VBInt 1.0

A Visual Basic DLL for the implementation of DOS/BIOS interrupts and functions.

By: Rick Esterling and Karl Peterson

*** WARNING ***

Consider yourself forewarned: These functions give you access to DOS and BIOS interrupts and services. These "functions", in case you are unfamiliar with them, include, but are not limited to, functions like "Format Disk Track", "Initialize Fixed Disk Table", ad naseum. Does that make you nervous? It should. Simply entering one single binary digit incorrectly can cause the complete and irrecoverable loss of all data on your hard drive. If you use this DLL, you use it with the complete and absolute understanding that you and you only are 100% utterly responsible for any damage that may be incurred as a result of having used this library. The author will not be held responsible for any ramifications suffered as a result of using the VBInt library. You use VBInt entirely at your own risk, no exceptions.

VBINT.DLL is **FREEWARE** and may be distributed with your applications without runtime fees.

All other files, if distributed, must be distributed in whole as one package. The package includes:

VBINT.DLL	VBINT.BAS	VBINT.WRI
INTDEMO.MAK	INTDEMO.FRM	INTDEMO.BAS
INTSUPT.BAS	INTDEMO.EXE	

Installing and Using VBINT.DLL

VBINT.DLL should be placed in the Windows system directory on any computer on which programs that use VBINT.DLL will be running. This is usually *C:\WINDOWS\SYSTEM.* VBINT.BAS should be loaded into any VB project in which you intend to call the VBINT library.

Requirements

You will need a reference manual that describes the services and explains how each is invoked. The user is expected to have a working knowledge of DOS interrupts and at least a basic understanding of how they are implemented. You will also need to know how to do binary manipulations and bitfield operations, some of which will be illustrated in this document and some of which are demonstrated in the demo included with VBINT.ZIP. VBInt is not meant to be used as a tutorial by which these concepts can be taught. It is provided for use by experienced programmers who already know about DOS interrupts and how to use them, but who could not do so in VB until now due to the inherent limitations of the Visual Basic programming environment.

Defining VBInt

Data is exchanged between Visual Basic and the DOS/BIOS services by means of a VB User Defined Type (UDT) called VBREGS. The UDT mirrors the actual registers that are found in the Intel 80-x series of microprocessors (8088, 80286, 80346, 80486, et al). Three functions comprise the VBInt library: VBInt(), GetSegment() and GetOffset(). The UDT and function declarations are defined in VBINT.BAS as follows:

```
Type VBREGS
  AX As Integer          ' These first four are called "general purpose" registers
  BX As Integer
  CX As Integer
  DX As Integer
  SI As Integer          ' These two are called the "index" registers
  DI As Integer
  cFlag As Integer       ' The carry flag
  DS As Integer          ' These are two of the four data segment registers.  The other two
  ES As Integer          ' (CS and SS) are not used by the DOS services or by VBInt.
End Type

Declare Function VBInt% Lib "VBINT.DLL" Alias "#1" (ByVal serviceNr%, inRegs As VBREGS,
 outRegs As VBREGS)
Declare Function GetSegment% Lib "VBINT.DLL" Alias "#2" (ByVal stringVar$)
Declare Function GetOffset% Lib "VBINT.DLL" Alias "#3" (ByVal stringVar$)
```

VBInt(), as shown above, receives three parameters:

1 - The service being requested
2 - The VBREGS UDT that contains the input register settings
3 - The output VBREGS UDT that will reflect the status of the registers after the

after the interrupt is generated. Note that the second and third parameters can be, and in fact usually are, the same variable.

GetSegment() and GetOffset() receive one parameter each; namely, the variable-length string variable for which the segment or offset address is being derived. See the section below titled "**Character Strings and VBInt**".

Calling a DOS Function

The best thing to do first is to jump right into an example. The question I am asked most often by VB programmers is how to determine the amount of free disk space available on any given drive. VB does not have a function for this and, in fact, neither does the Windows API. The DOS services, however, do include an internal function for deriving this information and for gathering other useful information about the disk as well.

