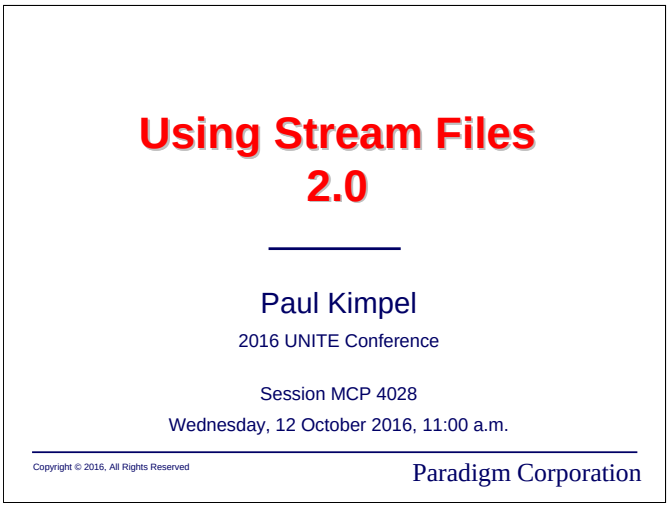



# Using Stream Files – 2.0

2016 UNITE Conference
Cleveland, Ohio

Session MCP 4028

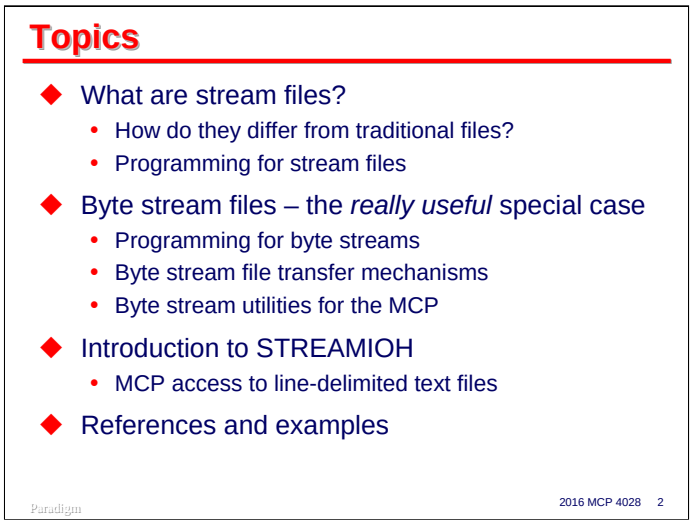
Wednesday, 12 October 2016, 11:00 a.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com

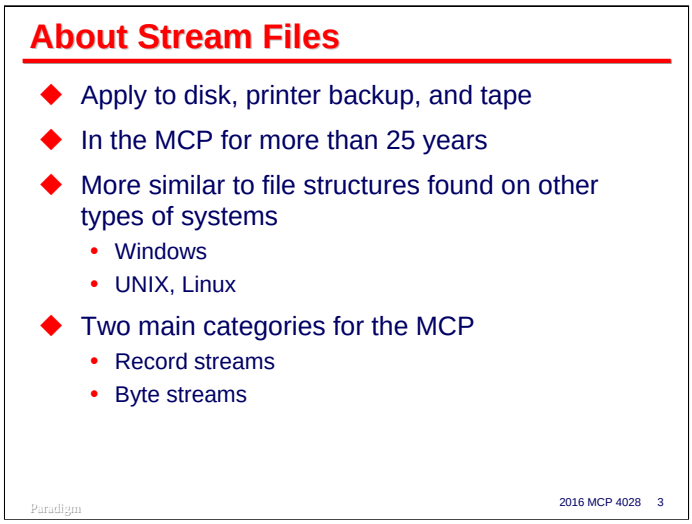



This presentation discusses stream files and their use under the Unisys ClearPath MCP.

We'll talk about how stream files differ from traditional MCP files, and the restrictions that apply to streams that do not apply to other kinds of files. We'll also talk about the file attributes that are relevant to stream files and how you program for streams.

Stream files are interesting and generally applicable to MCP applications, but there is a special case – *byte stream files* – that are the most interesting and useful. We'll spend most of the time taking about the byte stream version, including how you program specifically for byte streams, utilities for them in the MCP, and file transfer mechanisms.

Finally, we'll briefly examine some examples of Algol and COBOL programs that use byte stream files, and point out areas of the Unisys documentation where you can get more information.




Stream files are a variation on the file structures we have traditionally used with the MCP. They have been around for more than twenty-five years, at least since Mark 3.9.

Stream methods can be applied to files on disk, printer backup, or tape. In this presentation, we'll focus on stream files for disk and, to a lesser degree, for printer backup.

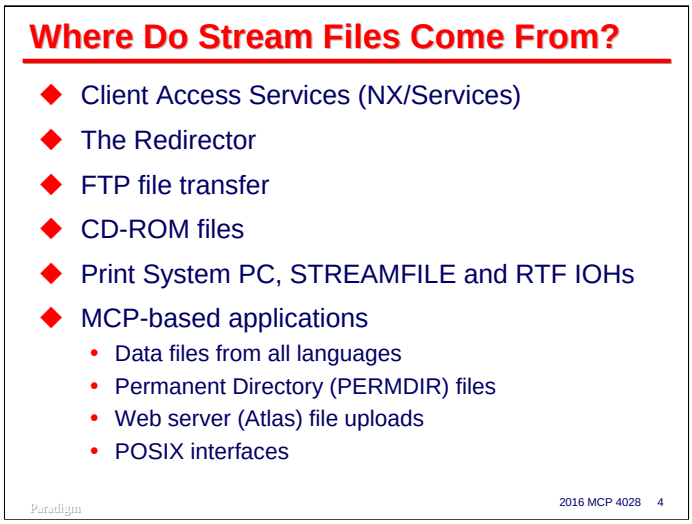
While the ClearPath MCP has a very rich I/O subsystem, the way that files are traditionally structured under the MCP differs quite a bit from the way files are typically structured on other systems. In particular, traditional MCP file structures are different from, and often incompatible with, files from Windows and UNIX systems.

Files on these other systems are often unstructured, without the formal record and block formats we are used to with the MCP. In other words, these unstructured files are stored simply as a "stream" of bytes.

The MCP supports two categories of stream files that bridge the gap between our traditional file structures and the unstructured approach common to other systems.

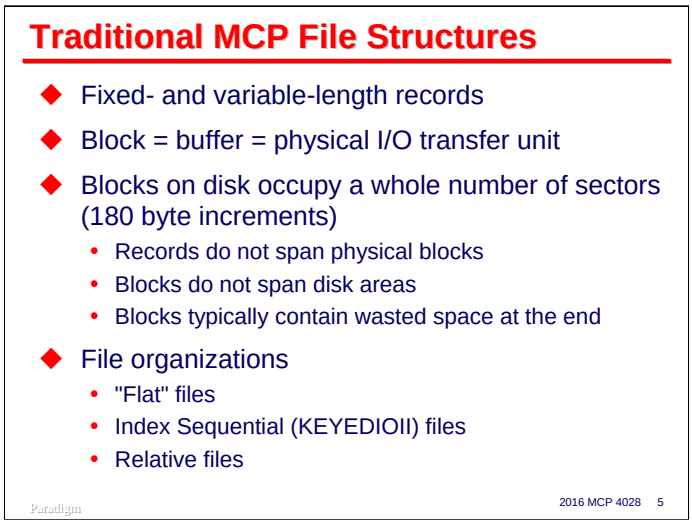
- Record streams are essentially traditional MCP files without blocks
- Byte streams are a special case of record streams, and are essentially the same as the unstructured files on other systems.

Since file interchange with other systems (especially using FTP and Client Access Services) are currently such important topics, we will talk about byte stream files in some detail.




The MCP environment can access stream files from a number of sources.

- A common source of stream files is Client Access Services. When you transfer a file to an MCP shared directory using Microsoft Networking, by default you get a byte stream file. There are also named pipes that can create either byte streams or traditional files on an MCP-based share.
- The Redirector is the complement to Client Access Services. It allows you to read and create files from an MCP application on a remote share, say on a Windows file server, or any other system that supports Common Internet File System (CIFS, port 445) or Server Message Block (SMB, port 139) protocols. Since you are directly accessing files stored under another operating system, they are not traditional MCP files, and MCP applications must access them as byte streams.
- Byte stream files can also be created by FTP. Files generated by incoming FTP transfers with **`FTPSTRUCTURE=FTPFILE`** are stored as byte streams. This includes the FTP mapping styles **`RAW`** and **`FTPDATA`**.
- Files read directly from ISO 9660 and Joliet CD-ROM media are always accessed as byte stream files.
- The Print System can both generate and print byte stream files. You can specify file attributes that will cause the Print System to generate byte stream files directly instead of the standard printer backup file format. Files generated by the **`PC`**, newer **`STREAMFILE`**, and the **`RTF`** I/O Handlers in the PRINTSUPPORT library are byte stream files. The Print System can also read stream data files generated by other systems and route them for local or remote printing.
- The Atlas web server can receive file uploads and store them in the MCP file system. Generally these files will be in the form of stream files.
- MCP-based applications in all languages can generate and read stream files. We will discuss later how this can be done using the Algol and COBOL-74/85 languages. Files produced by the POSIX interfaces (typically used with C) inherently read and write byte stream files. In addition, permanent directories themselves are considered to be stream files.



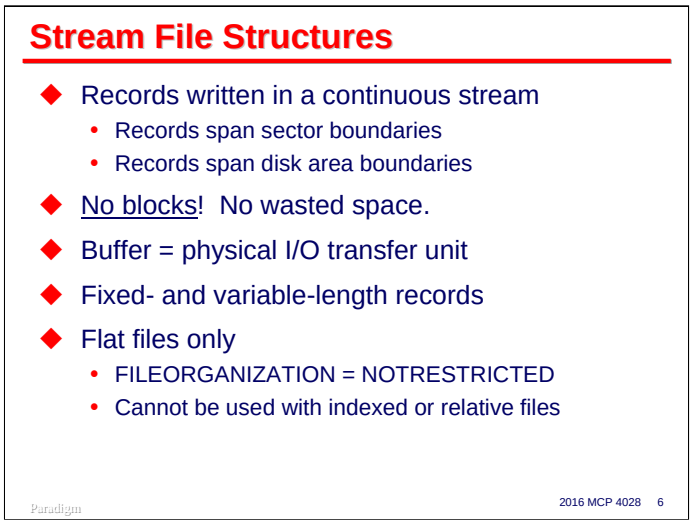

To understand how stream files relate to traditional MCP files, it's helpful to consider how traditional disk files are structured.

Traditional files support both fixed- and variable-length records of various types as determined by the **BLOCKSTRUCTURE** attribute. All traditional files share the characteristic that records are wholly contained within blocks. A block always contains some whole number of records. The size of the block determines the size of the physical I/O buffer in memory and the size of the physical I/O transfer between memory and the disk device. Thus, there is a relationship in traditional files between logical record sizes and physical I/O sizes – physical I/Os are always at least as large as a record, and at least one record is always transferred in each physical I/O.

