WMLScriptEase Manual

WMLScriptEase:ISDK/C version 0.90

Nombas Incorporated
64 Salem Street
Medford, MA 02155
USA

http://www.nombas.com/us/

WMLScriptEase: Integration SDK

Thank you for using Nombas' WMLScriptEase. The Integration SDK allows you to compile and execute WMLScript scripts from within your C or C++ application. Written entirely in C, the WMLScriptEase ISDK provides you with a collection of function calls for working with WMLScripts, including the ability to write wrapper functions that allow scripts to call directly into your application. This manual will describe how to add WMLScript capability to your application, perform common scripting tasks, and write these wrapper functions.

This document will refer to the Wireless Application Protocol (WAP)Language Specification as provided by the WAP Forum. Copies of this, and other WAP documents are freely available at `http://www.wapforum.com/docs/techincal.htm`.

Installation and Setup

The WMLScript distribution consists of three source code directories which you are free to put anywhere on your system that you like. These directories are named `shared`, `compile`, and `interp`. The remaining directories contain documentation, sample makefiles, and test scripts. When you unpack the distribution, feel free to move it to the most convenient place on your hard drive; you are not stuck with keeping it wherever it is now.

When you have finished placing it, you must determine the directory that contains the accompanying source files. We will label this directory **WML**. For instance, this file will be referred to as **WML**`\doc\wmlscript.pdf`. On Unix systems, directories are separated by a forward slash, not a backslash so the equivalent is **WML**`/doc/wmlscript.pdf`

The root directory to which you installed this code will have make files (or targets or projects) for some of the most common compiler environments. The following instructions are helpful for building with the supplied build files or for creating your own.

Compiler sample

You may wish to build the sample compiler. It is a command-line application that takes a script file (`.ws` extension), compiles it, and writes the output to the same filename except with a `.wsb` extension. Since the output file is completely portable, you can compile it on a different machine from where you want to interpret it. The file **WML**`/demos/compile/compile.c` has the sample *main()* that does this compiling; you may want to look at it to see working code when you go to integrate the ISDK into your application (described later).

To compile the wml compiler application, start a new target in your favorite C compiler environment. Predefine the macro `WML_COMPILER`. You will also need to add the directories **WML**`/shared` and **WML**`/interp` to the include paths the compiler will search. The resulting executable can be run by providing the name of the file you wish to compile as its only parameter.

Interpreter sample

A sample interpreter can be built similarly to the compiler. However, the interpreter is more platform specific, as it needs to know how to read various URL formats for instance. You may define the macros `__WML_WIN32__` or `__WML_UNIX__` to compile on those systems, or you may define none and implement the changes discussed in the 'Integrating with your application' section of this manual to build a version for a special environment.

When building for one of the systems described above, build the target much like you would for the compiler sample. Define the macro for your operating system and define `WML_INTERP` (instead of `WML_COMPILER`). Compile and link the application.

Compiling and Interpreting sample scripts

Now that you have the sample interpreter available, you can use it to run some of the sample scripts included with the distribution. The directory tree `tests` contains a number of scripts. If you have built the compiler, the sample interpreter application uses the rule that `main()` is the function to be run. In a typical WMLScript application you specify both the script module and the function to be run.

The only parameter to be given is the URL of the sample. This can be a relative URL in which it is relative to `file://./`; in effect, you can type a filename as the parameter such as `samples\lang\wrldtest.wsb`. Or you can give a complete URL such as `file://c/wml/myscript.wsb` or `http://localhost/scripts/script.wsb`.

Integrating with your application

The process to add WMLScriptEase ISDK to your application is straightforward. You need to tell your compiler where to find the accompanying header files. On most IDEs, there will be an option to specify additional include paths. On unix compilers, you specify the directories using the `-I` command line switch, which you should add to your `CFLAGS` macro. The directories to add are **`WML`**`/shared and` **`WML`**`/interp`. You will want to add both regardless of which routines your application will call.

Next, you need to add the source files to your application. The WML interpreter or compiler will be compiled directly into your application. You may need to edit some of the source files to correctly interact with your application. This process is described later in this manual. All of the C files in the 'shared' directory must be added. The C files in the other two directories need to be added if you will be doing the corresponding actions; if you are compiling, you will need those in **`WML`**`/compile` and likewise if you wish to execute scripts, you will add those in **`WML`**`/interp`.

Your application is now ready to make any calls to WMLScriptEase routines.

Using the WMLScriptEase API

The WMLScriptEase API consists of two basic actions, compiling a script and executing a script.

Compiling

You compile a script using the `wsCompile()` call. Its prototype is this:

```
ubyte *wsCompile(char *filename,wschar *buf,size_t *size);
```

As you can see, you pass three parameters. '`filename`' is the not an actual file that is read, but rather it is the file that is reported in any error messages. You pass whatever text string you want to appear in any error messages. The second parameter, which is either characters or unicode characters depending on your build, is the actual text of the script to be compiled. It should be terminated by a `\0` character. Finally, the last parameter is an output only parameter. The return from this function is `NULL` if the compile failed. In this case an error message describing the problem will have been generated. Otherwise, a buffer is returned. It contains the bytes that make up the standard WMLScript output format for this script. The 'size' parameter is filled in with the length of this buffer in bytes. You can store this script however you like, such as by writing it to a file. The output is usable by any WMLScript interpreter that follows the WMLScript specification (see the WMLScript Language Specification.)

When you are finished with the returned buffer, it must be freed. An API call is provided to do this. Its prototype is:

```
void wsFreeBytecodes(ubyte *buf,size_t size);
```

You pass it the return from the `wsCompile()` call along with the '`size`' output parameter from that call. If the return from `wsCompile()` was `NULL`, you should not make this call.

