Using STREAM Files

Relative to HMP 6.0 & SSR 47.1 *by Paul H. Kimpel*

The MCP has an extremely rich Logical I/O subsystem, especially in the capabilities it provides for disk files. The way the MCP has traditionally organized and accessed disk files, however, is quite a bit different from most other systems. In particular, Unix, Linux, and Microsoft Windows all treat disk files not as a set of records, but as a continuous array (or *stream*) of data.

The MCP has supported stream files for more than ten years, starting in Mark 3.9. With the integration between MCP and Microsoft Windows environments offered by the ClearPath product line, along with the rapid rise of electronic file transfer and the Internet, the ability to directly access stream-oriented files in ordinary MCP applications has become increasingly important.

This article describes the MCP implementation of stream files, how they differ from traditional MCP files, and the restrictions that apply to streams that do not apply to other kinds of files. It also describes the file attributes that are relevant to stream files and how you program for streams.

Stream files are presented here as if there are two types—record streams and byte streams. There is really only one type, with byte streams implemented as a special case of record streams. The way in which you typically program for and use byte streams, however, is very different from that for the more general case of record streams. Therefore, this article maintains a distinction between record streams and byte streams to separate the pragmatics of dealing with them.

Both types of stream files are available in MCP applications, but byte streams are the more interesting and useful. This article focuses primarily on byte stream files, including specific programming techniques, utilities in the MCP, and file transfer mechanisms. Finally, it presents some examples illustrating the use of byte stream files in Algol and COBOL, and points out areas of the Unisys documentation where you can get more information.

About Stream Files

Traditional MCP files are organized around the concepts of logical records and physical blocks. User programs read and write files in units of logical records. By default, these units are all of a fixed length, but a number of methods exist for handling variable length records as well. The MCP automatically assembles and disassembles these logical records into blocks as they are moved between the user program's record area and internal memory buffers. These fixed-length blocks become the unit of data that is physically transferred between memory and the disk device. The creator of a file determines the sizes of the records and blocks, which are maintained as attributes of the file in the disk header.

In contrast, user programs in other systems traditionally use stream I/O. The concepts of records and blocks do not exist, and there are typically no attributes that describe the internal structure or organization of a file. Instead, the programs simply read and write arbitrary lengths of data at arbitrary relative offsets into a file. The closest concept these systems have to a record is that of a "line"—a variable-length sequence of bytes terminated by one or more delimiter characters, such as a carriage-return or line-feed. We usually refer to these line-oriented files as "text files."

Stream methods can be applied to files on disk, CD-ROM, printer backup, or tape. This article focuses on stream files for disk.

Streams in the MCP

You encounter stream files from a number of sources within the MCP environment. A common source is **Client Access Services**, formerly known as NX/Services. When you transfer a file to an MCP shared directory using Microsoft Networking, by default you get a byte stream file.

The **File Redirector** is the complement to Client Access Services. It allows you to create or access files on a remote shared directory using an MCP application program. These remote shares typically reside on a Windows file server, but they can exist on any system that supports Server Message Block (SMB) protocols, such as SAMBA, which is available for many versions of Unix and Linux. Since the Redirector accesses files stored under another operating system directly, the files do not behave as traditional MCP files, and MCP applications must access them as byte streams.

Byte stream files can also be created by both **File Transfer Protocol** (FTP) and **OSI FTAM**. Files generated by incoming FTP transfers with an `FTPSTRUCTURE` of `FTPFILE` are stored as byte streams. This includes the FTP mapping styles `RAW` and `FTPDATA`. While not as commonly used as FTP, FTAM can also access and transfer byte stream files in the MCP environment.

Files read directly from ISO 9660 and Joliet **CD-ROM** media are always accessed as byte stream files.

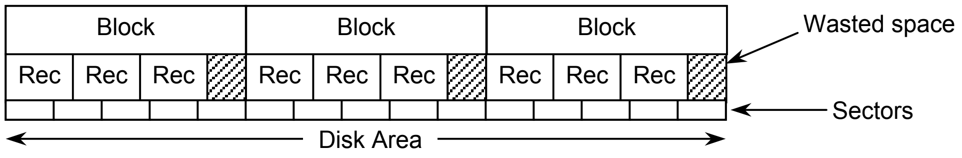
The MCP **Print System** can both generate and print byte stream files. You can specify file attributes that cause the Print System to generate byte stream files instead of the standard printer backup file format. Files generated by the `PC` and `RTF` printer drivers in the `PRINTSUPPORT` library are byte stream files. The Print System can also read stream files generated by other systems and route them for local or remote printing.

MCP-based applications in all languages can generate and read stream files. This article discusses how to do this using Algol, COBOL-74 and COBOL-85. The POSIX interfaces (typically used with C) inherently read and write byte stream files. In addition, permanent directories are considered to be byte stream files.

Stream File Structures

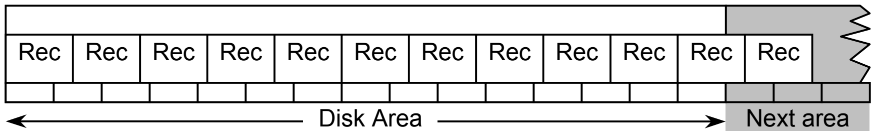
Anyone who has built MCP applications has probably had to fuss over block size calculations for their files. Records for traditional files must be contained entirely within a block. Blocks for a file are all of the same fixed length, so the size of the block is constrained by the number of whole records what will fit within it. Since disk devices read and write data in multiples of 180-byte sectors, blocks do not always fit perfectly in sector increments. Any difference between the declared block size and an exact multiple of 180 bytes represents wasted space on disk, and is sometimes referred to as "block slop." To minimize this wasted space, good file design practice attempts to create block sizes that are equal to (or slightly less than) a multiple of 180 bytes.

The figure below shows the layout of records and blocks in one disk area for a typical traditional file. The rectangles along the bottom represent physical disk sectors. The area is divided into blocks, each the same whole number of sectors in length. Records do not span blocks and blocks do not span areas. Since the record size in the figure does not create a block that is a multiple of 180 bytes in length, there is wasted space at the end of each block. This example shows fixed-length records, but the arrangement is similar for a file with variable-length records.



Stream files take a different approach in the way that records are laid out on disk—*they do not have blocks!* In fact, specifying the `BLOCKSIZE` attribute for a stream file results in a non-fatal attribute error.

The next figure shows the layout of records on disk for a stream file. The logical records are simply written continuously across the allocated space for the file. Just as with traditional files, there are no delimiter codes between records. Individual records span sector boundaries, and as shown, can even cross area boundaries, but logical records are not constrained to a fixed, pre-determined block size, hence *there is never wasted space on disk*.



The MCP Logical I/O routines automatically fragment and reassemble records as necessary when moving data between the user program record area and the stream file's memory buffers. Application programs simply perform reads and writes the same way they do with traditional files. Unlike traditional files, which always read or write whole records with each physical I/O, stream files may perform more than one physical I/O to read or write one logical record. Physical disk I/Os, of course, still start on a sector boundary and extend for a whole number of sectors.

