GB18030 Conversion Documentation

This document contains descriptions of conversion tables and associated methods for converting data between Chinese Standard GB18030, Unicode and EBCDIC encodings.

Table of Contents

Introduction

What is Unicode?

Unicode is an international standard for the universal character encoding scheme for written characters and text. It defines a consistent way of encoding multilingual text that enables the exchange of text data internationally, while also creating the foundation for global software. The Unicode standard is a superset of all characters in widespread use today. It contains the characters from major international and national standards as well as prominent industry character sets. Versions of the Unicode Standard are fully compatible and synchronized with the corresponding versions of International Standard ISO/IEC 10646. For example, Unicode 6.1 contains all the same characters and code points as ISO/IEC 10646:2012.The Unicode Standard provides additional information about the characters and their uses. Any implementation that is conformant to Unicode is also conformant to ISO/IEC 10646

A complete description of Unicode including code point assignments, encoding forms and the principles of the standard can be found on the Unicode web site.

What is GB18030?

GB 2312-80 (the primary collection of Chinese coded graphic characters published in 1981 as a national standard) covers only 6,763 Chinese characters. In 1995, GBK (Chinese Internal Code Specification) for GB Extension was published. It is the superset of GB and completely compatible with GB 2312-80. GBK expands its character set to 20,902 characters.

GB18030 was defined in order to meet the needs of Chinese customers such as financial institutions, insurance companies and postal services that require name and address information. The characters included in this super set meet these needs.

In GB 18030, one-byte, two-byte and four-byte encoding systems are adopted. The total capability is over 1.5 million code positions. Currently, GB 18030 contains more than 27,000 Chinese characters which have been defined in Unicode 3.0. This standard provides a solution for the urgent need for the Chinese characters used in names and addresses.

Table 1: Allocation of Code Ranges

Number of Bytes	Space of Code Positions				Number of Codes
One-byte	X'00'-X'80'				129 codes
Two-byte	First byte		Second byte		23,940 codes
	X'81'~X'FE'		X'40'~X'7E' X'80'~X'FE'		
Four-byte	First byte	Second byte	Third byte	Fourth byte	1,587,600 codes
	X' 81'~X'FE'	X' 30'~X'39'	X' 81'~X'FE'	X' 30'~X'39'	

Conversion Between GB18030 and Unicode

The GB 18030 standard contains all characters defined in Unicode, but they have completely different code assignments. All 1.1 million Unicode code points, U+0000-U+10FFFF (except for surrogates U+D800-U+DFFF), map to and from GB 18030 code points. Most of these mappings can be done algorithmically, except for parts of the BMP. This makes it an unusual mix of a Unicode encoding and a traditional code page.

This document provides descriptions for two sets of tables and their associated methods. The first set of tables and methods (identified as Combined GB18030 in this document) are the default CDRA conversion tables and methods for mapping between GB18030 and Unicode. The Combined GB18030 mapping uses three binary tables (1:2 byte, 2:2 byte and 4:2 byte) as well as transformation logic to do the conversion from GB18030 to Unicode. The conversion logic uses the Code Ranges (see Table 1) to select the appropriate binary table from the three existing tables. For the Unicode to GB18030 conversion, the Combined GB18030 method uses a single binary table along with the transformation logic. The binary table is a 2:4 byte mapping table where the input code is a 2 byte Unicode code and the output is a normalized 4 byte GB18030 code.

The second set of tables and methods (identified as Components GB18030 in this document) are custom CDRA conversion tables used by z/OS only. The Components GB18030 uses three binary tables (1:2 byte, 2:2 byte and 4:2 byte) as well as transformation logic to perform the conversion from GB18030 to Unicode. Another

three binary tables (2:1 byte, 2:2 byte and 2:4 byte) and the transformation logic are used to perform the conversion from Unicode to GB18030. In this case, choosing the appropriate binary table from the three available binary tables is done by trial and error. The conversion logic is explained in detail later in this document.

UTF-16 <-> GB18030 (1, 2 and 4-byte)

This section contains information required to perform conversions between UTF-16 formatted Unicode data and GB18030. The tables referenced in the following sections are available from the CDRA Conversion Table Repository.

