Open File Access

1. Introduction

This note explains the bewildering variety of Open File Access techniques that are available in GSM and GX (and GSMWIN32.EXE).

Throughout this document the terms "desktop" and "server" are used as follows:

Desktop	The file or application is being accessed or executed on the PC that is running a thin-client (i.e. GSMWIN32.EXE or GX.EXE)
Server	The file or application is being accessed or executed on the PC that is running a fat-client (i.e. GLOBAL.EXE)

Although most of the techniques described here are available in **both** GSM (Windows) and GSM (Unix) this document concentrates on the GSM (Windows) configuration and only makes references to GSM (Unix) when there are significant differences between the GSM (Windows) and GSM (Unix) run-time environments.

The following Access Methods are fully described in this document:

Description	**Section**
SVC-61	2
Direct Windows/Unix File Access Method (OR$98)	3.1
Direct Unix File Access Method (OR$97)	3.2
Open Text (TFAM) Access Method (OR$83O)	3.3
Line Based Open Text Access Method (OR$83L)	3.4
GX Open Access Method (OR$98X)	4.1
GX Write-Only Text Access Method (OR$83G)	4.2

The following related sub-routines are described briefly in this document:

Equivalent of OPEN$ for Windows/Unix folder (NOPEN$)	5.1
Equivalent of LIST$ for Windows/Unix folder (NLIST$)	5.2
Equivalent of CLOSE$ for Windows/Unix folder (NCLOS$)	5.3
Extended version of NOPEN$ (NEOPN$)	5.4
Extended version of NLIST$ (NELIS$)	5.5
Extended version of NCLOS$ (NECLS$)	5.6
Equivalent of RENA$ for Windows/Unix file (RENAT$)	5.7
Rename Windows/Unix file directly (RENAX$)	5.8
Equivalent of DELE$ for Windows/Unix file (DENAT$)	5.9
Delete Windows/Unix file directly (DENAX$)	5.10
Equivalent of COPY$ for Windows/Unix file COPYX$	5.11
Copy files using Windows filecopy function (COPYQ$)	5.12

Important Note: This document only describes techniques that are available with the 32-bit Global Development System. Other techniques, that are available **only** in the 16-bit Cobol or 16-bit Speedbase development systems, have either been replaced by improved 32-bit techniques or are so obscure that they can be considered obsolete.

2. The SVC-61 Interface

The lowest level Open File Access routine is SVC-61. For example, SVC-61 includes functions to Open, Close, Read and Write Windows files. This multi-functional interface, which includes many other non-file-based functions, is **fully** documented for GSM (Windows) in chapter 8 of the File Converters Manual and for GSM (Unix) in chapter 7 of the File Converters Manual.

Although there is no need to describe SVC-61 any further in this document some points are worth noting:

- SVC-61 is a very "raw" interface that, in general, makes direct calls to the host operating system. SVC-61 only includes very rudimentary checking and validation for some functions to prevent fatal errors that could crash GLOBAL.EXE or glintd. Unless there is a very good reason, you are encouraged to avoid the use of direct "low level" SVC-61 calls and use the equivalent "high level" Access Method, or sub-routine, instead. Notwithstanding, there may be **some** circumstances in which only an SVC-61 interface can be used (e.g. if the required technique does not exist as a sub-routine);

- No attempt has been made to provide a common SVC-61 interface for GSM (Windows) and GSM (Unix). For example, the control blocks, function codes and parameters for an SVC-61 function on GSM (Windows) are completely different for the same function on GSM (Unix). This incompatibility reflects the differences between the Windows and Unix operating systems. We have absolutely no plans to rationalise these differences. If you are writing code that includes SVC-61 calls and that code is expected to operate on both GSM (Windows) and GSM (Unix) platforms then you must be prepared to write separate interfaces for the 2 platforms;

- SVC-61 is currently a Server-only interface. For example, although SVC-61 can be used to open a Windows file on the computer running GLOBAL.EXE it cannot be used to open a Windows file on the computer running GX.EXE. In general, a reduced set of specialised interfaces are available for file access on the GX PC. Although, we have **plans** to extend some SVC-61 functions to invoke operations on GX, no release dates are available at the time of writing;

- By default, SVC-61 functions are "blocking" calls i.e. all users running from the GLOBAL.EXE must wait for the Windows or Unix operation to complete before they can resume processing. On GSM (Windows) configurations **only**, some SVC-61 functions can be replaced by asynchronous, non-blocking calls that are invoked by using the SVC-88 interface. SVC-88 is fully described in section 8.1.2 of the File Converters 8.2 Manual.

3. Server-based Open File Access Methods

This section describes the various Access Methods that are currently available to access Windows/Unix files on the Server (i.e. the computer or network that is hosting GLOBAL.EXE or glintd).

3.1 Direct Windows/Unix Access Method (OR$98)

The Direct Windows/Unix Access Method allows direct, random-access of Windows and Unix files. This access method is the "open file" equivalent of the Global file **BDAM (Basic Direct Access Method)** as described in chapter 6 of the Global File Management Manual (note that from now on the Global File Management Manual will be abbreviated to GFMM).

Note that the 32-bit Direct Windows/Unix Access Method replaced a plethora of operating-system specific and paged/non-paged equivalent 16-bit Access Methods.

The file accessed is treated as a string of bytes numbered sequentially from zero upwards. Records are accessed by supplying the starting byte number as the key, together with the number of bytes to be read or written. The format of the records is entirely under program control since you determine the length and starting byte of each record.

The attributes of a file, such as its unit-id, volume-id and file-id, specified in the file definition (FD) coded in the data division has no meaning but must be specified to satisfy the compiler syntax. It is recommended that the file-id is set to a descriptive variable name, the unit-id to "?" and the volume-id should not be set at all.

You must specify the name of the area that contains the record length. You can also define a key area to contain the starting byte number for use in random access READ and WRITE statements.

The following PROCEDURE DIVISION statements are provided to enable a file to be processed:

OPEN	which must be executed prior to any other statement affecting the file;
READ	to read a record at random;
READ FIRST	to read the very first record in the file;
READ LAST	to read the very last record in the file;
READ NEXT	to read the next record during sequential processing;
READ PRIOR	to read the previous record during sequential processing;
WRITE	to update an existing record at random;
WRITE NEXT	to write a record during sequential processing;
CLOSE	to terminate processing of a file.