Interrupt 21h/36h is an extended DOS function called "Get Disk Free Space". To use this function, you simply invoke service 21h with 36h loaded into the AH register. The logical drive number of the drive being queried is loaded into the DL register, or 0 can be used to query the current default drive, and then Interrupt 21 is requested. Consider the following example:

```
Function GetFreeSpace& (driveNr%)
  Dim regs As VBREGS, rtn%

  regs.ax = &H3600              ' Set AH = 36
  regs.dx = driveNr%            ' Set DL = logical drive number (0 = default, 1 = "A", etc)
  rtn% = VBInt(&H21, regs, regs)        ' Generate DOS interrupt
  If rtn% Then
    GetFreeSpace& = 0&          ' An unknown error occured
  ElseIf regs.ax = -1 Then
    GetFreeSpace& = -1          ' Drive not ready error
  Else
    GetFreeSpace& = CLng(regs.ax) * CLng(regs.bx) * CLng(regs.cx)
  End If
End Function
```

As you can see, the AX member of the VBREGS UDT was set to 3600h and the DX member was set to the value that was passed to this function via the driveNr% variable. The interrupt was generated by calling VBint(), passing &H21 (21h) as the service requested. DOS reference manuals tell us that this service returns the following information:

AX = sectors-per-allocated unit (cluster)
BX = number of available clusters
CX = number of bytes per sector
DX = total number of clusters

So, to ascertain the amount of disk space remaining, you start by multiplying the number of sectors-per-cluster (AX) by the number of available clusters (BX). So far, you have the total number of sectors available. To figure out how many bytes that equates to, multiply that value (AX * BX) by the number of bytes per sector (CX). In other words:

AX * BX * CX = total number of available bytes

In the sample above, I converted each paramater to a long before multiplying to prevent VB

overflow errors.

That's all there is to it!

Some other useful calculations can be made using the same information provided by this function:

```
Print CLng(regs.ax) * CLng(regs.cx)                    ' Bytes per cluster
Print CLng(regs.ax) * CLng(regs.cx) * CLng(regs.dx)    ' Total disk space
Print (regs.bx * 100) / regs.dx                        ' Percentage of free space
```

Error Handling

As you have seen in the declaration, VBInt() returns an integer. Please note that this value (the value returned from VBInt()) is *always* the exact same value as is held by the cFlag member of VBREGS which represents the carry flag in the registers. In fact, when VBInt() returns control to your program, it literally returns the value of VBREGS.cFlag itself. If the carry flag is set (i.e., non-zero), it's usually a very good indication that some kind of error occured. It is very important to realize, however, that each DOS Service has its own way of communicating to you that an error has been detected and that the carry flag is not always the only means by which that occurence is relayed to you.

To exemplify this point, try the example above again but this time send the function a 1 as its parameter (driveNr%) but do not put a disk in drive "A". You will discover that the program executes, the light for drive A comes on momentarily, and then the function just returns control to your program. Normally, DOS, and especially Windows, choke when you do something like attempting to read a floppy drive when there's no disk inserted. But in this example not only did nothing dramatic happen, but VBInt() returned a zero (and, in fact, the carry flag is not set) indicating that an error did not occur! How can this be? Some functions set the carry flag to indicate an error, some will just put an error code into one of the registers (usually AX, but not always) while leaving the carry flag untouched. Some do **both**! The important thing to remember is that each routine has its own way of communicating errors to you and that even if a specific function does not use the carry flag to indicate an error, it is still a good idea to check the carry flag anyway since it might be set if an error occurs that even DOS itself does not understand.