Another characteristic of disk files is that blocks always start on a 180-byte sector boundary and occupy a whole number of contiguous sectors. The reason for this is that disk devices can only address to a sector boundary. This leads to a number of consequences:

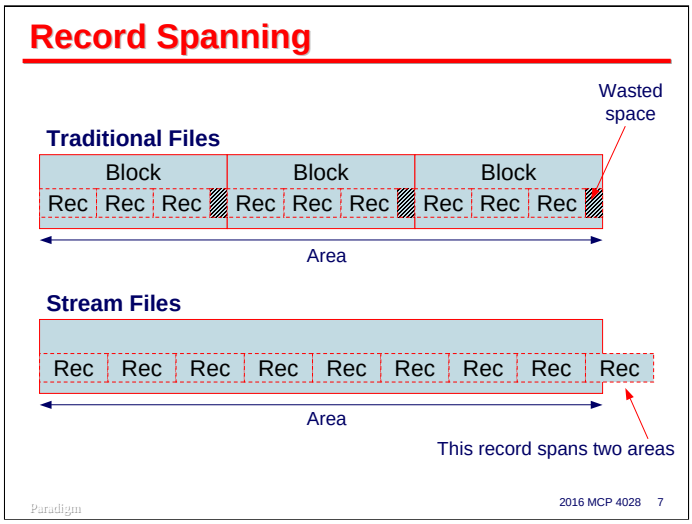
- While records may be of variable length, blocks for a given file are always of a fixed size.
- There is a resource/performance tradeoff between relatively large and relatively small blocks. Larger blocks reduce the number of I/Os (and often, dramatically reduce the program elapsed time), but require more memory while the file is open.
- Blocks must start on a disk sector boundary, but do not necessarily fill out a whole number of sectors. It is often the case that the block size is not an exact multiple of 180 bytes and there is some amount of space left over in the last sector for a block. This left-over space, or "block slop", is wasted. Good file design practices attempt to minimize this wasted space, but often cannot eliminate it.
- Because blocks are the unit of physical I/O transfer, blocks cannot span disk areas. For this reason, the MCP rounds up, if necessary, the program-specified area size for a file to the next whole multiple of sectors in a block.

In addition to the blocked behavior of disk files, the MCP supports a number of internal file layouts, as determined by the **FILEORGANIZATION** attribute. This attribute describes the internal organization of the file at a level higher than records and blocks. File organizations break down into three main categories: normal or "flat" files, index sequential files of a couple of varieties, and COBOL-74/85 relative organization files.




By contrast, stream files have a number of significant differences.

- The most significant difference between stream and traditional files is that stream files *do not have blocks!* In fact, specifying **`BLOCKSIZE`** for a stream file results in a non-fatal attribute error. Stream files completely separate the concepts of logical records and physical I/Os. Physical disk I/Os must still start on a sector boundary and cover a whole number of sectors, but logical records are not constrained to a pre-determined block size and can span area boundaries, hence *there is no wasted space*.
- In a stream file, logical records are simply written continuously across the allocated space for a file. There are no delimiter codes between records, just as with traditional files. Individual records span sector boundaries, and can even cross area boundaries. The MCP Logical I/O routines automatically fragment and reassemble records as necessary – application programs simply perform **`READ`**s and **`WRITE`**s the same way they do with traditional files.
- Although stream files do not support the concept of blocks, there is still a memory buffer area and a physical I/O transfer process between memory and the disk device. A logical file still has I/O buffers associated with its File Information Block (FIB), but in the default case, the MCP determines the size of these.
- Stream files support all of the fixed- and variable-length formats except **`BLOCKSTRUCTURE=LINKED`**.
- The primary restriction on stream files is that they are for flat file organizations only. **`NOTRESTRICTED`** and **`RELATIVE`** are the only values of **`FILEORGANIZATION`** allowed.



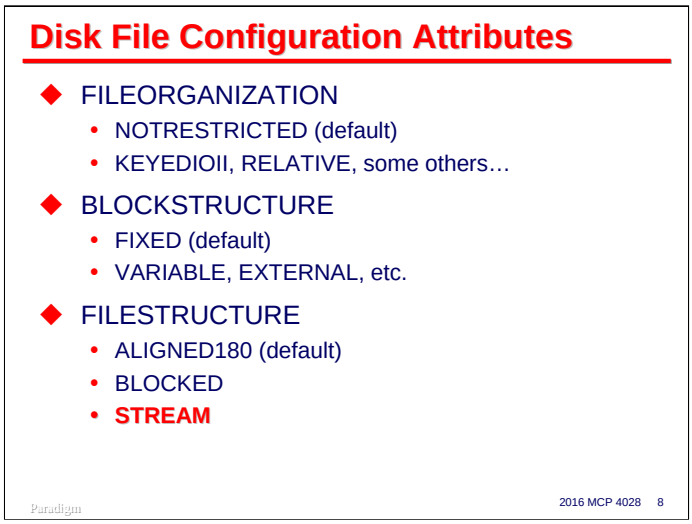

This diagram shows the difference in record and block structures for traditional and stream files.

With traditional files, there is no record spanning. Records fit entirely within blocks, blocks fit entirely within a whole number of sectors and within the space allocated for a disk area. If the block size is not an exact multiple of 180 bytes, there is wasted space on disk at the end of each block.

With stream files, there are no blocks, and the records are simply stored contiguously in the disk area. If the last record in an area does not fit, it is split. The part of the record that will fit in the area is written there; the rest of the record is written to the next area. Records and fragments of records are assembled in memory buffers and written to disk in 180-byte multiples. These buffers are typically a few thousand words long. You can control the size of these with the **`BUFFERSIZE`** attribute, but this sizing is normally left to the MCP.

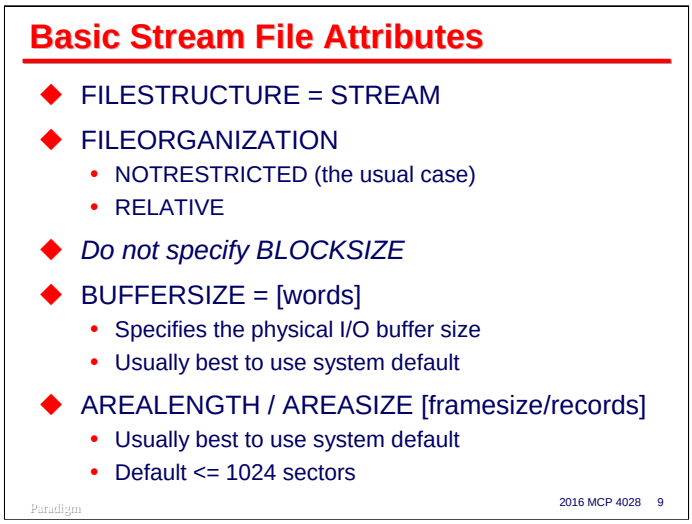
Unlike traditional files, which always read or write whole records with each physical I/O, stream files may perform more than one physical I/O to read or write one logical record.

This example shows fixed-length records, but the behavior is exactly the same with variable-length records for both traditional files and streams.



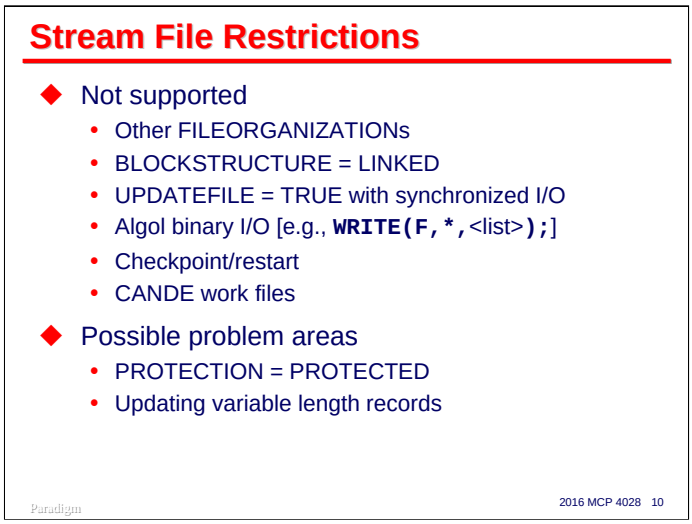

Three file attributes control the general format and nature of disk files: **FILEORGANIZATION**, **BLOCKSTRUCTURE**, and **FILESTRUCTURE**.

- **FILEORGANIZATION**, as previously mentioned, describes the high-level organization of data within a file. There are three main classes of organization: normal or "flat" files, indexed sequential files, and relative files.
- **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, variable length (and which of several varieties), or externally determined from the physical file medium.
- **FILESTRUCTURE** determines how the data is physically laid out on the disk. This attribute has three values:
  - **ALIGNED180**. This 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, sectors are 180 bytes in size, and all blocks occupy a whole number of sectors. 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**. This is similar to **ALIGNED180**, but has some 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 buffers and physical I/Os may be in multiples of the block size. **FILEORGANIZATION** must be **NOTRESTRICTED**. Reblocking is not allowed.
  - **STREAM**. *This is the attribute that makes a file a stream file.* As discussed previously, 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, using the **ANYSIZEIO** attribute, is supported.




Several attributes deserve special consideration when using stream files.

- The **`FILESTRUCTURE`** attribute must have a value of **`STREAM`**. Since the default is **`ALIGNED180`**, this must be specified explicitly for the file.
- **`FILEORGANIZATION`** may be **`NOTRESTRICTED`** or **`RELATIVE`**. **`NOTRESTRICTED`** is the default, and for most applications, is the only value of interest.
- **`BLOCKSIZE`** may not be specified for a stream file. Attempting to do so causes a non-fatal attribute error at run time.
- **`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 a **`FILESTRUCTURE`** of **`BLOCKED`** or **`STREAM`**. The Unisys documentation recommends that you normally let the MCP determine the buffer size automatically, but some programs may benefit from larger or smaller values. 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 will be between 2,000 and 5,000 words, depending on the amount of memory configured for your system. For details, 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*.
- **`AREALENGTH`** or **`AREASIZE`**. These attributes determine the physical size of disk areas, as for traditional files. **`AREALENGTH`** is specified in terms of **`FRAMESIZE`** units, while **`AREASIZE`** is specified in terms of records. The Unisys documentation recommends that, for stream files, the MCP determine this value. The MCP default will be the largest area size that is a multiple of **`MAXRECSIZE`** but not larger than 1,024 sectors. However, for very large files, you will probably want to override that default. For stream files, you will also usually want to set **`FLEXIBLE=TRUE`**. Note that **`AREASIZE`** is not meaningful for stream files if **`BLOCKSTRUCTURE`** is other than **`FIXED`**.

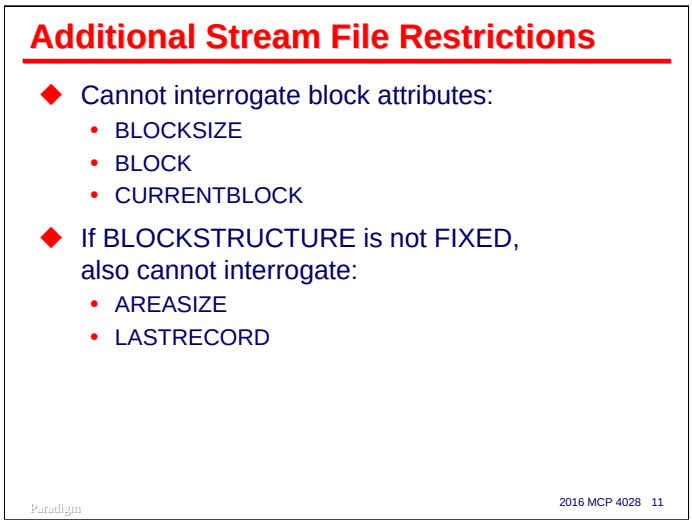



There are some restrictions on other file attributes used with stream files and how such files can be used.

- **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.
- Stream files *can* 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** will preserve the end-of-file position for open files across a system restart. However, for stream files, the EOF will be recovered to the end of the last sector written, which will typically be after the end of the last record written.
- If you rewrite a variable-length record and the size of the record differs from its original size, the MCP will generate a record length (data size) error.