When any of the statements are executed a file condition, returned by exception condition 2, may arise as explained in section 1.3.1 of the GFMM. In addition, if OPTION ERROR or ON ERROR is specified in the FD, an exception condition will be generated should an irrecoverable I/O error occur, as explained in section 1.6 of the GFMM. Therefore it is usual to follow each file processing statement with an ON EXCEPTION statement. If you do not, and an exception condition arises, your program will be terminated in error.

A write next pointer is not maintained for a direct file, and you may use WRITE NEXT, WRITE, READ FIRST, READ LAST, READ NEXT, READ PRIOR and READ statements to access information anywhere within the allocated file extent. However, should any of these statements attempt to read or write a record which is partially or wholly outside the file extent, the file boundary violation file condition will be returned. All file I/O errors are indicated by an exception with an error code "I" in $$RES and the host operating system error number in $$CRES. Note that a value of 1,000,000 in $$CRES indicates a 32-bit addressing error.

3.1.1 DATA DIVISION Statements

If a program uses the Direct Windows/Unix Access Method the statement:

```
ORGANISATION OR$98 TYPE 0 EXTENSION 128
```

must be coded in the DATA DIVISION before the first FD or data declaration.

The file definition for a direct file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$98
[ASSIGN TO UNIT "?" FILE file-id ]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
01    FILLER REDEFINES filename
 02   FILLER              PIC X(100)
 02   PAFILE
  03  PANAME     PIC X(99)           * directory path and filename
  03  PAZERO     PIC X               * Must be set to LOW-VALUE i.e. #00
 02   FILLER              PIC X(8)
 02   PAPERM     PIC 9(4) COMP       * Unix permissions
```

The FD establishes a special group data item whose name is *filename*. The file-id, directory path and filename, keyname, length and size appear as subordinate items within this group and can, if need be, be referred to by the application program.

If you can specify the size or length before the program executes, code the parameter as a numeric string. If the size or length is not known until run-time a symbol must be coded for the parameter. This symbol will then label a level 02 item which the user program is responsible for initialising.

Because it is not possible to include a VALUE clause in a DATA DIVISION redefinition, the **zero-terminated** string containing the required pathname must be moved to the PANAME field during program execution. Furthermore, as explained in section 3.1.1.1, the PAZERO must also be set to LOW-VALUE (#00) by a MOVE statement rather than a VALUE clause.

The permissions field is only of value for Unix files and allows you to set the Unix permissions before opening a file. In addition to the Unix permissions field, which is only recognised on GSM (Unix) configurations, a Windows Access Mode field is available only on GSM (Windows) configurations - see section3.1.1.4 for further details.

The filename is coded as a descriptive symbol. It serves to label the file definition, as explained in section 1.2.1 of the GFMM.

The ORGANISATION clause must be coded as shown. It indicates that the open direct access method is to be used.

The ASSIGN statement is required in working storage and should be coded as shown, where file-id is only required as a descriptive symbol.

The KEY, RECORD LENGTH, SIZE, OPTION and ON ERROR statements may appear in any order following the ASSIGN statement. The RECORD LENGTH statement must be coded: the others are optional.

The KEY statement is only required if records are to be accessed at random. Keyname is specified as a symbol and the statement:

```
02 symbol PIC 9(9) COMP
```

is generated within the FD.

The Direct Windows/Unix Access Method maintains the keyname field to contain the byte number of the start of the record last accessed. The byte number of the very first byte of the file counts as byte number zero.

The program must place the byte number of the start of the record it requires to access in the keyname field before executing a random READ or WRITE operation.

A successful READ NEXT or WRITE NEXT operation increments the keyname field by the length of the previous record accessed and then accesses the record thus identified: if the previous operation on the file was an OPEN then the first record is accessed.

A successful READ PRIOR decrements the keyname field by the length of the record to be read, and then accesses that record.

The RECORD LENGTH statement is always required. Length is specified as a symbol and the statement:

```
02 symbol PIC 9(4) COMP
```

is generated within the FD.

The program must place the length of the record to be read in the length field before executing a read or write statement. The field is set to zero when the file is opened, and is not altered by read and write operations.

The SIZE statement is required only if you wish to know the current size of the file. The file size will be maintained during processing.

OPTION ERROR should be coded only if you wish your program to regain control following an irrecoverable I/O error. ON ERROR should be coded if you wish to handle certain I/O errors specially. The processing of these statements is common to all file organisations, and is described in section 1.6 of the GFMM.

3.1.1.1 Zero-Terminated File Name

The last byte of the 100 byte PAFILE field (i.e. PAZERO) must be a binary-zero (LOW VALUE). If this field is not LOW-VALUE a STOP 9898 will be generated. Furthermore, the file name string specified in PAFILE **must** be terminated by a byte of binary-zero (LOW-VALUE). This is a requirement of the Windows/Unix function that is invoked to open the file. If the length of the filename string is exactly 99 bytes then both conditions will be met by a single byte of LOW-VALUES.

It is vital that the last significant character of the file name is **immediately** followed by a byte of binary-zero otherwise the OPEN operation will fail. It is not sufficient to include a string of SPACES between the last significant character of the filename and the byte of binary-zero in PAZERO. For example, if the following code is used to initialise the PAFILE field, a subsequent OPEN operation will fail:

```
        77      MYFILE        PIC X(?)
                VALUE         "c:\test\myfile.xxx"
...
        MOVE MYFILE TO PANAME                       * Filename has trailing spaces
```

but should be replaced with:

```
        77      MYFILZ        PIC X(?)
                VALUE         "c:\test\myfile.xxx"
                VALUE         #00
...
        MOVE MYFILZ TO PANAME                       * LOW-VALUE after filename
```

3.1.1.2 Long File Names

The first revision of the Direct Windows/Unix Access Method did not support a Windows/Unix filename string longer than 99 characters (i.e. the size of the PANAME redefinition in the FD extension). For GSM SP-9, and later, on GSM (Windows) configurations **only**, a filename string of any size can be specified. Rather than embedding the Windows/Unix filename **within** the FD extension it can be specified in a field that is separate from the FD. This is achieved by redefining the FD as follows:

```
        01      FILLER REDEFINES filename
          02    FILLER              PIC X(100)
          02    PAHIGH        PIC X                 * 1st byte set to HIGH-VALUES
```

```
  02   PAPTR              PIC PTR                  * Pointer to filename
  02   FILLER             PIC X(95)                * Rest of 100 bytes unused
  02   FILLER             PIC X(8)
  02   PAPERM       PIC 9(4) COMP      * Unix permissions
```

where PAPTR points to a zero-terminated filename string. For example:

```
 77    MYFILZ       PIC X(?)
       VALUE        "c:\test\myfilewithaverylongname.xxx"
       VALUE        #00

       MOVE HIGH-VALUE TO PAHIGH                   * Set byte of #FF
       POINT PAPTR AT MYFILZ                       * Point to filename string
```

The indirect filename option (i.e. a single byte of HIGH-VALUE followed by a pointer to the filename string) **must** be used if the filename string is longer than 99 characters. However, this technique can also be used, as an alternative to the "embedded filename string", even if the string is less than 99 characters.

3.1.1.3 Unix File Permissions

On GSM (Unix) configurations only, the Unix File Permissions for the OPEN NEW statement can be specified explicitly by setting a non-zero value in PAPERM. If PAPERM is zero a file permissions word of 384 decimal (600 octal) will be used (i.e. equivalent to rw-------). Note that as usual, the actual file permissions are subject to the current *umask* setting.

3.1.1.4 Windows Open Mode

For GSM SP-16, and later, on GSM (Windows) configurations only, the Windows Open Mode can be specified explicitly by redefining the FD as follows:

```
 01    FILLER REDEFINES filename
  02   FILLER             PIC X(208)
  02   WINMOD       PIC X
```

The WINMOD field can be set to any of the following values:

Windows Mode value	**Meaning**
#x0	Read-only mode
#x1	Write-only mode
#x2	Read/write mode
#0x	Share compatibility mode
#1x	No shared mode
#2x	Read share mode
#3x	Write share mode
#4x	Full share compatibility mode

Note that, for backwards compatibility, a WINMOD value of #00 is mapped to #02 (Read/write mode).

3.1.2 The OPEN Statement

The OPEN statement is coded:

OPEN *type filename*

where *type* is the word NEW or OLD; *filename* identifies the direct file definition. An OPEN statement must be executed before any other operation affecting the file. If an OPEN is attempted but the FD is already open your program will be terminated in error.

OPEN NEW is used to create a new file of the specified path and filename if one does not already exist or to truncate a file that already exists to zero length. If there is no existing file, the new file will be created. For Unix files the new file will be created with Unix permissions 600 octal (i.e. RW access for the owner) unless otherwise specified by the PAPERM field. OPEN OLD obtains exclusive access to an existing file.

3.1.2.1 File Conditions

When an OPEN OLD statement is executed GSM checks to see whether a file with the same path and filename exists. If this is not the case, a file not found exception is returned.

3.1.2.2 Successful Completion

Providing no file condition or irrecoverable I/O error occurs an OPEN NEW results in an empty file of size zero being opened on with the specified path and filename.

When an OPEN OLD statement completes successfully the file size is returned in the FD and can be accessed by the user program if a SIZE statement with the symbol option was coded.

The record length field in the FD is set to zero by a successful OPEN statement.

The keyname field in the FD is always set to zero following a successful OPEN statement. This is to allow the file to be processed sequentially using a sequence of READ NEXT or WRITE NEXT statements as described below.

3.1.3 The WRITE NEXT Statement

WRITE NEXT is used to write all or part of a file sequentially. It is coded:

WRITE NEXT filename FROM A

Here filename identifies the Direct file definition and A is a simple or indexed variable. If a WRITE NEXT is attempted on an FD which is not already open the program will be terminated in error.

3.1.3.1 Establishing the Length
The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE NEXT.

If the length is not established explicitly by the program, the length of the last record accessed will be used.

3.1.3.2 Successful Completion
Provided that no file condition or irrecoverable I/O error occurs, WRITE NEXT will set the key to the offset address of the byte following the previous record accessed (if any) and transfer A to the record thus identified. The number of bytes transferred is given by the length field.

3.1.3.3 Programming Note
If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to zero when the file is opened. In this case WRITE NEXT can be used to write the entire file sequentially.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. WRITE NEXT will continue writing from the record following the one accessed by READ or WRITE.

The READ NEXT statement sets the key field to the record retrieved, so that a subsequent WRITE NEXT statement will write the following record.

The WRITE NEXT can cause the file size to be extended if needed, subject to space being available on the file system.

3.1.4 The WRITE Statement
WRITE is used to write a record at random or rewrite the last record accessed. It is coded:

```
WRITE filename FROM A
```

Here filename identifies the direct file definition and A is a simple or indexed variable. If a WRITE is attempted on an FD which is not already open, the program will be terminated in error.

3.1.4.1 Establishing the Key
If WRITE is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
        02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the WRITE statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the WRITE statement is to rewrite the record previously retrieved by READ NEXT or written by WRITE NEXT.

3.1.4.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be written (1 to 32767 bytes) in this field before executing the WRITE.

If the length is not established explicitly by the program the length of the last record accessed will be used.

3.1.4.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs, WRITE will transfer the number of bytes given by the record length from A to the record identified by the key.

3.1.4.4 Programming Note

A WRITE may extend the file size if needed subject to the available space on the file system.

3.1.5 The READ FIRST and READ LAST Statements

READ FIRST is used to read the very first record of the file. It is coded:

READ FIRST *filename* INTO A

Similarly READ LAST is used to read the very last record in the file. It is coded:

READ LAST *filename* INTO A

In both cases, *filename* identifies the direct file definition and A is a simple or indexed variable. If READ FIRST or READ LAST is attempted on an FD which is not open the program will be terminated in error.

3.1.5.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing a READ FIRST or READ LAST.

If the length is not established explicitly by the program, the length of the last record accessed will be used.

3.1.5.2 File Conditions

A file boundary violation condition will be returned if a READ FIRST or READ LAST attempts to input a record which is larger then the total file size, and which would therefore start or end outside the file extent.

3.1.5.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs READ FIRST will set the key to address the first byte of the file (a value of zero), and READ LAST will set the key to be the file extent size less the length of the record to be processed. The record thus identified will be transferred to A, the number of bytes transferred being given by the length field.

3.1.5.4 Programming Note

Use of READ FIRST is normally used to re-position at the start of the file prior to reading records sequentially using READ NEXT.

Use of READ LAST presupposes that the program has sufficient information about the file to be able to determine the length of the very last record (possibly because the length is fixed). It is important to note that it is the responsibility of the program to determine an appropriate record length for use by READ LAST, not that of the direct access method itself.

3.1.6 The READ NEXT and READ PRIOR Statements

READ NEXT and READ PRIOR are used to process part or all of a file sequentially. READ NEXT is used to read the next sequential record, and it is coded:

```
READ NEXT filename INTO A
```

READ PRIOR is used to read the previous sequential record. It is coded:

```
READ PRIOR filename INTO A
```

In both cases filename identifies the direct file definition and A is a simple or indexed variable. If a READ NEXT or READ PRIOR is attempted on an FD which is not already open the program will be terminated in error.

3.1.6.1 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ NEXT or READ PRIOR.

If the length is not established explicitly by the program the length of the last record accessed will be used.

3.1.6.2 File Conditions

A file boundary violation condition will be returned if READ NEXT or READ PRIOR attempts to input a record which is wholly or partially outside the file extent.

3.1.6.3 Successful Completion

Provided that no file condition or irrecoverable I/O error occurs, READ NEXT will set the key to the offset address of the byte following the previous record accessed (if any); and READ PRIOR will set the key to the offset of the byte that is exactly one record length before the start of the previous record accessed. The record thus identified is transferred to A. In both cases the number of bytes transferred is given by the length field.

3.1.6.4 Programming Note

If a KEY statement was not specified in the file definition the FD still contains an internal key field which is set to 0 when the file is opened. In this case READ NEXT can be used to read sequentially through the entire file; READ PRIOR at any point may be used to retrieve the preceding record; WRITE updates the last record thus retrieved; and READ rereads it. In addition READ FIRST and READ LAST may be used to position at either the start or end of the file.

When a KEY statement is specified a READ or WRITE can be used to position the file at any point. READ NEXT will continue sequential processing from the record following the one retrieved by READ or output by WRITE, and READ LAST will process sequentially from the record before it.

A WRITE NEXT statement sets the key field to the start of the record written, so that a subsequent READ NEXT statement will read the following record, and a READ PRIOR statement will read the previous record.

Note that when using READ PRIOR it is the responsibility of the program to determine the length of the record to be processed by some means (possibly because it has a fixed length), and not that of the direct access method itself.

3.1.7 The READ Statement

READ is used to retrieve a record at random or reread the last record accessed. It is coded:

```
READ filename INTO A
```

Here filename identifies the direct file definition and A is a simple or indexed variable. If READ is attempted on an FD which is not already open the program will be terminated in error.

3.1.7.1 Establishing the Key

If READ is to be used as a random access operation the FD must contain a KEY statement coded as:

```
KEY IS symbol
```

which causes:

```
        02 symbol PIC 9(9) COMP
```

to be generated. The program must then move the byte number of the start of the record it is required to retrieve to this field before executing the READ statement.

If a KEY statement is not coded in the file definition the FD still contains an internal key field which is set to zero when the file is opened and incremented by READ NEXT and WRITE NEXT. In this case the only use of the READ statement is to reread the record previously retrieved by READ NEXT or written by WRITE NEXT.

3.1.7.2 Establishing the Length

The FD must contain the RECORD LENGTH statement with the length coded as a symbol. The program must establish the length of the record to be read (1 to 32767 bytes) in this field before executing the READ.

If the length is not established explicitly by the program the length of the last record accessed will be used.

3.1.7.3 File Conditions

A file boundary violation condition will be returned if READ attempts to input a record which is wholly or partially outside the file extent.

3.1.7.4 Successful Completion

Providing no file condition or irrecoverable I/O error occurs, READ will transfer to A the number of bytes given by the record length of the record specified in the key field.

3.1.8 The READ PHYSICAL Statement

As the key of a Direct Windows/Unix file is a byte number, the READ PHYSICAL statement functions in exactly the same way as a READ.

3.1.9 The CLOSE Statement

CLOSE must be used to terminate the processing of a file. It is coded:

```
        CLOSE filename [DELETE]
```

where filename identifies the direct file definition.

3.1.9.1 Standard Processing

CLOSE always completes any outstanding I/O operations on the file and returns the FD to the status it possessed prior to being opened. Following CLOSE, the FD can be re-opened for the same (or a different) file by a subsequent OPEN NEW or OPEN OLD statement.

3.1.9.2 File Conditions

If the FD passed to the CLOSE was not open, then a file not open condition will be returned.

3.1.9.3 Deletion

If a DELETE phrase is coded all the space the file occupies is returned to file system and the file name is removed from the directory. Following a CLOSE DELETE the file no longer exists.

3.1.9.4 Programming Notes

On GSM (Unix) configurations if you fail to close a direct file then the channel will remain open and will not be closed by any other GSM process including exiting from GSM. It is therefore very important to ensure that all direct FD's are closed.

On GSM (Windows) the STOP RUN processing automatically closes all open Direct Access files. However, it is still good coding practice to close all open files before program termination.

3.2 Direct Unix Access Method (OR$97)

As its name suggests the Direct Unix Access Method is the Unix-only equivalent of the general-purpose Direct Windows/Unix Access Method. Under normal circumstances there should be no reason to use File Organisation OR$97 rather than the more general OR$98.

3.3 Open Text (TFAM) Access Method (OR$83O)

The Open Text (TFAM) Access Method allows read/write access to text files on the host operating system as well as "normal TFAM access" to Global text files. The actual interface is transparent to the calling program - the decision on what file to actually access is made by the Access Method when opening the Global file indicated. If the target file is a normal Global text file (i.e. type TF) then it will be accessed directly. If the target file is a special Schema File (i.e. type SC, organisation 116) then the host operating system file described by that Schema File will be opened.

Before attempting to use the Open Text (TFAM) Access method you should be familiar with the "traditional" (Global) Text File Access Method as described in Chapter 5 of the GFMM. In particular, you should be familiar with the TFAM Record Format and the additional fields in the FD, all of which are faithfully emulated by the Open Text (TFAM) Access Method.
The following PROCEDURE DIVISION statements are supported:

```
OPEN NEW
OPEN OLD
CLOSE
CLOSE TRUNCATE
CLOSE DELETE
WRITE NEXT
```

```
READ NEXT
READ
READ FIRST
READ PHYSICAL
```

3.3.1 DATA DIVISION Statements

If a program uses the Open Text File Access Method then the statement:

```
ORGANISATION OR$83O TYPE 3 EXTENSION 552
```

must be coded in the DATA DIVISION before the first FD or data declaration.

The additional fields in the FD are exposed by the following redefinition:

```
01    FILLER REDEFINES FDTFAM
  02  FILLER          PIC X(98)           * Allow for 32-bit FD
  02  TXLLN           PIC 9(4) COMP       * Line Length Read
  02  TXSIG           PIC 9(4) COMP       * Significant index
  02  FILLER          PIC X(4)
  02  TXNCH      PIC 9(4) COMP       * Characters in body of text
```

3.3.2 User Notes

The following important notes apply to the "Open Text File (TFAM) Access Method".

3.3.2.1 The CLOSE DELETE operation

The Global format Schema File is never deleted by the CLOSE DELETE operation. The standard DELE$ routine must be used to delete the Schema File.

3.3.2.2 Terminating Characters

Any terminating file characters in the host operating system file (i.e. #1A or #00) will not be returned as part of the file size. For GSM (Windows) configurations, the terminating #1A will be returned in the extent size.

3.3.2.3 Terminating CR,LF

Windows files must be terminated with #0D0A. The CLOSE handling will check the last two bytes of the file to ensure that this is the case. If the file does not terminate correctly then the CLOSE handling will write the terminating #0D0A to the end of the file.

However, it is generally good practice when using Open TFAM to always write the first record of the file without a leading #0D0A and to ensure that the last record of the file only contains #0D0A. This will be compatible with most host editors, and although not standard Global format, will also be acceptable as a Global text file.

Unix text files do not contain #0D characters before #0A characters and these will be stripped out when the record is written to a Unix format text file and replaced when a record is read from a Unix format text file.

3.3.2.4 The $SCHEMA utility

The $SCHEMA utility is used to create and maintain the TFAM Schema Files.

For the "Text Schema File" option, $SCHEMA prompts for a host OPEN NEW file name as well as a standard file name. If the OPEN NEWfile name differs from the standard file name, then when an Open New operation is requested on the Schema File, the file will be opened with the Open New file name rather than the standard file name. When a close is issued the file will be renamed with the standard file name. This technique allows for synchronisation with 3rd party applications that are activated only when a particular file name is present in a directory. The technique may also be useful for editor type programs.

3.3.3 Extension to Open TFAM for Longer Record Lengths

The "traditional" TFAM Access Method, and thus the, Open Text (TFAM) Access Method support a maximum line length of 256 characters. For Open Text (TFAM) this limit can be extended to 16383 characters by using the following specialised coding technique.

In order to read records with a record length of more than 256 bytes, the TFAM Schema File must be opened in the following way:

CALL FDPTR USING *fd opcode buff*

where FDPTR is a PIC PTR redefinition of the first word of the FD, that is:

```
01      FILLER REDEFINES fd
  02    FDPTR PIC PTR
```

opcode is a PIC 9(4) COMP field which is set to 1 for OPEN NEW and 2 for OPEN OLD; and *buff* is the read buffer area for TFAM, which is used as follows:

```
01      BF
  02    BFLENG     PIC 9(4) COMP          * Buffer length
  02    BFAREA     PIC X(nnnn)            * Buffer area
```

The BFLENG field must be set to the length of the BFAREA field and must be at least twice the record length supplied in the record length field. The record length field will indicate the maximum record length to be read.

The buffer block will be used by Open TFAM and must not be overwritten whilst the Open TFAM file remains open. If more than a single Open TFAM FD is to be opened at a time then each Open TFAM FD will require its own buffer block.

3.4 Line-Based Open Text Access Method (OR$83L)

The Open Text (TFAM) Access Method has been deliberately coded to emulate the traditional Text File Access Method (TFAM). Although the "traditional" TFAM interface is suitable for some applications a somewhat simpler text-file Access Method, **that does not**

require the use of Global Schema Files, has been developed. Note that this access method does **not** support Global format text files.

When reading a text line the Line-Based Open Text Access Method returns the body of the text of a line (excluding any CR LF characters). When writing a text line the Line-Based Open Text Access Method adds the appropriate combination of CR, LF to the end of the line. The decision on which type of record (i.e. Windows or Unix) to be written will be defaulted by the Access Method when opening the file depending on the host operating system. However an override flag can be set on opening the file (see below)

The following PROCEDURE DIVISION statements are provided to enable a text file to be processed:

```
OPEN NEW
OPEN OLD
CLOSE
CLOSE TRUNCATE
CLOSE DELETE
WRITE NEXT
READ NEXT
READ
READ FIRST
```

3.4.1 DATA DIVISION Statements

If a program uses the Line-Based Open Text Access Method then the statement:

```
ORGANISATION OR$83L TYPE 3 EXTENSION 320
```

must be coded in the DATA DIVISION before the first FD or data declaration.

The file definition for an Open Text file is coded in either working storage or the linkage section as follows:

```
FD fdname ORGANISATION OR$83L
ASSIGN TO UNIT "?" FILE file-id
RECORD LENGTH IS length
[SIZE IS size]
[KEY IS key]
01    FILLER REDEFINES fdname
  02  FILLER            PIC X(80)
  02  LPTR        PIC PTR              * Pointer for OPEN operation
  02  FILLER            PIC X(24)
  02  LTNAME      PIC X(255)           * File name with terminating LOW-VALUE
  02  FILLER            PIC X(10)             * Always set to LOW-VALUES
  02  LTPERM      PIC 9(4) COMP        * Unix permissions
  02  FILLER            PIC X(4)
```

