i, begin
        = spec->m_ParamSpecs.begin(), end = spec->m_ParamSpecs.end();
    for ( i = begin ; i != end ; ++i )
    {
        // note: in all of the below code - any call to Send() will
        //       invalidate any pointers to the contents of the transport's
        //       buffer - due to vector's demand-reallocation.

        // process special data
        switch ( i->m_Type.m_Type )
        {
            case ( VAR_MEM_PTR ):
            case ( VAR_CONST_MEM_PTR ):
            {
                // get at the mem_ptr param
                mem_ptr* param = rcast <mem_ptr*> (
                    gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
                void* oldParam = param->mem;
                // clear pointer to help compression
                param->mem = NULL;
                // store rider
                gFuBiNetSendTransport.Store( oldParam, param->size );
            }   break;

            case ( VAR_CHAR ):
            {
                if ( i->m_Type.m_Flags
                     == (TypeSpec::FLAG_POINTER | TypeSpec::FLAG_CONST) )
                {
                    // get at the const char* param
                    const char** param = rcast <const char**>
                        ( gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
                    // copy what it was pointing to, get its size
                    // (check for NULL)
                    const char* oldParam = *param;
                    int oldParamLen = (oldParam == NULL)
                        ? 0 : (::strlen( oldParam ) + 1);
                    // set first DWORD to size of the rider
                    *rcast <DWORD*> ( param ) = oldParamLen;
                    // store rider
                    gFuBiNetSendTransport.Store( oldParam, oldParamLen );
                }
            }   break;

            case ( VAR_USHORT ):
            {
                if ( i->m_Type.m_Flags
                     == (TypeSpec::FLAG_POINTER | TypeSpec::FLAG_CONST) )
                {
                    // get at the const wchar_t* param
                    const wchar_t** param = rcast <const wchar_t**>
                        ( gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
                    // copy what it was pointing to, get its size
                    // (check for NULL)
                    const wchar_t* oldParam = *param;
                    int oldParamLen = (oldParam == NULL)
                        ? 0 : (::wcslen( oldParam ) + 1);
                    // set first DWORD to size of the rider
                    *rcast <DWORD*> ( param ) = oldParamLen;
                    // store rider
                    gFuBiNetSendTransport.Store( oldParam, oldParamLen * 2 );
                }
```

```
}   break;

case ( VAR_STRING ):
{
    gpassert( !i->m_Type.IsPassByValue() )

    // get at the gpstring param
    const gpstring** param = rcast <const gpstring**>
        ( gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
    // copy what it was pointing to, get its size (check for NULL)
    const gpstring* oldParam = *param;
    gpassert( oldParam != NULL );
    // set first DWORD to size of the rider
    *rcast <DWORD*> ( param ) = oldParam->length() + 1;
    // store rider
    gFuBiNetSendTransport.Store(
        oldParam->c_str(), oldParam->length() + 1 );
}   break;

case ( VAR_WSTRING ):
{
    gpassert( !i->m_Type.IsPassByValue() )

    // get at the gpwstring param
    const gpwstring** param = rcast <const gpwstring**>
        ( gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
    // copy what it was pointing to, get its size (check for NULL)
    const gpwstring* oldParam = *param;
    gpassert( oldParam != NULL );
    // set first DWORD to size of the rider
    *rcast <DWORD*> ( param ) = oldParam->length() + 1;
    // store rider
    gFuBiNetSendTransport.Store( oldParam->c_str(),
        (oldParam->length() + 1) * 2 );
}   break;

default:
{
    // check for POD types
    const VarTypeSpec* spec = gFuBiSysExports.GetVarTypeSpec(
        i->m_Type.m_Type );
    if ( (spec != NULL) && (spec->m_Flags & Trait::FLAG_POD) )
    {
        // pod types passed by ref can be tacked on to the end
        if ( !i->m_Type.IsPassByValue() )
        {
            // get at the param
            const void** param = rcast <const void**> (
              gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
            // copy what it was pointing to
            const void* oldParam = *param;
            // set first DWORD to true/false for NULL
            *rcast <DWORD*> ( param ) = (oldParam != NULL);
            // store rider
            gpassert( spec->m_SizeBytes > 0 );
            if ( oldParam != NULL )
            {
                gFuBiNetSendTransport.Store(
                    oldParam, spec->m_SizeBytes );
            }
        }
    }
    // special user type requires network cookie conversion
    else if ( IsUser( i->m_Type.m_Type ) )
```

```
                    {
                        const ClassSpec* classSpec
                            = gFuBiSysExports.FindClass( i->m_Type.m_Type );