One typical case when using this call is to store the bytecodes in a file which you then interpret using the `file://` url syntax described below. You can also store a file containing the bytecodes on a web server and retrieve it using the `http://` url syntax. However, when implementing WMLScriptEase on an embedded system, you may have no web or file access. In this case, you will store the bytecodes in some way of your choosing. You will need to modify the interpreter's url parser to be able to access these stored bytecodes. All of this is described later.

Interpeting

Interpreting a script is more complex. You have a number of options to do this interpreting. First, you need a context from which to interpret. A context is a data structure used internally by the WMLScriptEase interpreter. You can create as many contexts as you like. Each one is independent from the others and can interpret one script at a time. The call to make a new context is prototyped as follows:

```
wsContext wsNewContext();
```

The returned value is a magic cookie; you don't need to know what it means. However, a value of `NULL` is returned only if a new context could not be created. This happens when you run out of memory. If the context is not `NULL`, you are ready to begin using it to interpret scripts. When you are done with it, you must destroy the context using this call:

```
void wsDeleteContext(wsContext wsc);
```

Modifying the context

You now have a context that can be used to interpret scripts. However, before you do so, there are several routines you can call to modify the behavior of the context.

First, when a script is executed, by default any error is reported via a message to `stderr`. You can change this behavior by providing your own error handling function. The following typedef shows the type your function must be:

```
typedef wsbool (*wsErrorHandler)(wsContext wsc,char *msg);
```

As you can see, you will be passed an error message and the context it occurred in. The first 3 characters of the error message are a unique numeric code for that type of error (for example, 903: string not terminated.). For a list of default error codes see **`WML`**`/shared/wserror.h`. You are free to display or ignore the error message in any way you see fit. If you return `False`, the script will be terminated as normal for having an error occur. If you return `True`, the script will continue as if no error happened. Be warned that returning `True` is dangerous. The script may be unable to continue, such as when not enough memory remains.

You assign the error handler to the context by using the function prototyped as follows:

```
wsErrorHandler wsSetErrorHandler(wsContext wsc,wsErrorHandler
                                 handler);
```

The return is the previous error handler. You can save it and restore it at a later time if you wish.

You can set up a routine to be called by the interpreter occasionally as it executes the script. This routine should return `True` to keep executing the script. If it returns `False`, the script will be terminated. The routine should be of this type:

```
typedef wsbool (*wsContinueHandler)(wsContext wsc);
```

You install the routine using the following function:

```
wsContinueHandler wsSetContinueHandler(wsContext wsc,
                                 wsContinueHandler handler,
                                 uint32 instrs,uint32 *oldinstrs);
```

Like the error handler above, it returns the old continue handler as well as its instruction count so you can restore it if desired. The `instrs` parameter is simply the number of bytecodes to be executed between calls to your function. Bytecodes are conceptually very small; you can execute quite a number of them per second. Thus, you should make the `instrs` parameter somewhat large. 100 or more is good, you should experiment to determine the frequency you require for your application. If you make the number too small, your continue function will be called very often and much of the execution time of the program will be spent calling your function.

Executing a script

Now that you have a context set up, you can execute scripts. There are several routines to interpret a script, but the standard call is prototyped as follows:

```
wsvalue wsInterpURL(wsContext wsc,char *url);
```

The script and function you specify is loaded and executed. See the section below on WMLScript URLs for complete information on how you use the `url` parameter to specify the function you wish to execute. The return value is the result returned by the function. It is a standard wsvalue, a concept described fully in its own section. It is worth reiterating here that you must destroy this value when you finish with it. This is fully explained below.

There are additional functions to give you more control over what exactly you would like to interpret. Each is explained fully in the WML ScriptEase API section. These are the functions which you can choose from:

```
wsvalue wsInterpScript(wsContext wsc,script handle,char
                       *function);
wsvalue wsInterpFunc(wsContext wsc,char *function);
wsvalue wsInterpFuncArgs(wsContext wsc,char *function,...);
wsvalue wsInterpURLArgs(wsContext wsc,char *url,...);
```

Once you have finished with the return value and destroyed it, the context can be used to interpret additional scripts.

Speeding repeated execution

Each time you interpret a script, that script's bytecodes must be read in, parsed, and verified. If you are interpreting a single script and exiting, this is fine. If, however, you are making repeated calls to the same script, this is a considerable performance penalty. You can choose to load a script once and lock it in memory. It will be freed later when you explicitly release it or automatically when you close down the context. The following routines will load and then release a loaded script:

```
script wsLoadScript(wsContext wsc,char *url);
void wsUnloadScript(wsContext wsc,script handle);
```

Although `wsLoadScript()` takes a URL as described below, you are not allowed to include the function locator fragment. Whenever any URL references this script, it will already be located and not needed to be loaded. You can use the `wsInterpScript()` routine to directly execute functions in this script as well.

URL Syntax

For more information on the conventions for URL syntax please see WMLScript document section 9.2 (`http://www.wapforum.com/docs/techincal.htm`, document or visit the Wireless Application Protocol web site at `http://www.wapforum.com/` document `wmlss-30-apr-98.pdf`.

Customizing your WMLScriptEase interpreter

Customization of the interpreter involves two tasks. First, you need to get the code to compile on your system. This should be easy as the entire WMLScriptEase ISDK is written in standard ANSI C. The second process is to change the behavior of the engine for differences in your system. For instance, URLs may not be able to refer to files and TCP-IP connections, and the Dialogs routines may need to be modified to talk to your user. Each customization is described in its own section.

