HP VEE Application Note

Issue No. 2, December 1994

Getting Out Through Plug-in Cards

Overview

A customer called us wanting to know how to access a PC plug-in A/D card from HP VEE for Windows. There are several alternative ways to accomplish this (through the use of files shared with other Windows applications which you could write, using DDE, etc.), however, the customer was specifically interested in integrating the measurements from the A/D card directly into VEE. VEE for Windows allows you to create a Dynamic Link Library (DLL: an externally compiled function, usually written in C) and pass data between it and a VEE "call function" object. Your DLL then functions in much the same way as a standard VEE object: it accepts data and is scheduled for execution in the same way. The remainder of this article discusses the creation of this application environment.

What we'll do

- We will create a DLL, which can be called from VEE for Windows, that allows you to peek and poke the registers on I/O-mapped PC plug-in cards. Specifically, we want the capability to read and write a byte or a word of data from or to a specified I/O port.
- We'll explain the mechanics of building a DLL for use with VEE.
- We'll explain some of the semantics of DLL's to include memory models, pointer declarations, function declarations, make files, and definition files.
- We'll explain the mechanics of hooking the DLL into HP VEE for Windows.
- We'll show you how to send data to and read data from I/O ports.

What we won't do

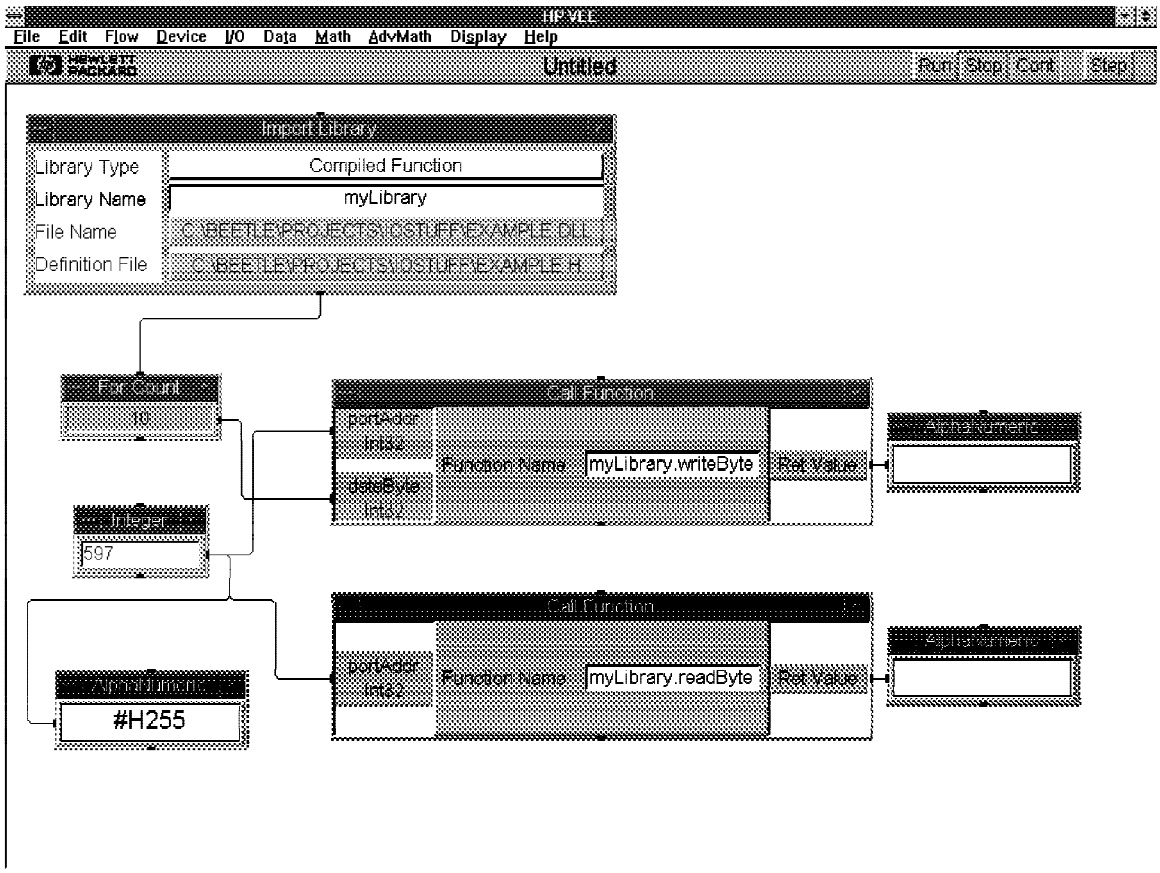
- We won't use anything other than a Microsoft® compiler. All of the popular Windows compilers can be used to build DLL's, but the mechanics differ between them, and a discussion of the intricacies of each would detract from the content of this article. Also, we have used a capability to communicate with the registers on PC plug-in cards that exists in every compiler, but not every compiler uses the same syntax. So, we will limit our discussion to Microsoft C compilers.
- We won't make any attempt to address the arbitration of two or more applications trying to use this DLL simultaneously. The issue of multiple processes trying to access a shared piece of hardware through a commonly-attached DLL presents an entire range of concerns that are beyond the scope of this article.

- We ***absolutely will not*** take responsibility for any damage caused by the use of this DLL or caused by applications built using the contents of this article. There is no way to prevent a DLL from causing physical damage by writing to an I/O port. Writing to the disk drive controller could be a potential problem, for instance. If you use the DLL, be careful to make sure you are using the correct port address.

- Do any memory-mapped I/O. There are more issues with memory mapping than there are with I/O mapping.

- Handle hardware interrupts. To do this properly requires either an MS-DOS® terminate-and-stay-resident (TSR) application, an MS-DOS device driver, a Windows installable device driver, a Windows Virtual Device Driver(VxD), or a SICL (HP's Standard Instrument Control Library) TULIP (The User-Land Interface Protocol) driver.

The VEE Program



The preceding picture is a screen dump of this VEE for Windows application. The application works by "importing" the DLL into the VEE environment, calling one of the functions in the DLL that writes data to a port, then calling one of the DLL functions that reads data from a port. This is a simple test to see that the data we thought we wrote actually got to the intended location. In this case, we used an HP BASIC Language Coprocessor (both the "Viper" and "Hyper Viper" varieties) to supply us a register address we could read and write.

The "Import Library" is the object responsible for attaching the DLL into VEE's address space, thereby making any exported functions the DLL contains directly callable. Notice that the "Import Library" box has several selection fields and an editing field. For this application the "Library Type" field specifies a "Compiled Function", the "File Name" field is the DLL's path and file name, and the "Definition File" is the path and file name of a file that describes to VEE what data input and output pins are necessary to pass information between itself and your DLL. It also specifies the names by which you will identify the functions within your DLL.

The top-most "Call Function" object has data supplied from an iterater and an integer constant. The integer constant specifies the address of the port we wish to write data to. Notice that the value it contained is 597. Most PC card manufacturers specify a card "base address" (which is in actuality an I/O port address) in hexadecimal notation. 597 is the decimal representation of hex 255, and you can get this value into an integer constant by typing into it the value "0x255". On the other data-in pin, we specify the value we wish to write to the port. This value is supplied by the iterater, so we will write ascending values to the I/O port numbered 0x255. The "Ret Value" data-out pin is the value the DLL thinks it wrote to the port. In this case, our register is only a byte wide, so we can't write a value greater than 255 (ffff hex) to it.

The other "Call Function" box takes as its input a port from which to read a byte of data. Its "Ret Value" data-out pin is the value the DLL function read from the port.

The VEE Definition File

We stated previously that VEE needs a definition file to know what input and output pins to create on the objects which call the functions in your DLL. The definition file for this application looks like this:

```
long writeByte(long portAddr, long dataByte)
long writeWord(long portAddr, long dataWord)
long readByte(long portAddr)
long readWord(long portAddr)
```

It specifies four functions which write and read bytes and words, respectively. Notice that all of the function parameters are specified as data type long. VEE's representation of an integer is a 32-bit signed number, which corresponds to a long in most compilers.

Some Things You Need to Know to Write a VEE DLL

Writing a DLL to perform data filtering and other tasks is actually fairly straightforward, so long as you are aware of a few things. Although you can use DLL's to perform some difficult Windows tasks, the majority of people using them with VEE will probably not need that level of capability. What we will discuss in this section covers the topics necessary for building a DLL to do the more usual tasks.

Pointers

Windows uses a segmented memory architecture, instead of a contiguous address space. Data and code segments are distributed throughout a number of 64-Kbyte segments.

Pointers to Data

Pointers to data can be either "near" or "far". A "far" pointer references data that is not in the same 64K data segment as the data that is local to the code segment. This is the case when VEE calls a DLL. In order to pass pointers between VEE and a DLL, any pointers must be declared with the keyword FAR (declared in <windows.h>) in your DLL source file. The most common reason for needing to pass pointers is that you are working with arrays. Instead of passing an entire array on the stack (a very resource-intensive way to do things), VEE passes pointers to array data. A "far" pointer, however, can reference no more than 64 Kbytes of data. In the event that your arrays are greater than 64 Kbytes in size, you will need to declare your pointers as "_huge". As an example, a declaration for a pointer to an array of VEE integers, whose total number of bytes is less than 64K, would look like this:

```
<vee data type> FAR *<array identifier>
long FAR *myVeeArray;
```

A pointer to an integer array whose total size would be greater than 64 Kbytes would look like:

```
long _huge *myVeeArray;
```

Pointers to Functions

Functions can also be located either in or out of the 64K code segment you are currently executing. By definition, a function within a DLL will be in a different code segment than VEE's, so DLL functions must be declared FAR in your DLL source.

