By Edward F. Steinfeld,
Industry Marketing Consultant,
Automata International.

FAT32 is made for data-intensive embedded applications

Writing multiple streams of high-speed data to a disk can be a frustrating programming experience. Some potential loss of data due to head seek times can be reduced using double buffering and pipes. But that makes for complex programming and reliance on either a high-level kernel or your expertise to handle the data. An easier method is to use the Windows 95 FAT32 file system along with a contiguous file system to write these high-speed streams of data to a disk.

This article presents an application of this method to show you how this is accomplished and then shows how the data can be presented to a Windows 95 application for presentation or manipulation. The application records a multichannel audio input stream to a disk for later use by a Windows 95 application.

It demonstrates the use of contiguous block allocation for embedded systems. It also demonstrates the convenience of using standard file system API calls to manipulate the files.

N (10) audio channels are multiplexed in the time domain. A DSP front-end digitizes the input into discrete 512-byte packets that are then written to a contiguous section of the disk. This section of the disk is assigned to N files that are interleaved in a cyclic pattern, so each block is assigned to a separate file representing the channel. Every Nth block is owned by a file assigned to channel N.

Once the data is collected, it must be demultiplexed and each channel must be streamed through an audio player. Due to the high data rates involved, it is not possible to perform disk seeks during the record or the playback sessions; the multiplexed data must be stored contiguously while it is collected and then demultiplexed to contiguous per-channel audio streams so the sound files can be played back.

- Both the record and playback sessions require real-time response. During the transition from a record to playback session, you have time to move the data around.
- The resulting sound files must be accessible by a sound editor application running under Windows 95.

Programmed IO is used to demonstrate the technique. In your application, the record and playback routines could use DMA to transfer the data to and from the contiguous regions of the disk, and the file system code could be executed by the DSP front-end.

To accomplish these goals simultaneously, we use an embedded realtime file system, the ERTFS Version 1.0 product, from EBS, Inc. (URL: www.etcbin.com). This embedded file system is Win95 compatible, has contiguous file support, and has direct block manipulation routines. Many embedded applications or embedded kernels have support for contiguous files and may have routines to directly manipulate disk blocks. But ERTFS has this plus Win95 FAT32 support, and the functionality can be added to either a stand-alone application or to an existing kernel.

Because the data sets are quite large and our blocking requirements are small (512 bytes per block), this application uses a large 2.1 Gigabyte hard disk formatted with the FAT32 file system. This format will provide a cluster size (minimum allocation unit) of one block per cluster.

THE FAT32 FILE SYSTEM

The FAT16 file system, the file system of the MS DOS, Windows 3.1, and most versions of Windows 95 operating systems, is 21 years old. It was first developed for floppy disks. The FAT (file allocation table) has been modified over the years to accommodate ever larger disks. It has finally reached its limit with the 2 Gigabyte drives.

The FAT32 file system is an enhancement of the FAT16 file system and now supports larger hard drives with improved disk space efficiency. What makes the space usage more efficient is the smaller clusters used on the larger disks. For disks up to 8 Gigabytes, the default cluster size is a modest 4 KB compared to the 32 KB cluster size for a 2 Gigabyte drive using the FAT16 format. For our application, we formatted a disk with ERTFS to specify a 512-byte cluster size.

There are some drawbacks to the use of the FAT32 format. The Windows NT operating system does not use the format, and converters are not available for Windows NT 3.5 and 4.0 systems. Windows NT 5.0 systems are supposed to have a conversion utility to convert from FAT32 format to NTFS (the native NT file system).

It was been pointed out by an embedded kernel developer the FAT32 format may have some performance problems in certain applications:

- With FAT32, twice as much data has to be read on FAT searches.
- With long file names, 3 to 5 times as much data has to be read on directory searches.
- The memory requirement (e.g. buffers for expanded Unicode filenames) is higher for a FAT32 file system. Code size will typically also be higher.

- Small cluster sizes are a performance disadvantage for large files, as they require more FAT accesses.

Another drawback: some compatibility problems with existing Application Programming Interfaces (APIs) and older MS DOS utilities. The cluster values for FAT32 now use 4 bytes as compared to 2 bytes in the FAT16 system. The Win32 APIs are not affected, and all disk utilities bundled with Windows 95 have been updated.

