Implementing a USB Equipment Interface Using the Microchip PIC16C745

COM ports are becoming scarce. Come learn how to interface projects the modern way with USB.

By Dick Lichtel, KD4JP

Since the advent of personal computers, Amateur Radio has striven to connect equipment to them. The primary interface has been the ubiquitous serial port. Intel developed the Universal Serial Bus[1] (USB) in the early 90s, and while many PC peripherals now support this interface, Amateur Radio experimenters and equipment manufacturers have not kept up with the technology. Owners of newer PCs have found the number of parallel and serial ports has dwindled down to one of each, while the number of available USB ports has substantially increased (there are six on my PC). It is likely that PCs in the future will contain no EIA-232 serial ports. The legacy-free system-requirements section of the PC2001 system design specification requires the removal of serial and parallel ports (among other legacy ports) from the PC hardware.[2] As part of the specification, USB ports are required on new machines. Each USB port can support up to 127 devices.

[1]Notes appear on page 8.

6545 Canal Rd.
Melbourne Village, FL 32904-3622
rlichtel@cfl.rr.com

Many believe that a strong knowledge of *Windows* programming, down to the device-driver level, is needed to support USB in their equipment. This article attempts to show that this is simply not true. *Windows* provides default drivers that we can use and that will take most of the work out of programming the *Windows* interface.

Microchip has several parts that include hardware support for USB 1.1. Microchip provides sample PIC code that implements the USB 1.1 protocol.[3] In this article, I will explain how to utilize the Microchip PIC16C745 microcontroller and discuss methods for communicating with this device. I have also included functional source code for the PIC firmware as well as PC based code to communicate with the PIC.[4]

Microchip PIC16C7X5 Controllers

In 2000, Microchip introduced the PIC16C745[5] and PIC16C765 8-bit microcontrollers with USB. These controllers are similar in architecture and instruction set to other PIC16C and PIC16F series microcontrollers, which are used in many Amateur Radio projects. This compatibility should make it easier to adapt many existing projects over to this PIC.

The PIC16C745/765 microcontroller features 8 kB of program memory and 256 8-bit memory registers, 11 interrupt sources, 22 ports (33 for the 765), three timers, five 8-bit A/D (eight for the 765) as well a USART and USB interface. The PIC16C765 also offers a parallel slave port.

Microchip supplies the development tools needed to develop applications for their microcontrollers for free.[6] The tools include a macro assembler, linker, librarian and *MPLab* Integrated Development Environment (IDE). The IDE also includes a PIC simulator and debugger. Microchip sells a *C* language compiler. There are third-party *C* compilers such as Hi-Tech (**www.htsoft.com**) that are supported by the *MPLab* IDE.

Microchip also sells PIC programmers; however, they are much more expensive than third-party programmers, such as the Newfound Electronics WARP-13a programmer (**www.newfoundelectronics.com**). The WARP-13a programmer is also compatible with the Microchip PICStart Plus! Programmer, so microcontrollers can be programmed inside of the *MPLab* IDE.

Microchip has written example source code for the PIC16C7X5, which provides the protocol for USB 1.1 in both assembly and *C*.[7] The example source code is not immediately useful since the routine necessary for bi-directional communications with the device is commented out. The documentation supplied with the source code describes how the firmware implementation works and documents the software methods.

In conjunction with this article, I have supplied functioning firmware, source code and *MPLab* project files that support bi-directional communications with the PIC16C745. Because the assembler is free and the *C* compiler is not, the source code for the firmware is in assembly.

Communicating with the PC

Once your PIC is programmed with the supplied firmware, the PIC will simply await reception of eight bytes of data and echo them back to the PC. The number of bytes sent and received is a function of a pair of variables set in the firmware and is of fixed size. One of the files included with the firmware is a linker definition file that has been adapted for this project; the default linker file provided by Microchip with *MPLab* will not work with this firmware.