Text Format Tables

The conversion table repository contains both text and binary conversion tables. The following are the text, human readable, conversion tables for conversions between UTF-16 and GB18030.

Combined GB18030 Tables

UCS_GB18030.TXT (2000-11-30) - Source mapping file between GB18030 and UTF-16 from Chinese Government sources

157004B0.TPMAP100 - GB18030 (CCSID 5488) to UTF-16 (CCSID 1200)

04B01570.RPMAP100 - UTF-16 (CCSID 1200) to GB18030 (CCSID 5488)

157004B0.UPMAP100 - GB18030 (CCSID 5488) to and FROM UTF-16 (CCSID 1200)

Component GB18030 Tables

24E404B0.TPMAP100 - Text table from GB 1-byte part (CCSID 9444) to UTF-16 (CCSID 1200)

04B024E4.RPMAP100 - Text table from UTF-16 (CCSID 1200) to GB 1-byte part (CCSID 9444)

24E404B0.UPMAP102 - Text table between GB 1-byte part (CCSID 9444) and UTF-16 (CCSID 1200)

256904B0.TPMAP100 - Text table from GB 2-byte part (CCSID 9577) to UTF-16 (CCSID 1200)

04B02569.RPMAP100 - Text table from UTF-16 (CCSID 1200) to GB 2-byte part (CCSID 9577)

256904B0.UPMAP102 - Text table between GB 2-byte part (CCSID 9577) and UTF-16 (CCSID 1200)

156F04B0.TPMAP100 - Text table from GB 4-byte part (CCSID 5487) to UTF-16 (CCSID 1200)

04B0156F.RPMAP100 - Text table from UTF-16 (CCSID 1200) to GB 4-byte part (CCSID 5487)

156F04B0.UPMAP102 - Text table between GB 4-byte part (CCSID 5487) and UTF-16 (CCSID 1200)

Table files with the extension RPMAPnnn, TPMAPnnn, and UPMAPnnn contain human readable formats. They have two columns containing the source code point value (in Hex), and the target code point value (in Hex). Each file contains a brief header and column descriptions. The header in each file contains information including the values of the defined SUB characters as well as any special handling requirements.

Converting from UTF-16 to GB18030

Combined GB18030 Tables

In the combined tables, all GB code points are normalized to 4 bytes by adding leading zero-bytes to the single and double-byte values. This normalization allows for the conversion table to be a fixed 2-byte to 4-byte structure. The logic of the associated conversion method strips out the leading zero-bytes before inserting target code point into the output data stream. Each single-byte target code point has 3 leading zero bytes while each double-byte target code point has 2 leading zero bytes. For example output target code point x'5B' will be represented as x'0000005B' in the conversion table while x'8147' will be represented as x'00008147'.

2-byte to 4-byte Conversion

04B01570.UGN-R-D

When converting a UTF-16 data stream to GB18030, it is done on a character by character basis. The first step when examining an input code point is to determine if it is a valid high order surrogate value. If it is, then next code point is taken from the input stream to determine if it is a valid low order surrogate value. If together the two points comprise a valid surrogate pair then the UTF-16 to GB18030 algorithmic transformation will be used to convert the pair to the appropriate GB18030 code. Otherwise, the two bytes will be fed into the 2:4 binary table as described later (see Method 2X).

In the algorithmic transformation the target code point is calculated as follows: (Note: * = multiplication; / = division; % = modular operation; - = subtraction; + = addition in the following equations)

index = (source_codepoint1 - X'D800')*1024+source_codepoint2-X'DC00';

b0=index/12600+X'90';

b1=index/1260%10+X'30';

b2=index/10%126+X'81';

b3=index%10+X'30';

target_code point=b0*X'1000000'+b1*X'10000'+b2*X'100'+b3;

For more details see the section on transformations.

Method 2X

The binary conversion table is similar to the tables used in existing CDRA Method 2, but extended to handle the normalized GB18030 4-byte code. The input data stream consists of UTF-16 2-byte code points. The output from the conversion table will be 4-byte normalized GB18030 code points. These 4-byte codes must be de-normalized before being inserted into the output data stream.