It is this 4-byte cluster value that makes the FAT32 format so relevant to high-speed contiguous files. You can define clusters that are small enough to be equal in size to the blocks of data being collected and written to the disk. The older FAT16 cluster size value could not accommodate a large number of clusters, so it had to use ever larger cluster sizes as the disk size increased. This kept the value of the cluster size small enough to fit into the 2 bytes provided for the value.

For a more technical description of the FAT32 file system, see www.microsoft.com/windows/pr/fat32.htm.

CONTIGUOUS FILES

The normal disk structure has data written in the first available space. When that space is filled, a link is created to the next available disk space. When writing to the disk, the disk head is constantly going back and forth between the file allocation table (FAT) and the area on the disk where the data is being written. If a program could know before hand that the entire file could be written in a contiguous portion of the disk and if it knows the address of this space, the program could issue direct read and write commands. This is what the ERTFS, LynxOS, and SMX products provide to the developer. (Note: The ERTFS and SMX products are MS-DOS compatible and the LynxOS product is POSIX compliant.)

ERTFS API CALLS	DESCRIPTION
`po_open`	`Open/create a file`
`po_close`	`Close a file`
`po_read`	`Read from a file`
`po_write`	`Write to a file`
`pc_unlink`	`Delete a file`
`pc_get_free_list`	`Get a disks free map`
`po_extend_file`	`Extend a file specifying block allocation`
`pc_get_file_extents`	`Retrieve the block map of a file`
`pc_raw_write`	`Direct block write to disk`
`pc_raw_read`	`Direct block read from disk`

Table 1 ERTFS API calls used in this application

A contiguous file is a section of the disk consisting of contiguous or sequential physical blocks which, after being allocated, are treated as a high-performance raw device. A contiguous file usually does not use the normal buffered file system but is accessed in block units of the file system.

The FAT32 compatible format in our application uses a 512-byte cluster size, which means all accesses to contiguous files are made in units of 512 bytes. Using this cluster size, we are able to interleave our 10 channels in a contiguous region of the disk. Because the DSP is collecting data in 512-byte blocks per channel, we are as efficient as possible in our disk writes of one cluster (512 bytes) per write.

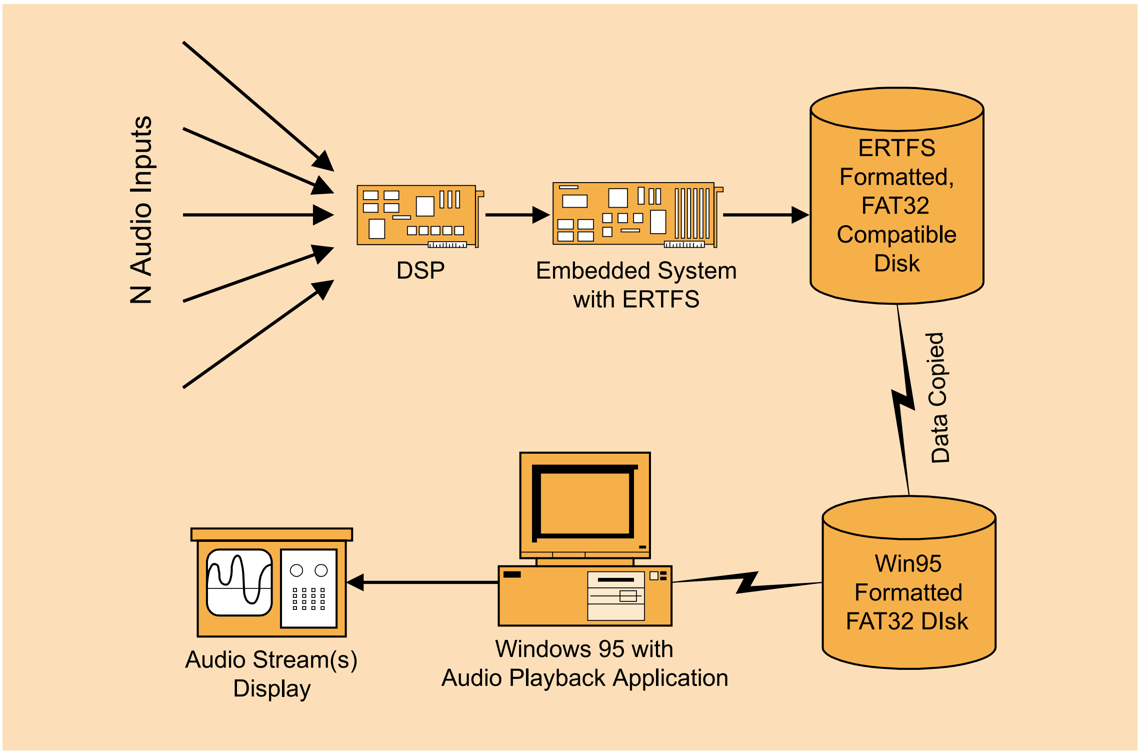
N Audio Inputs
DSP
Embedded System with ERTFS
ERTFS Formatted, FAT32 Compatible Disk
Data Copied
Win95 Formatted FAT32 DIsk
Windows 95 with Audio Playback Application
Audio Stream(s) Display