Calling Conventions

Windows applications and DLL's support a number of different function calling conventions. The two you are most likely to see are ***PASCAL*** and ***CDECL***. The PASCAL calling convention puts function arguments on the stack in a left-to-right order. This convention is faster then the C calling convention, but has the side effect that you can't have a variable number of pass parameters. The C run-time function printf() is an example of a function that uses a variable parameter list. CDECL (the C calling convention) puts function arguments on the stack right-to-left. This is marginally slower but allows variable parameter lists. In order for functions to work properly when called from an application (your DLL when it is called from VEE's "Call Function" box, for instance), the function call and function definition must agree on the convention. As of HP VEE 3.0, VEE DLL's can use both the CDECL convention, as well as Pascal.

So, a CDECL function declaration in your DLL source would look like this:

```
<return type> FAR CDECL <function name>(<data type> param1,...)
long FAR CDECL foo(long bar)
```

A Pascal function declaration in your DLL source code would look like this:

```
<return type> FAR PASCAL <function name>(<data type> param1,...)
long FAR PASCAL foo(long bar)
```

A CDECL function declaration in the ".def" file that the VEE import library object uses would look like this:

```
<return type> <function name>(<data type>param1,...)
long foo(long bar)
```

A Pascal function declaration in the ".def" file that the VEE import library object uses would look like this:

```
<return type> _pascal <function name>(<data type> param1,...)
long _pascal foo(long bar)
```

Memory Models

Windows and MS-DOS compilers support several memory models to help deal with the segmented architecture. When you compile your DLL, the best choice for a memory model is large. The linker will also need to bind against the appropriate libraries for the memory model you compiled under. We'll show an example of this when we walk through the DLL make file.

Required Functions

Every DLL requires two functions: ***LibMain*** and ***WEP***. Windows calls these functions when a DLL is attached by any application for the first time and when the DLL is detached by the last application respectively. They must be declared FAR PASCAL. Generally, they are used to perform systemic initialization and cleanup. They both return 1 to indicate success and 0 to indicate failure. Our DLL does no systemic initialization or shutdown, so we just return 1.

DLL Definition Files

DLL definition files are used to build extra information into Windows DLL's and applications. The specific requirements for a DLL is that it may have only one data segment, no matter what memory model it was compiled with. We have an example definition file in this article.

The DLL

This is the source for all of the files necessary to build the DLL. The central facility we use is a call to the functions ***outp()***, ***outpw()***, ***inp()***, and ***inpw()***. These functions write a byte, write a word, read a byte, and read a word from a specified I/O port, respectively.

DLL Function Source

```
#if defined(MSDOS)      // none of this will work with a UNIX® shared library

#include "examplex.h"   // contains function prototypes

// The functions LibMain and WEP are required in MS Windows
// DLL's.  LibMain is called once and only once when the DLL
// is first attached by any Windows application.  Generally,
// LibMain is where you do any systemic initialization necessary
// for handling attachments from multiple clients.  LibMain returns a
// 1 to indicate success and a zero to indicate failure.  We do no
// systemic initialization, so we just return a 1.
//
// WEP is called when the last application that had attached the DLL
// detaches it.  It is called only once.  Generally, you do systemic
// cleanup or shutdown in this function.
```

```
int FAR PASCAL LibMain( HANDLE  hModule, WORD wDataSeg,
   WORD cbHeapSize, LPSTR lpszCmdLine){

   return 1;
}

int FAR PASCAL WEP (int nParm){

    return 1;
}

// These functions are the compiled functions VEE will call to read
// and write information from and to I/O ports.  They are declared
// FAR and CDECL.  The FAR qualifier means that the code for this function
// resides in a memory segment different than the memory segment which
// contains the calling code.  The CDECL qualifier affects how data is
// passed between functions.  Calling and called functions must agree
// on the calling convention.  CDECL means the calling function will
// place parameters on the stack in a right-to-left order.  The functions
// LibMain and WEP (which stands for Windows Exit Procedure) are both using
// the PASCAL calling convention which places parameters on the stack
// left-to-right.

// This function writes one byte to the port at the specified address.
// "portAddr" is the data supplied on the top-most data input pin of VEE's
// call function box and should contain the integer address of the
// port you wish to write information to.  Note that most card
// manufacturers specify port addresses in hexadecimal notation.
// To address the intended port, "portAddr" must be the decimal integer
// equivalent of the hexadecimal port address.
// "dataByte" is the data you want to write to the port.  It is supplied
// to the bottom-most data input pin of the call function box.  Its value
// will be truncated such that only the least-significant byte of data is
// written to the port.

long FAR CDECL writeByte(long portAddr, long dataByte){
   int returnVal;       // the value the outp function thought it wrote
   unsigned port;       // the address of the port you want to write to
   int data;            // the data we want to write

   // The VEE integer data type (long) is a 32-bit signed integer, while
   // the "port" argument to the outp() function is an unsigned
   // int (a 16-bit integer).  We inclusively "and" the "portAddr"
   // argument passed from VEE with two bytes of all 1's, effectively
   // truncating the value to its least significant byte.  This is
   // a little pedantic but illustrates the data type differences.
   // The "data" argument to the outp() function is an integer, so we
   // "and" the "dataByte" value VEE passes in with one byte of all 1's.
   // The VEE function call box return pin is the data that this function
   // thinks it wrote to the port.  For instance, if you passed in the value
   // 1234567 to write to a port, the function would return 135.  The hex
   // value of 1234567 is 12D687.  When "and'ed" with ff, the data byte
   // written to the port will be hex 87 or decimal 135.

port=(unsigned)(0xffff & portAddr);
```

```
   data=(int)(0xff & dataByte);

   // write the byte, saving the value the outp() function thought it
   // wrote in the variable "returnVal"

   returnVal=outp(port, data);

   // send the results of the call to outp() to VEE's function call box
   // return pin.
   return((long)returnVal);
}

// This function writes a 16-bit word to the specified port

long FAR CDECL writeWord(long portAddr, long dataWord){
   unsigned returnVal;     // the value the outpw function thought it wrote
   unsigned port;          // the address of the port you want to write to
   unsigned data;          // the data we want to write

   port=(unsigned)(0xffff & portAddr);

   // note that "data" in this case is an unsigned int (16 bits)

   data=(unsigned)(0xffff & dataWord);

   returnVal=outpw(port, data);

   return((long)returnVal);
}

// This function reads a byte from the specified port.

long FAR CDECL readByte(long portAddr){
   int data;           // the data we read from the port
   unsigned port;      // the port from which we will read a byte

   port=(unsigned)(0xffff & portAddr);

   data=inp(port);

   return((long)data);
}

// This function reads a word from the specified port

long FAR CDECL readWord(long portAddr){
   unsigned data;      // the data we read from the port
   unsigned port;      // the port from which we will read a byte

   port=(unsigned)(0xffff & portAddr);

   data=inpw(port);

   return((long)data);
}
```

```
#endif
```

DLL Definition File

```
LIBRARY         ioex
DESCRIPTION         'dll for port io'
EXETYPE         WINDOWS
STUB              'WINSTUB.EXE'
CODE              PRELOAD MOVEABLE DISCARDABLE
DATA                PRELOAD MOVEABLE SINGLE
HEAPSIZE        8192
EXPORTS
  WEP       @1      RESIDENTNAME
  LibMain
  _writeByte
  _writeWord
  _readByte
  _readWord
```

Of note here are the entries for ***CODE***, ***DATA***, and ***EXPORTS***. The CODE segment is marked as ***PRELOAD***, which means that the Windows loader will load this DLL into memory immediately upon loading an application that requires its services. The alternative is to wait to load the DLL until an application makes a call into it. PRELOAD gets better run-time performance. ***MOVEABLE*** means that Windows can move the code around in memory when it needs to compact things due to increased demands for memory. This becomes an issue if you try to catch signals in the DLL. A signal handler registers an address for a signal handler, so, if your DLL is moved, the address the signal handler thought was your DLL will most likely be something entirely different. This causes problems. The simple thing to do if you need to handle signals is to declare the code segment FIXED instead of MOVEABLE. In this case, your CODE should not be marked DISCARDABLE.

The DATA segment is marked as ***SINGLE***. A DLL can have no more than one data segment. Some memory models (large, for instance) allow multiple code and data segments by default. You need to prevent this by including SINGLE in your definition file.

EXPORTS lists those functions that applications will need to get to. Your DLL can contain functions that are not exported, but those will be accessible only to other functions within your DLL. Notice that the two functions we said DLL's required are listed in the ".def" file. Also notice that each of the functions VEE will call have an "_" prepended to their names. The C compiler prepends this to function names when you use CDECL calling conventions, but adds no preceding "_" to functions declared "pascal". Your 'EXPORTS' section needs to add the leading "_" to functions declared as CDECL.