Although stream files do not support the concept of blocks, memory buffers are still of a fixed size. By default, the MCP determines the buffer size, but you can override it with the `BUFFERSIZE` attribute.

Record Streams

Of the two types of stream files, record streams are the more general case. The way that you program for and use record streams is very similar to that for traditional files.

Configuring Disk Files

Setting up an MCP application to use stream files is largely a matter of specifying the proper attributes. There are a number of attributes that interact, so it is useful to begin by looking at the three file attributes that control the general format and nature of disk files:

`FILEORGANIZATION`
: describes the organization of data within a file at a level higher than records and blocks. There are a number of mnemonic values for this attribute, but they break down into three main classes: normal or "flat" files (`NOTRESTRICTED`, the default), indexed sequential files (`KEYEDIO`, `KEYEDIOII`, plus some others), and relative files (`RELATIVE`).

`BLOCKSTRUCTURE`
: determines the format of records within a block. A more intuitive name for this attribute would probably be "recordstructure." It indicates whether the records are fixed length (`FIXED`, the default), variable length (`VARIABLE` plus some others), or externally determined from the physical file medium (`EXTERNAL`).

`FILESTRUCTURE`
: determines how the data is physically laid out on the disk. This attribute has three mnemonic values:

`ALIGNED180` —
: is the default layout and the one used by traditional files. Records are wholly contained within blocks, blocks are the physical unit of I/O transfer, all blocks start on a sector boundary, all blocks occupy a whole number of sectors, and sectors must be 180 bytes in size. These files can be "reblocked"—that is, they can be read with a different block size than the one they were created with, as long as the original and new block sizes are integral multiples of the sector size.

`BLOCKED` —
: is similar to `ALIGNED180`, but has some significant differences. Records are grouped in blocks, and blocks are written on sector boundaries, but the sectors are not required to be 180 bytes in size. Memory buffer size and physical I/Os may be in multiples of the declared block size. `FILEORGANIZATION` must be `NOTRESTRICTED`. Reblocking is not allowed.

`STREAM` —
: *is the attribute value that makes a file a stream file.* As discussed above, records are written to the disk in a continuous stream, spanning sector and area boundaries as necessary. Physical I/Os are buffered in memory and typically span multiple records. `FILEORGANIZATION` must be `NOTRESTRICTED` or `RELATIVE`. A feature somewhat like reblocking is supported, using the `ANYSIZEIO` attribute.

Configuring Stream Files

To use a record stream file in an MCP application, three attributes must have particular settings.

- FILESTRUCTURE must have a value of STREAM. Since the default value is ALIGNED180, this value must be explicitly specified.
- FILEORGANIZATION may be NOTRESTRICTED or RELATIVE. The default is NOTRESTRICTED, and for most applications, is the only value of interest.
- BLOCKSIZE *must not be specified for a stream file*. Attempting to do so causes a non-fatal attribute error at run time.

You should at least consider the settings of some additional attributes.

- BLOCKSTRUCTURE may be any value except LINKED. The default value is FIXED, but all other variable-length formats are supported for record streams.
- MAXRECSIZE specifies the size of records (for variable-length records, the largest record) in FRAMESIZE units.
- MINRECSIZE specifies, for variable-length records, the minimum record size in FRAMESIZE units. It is not used with fixed-length records.
- BUFFERSIZE determines the number of *words* in memory buffers for the file and the size of physical I/O transfers to disk. This attribute can be read for any type of file at any time. It can be set when a file is closed for files with FILESTRUCTURE=BLOCKED or STREAM.
- AREASIZE or AREALENGTH determine the physical size of disk areas, as for traditional files. AREASIZE is specified in units of records, while AREALENGTH is specified in FRAMESIZE units.

The Unisys documentation recommends that you normally let the MCP determine the value of the BUFFERSIZE attribute, but some programs may benefit from a specified value. Files that are primarily used for random I/O usually perform better with small buffer sizes. Very large files accessed sequentially usually perform better with large buffer sizes. In general, for stream files, do not set this attribute to a value smaller than the size of one record plus two sectors. The default value determined by the MCP will be between 2,000 and 5,000 words, depending on the amount of memory configured for your system. For details on this subject, see the discussion of BUFFERSIZE in the *File Attributes Programming Reference Manual* and the BUFFERGOAL option of the SF command in the *System Commands Operations Reference Manual*.

The Unisys documentation also recommends that the MCP should determine the area size for stream files. The MCP value will be the largest size that is a multiple of MAXRECSIZE but not larger than 1,024 sectors. However, for very large files, you probably want to override that default with a larger value. If you let the MCP compute the area size, you usually want to set FLEXIBLE=TRUE so the areas will grow as the file expands. Note that AREASIZE is not meaningful for stream files if BLOCKSTRUCTURE is other than FIXED. For other BLOCKSTRUCTURE values you should specify the area size using AREALENGTH.

There are some restrictions on certain attribute values for record streams and how such files can be used.

- FILEORGANIZATION values other than NOTRESTRICTED and RELATIVE are not allowed.
- The BLOCKSTRUCTURE value LINKED (normally used only by FORTRAN) is not allowed. All other BLOCKSTRUCTURE values can be used with stream files.
- The combination of update I/O (UPDATEFILE=TRUE) and synchronized I/O (SYNCHRONIZE=OUT or writes with the SYNCHRONIZE option) is not allowed. Attempting to do this turns off synchronization.
- Algol Binary I/O (using a "*" as the format part with a list) is not allowed.
- Programs cannot use checkpoint/restart facilities in the MCP if they have stream files open.
- CANDE does not support stream files. You cannot GET, MAKE, LIST, *etc.* a stream file from CANDE. You can, however, use Library/Maintenance commands (COPY, CHANGE, REMOVE, ALTER) and the PRINT command with stream files.
- Most Unisys utilities do not support stream files. Stream files can, however, be listed, copied, *etc.* with SYSTEM/DUMPALL.

There are some other file uses that are potential problem areas when applied to stream files:

- Setting `PROTECTION=PROTECTED` preserves the end-of-file position for open files across a system restart. For stream files, however, the EOF is recovered to the end of the *last sector* written, which is typically after the end of the *last record* written.
- If you rewrite a variable-length record in a stream file, and the size of the record differs from its original size, the MCP generates a record length (data size) error.
- Because stream files are not blocked like traditional files, attempting to access block-oriented attributes results in a non-fatal attribute error at run time. These attributes include:

  ```
  BLOCKSIZE
  BLOCK
  CURRENTBLOCK
  ```
- If `BLOCKSTRUCTURE` has a value other than `FIXED`, access to two additional attributes is not valid, since in those cases both attributes are reported in units of blocks rather than records:

  ```
  AREASIZE
  LASTRECORD
  ```

Programming for Record Streams

Programming for record-oriented stream files is easy. The major consideration is the value of several attributes when the file is opened.

For input files, the easiest thing to do is set `DEPENDENTSPECS=TRUE`. Since `FILESTRUCTURE`, `BLOCKSTRUCTURE`, and `FILEORGANIZATION` are physical file attributes, they will be established automatically from the disk header. You can additionally specify `BUFFERSIZE`, if desired.