Figure 1 N (10) channel multistreaming audio application

The ERTFS file system has functions to read and write up to 128 blocks directly to and from a disk. With these functions, you specify the starting block number and the number of blocks to transfer.

Although DOS APIs may happen to create contiguous files, there is no way to specifically request a contiguous file without contiguous or sequential file support. Any contiguous file created, however, can be read by DOS-compatible utilities.

PROGRAM STEPS

The algorithm implemented for this data-streaming application requires the following seven steps.

Step 1. Allocate a contiguous segment of the disk to store the incoming data stream.

Step 2. Interleave the blocks in the contiguous segment, so that every Nth (10th) block is associated with the same file. N is the number of input channels; in this case, 10.

Step 3. For the output files, allocate another contiguous segment of the disk to store a copy of the input stream in N (10) contiguous files (one for each channel).

Step 4. Collect the data and store interleaved in the contiguous segment of the disk created for the input data.

Step 5. Demultiplex the data by copying the data from the interleaved input file to the contiguous output files.

Step 6. Play back each channel.

Step 7. Clean up the disk.

The listings of the routines to execute this program have listing numbers that match the step numbers. The main routine and the definitions are in Listing 0.

Step 1. Allocate contiguous disk space

In Listing 1 the routine scans the disk drive for enough free space to run the application. If it finds the space, the size will be returned in the free_list array. If it does not find enough space, it will analyze the disk and print the free map for informational purposes.

Two contiguous regions on the disk are needed to hold the data collection of size (NUM_CHANNELS * NUM_BLOCKSPER) blocks - 512,000 blocks. The routine will first try to allocate all the data from one region. If that will not work, it will try to allocate the data in two segments.

All of the routines return 0 on success and -1 on failure.

```
int find_session_free_space()
{
int freelist_size;
int i;

/* Call ertfs and ask for a list of
free segments. The first argument is the
drive number, the second is the size of
the free_list array the third is the
free list and the fourth is the minimum
size of contiguous regions to report */
/* Try to get all of the space in one
chunk */
freelist_size =
    pc_get_free_list(DRIVENUMBER,
    FREELISTSIZE, &free_list[0],(2 *
    (NUM_BLOCKSPER*NUM_CHANNELS)));
if (freelist_size >= 1)
    return(0);        /* Got it */

/* Couldn't get it in one chunk try it
in two */
freelist_size =
    pc_get_free_list(DRIVENUMBER,
    FREELISTSIZE, &free_list[0],
    (NUM_BLOCKSPER*NUM_CHANNELS));
if (freelist_size >= 2)
    return(0);        /* Got it now */

/* There isn't enough contiguous space
to do what we want */

/* Just dump the free list to the con-
sole and return failure */
/* Note threshold of one will return all
free regions */
freelist_size =
    pc_get_free_list(DRIVENUMBER,
    FREELISTSIZE, &free_list[0], 1);

printf("Free List\n");
printf("CLUSTER     LENGTH\n");
for (i = 0; i < freelist_size; i++)
{
    printf("%8ul    %8ul\n",
    free_list[i].cluster,
    free_list[i].nclusters);
}
printf("Storage allocation failed\n");
return(-1);
}
```

Listing 1 Call the ERTFS routines and find free disk space.