So how are you supposed to find out there's no disk in drive A in the example above? The DOS manuals tell us that if DOS realizes that this service has failed (as would be caused by the lack of a disk in the drive being queried), then DOS will load FFFFh into the AX register. Since VB does not support *unsigned* 16-bit integers, this same number appears as "-1" in VB when represented as an VB integer. In a 16 bit register, the two are equal: FFFFh = -1. Therefore, to find out if **this** interrupt failed, you need to check the value of the AX register when VBInt() returns and make sure it is not -1. Why, then, bother checking the return value from VBInt (which is the same thing as checking the cFlag member) since this service does not use the carry flag to indicate an error? For a very important reason: If your call to VBInt was set up incorrectly and a critical error occured, one that might have nothing to do with the "Get Disk Free Space" service itself, then obvioulsy the AX register may or may not have been loaded with a proper error code. In this situation, the carry flag is how you would (hopefully) find out that something went wrong. Therefore, a general rule can be defined: *Always check the carry flag,* even if it is not the normal means by which a service is supposed to indicate an error. Also, of course, check whatever means the service provides for error handling.

Character Strings and VBInt

There are basically two types of DOS interrupts that you will be requesting: (1) Those that work strictly with numbers, like the example shown above ("Get Disk Free Space"); and (2) Those that require character strings such as is necessary to rename or delete a file.

From DOS' point of view, a character string variable is nothing more than a number (a "*pointer*") that represents the location in memory where a string starts. Once DOS knows the address of a string, it starts there and keeps reading memory contiguously until an ASCII 0 is located, which acts as a terminating character for strings much in the same way as a period indicates the end of this sentence. When DOS needs to know the address of a string, it looks in two registers: one of the segment registers for the first part of the address, and one of the general purpose or index registers for the second part. The first of these two parts is called the "*segment*" address and the second is called the "*offset*" address. The segment address narrows down the location of the string to a 64K block of memory while the offset address specified where inside that 64K block the string is actually located.

It is very important to realize that Visual Basic handles string variables in a very unique way. In languages like assembly and C/C++, the programmer has to do quite a bit of work just to allocate, maintain, and deallocate string variables, particularly under Windows. To make string variables easier for the programmer to manipulate in Visual Basic, a unique internal string-handling routine was devised that is, unfortunately, incompatible with functions written in any other language. As such, special consideration has to be made to pass string variables between Visual Basic programs and functions in DLLs that were written in other languages.

First of all, VB itself has two types of string variables: *fixed-length* and *variable-length*. **Fixed-length VB string variables are <u>not</u> compatible with VBInt and will not work if used.** An example of a fixed-length variable declaration follows:

```
Dim stringVar As String * 100     ' Declares a fixed-length string variable capable of
                                  ' holding up to 100 characters
                                  ' This declaration is not compatible with VBInt
```

Variable-length strings, on the other hand, are compatible with VBInt, but they must be buffered or "allocated" if they are to receive information just as you would be required to do if you were passing a string variable to any other DLL. An example of declaring and allocating a variable-length string variable follows:

```
      Dim stringVar As String          ' Declare a variable-length string, but no memory
or:   Dim stringVar$                   ' has been allocated yet

      stringVar$ = String$(100, 0)     ' Now memory has been allocated for 100 characters.
```

What's the all-important difference between fixed- and variable-length string variables in VB? Since VB string variables are not compatible with other languages, the Visual Basic kernel (VBRUN300.DLL) automatically intercepts your string variables whenever they are passed to a DLL and each is converted it to a variable that is compatible with the standard ASCII-0 terminated string thereby making the variable compatible with the DLL. With variable-length string variables, VB assumes that you've done all the of necessary allocating of memory and simply passes to the DLL the memory address of your variable. Fixed-length string variables, on the other hand, are <u>not</u> passed directly to DLLs. When the VB kernel realizes a fixed-length string is being passed to a DLL, VB makes a *copy* of the variable; i.e., a temporary "buffer", and this buffer is what is actually passed to the DLL. When the DLL returns control to the VB application, the VB kernel intervenes again and copies the contents of the buffer back into the VB fixed-length string and the temporary buffer is deallocated and deleted.