Standard types

The file **WML**`/shared/wstypes.h` determines what types correspond to the needed types. For instance, on most systems, an `int` is 32-bits, so `sint32` is typedefed to `signed int`. Perhaps an `int` on your system is only 16 bits long while a `long` is in fact 32 bits. You need to change the corresponding typedef in this file.

The typedefs defined in **WML**`/shared/wstypes.h` are:

```
wsbool
ubyte
sbyte
sint32
uint32
uint16
sint16
float32
wschar
```

Allocating memory

The file **WML**`/shared/wsmem.h` determines how memory is allocated. It documents each of the routines called internally. You can change these routines to force different kinds of memory to be alloced in different ways. Read this file for full information.

In order to support operating environments in which memory can be important, we've defined a number of kinds of memory allocation calls here. You can change the given macros to modify the behavior of all such kinds of calls.

wsMalloc()

Generic malloc used to allocate 'large' chunks of memory. These will be things like source files in which the item can be arbitrarily big. Note that many allocations through this routine will be small but the size could be very large. Only fixed-sized items are allocated here. If the item will be growing, the grow allocators are used.

wsGrowMalloc() and wsGrowRealloc()

Used to initially allocate some kind of pool that will be reallocated as needed. For example, the run-time operand stack.

wsSmallMalloc()

Used to allocate a structure (i.e. do a 'new') Although different sized-structures will be alloced, you can expect many similar sized ones and none will be particularly big. Open-ended structures (i.e. ones that have a size element and are alloced to some arbitrary size) will be considered to be 'growable' and alloced with the grow routines if it can change in size, otherwise it will be considered to be large and alloced with `wsMalloc()`.

wsFastMalloc()

This is used exactly like `wsSmallMalloc()` except it is used for the few structures that should be memory pooled. These structures are alloced/freed/and accessed many many times. You should use the fastest memory you have available for them and if you turn off memory pooling, be warned that you can expect them to be malloced/freed millions of times in a typical program's execution.

wsStringMalloc() and wsStrdup

Used to allocate a string of the given number of chars. This string will stick around for the execution of the program.

wsTempStringMalloc()

Just like `wsStringMalloc()` except the string is being created for a temporary purpose and will soon be freed.

wsFree(), wsStringFree(), wsGrowFree(), wsSmallFree(), wsFastFree()

Free an item alloced with the appropriate routine.

You are free to add new kinds of memory. If you make it look similar to the above, you will find it easier to keep in sync with updates. For all structures that we allocate, they are alloced in one routine. By modifying the given routine, you can change the way that structure is allocated. All strings are used pretty generically, and are all alloced using `wsStringMalloc()` and `wsTempStringMalloc()`. They are used only for strings, so these are in effect the allocators for all strings in the program.

`struct wsConst - constant.c - allocateConst() uses wsSmallMalloc().`

The standard library

The file **WML**`/interp/wsstdrun.c` provides the wrapper functions for each of the WMLScript standard library calls. The Dialogs and WMLBrowser libraries will likely need to be modified to correctly interact with your system. See the section below from more information on writing wrapper functions.

Reading URLs

The file **WML**`/interp/wsurl.c` handles all reading of urls. You may need to modify it to change existing URL types or add new ones as appropriate for your system. This goes along with how you want to store URLs described earlier.

Default URL

You can define `WS_DEFAULT_URL` to be the base url that relative URLs are relative to. This applies only to calls not already within a script, since then the calls are relative to the script's url. If you don't define this, the default base url is "`file://./`". You could change it to, for instance, "`http://localhost/`".

Meta tags

Meta tags exist in WMLScript and are, by default ignored. However, the user may create a custom use for them. In the file **WML**/compile/wscomp.c (search on METANOTE:) can be found the code that reads them in. You can modify this to do something of your choice when meta tags are found.

Floating point

You can turn off floating point by defining WS_NO_FLOAT. The behavior of WMLScriptEase when floating point is turned off is exactly as defined in the WMLScript specification, section 14.

Writing wrapper functions

Wrapper functions are routines called by the interpreter to perform a scripting function that is implemented in C. All of the standard WMLScript library functions are for instance implemented via wrapper functions. You may also write your own wrapper functions to make available to the script user.

A wrapper function is declared as follows:

```
wsvalue wrapper(wsContext wsc,wsvalue *argv)
{
   /* function body here */
}
```

The arguments are passed to the function in argv. These arguments are wsvalues as described later. The number of argument values passed to each function is fixed based on the table used to add that function wrapper function to the ISDK. Variable number of parameters are not allowed.

Every function wrapper returns a wsvalue to be the result of that function call. If NULL is be returned it is implicitly replaced with the empty string. The body of the function is dependent on the function s purpose. Look at **WML**/interp/wsstdrun.c for samples of wrapper functions.

Adding wrapper function libraries

In order to give scripts better control over your application, you will probably want to make additional wrapper functions callable by these scripts. This is a straightforward process that involves the creation and addition of a wrapper library to a context. The following structure allows you to define a single library wrapper function.

```
struct wsLibraryFunction
{
   char *funcname;           /* name of the function */
   int numargs;              /* number of arguments it takes */
   wsWrapper wrapper;        /* wrapper function to call */
};
```

The definition includes the name the script will refer to the function with, the number of arguments the function takes, and the corresponding C code function to call. A library is an array of these structures with the last element having all of its members set to NULL.