For output files, you must specifically declare the file as a stream file, along with record size and any other attributes you need to configure the file.

- In most cases you can simply replace the `BLOCKSIZE` specification with a `FILESTRUCTURE` value of `STREAM`.
- Depending on your specific requirements, you may want to specify the number of words for memory buffers using `BUFFERSIZE` or the length of disk areas with `AREALENGTH`. Usually the MCP defaults for these attributes will suffice.
- If you are creating or extending a stream file and using the MCP default area size, you probably want to set `FLEXIBLE=TRUE`, since you are not controlling the area size.

Once stream files are open, there are essentially no differences compared to the way you access traditional files. The syntax and semantics of `READ`, `WRITE`, `SEEK`, *etc.*, are identical for stream files. With the exceptions noted above, both random and sequential I/O are supported for stream files in the same way they are for traditional files.

All of the discussion so far has been about stream files in general, and more specifically about record-oriented streams. These record streams are useful and efficient. They avoid wasted space on disk and eliminate the need to perform block sizing calculations when designing files. Because the system determines buffer sizes based on total system memory, physical I/O for stream files is often at least as efficient as for traditional files. Except for some inconveniences (such as the inability to access them with CANDE), streams are arguably a superior file structure for many applications.

Even so, the discussion to this point has primarily been in preparation for a special case of stream files—*byte streams*.

Byte Streams

Byte streams are typically the most common and most useful application of stream file structures. They closely resemble the unstructured nature of files for other systems, and are usually compatible with them.

Given the ease and increasing importance of file transfer between the MCP and other operating systems, and the excellent integration with Microsoft Windows that is a hallmark of the ClearPath architecture, the ability to generate and process byte streams directly in MCP applications is extremely useful and often necessary. Byte streams can also be useful in some applications that are solely MCP based. The remainder of this article focuses primarily on byte streams.

Byte Stream Characteristics

A byte stream file is simply a stream of contiguous bytes. The MCP assumes no record or block structure for the file. Any structure within the file must be handled by the application programs reading and writing the file.

Another significant characteristic of byte streams is that the bytes are individually addressable. Programs can seek to any byte position in the file and read an arbitrary number of bytes from that position. To application programs, byte streams are essentially one long character string that can be accessed either sequentially or randomly.

Note that this is a logical view of the file. Physically, the file must still be read and written in units of disk sectors. The MCP's Logical I/O subsystem must handle buffering of data between the disk and the application programs, and must worry about the details of logical reads and writes starting at other than sector boundaries.

One of the most common uses of byte streams on other systems is for text, or line-oriented files. If you have ever used a text editor on another system, such as `NotePad` under Windows or `vi` under UNIX, you have been manipulating a text file.

Text files use delimiter characters to divide the file into logical lines. You can think of these lines as "records" of the file, but the concept of a record size for a text file does not exist. A line can be as long or short as necessary, although some applications place a limit on the maximum length they will correctly handle, often 255 or 1,023 bytes.

Each operating system has its own convention for the delimiters used to divide a file into lines. There are three common ones:

- Microsoft Windows and DOS use a carriage-return (`CR`) followed by a line-feed (`LF`) as the delimiter. Although the delimiter is officially a two-character pair, many text editors and software tools under Windows accept either a carriage-return, or a line-feed, or the `CR-LF` pair as a line delimiter.
- UNIX and Linux systems use the `LF` character as a line delimiter. This is sometimes referred to as the "new line" (`NL`) character. CTOS systems also used this convention.
- Apple Macintosh systems use `CR` as the text delimiter.

A form-feed (`FF`) in the text typically indicates the start of a new page if the file is sent to a printer. Many applications also treat `FF` as a line delimiter.

Byte Streams in the MCP

The MCP supports byte stream files as a special case of record streams. You define a byte stream file the same way you define a record stream file, but a few attributes must have specific values.

The following physical attributes are associated with byte streams. These are stored in the disk header for permanent files.

- `FILESTRUCTURE` must, of course, have a value of `STREAM`.
- `MAXRECSIZE` must have the value `1`. Do not specify `MINRECSIZE`.
- `FRAMESIZE` must have the value `8`, indicating eight-bit character frames.
- `BLOCKSTRUCTURE` must have the value `FIXED`, which is the default.
- `EXTMODE` must be one of the values that is consistent with an 8-bit data frame. `EBCDIC` is the default, but `ASCII`, `OCTETSTRING`, and other mnemonics for 8-bit national character sets are also allowed.
- `FILESTRUCTURE` must be `NOTRESTRICTED`, which is the default. `RELATIVE` is not permitted for byte stream files.

There are two additional physical attributes which are specifically applicable to byte stream files.

`FILECLASS`
: is a read-only attribute that describes the class of a physical file's structure. It has three values:

  `CHARACTERSTREAM` —
  : any disk, printer backup, or CD-ROM file with `FILESTRUCTURE=STREAM`, `MAXRECSIZE=1`, and `EXTMODE` other than `SINGLE`. It is also set for certain TCP/IP port files and POSIX `FIFO` files.

  `WORDSTREAM` —
  : any disk, printer backup, or CD-ROM file with `FILESTRUCTURE=STREAM`, `MAXRECSIZE=1`, and `EXTMODE=SINGLE`, plus certain TCP/IP port files.

  `RECORDORIENTED` —
  : all other physical files.

`EXTDELIMITER`
: specifies the delimiter characters used to separate records or lines in a file. It presently has meaning only for files with a `FILECLASS` of `CHARACTERSTREAM`. The possible values are:

`UNSPECIFIED`	the default
`CR`	carriage-return only
`NL`	new-line (line-feed) only
`CRLF`	carriage-return followed by line-feed
`CRCC`	carriage-return followed by either line-feed or form-feed

The MCP automatically sets `EXTDELIMITER` to `CRCC` when a file with the following characteristics is created:

```
KIND = PRINTER
BACKUPKIND = DISK
FILESTRUCTURE = STREAM
```

In addition, the MCP places the actual `CR`, `LF`, and `FF` characters in the file, creating a delimited byte stream (text) file.

You can specify `EXTDELIMITER` for other types of byte stream files, but the setting is purely advisory. The Print System consults this attribute when printing byte stream files to determine what kind of line delimiters and carriage control to look for.

Some logical attributes also affect byte streams. These are not stored in the disk header, but may be specified at run time before the file is opened.

- Update I/O is not supported for byte streams, so `UPDATEFILE` must be `FALSE`, which is the default. You can open a byte stream in input-output mode and perform mixed reads and writes, but the sequential rewrite-after-read behavior implied by the `UPDATEFILE` attribute is not supported.
- When using byte stream files, you almost always want to set `ANYSIZEIO` to `TRUE`. The reason for this is discussed below in the section on `ANYSIZEIO`.
- `DEPENDENTSPECS`, `DEPENDENTINTMODE`, and `ADAPTABLE` are useful when opening byte streams. These attributes are also discussed below, under Opening Byte Streams for Input.