Assumed normalization for GB18030:

GB Byte	*Normalized*	*Comment*
xx (Single byte)	000000xx	Four byte, with 3 leading zero bytes
xxxx (Double byte)	0000xxxx	Four byte, with 2 leading zero bytes
xxxxxxxx (Four byte)	xxxxxxxx	Four byte

To describe the conversion method we first define the concept of a "ward". A ward is a section of a double-byte code page. It is equivalent to a "row" of code points in ISO/IEC 10646 and in Unicode. All of the code points contained in a specific ward begin with the same first byte. A ward is populated if there is at least one character in the double-byte code page (UTF-16 in this case) whose first byte is the ward value. There are 256 wards numbered X'00' to X'FF'.

This 2 to 4-byte binary conversion table is made up of several 1024-byte vectors. The first vector acts as an index into the rest of the table. It contains 256 two-byte vector numbers (corresponding to each of the 256 wards, for a total of 512 bytes) and the remaining 512 bytes are unused (filled with zeros). There is one 1024-byte vector for each populated ward in the source code page and one additional vector used for mapping all unassigned and invalid wards.

The method extracts two bytes at a time from the input data stream. The first byte is used as a pointer into the index vector (shown in Figure 1.1). Each vector number in the index vector is two bytes long. Therefore the first byte from the input code point is multiplied by two before calculating the offset into this index vector. The two-byte value found at the corresponding position in the index vector gives the vector number in which to perform the second lookup.

The second byte of the input code point is used as a pointer into the vector specified by the index vector. When calculating the offset into this vector there are two things to remember; first, the indexing starts at zero, and second, each entry is four bytes long (normalized GB18030 code point).

In the example shown in Figure 1.1 the input code point is X'4E02'. Taking the first byte, x’4E’ and multiplying by 2 you get x’9C’ as the pointer value into the index vector. The index vector specifies x’0050’ as the vector where the output code point will be found. Taking the second byte of the input code point, x’02’ and multiplying by 4 (each output code point is 4-bytes long), indicates that the output code point will be found in the specified vector (x’50’) at location x’0008’. In the example, the resultant 4-byte output code point is x’00008140’. This value would subsequently be de-normalized to the 2-byte value x’8140’ prior to being placed in the output data stream.

In the example vector (X'0001') is used for handling code points from invalid or unassigned wards in the input data. All of the 256 four-byte (normalized) code point values found in this vector are those of the "Substitute" (SUB) character of the

target code page (X'8431A437' for GB18030). All of the entries in the index vector for unused and invalid wards point to this "substitute" vector.