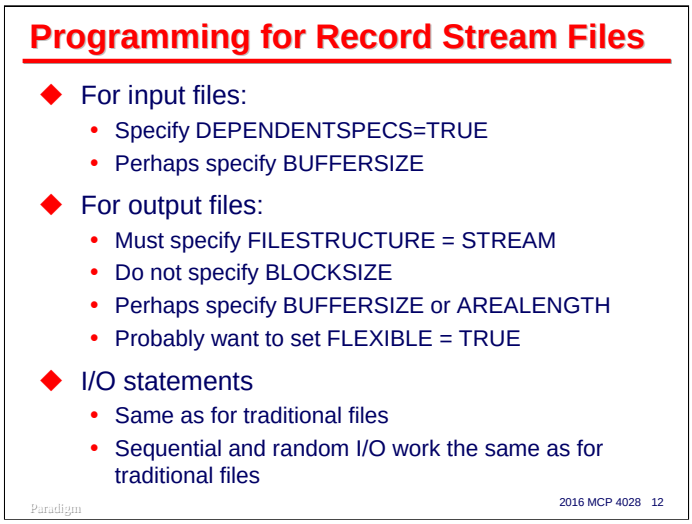



Because stream files are not blocked like traditional files, attempting to access block-oriented attributes will result in a non-fatal attribute error at run time. These attributes include

- **BLOCKSIZE**
- **BLOCK**
- **CURRENTBLOCK**

If **BLOCKSTRUCTURE** has a value other than **FIXED**, access to two additional attributes is not valid, since in that case both attributes are in units of blocks rather than records.

- **AREASIZE**
- **LASTRECORD**



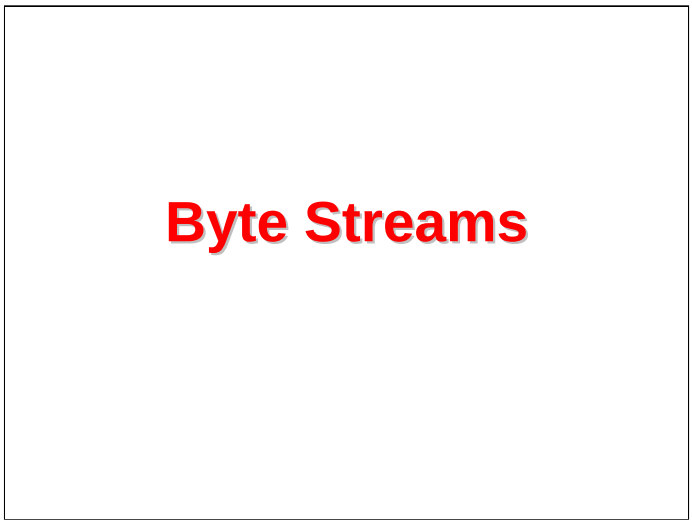

Programming for record-oriented stream files is easy. The major consideration is the value of several attributes when the file is opened.

For input files, the easiest thing to do is specify **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 control.

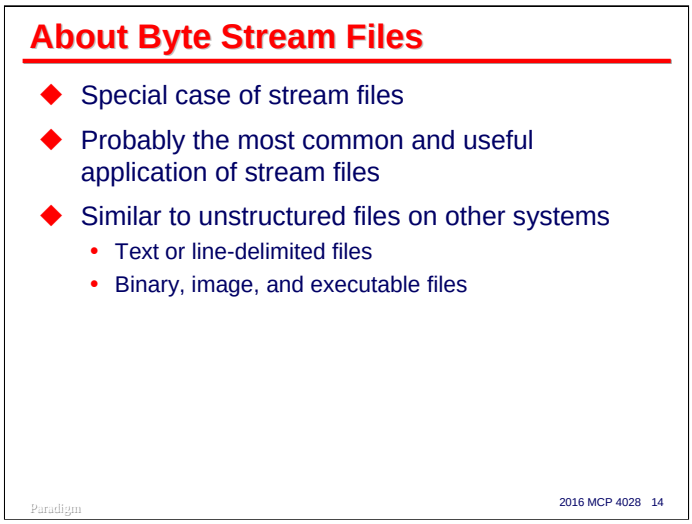
- In most cases you can simply replace the **BLOCKSIZE** specification with **FILESTRUCTURE=STREAM**.
- Depending on your specific requirements, you may want to specify the number of words for memory buffers using **BUFFERSIZE** or the length of disk areas with **AREALENGTH**. In most cases, the MCP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

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. Both random and sequential I/O are supported for stream files in the same way they are for traditional files.




All of the foregoing discussion has been about stream files in general, and 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*.

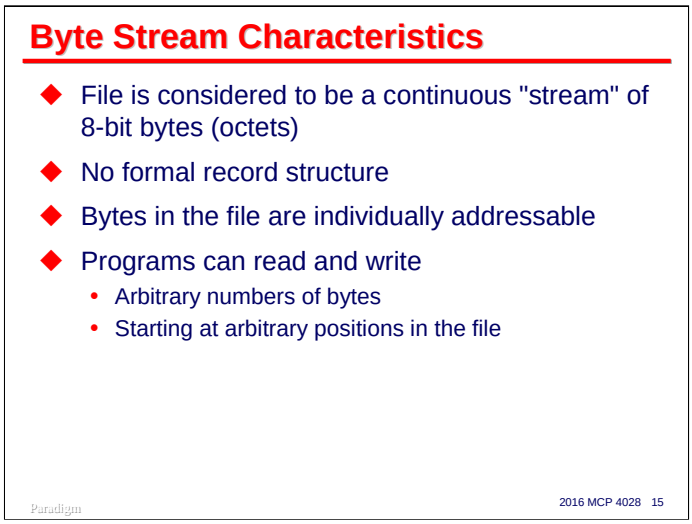



Byte streams are typically the most common and most useful application of stream file structures. They closely resemble the unstructured nature of files for other systems, and are usually compatible with them. Such files include:

- Text or line-delimited files, e.g., those produced by NotePad and other text editors.
- Binary byte streams of all kinds. These include application specific file formats, bitmaps and other image files, and executable files for other systems.

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.

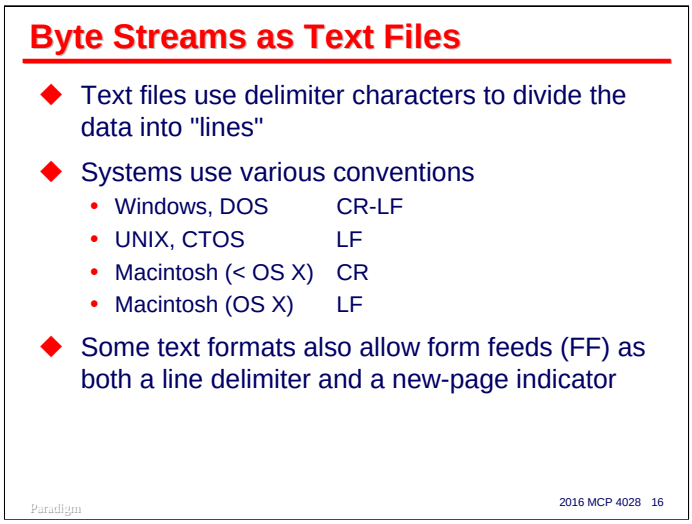
The remainder of this presentation will focus on these byte streams.




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

Another important 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 be read and written in units of disk sectors. The 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.



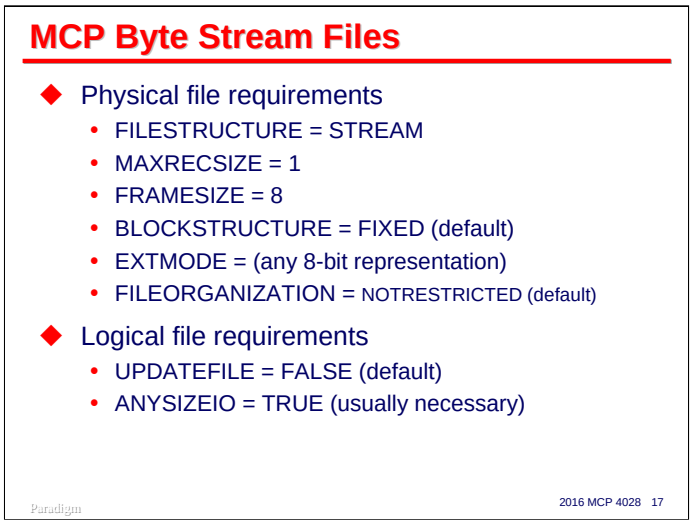

One of the most common uses of byte streams on other systems is for text, or line-delimited files. If you have ever used a text editor on another system, such as NotePad under Windows or *vi* under UNIX, you have been manipulating a text file.

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 1023 bytes.

Different systems use different delimiters to divide a file into lines. There are three common conventions:

- Microsoft Windows and DOS operating systems have traditionally used 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 will accept either a carriage-return, or a line-feed, or the CR-LF pair as a line delimiter.
- UNIX systems have traditionally used the line-feed (**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 prior to OS X use **CR** as the text delimiter.
- Apple Macintosh OS X systems are a variant of UNIX, and thus use **LF** as the line 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.




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.

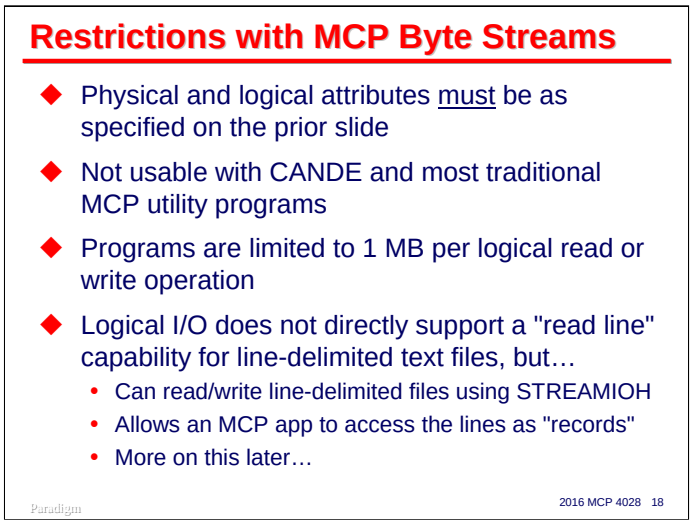
The following physical attributes are associated with byte streams. These are stored the the disk header for permanent files.

- **FILESTRUCTURE** must, of course, have a value of **STREAM**.
- **MAXRECSIZE** must have the value 1. Do not set **MINRECSIZE**.
- **FRAMESIZE** must have the value 8, indicating eight-bit character frames.
- **BLOCKSTRUCTURE** must have the value **FIXED**. This is the default for this attribute.
- **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.
- **FILEORGANIZATION** must be **NOTRESTRICTED**. This is the default for this attribute. **RELATIVE** is not permitted for byte stream files.

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.

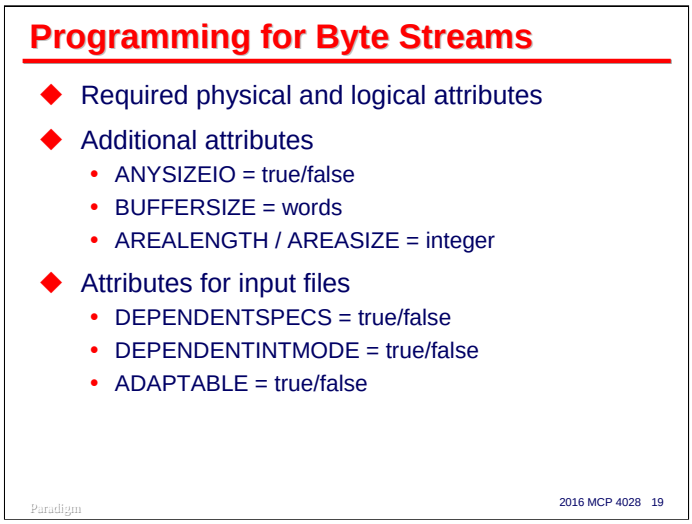
- Update I/O is not supported for byte streams, so **UPDATEFILE** must be **FALSE**. This is the default value for this attribute. You can open a byte stream in input-output mode and perform mixed reads and writes, but the sequential rewrite after read behavior implied by the **UPDATEFILE** attribute is not supported.
- When using byte stream files, you almost always want to set **ANYSIZEIO** to **TRUE**. The reason for this will become clear shortly.

