File I/O with ABAP — Problems, Workarounds, and Prudent Practices

Gerd Kluger

Gerd Kluger works in the Business Programming Languages Group at SAP AG, where his main responsibility is in the development of ABAP Objects and system interfaces, especially with regards to the file system. Prior to joining SAP, Gerd worked for a company focused on development of programming languages for business applications.

(complete bio appears on page 38)

File input and output (I/O) is one of the rare instances where an ABAP programmer comes into contact with the world outside of an SAP system. He leaves a perfect world where everything fits neatly together, and is confronted with the harsh reality of different operating and file systems.

As experience often shows, file I/O frequently leads to problems, such as corrupted data and mysterious program behavior. Programmers run into these issues because they either don't know about the challenges they have to expect outside of an SAP system (e.g., platform-specific peculiarities), or they aren't entirely aware of all the implicit processes happening behind the scenes (e.g., how datasets in ABAP are mapped to physical files of the file system).

This article provides an overview of the different aspects involved in working with files in ABAP, with an emphasis on typical traps and pitfalls and how to avoid them. The first part of the article concentrates on the file interface of releases up to and including 4.6. With Release 6.10, there are many changes and improvements that make working with files more secure. These changes are discussed in the second part of the article, together with the most important new features of the file interface.

Basic File I/O

A typical SAP system has a multi-tier architecture consisting of a database, one or more application servers, and many separate client frontends (presentation servers). So when we talk about file I/O, it's not immediately clear what we mean: do we mean access to files on the *presentation* server or on the *application* server? For the purposes of this article, “file handling” always happens on the *application* server. There is no way to manipulate files on the presentation server directly. If you want to access files from the client frontend, you have to copy them by using special function modules (i.e., `GUI_UPLOAD`/`GUI_DOWNLOAD`[1]), and then process them.

The two basic file operations are: reading the content of a file, and transferring some sort of data to it. In order to do so, you typically have to open the file before you can use it and then you close it if you don’t need it anymore.

In ABAP, the corresponding operations to these four tasks are:

- `READ DATASET dsn INTO f.`
  to read from a file
- `TRANSFER f TO dsn.`
  to write to a file
- `OPEN DATASET dsn.`
  to open a file
- `CLOSE DATASET dsn.`
  to close a file

In all operations, `dsn` is the name of the file as it is known to the particular operating system. That is to say, it is the “physical” file name. There is no notion of a “file handle” in ABAP to denote a file, as in other programming languages. Rather, the name of the file itself serves as a file handle. As a consequence, you cannot have the same file open multiple times at the same time in the same program.

Since `dsn` is the name of the file as it is known to the operating system, you may run into problems in a heterogeneous landscape. For example, in a UNIX system, a file is denoted by a path where the path consists of directories followed by the real name of the file, all separated by the special forward slash character “/”. So “foo/bar” is a correct file name, meaning the file “bar” in the directory “foo”. While in Windows you also have the notion of a path, directories and file names are separated by the special backslash character “\”, and directories may be related to different volumes (denoted by “drive letters”). A typical file name under Windows may be “C:\foo\bar”, meaning the file “bar” in the directory “foo” on drive “C”. As you can see from this very simple example, it’s not desirable to use file names directly (though we will do so for simplicity in the rest of this article), since if you run application servers on different platforms, a file name that is well suited for one platform is not necessarily correct in another. A much better practice is to use platform-independent, *logical* file names, whereby you define the mapping from logical file names to the corresponding physical file names in transaction FILE. Then you can use the function module `FILE_GET_NAME` in your code to get the physical name corresponding to a logical file.

Another thing to mention before going into more detail is access rights and authority checks. From the operating system’s point of view, all ABAP file operations originate from the SAP kernel. They are therefore executed under the identity of the OS user who owns the working process — usually the OS user under which the SAP system was installed. This has the following consequences:

- The OS user who started the SAP system must have OS access rights to all files and directories involved in ABAP file operations.
- OS access permissions cannot be used to control file access by different SAP users. Another mechanism is needed to control file access by different users. This is achieved by the SAP authority check, which can be configured by transactions SU02/SU03 and is integrated into the kernel to guarantee that users can only gain access to operating system resources if they are explicitly allowed to. For example, if someone tries to open a file without having the permission to do so, the `OPEN DATASET` will result in a runtime error.[2]

[1] The function modules are called `WS_UPLOAD`/`WS_DOWNLOAD` in pre-6.10 releases.

[2] In order to avoid the runtime error, use the function module `AUTHORITY_CHECK_DATASET` to check if the user has access rights.

Opening a File

As noted previously, the `OPEN DATASET` command is used to open a specific file. It has an abundance of options to tell the system what you plan to do with the file, the nature of the contents of the file, and so on.

Let's talk first about the four different ways in which you can access the file[3]:

- The `FOR INPUT` option: This option allows you to read from an existing file.
- The `FOR OUTPUT` option: This option allows you to write to a file, creating the file if it doesn't exist, or destroying its original content if it already exists.
- The `FOR APPENDING` option: This option allows you to extend an existing file (or create it if it doesn't exist).
- The `FOR UPDATE` option (as of Release 6.10): This option allows you to change the content of an existing file (i.e., to read from *and* write to an existing file). Note that the file must exist.

In the pre-6.10 releases that you work with, where the `FOR UPDATE` option does not exist, to update a file you have to open it using `FOR INPUT`, then carry out a `TRANSFER`, ignoring the `FOR INPUT` option. Why does this work? Well, things are a bit more lax than they should be! Even if a file is opened for input, you are still able to write to the file. The lack of stricter measures can lead to undesired results, since you can overwrite your file by accident. Hence our decision at SAP to introduce the new `FOR UPDATE` option, and enforce more stringent limitations on the `FOR INPUT` option, as I will show you later on.