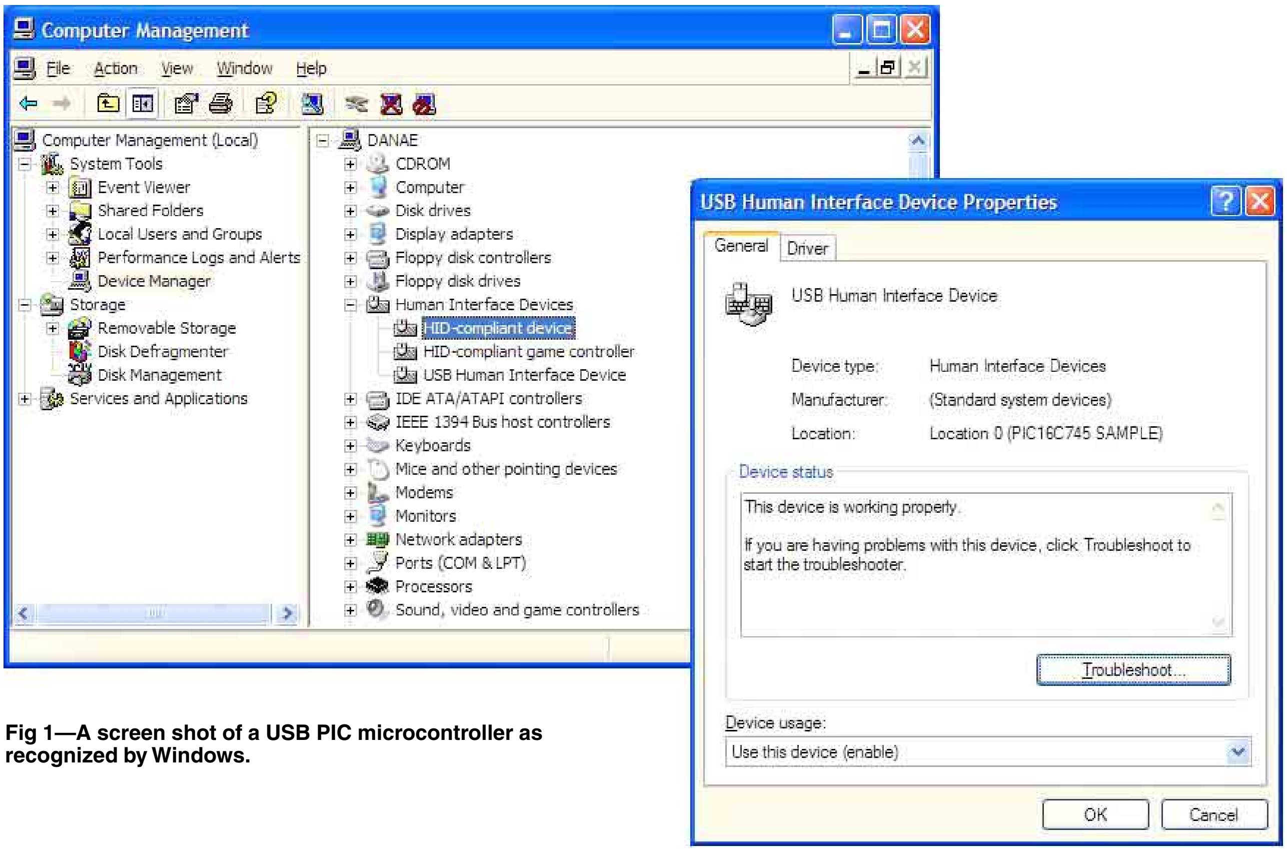
To adapt this assembly code to work with your specific project, you may need to make a couple of changes in *descript.asm* that depend upon your implementation. Change only these fields!

First, you must decide whether the PIC is to be powered from an external supply or from the USB connection. If powered by the USB bus, you must know how much current it will draw. If the PIC circuitry will be self-powered, set the hex values to 0. If it is to be powered from the USB bus, leave its value at 0x80 and change the subsequent hex value of 0x0D (26 mA) to the maximum current your circuit will draw.

$$Value = \frac{MaxCurrent}{2} \qquad \text{(Eq 1)}$$

Current is expressed in milliamperes, with a maximum draw of 100 mA. These variables are found in the routine "Config1" as shown in Code D.