When you need to pass a character string to VBInt, you do so not by passing the string itself as you would within a VB program or even with a DLL; rather, you determine the segment address and offset address of the string and load these two values into the appropriate members of the VBREGS UDT according to the service being requested. Then, just as you would with a "simple" interrupt, you pass the UDT to VBInt() which loads the actual registers with the address specified in the UDT and generates the interrupt. How do you get the address of the string variable? That is the purpose of the other two functions in VBINT.DLL.

```
GetSegment()        ' Returns the segment address of a string variable
GetOffset()         ' Returns the offset address of a string variable
```

Obviously, it would do no good to obtain the segment and offset address for a variable that is only a temporary buffer, and it is for this reason that fixed-length string variables **cannot** be used with VBInt. By the time you get the address back from GetSegment() or GetOffset(), the address will no longer be valid. Consider this example of a function that will not work:

```
Sub BadExample()
  Dim stringVar As String * 100        ' Declare a fixed length string of 100 characters
  Dim segAddress%, offAddress%         ' Declare variables to hold address

  segAddress% = GetSegment(stringVar)
  . . .
```

Already, we can see the problem. When *stringVar*, a fixed-length variable, is passed to GetSegment(), the VB kernel intervenes as described above by creating a temporary buffer and passing the address of **the temporary buffer** to GetSegment(); the address of *stringVar* itself does **not** get passed and is completely hidden away somewhere inside the VB kernel. Still, GetSegment() does its job by returning the segment address of the variable that was passed to it (the temporary buffer, as it turns out) which is assigned to segAddress% when GetSegment() returns. The action of GetSegment() returning control to the VB program triggers the automatic intervention of the VB kernel again, which copies the contents of the temporary buffer, whatever they may be, into *stringVar* and then the temporary buffer is deallocated and deleted. The problem, of course, is that segAddress% now points to a location in memory where the temporary buffer used to be which, of course, has since been deleted by the VB kernel. This address has nothing whatsoever to do with the address of *stringVar*. In fact, since the temporary buffer has been deleted, segAddress% doesn't point to anything useful whatsoever and if referenced, stands a very good chance of crashing the entire Windows operating environment!

When a variable-length string variable is passed to a DLL, on the other hand, the VB kernel still intervenes, but it does not create a temporary buffer. It passes the actual address of the VB variable to the DLL. With that in mind, consider the following code:

```
 Sub Example()
   Dim stringVar$, segAddress%              ' Declare a variable-length string variable
                                            ' and an integer
   stringVar$ = String(100, 0)
   segAddress% = GetSegment(stringVar$)     ' Actual address of stringVar is passed to
                                            ' GetSegment()
   Print Hex$(segAddress%)                  ' Prints segment address of stringVar in hex
End Sub
```

The information above certainly requires an advanced understanding of how string variables are referenced by VB, DLLs and even DOS itself. If you are not interested in all of these details, that is quite alright as long as remember the rule: Fixed-length string variables cannot be used with VBInt; variable-length string variables can.

Finally, there are two more important things to remember when passing string variables to any DLL functions including the functions in VBINT.DLL: (1) If the DLL is going to place a string into the variable passed by you, then you as the VB programmer are responsible for making sure that the variable is large enough to accept the maximun number of characters that might be loaded into your variable. DLLs **do not** allocate space for your variable before inserting characters into it - you must do it prior to the variable being passed to the DLL. The example above shows 100 bytes of memory being allocated to *stringVar.* (2) If the DLL is going to read characters out of your variable, then you **must** ensure that you terminate the string with an ASCII 0. If you are sending a variable to function and you send it ByVal, then VB will do this for you automatically. With VBInt, however, you will be defining a string variable, placing data into this variable and then sending the segment/offset address of the variable to VBInt so that the string can be located and evaluated. Obviously, since only the segment/offset address, stored in an UDT nonetheless, is actually being passed to VBInt, there is no possible way that VB could understand what you are doing so there is no way VB will jump in and assist you by terminating your strings. Therefore, you must do it manually lest DOS continue reading characters from memory until some random ASCII 0 is located, if ever. This is performed easily enough:

```
myPath$ = "C:\WINDOWS" & Chr$(0)     ' How to terminate a string variable
```