Once you have specified how to access the file, you should specify how your data is structured:

- If the data is organized as lines, open the file `IN TEXT MODE`.

[3] This is with Release 6.10; in former releases, there were only the first three options.

- If the data is unstructured and you want to process it byte by byte, open the file `IN BINARY MODE`.

In pre-6.10 releases, the main difference between `TEXT MODE` and `BINARY MODE` is the use of end-of-line markers. Unfortunately data is always transferred (respectively read) in binary fashion in *both* modes. This leads to unexpected results if you open a file in text mode but some of your data contains byte patterns identical to the end-of-line marker, which is not unlikely if you transfer integers, for example. As you will see in just a bit, the new 6.10 release takes precautions against this situation.

One thing to note regarding end-of-line markers: As you probably know, the end-of-line marker is different on different platforms. On UNIX platforms, it is a single "linefeed" character. Windows uses the two-character sequence "carriage-return linefeed". On Windows, if you open an existing file in `TEXT MODE`, the runtime system tries to guess from the content of the file how lines are separated, and uses that style later on. The platform-dependent style is only used if the file doesn't exist already. This often confuses people, especially if a file is opened for output, since the original content is destroyed but the line style is retained.

✓ ***Tip***

In order to avoid this situation, delete the file before opening it for output. A file can be deleted by the statement DELETE DATASET.

Two additional `OPEN DATASET` options are worth mentioning:

- The `AT POSITION pos` option: To open a file midstream (e.g., not at the beginning, but at some specific position), use this option. Be careful if you want to change an existing file at some specific position! Open your file using `FOR UPDATE` in that case, since `FOR OUTPUT` will first delete the file and then go to the position given, and that's not really what you want to do! Since in pre-6.10 releases there is no `FOR UPDATE` addition to the `OPEN DATASET`, the only way to achieve this is to open the file `FOR INPUT` and take advantage of the laxness in the access type checking.

- The `FILTER f` option: This filtering option was designed to pipe the data through some external operating system command, like "compress" or "uncompress". Many programmers, however, use it to start OS commands from within ABAP.

Closing a File

If you don't need a file anymore in terms of reading or writing, you should close it by use of the `CLOSE DATASET` statement. Although not absolutely necessary, since a file is (eventually) closed automatically by the runtime system, there are several very good reasons why you should always close it explicitly:

1. Closing the file frees internal resources. There are a limited number of files that can be open at the same time, so keeping a file open longer than it needs to be is a waste of file handle resources and may come at the expense of a file that does need to be open.

2. Since all I/O is buffered, written data may not be stored physically on disk until the file is closed, at which time all buffers are flushed. With buffered output, many problems are not detected until the buffers are flushed (e.g., if there is no more disk space). The return value of the `CLOSE DATASET` may therefore be important for the continuation of your program.

3. If you opened the file with the `FILTER` and `FOR OUTPUT` options, the close is important because the background filter command will only then receive an end-of-file and will terminate on that condition. Also, it's the only way to check if the filter command succeeded.

When closing a file, the meaning of the return value of the command (`SY-SUBRC`) varies, depending on whether or not the filter option was specified during the opening of the file. If it wasn't, a return value of zero indicates that everything went okay. If the filter option was specified, however, the exit status of the shell that invoked the filter command is returned. Usually the shell exits with the exit status of the command that was executed, and it is good practice for a command to exit with a zero exit status if it succeeds, but all this is highly platform-dependent and outside of the control of the runtime system. So numerous problems can arise if you don't know exactly how your (external) command behaves, or even if the shell really knows about the command. Note that with this scheme, it is not possible to distinguish an error reported by your shell (e.g., because the command wasn't found) from an exit status of the command itself that has exactly the same return value. Sounds pretty awful, doesn't it? In practice, it's not so bad. You can generally assume that if `SY-SUBRC` is zero, everything is all right.

If something goes wrong during file close and there is a risk of losing data, an exception is thrown, particularly if there is some data left in the output buffer that cannot be transferred to disk, or if a file was opened `FOR OUTPUT` with the filter option and the filter command exits with a non-zero exit status.

Reading from a File

You read from a file by using the `READ DATASET` statement, specifying the name of the file and a variable into which the data should be transferred. The variable can be of various types — e.g., a character or numeric type. It can even be a *flat* structure, which is a structure that doesn't contain any elements that cannot be stored in-place, such as strings, internal tables, or data/object references.[4]

The return value (`SY-SUBRC`) indicates whether or not data was read — e.g., a `SY-SUBRC` of 4 tells

[4] Strings are allowed if, and only if, they are addressed explicitly as "standalone" (not as part of a structure).

you that the end-of-file was reached and no more data was available.

The way the file's data is read depends on whether the file was opened in text or binary mode. If the file was opened in binary mode, data is read until the variable you specified is filled.

> ✓ ***Tip***
>
> *Be careful not to read a large binary file into an XSTRING. Since an XSTRING has no fixed length, this will result in the whole file being entirely read into the string!*[5]

If the file was opened in text mode, it is assumed that the file consists of individual lines and each `READ DATASET` reads exactly one such line.[6] The data is then transferred to the variable. If the variable is smaller than the amount of data read, the overflow data is discarded. If the variable is bigger, it is padded with spaces.