Second, you should decide how



USB Human Interface Device

Device type: Human Interface Devices
Manufacturer: (Standard system devices)
Location: Location 0 (PIC16C745 SAMPLE)

Device status
This device is working properly.
If you are having problems with this device, click Troubleshoot to start the troubleshooter.

Device usage: Use this device (enable)

Fig 1—A screen shot of a USB PIC microcontroller as recognized by Windows.

Code A

```
pagesel         InitUSB               ; These lines of code show the appropriate
call            InitUSB               ;  way to initialize the USB. First, initialize

ConfiguredUSB   ; wait until the enumeration process is
                                        complete.

CheckEP1                              ; Check Endpoint 1 for an OUT transaction
    bankisel    Buffer                ; point to lower banks
    pagesel     GetEP1
    banksel     Buffer
    movlw       Buffer                ; Data to be recv'd will be put in Buffer
    movwf       FSR                   ; Point FSR to our buffer
    call        GetEP1                ; if data is ready, it will be copied.
    Pagesel     CheckEP1
    btfss       STATUS,C              ; was there any data for us?
    goto        CheckEP1              ; Nope, check again.

PutBuffer
    bankisel    Buffer                ; point to lower banks
    pagesel     PutEP1
    movlw       Buffer
    movwf       FSR                   ; Point FSR to our buffer
    movlw       0x08                  ; send 8 bytes to the Host
    call        PutEP1
    pagesel     PutBuffer
    btfss       STATUS,C              ; was it successful?
    goto        PutBuffer             ; No: try again until successful
    pagesel     CheckEP1
    goto        CheckEP1              ; Yes: restart loop
```

Code B

```
char HIDDevicePath[]=
"\\?\hid#vid_04D8&pid_1234#7&1bf5e077&0&0000#{4d1e55b2-f16f-11cf-88cb-001111000030}";
FileHandle=CreateFile (HIDDevicePath,
        GENERIC_READ|GENERIC_WRITE,
        FILE_SHARE_READ|FILE_SHARE_WRITE,
        (LPSECURITY_ATTRIBUTES)NULL,
        OPEN_EXISTING,
        0,
        NULL);
```

Code C

```
// default data size is 9 (8 data bytes + 1 for the report #)
CHIDDevice HIDDevice;
//The ID, product ID & version# are from the PIC firmware
DWORD dwVendorID(0x04D8),dwProductID(0x1234),dwVersionNumber(0x0079);
if(HIDDevice.Connect(&dwVendorID,&dwProductID,&dwVersionNumber))
{
        byte abytes[9];
        memset(abytes,0,sizeof(bytes));
        memcpy(abytes+1,"1234",4);      // Byte[0] must always be 0
        HIDDevice.Write(abytes);
        HIDDevice.Read(abytes);
        // HIDDevice automatically closes the device when it goes out of scope
} //end if
```

large the data packet size needs to be. Data exchanged between the PC and the microcontroller is a fixed size. The supplied code assumes the data arrives and is sent in eight-byte blocks. If you change the block size, two instances of "report Count" need to be changed. These values are found in the routine "ReportDescriptor" in descript.asm as shown in Code F. Low-speed devices like this one are limited to a maximum of eight-byte blocks.

Next, change the hex values for the idVendor(0x4D8), idProduct(0x1234) and bdcDevice(0x0079) values in descript.asm as shown in Code E. These don't really need to be changed unless they conflict with other USB devices in your system or you plan on mass-producing the part. If you are an equipment manufacturer, you will need to purchase your own unique Vendor ID. More information on obtaining a Vendor ID can be found at **www.usb.org/developers/vendor/**. The default idVendor is 0x04D8 and is the Microchip Vendor ID. These values are found in the routine "DeviceDescriptor."