Note that, since `MAXRECSIZE` is always `1`, record spanning is never an issue with byte stream files. A program could, however, issue a logical read or write request that spans an area boundary, and the MCP would perform more than one physical I/O to satisfy the request.

The same restrictions apply to byte streams as to record streams. There are also some additional restrictions for byte streams on how the files can be declared and used.

- First, the physical attributes of the file *must* include:

  `FILESTRUCTURE = STREAM`
  `MAXRECSIZE = 1`
  `FRAMESIZE = 8`
  `BLOCKSTRUCTURE = FIXED` (the default)
  `FILEORGANIZATION = NOTRESTRICTED` (the default)
  `EXTMODE` must specify an 8-bit representation

- Like record stream files, byte streams cannot be used as CANDE files. CANDE Library/Maintenance commands and the PRINT command can be used with byte streams, however. Most system utilities do not yet support byte streams.
- Since a very common use of byte streams is for text files, it would be nice if the MCP provided the equivalent of a "read line" function for these files as part of Logical I/O. Until recently it didn't. Starting with HMP 7.0 (SSR 48.1), however, the MCP provides a new type of virtual file, STREAMIOH, which can read and write line-oriented byte stream files and present a record-oriented interface to the application. If you are running on an earlier release, you must parse lines in text files yourself, unless you are using the I/O libraries for C. Some of the example programs discussed at the end of this article deal with line parsing in text files.
- MCP application programs are limited to transferring 2^{20}-1 (1,048,575) bytes in a single read or write statement. For most applications, this is not much of a limitation.

Programming for Byte Streams

To open a byte stream file, it must have the minimum physical and logical attribute values discussed above. You may also want to specify BUFFERSIZE or AREALENGTH, but the MCP defaults for these attributes are usually adequate.

Using ANYSIZEIO with Byte Streams

As previously mentioned, you almost always want to set the ANYSIZEIO attribute to TRUE when programming for byte stream files. Here's why.

In the default case, when ANYSIZEIO is FALSE, the number of frames (bytes) transferred between file buffers and your program for reads and writes is limited by the MCP to the minimum of:

- MAXRECSIZE for the file.
- The physical length of the record area in your program.
- The number of frames you request to be transferred in the read or write statement. In Algol programs using array-row I/O, you specify this value directly in the second parameter of a read or write statement. In COBOL programs, however, this value is normally the same as the size of the largest record declared for the file. The section Programming Byte Streams in COBOL, below, describes how to control this value dynamically.

Since by definition a byte stream file has MAXRECSIZE set to 1, this default situation only allows you to read or write one byte at a time. In most cases this results in extremely tedious coding and inefficient execution, due to the overhead of making a separate Logical I/O call for each byte in the file.

The ANYSIZEIO attribute was introduced specifically to address this problem. By setting ANYSIZEIO to TRUE, you remove the restriction imposed by MAXRECSIZE on the number of frames that can be transferred to and from your record area by a read or write. Instead, the size of the transfer is limited by the minimum of:

- The physical length of the record area in your program.
- The number of frames you request to be transferred in the read or write statement.
- The number of frames left in the file.
- 2^{20}-1 (1,048,575) frames.

This means that the transfer spans record boundaries as necessary to satisfy the minimized length requested.

Therefore, setting ANYSIZEIO to TRUE allows your program to read an arbitrary number of frames (up to 2^{20}-1) from the byte stream with one I/O statement. In addition, since the MCP considers the file to have a record length of one frame (one byte), random I/Os can start at any byte position in the file. This allows MCP applications to read and write arbitrary amounts of data at arbitrary locations in the file, just as for byte streams under other operating systems.

While ANYSIZEIO=TRUE is particularly useful for byte streams, it can be set for any stream file which has the attributes:

```
BLOCKSTRUCTURE = FIXED
FILEORGANIZATION = NOTRESTRICTED
UPDATEFILE = FALSE
```

If these restrictions are not adhered to, the MCP issues an open error.

When ANYSIZEIO is used with *record stream* files, data transfer always starts at a record boundary and continues for the number of frames requested, spanning records as necessary. If the transfer does not cover a whole number of records, the remaining frames in the final record are skipped, since the next I/O must start its data transfer at the next record boundary. Byte stream files are not subject to frame skipping, because the records are always exactly one frame in length.

Opening Byte Streams for Input

When reading byte stream files, it's usually best to open them with DEPENDENTSPECS set to TRUE. This initializes a number of attributes from settings for the physical file, including FILESTRUCTURE, MAXRECSIZE, FRAMESIZE, BLOCKSTRUCTURE, FILEORGANIZATION, EXTMODE, FILECLASS, and EXTDELIMITER. You probably also want to set ANYSIZEIO to TRUE.

You can use the INTMODE and EXTMODE attributes to control character translation. Most MCP applications are written to process EBCDIC internally. DEPENDENTSPECS sets EXTMODE from the physical file attributes, but does not affect the INTMODE setting, which defaults to EBCDIC. If INTMODE and EXTMODE are different, translation can occur.

If you are processing image data, or data with mixed text and binary content, you normally do not want translation to take place. By setting the DEPENDENTINTMODE attribute to TRUE, you force the MCP to automatically set the INTMODE for the logical file to the EXTMODE of the physical file. This suppresses character translation.

If you need to have a general-purpose application that can handle any kind of file (traditional, record stream, or byte stream), the restrictions on ANYSIZEIO can cause a problem. If you set ANYSIZEIO to TRUE and try to open a file that does not meet the restrictions for ANYSIZEIO, the MCP issues an open error. By setting the ADAPTABLE attribute to TRUE, however, you cause the MCP to check the compatibility of the file with ANYSIZEIO. If the file *is* compatible, the setting of ANYSIZEIO is not altered; if the file *is not* compatible, the MCP sets ANYSIZEIO to FALSE. This allows the open to succeed, after which your program can interrogate other attributes to determine how to process the file.

There are no differences in the syntax of read statements for byte stream files. You can perform both sequential I/O and random I/O using relative record numbers—just keep in mind that for byte stream files, "record number" means "byte offset into the file."

Random I/O in COBOL is accomplished using the ACTUAL KEY clause of the SELECT statement. You also need to declare the record size in a special way to allow ANYSIZEIO to work in a useful manner, as the section below on COBOL programming describes.

Opening Byte Streams for Output

When creating a byte stream file, you must explicitly set the FILESTRUCTURE attribute to STREAM, since default attribute settings result in a traditional MCP file. You should also explicitly set MAXRECSIZE=1 and FRAMESIZE=8, even if you are programming in COBOL. The required values for the other attributes are the defaults.

As with input files, you probably want to set ANYSIZEIO=TRUE. You may also want to specify INTMODE and/or EXTMODE. Setting INTMODE=EBCDIC (the default) and EXTMODE=ASCII allows your program to write EBCDIC text, but the text is translated and physically stored as ASCII. If you are writing binary or image data, it is usually best to set INTMODE and EXTMODE both to OCTETSTRING.

You can set AREALENGTH or AREASIZE when creating a byte stream file, but in most cases the MCP default of 1,024 sectors will probably be adequate. You should also set a value for AREAS. If you are using the MCP default for area size, you probably also want to set FLEXIBLE=TRUE, so the number of areas will automatically expand as the file grows.

