

Interfacing the S232 Drivers to Watcom TDT Tech Note #157

Overview

This technote describes how to interface the S232 Drivers, specifically the DLLs, to Watcom Power++.

What is required?

In order to work with the System II API, Watcom users must have the DLL compiled with the stdcall interface. These files are the s2drv32s.lib and s2drv32s.dll.

Are there any modifications to make?

Previous users did make some modifications. Those modifications are illustrated below.

```
#define DODLL_VAR    __declspec(dllimport)
#define DODLL_FUN    __declspec(dllimport) __declspec(__stdcall)

void DODLL_FUN PA4atten(int din, float atten);
int DODLL_VAR DSPid[32];
```

Has anyone successfully used Watcom Power++?

Yes! The first was Dr. Don Gans. His email address is:
<dpg@riker.neoucom.edu>.

Is there any additional information?

Yes. The following is technical support documents from the Sybase website. Sybase supports Watcom Power++.

[FROM SYBASE]

Using Visual C++ EXEs and DLLs with Power++ and Watcom C/C++

SUMMARY: When providing an interface between modules that are built using two different vendor's compilers, it is important to ensure that the modules use common calling- and naming-conventions. This document discusses interface techniques for mixing modules created with Microsoft Visual C++ and modules created with Power++ or Watcom C/C++.

Document 44623
Last Revised: 06/13/97

ID:

Topic: App Development

Document Type: TechNote

Product: Watcom C/C++, Power++

Version: Not Version Specific

Platform: PC

Operating System: Windows 95, Windows NT

Document:

Using Visual C++ EXEs and DLLs with Power++ and Watcom C/C++

When providing an interface between modules that are built using two different vendor's compilers, it is important to ensure that the modules use common calling- and naming- conventions. This document discusses interface techniques for mixing modules created with Microsoft Visual C++ and modules created with Power++ or Watcom C/C++.

Notes:

- In this document, the term "modules" refers to executable images such as EXEs or DLLs. In general, you cannot statically link OBJs or LIBs that have been compiled with different vendor's compilers. This is due to, among other things, incompatibilities the vendors' run-time libraries.
- The document assumes you are using 32-bit Visual C++ for Win32 targets (Windows 95 and NT). The samples will not work with 16-bit Visual C++ for Windows 3.x. EXEs and DLLs.

To provide an interface between Visual C++ and Power++ or Watcom C/C++, you will need to export and import functions and/or variables. Function definitions and prototypes should include the extern "C" directive and the `__stdcall` keyword. The former forces C linkage (i.e., turns off C++ name mangling); the latter ensures the standard Win32 calling convention is used. Variable definitions and declarations should also include the extern "C" directive.

Power++ or Watcom C/C++ EXE calling Visual C++ DLL

If you want to write an Power++ or Watcom C/C++ EXE that calls functions or accesses variables exported from a Visual C++ DLL, then you must use the Visual C++ lib command to create an import library for the DLL. This import library can then be added to your Power++ or Watcom IDE project. The import library can also be used with the Watcom C/C++ wlink command if you are using command-line tools.

Calling Exported Functions

For each exported function in the DLL that is to be called from the EXE, do the following:

In the DLL source, define the function as:

```
extern "C" type __declspec(dllexport) __stdcall name( parameters )
{
    ...
}
```

In the EXE source, declare (prototype) the function as:

```
extern "C" type __declspec(dllimport) __stdcall name( parameters );
```

Accessing Exported Variables

For each exported variable in the DLL that is to be accessed from the EXE, do the following:

In the DLL source, define the variable as:

```
extern "C" { type __declspec(dllexport) name; }
```

Note: The braces ({}) are required. If you omit them, this becomes a declaration, not a definition.

In the EXE source, declare the variable as:

```
extern "C" type __declspec(dllimport) name;
```

Example

The following sample illustrates this technique:

```
msdll.cpp:
#include <stdio.h>

extern "C" { int __declspec(dllexport) var1 = 10; }

extern "C" int __declspec(dllexport) __stdcall func1( int i )
{
    printf( "func1(): var1 = %d\n", var1 );
}
```

```

    printf( "func1(): i = %d\n", i );
    return i * 2 + var1;
}

```

watexe.cpp:

```
#include <stdio.h>
```

```
extern "C" int __cdeclspec(dllimport) var1;
extern "C" int __cdeclspec(dllimport) __stdcall func1( int i );
```

```
void main(void)
{
    printf( "main(): var1 = %d\n", var1++ );
    printf( "main(): func1 = %d\n", func1( 5 ) );
}

```

If you are building the EXE with Power++, select Files from the View menu to open the Files view window. Then in the Files view window, select Add File... from the File menu to add the import library to the Power++ project.

If you are building the EXE with Watcom C/C++ and you are using the Watcom IDE, click on the Target window for the Target that uses the DLL. Then, select New Source... from the Source menu to add the import library to the Watcom IDE project.