Applying everything that is discussed above, we are ready to call a DOS Service that uses a character string. The following example invokes the extended DOS service, "Rename File" (Int 21h/56h) which uses two character strings. This service is much better than the DOS rename command insomuch as it, like VB's *Name* command, allows you to specify different subdirectories as you rename the file. This gives you the ability to logically move a file across subdirectories without performing a physical copy as long as the file remains on the same device. In the event of an error, this service places an error code in the AX register (see source code) and it sets the carry flag as well.

```
Sub RenameFile (origFilename$, newFilename$)
  Dim regs As VBRegs, rtn%
  ' The two parameters received by this function must be variable-length string variables

  ' Ensure the two variables received by this function are properly terminated with ASCII 0
  If Instr(origFilename$,Chr$(0)) = 0 Then
    origFilename$ = origFilename$ & Chr$(0)
  End If
  If Instr(newFilename$,Chr$(0)) = 0 Then
    newFilename$ = newFilename$ & Chr$(0)
  End If

  regs.ax = &H5600                          ' DOS service requested
  regs.ds = GetSegment(origFilename$)       ' Get segment address of first input string
  regs.dx = GetOffset(origFilename$)        ' Get offset address of first input string
  regs.es = GetSegment(newFilename$)        ' Get segment address of second input string
  regs.di = GetOffset(newFilename$)         ' Get offset address of second input string

  rtn% = VBInt(&H21, regs, regs)            ' Generate the interrupt
  If rtn% <> 0 Then                         ' Check return value (carry flag)
    Select Case regs.ax                     ' Error handling
      Case &H2
        MsgBox "File not found: " & origFilename$, MB_ICONEXPLANATION, "VBInt Error"
      Case &H3
        MsgBox "Path not found", MB_ICONEXPLANATION, "VBInt Error"
      Case &H5
        MsgBox "Access denied", MB_ICONEXPLANATION, "VBInt Error"
```

```
      Case &H17
        MsgBox "Not same device", MB_ICONEXPLANATION, "VBInt Error"
      Case Else
        MsgBox "An unknown error has occured: " & regs.ax, MB_ICONEXPLANATION, "VBInt
          Error"
      End Select
    Else
      MsgBox origFilename$ & " has been renamed to " & newFilename$   ' Success!
    End If
  End Sub
```

Binary Manipulations from VB

This section has very little to do with VBInt specifically and more to do with how binary operations are performed in VB; i.e., AND, OR, Shift Right, etc. An effective way to demonstrate these operations is with the basic DOS service that retrieves the current system time, Int 21h/2Ch.

The "Get Time" function returns the current system time in the registers specified below:

CH - Hours
CL - Minutes
DH - Seconds
DL - Hundredths of seconds

Given that information, here is how the values of those registers can be determined in VB:

```
Sub GetTime ()
 Dim regs As VBRegs, rtn%, cTime$

 regs.ax = &H2C00                          ' Load 2Ch into AH
 rtn% = VBInt(&H21, regs, regs) ' Generate the interrupt

 cTime$ = Format$((regs.cx And &HFF00) \ 256, "00")
 cTime$ = cTime$ & ":" & Format$((regs.cx And &HFF), "00")
 cTime$ = cTime$ & ":" & Format$((regs.dx And &HFF00) \ 256, "00")
 cTime$ = cTime$ & "." & Format$(regs.dx And &HFF, "00")
 Print cTime$
End Sub
```