There are no differences in the syntax of write statements for byte stream files. You can perform both sequential I/O and random I/O using relative record numbers, but as with reads, keep in mind that "record number" means "byte offset into the file."

Programming Byte Streams in Algol

Access to byte streams in Algol is very straightforward. You specify the attributes you need directly in the file declaration or through file attribute assignments in WFL, CANDE, or MARC. A typical byte stream file declaration might look like this:

```
FILE STREAM (KIND=DISK, FILESTRUCTURE=STREAM,
    MAXRECSIZE=1, FRAMESIZE=8, ANYSIZEIO,
    EXTMODE=ASCII, INTMODE=EBCDIC, FLEXIBLE,
    FILEUSE=IO, TITLE="MY/BYTE/STREAM ON PACK.");
```

Sequential reads and writes are done in the same way as for traditional files. With ANYSIZEIO set to TRUE, the record length parameter always specifies the number of 8-bit frames to read or write in the file. Typical I/O statements follow these examples:

```
DEFINE MAXCHUNK = 6000 #;
EBCDIC ARRAY REC [0:MAXCHUNK-1];
BOOLEAN RESULT;
INTEGER N, BYTEX, BYTEL;

%-- SEQUENTIAL READ --
RESULT:= READ (STREAM, MAXCHUNK, REC);
BYTEL:= REAL(RESULT).[47:20];

%-- RANDOM WRITE --
RESULT:= WRITE (STREAM[BYTEX], BYTEL, REC[N]);
```

To perform random I/O, you specify an arithmetic expression in square brackets after the file identifier. For byte stream files, this is the *zero-relative* byte offset into the file where the read or write begins transferring data. Algol also allows you to transfer data into or out of the middle of the record area by using a pointer expression or an indexed array identifier.

Algol read and write statements return a Boolean result value that has the same format as the STATE attribute. Some useful fields in this word result are:

Bit Field	Meaning
[47:20]	the number of frames (bytes) actually read or written
[7:1]	parity data transfer error
[9:1]	end-of-file encountered
[4:1]	length or size error (sometimes called "data error")
[0:1]	some exception occurred (this bit determines the result word's TRUE or FALSE value)

Programming Byte Streams in COBOL

Programming for byte stream files in COBOL-74 or -85 is somewhat more cumbersome than Algol, but all of the facilities are available. The coding in both COBOL dialects is identical.

In the SELECT statement, the file should have ORGANIZATION SEQUENTIAL (the default) and an ACCESS MODE of either SEQUENTIAL (again, the default) or RANDOM. If you are doing random I/O or using SEEK statements, you need to include an ACTUAL KEY clause referencing a numeric data item in WORKING-STORAGE. Keys defined as USAGE BINARY or REAL are more efficient than ones defined as DISPLAY or COMP. Here is an example of a declaration of a random file.

```
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 FILE-CONTROL.
     SELECT BSF-BYTE-STREAM
         ASSIGN TO           DISK
```

```
        ORGANIZATION        SEQUENTIAL
        ACCESS MODE         RANDOM
        ACTUAL KEY          W-BSF-KEY
        FILE STATUS         WBS-FILE-STATUS.
```

The `FILE STATUS` clause is not necessary, but it is a useful and convenient way to get detailed I/O results in a COBOL program. It is a two-character value declared in `WORKING-STORAGE`.

When working with byte stream files, you frequently need to be able to read and write variable length strings of data and control the length programmatically. In order to do that, you must declare the COBOL `FD` in a particular way:

- Specify a `RECORD CONTAINS` clause with a range of `1 TO` *`n`*, where *`n`* is the maximum size of the string you need to read or write. The clause must also have a `DEPENDING ON` phrase, naming a numeric data item in `WORKING-STORAGE`. The value of this item controls the number of frames (bytes) read or written by each I/O statement for the file. As for `ACTUAL KEY` items, it is more efficient to declare this length item as `USAGE BINARY` or `REAL`. In COBOL-85 you can alternatively specify a `RECORD IS VARYING` clause with a `DEPENDING ON` phrase.
- Specify the stream-related attributes using `VALUE OF` clauses in the `FD`. Alternatively, you can specify these externally using file attribute assignments in WFL, CANDE, or MARC. When reading existing files, you can usually specify just `VALUE OF DEPENDENTSPECS IS TRUE` and `VALUE OF ANYSIZEIO IS TRUE`.
- Declare at least one record area for the file that is as large or larger than the maximum size specified in the `RECORD CONTAINS` or `RECORD IS VARYING` clause. COBOL allocates the record area in your program based on the longest record area you define for the file.

Here is an example of typical `DATA DIVISION` coding that works for both COBOL-74 and COBOL-85.

```
 DATA DIVISION.
 FILE SECTION.
 FD  BSF-BYTE-STREAM
     RECORD CONTAINS        1 TO 8192 CHARACTERS
                                DEPENDING ON W-BSF-SIZE
     VALUE OF
       FILESTRUCTURE IS     STREAM
       MAXRECSIZE IS        1
       BLOCKSTRUCTURE IS    FIXED
       FRAMESIZE IS         8
       EXTMODE IS           ASCII
       ANYSIZEIO IS         TRUE
       FLEXIBLE IS          TRUE.
 01  BSF-REC.
     05 FILLER              PIC X(8192).

 WORKING-STORAGE SECTION.
 77  W-BSF-SIZE             REAL.
 77  W-BSF-KEY              REAL.
 01  WBS-BYTE-STREAM-INFO.
     05 WBS-FILE-STATUS.
       10 WBS-FILE-STATUS-1  PIC X(1).
       10 WBS-FILE-STATUS-2  PIC X(1).
```

To read or write a string of bytes in COBOL, you first move the length of the string you want to transfer to the data item specified in the `DEPENDING ON` phrase of the `RECORD CONTAINS` or `RECORD IS VARYING` clause for the file, thus:

```
 PROCEDURE DIVISION.

     OPEN I-O BSF-BYTE-STREAM.

*    -- SEQUENTIAL READ
     MOVE 4000 TO W-BSF-SIZE.
     READ BSF-BYTE-STREAM AT END ...

*    -- RANDOM WRITE --
     COMPUTE W-BSF-SIZE = W-INDEX + 1.
     MOVE W-OFFSET TO W-BSF-KEY.
     WRITE BSF-REC INVALID KEY ...
```

It's a good practice to reestablish this length value before each I/O statement, even if you are always reading strings of the same size. The reason for this is that the `DEPENDING ON` data item is updated after each I/O with the actual length of data that was transferred between your record area and the file buffers.

If you are reading or writing randomly, you may also need to place the relative record number (byte offset) where the data is to be read or written into the data item specified in the `ACTUAL KEY` clause. Note that for COBOL this is a *one-relative* byte offset into the file, since COBOL numbers records starting at `1`.

Determining Actual Length

One thing to keep in mind when accessing byte stream files is that you may not get all the bytes you asked for. This happens when you read the last chunk of data at the end of the file. You may have asked for, say, 3,000 bytes, but perhaps there were only 1,255 left in the file before the EOF. The MCP will not read beyond the current end of file, and only transfers data to your program up to that point.