Finally, you will want to change the string descriptors, which designate the product name and version of your PIC (shown in Code G). You can also add support for different languages. The product name shows up in the *Windows* Device Manager when you display the properties for the device. The version string is useful if you want to poll the firmware for version number. These variables are found in the "String" routines at the end of the module (descript.asm).

The lines that can be changed are noted with identifying comments; search the file for the string "****Change."

The documentation supplied with the Microchip sample code details the USB methods. There are really only two routines of interest to the programmer: PutEP1 and GetEP1. These methods send and receive data from the PC host. Of secondary importance are InitUSB and ConfiguredUSB. These methods initialize the PIC and have it wait for the device to be connected to a host computer. Until the PIC is connected to a computer, the code will stay in a loop inside of ConfiguredUSB. Code A is a snippet of code from *main.asm* that shows how to read data from the PC and send it back.

I have added a little more code in my version of *main.asm*; if the PIC is sent "?V" (without quotes), the firmware will return the firmware version string.

While the PIC16C745 runs off an external 6 MHz clock, the PIC must be configured to run internally at 24 MHz for the device to function properly running the USB firmware. Do this by setting the PIC configuration bit E4_OSC if you are using a clock generator, or H4 if you are using a crystal oscillator. The USB specific code consumes about 1 kB of the 8 kB program memory and 40 bytes of memory in Bank 2.

Communicating with the Microcontroller

Microchip implemented the USB protocol so that the device appears as a Human Interface Device (HID). This is the same class of device as mouse, joystick or keyboard. *Windows* provides native support for HID devices.[8] If the firmware is implemented correctly, when the device is plugged into a PC *Windows* will recognize a new device and automatically install the HID driver (if its not already installed). If you open the *Windows* Device Manager you will see your device under the HID device(s). Fig 1 shows the microcontroller as listed in the *Windows* Device Manager. This microcontroller is also recognized by *LINUX* as an HID device. The methodology for communicating with this device under Windows should also apply to *LINUX*.

To communicate with the microcontroller, you need to know the path to the device. Once this path is known, the microcontroller can be opened as a file; you then can read/write to the device just like any other file.

There are two ways of finding the path to the microcontroller. The first method involves using the Microchip developed HIDComm.ocx[9] Active-X control. This control provides a method that browses the HID devices, displaying a dialog that allows the user to pick the device and returns the full path to the device. It also supplies methods to search for the device based upon vendor and product IDs (among other fields). These are the variables you may have changed when you created the firmware. The Active-X control is most useful to *BASIC* programmers. It can also be used with *C* and *C*++; but some of the other meth-