As you can see, this service requires that we evalute the hi and lo bytes of the CX and DX registeres independently. Normally, a bitwise Shift Right operator (>>) would be used to perform such a task, but VB does not have shift operators. A shift to the right, however, is nothing more than a little base 2 division, so we can easily replicate the same functionality. To determine the current hour, we are only interested in the hi byte of the CX register. That means we can get rid of the lo byte by And-ing it with 0:

```
HiByte% = regs.cx And &HFF00 ' Zeroizing the lo byte of CX
```

Now we have the value of just the hi byte, but it cannot be evaluated yet since it is still positioned over to the left. We need to shift everything to the right by **eight bits** in order to get a true representation of the hi byte value. Two to the **eighth** power is 256, so all we have to do is divide the value by 256 and we will have effected the same thing as shifting right eight bits.

```
HiByte% = HiByte% \ 256
```

Then use VB's Format() function to pad zeros in where there might be a single digit number:

```
Print Format$(HiByte%,"00")      ' Prints the hour portion of the current time.
```

Now, we have the printed the true, unadulterated value of the hi byte of CX. To get the value of the lo byte, we simply And the value with &H00FF. No shift is necessary.

```
LoByte% = regs.cx And &HFF    ' Leading zeros are assumed (&H00FF = &HFF)
```

INTDEMO

So far, I have not really shown you anything that demonstrates the real usefulness and power of VBInt and the massive amount of information that is opened up to you by having gained access to the DOS/BIOS services and functions. I have used simple and even redundant examples, chosen simply because I felt they would clearly illustrate the functionality of VBInt regardless of how irrelevant the code itself may be. If you are starting to feel like this is all a lot of work with only a little payback, then one look at Karl Peterson's sample program, INTDEMO, will change your mind.

INTDEMO is a perfect example of the type of system information and program functionality that is available to you as a VB programmer when armed with VBINT.DLL. There is far too much magic in this demo for me to explain here. Rather, I recommend printing the source code and studying the operation of each function that is demonstrated.

Among other things, one truly amazing feat comes out of Karl's demo. Remember that a string variable is nothing more than a pointer (an address) to a contiguous block of memory that is capable of storing ASCII characters. UDTs, declared correctly, are really the same thing. Karl exploits this fact by aliasing GetSegment() and GetOffset() so that they receive as a parameter a pointer to a UDT instead of a pointer to string variable. The address of the UDT is then passed to VBInt to collect information about the files that are being scanned (date, time, size, etc), which subsequently fills up the members of the UDT with the proper information, ALL IN ONE CALL! Specifically, check out the declarations for UDTSegment() and UDTOffset(), the UDT called DTAType, and the functions called FileFindFirst() and FileFindNext(), all of which may be found in INTDEMO.BAS.

I have singled out that particular section of Karl's demo because of its *ingenious* implementation, but the fact of the matter is the that **all** of the source code for INTDEMO is a must for your perusal. Furthermore, despite the enormous functionality that is illustrated by the entire demo, it serves only as a preliminary example of what can be accomplished with VBInt. The demo contains only a *fraction* of the types of operations that can be performed.

Compatibility and Limitations

VBInt has been tested on Windows 3.0, Windows 3.1 and Windows NT. The DLL is compatible with all three, but some of the DOS Services themselves may be only partially implemented on each platform. Factors such as whether Windows is running in Enhanced or Standard mode, or having Windows configured to use 32-bit disk access, can also affect the operation of specific DOS services. "Undocumented" services are not specifically supported by VBInt although some of them may work. Truename (Int 21h/60h), for example, does not.