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




There are a few additional restrictions on byte stream files over those for record streams.

- First, the physical attributes of the file *must* include:
  - **`FILESTRUCTURE = STREAM`**
  - **`MAXRECSIZE = 1`**
  - **`FRAMESIZE = 8`**
  - **`BLOCKSTRUCTURE = FIXED`** (default)
  - **`FILEORGANIZATION = NOTRESTRICTED`** (default)
  - **`EXTMODE`** must specify an 8-bit representation
- Like other stream files, byte streams cannot be used as CANDE files. Library/Maintenance commands and the **`PRINT`** command can be used with byte streams, however. Most system utilities do not yet support byte streams.
- 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.
- 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. Alas, it doesn't do that directly, but since MCP 7.0 (SSR 48.1) we have had something almost as good -- a virtual file I/O Handler known as **`STREAMIOH`**. This IOH can be applied to a logical file solely through file attributes. It provides a bridge between the fixed- or variable-length view most MCP applications have of files and the line-delimited nature of a text file. **`STREAMIOH`** has a number of options for specifying delimiter characters, tab expansion, trimming, etc. We'll discuss this feature a little later in the session. Of course, you can always read and write a byte stream file as just a sequence of bytes and parse the lines yourself, and in some applications that my be the best way to do it.




To open a byte stream file, it must satisfy at least the minimum physical attributes

- **`FILESTRUCTURE = STREAM`**
- **`MAXRECSIZE = 1`**
- **`FRAMESIZE = 8`**.

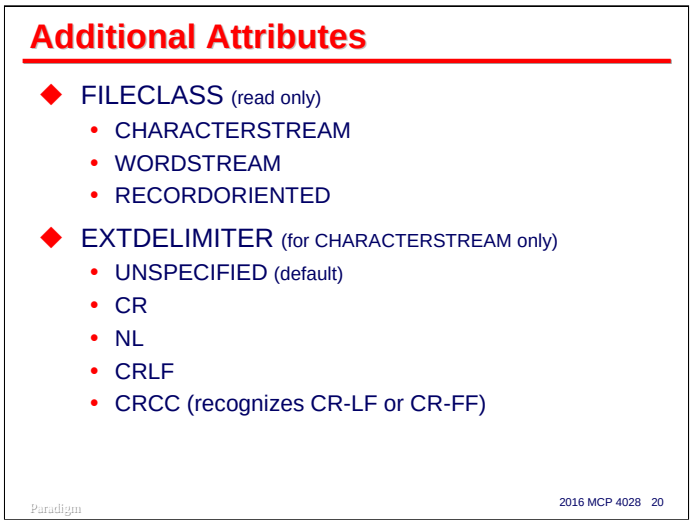
You may also need to specify the following attributes, but their default settings are compatible with byte streams:

- **`EXTMODE =`** (any 8-bit character representation)
- **`BLOCKSTRUCTURE = FIXED`**
- **`FILEORGANIZATION = NOTRESTRICTED`**
- **`UPDATEFILE = FALSE`**.

Once again, you almost always want to specify **`ANYSIZEIO=TRUE`** with byte stream files, for reasons we will discuss next.

You may also want to specify **`BUFFERSIZE`** or **`AREALENGTH`**, but the MCP defaults for these attributes are usually adequate.

**`DEPENDENTSPECS`**, **`DEPENDENTINTMODE`**, and **`ADAPTABLE`** are often useful when opening byte streams for input. We will discuss these attributes shortly.



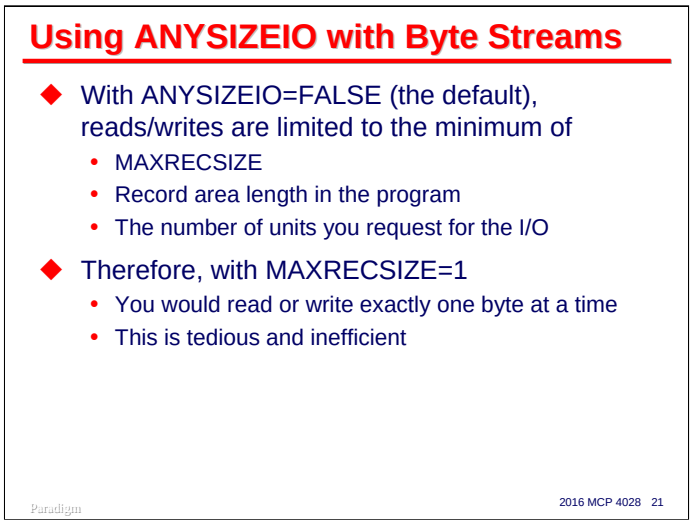

There are two additional attributes which are specifically applicable to byte stream files.

- **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 **FILECLASS=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 will automatically set **EXTDELIMITER** to **CRCC** when a file with the following characteristics is created:
  - **KIND = PRINTER**
  - **BACKUPKIND = DISK**
  - **FILESTRUCTURE = STREAM**

  In addition, the MCP will place the actual delimiter characters in the file, creating a delimited byte stream file.

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



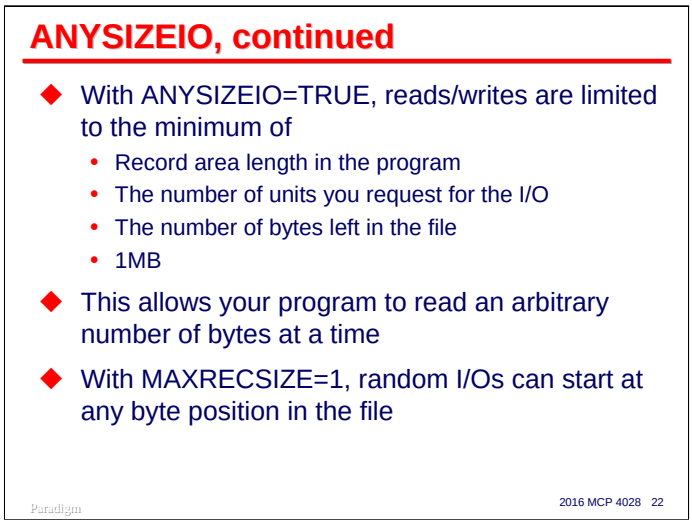

I have mentioned that you will almost always want to set the **ANYSIZEIO** attribute to **TRUE** when programming for byte stream files. Here's why.

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. Note that in COBOL programs, this value is normally the same as the size of the largest record declared for the file. We'll see shortly how to control this dynamically. In Algol programs using array-row I/O, you specify this value directly in the second parameter of a read or write statement.

Since by definition byte stream files have **MAXRECSIZE=1**, this default situation will only allow 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 for byte stream files.




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:

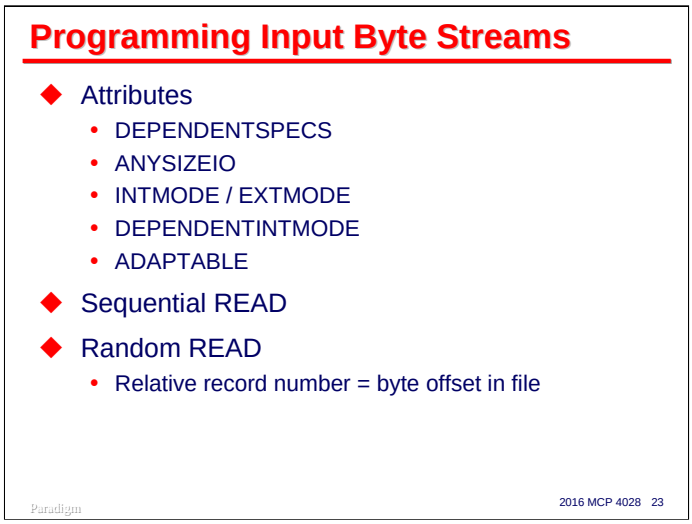
- The physical length of the record area in your program
- The number of frames you request to be transferred in the read or write statement
- The number of frames left in the file
- 1MB.

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

Therefore, setting **ANYSIZEIO=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 at arbitrary locations in the file, just as for byte streams on other systems.

While **ANYSIZEIO** is particularly useful for byte stream files, it can be used for any file with **FILESTRUCTURE=STREAM**, **BLOCKSTRUCTURE=FIXED**, **FILEORGANIZATION=NOTRESTRICTED**, and **UPDATEFILE=FALSE**. If these restrictions are not adhered to, you will get 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 last record will be skipped, since the next I/O will start at the next record boundary.




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

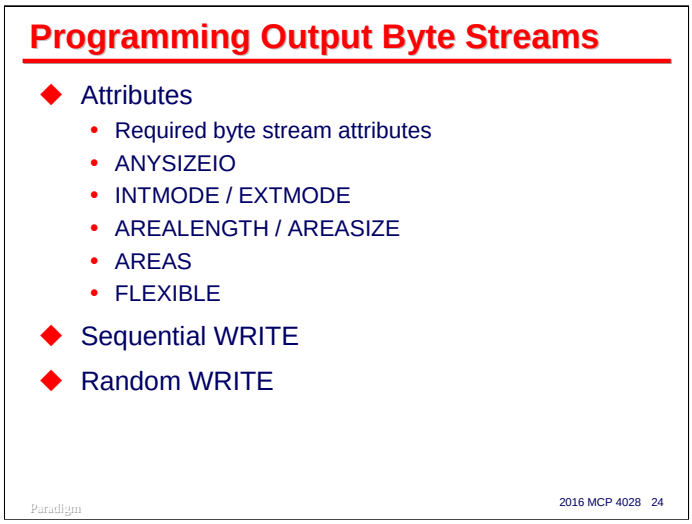
You can use the **INTMODE** and **EXTMODE** attributes to control character translation. Most MCP applications are written to process **EBCDIC** internally. **DEPENDENTSPECS** sets **EXTMODE** from the physical file attributes, but does not affect the **INTMODE** setting, which defaults to **EBCDIC**. 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 **DEPENDENTINTMODE=TRUE**, you force the MCP to automatically set the **INTMODE** for the logical file to the **EXTMODE** of the physical file. This will suppress 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=TRUE** and try to open a file that does not meet the restrictions for **ANYSIZEIO**, you will get an open error. By setting **ADAPTABLE=TRUE**, you tell 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, **ANYSIZEIO** is set to **FALSE**. This allows the open to succeed, after which your program can interrogate the 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 we will see shortly.



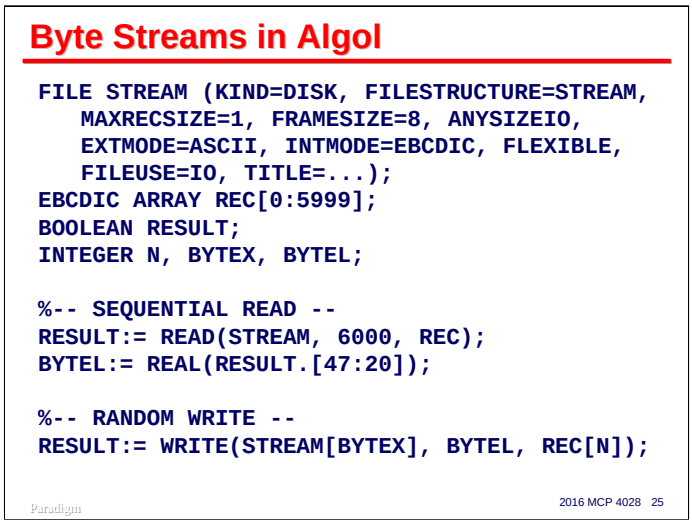

When creating a byte stream file, you must explicitly set the **FILESTRUCTURE** attribute to **STREAM**, since default attribute settings result in a traditional MCP file. You should also explicitly specify **MAXRECSIZE=1** and **FRAMESIZE=8** as attributes for the file, even if you are programming in 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** to **EBCDIC** (the default) and **EXTMODE** to **ASCII** will allow your program to write **EBCDIC** text, but the text will be translated and physically stored in **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 **AREALENGTH**, you will also probably 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 – as with reads, keep in mind that "record number" means "byte offset into the file".