One other thing to note about the 'EXPORTS' section is that the function 'WEP' is marked as '***RESIDENTNAME***'. 'RESIDENTNAME' tells Windows to keep the WEP routine's name resident in memory so that the WEP can always be called — even when there is little available memory. Under conditions of high memory use, Windows has the option of discarding memory segments. If it were to discard your 'WEP' routine, the condition could arise that Windows wouldn't be able to unload your DLL

because there isn't enough memory to page in the tables that contain the information describing the location of your 'WEP' routine. This can cause valuable system resources to be effectively unavailable. To prevent this, mark your 'WEP' routine as 'RESIDENTNAME' so that Windows can always find it and unload your DLL when it needs to.

DLL Header File

```
#if defined(MSDOS)

# include <windows.h>
# include <conio.h>

int FAR PASCAL LibMain( HANDLE, WORD, WORD, LPSTR);
int FAR PASCAL WEP(int);

long FAR CDECL writeByte(long portAddr, long dataByte);
long FAR CDECL writeWord(long portAddr, long dataWord);
long FAR CDECL readByte(long portAddr);
long FAR CDECL readWord(long portAddr);

#endif
```

Make File

```
all: example.dll

CCDLLOPTS=-nologo -ALw -GDs -Od -Zpi -W3

CC=cl -c

LINK=c:\msvc\bin\link /nologo

example.obj: examplex.h example.h example.c example.mk
  $(CC) $(CCDLLOPTS) example.c

example.dll: example.obj
     $(LINK) /co /map /NOE /NOD example /align:16,example.dll,,libw ldllcew oldnames,
example.def
  rc example.dll

clean:
  rm -f example.obj example.dll
```

Note that this make file is intended for use with Microsoft C/C++ 7.0 or 8.0. To build the DLL, type nmake -f makefilename.

A More Complete Example

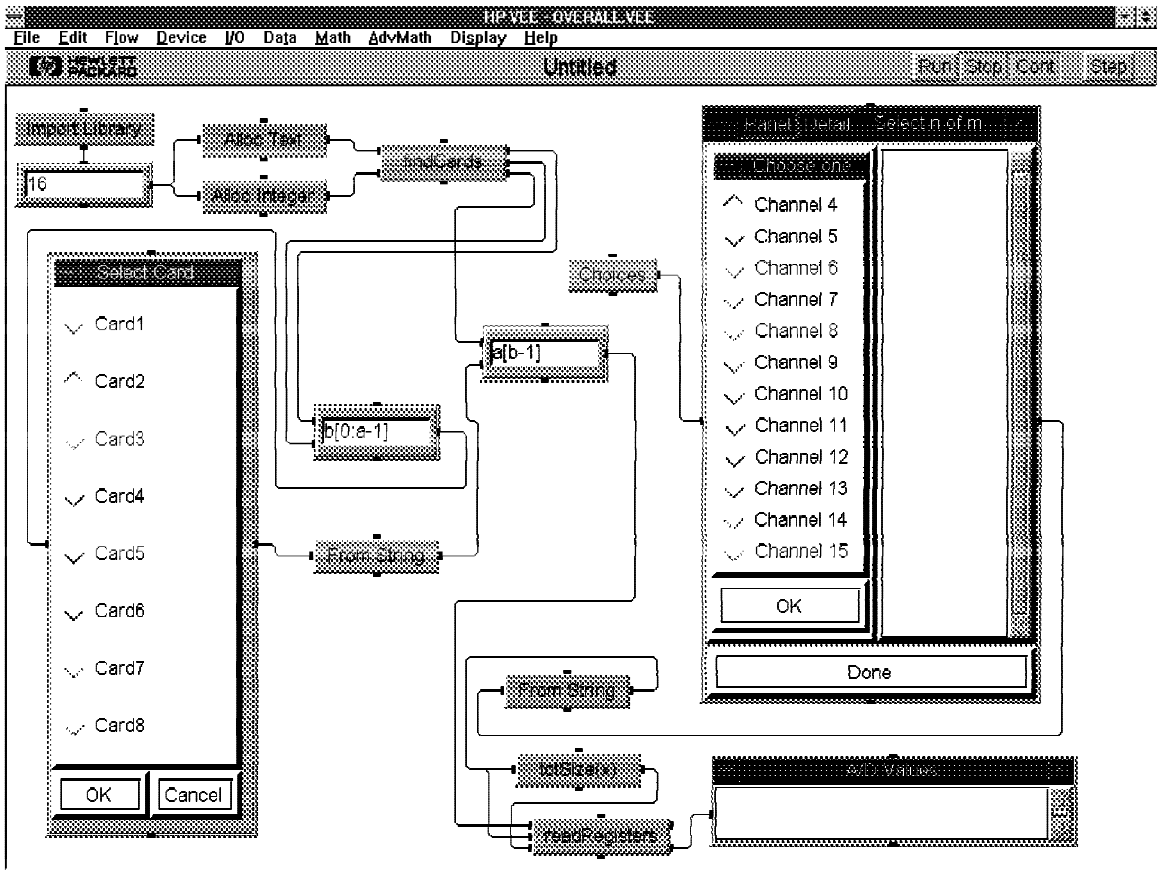
In the preceding example, you saw how to pass a single value to a Compiled Function and how to return a single value from that function. In fact, every "Call Function" object requires a return pin, and the returned value must be one of the VEE scalar types: a double, a long, a void, or a character string. There may be a

time when you want to take more than just a single reading from an A/D card, which would require that VEE Compiled Functions provide you with the capability to propagate an array of values to the rest of your program. This next section describes how to do just that.

Let's construct ourselves a typical application scenario. In actuality, this is a fictitious example, because the card I used to test the functions presented in this paper doesn't look anything like an A/D except that it has registers. We have one or more A/D cards that each have 16 input channels. Each channel can take a 16-bit reading and has its own I/O port. So, channel zero is assigned to the base address of the A/D card, channel one is at an offset of two bytes from the base address, and so on. Further, the 16 bits are an encoding where:

- The most significant bit represents the sign of the measurement: set means that the value is negative, clear means that the value is positive.
- The next three most significant bits represent a measurement range.
- The 12 least significant bits are raw A/D values.

We would like for our user (the consumer of the VEE program and associated Compiled Functions) to be able to choose to acquire data from some number of the available 16 channels in any order on any of the A/D cards configured on a particular system. To do that we must provide a way for the user to select up to 16 values which will be passed to the Compiled Function that takes the readings. We must then provide a scaling utility (a VEE User Object or User Function) which converts the raw A/D return values into a meaningful representation. We must also provide an easy way for a user to let our VEE program know the base addresses for each of the configured A/D cards.



Passing an Array to a Compiled Function

In the preceding section, we saw that VEE uses a definition file to tell the "Call Function" objects in your program how they should configure their data input terminals. The file looked something like this:

- long functionName(long portNumber)

If we look at the argument to the function, we can tell that "portNumber" is "pass-by-value" parameter. (We'll show you how to identify an argument shortly.) In other words, VEE will take the variable's value and place it on the call stack. When "functionName" runs, it removes the value from the call stack and uses it in the appropriate computations. So, a "called" function that receives pass-by-value parameters gets its own local copies of those values. You can see that you don't want to pass large arrays this way, due to the substantial data copying (and subsequent loss of performance) that would result.

You can also see that, because a called function gets a copy of the variable's value, that called function cannot alter the value of the variable in the calling context. The "calling" program will retain the original value of the pass-by-value even variable after it has called the function.

When you pass an array to a Compiled Function, what you really pass is the address of the first element in the array. Passing data to a function by specifying the location of the associated data is known as "passing by reference". When you pass an array this way, the only value pushed onto the call stack is the data's starting address. This increases performance by copying a single value (the data's address) as opposed to copying the entire data structure.

Another effect of passing by reference is that the called function can now alter the value of the data structure in the calling context. Because the called function now knows where the data is located, any changes it makes to the data will be reflected in the calling VEE program. This is how we will pass an array of A/D values back to VEE.

Definition File Syntax

To pass a variable by reference into a Compiled Function, you precede the variable name with "*". This is the standard C symbol for a pointer. So, a VEE Compiled Function definition file where you want to change the value of an incoming array would look something like this:

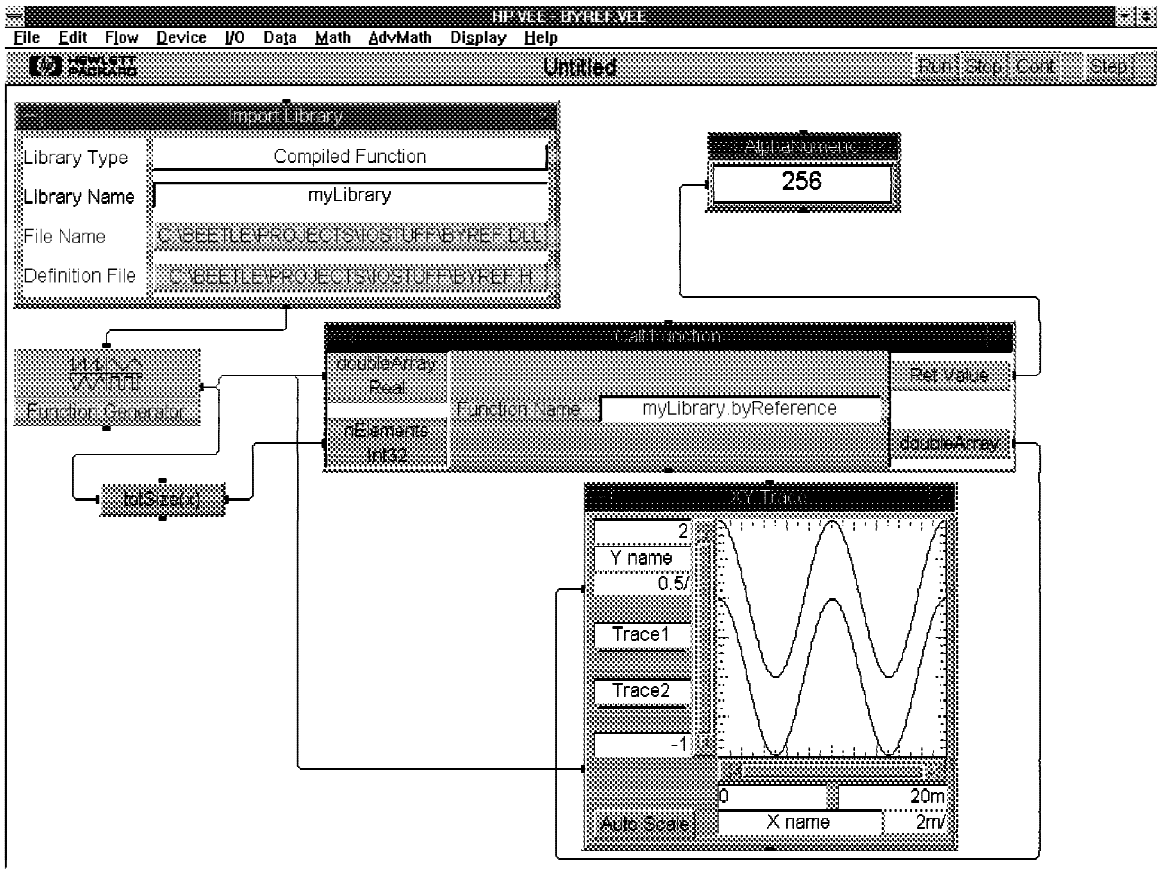
- long functionName(long *channelList, long nElements)

Notice that we have included a separate parameter called "nElements". We include this to let our Compiled Function know how many "long" values are in the array VEE has passed us a pointer to. This is necessary to prevent the Compiled Function from writing beyond the end of the incoming array. We know where the array starts, but we don't know (without the VEE program telling us) how big the array is. Writing beyond the end of an array will, at best, cause us to over-write what would have been useful information and could cause a data access violation. A data access violation shows up as the dreaded Windows General Protection Fault.

A Small Pass-by Reference Example

So, let's start by having VEE pass an array of numbers to a Compiled Function and add one to each of those values. The following are the applicable files.

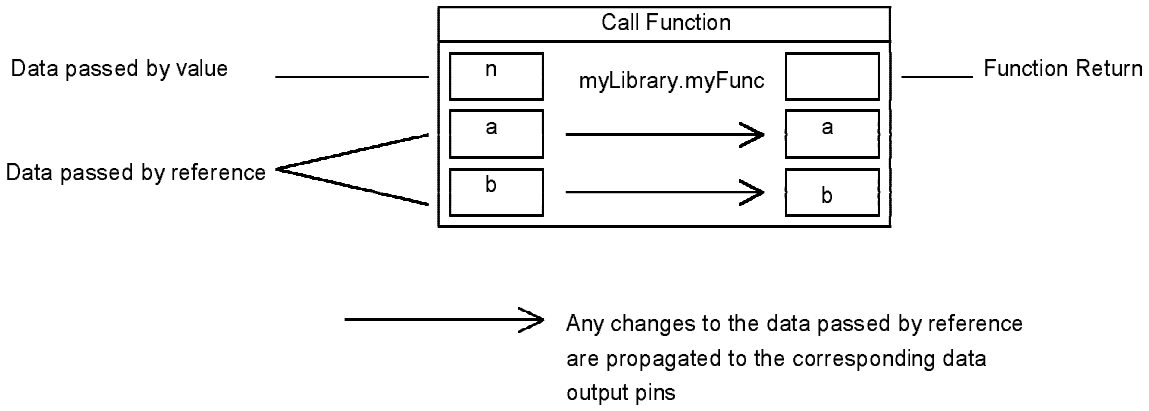
The VEE Program



There are a few things worth mentioning as we examine the above VEE program. Notice that the Call Function object has two data output pins. The top-most pin is the function's return value; we saw that in the previous section.

The other data output pin carries the data that we changed in the Compiled Function. We passed in the address of an array of floating-point values and added one to every element of that array. That changed data then appears on the second output pin.

Notice that there is no data output pin associated with the data item we passed by value. The Call Function object will create one data-out pin for each value you pass by reference to the Compiled Function. The data-out pins are related to their associated data input pins by location even though they, by default, carry the same name as the input pins. So, as long as you know how much data you want to acquire from an A/D card, you can pass in array big enough to hold the readings, then fill up the array with the acquired data.



The last item of interest is the use of the "totsize" object, which tells the Compiled Function the number of data elements an incoming array contains. By using this object to feed the Compiled Function, you can construct an application that dynamically scales itself to the size of data that the user wishes to observe. The alternative is to create an array large enough to hold the biggest number of data items a user could ever be reasonably expected to request.

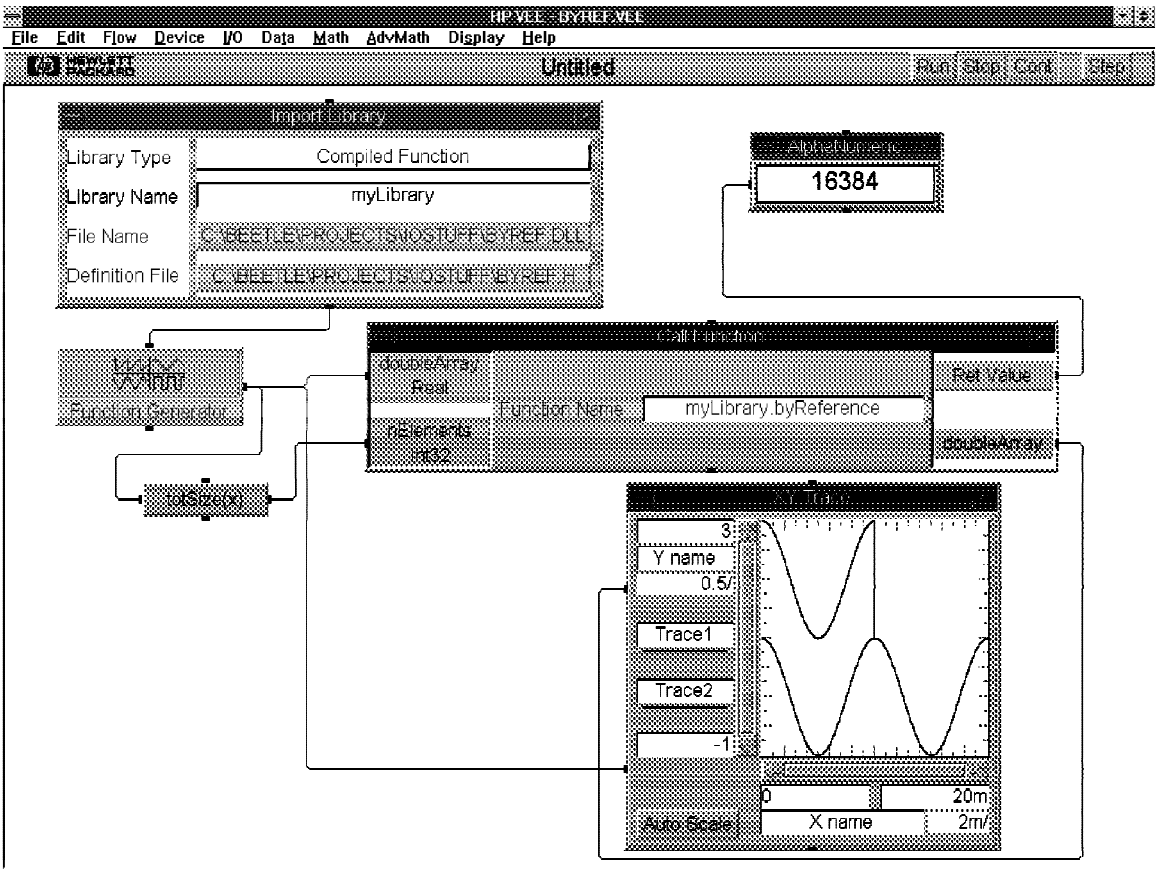
Compiled Function Code

```
#include "byrefx.h"

int FAR PASCAL LibMain( HANDLE  hModule, WORD wDataSeg,
   WORD cbHeapSize, LPSTR lpszCmdLine)
{
   return 1;
}

int FAR PASCAL WEP (int nParm)
{
    return 1;
}


long FAR CDECL byReference(double FAR *doubleArray, long nElements){
          long i;

          for(i=0; i<nElements; i++, doubleArray++){
                    *doubleArray += 1.0;
          }

          return(nElements);
}
```

The item of interest here is the use of the key word "FAR" preceding the declaration of the pass-by-reference data. Windows uses a 16-bit segmented memory architecture. Each 16 bits can address one 64-Kbyte segment. A Windows application can have more than one code segment and/or data segment. When pointers to data or code are addresses that reside in the same segment as the entity wishing to use the pointer, that pointer is termed "near". It takes only the 16-bit offset within the 64-Kbyte segment to uniquely identify the data or code.