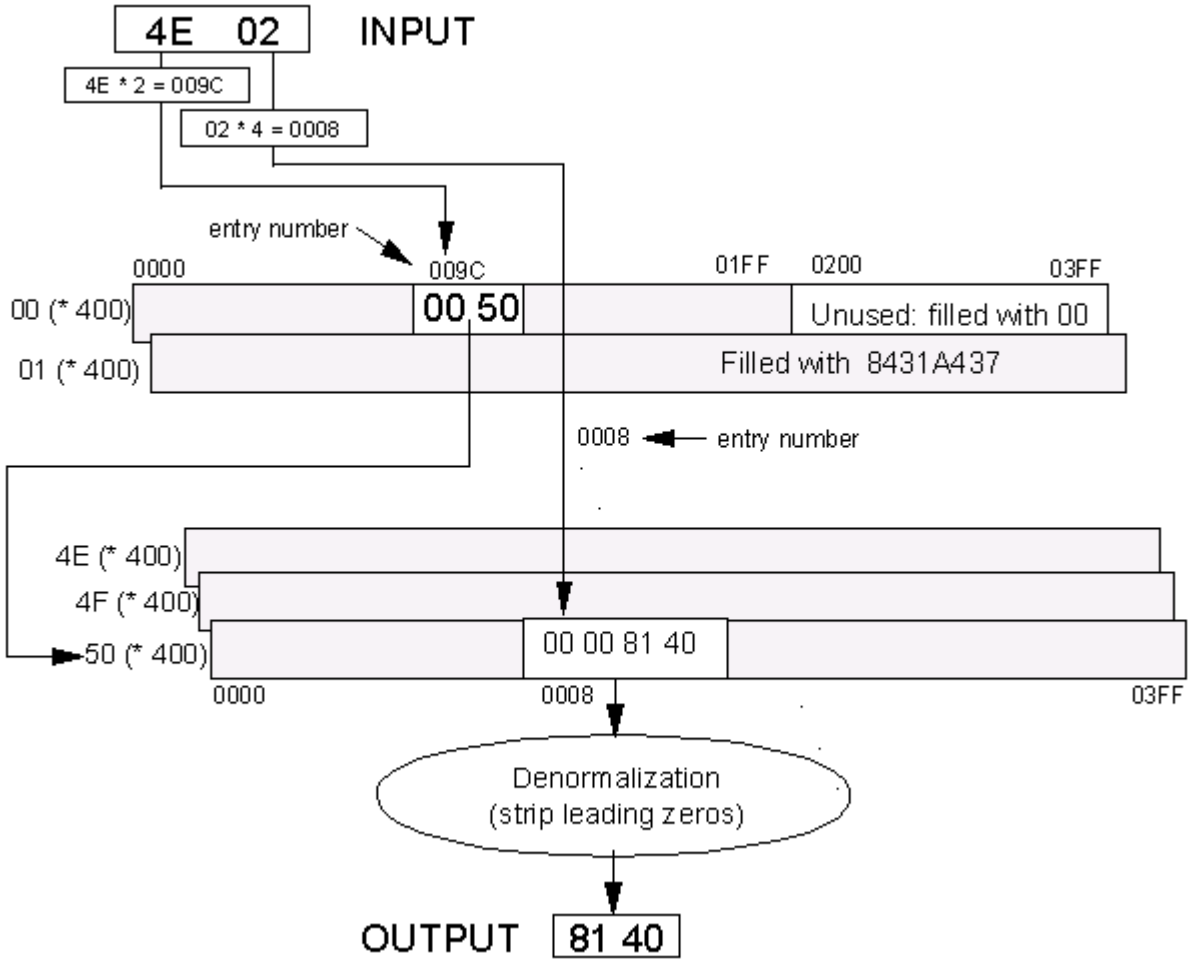

Figure 1.1 - **UCS-2 to GB18030 Binary Table & Conversion Method**

Note: For Phase 1, code points in the surrogate area (X'D800' through X'DFFF') for UCS-2 are invalid. If there is a need to distinguish between invalid and unassigned code points in the input, the method should detect and map input code points in this range to output SUB code points. In Phase 2, the surrogate area will be used for UTF-16 representation of Planes 1 to 16.

For the example shown in figure 1.1, the target four bytes could be found at the following positions in the binary table.

Byte	Value	Position (hex)	Position (decimal)
first	0	X'0050' * X'400' + X'0008'	80 * 1024 + 8 = 81928
second	0	X'0050' * X'400' + X'0008' + 1	80 * 1024 + 8 + 1 = 81929
third	81	X'0050' * X'400' + X'0008' + 2	80 * 1024 + 8 + 2 = 81930
fourth	40	X'0050' * X'400' + X'0008' + 3	80 * 1024 + 8 + 3 = 81931

De-normalization:

Since the resultant single-bytes and double-bytes in the table have been prefixed with leading zero-bytes, when composing the output string the leading zero-bytes are removed from the 4-byte output, (three zero-bytes for single-byte and two zero-bytes for double-byte).