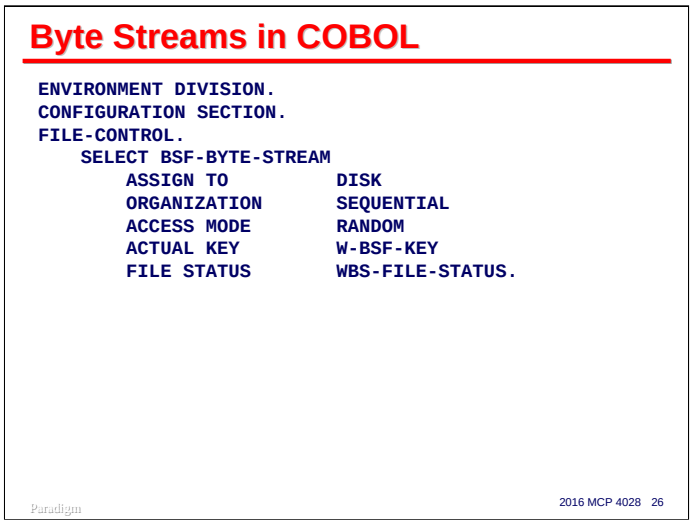

Access to byte streams is Algol is very straightforward. You specify the attributes you need directly in the file declaration or through file attribute assignments in WFL, CANDE, or MARC.

Sequential reads and writes are done in the same way as for traditional files. With **ANYSIZEIO=TRUE**, the record length parameter always specifies the number of 8-bit frames to read or write in the file.

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:

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

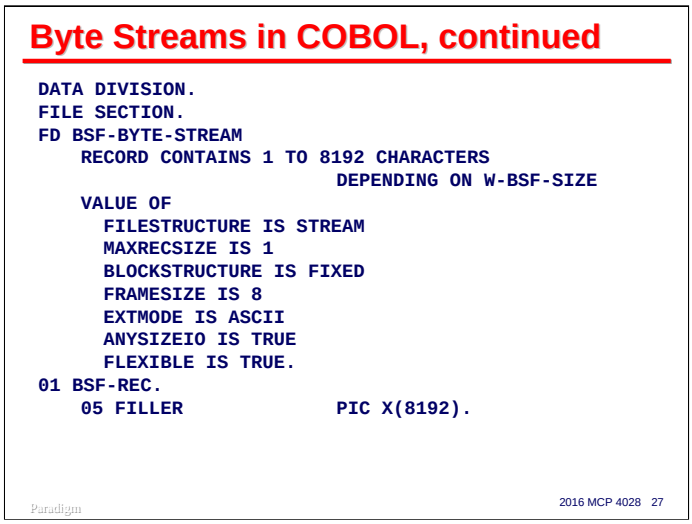
To perform random I/O, you specify an arithmetic expression in square brackets after the file identifier. For byte stream files, this will be the *zero-relative* byte offset into the file where the read or write will begin transferring data. Algol also allows you to transfer data into the middle of the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




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

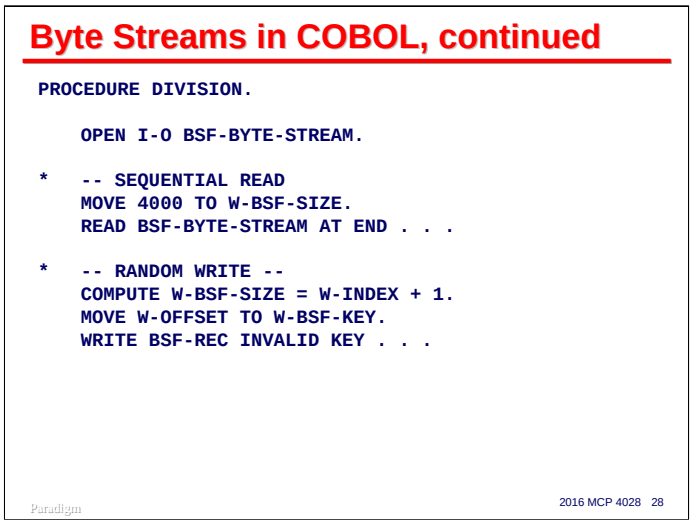
In the **SELECT** statement, the file should have **ORGANIZATION SEQUENTIAL** (the default) and an **ACCESS MODE** of either **SEQUENTIAL** (again, the default) or **RANDOM**. If you will be doing random I/O or using **SEEK** statements, you need to include an **ACTUAL KEY** clause referencing a numeric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

The **FILE STATUS** clause is not required, but it is a useful and convenient way to get detailed I/O results in a COBOL program. The data name references a two-character field.




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 particular way:

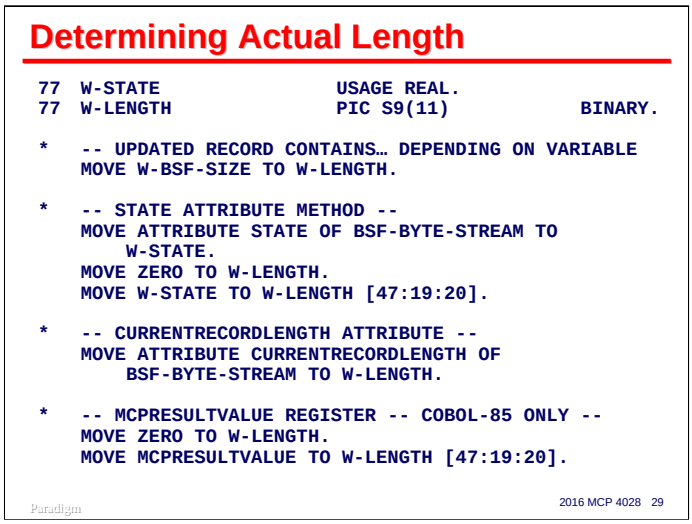
- Specify a **RECORD CONTAINS** clause with a range of **1 TO** *n*, where *n* is the maximum size of the string you need to read or write. The clause must also have a **DEPENDING ON** phrase, naming a numeric data item in **WORKING-STORAGE**. The value of this item will control the number of frames (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**.
- Specify the stream-related attributes using **VALUE OF** clauses in the **FD**. Alternatively, you can specify these externally using file attribute assignment. When reading existing files, you can usually specify just **DEPENDENTSPECS=TRUE** and **ANYSIZEIO=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** clause. COBOL will allocate the record area in your program based on the longest record you define for the file.




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** clause for the file.

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.




One thing to keep in mind when working with 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, 3000 bytes, but perhaps there were only 1255 left in the file before the EOF. The MCP will not read beyond the current end of file, and will only transfer data to your program up to that point.

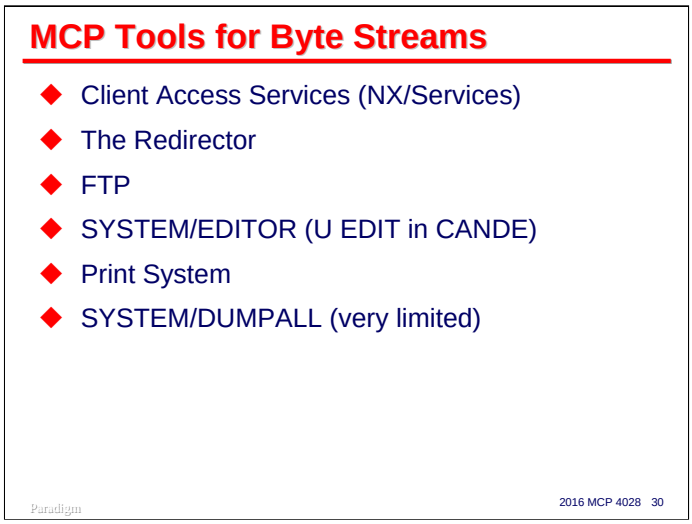
There are several ways to determine how may bytes were actually transferred between the file buffer and your program's record area.

- If you declared the file with a **RECORDS CONTAINS** … **DEPENDING ON** clause, the data item referenced by the clause will be 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 on the slide 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 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 each time. While this is an 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, an equivalent mechanism 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:

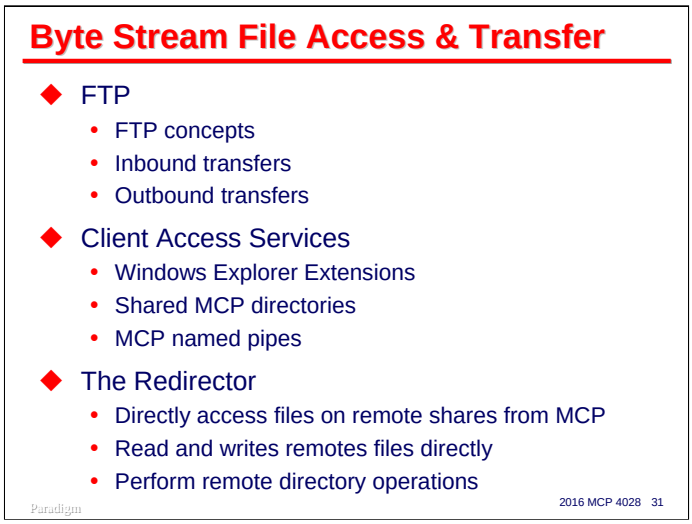
```
MOVE ZERO TO W-LENGTH.
MOVE MCPRESULTVALUE TO W-LENGTH [47:19:20].
```




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 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.
- MCP applications can use the Redirector to access files on external Microsoft Networking shares. These files appear as byte streams in the MCP environment.
- The File Transfer Protocol (FTP) component of TCP/IP can transfer byte stream files to and from MCP systems. FTP also has the ability to map both incoming byte streams to traditional MCP file formats and outgoing traditional files to byte streams.
- **SYSTEM/EDITOR** can read, edit, and convert byte stream files.
- 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.
- **SYSTEM/DUMPALL** can read and write byte stream files, but the functionality is limited. **DUMPALL** is driven largely by file attributes. Since **MAXRECSIZE=1** for byte stream files, normally listing such a file with **DUMPALL** results in a tabulation of one-character records, which is not very useful. Similarly, copying byte streams with **DUMPALL** is by default not very efficient, since each byte in the file is treated as a separate logical record. Since MCP 6.0, however, **DUMPALL** has supported a **STREAM** qualifier in many of its commands that can be used with byte stream files. It specifies how big a chunk to read and treat as a logical "record." **DUMPALL** will automatically use **ANYSIZEIO** when reading a byte stream file with the **STREAM** qualifier, e.g.,

```
RUN *SYSTEM/DUMPALL(
    "LIST ""XFER/UPLOAD.TXT"" ON PACK STREAM 100");
```

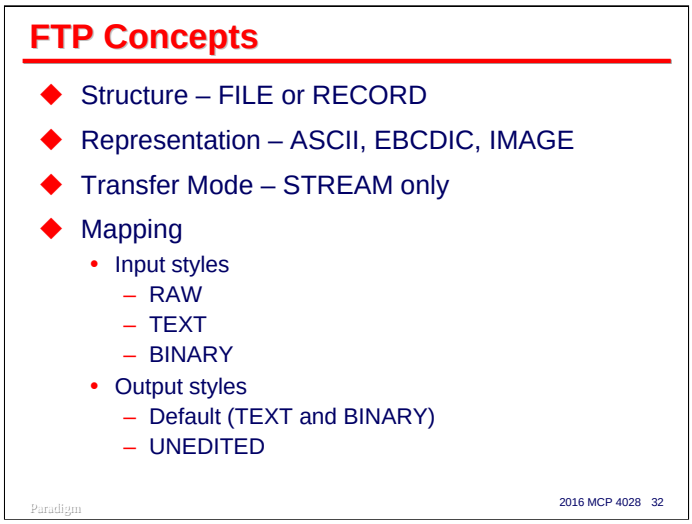



There are three principal methods of transferring or accessing byte stream files to and from the MCP.