Several DOS interrupts and functions are not supported in the Windows environment as listed below:

Interrupt	Description
20h	Terminate Program
25h	Absolute Disk Read
26h	Absolute Disk Write
27h	Terminate and Stay Resident
21h/00h	Terminate Process
21h/0Fh	Open File with FCB
21h/10h	Close File with FCB
21h/14h	Sequential Read
21h/15h	Sequential Write
21h/16h	Create File with FCB
21h/21h	Random Read
21h/22h	Random Write
21h/23h	Get File Size
21h/24h	Set Random Record Number
21h/27h	Random Block Read
21h/28h	Random Block Write

The following interrupts and functions are "partially" supported in the Windows environment since they behave differently in protected mode than they do in real mode. Please refer to the Windows SDK if you have any questions regarding the usability of any of these interrupts/functions:

Interrupt	Description
21h/25h	Set Interrupt Vector
21h/35h	Get Interrupt Vector
21h/38h	Get/Set Current Country Information
21h/4402 - 4405h	Send/Receive Control Data
21h/440Ch	Generic IOCTL for Character Devices
21h/6501-6506h	Get Extended Country Information

Technical Support

If you are having a problem with VBInt, the best means by which to obtain help or technical advice is to leave a message for either Karl or me in Section 5, "Programming", of the MSBASIC forum on Compuserve (see credits for CIS addresses). Karl and I frequent this forum daily and are as quick to offer our expertise as we are to receive the expertise of others.

We do ask you to remember that VBInt is *freeware*. We are not guaranteeing support and we cannot promise results. A lot of work has already gone into VBInt and we truly hope that others can benefit from this effort. We are anxious to help those who are anxious to help themselves, but this project does not pay the bills so we cannot promise anything. Generally, we will help you if we can - if for some reason we cannot, we hope you will understand. If you don't understand, then delete VBINT.DLL and pretend like all of this never happened.

Technical Specifications

VBINT.DLL is written in Visual C++.
DOS Protected Mode Interface (DPMI) is used extensively to implement the DOS interrupts via 16-bit inline assembly.

Using VBInt With C/C++

There is no great advantage to using VBInt with C/C++ since programs written in C/C++ can access the registers and interrupt services directly. If, however, you like the implementation of this DLL and want to use it in your C/C++ programs, the following header information is provided.

```
struct VBREGS {         // Note that this structure is not compatible with any of the
   int ax,bx,cx,dx;     // register structures already defined in DOS.H
   int si,di;
   int cflag;
   int ds,es;
};

int VBInt(int serviceNr, VBREGS FAR* VBinreg, VBREGS FAR* VBoutreg);
UINT GetSegment(LPSTR lpszVar);
UINT GetOffset(LPSTR lpszVar);
```

Credits

VBINT.DLL
Rick Esterling is a computer scientist for Boeing Information Services, contracted to NASA at Marshall Space Flight Center in Huntsville, AL. He primarily uses Visual C++ and Visual Basic to develop Windows client/server applications and utilities. He also uses C, C++, and Clipper 5.2 to develop applications and utilities for the DOS environment. He enjoys golf, flying and playing gigs (keyboards/vocals) in his classic rock-and-roll band in and around Huntsville, where he lives with his girlfriend of two years. Rick can be contacted at:

Rick Esterling
Three Cruse Alley
Huntsville, AL 35801

Internet: rick.esterling@msfc.nasa.gov
CIS: 7332,702

INTDEMO.EXE
Karl Peterson is a Senior Technical Transportation Planner/GIS Analyst for the Southwest Washington Regional Transportation Council. Cartography and spatial analysis of demographic, economic and natural data, as well as being the general PC support person and network administrator for the agency keep him busy. Programming has always (20 years now with Basic!) been a "hobby", and he tries to fit it into the work program whereever it can be "justified". Karl lives with his wife, two sons, and mongrel border collie in Vancouver less than 25 air miles from Mt. St. Helens. Besides writing INTDEMO, Karl was also instrumental in specifying the implementation of the VBINT library itself. Karl can be contacted at:

Karl Peterson
Regional Transportation Council
1351 Officers' Row
Vancouver, WA 98661

CIS: 72302,3707