After you have written your wrapper functions and created a library description array, you need to add this library to the context. You do this using the wsAddLibrary() call prototyped as:

```
wsbool wsAddLibrary(wsContext wsc,char *libname, struct
                        wsLibraryFunction *funcs);
```

The library is added and given the name libname. This name is a URL that will be redirected to use this library. Normally a URL loads bytecodes, in this case you specify that a reference to this URL instead refers to the wrapper functions you provide. Any URL is acceptable, it does not have to match a standard URL type, although you could do that. The user will access your functions using the use url syntax, for example:

```
use url myurl "myurl";
```

or

```
use url myurl "http://localhost/library.wsb";
```

Modifying the standard library

The second way to make wrapper functions available is to extend the standard library with extra functions, but this is not recommended. If you do modify the standard library then your bytecodes will no longer match the WMLScript Specification and therefore will not be portable with other WMLScript implementations.

The file **WML**/shared/wsstdlib.c enumerates the various standard functions. You must add the names and number of arguments your functions take, either by extending an existing library or adding a new library. In the file **WML**/interp/wsstdrun.c there are corresponding tables to indicate which wrapper functions are to be called when the function is invoked. You must extend these tables in the same way. The tables must sync up or scripts will not run correctly.

Storing information

There will be times when you need to associate information with a particular context and retrieve it. You can associate a single pointer with each context and later retrieve it. This is useful to point to a particular structure that contains some information you'd like to be able to retrieve. The following two functions are provided:

```
void  wsSetContextData(wsContext wsc,void *data);
void *wsGetContextData(wsContext wsc);
```

Reporting errors

In a wrapper function, you may determine that some error condition exists. You can report that to the engine using this function:

```
void wsReportError(wsContext wsc,wschar *format,...);
```

It uses a `printf` format string followed by arguments. The error message is reported and the script terminated when you return from the function. Any return value is ignored.

Other times, you want the script to exit and return a particular value, like the C library function `exit()` does. This function will tell the interpreter to terminate the script when your function returns. The return from your function becomes the return for the script.

```
void wsShouldExit(wsContext wsc);
```

Working with wsvalues

A primary task of a wrapper function is to extract the WMLScript values of its parameters and create a WMLScript value to return as the result of the function. This is done with wsvalues. A wsvalue holds one single WMLScript value, be it an integer, float, boolean, string, or invalid value. You can create new wsvalues, query their type, or get at the actual value. Wsvalues are immutable once created.

Each wsvalue contains a particular type of data. You can use this function to find out what it is:

```
int wsValueType(wsContext wsc,wsvalue val);
```

It returns one of 5 values: `WS_VT_INT`, `WS_VT_FLOAT`, `WS_VT_STRING`, `WS_VT_BOOL`, or `WS_VT_INVALID`. Once you know the type, you can access its value. Only the `WS_VT_INVALID` has no underlying value. The following routines access a wsvalues value. Note that if you use the wrong extraction routine for the type of the wsvalue, you will get a nonsensical result. Make sure to verify the type first.

```
sint32 wsValueGetInt(wsContext wsc,wsvalue val);
float wsValueGetFloat(wsContext wsc,wsvalue val);
wsbool wsValueGetBool(wsContext wsc,wsvalue val);
sint32 wsValueGetLength(wsContext wsc,wsvalue val);
wschar *wsValueGetString(wsContext wsc,wsvalue val);
```

A key concept to a wsvalue is ownership. If you own a wsvalue, you must relinquish that ownership at some time. If you don't, the value will never be freed and you will have a memory leak. Wsvalues can be generated in one of two places. The first place are the wsvalues passed to you as parameters. You do not own these. You can access them, but they will disappear when your wrapper function exits. The second place is by creating a new wsvalue. The following functions will all create a new wsvalue that you then own.

```
wsvalue wsValueNewInvalid(wsContext wsc);
wsvalue wsValueNewString(wsContext wsc,wschar *string,int length);
wsvalue wsValueNewInt(wsContext wsc,sint32 val);
wsvalue wsValueNewFloat(wsContext wsc,float val);
```

```
wsvalue wsValueNewBool(wsContext wsc,wsbool val);
wsvalue wsValueNewEmpty(wsContext wsc);
```

You must destroy these values when you are done with them since you own them. The following function will destroy a wsvalue you own, relinquishing your ownership on it. Do not destroy a value you do not own.

```
void wsDestroyValue(wsContext wsc,wsvalue val);
```

When you return from a wrapper function, you return a wsvalue that is to be the result of the function. This is analogous to destroying it, you are relinquishing ownership on this wsvalue. This has two consequences. First, you do not also destroy the value. Destroying a wsvalue or returning it from a wrapper function are two different ways to relinquish ownership of the wsvalue; choose only one. Second, since you do not own your parameters, you cannot return them. You cannot relinquish ownership you do not have, if you do you will certainly cause the engine to crash. In this case, the following function is provided:

```
wsvalue wsValueAddUser(wsContext wsc,wsvalue val);
```

It creates a new ownership on the given wsvalue for you. You can use it on a parameter, so that the result is a wsvalue identical to the parameter but which you now own. Thus, you can return the wsvalue from your wrapper function. Each call to this routine creates a new ownership of the wsvalue. You can create more than one. For instance, if you really wanted to, you could create a new wsvalue (giving you an ownership of it) then use `wsValueAddUser()` to make a second ownership of it. Before you exit your function, you must deal with both ownerships. You might use `wsDestroyValue()` to relinquish one ownership and return the wsvalue to relinquish the second.

Although these functions indicate the complexity of wsvalues, you can think of them simply. If you want to return one of your parameters, you return the `wsValueAddUser()` of that parameter. If you create a wsvalue using any of the creation functions listed above, you either destroy it using `wsDestroyValue()` or return it from your wrapper function.