In order to figure out how much data is read from the file, call `READ DATASET` together with the `LENGTH len` option. The value returned in the variable `len` gives the actual length of data read. The value depends on the open mode of the file. If the file was opened in binary mode, length is rendered in bytes. If it was opened in text mode, it is rendered in characters. Note that in text mode, the line separator is not returned and does not count as an extra character.

Writing to a File

You write to a file using the `TRANSFER` statement, giving the value to write and the name of the file where you want the data to go. As with the `READ DATASET` command, you can write various types of data. If you are using structures, however, you are limited to "flat" data types (similar to the `READ` statement). Usually the amount of data written is determined from the data type, but you can also give an explicit length with the `LENGTH` option. This is especially useful if you want to write only part of a larger structure (then the explicit length has to be smaller than the size of the data structure), or if you want to pad the data in the file (then the explicit length has to be larger).

The way data is written again depends on the mode specified when opening the file: in binary mode, the memory layout of the data is just transferred to the file; in text mode, trailing spaces are not written and the data is additionally terminated with an end-of-line marker.

> ✓ ***Tip***
>
> *Keep in mind that all output is buffered, so you cannot assume that after a TRANSFER the data is in the file. The data is physically stored on disk only after a flush. Flushing happens, for example, if the buffer is full, when you switch from transfer to read, when you change the file position, or if you close the file.*

Traps and Pitfalls with File I/O

When working with files, several types of problems tend to pop up over and over again. In the next sections, we'll look at some of these trouble spots, including problems induced by automatic mechanisms, different platforms and opening modes, and networking and multiple writer problems.

Problems with Automatic Mechanisms

In the early days of ABAP, a major design goal was to make life easier for the application programmer by taking care of everything that could somehow be automated. Implicit working areas for internal tables are a good example. Another is the renunciation of

[5] In 6.10 you can limit the amount of data that gets read by using the new `MAXIMUM LENGTH` option.

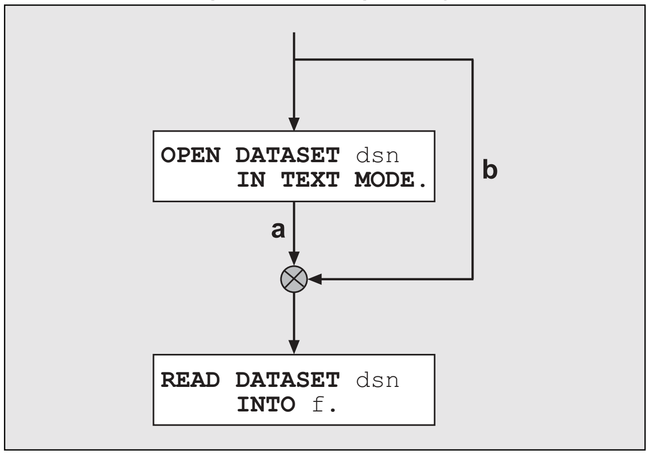
[6] Except if the `MAXIMUM LENGTH` option is used.

strict typing requirements, or simply the addition of semantics like the MOVE statement, which allows you to move (almost) any data to any variable. While these types of automated mechanisms made programming easier, they often produced insidious problems.

The dataset interface also suffers from such automatic mechanisms. Take the OPEN DATASET statement, for example. You need not tell the system if you want to write to a file[7] — even if you open a file just for input, you are able to write to the file. It is pretty obvious that this is potentially dangerous, since the file might be accidentally destroyed.

Not so obvious is the problem related to the automatic opening of a file using the first READ or TRANSFER statement. In ABAP, you need not explicitly open a file using OPEN DATASET. The file is opened automatically by the first READ or TRANSFER statement. However, since READ or TRANSFER does not include options to determine the opening mode, a default must be used. The default is the same as with an OPEN DATASET without any options: the file will be opened FOR INPUT and IN BINARY MODE. This may lead to the nasty situation illustrated in **Figure 1**.

Figure 1 Consequences of Implicit File Opening



[7] Actually you need not tell the system at all how you want to access the file. If you don't say anything, FOR INPUT is taken as the default.

Imagine that in your application a file is opened in text mode. Data is read from that file, but instead of appearing right after the OPEN DATASET statement, the READ DATASET statement appears deep down in some function module (path "a"). If the OPEN DATASET accidentally doesn't get processed (perhaps it is part of an IF statement), the READ DATASET command will open the file implicitly in BINARY MODE, not in TEXT MODE as you intended (path "b"). Since, however, the semantics of reading/writing depends to a great extent on the opening mode of a file, you will get completely different results.

The automatic opening of a file was designed to make life easier for the programmer. He shouldn't have to care if a file is really open. This sentiment was taken so far that you can call OPEN DATASET on a file that is already open.[8] But then what happens if both OPEN DATASET statements have contradicting options — e.g., one opens the file in text mode while the other opens it in binary mode? Is the first mode retained, or will the newer mode override the old one?[9] This is all pretty confusing, and it is plain that automatic mechanisms don't really add to a clear and simple understanding of the dataset interface. Therefore, with 6.10 SAP made some substantial changes in that area, as I will describe later.

Platform-Dependent Problems

Platforms are different in many aspects. One prominent example is line termination by different end-of-line markers, as noted earlier. Although SAP accounts for these types of differences inside the system kernel, the same cannot be said of all third-party vendors, and you probably need to take extra care when exchanging data with these types of solutions.

[8] Actually, in pre-6.10 releases, you *have* to resort to this practice since OPEN DATASET, together with the AT POSITION option, is the only way to change file position. In 6.10 there are now extra statements for that.

[9] The answer is that the first mode is retained, except for the file position if the AT POSITION option is used.