The FTP component of TCP/IP can transfer files into and out of the MCP environment. These transfers can be initiated either from the MCP environment or from a remote system.

Client Access Services can be used both to transfer files into and out of the MCP environment, and to access MCP files directly from a remote system through a directory share. Both mechanisms are initiated from the remote system. You can use either Windows Explorer (with or without the Unisys Extensions add-in) or a number of types of named pipes to access the MCP shares.

Finally, the Redirector can be used to access files on remote system shared directories from MCP applications. Through the Redirector API (a virtual file I/O Handler), you can directly read and write files in a remote shared directory. You can also perform directory operations on the remote directory and its subdirectories, including reading the directory.

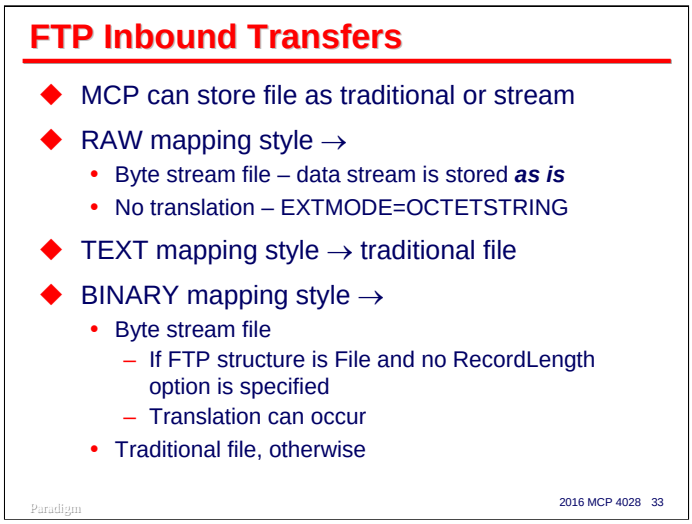



FTP can be used to transfer files between the MCP environment and virtually any other system which supports a minimal implementation of FTP. Since the capabilities of file systems vary widely among systems, FTP accomplishes this nearly universal interchange by reducing the files to a simple set of common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

FTP characterizes file transfers in three ways:

- Structure – the file can be transferred as a single entity or a record at a time.
- Representation – the data transferred can be encoded in **`ASCII`** or **`EBCDIC`**, or it can be unencoded. In the unencoded, or **`IMAGE`** form, the data is considered simply to be a stream of 8-bit bytes (octets).
- Transfer Mode – the FTP standard defines this as a third characteristic, but the MCP currently supports only one mode, Stream.

To convert the standard FTP parameters 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 byte stream files from other systems to either traditional or byte stream formats under the MCP.




When you transfer files to the MCP environment, 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**.

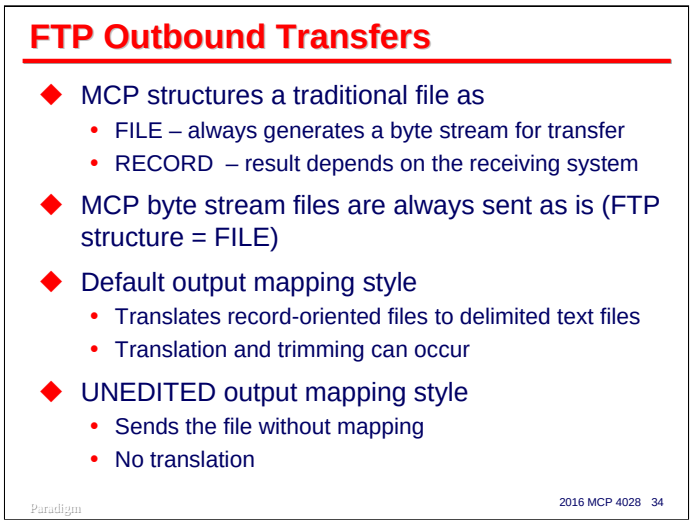
The **TEXT** mapping style will always store a traditional MCP file.

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

- If the Structure for the FTP transfer is specified as File and no RecordLength option was specified in the mapping parameters, you will get a byte stream file. With this type of inbound transfer, character translation can occur.
- If the Structure is Record or a RecordLength parameter is specified, FTP will store 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.

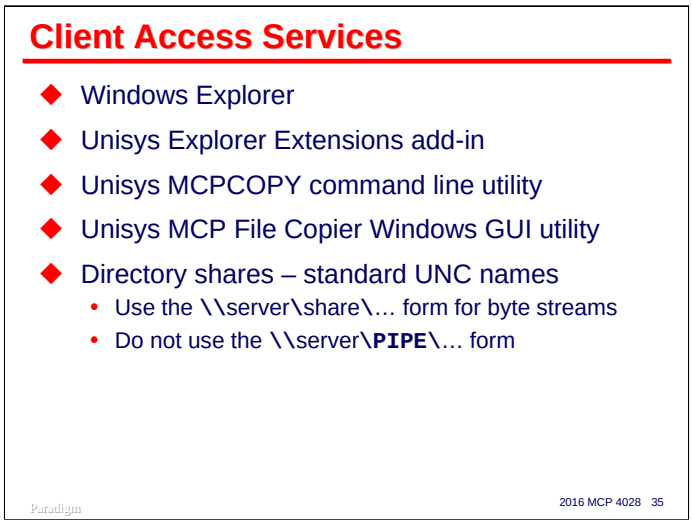



For outbound transfers, FTP can send the file with either the **FILE** or **RECORD** structure. If you are starting with an MCP byte stream file, it will always be sent with a structure of **FILE**, since the file is not record oriented.

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.




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 will 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 named pipes. 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. 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.

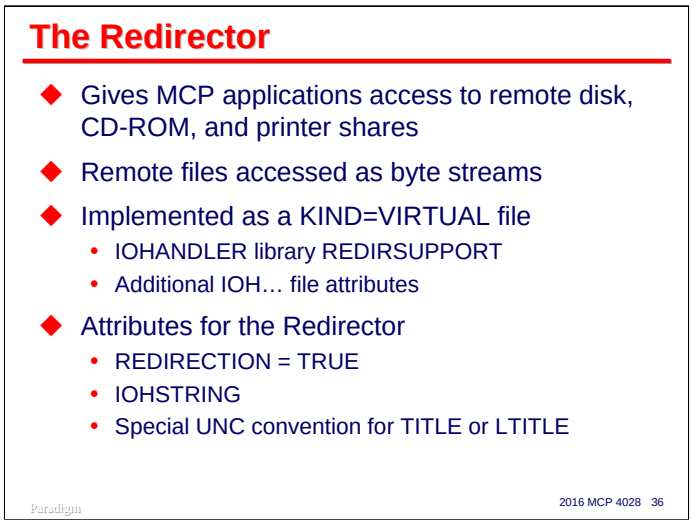
The Unisys MCP File Copier utility is a Windows GUI version of MCPCOPY. Both utilities are on the MCP Installs share.

There are several Windows APIs and tools that support reading and writing to disk shares as well as Windows-resident files, including the **TYPE** and **MORE** commands, and NotePad. To access shares using these tools, use the Universal Naming Convention (UNC) form of file name,

**\\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 start with **\\server\PIPE\…** These pipes either convert between Windows byte streams and MCP traditional files or connect to port files in MCP application. Do not use these if you want to transfer byte stream files to the MCP and have them stored as byte streams.




The Redirector, first introduced in MCP 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. Using the Redirector you can directly read and write files on these shares – no separate file transfer process is necessary. You program these files in your MCP application as byte stream files.

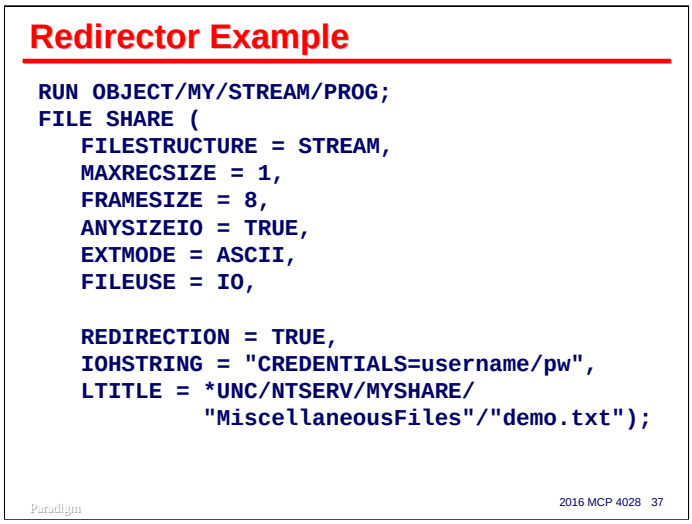
The Redirector is implemented as an instance of a new kind of file, **`KIND=VIRTUAL`**. The semantics of virtual files are not implemented within the Logical I/O subsystem of the MCP. Instead, the semantics are implemented by a library program, called an IOHANDLER (IOH), that Logical I/O calls in response 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 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`**
- A specially formatted **`TITLE`** or **`LTITLE`** string that represents the Uniform Naming Convention (UNC) name for the file on the remote system.

The Redirector is also documented in the *I/O Subsystem Programming Guide*.




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.

This slide 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** is used instead of **TITLE** because one of the nodes of the name contains more than 17 characters.

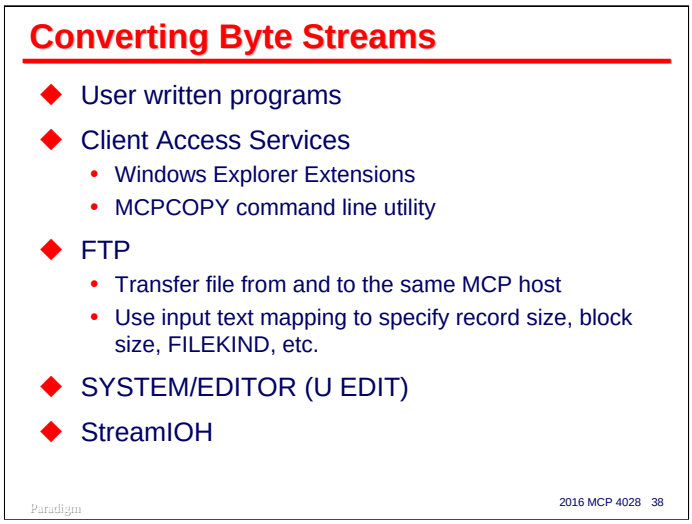
The ***UNC** prefix on the file name is a special node recognized by the Redirector. It indicates, in place of the standard double backslash (\\), that the rest of the name is in UNC format. Note that forward slashes are used instead of backslashes in these MCP title strings.

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:

- **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 (seconds) for remote server connections to complete
- **SMBTRACE** – invokes SMB diagnostics (true/false).

Instead of using the ***UNC** form of **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 path after the share name.

Since specifying **CREDENTIALS** in the **IOHSTRING** potentially exposes passwords, the Redirector can use credentials files created by ***SYSTEM/NXSERVICES/MAKECREDENTIALS**. This utility will create 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 user domain name. The file is stored under your usercode as **NXSERVICES/CREDENTIALS/***hostname*. The Redirector will automatically access this credentials file under your usercode if **CREDENTIALS** is not specified in the **IOHSTRING**.




Sometimes you have a byte stream file in the MCP environment and need to process it using tools that only support 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.