If you wish to access data that resides in a different segment, you need to use a "far" pointer. A "far" pointer identifies the data (or code) segment the given value resides in and also indicates the offset within the given 64-Kbyte segment where the item of interest is located. When you pass data by reference into a Compiled Function, that data will always be in a different segment than the code which makes up the DLL. Because of this, all of your pass-by-reference data must be passed through a "far" pointer.

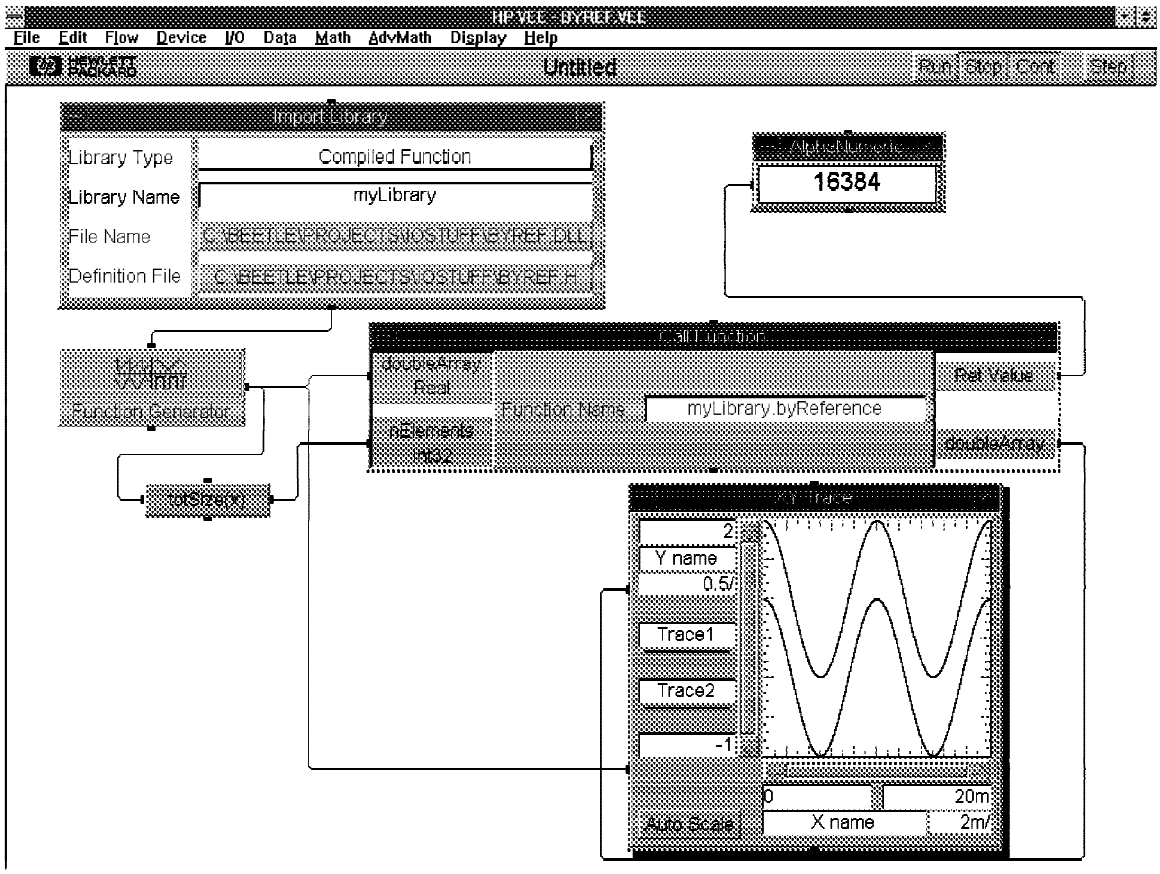
Using Data That Requires Greater Than 64 Kbytes



Notice what happened when we made the number of points comprising the wave form greater than 8192. The wave form data consists of eight-byte, floating-point values. 8192 eight-byte values occupy exactly 64 Kbytes of memory. Once we increase the number of wave form points beyond 8192, the data will no longer fit in a 64-Kbyte data segment, which is all that a "far" pointer can address. Remember that a "far" pointer identifies a 64-Kbyte segment and an offset within that segment.

To access data that occupies greater than 64 Kbytes, you must declare the pointer to the array as a "huge" pointer. A "huge" pointer is a flat 32-bit address, so it is not limited to the 64-Kbyte limit. Your Compiled Function source code should look like this:

```
#include "byrefx.h"

int FAR PASCAL LibMain( HANDLE  hModule, WORD wDataSeg,
   WORD cbHeapSize, LPSTR lpszCmdLine)
{
   return 1;
}

int FAR PASCAL WEP (int nParm)
{
    return 1;
}


long FAR CDECL byReference(double _huge *doubleArray, long nElements){
          long i;

          for(i=0; i<nElements; i++, doubleArray++){
                    *doubleArray += 1.0;
          }

          return(nElements);
}
```



The VEE Definition File

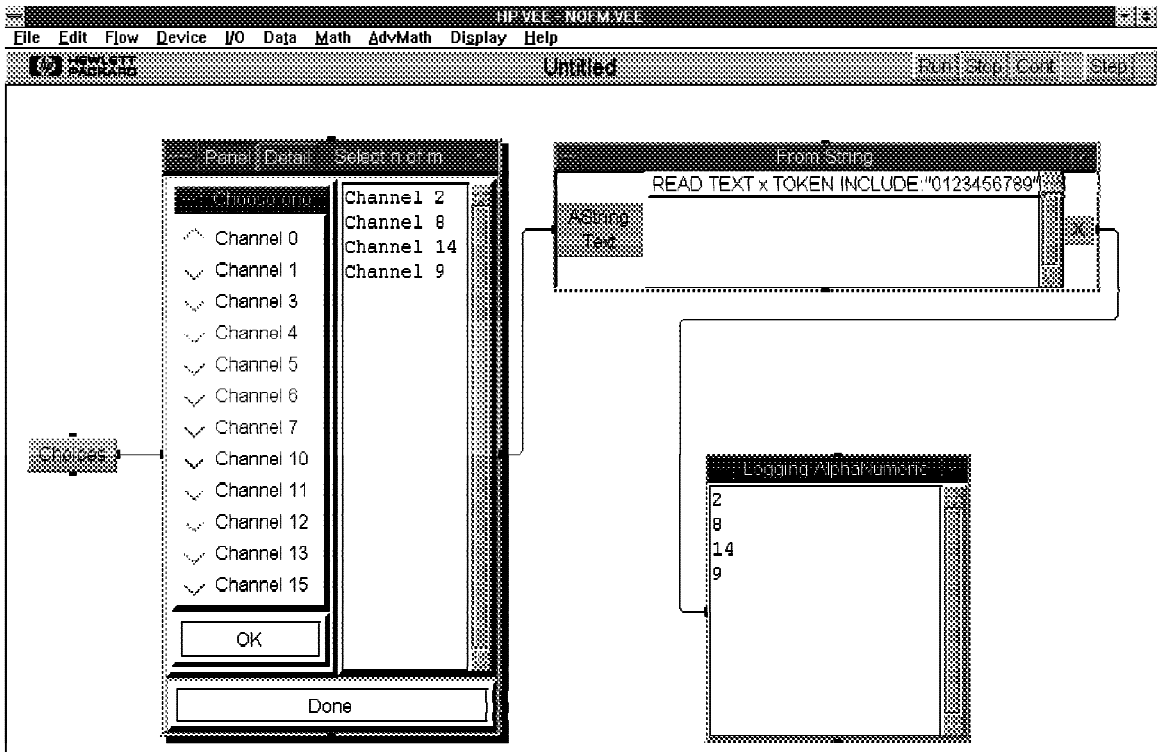
```
long byReference(double *doubleArray, long nElements)
```

Again, this is the file VEE will read to find out what functions you want to be able to call and what data types those functions will require. Notice that it is not necessary (and will cause a VEE error) to include any data access specifiers (the "FAR" keyword, for example).