A more serious problem concerns discrepancies in byte order, which describes the way multibyte numbers are stored in memory. One way this is done is called *big endian*, which stores the most significant byte in the lower memory address. Big endian is common among most UNIX platforms and AS/400 systems. The alternative is *little endian*, which stores the most significant byte in the higher address (e.g., Linux on Intel processors and Microsoft Windows). If you use a file in a mixed environment, either because you have different application servers or for external data exchange, I strongly advise you *not* to store any numbers in binary format. It's always better to store the textual representation of the number instead.[10]

Problems Related to Opening Mode

When opening a file, what distinguishes text mode from binary mode? Text mode employs line separators and trailing spaces are cut off. Nothing prevents you from writing binary data in text mode. But is this a prudent practice? No. Writing binary data in text mode can lead to nasty situations.

Assume, for example, that your file is opened in text mode, and you want to write some integer variable "i" whose value is 487202848, which in binary is the byte sequence 1D0A2020.

Since the last two bytes happened to have a hexadecimal value of 20, which is equal to the character code of a space, they are handled as trailing spaces and cut off. Since we are in text mode, an additional end-of-line character (which is hexadecimal 0A on UNIX platforms) is written to the file and we end with the byte sequence 1D0A0A, as shown in **Figure 2**.

Writing, however, is not the only activity that will yield incorrect results. Take a look at **Figure 3**.

[10] See also the section on flat structures with numeric elements in anonymous containers in Christoph Stöck and Horst Keller's article "Steering Clear of the Top 10 Pitfalls Associated with ABAP Fundamental Operations and Data Types," which appeared in the July/August 2001 issue of *SAP Professional Journal*.

Let's assume you managed to get the binary value of "i" into a file, and that by chance it is even correctly terminated with an end-of-line marker. When you read that in text mode, one line consisting of the first two bytes will be read from the file since the second byte is identical to the end-of-line character. This character is then removed (remember: end-of-line characters are never returned in text mode). Since the target variable is not filled completely, remaining bytes are filled with spaces. You will end with the byte pattern 1D202020, which corresponds to an integer value of 488644640, and this is clearly not what you wanted.

What is shown for a single integer in Figure 3 is of course also true if you read or write a complete structure that contains some components of type `I` or `F`.

Multiple Writer Problem

Sometimes more than one program must write to a specific file. Even if this is not the case, you often have to take extra care that files already in use aren't modified by someone else. Logging is a typical example for the first case, while temporary files often are a culprit in the second case.

Figure 2 Writing Binary Data in Text Mode

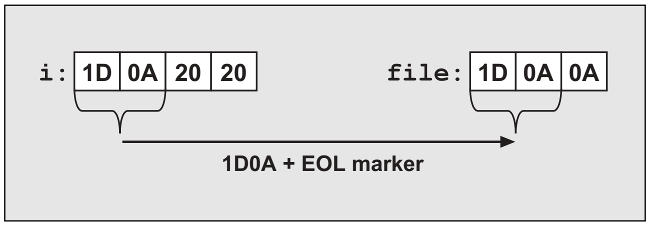


Figure 3 Reading Binary Data in Text Mode

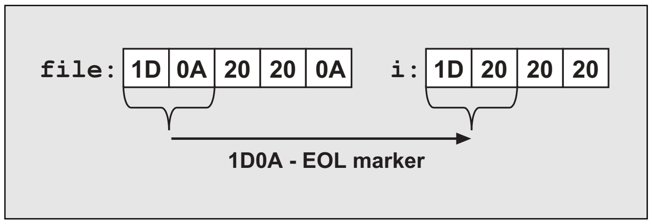
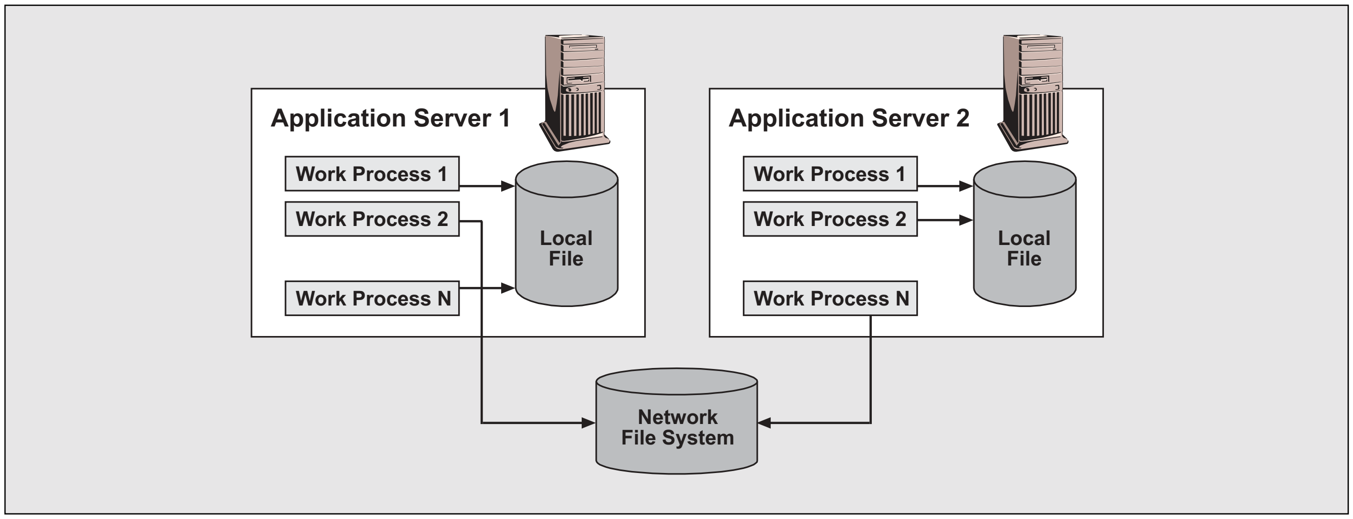


Figure 4 Multiple File Access



What problems can crop up? Turn your attention to **Figure 4**. You see that either two programs running on one application server can open a local file simultaneously, or programs on different application servers can share a network file mounted on both servers.