There are several ways to determine how may bytes were actually transferred between the file buffer and your program's record area.

- In COBOL, if you declared the file with a `RECORDS CONTAINS … DEPENDING ON` clause or a `RECORD IS VARYING … DEPENDING ON` clause, the data item referenced by the `DEPENDING ON` phrase is updated after each read or write with the actual number of frames transferred. This is usually the most convenient method.
- As mentioned earlier, the `STATE` attribute holds the result of the last I/O operation in the form of a bit-packed word. Bits `[47:20]` in that word indicate the actual number of frames transferred. The first example below shows how you can extract these bits into a COBOL data item. Note that the partial word notation in square brackets works only with word-oriented data items, such as `USAGE REAL` and `BINARY`.
- The `CURRENTRECORDLENGTH` attribute indicates directly how many frames were transferred by the last I/O operation.

The following example shows these last two methods.

```
 77  W-STATE              REAL.
 77  W-LENGTH             PIC S9(11)  BINARY.

*    -- STATE ATTRIBUTE METHOD --
     MOVE ATTRIBUTE STATE OF BSF-BYTE-STREAM TO W-STATE.
     MOVE ZERO TO W-LENGTH.
     MOVE W-STATE TO W-LENGTH [47:19:20].

*    -- CURRENTRECORDLENGTH ATTRIBUTE METHOD --
     MOVE ATTRIBUTE CURRENTRECORDLENGTH OF BSF-BYTE-STREAM TO W-LENGTH.
```

The last two methods have the disadvantage that you must get their value from an attribute at run time. This involves making a procedure call to the MCP after each I/O statement. While this is a relatively efficient operation, there is some overhead to it, especially if you are doing it thousands of times. Algol provides a more efficient mechanism by returning a copy of the `STATE` attribute as the result of each read or write statement.

In COBOL-85, a mechanism equivalent to the one for Algol is provided by the `MCPRESULTVALUE` special register. After each I/O statement, the COBOL-85 run time stores a copy of the `STATE` attribute in this register, much the same way it stores DMSII results in the `DMSTATUS` special register. Accessing `MCPRESULTVALUE` is many times more efficient than obtaining an attribute value. The actual length read or written can be obtained from this register in the same way as for the `STATE` attribute, thus:

```
MOVE ZERO TO W-LENGTH.
MOVE MCPRESULTVALUE TO W-LENGTH [47:19:20].
```

MCP Tools for Byte Streams

While MCP support for byte streams is not universal, and most of the utility programs from Unisys do not currently support them, there are a number of facilities and tools for the MCP besides Logical I/O that you can use to generate and access byte stream files.

Client Access Services

Client Access Services can access and transfer both traditional and byte stream files between MCP shares and Microsoft Networking clients. The Windows Explorer Extensions add-in, the `MCPCOPY` command line utility, and several types of named pipes can convert byte stream files to and from traditional MCP file formats.

With Client Access Services, you can both transfer and directly access files on MCP shares. Since Windows does not support record-oriented files, the MCP files appear to Windows applications as byte streams having carriage-return/line-feed record delimiters.

When transferring a file to the MCP environment, you can use either Windows Explorer, the `MCPCOPY` command line utility, or a Windows-based application that supports Universal Naming Convention (UNC) file names. If what you want to store in the MCP environment is a byte stream file, *do not* use the Unisys Explorer Extensions add-in that is activated by right-click-dragging a file to an MCP share, as that creates an MCP file in traditional format. Instead, use a left-click-drag.

The Unisys `MCPCOPY` utility can transfer files to the MCP as byte streams or as record-oriented files based on command line switches. In general, to get a byte stream file stored in the MCP environment, do not specify the `/Z:SR`, `/R`, `/D`, `/B`, `/U`, `/T`, `/W`, or `/F` switches for `MCPCOPY`.

There are several Windows APIs and tools that support reading and writing to disk shares with UNC file names, including the `TYPE` and `MORE` commands, `NotePad`, and the `FileSystemObject` component available in any environment that supports Microsoft COM objects, including Visual Basic and VBScript. A UNC file name has the general form

```
\\server\share\pathname
```

instead of the `drive-letter:pathname` form.

Note that there are a number of named pipes supported by Client Access Services that have UNC names starting with `\\server\PIPE\`… These pipes either convert between Windows byte streams and MCP traditional files or connect to CCF or port files for MCP applications. Do not use these if you want to transfer byte stream files to the MCP and have them stored as byte streams.

The File Redirector

The File Redirector, which was first introduced in HMP 5.0, is a complement to Client Access Services. It allows MCP applications to access disk, CD-ROM, and printer shares on a remote system using Microsoft Networking (SMB) protocols. Using the Redirector you can directly read and write files on these shares—no separate file transfer process is necessary. You must program for these files in your MCP application as byte stream files.

The Redirector is implemented as an instance of a new kind of file, `KIND=VIRTUAL`. The semantics of virtual files are not implemented within the Logical I/O subsystem of the MCP. Instead, the semantics are implemented by a library program, called an IOHANDLER, that Logical I/O calls in response to standard open/close, read/write statements in an application. The interface for an IOHANDLER is documented in the *I/O Subsystem Programming Guide*. You can even write your own IOHANDLER if you have a unique input/output processing need.

There are a number of new attributes that support the use of virtual files. Most of these (*e.g.*, `IOHLIBACCESS`, `IOHTITLE`, `IOHFUNCTIONNAME`) are used to establish the library linkage between Logical I/O and the appropriate IOHANDLER when a file is opened. There is also a string parameter, `IOHSTRING`, that is used to pass file-specific options from an application program to the IOHANDLER library.

The File Redirector is implemented by a Unisys-supplied IOHANDLER library, `SYSTEM/REDIRSUPPORT`. There is a new Boolean file attribute, `REDIRECTION`, that can be used as a shorthand to configure the IOH library linkage attributes for `REDIRESUPPORT`. Therefore, to use the Redirector, you have to specify the attributes you would normally need to read or write a byte stream file, plus

- `REDIRECTION = TRUE`
- Some redirection-specific options in `IOHSTRING`

- Optionally, a specially formatted `TITLE` or `LTITLE` string that represents the UNC name for the file on the remote system.

One of the nice things about the Redirector is that if your program already understands how to read or write byte stream files, you can simply use file attribute assignments to have it work with files on remote shares.

The following WFL snippet shows a straightforward example of the attributes needed to access a file called `MiscellaneousFiles\demo.txt` from the share `MYSHARE` on the server `NTSERV`. The first set of attributes are standard byte stream attributes. The second set invoke the Redirector. Note that the long filename attribute `LTITLE` must be used instead of `TITLE` if any nodes of the name contain more than 17 characters.

```
RUN OBJECT/MY/STREAM/PROG;
    FILE SHARE (
        %-- BYTE STREAM ATTRIBUTES --
        FILESTRUCTURE = STREAM, MAXRECSIZE = 1, FRAMESIZE = 8,
        ANYSIZEIO = TRUE, EXTMODE = ASCII, FILEUSE = IO,
        %-- FILE REDIRECTION ATTRIBUTES --
        REDIRECTION = TRUE,
        IOHSTRING = "CREDENTIALS=username/pw",
        LTITLE = *UNC/NTSERV/MYSHARE/"MiscellaneousFiles"/"demo.txt");
```

The `*UNC` prefix on the file name is a special node recognized by the Redirector. It indicates, in place of the standard double backslash (\\), that the rest of the name is in UNC format. Note that forward slashes are used instead of backslashes in these MCP title strings.