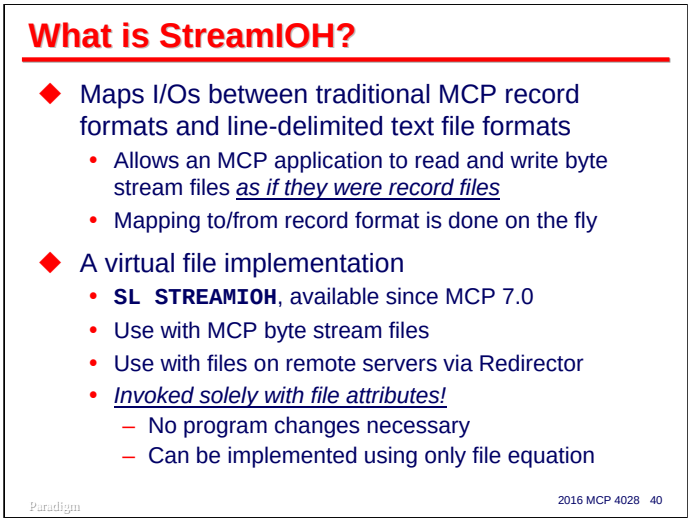
By writing your own program, you can accomplish any kind of conversion you want, and the process will generally be quite efficient in the MCP environment. The problem with this approach, of course, is the time and effort you need to spend designing and coding an appropriate program.

There are also a number of ways to convert byte streams 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 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 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 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.
- 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 next slide shows an example using FTP to convert files locally.
- **`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 a shell program (**`U EDIT`** <file name>).
- Finally, the **`STREAMIOH`** virtual file I/O Handler provides a convenient and efficient way to read and write byte stream files directly in MCP applications as if they were traditional files.

## Introduction to STREAMIOH

A bridge between
MCP record-oriented files
and line-delimited text files



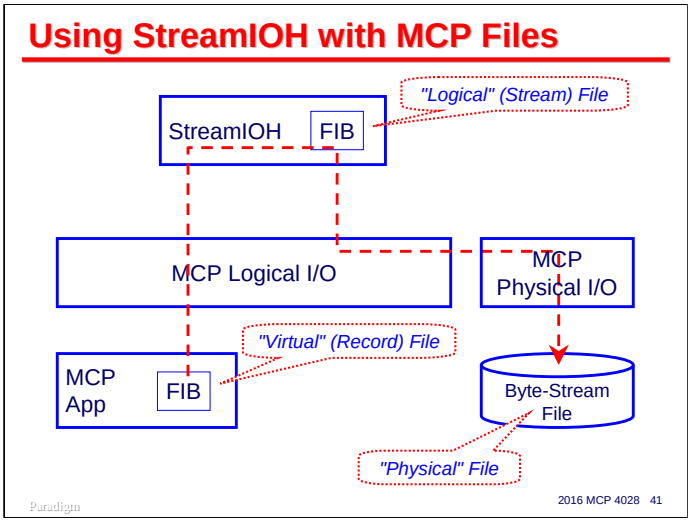

**StreamIOH** is a facility that can map logical I/Os between the record-oriented formats we traditionally use with MCP applications and line-delimited text file formats typically used on other systems. It allows MCP applications to read and write line-delimited text files as if they were record files. This conversion is generally done on the fly, as records are being read and written. With one exception that we will discuss later, the file is not converted en masse between line-delimited and record format in a separate pass.

Like the Redirector, StreamIOH is virtual file implementation. By default, it is accessed through the SL function name **`STREAMIOH`**. It has been available since MCP 7.0 (SSR 48.1), ca. 2002, but its implementation was incomplete until MCP 9.0. It is bundled in the standard system software release.

You can use StreamIOH in two ways:

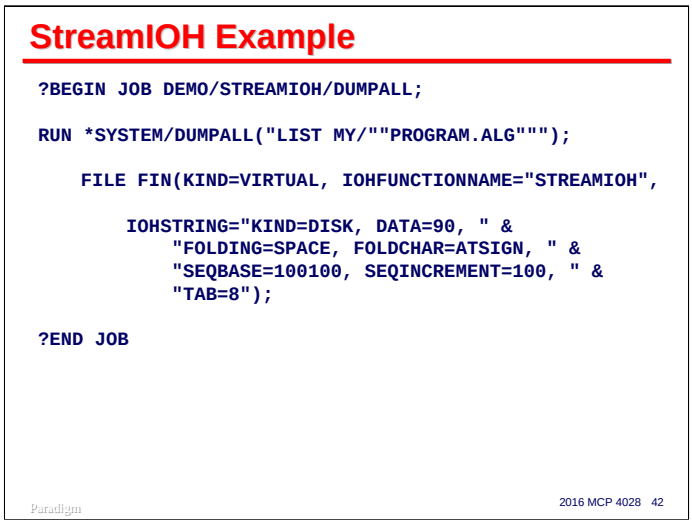
- To read and write line-delimited byte stream files within the MCP file system.
- In conjunction with the Redirector, to read and write line-delimited byte stream files on a network share.

One of the really nice things about StreamIOH is that can be invoked solely with file attributes. This means that you can give an existing program the ability to read and write variable-length, line-delimited text files without changing the program or even recompiling it. StreamIOH can be implemented simply using file equation.




This diagram shows how StreamIOH works when accessing byte stream files within the MCP file system. Since it is implemented as a virtual file, MCP applications access it through a standard file declaration and I/O verbs in the programming language. The data structure that represents a file at run time is called a File Information Block, or FIB. The MCP's Logical I/O module connects to the StreamIOH library to process the MCP application's I/O requests. The result of these requests will be viewed by the application as full records.

The StreamIOH library dynamically creates another FIB to access the physical byte stream file. The library reads or writes chunks of stream data for the physical file as necessary, and translates between the line-delimited text format of the physical file and the record-oriented format that the MCP application sees.



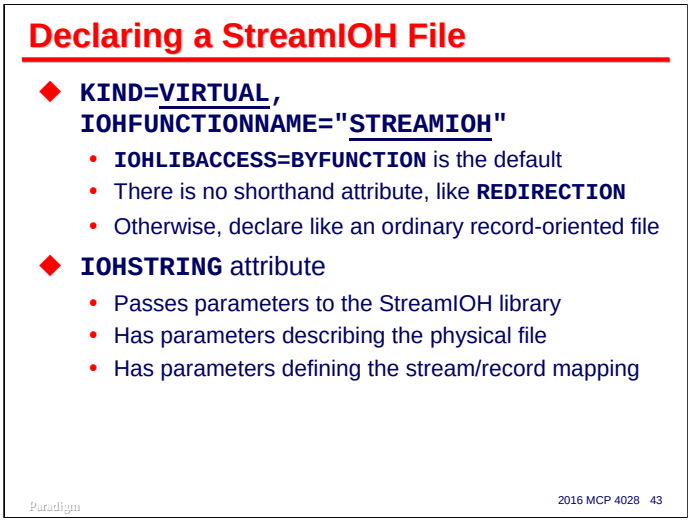

Remember when we talked about the **STREAM** keyword for **SYSTEM/DUMPALL** a few slides ago? This slide shows how to really read a line-delimited text file using **DUMPALL**.

Note that the **DUMPALL** parameter is written as if **MY/PROGRAM.ALG** is a record-oriented file, because a record-oriented file is what **DUMPALL** really sees. StreamIOH converts the lines of the text file to fixed-length logical records and presents those fixed-length records to **DUMPALL**.

Also notice that all of this is done using only file attributes:

- **FIN** is the internal name for **DUMPALL's** input file declaration.
- **KIND=VIRTUAL** indicates the logical file will be a virtual file.
- **IOHFUNCTIONNAME="STREAMIOH"** designates the function name of the library that will provide the IOHANDLER functionality for the virtual file. This name is defined in the MCP's SL list.
- **IOHSTRING** is a string-valued attribute that supplies parameters to the IOH:
  - **KIND=DISK** specifies that the text file will reside on disk in the MCP file system.
  - **DATA=80** specifies the length of the logical record **DUMPALL** will see in **FRAMESIZE** units (8-bit bytes in this case).
  - **FOLDING=SPACE** specifies that if the text line is too long for the fixed-length logical record, it will be split at a space and converted to two or more records.
  - **FOLDHCAR=ATSIGN** specifies that if the record is split, an "@" at the end of the record will indicate where it was split.
  - **SEQBASE=100100** specifies that the IOH is to supply sequence numbers and what the starting sequence number will be. Since the disk file has an "**.ALG**" extension, the logical records will be formatted as **ALGOLSYMBOL** with sequence numbers in columns 73-80.
  - **SEQINCREMENT=100** specifies the sequence number increment.
  - **TAB=8** specifies that any embedded horizontal tab characters will be replaced with spaces to the next column that is a multiple of eight positions.

These parameters will be discussed in more detail on the following slides.



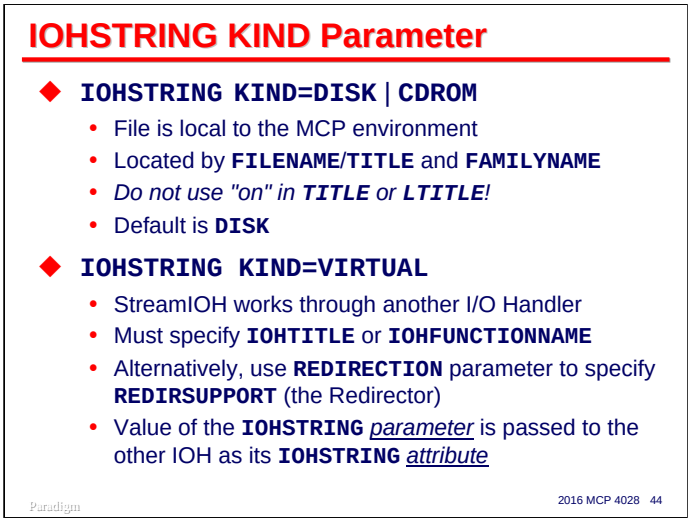

To use StreamIOH with a file, you need to declare or file-equate **KIND=VIRTUAL** and **IOHFUNCTIONNAME="STREAMIOH"**. **IOHLIBACCESS=BYFUNCTION** is the default, so this does not need to be specified. Alas, there is no convenient shorthand attribute like **REDIRECTION** as there is for the Redirector.

In all other respects, though, you declare and use the file declaration in your program as you would for an ordinary record-oriented file.

As with the Redirector, the **IOHSTRING** attribute is used to pass parameters to the StreamIOH library. There are two types of parameters used with StreamIOH:

- Parameters describing the physical file
- Parameters defining how the mapping between line-delimited and record-oriented formats is to be done.

Both types of parameters may be intermixed and specified in any order. They are written as a sequence of name=value pairs, separated by commas. Additional spaces may appear around the commas.

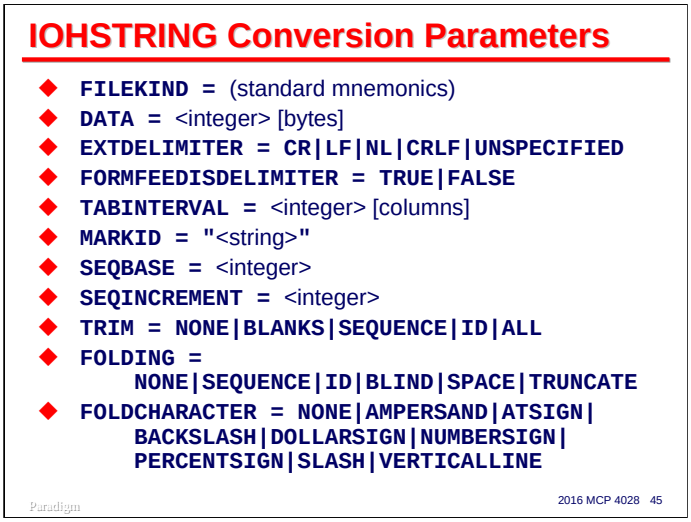



The **KIND** parameter of the StreamIOH **IOHSTRING** attribute determines where and how StreamIOH will be reading or writing the line-delimited text data.