Conversions

WMLScript defines standard ways to convert values of one type to another. They are typically applied to parameters of the standard library functions. The following functions perform the standard conversion. They all return a new wsvalue which you have ownership of; in effect they create a new wsvalue which is the result of the conversion. The original source wsvalue is unchanged. The type of the returned value will always be one of the target types or `WS_VT_INVALID`. The later indicates an error in converting.

```
wsvalue wsConvertToString(wsContext wsc,wsvalue src);
wsvalue wsConvertToInteger(wsContext wsc,wsvalue src);
wsvalue wsConvertToBoolean(wsContext wsc,wsvalue src);
wsvalue wsConvertToFloat(wsContext wsc,wsvalue src);
wsvalue wsConvertToIntOrFloat(wsContext wsc,wsvalue src);
```

The last function is different in that it takes two values and converts both to ints or both to floats (using the standard WMLScript rules). Thus, it has two output wsvalue parameters which are filled in with the resulting wsvalues. Like the above, both of the new values are owned by you. Both values will be of `WS_VT_INVALID` if the conversion failed.

```
void wsConvertIntsAndFloats(wsContext wsc,wsvalue src1,wsvalue
                            src2,wsvalue *dst1,wsvalue *dst2);
```

WMLScriptEase API

Here is an alphabetical listing of all functions in the WMLScriptEase API along with a description and usage information.

---

wsAddLibrary

DESCRIPTION

Adds a new library of compiled wrapper functions to the engine.

SYNTAX

```
wsbool wsAddLibrary(wsContext wsc,char *libname,
                    struct wsLibraryFunction *funcs);
```

COMMENTS

To use this function, first you need to build a table of wrapper functions. A wrapper function itself is defined as follows:

```
wsvalue wrapper(wsContext wsc,wsvalue *argv)
{
}
```

Wrapper functions are completely defined <link to manual chapter.> You build an array of library function structures which refer to these wrapper functions and specify the number of arguments the functions take along with their name:

```
struct wsLibraryFunction
{
   char *funcname; /* name of the function */
   int numargs; /* number of arguments it takes */
      wsWrapper wrapper;
      /* wrapper function to call */
};
```

Once you've built the array of these (terminated by an entry of all `NULL`), call this routine to register it. It will masquerade as the give URL name. The function table itself is retained so you must make sure to keep it intact even after the function returns.

RETURN

A boolean indicating success.

EXAMPLE

The following code fragment shows how a small library named `nombas` may implement the `nombas#printf` function.

```
wsvalue nombasPrintf(wsContext wsc,wsvalue *argv)
{
   /* code for   printf   function goes here */
}

/* A table of the functions included, in this case just one.
 */
struct wsLibraryFunction nombasLib[] =
{
   { "printf", 1, nombasPrintf },
   { NULL, 0, NULL }
};

main()
{
     .... initialization code here ....
     wsAddLibrary(wsc,"nombas",nombasLib)
     .... interpret and termination code here ....
}
```

wsCompile

DESCRIPTION Compile a WMLScript script and return the bytecodes for it.

SYNTAX
```
ubyte *wsCompile(char *filename,wschar *buf,
                 size_t *size);
```

COMMENTS This function takes a '\0'-terminated string of characters 'buf' which is ASCII or UNICODE depending on the system (see **WML**/shared/wstypes.h.) It is compiled into bytecodes, and returned with the size parameter having the size of the bytecodes in bytes filled in.

The filename parameter is used only to label errors that occur; you should give the name of the file the script is associated with or some other way the user can figure out what the error message is referring to.

RETURN NULL if an error, else the bytecodes. When finished with this return, use wsFreeBytescodes() to get rid of it.

SEE ALSO wsFreeBytecodes

wsConvertToBoolean

DESCRIPTION Converts a given wsvalue to a boolean value according to the WMLScript rules.

SYNTAX
```
wsvalue wsConvertToBoolean(wsContext wsc,
                           wsvalue src);
```

COMMENTS Stock WMLScript conversion.

RETURN A new wsvalue which you must destroy when done. If the conversion fails, an invalid value is returned (which must still be destroyed).

SEE ALSO wsConvertToString, wsConvertToInteger, wsConvertToFloat, wsConvertToIntOrFloat, wsConvertIntsAndFloats

wsConvertToFloat

DESCRIPTION Converts a given wsvalue to a float value according to the WMLScript rules.

SYNTAX
```
wsvalue wsConvertToFloat(wsContext wsc,
                         wsvalue src);
```

COMMENTS Stock WMLScript conversion.

RETURN A new wsvalue which you must destroy when done. If the conversion fails, an invalid value is returned (which must still be destroyed).

SEE ALSO wsConvertToString, wsConvertToInteger, wsConvertToBoolean, wsConvertToIntOrFloat, wsConvertIntsAndFloats

wsConvertToInteger

DESCRIPTION Converts a given wsvalue to an integer value according to the WMLScript rules.

SYNTAX
```
wsvalue wsConvertToInteger(wsContext wsc,
                           wsvalue src);
```

COMMENTS Stock WMLScript conversion.

RETURN A new wsvalue which you must destroy when done. If the conversion fails, an invalid value is returned (which must still be destroyed).

SEE ALSO wsConvertToString, wsConvertToBoolean, wsConvertToFloat, wsConvertToIntOrFloat, wsConvertIntsAndFloats

wsConvertToIntOrFloat

DESCRIPTION Converts a given wsvalue to an integer or float value according to the WMLScript rules.

SYNTAX

```
wsvalue wsConvertToIntOrFloat(wsContext wsc,
                              wsvalue src);
```

COMMENTS Stock WMLScript conversion. The choice to convert to float or integer is specified in the WMLScript specification. The value is converted to an integer. If that is not possible, it is converted to a float.

RETURN A new wsvalue which you must destroy when done. If the conversion fails, an invalid value is returned (which must still be destroyed).