To build this sample using command line tools, follow these steps:

1. Compile and link the DLL:

```
cl /c msdll.cpp
link /dll msdll.obj
```

2. Create an import library for the DLL:

```
lib /NAME:msdll.dll /OUT:msdll.lib
```

3. Compile and link the EXE using the import library generated in step 2:

```
wpp386 watexe.cpp
wlink sys nt n watexe f watexe l msdll
```

Note: If you want to use the __cdecl calling convention instead of

`__stdcall`, simply replace all occurrences of `__stdcall` in the above sample with `__cdecl`.

Visual C++ EXE calling Power++ or Watcom C/C++ DLL

If you want to write a Visual C++ EXE that calls functions or accesses variables exported from an Power++ or Watcom C/C++ DLL, then you will need to create a module definition (DEF) file for the DLL. Using the Visual C++ `lib` command, you can create an import library from the DEF file. This import library can then be added to your Visual C++ project.

Calling Exported Functions

For each exported function in the DLL that is to be called from the EXE, do the following:

In the DLL source, define the function as:

```
extern "C" type __declspec(dllexport) __stdcall name( parameters )
{
    ...
}
```

In the EXE source, declare (prototype) the function as:

```
extern "C" type __declspec(dllimport) __stdcall _name( parameters );
```

Note: The leading underscore in the function name is required.

In the DEF file, add the following line:

```
EXPORTS _name@n
```

where `n` is the size of the parameter list. To calculate `n`, take the size of each parameter, round it up to the nearest multiple of 4, and sum over all parameters. For example, if the function takes one `int` and one `char` as parameters, then the value of `n` is 8, because an `int` takes 4 bytes and a `char` takes 1, which is rounded up to 4. If you are still not sure how to calculate `n` for a given function, then you can use the `wlib` command to generate a listing from the DLL.

Accessing Exported Variables

For each exported variable in the DLL that is to be accessed from the EXE, do the following:

In the DLL source, define the variable as:

```
extern "C" { type __declspec(dllexport) name; }
```

Note: The braces ({}) are required. If you omit them, this becomes a declaration, not a definition.

In the EXE source, declare the variable as:

```
extern "C" type __declspec(dllimport) _name;
```

Note: The leading underscore in the variable name is required.

In the DEF file, add the following line:

```
EXPORTS _name
```

Example

The following sample illustrates this technique:

watdll.cpp:

```
#include <stdio.h>
```

```
extern "C" { int __declspec(dllexport) var1 = 10; }
```

```
extern "C" int __declspec(dllexport) __stdcall func1( int i )
{
    printf( "func1(): var1 = %d\n", var1 );
    printf( "func1(): i = %d\n", i );
    return i * 2 + var1;
}
```

msexec.cpp:

```
#include <stdio.h>
```

```
extern "C" int __declspec(dllimport) _var1;
extern "C" int __declspec(dllimport) __stdcall _func1( int i );
```

```
void main(void)
```

```
{
  printf( "main(): var1 = %d\n", _var1++ );
  printf( "main(): func1 = %d\n", _func1( 5 ) );
}
```

```
watdll.def:
LIBRARY watdll.dll
EXPORTS _func1@4
EXPORTS _var1
```

To build this sample using command line tools, follow these steps:

1. Compile and link the DLL:

```
wpp386 /bd watdll.cpp
wlink sys nt_dll initi termi n watdll f watdll
```

2. Create an import library for the DLL:

```
lib /DEF:watdll.def /OUT:watdll.lib
```

3. Compile and link the EXE using the import library generated in step 2:

```
cl /c msexec.cpp
link msexec.obj watdll.lib
```

Note: If you want to use the `__cdecl` calling convention instead of `__stdcall`, replace all occurrences of `__stdcall` in the above sample with `__cdecl` and remove the `@n` from each exported function name in the `watdll.def` file.

Notes

LoadLibrary() and GetProcAddress(): An Alternative to Using Import Libraries

If you do not want to use an import library to link your EXE, you can use the `LoadLibrary` and `GetProcAddress` Win32 API functions. The `LoadLibrary` function maps the specified executable module into the address space of the calling process and returns a handle to the module. You can then use this handle to get the address of any exported function in the DLL via `GetProcAddress`.

Again, be aware that the actual name of the exported function may be different than what you expect if the compiler used to generate the DLL is different from the one used to generate the EXE or DLL that calls it. You can view the exported functions using Quick View, a utility provided with Windows 95 and NT. Run the Windows Explorer, right-click on any DLL file and select Quick View. Then scroll down until you see Export Table and search for the name of the function you want to call. For Watcom C/C++ DLLs, you can use the command

```
wlib /l file
```

to generate a listing of the DLL file or import library file.

Exporting Entire Classes

In general, you cannot write an EXE that calls member functions or accesses member variables from an exported class in a DLL unless both the EXE and DLL were compiled with the same vendor's compiler. This restriction stems from the fact that C++ name mangling is not compatible across different vendor's compilers. As an alternative, you can write extern "C" cover-functions to handle the passing of data between the EXE and DLL.