Components for GB18030

Introduction

z/OS implementation does not use the combined tables. In the z/OS environment a call is made to CONVERT DATA indicating that the source is Unicode and the target CCSID is 5488. The converter logic looks up CCSID 05488 and gets the component conversion tables and sets up 2 to 1, 2 to 2 and 2 to 4 logic and resources. UTF-16 contains 2-byte code points and GB18030 contains 1, 2, and 4-byte code points. According to the z/OS conversion method, in order to do conversions from UTF-16 to GB18030 three binary tables are required. They are the following:

04B024E4.UG1-C0-A1 - Binary table that maps Unicode (1200) to GB 1-byte part (9444)
(CDRA 2-bytes to 1-byte conversion method, with X'FF' as STOP)

04B02569.UG2-C0-A1 - Binary table that maps Unicode (1200) to GB 2-byte part (9577)
(CDRA 2-byte to 2-byte conversion method, with X'FFFF' as STOP)

04B0156F.UG4-C0-A1 - Binary table that maps Unicode (1200) to GB 4-byte part (5487)
(CDRA 2-byte to 4-byte conversion method for GB18030, with X'FFFFFFFF' as STOP)

All of the Unicode code points located beyond the basic multilingual plane (BMP) are mapped to GB18030 4-byte code points. The GB18030 4-byte code points when converted to Unicode must be represented as either UTF-16 surrogate pairs or as 4-type, UTF-32 values.

The high level logic of this type of conversion is as follows:

The UTF-16 input stream (sequences of two-bytes) will be fed into the conversion logic. Within the conversion logic, the UTF-16 input stream will be passed through the following steps:

2:1 logic use UTF-16 to GB18030 single-byte binary table to get GB18030 single-byte output (see 2-byte to 1-byte conversion for more details)

2:2 logic use UTF-16 to GB18030 double-byte binary table to get GB18030 double-byte output (see 2-byte to 2-byte conversion for more details)

2:4 logic detects valid surrogate high and surrogate low pair, then use UTF-16 to GB18030 transformation to get GB18030 four-byte output. (see 2-byte to 4-byte conversion for more details)

2:4 logic use UTF-16 to GB18030 four-byte binary table to get GB18030 four-byte output (see 2-byte to 4-byte conversion for more details)

All invalid and incomplete surrogate pairs are dealt with separately.

The structure of the binary tables used for simulating the z/OS logic is described below:

The UTF-16 to single-byte GB18030 binary table consists of a collection of 256-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see 2-byte to 1-byte conversion for more details).

The UTF-16 to double-byte GB binary table is made up of a collection 512-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see 2-byte to 2-byte conversion for more details).

The UTF-16 to four-byte GB binary table is made up of a collection 1024-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see 2-byte to 4-byte conversion for more details).