The `IOHSTRING` attribute can specify a number of options for the connection to the remote server and share. In addition to `CREDENTIALS`, keywords which can be specified in this string include:

Keyword	Meaning
DOMAINNAME	DNS name for the remote server
IPADDRESS	IP address for the remote server
SERVERNAME	Microsoft Networking host name for the remote server
SHARENAME	Name of the share on the remote server
USERDOMAIN	Domain under which user credentials will be authenticated
TIMEOUT	Timeout value (in seconds) for remote server connections to complete
SMBTRACE	Invokes SMB diagnostics (true/false)

Instead of using the `*UNC` form of the `TITLE` attribute, you can identify the server by `DOMAINNAME`, `IPADDRESS`, or `SERVERNAME`, and the share by the `SHARENAME` options in `IOHSTRING`. The `TITLE` or `LTITLE` attribute would then specify just the pathname after the share name.

Since specifying `CREDENTIALS` in the `IOHSTRING` potentially exposes passwords, the Redirector can use credentials files created by the Unisys utility program `*SYSTEM/NXSERVICES/MAKECREDENTIALS`. This utility generates an encrypted file from a host name (which can be a domain name, IP address or server name), a username, a password, and an optional Windows user-domain name. The file is stored under your usercode as `NXSERVICES/CREDENTIALS/`*`hostID`*. The Redirector automatically accesses this credentials file under your usercode if `CREDENTIALS` is not specified in the `IOHSTRING`.

For more information on the File Redirector, see the *I/O Subsystem Programming Guide*.

FTP

FTP can be used to transfer files between the MCP environment and virtually any other system which supports a minimal implementation of FTP. Since the capabilities of file systems vary widely among systems, FTP accomplishes this nearly universal interchange by reducing the files to a simple set of common characteristics. It leaves to each endpoint of the transfer the task of converting this simple representation of files to a form that is meaningful for that endpoint's environment.

FTP characterizes file transfers in three ways:

Structure
indicates the file can be transferred as a single entity (FILE mode) or as delimited records (RECORD mode). In WFL, this characteristic is specified with the `FTPSTRUCTURE` transform attribute, having val

ues of FTPFILE and FTPRECORD. In the interactive FTP client, this characteristic is controlled by the STRUCT command.

Representation
indicates the data can be encoded in ASCII or EBCDIC, or it can be unencoded. In the unencoded, or binary form, the data is considered simply to be a stream of 8-bit bytes (octets). In WFL, this characteristic is specified with the FTPTYPE transform attribute, having values of ASCIINONPRINT, EBCDICNONPRINT, and IMAGE (LOCAL 8 is a synonym for IMAGE). In the interactive FTP client, this characteristic is controlled by the ASCII, BINARY, IMAGE, EBCDIC, and TYPE commands.

Transfer Mode
indicates how the data is physically transferred. The FTP standard defines this as a third characteristic, but the MCP currently supports only one if its modes, STREAM. There is no transform attribute corresponding to this characteristic.

To convert the standard FTP data streams to the Logical I/O subsystem, the MCP implementation of FTP supports a mapping process during the transfer. Mapping can translate the simplified characteristics of FTP data to a wide variety of the richer file structures available under the MCP. Specifically, the mapping process can convert between byte stream files from other systems and either traditional or byte stream formats under the MCP.

Inbound FTP Transfers

When you transfer files to the MCP environment with FTP, you want to receive the files in either traditional or byte stream format. If byte streams are what you want, you essentially bypass FTP's mapping process, except possibly for character translation.

By specifying the RAW mapping style for an inbound transfer, you are guaranteed of getting a byte stream file stored in the MCP environment. The bytes are stored exactly as sent by the remote system. No character translation is available with RAW. The file is stored with an EXTMODE of OCTETSTRING.

The TEXT mapping style always stores a traditional MCP file, so it is not discussed further here.

The BINARY mapping style may or may not store a byte stream file in the MCP environment, depending on the parameters specified for the transfer.

- If the Structure characteristic for the transfer is FILE and no RECORDLENGTH option is specified in the mapping parameters, FTP stores a byte stream file. With this type of inbound transfer, character translation can occur.
- If Structure is RECORD or a RECORDLENGTH parameter is specified, FTP stores a file in the traditional (blocked) format.

Note that the characteristics of an FTP transfer can be affected by two external sources:

- If the transfer is remotely initiated, the originator has control over the Structure and Representation characteristics of the transfer.
- FTP supports default settings through a hierarchy of configuration files. You may need to override these defaults on a case-by-case basis.

Outbound FTP Transfers

For outbound transfers, FTP can send the file with either the FILE or RECORD structure. If you are starting with an MCP byte stream file, FTP always sends it with a structure of FILE, since the file is not record oriented to begin with.

FTP output mapping supports two modes: a default mapping style and the UNEDITED style.

The default style is typically used with traditional or record-oriented files, and is usually the one to use if you want to transfer text. With this style, character translation can occur, as well as selected trimming of trailing blanks, sequence numbers, and identification (patchmark) fields.

The UNEDITED style for output is the complement of the RAW style for input. It causes the file to be transferred as is, without mapping or character translation. This is the style to use if you need to transfer a traditional MCP file as a binary data stream without record delimiters.

MCP Utilities and Conversion Facilities

Most MCP utility programs do not directly support byte stream files. The new STREAMIOH virtual file handler in HMP 7.0 will enable some of the standard utilities to be used with byte streams through file attribute assignment.

Utility Programs for Byte Streams

Even without STREAMIOH, there are still some standard MCP utilities that support byte streams.

- SYSTEM/EDITOR (U ED) can read, edit, and convert byte stream files. The functionality of this program is oriented primarily toward program symbolic files, so it may not be suitable for dealing with data files having very long records.
- The Print System can generate printer backup files in byte stream format as well as standard backup file format. It can also print byte stream files from other systems, obeying line delimiter conventions and form feeds. The Print System determines the line delimiter from the EXTDELIMITER attribute, which can be set with the WFL ALTER statement.
- SYSTEM/DUMPALL can read and write byte stream files, but the functionality is limited. DUMPALL is driven largely by file attributes. Since MAXRECSIZE is always 1 for byte stream files, listing such a file with DUMPALL results in a tabulation of one-character records, which is not very useful. Copying byte streams with DUMPALL is not very efficient, since each byte in the file is treated as a separate logical record, resulting in a separate read and write for every record in the file.

Byte Stream Conversion Facilities

Sometimes you have a byte stream file in the MCP environment and need to process it using an application that supports only traditional files. In these cases you have two choices: write a program yourself to process the data, or convert the byte stream to a traditional file format.