SEE ALSO wsConvertToString, wsConvertToInteger, wsConvertToBoolean, wsConvertToFloat, wsConvertIntsAndFloats

wsConvertToIntsAndFloats

DESCRIPTION Converts the two given wsvalues to integer or float value according to the WMLScript rules.

SYNTAX

```
void wsConvertIntsAndFloats(wsContext wsc,wsvalue
                    src1, wsvalue src2,
                    wsvalue *dst1, wsvalue *dst2);
```

COMMENTS Stock WMLScript conversion. The choice to convert to float or integer is specified in the WMLScript specification. If either value is a floating point, then both are converted to float. Otherwise, integer is tried for both then float for both if it fails.

RETURN None. Two new wsvalues are filled in, and must be destroyed when done. If either conversion fails that value will be filled by an invalid value (which must still be destroyed).

SEE ALSO wsConvertToString, wsConvertToInteger, wsConvertToBoolean, wsConvertToFloat, wsConvertToIntOrFloat

wsConvertToString

DESCRIPTION Converts a given wsvalue to a string value according to the WMLScript rules.

SYNTAX

```
wsvalue wsConvertToString(wsContext wsc,
                          wsvalue src);
```

COMMENTS Stock WMLScript conversion.

RETURN A new wsvalue which you must destroy when done. If the conversion fails, an invalid value is returned (which must still be destroyed).

SEE ALSO wsConvertToInteger, wsConvertToBoolean, wsConvertToFloat, wsConvertToIntOrFloat, wsConvertIntsAndFloats

wsDeleteContent

DESCRIPTION Destroy a context you are done with.

SYNTAX

```
void wsDeleteContext(wsContext wsc);
```

COMMENTS When you finish using a context, you use this routine to destroy it and free all associated memory.

RETURN none

SEE ALSO wsNewContext

wsDestroyValue

DESCRIPTION Relinquish a wsvalue that you own

SYNTAX
```
void wsDestroyValue(wsContext wsc,wsvalue val);
```

COMMENTS You will no longer own the given wsvalue. You can only use this on values you own, such as values you created using one of the `wsCreateXXX()` functions.

RETURN none

SEE ALSO wsValueNewBool, wsValueNewFloat, wsValueNewInt, wsValueNewInvalid, wsValueNewString

wsFreeBytecodes

DESCRIPTION Free the bytecodes returned from `wsCompile()`.

SYNTAX
```
void wsFreeBytecodes(ubyte *buf,size_t size);
```

COMMENTS When you are finished with the bytecodes returned by `wsCompile()`, use this function to discard them. The second parameter is the size (an output parameter from `wsCompile()`). Don't call this function if the output was `NULL`.

RETURN none

SEE ALSO wsCompile

wsGetContextData

DESCRIPTION Get the saved generic pointer you associated with this context.

SYNTAX
```
void *wsGetContextData(wsContext wsc);
```

COMMENTS This simple retrieves a pointer you have previously saved with `wsSetContextData()`. What it points to is up to you.

RETURN The saved pointer.

SEE ALSO wsSetContextData

wsInterpFunc

DESCRIPTION Call a function in the current script

SYNTAX
```
wsvalue wsInterpFunc(wsContext wsc,
                     char *function);
```

COMMENTS This routine is identical to `wsInterpScript()` except the called function is looked for in the current script. You, thus, do not include the host part of the url, only the function, for example: "`#myfunc(1,2)`"

RETURN The wsvalue result of the called function. You own it and must destroy it when you are done with it.

SEE ALSO wsInterpURL, wsInterpScript, wsInterpFuncArgs, wsInterpURLArgs

wsInterpFuncArgs

DESCRIPTION Call a function in the current script passing wsvalue arguments.

SYNTAX
```
wsvalue wsInterpFuncArgs(wsContext wsc,
                         char *function,...);
```

COMMENTS This routine is exactly like `wsInterpFunc()` in that you specify a function in the current script to execute. However, you do not specify the arguments. Instead, you pass a series of wsvalues as arguments, terminated by `NULL`. These values will not be freed; when the function returns you still own them. You specify the function name such as "`#myfunc`".

wsInterpFuncArgs

RETURN The wsvalue result of the called function. You own it and must destroy it when you are done with it.

SEE ALSO wsInterpURL, wsInterpScript, wsInterpFunc, wsInterpURLArgs

wsInterpScript

DESCRIPTION Call a function in the given script.

SYNTAX

```
wsvalue wsInterpScript(wsContext wsc,
                       script handle,
                       char *function);
```

COMMENTS Similar to wsInterpFunc, but instead of calling the function in the current script, you specify the handle of the script the function is in.

RETURN The wsvalue result of the called function. You own it and must destroy it when you are done with it.

SEE ALSO wsInterpURL, wsInterpFunc, wsInterpFuncArgs, wsInterpURLArgs

wsInterpURL

DESCRIPTION Stock call to interpret a function in a script.

SYNTAX

```
wsvalue wsInterpURL(wsContext wsc,char *url);
```

COMMENTS This is the stock interpret of a script function. It needs the host reference (`http://`, `file://`) part of the url (or a relative version), than name of the function (separated by '#') and any arguments in parenthesis (e.g. `(4,5)`).

RETURN The wsvalue result of the called function. You own it and must destroy it when you are done with it.

SEE ALSO wsInterpScript, wsInterpFunc, wsInterpFuncArgs, wsInterpURLArgs

wsInterpURLArgs

DESCRIPTION Call to interpret a script but passing args using wsvalues.

SYNTAX

```
wsvalue wsInterpURLArgs(wsContext wsc,
                        char *url,...);
```

COMMENTS Identical to wsInterpURL except you do not include any arguments in the URL. Instead you include a series of wsvalues as additional parameters to this function terminated by `NULL`.