Step 2. Create the interleaved files

In Listing 2, 10 interleaved files over a single contiguous segment of the disk are created. (Actually it could be N number of channels and files where N is any number.) This routine will create space for 10 interleaved files, each containing 1000 blocks of data. The data will be laid out so every Nth block belongs to a specific input channel (Figure 2). CH0 is a member of INPUT_FILE_1; CH1 is a member of INPUT_FILE_2 and so on.

```
int create_input_data_files()
{
int n, j;
PCFD fd;
long cluster;
for (n = 0; n < NUM_CHANNELS; n++)
{
     /* Open channel n */
     fd = po_open(infile_names[n],
        PO_BINARY|PO_RDWR|PO_CREAT,
        PS_IWRITE|PS_IREAD);
     if (fd < 0)
     {
     printf("File creation error \n");
     return(-1);
     }
     /* The start cluster offset for the
     files is 0,1,2,3,.. for each of the
     channels 0,1,2,.. respectively */
     cluster = free_list[0].cluster + n;

     /* Now allocate one block every
     N'th block for the data channel */
     for (j=0; j < NUM_BLOCKSPER; j++)
     {
        if (po_extend_file(fd, 512,
        cluster, PC_FIXED_FIT,
        FALSE) < 0)

        {
        printf("File extend error \n");
        po_close(fd);
        return(-1);
        }
        cluster += NUM_CHANNELS;
     }
     /* Close this file and go do anoth
     er */
        po_close(fd);
     }
     /* Okay. We have created 10 files
     of 512000 bytes each. The blocks
     are laid out CHAN0|CHAN1|CHAN2
     |CHAN3....|CHAN0.. */

     return(0);
}
```

Listing 2 Creates N (ten) interleaved files over a contiguous segment of the disk

Step 3. Allocate space for 10 output files

In Listing 3, the routine creates the N (10) contiguous files to hold data that will be streamed through an audio playback system. These files are used by the either the embedded system or copied to the Windows 95 application system. All blocks within a file should be contiguous to minimize disk head movement and reduce disk access times. This is not necessary for the application to function, but it makes it simpler to implement and also eliminates fragmentation and ensures the output data will be streamed smoothly (no breaks while waiting for disk head movement).

```
int create_output_data_channels()
{
int n;
PCFD fd;
long cluster;
long file_size;
     /* Get the cluster in the contigu
     ous region If we get here we know
     there is enough free blocks */
     if (free_list[0].nclusters >=
     (2*NUM_BLOCKSPER*NUM_CHANNELS))
     {
     /* If we were able to allocate both
     the input and output files in one
     segment we use the second half of
     the segment for the allocation. */
     cluster = free_list[0].cluster +
     (NUM_BLOCKSPER*NUM_CHANNELS);
     }
     else
     {
     /* Otherwise use the beginning of
     the second segment */

        cluster = free_list[1].cluster;
     }

     /* The size in bytes of each of the
     data files */
     file_size = (long)NUM_BLOCKSPER;
     file_size = (long) (file_size * 512);
     for (n = 0; n < NUM_CHANNELS; n++)
     {
     /* Open channel n */
     fd = po_open(outfile_names[n],
     PO_BINARY|PO_RDWR|PO_CREAT,
     PS_IWRITE|PS_IREAD);
     if (fd < 0)
     {
     printf("File creation error \n");
     return(-1);
     }
     /* Now allocate one contiguous
     chunk for the output channel
     Note The final argument specifies
     the device driver to preerase the
     data blocks if the device supports
     pre-erase */

     if (po_extend_file(fd, file_size,
     cluster, PC_FIXED_FIT, TRUE) < 0)
     {
     printf("Outut File extend error \n");
     po_close(fd);
     return(-1);
     }
     /* The next file will be offset by
     NUM_BLOCKSPER */
     cluster += NUM_BLOCKSPER;
     /* Close this file and go do ano-
     ther */
     po_close(fd);
     }
     /* Okay. We have created 10 files
     of 512000 contiguous bytes  */
     return(0);
}
```

Listing 3 Routine to create output files.

Step 4. Collect the input data

In (Listing 4) this routine collects the 10 channels of multiplexed audio data into N interleaved files.