There is no integrated synchronization mechanism in the dataset interface to address the potential risk that multiple sources write to the same file. If two programs open a file for output and both transfer data to it at the same time, you will get mangled results.

In order to synchronize write access to a file, it's best to use SAP's locking mechanism. Define some locking object (transaction SE11) and call the generated function modules `ENQUEUE_*` and `DEQUEUE_*`. Of course, all users of the file have to follow that rule. If all you need is some local file and you just want to make sure that no one else uses it, you can also generate a unique file name — e.g., from a GUID (global unique identifier).

✓ ***Tip***

One way to get a GUID is by calling the function module GUID_CREATE.

Networking Problems

Mounted file systems can be a source of file I/O problems. Files are often accessed via a network — e.g., by using the NFS or LAN Manager. Directories are mapped to drive letters, or files are accessed directly by UNC path names.[11] In all cases, the following symptoms can be observed:

- Files are incompletely read or written.
- You get a premature end-of-file condition.
- You get a `DATASET_CANT_CLOSE` error.

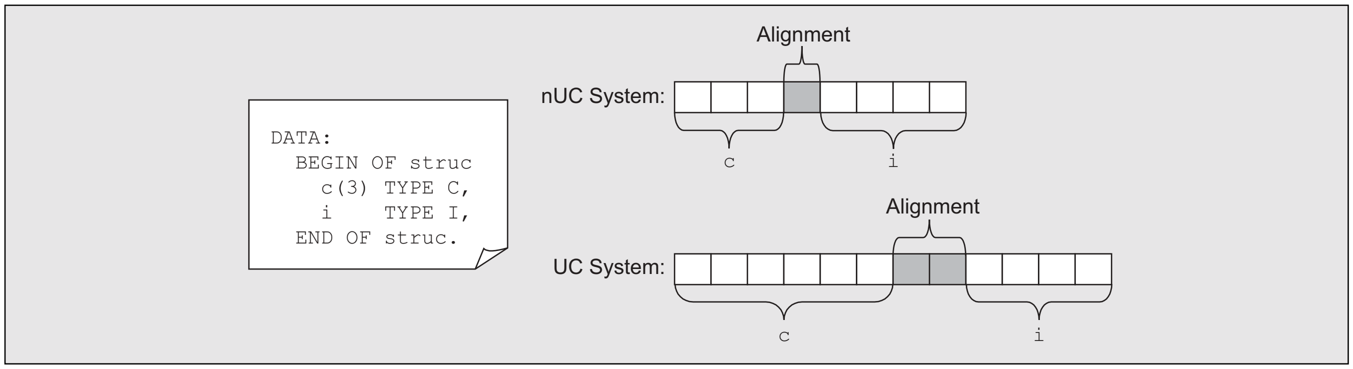
If you encounter such problems, the best thing you can do is to work on the local file system and copy the file to or from the desired location using OS commands.

The New Release 6.10 Approach

Release 6.10 is the release of the SAP Web Application Server. One of the prerequisites of the release is Unicode-enabling — i.e., making ABAP work with the Unicode character set. Unicode signifies a step away from traditional 8-bit characters, where the same character number can represent different

[11] UNC is the Universal Naming Convention. UNC path names follow the pattern "\\<host>\<share>\…"

Figure 5 Non-Unicode and Unicode Systems' Memory Layouts for a Simple Structure



```
DATA:
  BEGIN OF struc
    c(3) TYPE C,
    i    TYPE I,
  END OF struc.
```

characters in different alphabets, to a system that assigns each character one unique number in each of the major languages of the world.

One of our design goals at SAP was a smooth transition to Unicode, and one key point was a common code base: all programs should work on a Unicode (UC) system as well as on a non-Unicode (nUC) system without any changes or special programming. However, programmers do need to be careful when making the transition to Unicode with existing nUC programs. The original programmer may have made implicit assumptions about the size of a character or the layout of a data structure in memory that may no longer hold true with Unicode.

To make your ABAP program work with Unicode, you have to review the program itself to find the places where the code must be adapted. So since you have to inspect your program anyway, SAP took the opportunity to introduce a cleaner and safer dataset interface. It is not fully compatible with the old one, but your coding can be easily adapted and you will benefit from the additional safety of stricter checks. This is comparable to the introduction of ABAP Objects, which did away with old and obsolete language elements. So Unicode-enabling offers more than just a way to deal with the Unicode character set — it also offers a better way to program with ABAP. Regarding the dataset interface, this signals a move away from error-prone automatic mechanisms to a more explicit style of programming: "you say what you mean," instead of "we guess what you probably mean."

To make an nUC program a UC program, you have to subscribe to some stricter rules when programming with ABAP. This is especially true when working with files since Unicode introduces new demands with respect to the dataset interface, which I will discuss in a moment. If Unicode is not of concern to you, you might be concerned about your existing code. You need not worry, though, because as long as your program is marked as "nUC," everything will remain unchanged — and you will still be able to take advantage of all the new language elements described in the next section. However, you will not get the additional benefits from the stricter checks available with Unicode. Since in an nUC system both program types (UC and nUC) can work together, you can adapt your programs step by step or simply make only new programs UC-enabled.