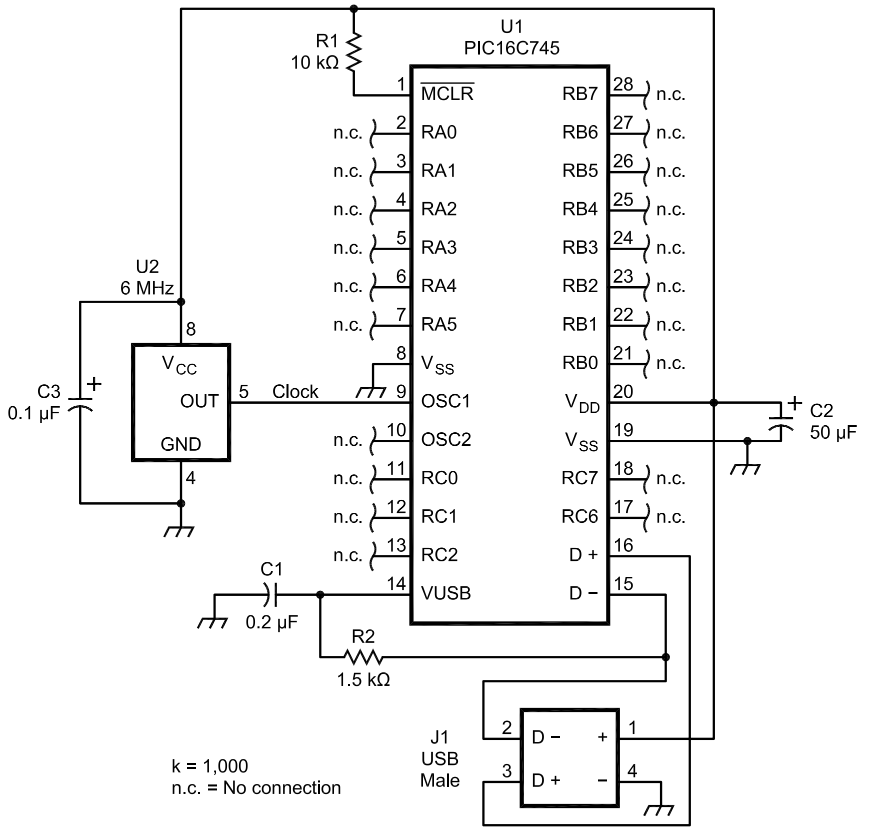


Fig 2—PIC16C745 schematic, USB powered.

U1—PIC16C745, Digi-key part #PIC16C745/JW-ND
J1—USB 4P Male Type BConnector, Digi-key part #AE1085-ND
U2—6 MHz Epson Electronics CMOS/TTL Oscillator, Digi-Key part #SE1206-ND

ods require data conversion to adjust for the way strings are implemented in *BASIC*, which is very tedious for *C*/*C*++ programmers.

The second method, which is probably more appealing to *C*/*C*++ programmers, utilizes a couple of routines from the *Windows* Driver Development Kit (DDK). It involves enumerating the HID devices, searching the list for your device, and then getting the path. The *Windows* DDK is free (nearly) from Microsoft (**www.microsoft.com/ddk/**). The DDK itself is free but Microsoft insists on charging $15 for priority shipping.

Once you know the path to device, a *Windows* API call can be used to connect to the device. Notice in snippet Code B, that the path is rather complex. Also note that the vendor and product IDs (from the firmware) show up in the path.

I have included a *C*++ class developed for communicating with the PIC microcontroller. Along with the *C*++ class, I have included the HID library(hid.lib) and header file(hid.h) which are needed to use the *C*++ class. Since the DDK library only supports *C*, I have adapted the hid.h header file so it can be used with both *C* and *C*++. Code C shows how to use my *C*++ class for communicating with the PIC.

While the firmware is configured to send and receive eight bytes, the USB protocol requires an additional byte for the report number. The firmware expects the report number as the first byte and for it to always be 0. So, the PC needs to send nine bytes, the first byte always being 0 and the remaining eight bytes being your data. The PC will receive nine bytes with the first byte being 0 and the remaining eight bytes containing the data.

Another important consideration when sending multiple byte numbers to the PIC is that the PC formats numbers in Little Endian format[10] and the equipment you have interfaced to the PIC may expect numbers in Big Endian format.

Fig 2 shows a schematic for a self-powered PIC16C745. Because the device operates as a slow-speed USB device, D– (pin 15) is tied to VUSB (pin 14) via R2. The power coming from the PC can be rather noisy, so a large capacitor is used across the supply. R1 is not strictly necessary and $\overline{\text{MCLR}}$ could be tied directly to VCC.

I wrote a benchmark program to measure the data rate for sending and receiving eight bytes of data (18 bytes total exchanged including the report-number byte) over 1000 cycles. The data rate was calculated a little more than 2 kB/s. Because the PIC is a low-speed USB device, this is the maximum data rate.