This routine collects data from a Digital Signal Processor (DSP) or any other front-end data collection system and writes the raw blocks to the interleaved data block region. The DSP system provides 100 block buffers of data at a time to the upper layers of the application software. The buffers are in a ring buffer, so the application calls the DSP software layer to provide a buffer. When it returns a buffer, the data is written to disk and once the write is completed the buffer is given back to the DSP layer. The DSP system and the application run asynchronously.

```
int collect_data()
{
PCFD fd;
long blockno;
char *pdata;
FILESEGINFO fileinfo;
int n_samples;
     /* Open channel 0 and get the block
     number of the first block in the
     file. This is the beginning of the
     contiguous region we allocated */
     fd = po_open(infile_names[0],
     PO_BINARY|PO_RDWR,0);
     if (fd < 0)
     {
        printf("File open error\n");
        return(-1);
     }
     /* Ask for a list of block extents
     that make up the file.
     we only ask for a list of one since
     we only need the first block. The
     raw_io flag is false since we will
     include partition mapping when we
     write the file */
     if (pc_get_file_extents(fd, 1,
     &fileinfo, FALSE) < 0)
     {
        po_close(fd);
        printf("Error getting file
        extents\n");
        return(-1);
     }
     /* Close the file.. we don't need
     it any more. */
     po_close(fd);
     blockno = fileinfo.block;
     /* Here it is */
     /* How many samples to collect
     (total number of blocks/SAMPLESIZE)
     */
     n_samples =
(NUM_CHANNELS*NUM_BLOCKSPER/SAMPLESIZE);

     /* Tell the DSP to Collect n_sam-
        ples of SAMPLESIZE each */
     dsp_start(n_samples, SAMPLESIZE);

     /* Now loop. wait for the samples
     and write them to disk */
     while (n_samples--)
     {
     pdata = dsp_get_sample();
     /* Wait for a sample */

     /* Write SAMPLESIZE blocks from
     pdata to the block at blockno, the
     raw_io argument is false because we
     want partition mapping */

     if (pc_raw_write(DRIVENUMBER, pdata,
     blockno, SAMPLESIZE, FALSE) < 0)
     {
        printf("Error writing to disk\n");
        return(-1);
     }
     blockno += SAMPLESIZE;
     /* We just wrote SAMPLESIZE blocks.
     So increment our block pointer. */
     }
     return(1);
}
```

Listing 4 Collect the audio data.

Step 5. Demultiplex the data for output

In Listing 5, the contents of the input data files are copied to the output files. The data is automatically demultiplexed, because the input files are interleaved and the output files are contiguous. This is not a real-time process, because the disk must seek as it reads the data blocks from the input files. The interesting thing about this routine is that a simple file copy using standard API calls demultiplexes the data into contiguous output files.

Next we finally get to stream the contiguous data to the audio player. This can be either done by the embedded system or through a Windows 95 application.

```
/* Demultiplex the input data so it may
be streamed to the audio player */
int copy_input_to_output()
{
int n,i;
PCFD in_fd;
PCFD out_fd;

     for (n = 0; n < NUM_CHANNELS; n++)
     {
        /* Open channel n */
        in_fd = po_open(infile_names[n],
        PO_BINARY|PO_RDONLY,0);
        out_fd = po_open(outfile_names[n],
        PO_BINARY|PO_WRONLY,0);
        if ( (in_fd < 0) || (out_fd < 0) )
        {
        if (in_fd >= 0)
                po_close(in_fd);
        if (out_fd >= 0)
                po_close(out_fd);
        printf("File open error \n");
        return(-1);
        }
        /* Now read from the input and
        write to the output */
        /* Work 20 blocks at a time
        since we have a 10240 byte
        buffer */
        for (i = 0; i < NUM_BLOCKSPER;
        i += 20)
        {
        if (!((po_read(in_fd, big_buffer,
        10240) == 10240) &&
        (po_write(in_fd, big_buffer,
        10240) == 10240)
                ))
        {
                po_close(in_fd);
                po_close(out_fd);
                printf("File copy error \n");
                return(-1);
                }
        }
        /* close the files and loop back
        for the next pair */
        po_close(in_fd);
        po_close(out_fd);
     }
     /* Okay we've copied all of our
     files. We are done */
     return(1);
}
```

Listing 5 Demultiplex (copy) the data into separate contiguous files.