Let's take a closer look at what Unicode means in an ABAP environment.

New Demands with Unicode

Unicode-enabling your system means that each character is no longer represented by one byte, but by two or possibly four bytes.

The consequences of the change in character size can be seen in **Figure 5**, which shows the memory layout of a simple structure in an nUC system and a UC system.

When comparing nUC and UC systems, what you notice first is that structures containing elements of type `C` no longer match, since the `C` components double in size (three versus six bytes). A more subtle thing to note is that the alignment may change as well, and that the byte order also becomes relevant for characters.

Data in files is often just a reflection of the memory layout, so with respect to the dataset interface there are new demands with Unicode, since there must be support for data exchange between:

- UC systems and nUC systems
- Different UC systems
- nUC systems, probably in different (traditional) code pages[12]

Note that users must be able to work with data from nUC files in a UC system and vice versa. As I said before, a major design goal was to make your programs work in a Unicode environment as well as in a non-Unicode environment without explicitly differentiating between UC and nUC systems in the application coding.

Text Format Extensions

When opening a file in text mode, simply the knowledge that it's a text file is no longer sufficient. In addition, you have to specify the encoding of the file. As of this writing, three different encoding options are currently supported:

- `ENCODING UTF-8`
  The corresponding file is read and written in UTF-8, a character format that is fully compatible with ASCII as long as only 7-bit ASCII is used. (Only characters beyond character code 127 require two or more bytes for storage.) This is the most popular Unicode format in external storage because of its transparency to 7-bit ASCII.

- `ENCODING NON-UNICODE`
  The corresponding file is read and written in the code page defined by the current language. This enables you to exchange data on a UC system with an nUC system.

- `ENCODING DEFAULT`
  The corresponding file is read or written according to the system the program currently runs on — UTF-8 in a UC system, and non-Unicode on an nUC system.

Listing 1 shows an example of a text file written with default encoding. Default encoding will be your choice if you don't want to make any assumptions about the platform (UC versus nUC) or the dataset format — e.g., because the file will only be used inside of your system. Of course, this will not be adequate if the file is for data exchange with a third-party product that requires some specific encoding.

Extended Checks in UC Programs

One very positive side-effect of Unicode-enabling is that it makes programming with datasets less error-prone. So whenever the Unicode flag is set for a program, the following restrictions apply inside of the program:

- A file must be opened explicitly by `OPEN DATASET` before you can read from or write to the file. The `OPEN` statement will fail if the file is already open.
- When you open the file you have to tell exactly for which kind of access the file shall be opened (`INPUT`, `OUTPUT`, `UPDATE`, or `APPENDING`) and in which mode (`BINARY` or `TEXT`).
- If you open the file in `TEXT MODE`, the `ENCODING` option is mandatory.

[12] For more information on code pages, see Michael Redford's article "Globalizing Applications Part 1: Pre-Unicode Solutions" in the September/October 2001 issue of *SAP Professional Journal*.

Listing 1: Writing Mixed Structures in Text Files with Default Encoding

```
DATA: BEGIN OF mixed_struc,
        last_name(30)  TYPE C,
        first_name(30) TYPE C,
        age            TYPE I,
      END OF mixed_struc.
DATA: BEGIN OF stored_struc,
        last_name(30)  TYPE C,
        first_name(30) TYPE C,
        age(3)         TYPE N,
      END OF stored_struc.

OPEN DATASET dsn FOR OUTPUT IN TEXT MODE ENCODING DEFAULT.
MOVE CORRESPONDING mixed_struc TO stored_struc.
TRANSFER stored_struc TO dsn.
CLOSE DATASET dsn.
```

- If you open the file `FOR INPUT`, you can only read from the file. If you open the file `FOR APPENDING`, you can only append data to the file. In order to both read *and* overwrite, you must open it `FOR UPDATE`.

- If a file is opened in `TEXT MODE`, only character-like data can be read from or transferred to the file. Character-like data means data of type `C`, `N`, `D`, `T`, and `STRING` as well as structures containing only components of the types `C`, `N`, `D`, and `T`.

- If a file is opened in `BINARY MODE`, only binary-like data may be read or transferred. Binary-like means data of type `X` or `XSTRING`.

The latter two restrictions are clearly the most drastic. You will find, however, that they are extremely useful for avoiding the errors described earlier in this article, like the ones described in Figures 2 and 3.

What are the consequences for mixed structures in a UC program that contains character-like data and non-character-like data (e.g., integers) at the same time? Well, you should decide which mode is appropriate. If the file is used for archiving or data exchange with other software, probably on different platforms, it's always better to use a text format, since it doesn't depend on memory layout or platform-specific properties like byte order. If, however, you want to store data temporarily and need not worry about different platforms, you may want to use binary format because storage and retrieval is much quicker.

If you have mixed structures and want to store them in a text file, you have to convert them to character-like structures, as shown in the example in Listing 1.

If you decided to write your mixed structure to a binary file, you need not convert explicitly to type `X` or `XSTRING`, instead you can look at the data

Listing 2: Writing Mixed Structures in Binary Files

```
DATA: BEGIN OF mixed_struc,
        last_name(30)  TYPE C,
        first_name(30) TYPE C,
        age            TYPE I,
      END OF mixed_struc.

FIELD-SYMBOLS: <x> TYPE X.

OPEN DATASET dsn FOR OUTPUT IN BINARY MODE.
ASSIGN mixed_struc TO <x> CASTING.
TRANSFER <x> TO dsn.
CLOSE DATASET dsn.
```

through a binary typed field symbol, as you can see in the example in **Listing 2**.