Selecting Up to 16 Channels for Digitizing

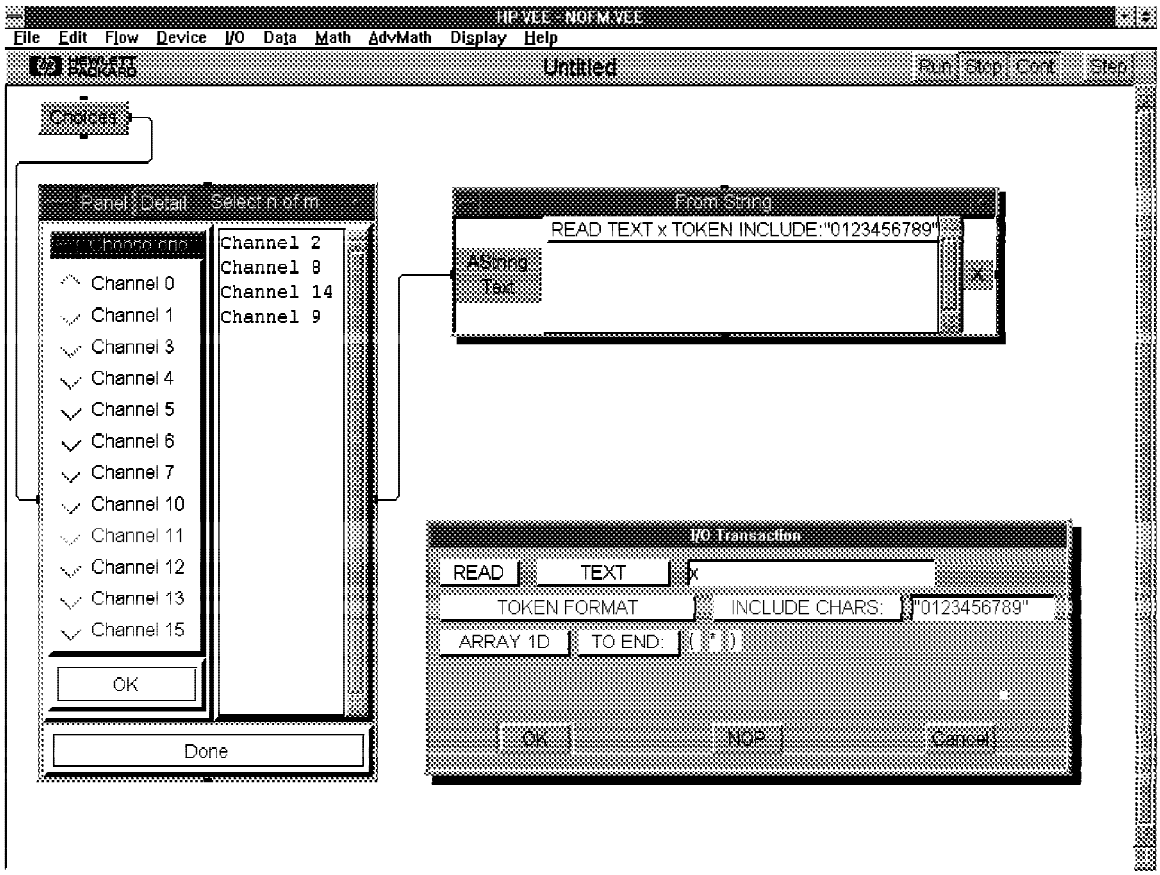
We need three pieces to construct this application. We need to allow the user some way to select from 1 to 16 digitizer channels, in any order, from a list that we will define. We must then develop a Compiled Function that will digest the list of channels our user chose and return the appropriate values. We then need to take the raw values from the A/D card and turn them into voltage values.

An "N of M" Selection Object



The User Object labeled "Select n of m" allows a user to click on one of the selections in the enumerated list labeled "Choose one". When users want to add the highlighted channel to the list of desired selections, they can click the "OK" button. When they are done making selections, they can click "Done". The selection User Object will propagate a list of the selected channels when users have selected all of the channels from the list we have provided or when they have explicitly selected "Done".

Notice that the "From String" object gives us a list of only the numeric values of selected channels. This is the information we want to pass to our Compiled Function so that we don't need to parse an array of strings to get the channel numbers. The transaction that accomplishes this is shown below. Basically, what it says is "look at all the incoming strings and give us back an array of strings containing only numeric values." The selection labeled "ARRAY 1D: TO END" instructs the transaction to look at any number of values we want to pass in. In this way, we allow our users to choose as many or few channels as they like, because we don't need to know in advance how many channels they have chosen.



The ".ini" File

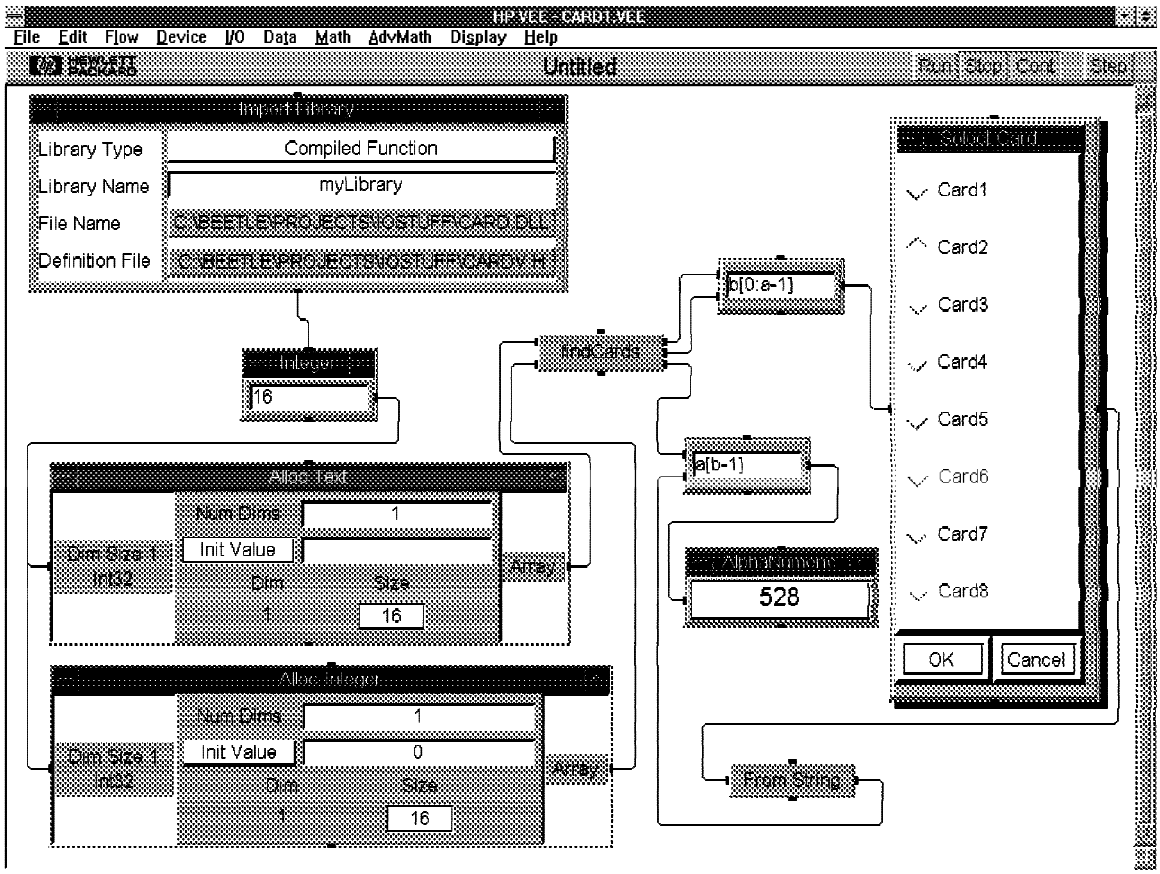
```
[cardAddresses]
Card1=0x370
Card2=0x210
Card3=0x230
Card4=0x250
Card5=0x270
Card6=0x290
Card7=0x310
Card8=0x330
```

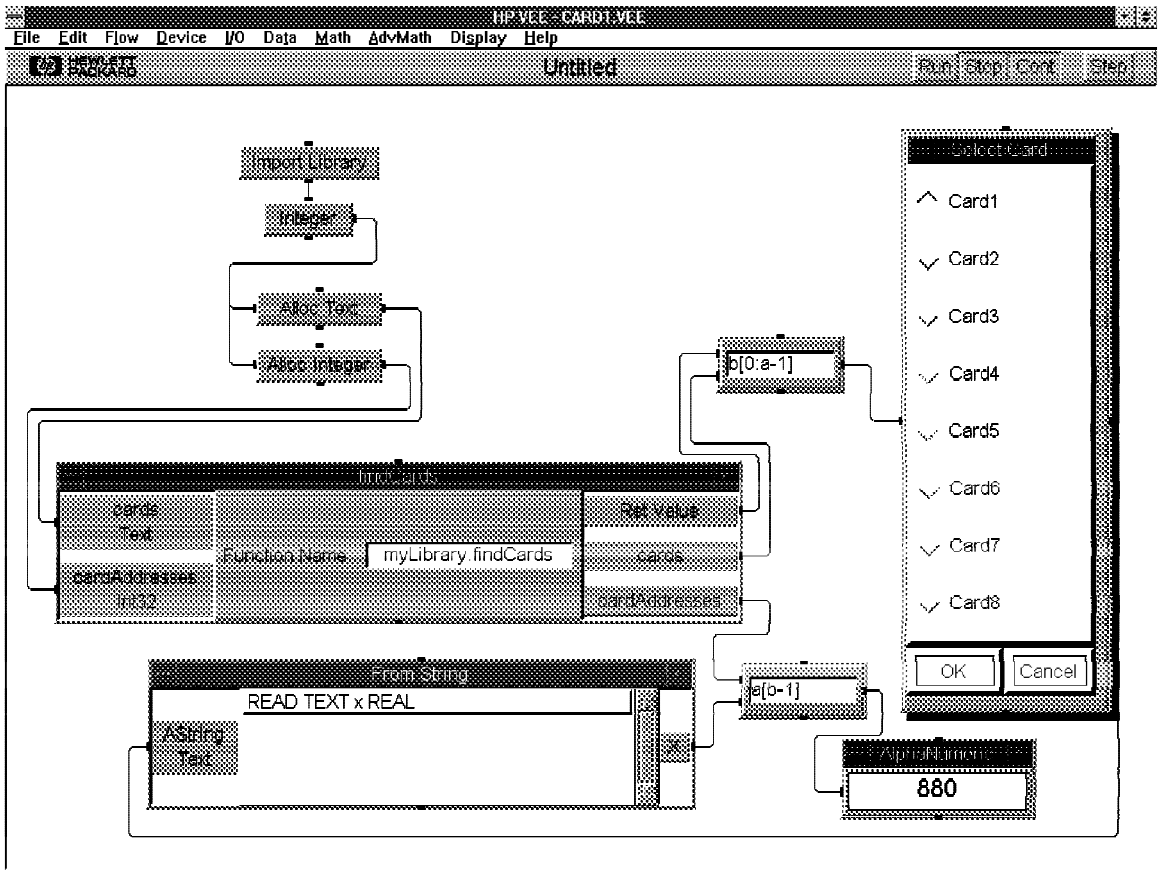
This file contains the base addresses for each of the cards in a system. It is the usual practice (if your card comes with an installation utility) for installation software to build this file. It may also be built by hand. This file has been deposited in the Windows installation directory (to make it easy to find), and, although not strictly necessary, we have named it with a ".ini" extension. In our case, it's "example.ini".