RETURN The wsvalue result of the called function. You own it and must destroy it when you are done with it.

SEE ALSO wsInterpURL, wsInterpScript, wsInterpFunc, wsInterpFuncArgs

wsLoadScript

DESCRIPTION Preload a script and lock it in memory.

SYNTAX

```
script wsLoadScript(wsContext wsc,char *url);
```

COMMENTS If you know you are going to be executing a number of functions in a single script, you can load the script in once using this call and unload it when you are finished (if you don't unload it, it automatically unloads when the context is destroyed.) You can also use the handle to refer to this script, or you can continue to use full URLs which end up referring to the script.

wsLoadScript

RETURN Script handle or `NULL` on failure

SEE ALSO wsUnloadScript

wsNewContext

DESCRIPTION Create and initialize a new context.

SYNTAX

```
wsContext wsNewContext();
```

COMMENTS Each context is capable of executing one script at a time. Typically, you will need one context to run the scripts for your application, but you can have as many as you need. Each is created by a call to this routine. You must delete the context when you are done with it using `wsDeleteContext()`.

RETURN The new context or `NULL` if not enough memory was available to create a new context.

SEE ALSO wsDeleteContext

wsReportError

DESCRIPTION Print an error message and note that an error occurred.

SYNTAX

```
void wsReportError(wsContext wsc,
                   wschar *format,...);
```

COMMENTS This routine expects the format string to be printf-compatible. An error message is generated and when you return from the function, the script will exit.

RETURN none

SEE ALSO wsShouldExit

wsSetContextData

DESCRIPTION Set the context's data pointer.

SYNTAX

```
void wsSetContextData(wsContext wsc, void *data);
```

COMMENTS The data pointer is user defined; this routine sets the pointer, and you can later retrieve it with `wsGetContextData()`.

RETURN none

SEE ALSO wsGetContextData

wsSetContinueHandler

DESCRIPTION Set a function to be called periodically during execution.

SYNTAX

```
typedef wsbool (*wsContinueHandler)
               (wsContext wsc);
wsContinueHandler wsSetContinueHandler(wsContext
                  wsc, wsContinueHandler handler,
                  uint32 instrs,
                  uint32 *oldinstrs);
```

COMMENTS You continue function will be called every '`instrs`' bytecodes executed. Since bytecodes encompass very small actions, you will want to execute a number of bytecodes between calls to your routine or the vast majority of the time will be spent in your routine. 100, 1000, or more depending on your needed frequency. Your routine returns a boolean that if `False` causes execution of the script to be terminated.

wsSetContinueHandler

RETURN 'oldinstrs' is filled in with the old number of instructions between calls and the old handler is returned, both so you can restore them when done if you like.

SEE ALSO wsSetErrorHandler

wsSetErrorHandler

DESCRIPTION Install an error handler.

SYNTAX

```
typedef wsbool (*wsErrorHandler)(wsContext wsc,
                                 char *msg);
wsErrorHandler wsSetErrorHandler(wsContext wsc,
                                 wsErrorHandler handler);
```

COMMENTS Normally an error causes a message to be sent to the screen. If you install an error handler, the message will be sent to it instead. You can print it, store it to disk, or whatever. The return is usually `False`, but `True` will cause execution to try to continue. Be warned that this may not work. It is probably best to return `False` always except when you are using this mechanism to communicate with your wrapper functions.

RETURN The last error handler if you wish to restore it.

SEE ALSO wsSetContinueHandler

wsShouldExit

DESCRIPTION Tell the interpreter to exit.

SYNTAX

```
void wsShouldExit(wsContext wsc);
```

COMMENTS The script will terminate, analogous to `exit()` in a C program. However, the termination is not a failure; your return value is returned to the caller when you return from this function.

RETURN none

SEE ALSO wsReportError

wsUnloadScript

DESCRIPTION Release a script handle from wsLoadScript.

SYNTAX

```
void wsUnloadScript(wsContext wsc, script handle);
```

COMMENTS The script will no longer be locked in memory. It will be unloaded unless it is being used or another `wsLoadScript()` handle to it exists.

RETURN none

SEE ALSO wsLoadScript

wsValueAddUser

DESCRIPTION Create a new look on a wsvalue.

SYNTAX

```
wsvalue wsValueAddUser(wsContext wsc,wsvalue val);
```

COMMENTS The manual chapter on wsvalues explains extensively the concept of wsvalues and locks. This routine creates a new lock of the given wsvalue. All old locks remain.

RETURN The created lock

SEE ALSO wsDestroyValue

wsValueGetBool

DESCRIPTION Extract the boolean value from a wsvalue

SYNTAX
```
wsbool wsValueGetBool(wsContext wsc,wsvalue val);
```

COMMENTS This function is only valid if the wsvalue in question is of type `WS_VT_BOOLEAN` (see `wsValueType()`). If it is not, the return will be nonsensical.

RETURN The boolean value

SEE ALSO wsValueGetInt, wsValueGetFloat, wsValueGetString, wsValueGetLength, wsValueType

wsValueGetFloat

DESCRIPTION Extract the float value from a wsvalue

SYNTAX
```
float32 wsValueGetFloat(wsContext wsc,
                        wsvalue val);
```

COMMENTS This function is only valid if the wsvalue in question is of type `WS_VT_FLOAT` (see `wsValueType()`). If it is not, the return will be nonsensical.

RETURN The float value

SEE ALSO wsValueGetInt, wsValueGetBool, wsValueGetString, wsValueGetLength, wsValueType

wsValueGetInt

DESCRIPTION Extract the integer value from a wsvalue

SYNTAX sint32 wsValueGetInt(wsContext wsc,wsvalue val);

COMMENTS This function is only valid if the wsvalue in question is of type `WS_VT_INT` (see `wsValueType()`). If it is not, the return will be nonsensical.