Compatibility and Conversion Issues

As we saw above, the semantics of `TEXT MODE` and `BINARY MODE` changed substantially in 6.10. So what do you do when you need to read pre-6.10 formatted data, or when you want to store data in the pre-6.10 format so that it can be exchanged with Release 4.6 systems or older? To make things even more complicated, you might want to change to Unicode someday and still be able to process old files.

Two new modes will support all these activities in 6.10: the `LEGACY TEXT MODE` and `LEGACY BINARY MODE`. They will give you the same semantics as the `TEXT` and `BINARY` mode in pre-6.10 releases, but in addition allow you to read and write nUC structures in UC programs. If run on a UC system, textual data will automatically be translated to and from the code page defined by the current language, and structures will be converted between UC and nUC appropriately, which is necessary because of different alignment demands, as I described earlier (see Figure 5).

In the pre-6.10 world, conversion between different code pages and byte orders is done with the statements:

```
TRANSLATE … FROM / TO CODE PAGE …
TRANSLATE … FROM / TO NUMBER FORMAT …
```

Both perform the conversion in-place, so after a conversion you are supposed to refrain from using the data further within your program (at least not without converting it back again). Not following this practice has been the source of many problems! The use of the `TRANSLATE` statement in a UC program is therefore now forbidden. Instead, the conversion functionality has been integrated directly into the `LEGACY` modes, so conversion takes place automatically when data crosses the border between external (i.e., file) representation and internal representation.

Automatic conversion when using the `LEGACY` modes is enabled via two new options of the `OPEN DATASET` statement:

- The `CODE PAGE cp` option: Textual data in the file is interpreted as being of code page `cp`.
- The `LITTLE ENDIAN`/`BIG ENDIAN` option: Byte-order-dependent data in the file is interpreted as being of little or big endian type.

The example in **Listing 3** shows how to write a mixed structure in a binary file that is compatible with Release 4.6 or earlier, where textual data will be stored as EBCDIC,[13] and numerical data in big endian format.

[13] IBM's Extended Binary Coded Decimal Interchange Code.

Listing 3: Writing a File in Legacy Mode

```
DATA: BEGIN OF struc,
        c(3) TYPE C,
        i    TYPE I,
      END OF struc.

OPEN DATASET dsn FOR OUTPUT IN LEGACY BINARY MODE
    CODE PAGE '0120' BIG ENDIAN.
TRANSFER struc TO dsn.
CLOSE DATASET dsn.
```

For the conversion demands beyond what we have just discussed, there are ABAP system classes that follow a stream-based approach — that is, they read from or write to a binary stream, represented by some `XSTRING`, and convert the data appropriately. The following classes are available:

- `CL_ABAP_CONV_IN_CE`: This class is used to read data from some external representation.
- `CL_ABAP_CONV_OUT_CE`: This class is used to write data in some specific format.
- `CL_ABAP_CONV_X2X_CE`: This class is used to convert between external formats.
- `CL_ABAP_CHAR_UTILITIES`: This class defines miscellaneous things like special characters, end-of-line markers, etc.

When converting textual data, there is always the risk that some character cannot be represented in a given code page. Since conversion is now automatic with the dataset interface, we need some error handling for these types of situations. So now, in the `OPEN DATASET` command, you can specify what will happen in that situation for a specific file. Being able to define this separately for each file makes error handling a property of a file. This is important, since for some files conversion errors might be critical, while for others they are harmless.

Conversion errors can be dealt with in two respects:

- You can specify a replacement character to use when some specific character is not available in a given code page. The default replacement character is the hash sign ("#"). Other replacement characters can be specified by the option `REPLACEMENT CHARACTER rc` of the `OPEN DATASET` command.
- You can specify to ignore conversion errors and use the replacement character for substitution with the `IGNORING CONVERSION ERRORS` option of the `OPEN DATASET` command. When not using this option, an exception is thrown every time a conversion error pops up. This exception can then be caught and processed.

Typically you want to ignore conversion errors when reading data from a file just for display purposes. Even if some of the characters cannot be displayed, you will usually prefer to show the user mangled strings instead of telling her that they won't be displayed at all. On the other hand, when changing existing data or writing data permanently to storage, you want the correct data, so you will most likely prefer an exception when a conversion error occurs instead of silently replacing unknown characters with a replacement character.

Other Features New with Release 6.10

In addition to the new capabilities described so far, there are some other new features in the dataset interface that address frequent customer requests, such as support for very large files, support for file names containing spaces, and features that simply make programming with datasets much easier, such as easier ways to position the file pointer, and to determine and change attributes of open files.

Large File Support

One highly sought-after feature has been support for files larger than two gigabytes. Although there are patches available for older releases to read and write progressively, it was not possible to position explicitly beyond the two gigabyte file position.

Release 6.10 closes that gap in functionality and you can now position the file pointer anywhere inside files of any size. Note, however, that offsets larger than two gigabytes cannot be stored in a variable of type `I`, so use variables of type `P`, `F`, or `N` instead if you expect to work with files that large.

Positioning the File Pointer

Before you can position the file pointer to some specific location, the question arises, "How do I determine that position?" In pre-6.10 releases, there is no way to determine the current offset of the file pointer relative to the beginning of the file. You have to count all the bytes written so far. That's not an easy task, especially in `TEXT MODE`, where you have to remember that trailing spaces are cut off and end-of-line markers are appended automatically. The fact that end-of-file markers vary in length on different platforms only serves to make matters more complicated. With Unicode, counting bytes becomes almost impossible. You can't know in advance how many bytes will be needed for a string, since in UTF-8 the number of bytes needed for a character depends on the character itself.