                        // don't bother with pointer classes
                        if ( !(classSpec->m_Flags & ClassSpec::FLAG_POINTERCLASS) )
                        {
                            // verify system integrity
                            gpassert( (classSpec != NULL)
                               && (classSpec->m_Flags & ClassSpec::FLAG_CANRPC) );
                            gpassert( !(classSpec->m_Flags
                               & ClassSpec::FLAG_SINGLETON) );
                            gpassert( classSpec->m_InstanceToNetProc != NULL );
                            gpassert( !i->m_Type.IsPassByValue() );

                            // patch pointer with cookie
                            DWORD* param = rcast <DWORD*> (
                              gFuBiNetSendTransport.GetPtrFromIndex( paramIter ) );
                            *param = (*classSpec->m_InstanceToNetProc)
                              ( (void*)(*param) );
                        }
                        else
                        {
                            // must be a pointer
                            gpassert( !i->m_Type.IsPassByValue() );
                        }
                    }
                }
            }

            // move on to next param
            paramIter += i->m_Type.GetSizeBytes();
        }
    }

    // pack "this" at the end if not a singleton
    if ( spec->m_Flags & FunctionSpec::FLAG_CALL_THISCALL )
    {
        const ClassSpec* classSpec = spec->m_Parent;
        gpassert(   (classSpec != NULL)
                 && (classSpec->m_Flags & ClassSpec::FLAG_CANRPC) );
        if ( !(classSpec->m_Flags & ClassSpec::FLAG_SINGLETON) )
        {
            gpassert( classSpec->m_InstanceToNetProc != NULL );

            // $ this assert will fire if you have used FUBI_RPC_CALL rather than
            //   FUBI_RPC_THIS_CALL in your exported function.
            gpassert( thisParam != NULL );

            DWORD cookie = (*classSpec->m_InstanceToNetProc)( thisParam );
            gFuBiNetSendTransport.Store( cookie );
        }
    }

// Seal off the RPC packet.

    // done
    gFuBiNetSendTransport.EndRPC();
    return ( cookie );
}

Cookie SysExports :: DispatchNextRpc( DWORD callerAddr )
{
```

```
// Preprocess.

    // figure out which function was called, bail if invalid
    UINT serialID = ReceiveRef <WORD> ();
    const FunctionSpec* spec = FindFunctionBySerialID( serialID );
    if ( spec == NULL )
    {
        gperrorf(( "SysExports::DispatchNextRpc(): invalid serial ID %d "
                   "received\n" ));
        return ( RPC_FAILURE_IGNORE );
    }

    // safety check - make sure that this function can be rpc'd. this will fail
    // if the versions of the game are mismatched (i.e. the digest was not
    // checked as a precondition to connecting to the host).
    gpassertf( spec->m_Flags & FunctionSpec::FLAG_CAN_RPC,
             ( "FuBi RPC receiver: unexpected dispatch of a non-RPC function!\n\n"
               "Received serialID = 0x%04X, found function '%s'.\n",
               serialID, spec->BuildQualifiedName().c_str() ));

    // default cookie
    Cookie fubiCookie = RPC_SUCCESS;
    DWORD rc = 0;

// Unpack parameters.

    // unpack parameters
    void* params = gFuBiNetReceiveTransport.ReceivePtr( spec->m_ParamSizeBytes );

    // clear out temp strings
    m_RpcStrings.clear();
    m_WRpcStrings.clear();

    // unpack riders and special data
    if ( !(spec->m_Flags & FunctionSpec::FLAG_SIMPLE_ARGS) )
    {
        BYTE* paramIter = rcast <BYTE*> ( params );

        FunctionSpec::ParamSpecs::const_iterator i, begin
            = spec->m_ParamSpecs.begin(), end = spec->m_ParamSpecs.end();
        for ( i = begin ; i != end ; ++i )
        {
            // process special data
            switch ( i->m_Type.m_Type )
            {
                case ( VAR_MEM_PTR ):
                case ( VAR_CONST_MEM_PTR ):
                {
                    // get at the mem_ptr param
                    mem_ptr* param = rcast <mem_ptr*> ( paramIter );
                    // point to rider and advance
                    if ( param->size != 0 )
                    {
                        param->mem = gFuBiNetReceiveTransport
                            .ReceivePtr( param->size );
                    }
                }   break;

                case ( VAR_CHAR ):
                {
                    if ( i->m_Type.m_Flags
                        == (TypeSpec::FLAG_POINTER | TypeSpec::FLAG_CONST) )
                    {
                        // get at the const char* param
```