Code D

```
Config1 ...
   retlw 0x80 ; bmAttributes attributes - bus powered ****Change this to 0 if your device is self powered
   retlw 0x0D ; MaxPower 26 mA from the bus. ****Change this to match the current drawn by your circuit.
   ; Do not exceed 100mA otherwise Windows might not accept your device. The value is the max current/2.
```

Code E

```
StartDevDescr
   ....
   DT 0xD8,0x04 ; idVendor 0x04D8 ****Change this to uniquely ID your PIC. Note low order byte 1st
   DT 0x34,0x12 ; idProduct 0x1234 ****Change this to uniquely ID your PIC. Note low order byte 1st
   DT 0x79,0x00 ; bcdDevice 0x0079 ****Change this to uniquely ID your PIC (device version#) low byte 1st
```

Code F

```
ReportDescriptor
   ...
   DT 0x95,0x08 ; report count (8) (fields) .. ****Change 0x08 to the # of bytes you need, max of 8 ...
   DT 0x95,0x08 ; report Count (8) (fields).. ****Change 0x08 to the # of bytes you need, max of 8
```

Code G

```
String1_l1
        retlw String2_l1-String1_l1 ; length of string
        DT 0x03 ; string descriptor type 3
        DT'M',0x00,'i',0x00,'c',0x00,'r',0x00,'o',0x00,'c',0x00,'h',0x00,'i',0x00,'p',0x00

String2_l1 ;****Change this string to match your product name
        retlw String3_l1-String2_l1 ;
        DT 0x03
        DT 'P',0x00,'I',0x00,'C',0x00,'1',0x00,'6',0x00,'C',0x00,'7',0x00,'4',0x00,'5',0x00,' ',0x00
        DT 'S',0x00,'A',0x00,'M',0x00,'P',0x00,'L',0x00,'E',0x00
        global String3_l1

String3_l1 ;****Change this string to match your product version
        retlw String4_l1-String3_l1
        DT 0x03
        DT 'V',0x00,'1',0x00,'.',0x00,'0',0x00,'0',0x00
```

Summary

In summary, this paper has provided the reader with basic information for integrating a USB interface into Amateur Radio hardware. Our goal as Amateur Radio operators has always been to develop hardware and software that furthers the state of the art; but in this area, we have clearly fallen somewhat behind. I hope this paper has shown that USB can be easily adopted, and that programming for USB is not all that difficult. With the supplied PIC firmware and *Windows C*++ class, the task of developing the hardware and software to utilize the USB PC interface has been made even easier. Finally, I hope this article also inspires the major equipment manufacturers to stop relying on serial ports for their PC interfaces and begin to integrate USB technology into their products.

Notes

[1]A good overview of the Universal Serial Port Implementation can be found at **www.quatech.com/support/techoverview. php.**

[2]Intel and Microsoft Corporation, *PC 2001 V 10*, Chapter 3.

[3]The USB 1.1 and 2.0 specifications can be found at **www.usb.org/developers/docs/**.

[4]You can download this package from the ARRLWeb **www.arrl.org/qexfiles/**. Look for 0504Lichtel.zip.

[5]Microchip, PIC16C745/765 datasheet Rev DS41124C; **www.microchip.com/1010/pline/picmicro/category/perictrl/14kbytes/devices/16c745**.

[6]Microchip development tools can be obtained from: **www.microchip.com/1010/pline/tools/picmicro/devenv/mplabi/mplab6/index.htm**.

[7]Assembly and *C* based firmware can be obtained from: **www.microchip.com/1010/pline/picmicro/category/perictrl/14kbytes/devices/16c745/index.htm**.

[8]OSR2 is required for *Win95*. *NT* 4 users should install the latest service pack.

[9]HIDComm can be obtained from **www.microchip.com/1010/suppdoc/appnote/codxamp/9073/**.

[10]Dr. W. T. Verts, "An Essay on 'Endian' Order," **www.cs.umass.edu/~verts/cs32/endian.html**, April 1996.