There is a new 6.10 instruction that solves this problem:

```
GET DATASET dsn POSITION pos.
```

This instruction gives you the position of the file pointer in file `dsn` and stores it in variable `pos`.

The counterpart to this instruction is:

```
SET DATASET dsn POSITION pos.
```

This instruction changes the position of the file pointer within the file. Note that you no longer need to call `OPEN DATASET` with the `AT POSITION pos` option just to change the file position. The fact of the matter is that you are not even allowed to do so if your program is a UC program, since this would be inconsistent with the stricter rules for opening a file.

With the above instructions, positioning in files becomes an easy task, as shown in the example in **Listing 4**, which works with files of any size if variable `pos` is of the appropriate type.

Determining and Changing File Attributes of Open Files

With 6.10, we say goodbye to the laxness in the dataset interface. It therefore becomes important to know about the properties of a file — e.g., the opening mode, the access type, error handling, and so on. Until now you always had to remember such things separately from the file, but now you can just query the file for these properties. Even better, you can change some of the settings at runtime and in so doing change the way the file is dealt with, but without the need to close and reopen the file again.

You can use the `GET DATASET dsn` and `SET DATASET dsn` instructions to review and set a file's characteristics, respectively. Above, I showed you how to determine *one* specific property of a file — the position of the file pointer — by using the `POSITION pos` option together with the `GET DATASET` command. To get all the other properties there is the option `ATTRIBUTES attr`, where `attr` must be a variable of type `dset_attributes` (a structure that is defined in the type pool `dset`) where the attributes are stored respectively. The structure is divided into two parts:

Listing 4: Positioning the File Pointer

```
DATA: pos TYPE P.

OPEN DATASET dsn FOR OUTPUT IN TEXT MODE ENCODING DEFAULT.
... " transfer as many data to the file as you like
GET DATASET dsn POSITION pos.
... " more transfers
CLOSE DATASET dsn.
...
OPEN DATASET dsn FOR INPUT IN TEXT MODE ENCODING DEFAULT.
... " read some data from the file
SET DATASET dsn POSITION pos.
... " read data from the file at position pos
```

Listing 5: Changing File Attributes at Runtime

```
TYPE-POOLS: dset.
DATA: attr TYPE dset_attributes.

OPEN DATASET dsn ...
...
GET DATASET dsn ATTRIBUTES attr.
IF attr-fixed-mode = dset_text_mode OR
   attr-fixed-mode = dset_legacy_text_mode.
  CLEAR attr-changeable.
  attr-changeable-indicator-repl_char   = dset_significant.
  attr-changeable-indicator-conv_errors = dset_significant.
  attr-changeable-repl_char = '*'.
  attr-changeable-conv_errors = dset_ignore_conv_errors.
  SET DATASET dsn ATTRIBUTES attr-changeable.
ENDIF.
...
```

- **Fixed attributes:** These are attributes that cannot be changed for an open file. The open mode, access type, encoding (if the file is a text file), and filter option belong to this category.

- **Changeable attributes:** These are attributes that might be changed without closing the file. Typical properties are the code page or the endian type (if the file is a legacy file), and the replacement character or error handling (if the file is a text or legacy file).

Both categories have an indicator structure that determines which attributes are valid for the specific file. If you want to change some of the changeable settings, don’t forget to set the corresponding flags in the indicator structure in order to tell which settings you want to change.

The example in **Listing 5** shows how the structure is to be used: After getting the file attributes, we check if the file is opened in text mode, and if this is the case we change the replacement character to “*”

and tell the runtime system to ignore conversion errors.

File Names Containing Spaces

Up to Release 4.6, file names containing spaces were not supported. If your file name contained any spaces, the name was cut off at the first space. The lack of support for spaces in file names was not problematic in the days of UNIX, but with Microsoft Windows it's no longer acceptable, since on Windows it's quite normal to use spaces in file names. With 6.10, spaces in file names *are* supported, however only in UC programs for backward compatibility reasons.

Conclusion

As you saw, working with files in ABAP is not as easy as you might have expected at first glance. There are many problems waiting to be "discovered" — even for the experienced programmer.

You can attribute the greater part of these problems to the proximity of the file interface to the operating system layer. Platform-specific peculiarities become visible and make programming more difficult. However there are also problems based on the way the file interface was defined before Release 6.10.

Platform-specific peculiarities are a fact of life every programmer has to live with. Drawing a programmer's attention to all sources of danger waiting for him is the best way to prevent problems, and I hope this article has provided you with a guide to avoiding the most insidious traps.

We have also learned within the ABAP development group at SAP about the pitfalls regarding the homegrown problems of the file interface. As a consequence, the file interfaces were revised completely with Release 6.10, and now most of the stumbling blocks are removed. Admittedly this means that your coding becomes more extensive, but avoiding hard-to-find bugs makes it worth any extra time.

Gerd Kluger studied computer science at the University of Kaiserslautern, Germany. After receiving his degree, he worked for a company whose main focus was the development of programming languages for business applications. He was responsible for the development of the compiler and programming environment for the object-oriented programming language Eiffel.

Gerd joined SAP AG in 1998 and since then has been working in the Business Programming Languages Group. His main responsibility is in the development of ABAP Objects and the further development of system interfaces, especially with regards to the file system.

He can be reached at gerd.kluger@sap.com.