```
            const char** param = rcast <const char**> ( paramIter );
            // fetch the size
            DWORD size = *rcast <const DWORD*> ( param );
            // point to rider and advance
            if ( size != 0 )
            {
                *param = rcast <const char*> (
                    gFuBiNetReceiveTransport.ReceivePtr( size ) );
            }
        }
    }   break;

    case ( VAR_USHORT ):
    {
        if ( i->m_Type.m_Flags
            == (TypeSpec::FLAG_POINTER | TypeSpec::FLAG_CONST) )
        {
            // get at the const wchar_t* param
            const wchar_t** param = rcast <const wchar_t**>
                ( paramIter );
            // fetch the size
            DWORD size = *rcast <const DWORD*> ( param );
            // point to rider and advance
            if ( size != 0 )
            {
                *param = rcast <const wchar_t*> (
                  gFuBiNetReceiveTransport.ReceivePtr( size * 2 ) );
            }
        }
    }   break;

    case ( VAR_STRING ):
    {
        gpassert( !i->m_Type.IsPassByValue() )

        // get at the gpstring param
        const gpstring** param = rcast <const gpstring**>
            ( paramIter );
        // fetch the size
        DWORD size = *rcast <const DWORD*> ( param );
        // assign it to rider and advance
        gpstring str;
        if ( size != 0 )
        {
            str.assign( rcast <const char*> (
                gFuBiNetReceiveTransport.ReceivePtr( size ) ),
                size - 1 );
        }
        m_RpcStrings.push_back( str );
        // reassign to point to the new string
        *param = &m_RpcStrings.back();
    }   break;

    case ( VAR_WSTRING ):
    {
        gpassert( !i->m_Type.IsPassByValue() )

        // get at the gpwstring param
        const gpwstring** param = rcast <const gpwstring**>
            ( paramIter );
        // fetch the size
        DWORD size = *rcast <const DWORD*> ( param );
        // assign it to rider and advance
        gpwstring str;
```

```
            if ( size != 0 )
            {
                str.assign( rcast <const wchar_t*> (
                    gFuBiNetReceiveTransport.ReceivePtr( size * 2 ) ),
                    size - 1 );
            }
            m_WRpcStrings.push_back( str );
            // reassign to point to the new string
            *param = &m_WRpcStrings.back();
        }   break;

        default:
        {
            // check for POD types
            const VarTypeSpec* spec
                = gFuBiSysExports.GetVarTypeSpec( i->m_Type.m_Type );
            if ( (spec != NULL) && (spec->m_Flags & Trait::FLAG_POD) )
            {
                // pod types passed by ref have their data tacked on
                // to the end
                if ( !i->m_Type.IsPassByValue() )
                {
                    // get at the param
                    const void** param = rcast <const void**>
                        ( paramIter );

                    // if non-null, point to rider
                    gpassert( spec->m_SizeBytes > 0 );
                    if ( *rcast <const DWORD*> ( param ) != 0 )
                    {
                        *param = gFuBiNetReceiveTransport.ReceivePtr(
                            spec->m_SizeBytes );
                    }
                }
            }
            // special user type requires reverse network cookie conversion
            else if ( IsUser( i->m_Type.m_Type ) )
            {
                const ClassSpec* classSpec
                    = gFuBiSysExports.FindClass( i->m_Type.m_Type );

                // don't bother with pointer classes
                if ( !(classSpec->m_Flags & ClassSpec::FLAG_POINTERCLASS) )
                {
                    // verify system integrity
                    gpassert( (classSpec != NULL)
                        && (classSpec->m_Flags & ClassSpec::FLAG_CANRPC) );
                    gpassert( !(classSpec->m_Flags
                        & ClassSpec::FLAG_SINGLETON) );
                    gpassert( classSpec->m_NetToInstanceProc != NULL );
                    gpassert( !i->m_Type.IsPassByValue() );

                    // patch net cookie with pointer
                    DWORD* param = rcast <DWORD*> ( paramIter );
                    *param = (DWORD)(*classSpec->m_NetToInstanceProc)
                        ( *param, &fubiCookie );

                    // abort if told to
                    if ( fubiCookie != RPC_SUCCESS )
                    {
                        return ( fubiCookie );
                    }
                }
                else
```

```
                        {
                            // must be a pointer
                            gpassert( !i->m_Type.IsPassByValue() );
                        }
                    }
                }
            }

            // move on to next param
            paramIter += i->m_Type.GetSizeBytes();
        }
    }

// Perform security and verification checks.