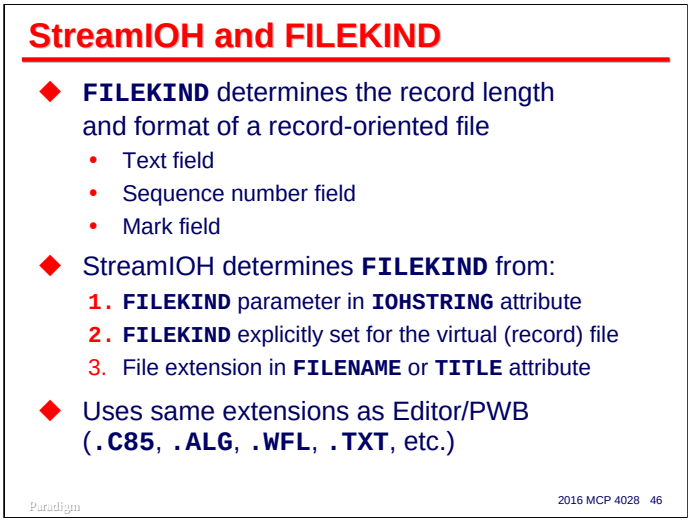
Setting the **KIND=DISK** (the default) or **KIND=CDROM** parameters means that StreamIOH will be accessing a physical file within the MCP file system. That physical file will be located by the file name attribute specified for the virtual file. Note that the family on which the file resides must be specified using the **FILENAME** attribute. Do not use an on-part when specifying the file name using **TITLE** or **LTITLE**, as that implicitly changes the **KIND** of the virtual file to **DISK**, thus making it no longer a virtual file.

Setting the **KIND=VIRTUAL** parameter means that StreamIOH will not be accessing a physical file, at least not directly. Instead it will be accessing the line-delimited data through another virtual file I/O Handler. This could be an IOH that you write, or it could be the Redirector.

- **IOHTITLE** can be used to specify the other IOH by its codefile title.
- **IOHFUNCTIONNAME** can be used to specify the IOH by its SL function name. You should specify the IOH library either by title or by function name, not both.
- **IOHSTRING** can be used to pass parameter information to the other I/O Handler's **IOHSTRING** file attribute. Note that this is an **IOHSTRING** *parameter* embedded inside an **IOHSTRING** *attribute*. This can look a little weird when you write it, but as we will see later, it enables some very useful behavior.
- **REDIRECTION** specified as a parameter is equivalent to the **REDIRECTION** file attribute. Specifying this is a shorthand way of specifying the parameters **KIND=VIRTUAL** and **IOHFUNCTIONNAME="REDIRSUPPORT"**.

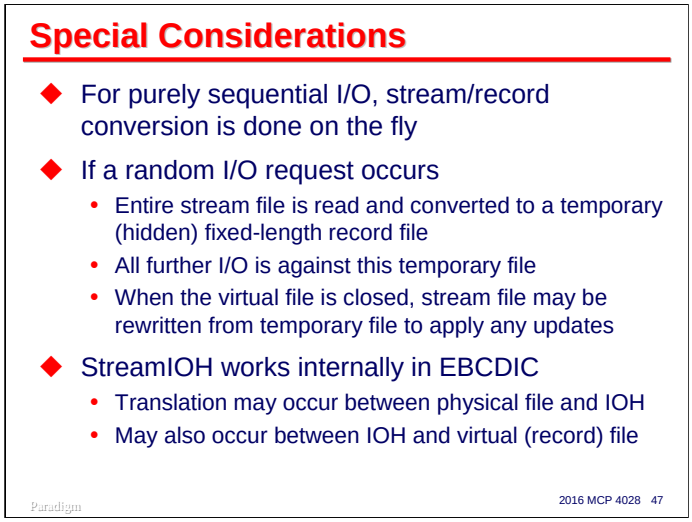



The rest of the parameters in a StreamIOH **IOHSTRING** are used to control the conversion between line-delimited and record-oriented formats. These fall logically into a few groups, which I will discuss on the next set of slides.




The **FILEKIND** file attribute specifies the internal layout of records. One of its main uses is to determine the type of compiler associated with a source or object code file. For source files, the **FILEKIND** determines the size and position of the text, sequence number, and mark ID fields in a record.

StreamIOH determines the **FILEKIND** of the virtual file from three places, in the order of descending priority as shown on the slide. If **FILEKIND** has not been specified in some other way, and the file name of the physical file contains an extension (**.TXT**, **.DAT**, etc.), then the extension determines the **FILEKIND**. StreamIOH uses the same extension codes as the Editor utility. It also recognizes the extensions used by Programmer's Workbench (PWB, NX/Edit), which all end in "**_m**".

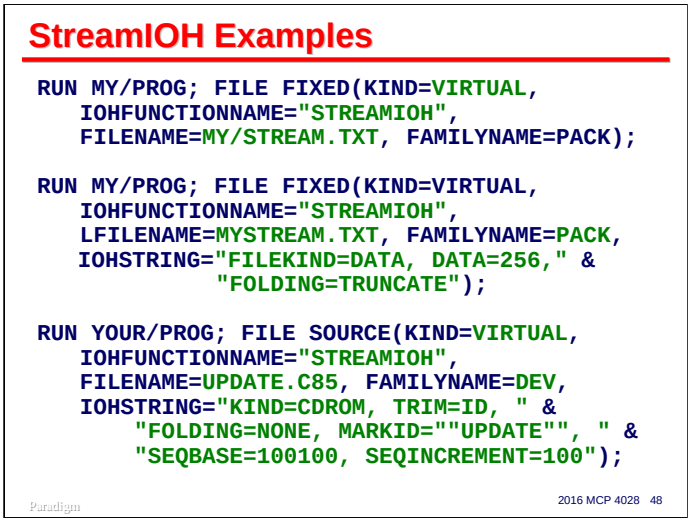



There are a couple of special considerations when using StreamIOH.

First, we have been talking thus far about using StreamIOH for sequential access to a line-delimited byte stream file. When used in this fashion, all conversion between the line-delimited and record-oriented format is done on the fly as reads and writes are issued by the MCP application.

StreamIOH also supports random I/O against the line-delimited file. The problem with random I/O is that there is no way to predict where lines in the line-delimited file occur, since they can be variable length. To handle this, the first random I/O operation causes StreamIOH to read the entire stream file, converting it into a temporary, hidden, fixed-length record file. All further I/O for the virtual file will occur against this temporary file. When the virtual file is closed, the temporary file is discarded. If writes occurred to the temporary file, it is used to recreate the line-delimited file before being discarded.

The second consideration is that StreamIOH always works internally using the EBCDIC character code. Depending on the **`INTMODE`** of the virtual file, translation may occur between the MCP application's record area and StreamIOH. Depending on the **`EXTMODE`** of the physical file, translation may also occur between StreamIOH and the physical file. In most cases involving line-delimited text files, however, this translation something you want to have happening.

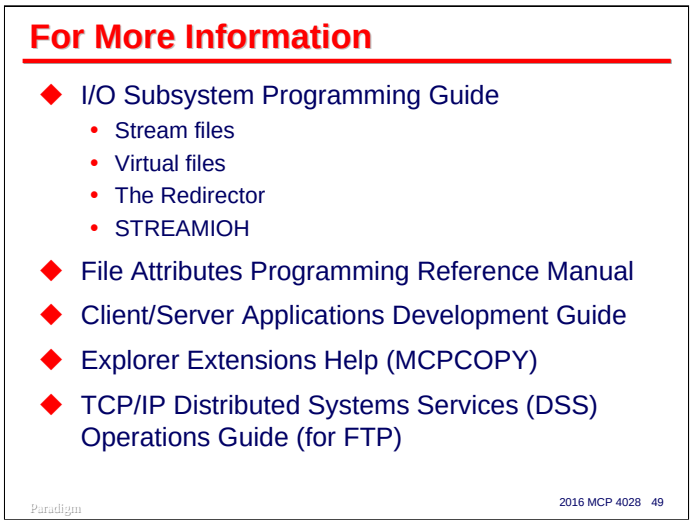



This slide shows three more examples of file attribute equations that invoke StreamIOH.

In the first example, the line-delimited file is converted using StreamIOH defaults. This usually works well for files coming from Windows systems.

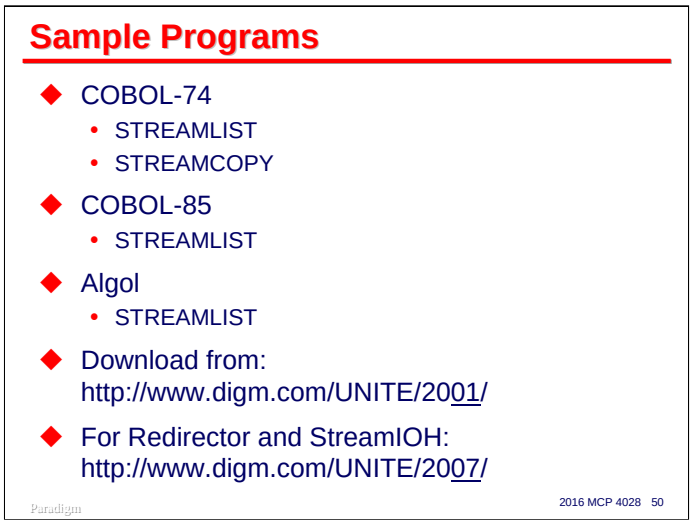
In the second example, the line-delimited file is considered to be just textual data, and is converted to a fixed-length, 256 character record area.

In the third example, a line-delimited file is read from CDROM and converted to COBOL-85 record format (as determined by the extension on the file name). Any mark ID from the line-delimited file is discarded and replaced by the literal "**UPDATE**". The records are resequenced 100100+100 as they are read. Since **FOLDING=NONE**, if lines longer than the COBOL-85 record length are read, a **DATAERROR** is returned to the MCP application. The **TRIM=ID** parameter is redundant in this example, as the **MARKID** parameter causes any mark ID field read from the line-delimited file to be replaced according to the parameter value supplied.




A number of Unisys documents contain information relevant to byte stream files in the MCP environment. All of these references are relative to the MCP 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 through 3 for general information on byte stream files and Section 12 for a discussion of Virtual files, the Redirector, and StreamIOH.
- 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 need 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 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.




For the original version of this presentation, I write a few example programs that illustrate the basic techniques for reading and writing sequential byte stream files.

There two COBOL-74 examples. 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 will 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 copy of this presentation in PDF format, from our web site.:

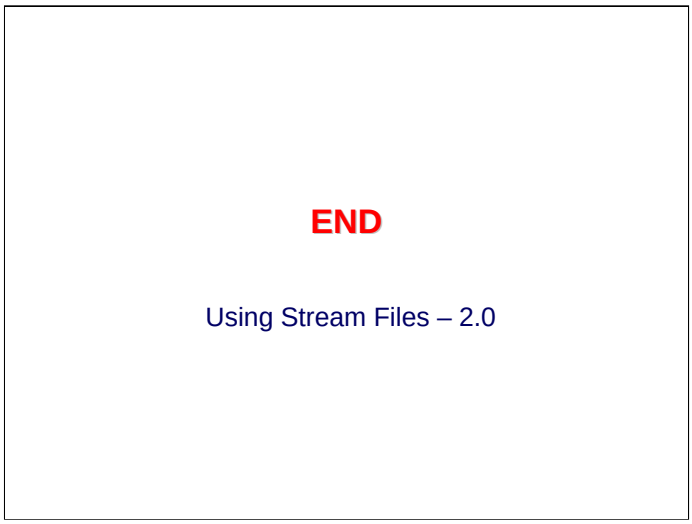
**`http://www.digm.com/UNITE/2016/`**

For the source of stream file samples, see:

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

For the source of Redirector and StreamIOH samples, see:

**`http://www.digm.com/UNITE/2007/`**


END
Using Stream Files – 2.0