By writing your own program, you can accomplish any kind of conversion you want, and the process will generally be quite efficient in the MCP environment. The problem with this approach, of course, is the time and effort you need to spend designing and coding an appropriate program. Instead, there are a number of ways to convert byte streams to traditional files using standard MCP facilities.

Client Access Services (NX/Services) offers a number byte stream conversion capabilities, but only when transferring files to or from an MCP disk share. To use these to convert a file that is already in the MCP environment, you would have to first copy the file to an external system, then copy it back to the MCP, or copy it between two MCP shares. In either case, all of the data being converted must travel to an external system and back to the MCP over your network. There is a fair amount of overhead in doing this, but it's fine for a quick-and-dirty solution. The two tools you can use are:

Windows Explorer Extensions
This is a Unisys add-in you install in your Windows environment. It is activated when you right-click-drag files to an MCP share in Windows Explorer. A pop-up menu gives you a number of choices for source file formats, data file record sizes, and file names.

MCPCOPY
This is a command line utility that also runs in the Windows environment. It is installed automatically in your WINDOWS or WINNT directory when you install the Explorer Extensions add-in. You can run this program directly, or use it in ".bat" files and Windows Scripting Host (WSH) scripts. Conversion formatting is controlled by a number of parameter switches, which are documented in the help file for the Explorer Extensions. This help file is also on the documentation CD-ROM.

As mentioned above, the SYSTEM/EDITOR utility can read and write byte stream files and convert to and from traditional symbol file formats. When using EDITOR from CANDE with byte streams, you need to initiate it using its shell program, OBJECT/EDIT (*e.g.*, U EDIT <*file name*>).

FTP has extensive file conversion facilities, and can map byte stream files into a wide variety of traditional MCP file formats, including symbol files, fixed length data files, and variable length data files. The following WFL job shows an example using FTP to convert files locally.

```
? BEGIN JOB BYTESTREAM/CONVERT;
USER = uc/pw;

COPY [FTP]
    'MY/BYTE/STREAM ON DEV' AS MY/FLAT/FILE
      (FTPTYPE=ASCIINONPRINT, FTPSITE=
         "MAPIN TEXT(FILEKIND=COBOL85SYMBOL, RECORDLENGTH=84, " &
         "BLOCKINGFACTOR=30, FOLD=SPACE, ADD=NONE, TAB=INTERVAL 8)")
FROM DISK(HOSTNAME=localhost, USERCODE=uc/pw)
TO DEV(DISK);
? END JOB
```

This converts a byte stream to a traditional file locally on the host using FTP. You don't even have to be connected to a network in order to use FTP this way, but both the TCP/IP network provider and the FTP DSS must be running.

The trick here is that you are using FTP to transfer the file to and from the same host. That's a pretty inefficient way to copy a file, but in doing so you can take advantage of FTP's extensive input data mapping capability. Input mapping allows FTP to convert byte streams into a number of traditional MCP file formats, including all of the symbol file formats supported by CANDE, and most common data file formats.

This example shows how to import a COBOL-85 source file in byte stream format and convert it to a FILEKIND of COBOL85SYMBOL with the correct record size and blocking. The RECORDLENGTH and BLOCKINGFACTOR options are not really needed here, since FTP uses the correct sizes by default when the FILEKIND for a symbol file is specified.

In addition, specifying:

FOLD=SPACE
: folds lines at a space (*i.e.,* between words) if a line is too long for a standard COBOL record.

ADD=NONE
: assumes the byte stream already has sequence numbers in columns 1-6. Without this option, FTP would insert sequence numbers in front of the text.

TAB=INTERVAL 8
: expands tabs to increments of eight spaces.

Note that the mapping options BLOCKINGFACTOR and ADD are new with HMP 6.0.

To adapt this example to your own use, you need to change the source and destination file names, replace "localhost" by the host name of your MCP system, supply an appropriate usercode and password in place of "uc/pw", and change the destination family name ("DEV") to one for your system. You do not need a REMOTEUSER specification in the USERDATAFILE to use FTP locally in this way.

There are a number of options for FTP input mapping other than those shown here. For more information on FTP input and output mapping, see the *TCP/IP Distributed Systems Services Operations Guide*.

Examples

I have written a few example programs that illustrate the basic techniques for reading and writing sequential byte stream files.

There are two examples written in COBOL-74. STREAMLIST reads a byte stream as a text file and lists it to a standard printer file, adding page headings and line numbers, and folding lines that are too long to fit on a single print line. STREAMCOPY copies a traditional MCP file as a byte stream file, trimming trailing blanks and appending CR-LF delimiters after each record. Both of these programs also compile with COBOL-85.

There is also a COBOL-85 version of STREAMLIST. It is functionally equivalent to the COBOL-74 version, but uses some of the nicer control structures available in the 1985 standard.

Finally, there is an Algol version of STREAMLIST as well. It is also functionally equivalent to the COBOL-74 version.

You can download the source for all of these examples from our web site at:

`http://www.digm.com/UNITE/2001`

This article is based on a presentation I gave at the 2001 Fall UNITE conference in Phoenix. You can also download a copy of that presentation in PowerPoint 95 format from the site.

These example programs are also available at the Gregory Publishing download area in file `jan2002.zip`.

Sources for More Information

A number of Unisys documents contain information relevant to byte stream files in the MCP environment. All of these references are relative to the HMP 6.0 release, and all are on the Product Information CD-ROM.

- The *I/O Subsystem Programming Guide* has a wealth of information on programming for various types of files. See Sections 1 and 2 for general information on byte stream files, Section 29 for a discussion of Virtual files, and Section 30 for information on the Redirector. The new `STREAMIOH` virtual file handler is described in Section 31 of the HMP 7.0 edition of this document.
- The *File Attributes Programming Reference Manual* is a dictionary of file attributes and their permissible settings. This is the best source if you know the attribute you need to use but want detailed information on its values and effects.
- The *Client/Server Applications Development Guide* is a standard Windows help file. It is the primary resource for information on MCP file shares and named pipes.
- The Unisys Explorer Extensions add-in comes with another Windows help file which describes how to use the extensions. It also contains documentation for the `MCPCOPY` command line utility. This file is installed automatically when you install the extensions.
- The *TCP/IP Distributed Systems Services Operations Guide* contains complete information on several TCP/IP components, including FTP. Sections 2 through 8 are the best resource for information on FTP and its input and output mapping capabilities.

Acknowledgement

The author thanks David Grubb for his careful review of this article and his insightful comments on it.

Paul Kimpel has been working with MCP-based systems for more than 30 years, starting with the B5500 in college. He began his career at Burroughs in 1970 on the then-new B6500. After leaving Burroughs in the mid '70s, he worked for a number of Burroughs and IBM customers before starting his own business in 1979. That business, now Paradigm Corporation, specializes in consulting, training, and custom software development for ClearPath MCP systems. Paul's main interests are in the areas of data base and transaction processing system design, web enabling MCP applications, TCP/IP networking, and object oriented environments. His email address is `paul.kimpel@digm.com`.