```
    02   LTLENG      PIC 9(4) COMP      * Record length
    02   LTOVER      PIC 9 COMP         * Line terminator override flag
                                        * 0 = No override
                                        * 1 = Terminate lines with <CR>
                                        * (Unix file)
                                        * 2 = Terminate lines with <CR><LF>
                                        * (Windows file)
```

Note that the End of File character will be included in the *size* field on GSM (Windows) configurations.

3.4.2 Open Operations

The path and file name of the file to be opened together with any terminating #00 must be moved into LTNAME before the Open operation is attempted.

The record length in the FD must be set up on opening the file. This record length indicates the maximum length of line that can be read into the calling program. The OPEN verbs must not be coded directly. All Open operations must be coded in this indirect manner:

```
        CALL LTPTR USING fd opcode buff
```

where opcode is a PIC 9(4) COMP field which is set to 1 for OPEN NEW and 2 for OPEN OLD and buff is the read buffer area for use by the access method as follows:

```
   01    BF
    02   BFLENG      PIC 9(4) COMP           * Buffer length
   02    BFAREA      PIC X(nnnn)             * Buffer area
```

The BFLENG field which indicates the size of BFAREA must be at least twice the maximum record length supplied in the record length field.

The buffer block will be used by the access method and must not be overwritten whilst the FD remains open. If more than a single FD is opened at any time then each FD will require its own buffer block.

3.4.3 Write Operations

When writing a line to the text file you must either set the length of the line to be written in LTLENG, or set LTLENG to -1 to indicate that the line is terminated by #00. If LTOVER is set to 0 then the access method will terminate the line when writing with a line terminator appropriate to the host operating system. If LTOVER = 1 it will always terminate the line with #0A and if LTOVER = 2 it will always terminate the line with #0D0A.

3.4.4 Write Operations

The line read will always be terminated with #00. Your record length should therefore be one byte longer to accommodate this. The length of line returned excluding the terminating #00 will be returned in LTLENG.

4. Desktop-based (GX Client) Open File Access Methods

This section describes the various Access Methods that are currently available to access Windows files on the GX Client PC (i.e. the PC that is running GX.EXE).

4.1 GX Direct Access Method (OR$98X)

The GX Direct Access Method allows direct, random-access of Window files. This access method is the "GX PC" equivalent of the Direct Windows/Unix Access Method described in section 3.1.

4.1.1 DATA DIVISION Statements

If a program uses the GX Direct Access Method then the statement:

```
ORGANISATION OR$98X TYPE 0 EXTENSION 128
```

must be coded in the DATA DIVISION before the first FD or data declaration.

The file definition for a direct file is coded in either working storage or the linkage section as follows:

```
FD filename ORGANISATION OR$98
[ASSIGN TO UNIT "?" FILE file-id ]
[KEY IS keyname]
RECORD LENGTH IS length
[SIZE IS size]
[OPTION ERROR]
[ON ERROR intercept]
01    FILLER REDEFINES filename
  02  FILLER               PIC X(100)
  02  ODFILE
    03 ODNAME    PIC X(99)            * directory path and filename
    03 ODZERO    PIC X                * Must be set to LOW-VALUE
  02  FILLER               PIC X(8)
  02  ODMODE     PIC X                * File Open mode
                                      * #01 - Read access
                                      * #02 - Write access
                                      * #04 - Allow shared read access
                                      * #08 - Allow shared write access
  02  ODTYP      PIC X                * Open flags
                                      * #01 - Normal file
                                      * #02 - Read only file
                                      * #04 - Archive
                                      * #08 - Hidden file
```

```
                                        * #10 - Write through cache
```

Long (i.e. > 99 character) filenames are supported by redefining the FD as follows:

```
01    FILLER REDEFINES filename
  02  FILLER          PIC X(100)
  02  ODHIGH     PIC X               * 1st byte set to HIGH-VALUES
  02  ODPTR      PIC PTR             * Pointer to filename
  02  FILLER          PIC X(95)           * Rest of 100 bytes unused
```

where ODPTR points to a zero-terminated filename string. For example:

```
77    MYFILZ     PIC X(?)
      VALUE      "c:\test\myfilewithaverylongname.xxx"
      VALUE      #00

      MOVE HIGH-VALUES TO ODHIGH
      POINT ODPTR AT MYFILZ
```

The indirect filename option (i.e. a single byte of HIGH-VALUE followed by a pointer to the filename string) **must** be used if the filename string is longer than 99 characters. However, this technique can also be used, as an alternative to the "embedded filename string", even if the string is less than 99 characters. Care must be taken when using the long name pointer: The 32-bit page containing the long name string must always be resident in memory when the FD is accessed. You are strongly recommended to ensure this data item is always defined in the same 32-bit page (i.e. same program or frame) that contains the FD declaration.

4.1.2 PROCEDURE DIVISION operations

The following file operations are supported:

```
OPEN NEW
OPEN OLD
CLOSE
CLOSE TRUNCATE
CLOSE DELETE
WRITE NEXT
WRITE
READ NEXT
READ
READ PRIOR
READ FIRST
READ LAST
```

4.1.2.1 OPEN NEW

The file name may be specified in various forms. For example, if the GX install directory is C:\GX, the LocalTempDirectory is C:\TEMP and the DTEMP setting in the GX.INI file is D:\TEMP, then:

Filename string	**Absolute path\file used by GX**
C:\WINDOWS\EXAMPLE	C:\WINDOWS\EXAMPLE
EXAMPLE	C:\GX\EXAMPLE
.\EXAMPLE	C:\GX\EXAMPLE
SUBDIR\EXAMPLE	C:\GX\SUBDIR\EXAMPLE
$TEMP\EXAMPLE	C:\TEMP\EXAMPLE
$WINDOWS\EXAMPLE	C:\WINDOWS\EXAMPLE
$DTEMP\EXAMPLE	D:\TEMP\EXAMPLE

If ODMODE is not established then the mode will be defaulted to read/write access. If ODTYP is not specified the type will be defaulted to "Normal".

4.1.2.2 OPEN OLD

If ODMODE is not established then the mode will be defaulted to read/write access. If ODTYP is not specified the type will be defaulted to Normal.

4.2 GX Text Access Method (OR$83G)

This Access Method provides a Write-only facility to create text files on the PC running GX. Note that this interface is also supported by thin clients running GSMWIN32.EXE.

4.2.1 DATA DIVISION Statements

If a program uses the GX Text Access Method then the statement:

```
ORGANISATION OR$83G TYPE 3 EXTENSION 100
```

must be coded in the DATA DIVISION before the first FD or data declaration.

The FD is defined as follows:

```
FD file ORGANISATION OR$83G
ASSIGN TO UNIT "?" FILE "?"
01      FILLER REDEFINES file
  02    FILLER              PIC X(92))
  02    FILENAME    PIC X(100)
```

The file name string may be up to 99 characters but must be terminated by LOW-VALUES.

A TFAM record structure is assumed.

The directory that the file will be opened in is defined in the GX.INI (or GSMWIN32.INI) file. The filename, terminated with a byte of LOW-VALUES, must be established in the FD before the OPEN NEW operation.

4.2.2 PROCEDURE DIVISION operations

Only the following operations are supported:

OPEN NEW

Note that a maximum record length is not required for this version of TFAM. The OPEN NEW operation will never return an exception.

WRITE NEXT

The WRITE NEXT operation will never return an exception.

CLOSE

The CLOSE operation will never return an exception

4.2.3 Programming Notes

The *filename* specified in the FD extension must be just a filename rather than a full path-name. When GX (or GSMWIN32.EXE) opens the file on the client PC, the directory is determined by the “LocalTempDirectory” option in the GX.INI (or GSMWIN32.INI) file. If the LocalTempDirectory setting is not specified, the “TEMP” Windows environment variable is used. If neither the LocalTempDirectory setting nor the “TEMP” environment variable are defined, the default is the current Windows directory (i.e. normally the directory that contains GX.EXE or GSMWIN32.EXE).

5. Server-based Open File Sub-Routines

This section describes the various high-level "Server-based" sub-routines that allow the manipulation of Windows/Unix files on the Server. All of these routines have been built upon the various low-level SVC-61 functions.

5.1 Equivalent of OPEN$ for Windows/Unix folder (NOPEN$)

The NOPEN$ sub-routine emulates the traditional OPEN$ function (which is used to open a Global directory for subsequent listing) for a Windows/Unix folder. NOPEN$ is fully documented in nopen$.doc.

5.2 Equivalent of LIST$ for Windows/Unix folder (NLIST$)

The NLIST$ sub-routine emulates the traditional LIST$ function (which is used to obtain a list of the files in a Global directory, following a call of OPEN$) for a Windows/Unix folder. See also NELIS$ which is an extended version of NLIST$ (whereas NLIST$ returns the Windows/Unix filename in a 20 character field, NELIS$ returns the filename in a 256-character field). NLIST$ is fully documented in nlist$.doc.

5.3 Equivalent of CLOSE$ for Windows/Unix folder (NCLOS$)

The NCLOS$ sub-routine emulates the traditional CLOSE$ (which is used to close an open Global directory, following a call of OPEN$) for a Windows/Unix folder. NCLOS$ is fully documented in nclos$.doc.

5.4 Extended version of NOPEN$ (NEOPN$)

The NEOPN$ sub-routine provides an "extended version of NOPEN$". NEOPN$ is fully documented in neopn$.doc.

5.5 Extended version of NLIST$ (NELIS$)

The NELIS$ sub-routine provides an "extended version of NLIST$". Whereas NLIST$ returns the Windows/Unix filename in a 20 character field, NELIS$ returns the filename in a 256-character field. NELIS$ is fully documented in nelis$.doc.

5.6 Extended version of NCLOS$ (NECLS$)

The NECLS$ sub-routine provides an "extended version of NCLOS$". NECLS$ is fully documented in necls$.doc.

5.7 Equivalent of RENA$ for Windows/Unix file (RENAT$)

The RENAT$ sub-routine is used to rename a Windows/Unix file on the server. RENAT$ is fully documented in renat$.doc. Note the difference between RENAT$ and the related RENAX$: RENAT$ is called with a Direct Windows/Unix Access Method FD and a filename string; RENAX$ is called with two filename strings.

5.8 Rename Windows/Unix file directly (RENAX$)

The RENAX$ sub-routine is used to rename a Windows/Unix file on the server. RENAX$ is fully documented in renax$.doc. Note the difference between RENAX$ and the related RENAT$: RENAX$ is called with two filename strings; RENAT$ is called with a Direct Windows/Unix Access Method FD and a filename string.

5.9 Equivalent of DELE$ for Windows/Unix file (DENAT$)

The DENAT$ sub-routine is used to delete a Windows/Unix file on the server. DENAT$ is fully documented in denat$.doc. Note the difference between DENAT$ and the related DENAX$: DENAT$ is called with a Direct Windows/Unix Access Method FD; DENAX$ is called with a filename string.

5.10 Delete Windows/Unix file directly (DENAX$)

The DENAX$ sub-routine is used to delete a Windows/Unix file on the server. DENAX$ is fully documented in denax$.doc. Note the difference between DENAX$ and the related DENAT$: DENAX$ is called with a filename string; DENAT$ is called with a Direct Windows/Unix Access Method FD.

5.11 Equivalent of COPY$ for Windows/Unix file COPYX$

The COPYX$ sub-routine emulates the traditional COPY$ function (which is used to copy a Global file)) for a Windows/Unix file. COPYX$ is fully documented in copyx$.doc.

5.12 Copy files using Windows filecopy function (COPYQ$)
The COPYQ$ sub-routine provides a "quick" version of COPYX$ by using the Windows CopyFile function to effect the file copy. COPYQ$ is fully documented in copyq$.doc.

5.13 Create (make) new Windows directory (MKDIR$)
The MKDIR$ sub-routine can be used to create (i.e. make) a Windows/Unix directory on the server. MKDIR$ is fully documented in mkdir$.doc.

5.14 Test for presence of Windows file (TESTF$)
The TESTF$ sub-routine can be used to test for the presence of a Windows file. Note that TESTF$ is a non-blocking sub-routine thus other GSM users will not appear to hang if the Windows CreateFile (sic) function takes a considerable time to complete). TESTF$ is fully documented in testf$.doc.

5.15 Simple Global to Windows file copy (CFFGS$)