Step 6. Play back each channel

In Listing 6, the routine assumes the embedded system is used to play back the audio streams. It reads blocks from output files that are contiguous and commits the blocks to the DSP system to be played as audio.

```
int copy_output_to_audio()

/* Play the data on the output device */

{
PCFD fd;
long blockno;
char *pdata;
FILESEGINFO fileinfo;
int i;
int channel;
     for (channel = 0; channel <
     NUM_CHANNELS; channel++)
     {
     /* Open the channel */
     fd = po_open(outfile_names[channel],
     PO_BINARY|PO_RDONLY,0);
     if (fd < 0)
     {
        po_close(fd);
        printf("File open error \n");
        return(-1);
     }
     /* Get the starting block of the
        file. We know it is contiguous */
     if (pc_get_file_extents(fd, 1,
     &fileinfo, FALSE) < 0)
     {
        po_close(fd);
        printf("Error getting file
        extents\n");
        return(-1);
     }
     else
        blockno = fileinfo.block;
        /* Here it is */
     /* Close the file.. we don't need
        it any more. */
     po_close(fd);

     /* Now play all data in one channel */
     /* How many samples to play is
        (total number of blocks/
        SAMPLESIZE) */
     for (i = 0; i < (NUM_BLOCKSPER/SAMPLESIZE);
     i++)
     {
        /* Call the DSP for a place to
           put the data */
        pdata = dsp_get_play_buffer(SAMPLESIZE);
        /* Read the contiguous blocks
           from the disk */
        if (pc_raw_read(DRIVENUMBER,
        pdata, blockno, SAMPLESIZE,
        FALSE) < 0)
        {
                printf("Error reading
                output sample\n");
                return(-1);
        }
        dsp_play_play_buffer(pdata);
        /* Tell DSP subsystem it is
          loaded */

        blockno += SAMPLESIZE;
        /* Increment our block pointer. */
        }
     }
     return(0);
}
```

Listing 6 Play the data back.

Step 7. Clean up the disk

The routine in Listing 7 deletes all the input files and output files to release the contiguous space. This routine calls pc_unlink() for each file that may have been created. If the algorithm ran to completion, it will delete every file. If it did not run to completion but left some files on the disk, it will delete only those files and the unlink call will fail on the files that we did not create. This will not cause any harm.

```
void delete_all_files()
{
int n;
     for (n = 0; n < NUM_CHANNELS; n++)
     {
        pc_unlink(infile_names[n]);
        pc_unlink(outfile_names[n]);
     }
}
```

Listing 7 Delete all the files so you can do it all over again.

Main Program

The main session and the program definitions are in Listing 0. The main program makes sure there is enough contiguous disk space to collect the multiplexed data in a contiguous section and to store the demultiplexed data files in contiguous sections. Then it creates NUMCHANNEL interleaved files so the multiplexed input stream can be collected to contiguous sectors and later demultiplexed. After it collects the data, it demultiplexes the data by copying the data on a per-channel basis from the interleaved input files to the contiguous output files. Then it plays back each channel and deletes all the files.

FAT32 and Embedded Files Work Together

Using the Windows 95 FAT32 file system with its smaller cluster sizes, that can be sized to your incoming data stream, and your embedded application to allocate contiguous files and to read and write directly to disk blocks can make for a fast data collection system for continuous data streams. The reduced head seek times made possible by using contiguous files mean you will not lose incoming data. The ability then to copy the input file into any number of contiguous output files means the output device can stream the data without breaks due to disk head movement.

The ERTFS product has a unique function that may be required when the length of the data stream is not known. If the file size you preallocated was not sufficient to contain the data stream, the ERTFS product has a function to extend the contiguous file. The ERTFS API provides for calls to get the first available contiguous chain of clusters of sufficient size to contain the extension (fastest method), the chain of clusters that allow for the best fit, or the longest chain of available clusters. The extension may not be contiguous with the first allocated space but might be linked like non-contiguous files.