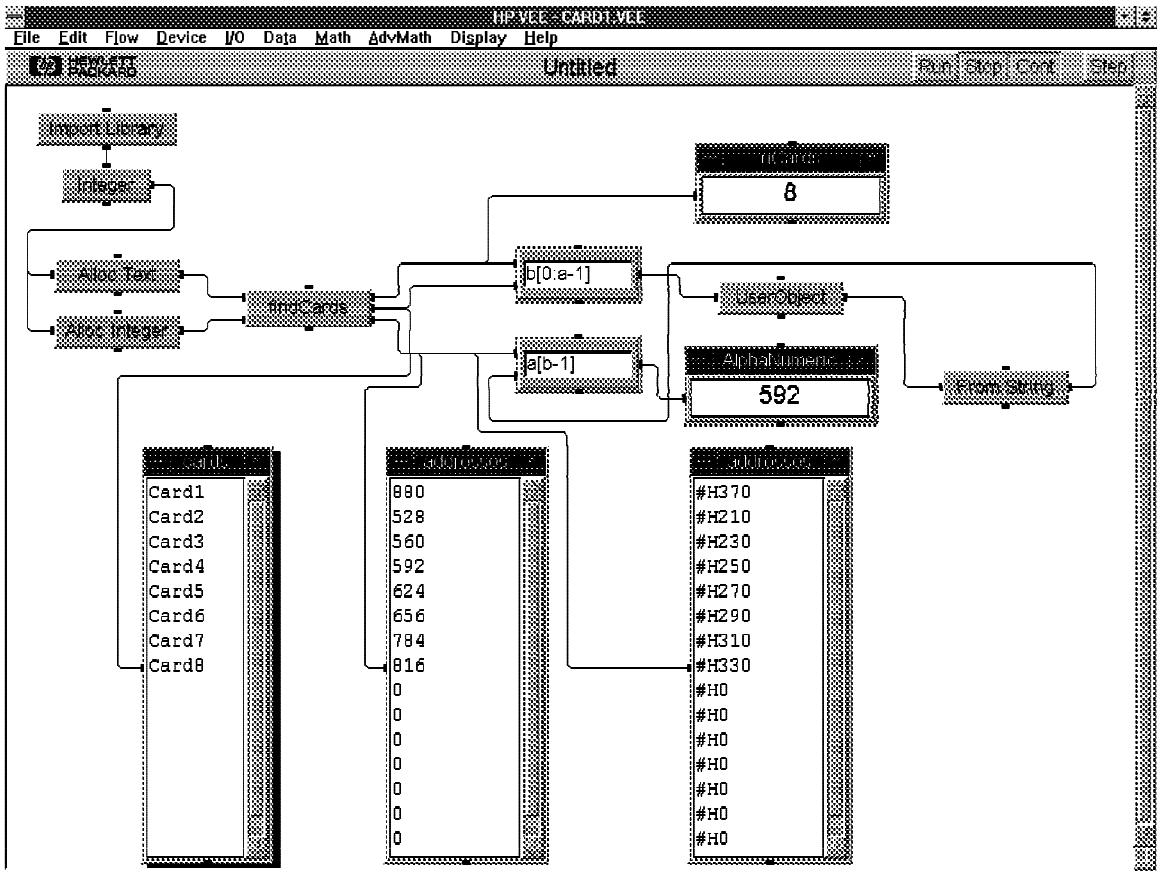
A Windows function knows how to identify different areas in an initialization file based on the bracket-delimited section headings. In our case, Windows will be able to identify a section named "cardAddresses", which contains an unlimited number of related entries. Notice that for convenience sake, we have entered the card addresses using hexadecimal notation. A Windows function knows how to translate these values into integers our Compiled Function can use.

The Compiled Functions

We'd like to show a couple things with these functions. The first is the use of the user-provided channel list to allow us to index into the correct I/O port address to get the requested value. The second is the use of a Windows function. You're probably familiar with editing an application's ".ini" (usually pronounced "dot inney") file to set default conditions. We'll use a ".ini" file to allow our users to set default base addresses for their A/D cards. The cards may exist at different addresses on different workstations, and we don't want to force our user to memorize these locations and type them into a VEE program as they move between machines.




HP VEE - CARDT.VEE
File Edit Flow Device I/O Data Math AdvMath Display Help
Untitled
Run Stop Cont Step
Import Library
Integer
Alloc Text
Alloc Integer
b[0:a-1]
Select Card
Card1
Card2
Card3
Card4
Card5
Card6
Card7
Card8
findCards
cards Text
cardAddresses Int32
Function Name: myLibrary.findCards
Ref value
cards
cardAddresses
From String
READ TEXT x REAL
AString Text
x
a[0-1]
OK
Cancel
AlphaNumeric
880



Identifying the Cards

In the preceding three screen dumps, we see a Compiled Function called "findCards" that reads our initialization file to get the base addresses we have recorded for each of the installed A/D's. There are a few things worth mentioning as we look at the screen dumps.

We have used two "Allocate Array" objects to give ourselves data space we can fill in with the desired information. The initial values for each element in the resulting arrays is set to whatever the value is in the "Initial Value" fields of the Allocate Array objects. In this case, we will have an array of 16 (an arbitrarily chosen maximum number of cards in a system) character strings which contain 32 spaces and an array of 16 longs whose value is 0. When the Compiled Function executes, its return value indicates the number of entries it finds in the initialization file. We use that return value to look only at valid information which the Compiled Function has placed into the two array spaces we created. The final screen shot shows that the Compiled Function overwrites the original arrays with pertinent information.

The VEE Definition File

```
long findCards(char **cards, long *cardAddresses)
```

Notice that we have preceded the first parameter, "cards", by two "*" characters. This tells VEE that we are passing an array of character strings to our Compiled Function. Remember that the C language defines a string as a zero-terminated array of characters. If each string is one array, then we must pass an array of character arrays to get a list of symbolic card names.

The Code

```
#include <windows.h>
#include <string.h>
#include "card.h"

int FAR PASCAL LibMain( HANDLE  hModule, WORD wDataSeg,
   WORD cbHeapSize, LPSTR lpszCmdLine)
{
   return 1;
}

int FAR PASCAL WEP (int nParm)
{
    return 1;
}

#define MAXCARDS    16
#define CARDNAMELEN 32

long FAR CDECL findCards(char FAR **cards, long FAR *cardAddresses){
   long nCards=0;
   char cardName[CARDNAMELEN];
   long i;

   for(i=1; i<=MAXCARDS; i++){
      _fmemset((void FAR *)cardName, '\0', CARDNAMELEN);
      wsprintf((char FAR *)cardName, "Card%d", (int)i);

      *cardAddresses=GetPrivateProfileInt((LPCSTR)"cardAddresses",
         (LPCSTR)cardName, 0, (LPCSTR)"example.ini");

      if(0 != *cardAddresses){
         _fstrcpy(*cards, (char FAR *)cardName);
         cards++;
         cardAddresses++;
         nCards++;
      }
   }

   return(nCards);
}
```

Notice that we have used several of the Windows equivalents to the C standard I/O routines. They are ***wsprintf()***, ***_fstrcpy()***, and ***_fmemset()***. They are the equivalents to sprintf(), strcpy(), and memset(). We use these functions because the C standard I/O routines by default assume that the calling routine's data segment is the same as its stack segment. Because DLL's don't have a stack segment (they use the stack of the application they are attached to), a DLL's stack segment will never be the same as its data segment. The routines we use in the preceding code snippet don't make this assumption, rather, they take "far" pointers as their arguments. Doing so makes them safe for use in DLL's.

Notice also that we have used the macro ***LPCSTR*** in our DLL code. This is defined in <windows.h> and is read as a "**l**ong **p**ointer to a **C** **str**ing". A "long pointer" is another way of saying that the value is a "far" pointer. We bring this up because the Windows prototypes use macros like those we used in the above example. Hopefully, this will make it easier to relate VEE data types to the macro definitions you will find in the Windows documentation.

We used the Windows function ***GetPrivateProfileInt()***. This is Windows' standard mechanism for allowing a program to read the configuration values a user prefers. Using the ".ini" file gives our application some capability to port between machines and provides the ability to expand or contract as our environment changes. This flexibility does not require us to change our application programs.

GetPrivateProfileInt() will read a file called "example.ini", looking for a section labeled "cardAddresses". When it finds that section, it will attempt to match the string "Card*", where "*" is a digit representing the card we are interested in. If we find a match for Card*, GetPrivateProfileInt() returns the value following the "=".