    // check server permissions
    if ( spec->m_Flags & FunctionSpec::FLAG_CHECK_SERVER_ONLY )
    {
        if ( callerAddr != RPC_TO_SERVER )
        {
            // permission failure!
            gperrorf(( "SysExports::DispatchNextRpc(): illegal receipt of "
                       "SERVER-only RPC from non-server machine!\n"
                       "  RPC = %s\n", spec->BuildQualifiedName().c_str() ));
            return ( RPC_FAILURE_IGNORE );
        }
    }

    // get address of object
    void* object = NULL;
    if ( spec->m_Flags & FunctionSpec::FLAG_CALL_THISCALL )
    {
        const ClassSpec* classSpec = spec->m_Parent;
        gpassert(   (classSpec != NULL)
                 && (classSpec->m_Flags & ClassSpec::FLAG_CANRPC) );

        if ( classSpec->m_Flags & ClassSpec::FLAG_SINGLETON )
        {
            // get the singleton
            object = (*classSpec->m_GetClassSingletonProc)();
        }
        else
        {
            // unpack cookie
            DWORD netCookie;
            gFuBiNetReceiveTransport.ReceiveCopy( netCookie );

            // convert it to "this"
            gpassert( classSpec->m_NetToInstanceProc != NULL );
            object = (*classSpec->m_NetToInstanceProc)( netCookie, &fubiCookie );

            // abort if told to
            if ( fubiCookie != RPC_SUCCESS )
            {
                return ( fubiCookie );
            }
        }
    }

    // ask callback to verify the RPC stack
    if ( m_DispatchRpcVerifyCb && !AssertData::IsAssertOccurring() )
    {
        RpcVerifySpec verifySpec;
        verifySpec.m_Address  = callerAddr;
```

```
            verifySpec.m_Function = spec;
            verifySpec.m_Params   = params;
            verifySpec.m_This     = object;
            if ( !m_DispatchRpcVerifyCb( true, verifySpec ) )
            {
                // client doesn't want to allow this
                return ( RPC_FAILURE_IGNORE );
            }
        }

    // Call the function.

        // make the function call
        ms_IsDispatching = true;
        if ( spec->m_Flags & FunctionSpec::FLAG_CALL_CDECL )
        {
            rc = BlindCallFunction_CdeclCall(
                params, spec->m_ParamSizeBytes, spec->m_FunctionPtr );
        }
        else if ( spec->m_Flags & FunctionSpec::FLAG_CALL_STDCALL )
        {
            rc = BlindCallFunction_StdCall(
                params, spec->m_ParamSizeBytes, spec->m_FunctionPtr );
        }
        else if ( spec->m_Flags & FunctionSpec::FLAG_CALL_THISCALL )
        {
            // call the function on the resulting object (if any)
            if ( object != NULL )
            {
                rc = BlindCallFunction_ThisCall(
                    params, spec->m_ParamSizeBytes, object, spec->m_FunctionPtr );
            }
            else
            {
                gperrorf(( "Illegal member function call %s, with 'this' == NULL!\n",
                           spec->BuildQualifiedName().c_str() ));

                fubiCookie = RPC_FAILURE;
                ms_IsDispatching = false;
            }
        }
        else
        {
            gpassert( 0 );  // should never get here
        }

        // should have been reset by the test inside the function
        gpassertm( ms_IsDispatching == false,
                   "Serious error! Dispatched an RPC to a non-RPC function! Do "
                   "the EXE's match exactly? Perhaps you forgot a FUBI_RPC_TAG() "
                   "in your function and do a return statement before the "
                   "FUBI_RPC_CALL() macro?" );

    // Finish.

        // set cookie properly
        if (   (fubiCookie == RPC_SUCCESS)
            && (spec->m_Flags & FunctionSpec::FLAG_RETRY) )
        {
            fubiCookie = (Cookie)rc;
        }

        // tell callback about it so it can do another verify
        if ( m_DispatchRpcVerifyCb && !AssertData::IsAssertOccurring() )
```

```
    {
        RpcVerifySpec verifySpec;
        verifySpec.m_Address  = callerAddr;
        verifySpec.m_Function = spec;
        verifySpec.m_Params   = params;
        verifySpec.m_This     = object;
        m_DispatchRpcVerifyCb( false, verifySpec );
    }

    // return cookie
    return ( fubiCookie );
}
```

## Conclusion

I'm hoping that this paper will provide a decent supplement to the main FuBi paper by providing some sorely-needed sample code. Unfortunately, as I read this back to myself, the organization is lacking, and you may have had to skip around quite a bit to follow what's going on…sentences referencing functions that haven't been written about yet, that sort of thing. There may also be missing implementations of functions that you are interested in. If so, please let me know and I will update this doc accordingly. Ultimately, what matters is that you get the importing, type extraction, and blind callbacks working properly. Everything else is an implementation detail.