Some embedded kernels have contiguous (sequential) file capability. The ERTFS file system from EBS, Inc. can be used either with kernels that do not possess this feature, or it can be incorporated easily into embedded products that do not contain a file system.

```
#define DRIVENUMBER   0       /* Assume drive number 0. We only have one drive */
#define FREELISTSIZE 200      /* Assume a maximum of 200 discontiguous free segments */
#define NUM_CHANNELS 10       /* We will be collecting 10 audio channels */
#define NUM_BLOCKSPER 1000    /* 512000 bytes per channel is collected */
#define SAMPLESIZE 100        /* 100 blocks per dsp sample */

/* Global array to hold the free segment list */
FREELISTINFO free_list[FREELISTSIZE];
/* Buffer for moving chunks of data around */
unsigned char big_buffer[10240];

char *infile_names[NUM_CHANNELS] = {
              "input_file_0.snd",    "input_file_1.snd",
              "input_file_2.snd",    "input_file_3.snd",
              "input_file_4.snd",    "input_file_5.snd",
              "input_file_6.snd",    "input_file_7.snd",
              "input_file_8.snd",    "input_file_9.snd"};
```

CONTINUED ON PAGE 99

Listing 0. The Main Session

CONTINUED FROM PAGE 98

```
char *outfile_names[NUM_CHANNELS] = {
               "output_file_0.snd",   "output_file_1.snd",
               "output_file_2.snd",   "output_file_3.snd",
               "output_file_4.snd",   "output_file_5.snd",
               "output_file_6.snd",   "output_file_7.snd",
               "output_file_8.snd",   "output_file_9.snd"};

/* hypothetical DSP subsystem */
extern void dsp_start(int n_samples, int samplesize);
extern char *dsp_get_sample();
extern char * dsp_get_play_buffer(int samplesize);
extern void  dsp_play_play_buffer(char *pdata);

/*  Capture N channels of multiplexed audio and play it.
    Returns 0 on success and -1 on failure. */
int record_and_play()
{
int return_value;

    /* Make sure we have space and load the coordinates into the global array
       free_list */
    if (find_session_free_space() == -1)
    {
       printf("Not enough contiuous disk space for session\n");
       return(-1);
    }

    return_value = -1;       /* If we break out before completion report error */

    /* Create N interleaved files for incoming multiplexed data
       stream */
    if (create_input_data_files() == -1)
       printf("Failed to create input files\n");
    /* Create N contiguous files for outgoing audio streams */
    else if (create_output_data_channels() == -1)
       printf("Failed to create output files\n");
    /* Collect mutiplexed data to the interleaved files */
    else if (collect_data() == -1)
       printf("Failed while collecting data\n");
    /* Demultiplex the data by copying it from the interleaved files to the
       contiguous files */
    else if (copy_input_to_output() == -1)
       printf("Failed copying input to contiguous output files\n");
    /* Stream the demutiplexed data from the contiguous files to the
       audio system for playback */
    else if (copy_output_to_audio() == -1)
       printf("Failed while playing back audio\n");
    /* If none of the subsystems failed report success */
    else
       return_value = 0;
    /* Clean up. */
    delete_all_files();
    /* And leave */
    return(return_value);
}
```

Listing 0. The Main Session

SOURCES

ERTFS

EBS, Inc.
Box 873
Groton, MA 01450-0873
(978) 448-9340
Fax: (978) 448-6376
http://www.etcbin.com

FAT32

Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
(206) 882-8080
Fax: (206) 936-7329
http://www.microsoft.com/windows/pr/fat32.htm

LynxOS

Lynx Real-Time Systems, Inc.
239 Samartian Drive
San Jose, CA 95124
(498) 879-3900
Fax: (408) 879-3920
http://www.lynx.com

SMX

Micro Digital, Inc.
12842 Valley View Street, #208
Garden grove, CA 92845
(714) 373-6862
Fax: (714) 891-2363
http://www.smxinfo.com

Ed has more than 25 years experience in realtime and embedded computing. He began as a programmer writing code to test hybrid circuit boards. He has marketed embedded and realtime products to OEMs and resellers for Digital Equipment Corporation, VenturCom, Inc., and Phar Lap Software. He has international experience including working in Hong Kong as the Far East Channels Manager and was responsible for international OEM sales in Europe and the Pacific Rim. He now provides marketing services to these same markets.