Reading the Registers

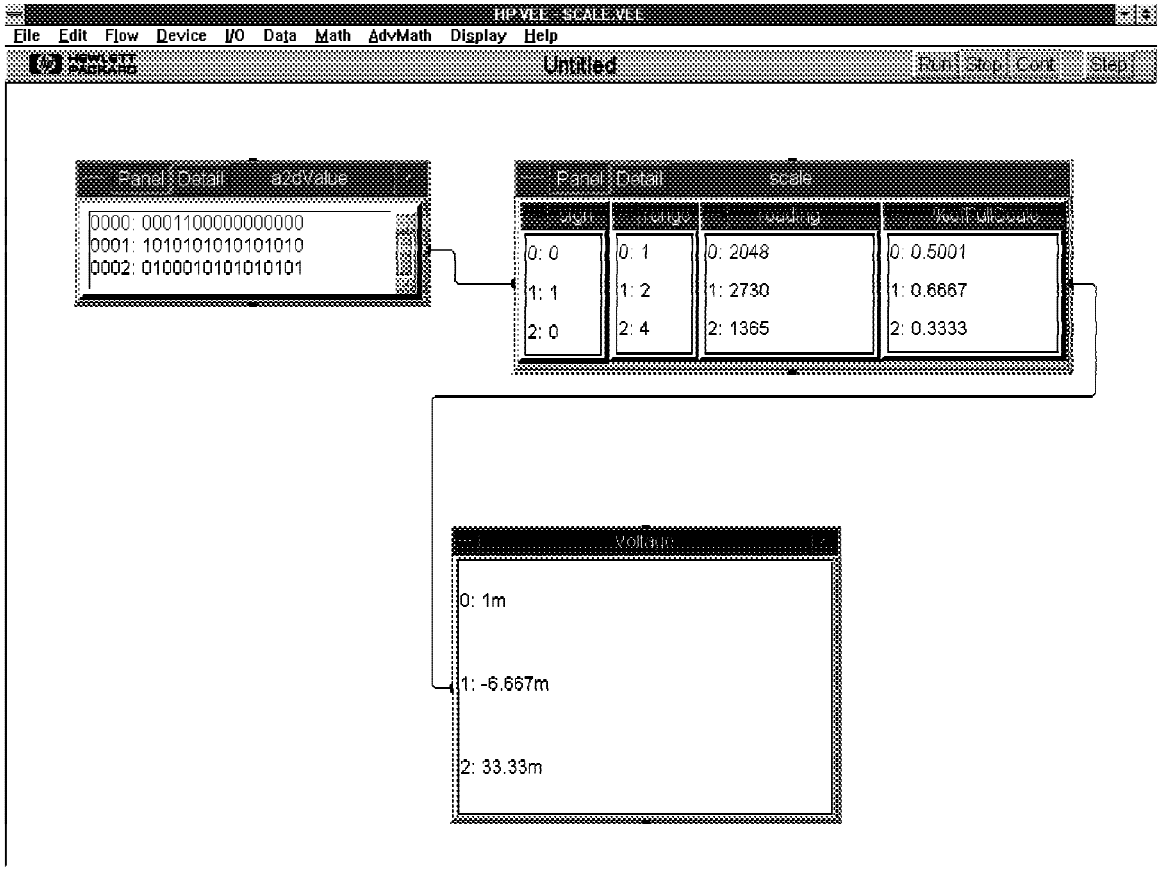
```
#include <windows.h>
#include <string.h>
#include <conio.h>
#include "card.h"

int FAR PASCAL LibMain( HANDLE  hModule, WORD wDataSeg,
   WORD cbHeapSize, LPSTR lpszCmdLine)
{
   return 1;
}

int FAR PASCAL WEP (int nParm)
{
    return 1;
}

#define MAXCARDS    16
#define CARDNAMELEN 32

long FAR CDECL findCards(char FAR **cards, long FAR *cardAddresses){
   long nCards=0;
   char cardName[CARDNAMELEN];
   long i;

   for(i=1; i<=MAXCARDS; i++){
      _fmemset((void FAR *)cardName, '\0', CARDNAMELEN);
      wsprintf((char FAR *)cardName, "Card%d", (int)i);

      *cardAddresses=GetPrivateProfileInt((LPCSTR)"cardAddresses",
         (LPCSTR)cardName, 0, (LPCSTR)"example.ini");

      if(0 != *cardAddresses){
         _fstrcpy(*cards, (char FAR *)cardName);
         cards++;
         cardAddresses++;
         nCards++;
```

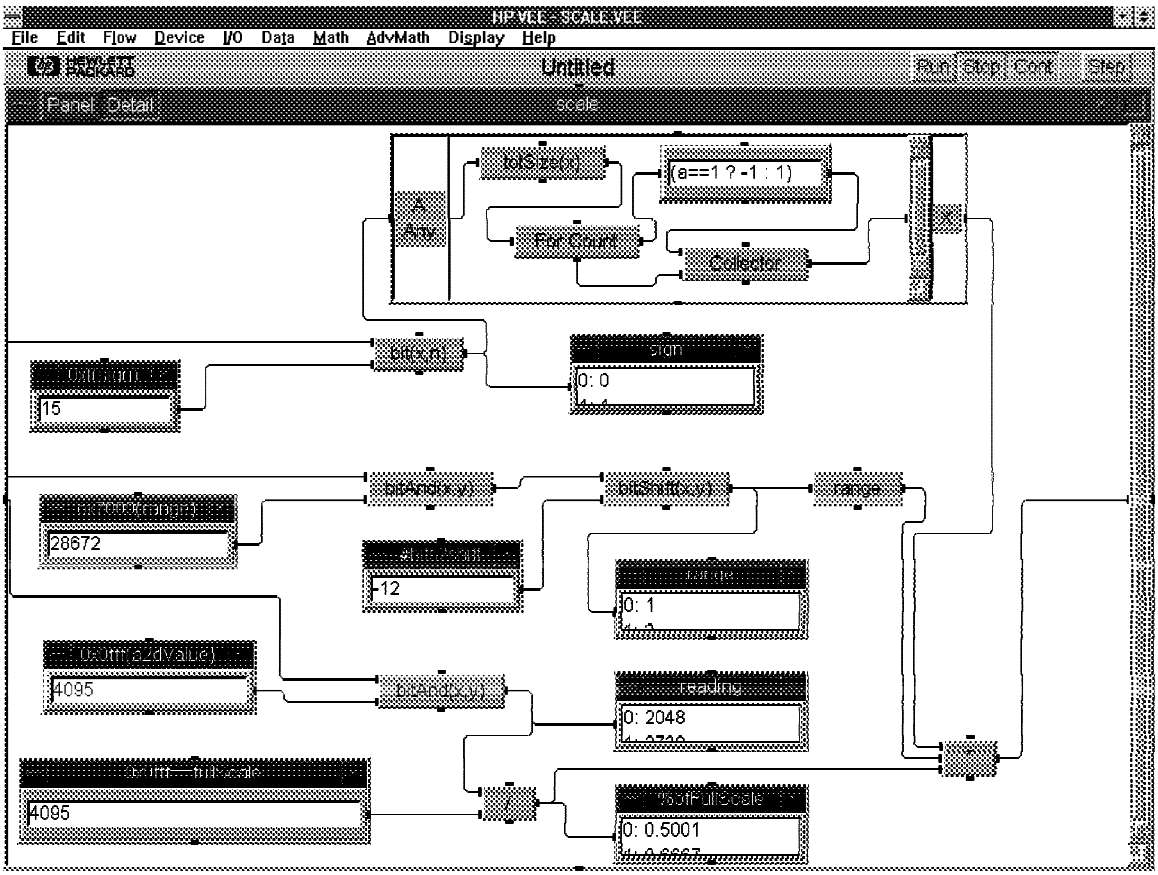
```
    }
  }

  return(nCards);
}

long FAR CDECL readRegisters(long baseAddress, long FAR *channelList, long nChannels){

        long i;
        unsigned portAddr;
        unsigned data;

        for(i=0; i<nChannels; i++, channelList++){
                portAddr = (unsigned)(baseAddress + (*channelList * 2));
                data = inpw(portAddr);
                *channelList = (long)data;
        }

        return(nChannels);
}
```

There are just a couple of items to point out in the function called "readRegisters". We have used the base address returned by the Compiled Function called "findCards" to let us know the address of the first A/D channel on the card. We determine the exact ports which correspond to the channel selections the user made by adding the channel number *2 to the card's base address. Each A/D channel is 16 bits, so we need to get the readings from 2-byte boundaries.

Notice also that we have returned the A/D readings through the channel list that the user passed in. The data is of the same type and contains the same number of elements, so let's use the data space that otherwise would have gone to waste.

Unpacking the Returned Values





The preceding two screen dumps show how to scale the returned values to an actual voltage (or whatever) value, given the definition of the A/D word contents. The scaling is performed quite easily using the entries found under VEE's ***Math⇒Bitwise*** menu.

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. *HP makes no warranties of any kind with regard to this document, whether express of implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.* HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Microsoft® and MS-DOS® are U.S. registered trademarks of Microsoft Corp.

Windows or MS Windows is a U.S. trademark of Microsoft Corp.

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

HP VEE Application Note

Issue No. 2

December 1994