Dick was first licensed in 1977 and in 1999 upgraded to Extra. He is an ARRL life member. Dick graduated from Siena College in 1979 with a BS in Physics. He has worked for Harris Semiconductor and Intel and is now somewhat retired. He has authored several IEEE papers and is an inventor on four patents on semiconductor processing. Dick wrote PakTerm *(later licensed by AEA and became* PcPackratt II) *and* PcPakratt for Windows. □□

REQUEST FOR INFORMATION ON GIGAHERTZ AND TERAHERTZ SPECTRUM

ARRL seeks information on spectrum requirements and preferred frequency bands in the spectrum above 275 GHz toward studies now underway by the United States Government and the International Telecommunication Union (ITU). The preliminary agenda for the 2010 World Radiocommunication Conference includes an item to consider frequency allocations between 275 GHz and 3000 GHz. Other studies are being conducted on bands well above 20 THz.

While no frequencies have been allocated above 275 GHz, there is growing interest, particularly for scientific and space applications. Amateurs can presently use any frequency above 300 GHz but this could eventually change if the ITU makes allocations to radio services. Amateur spectrum requirements and preferred frequency bands in the range 275-1000 GHz have been documented by the International Amateur Radio Union in **www.iaru.org/ac-spec02.html** and are being studied by the ITU along with similar information from other radio services. The preferred bands in the 275-1000 GHz range were chosen primarily because they are bands where atmospheric attenuation is low.

The need is for information on frequencies above 1000 GHz:

- What is the best scientific data available on the attenuation (or other propagation phenomena) in parts of this band? (Please see Figs 1 and 2, which pertain only to vertical paths.)
- What are the anticipated amateur and amateur-satellite uses of these bands?
- How much bandwidth is required for these uses?
- If sharing is necessary, which other radio services would be the preferred sharing partners and why?

Please provide information to ARRL Chief Technical Officer Paul Rinaldo, W4RI; e-mail **prinaldo@arrl.org**.

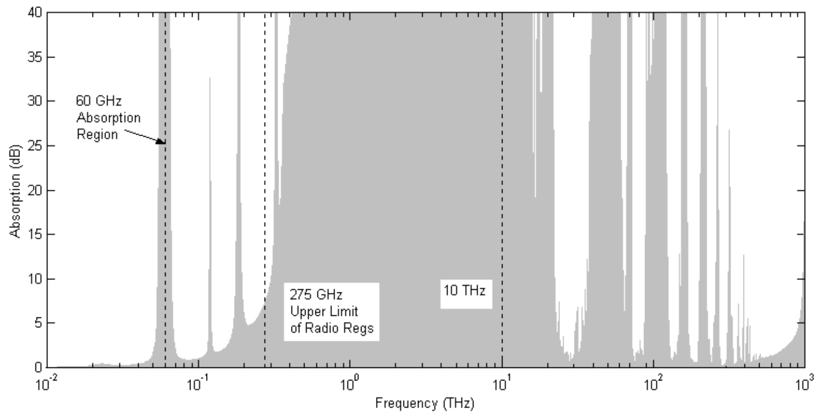


Fig 1—Absorption (shaded area) of a standard atmosphere along a vertical path.

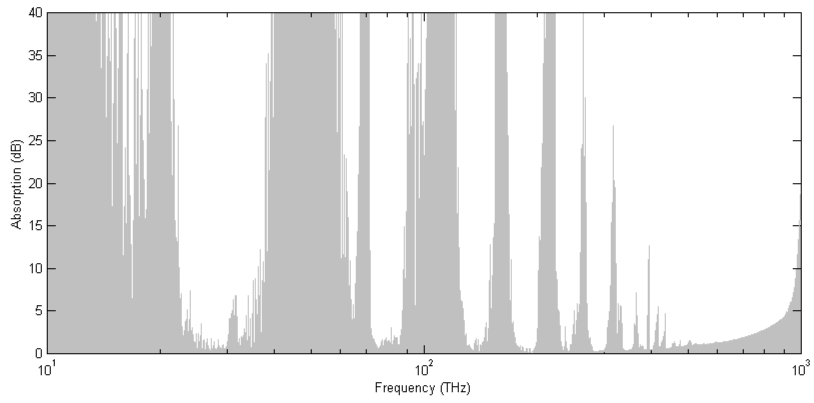


Fig 2—Absorption (shaded area) above 10 THz of a standard atmosphere along a vertical path. □□