RETURN The integer value

SEE ALSO wsValueGetFloat, wsValueGetBool, wsValueGetString, wsValueGetLength, wsValueType

wsValueGetLength

DESCRIPTION Get the length of the stored string

SYNTAX
```
sint32 wsValueGetLength(wsContext wsc,
                        wsvalue val);
```

COMMENTS This function is only valid if the wsvalue in question is of type `WS_VT_STRING` (see `wsValueType()`). If it is not, the return will be nonsensical.

RETURN The length of the string in characters.

SEE ALSO wsValueGetInt, wsValueGetFloat, wsValueGetBool, wsValueGetString, wsValueType

wsValueGetString

DESCRIPTION Extract the string value

SYNTAX
```
wschar *wsValueGetString(wsContext wsc,
                         wsvalue val);
```

COMMENTS WMLScript strings can have embedded '\0's in them. You use `wsValueGetLength()` to find the actual length of the string. For you convenience, a '\0' is always appended to the string. This means any returned string will always be '\0'-terminated for passing to C library functions. This is NOT counted in the string's length.

wsValueGetString

The returned pointer points to memory internal to the wsvalue and is valid until the wsvalue is destroyed. If, for instance, you get the string value of a parameter, when you exit the wrapper function, the valid will no longer be valid.

RETURN The string value.

SEE ALSO wsValueGetInt, wsValueGetFloat, wsValueGetBool, wsValueGetLength, wsValueType

wsValueNewBool

DESCRIPTION Create a new boolean wsvalue.

SYNTAX

```
wsvalue wsValueNewBool(wsContext wsc,wsbool val);
```

COMMENTS This function creates a new wsvalue which is initialized with the given boolean value. You own the wsvalue and must destroy it when you are done. Remember, wsvalues are readonly. If you directly access the wsvalue structure and change the value, you will break the interpreter.

RETURN The new boolean value

SEE ALSO wsValueNewEmpty, wsValueNewFloat, wsValueNewInt, wsValueNewInvalid, wsValueNewString

wsValueNewEmpty

DESCRIPTION Create a new empty string wsvalue.

SYNTAX

```
wsvalue wsValueNewEmpty(wsContext wsc);
```

COMMENTS This function creates a new wsvalue which is initialized as the empty string. You own the wsvalue and must destroy it when you are done. Remember, wsvalues are readonly. If you directly access the wsvalue structure and change the value, you will break the interpreter.

RETURN The new empty string wsvalue.

SEE ALSO wsValueNewBool, wsValueNewFloat, wsValueNewInt, wsValueNewInvalid, wsValueNewString

wsValueNewFloat

DESCRIPTION Create a new float wsvalue.

SYNTAX

```
wsvalue wsValueNewFloat(wsContext wsc,float val);
```

COMMENTS This function creates a new wsvalue which is initialized with the given float value. You own the wsvalue and must destroy it when you are done. Remember, wsvalues are readonly. If you directly access the wsvalue structure and change the value, you will break the interpreter.

RETURN The new float wsvalue.

SEE ALSO wsValueNewBool, wsValueNewEmpty, wsValueNewInt, wsValueNewInvalid, wsValueNewString

wsValueNewInt

DESCRIPTION Create a new integer wsvalue.

SYNTAX

```
wsvalue wsValueNewInt(wsContext wsc,sint32 val);
```

COMMENTS This function creates a new wsvalue which is initialized with the given integer value. You own the wsvalue and must destroy it when you are done. Remember, wsvalues are readonly. If you directly access the wsvalue structure and change the value, you will break the interpreter.

RETURN The new integer wsvalue.

SEE ALSO wsValueNewBool, wsValueNewEmpty, wsValueNewFloat, wsValueNewInvalid, wsValueNewString

wsValueNewInvalid

DESCRIPTION Create a new invalid wsvalue.

SYNTAX

```
wsvalue wsValueNewInvalid(wsContext wsc);
```

COMMENTS This function creates a new wsvalue which is initialized as the invalid value. You own the wsvalue and must destroy it when you are done. Remember, wsvalues are readonly. If you directly access the wsvalue structure and change the value, you will break the interpreter.

RETURN The new invalid value.

SEE ALSO wsValueNewBool, wsValueNewEmpty, wsValueNewFloat, wsValueNewInt, wsValueNewString

wsValueNewString

DESCRIPTION Create a new string wsvalue

SYNTAX

```
wsvalue wsValueNewString(wsContext wsc,
                         wschar *string,
                         int length);
```

COMMENTS This function creates a new wsvalue which is initialized with the given string value. You own the wsvalue and must destroy it when you are done. Remember, wsvalues are readonly. If you directly access the wsvalue structure and change the value, you will break the interpreter.

RETURN The new string value.

SEE ALSO wsValueNewBool, wsValueNewEmpty, wsValueNewFloat, wsValueNewInt, wsValueNewString

wsValueType

DESCRIPTION Get the type of the value

SYNTAX

```
int wsValueType(wsContext wsc, wsvalue val);
```

COMMENTS An enumeration is defined for you which specifies the possible types a wsvalue may have:

```
enum wsValueTypes
{
   WS_VT_INT = 0,
   WS_VT_FLOAT = 1,
   WS_VT_STRING = 2,
   WS_VT_BOOL = 3,
   WS_VT_INVALID = 4
};
```

wsValueType

RETURN The wsvalue's type.

SEE ALSO wsValueGetInt, wsValueGetFloat, wsValueGetBool, wsValueGetString, wsValueGetLength