The CFFGS$ sub-routine is a composite sub-routine that provides a very simple Global to Windows file export function. CFFGS$ is fully documented in cffgs$.doc.

5.16 Simple Windows to Global file copy (CFTGS$)
The CFTGS$ sub-routine is a composite sub-routine that provides a very simple Windows to Global file import function. CFTGS$ is fully documented in cftgs$.doc.

5. Other Server-based Sub-Routines that use the SVC-61 Interface
This section briefly describes the other sub-routines that are built upon the various low-level SVC-61 functions but which do not directly access Window or Unix files.

6.1 Blocking SVC-61 call (WINOP$)
The WINOP$ sub-routine merely provides a shell around the SVC-61 interface. It can be used by applications that wish to avoid "unstructured" SVC calls. WINOP$ is fully documented in winop$.doc.

6.2 Asynchronous SVC-61 (SVC-88) call (WINOX$)
The WINOX$ sub-routine merely provides a shell around the SVC-88 interface. It can be used by applications that wish to avoid "unstructured" SVC calls. WINOX$ is fully documented in winox$.doc.

Important Note: Only a small sub-set of the possible SVC-61 functions are supported by SVC-88. Please refer to the File Converters 8.2 Manual for full details.

6.3 Speedbase Gateway call (GATE$)

The GATE$ sub-routine provides a shell around the SVC-61 functions that interface with the Speedbase Gateway. The use of GATE$ is reserved for internal use only and is consequently undocumented.

6.4 Format Windows Error Code to Message (FMESS$)

The FMESS$ sub-routine converts a numeric Windows error code into a verbose description. FMESS$ is fully documented in fmess$.doc.

6.5 Suspend for less than a second (SUSP$)

The SUSP$ sub-routine invokes a "null" SVC-88 call in order to provide a Suspend Period of less than 1 second (the time granularity of the SUSPEND verb is 1 second). SUSP$ is fully documented in susp$.doc.

6.6 Test string for Boolean setting (BOOL$)

The BOOL$ sub-routine can be used to test if a string, normally read from the registry using the REGRS$ sub-routine (see below), is a valid Boolean setting.. BOOL$ is fully documented in bool$.doc.

6.7 Read registry string value (REGRS$)

The REGRS$ sub-routine returns value the string associated with a REG_SZ setting in the global key of the Windows registry. REGRS$ is fully documented in regrs$.doc.

6.8 Read registry numeric value (REGRV$)

The REGRV$ sub-routine returns value the string associated with a REG_DWORD setting in the global key of the Windows registry. REGRV$ is fully documented in regrv$.doc.

6.9 Determine IP address of host (HOSIP$)

The HOSIP$ sub-routine returns the IP address of the Host computer. HOSIP$ is often used in conjunction with GETIP$ (return the IP address of the PC running GX) and CMPIP$ (compare two IP addresses). HOSIP$ is fully documented in hosip$.doc.

6.10 Specify print file name for next print operation (PRIFN$)

The PRIFN$ sub-routine can be used in conjunction with the GSM (Windows) DOSPrint and DOS.PRI printer controllers to change the name of the Windows file created in a pseudo spool directory. PRIFN$ is fully documented in prifn$.doc.

6.11 Write record to Windows Logging system (LOG$)

The LOG$ sub-routine can be used to write a record to the Windows Event Logging System. LOG$ is fully documented in log$.doc.

7. Desktop-based (GX Client) Open File Sub-Routines

This section describes the various high-level "Desktop-based" sub-routines that allow the manipulation of Windows files on the PC running GX. All of these routines involve direct interaction with GX rather than low-level SVC-61 functions.

7.1 Copy files to/from GX/Server (GXCOP$)

The GXCOP$ sub-routine can be used to copy a file to/from the server from/to the PC running GX. Note that GXCOP$ does not actually perform the copy directly but uses a combination of the Direct Windows/Unix Access Method (see section 3.1) and the GX Open Access Method (see section 4.1). GXCOP$ can be used to effect any of the 4 possible copy file combinations (i.e. server to server; server to GX; GX to server; GX to GX). GXCOP$ is fully documented in gxcop$.doc.

7.2 Create (make) new directory on GX PC (GXMKD$)

The GXMKD$ sub-routine can be used to create (i.e. make) a Windows directory on the PC running GX. GXMKD$ is fully documented in gxmkd$.doc.

7.3 Check if file exists on GX PC (GXCFE$)

The GXCFE$ sub-routine can be used to check if a file exists on the PC running GX. GXCFE$ is fully documented in gxcfe$.doc.

7.4 Delete file on GX PC using direct filename (GXDEX$)

The GXDEX$ sub-routine can be used to delete a file on the PC running GX. GXDEX$ is fully documented in gxdex$.doc. Note the difference between GXDEX$ and the related GXDEL$: GXDEX$ is called with a filename string; GXDEL$ is called with a GX Open Access Method FD.

7.5 Delete file on GX PC using GX BDAM FD (GXDEL$)

The GXDEL$ sub-routine can be used to delete a file on the PC running GX. GXDEL$ is fully documented in gxdel$.doc. Note the difference between GXDEL$ and the related GXDEX$: GXDEL$ is called with a GX Open Access Method FD; GXDEX is called with a filename string.

7.6 Scratch directory of GX PC (GXSCR$)

The GXSCR$ sub-routine can be used to scratch (i.e. delete all files from) a Windows directory on the PC running GX. GXSCR$ is fully documented in gxscr$.doc.

7.7 Get Open GX BDAM file date/time (GXFDT$)

The GXFDT$ sub-routine can be used to obtain the Last Modified Date and Time of a file on the PC running GX. GXFDT$ is fully documented in gxfdt$.doc.

7.8 Specify print file name for next GX print operation (PRGX$)

The PRFGX$ sub-routine can be used in conjunction with the GSM (Windows) GXPrint printer controller to change the name of the Windows file created in a pseudo spool directory on the GX PC. PRFGX$ is fully documented in prfgx$.doc. Note that PRFGX$ is invoked by PRIFN$ automatically when the target printer is a GXPrint printer.