Conversion Logic

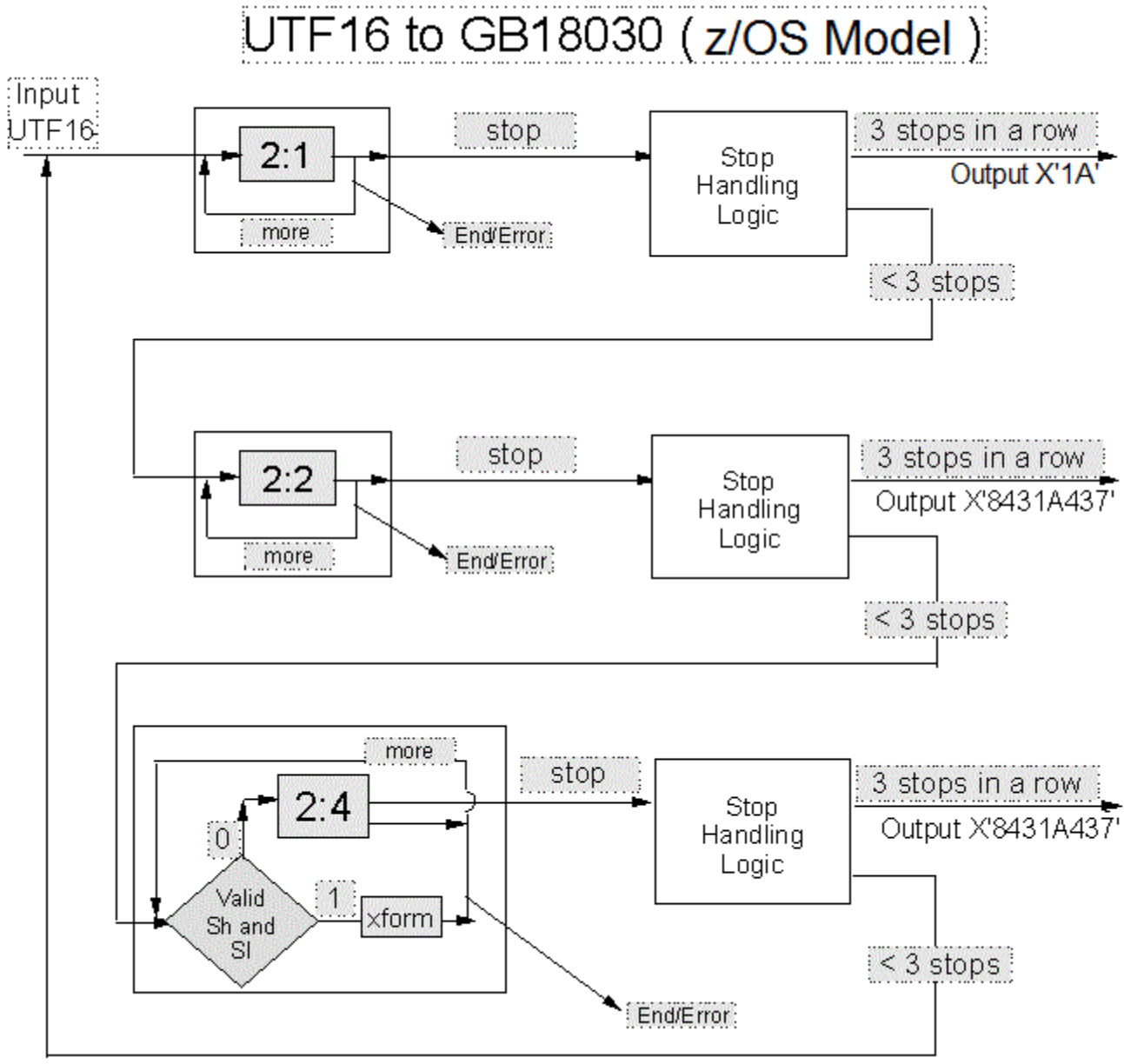

Figure 1.2 - **UTF-16 to GB18030 Conversion Method**

In understanding the UTF-16 to GB18030 z/OS conversion method, we refer to figure 1.2 for a graphical representation of the following description. Note that there exist 3 binary tables; separate tables for dealing with 2:1 byte, 2:2 byte and 2:4 byte mappings. Any input code point that has no defined mapping within any of the three tables will be detected and substitute. The precise means by which this is accomplished is described later in detail.

In this model, the UTF-16 input stream will first be fed into the conversion logic. The logic initially starts with the 2:1 mapping stage of the conversion operation. In this stage of the conversion, each two-byte segment from the UTF-16 input stream is entered into the 2:1 binary table in an attempt to convert. If the bytes are successfully converted, a valid single-byte output code is found, the conversion continues with the next two-byte input code point. This process continues until the

output code is the defined ‘stop’ (x’FF’) or the end of the input steam is encountered or an error in the input stream is detected. If an end or error condition is encountered, execution will be terminated. When a ‘stop’ character is encountered, the *stop handling* logic will be executed and the process will go on to the 2:2 mapping stage of the conversion operation.

In the 2:2 stage of operation, each two-byte segment from the UTF-16 input stream is entered into the 2:2 binary table in an attempt to convert. If the bytes are successfully converted, a valid 2-byte output code is found, the conversion continues with the next two-byte input code point. This process continues until the output code is the defined ‘stop’ (x’FFFF’) or the end of the input steam is encountered or an error in the input stream is detected. If an end or error condition is encountered, execution will be terminated. When a ‘stop’ character is encountered, the *stop handling* logic will be executed and the process will go on to the 2:4 mapping stage of the conversion operation.

In the 2:4 stage of operation, the first two bytes from the input stream will be taken to check whether it is valid surrogate high. If it is a valid surrogate high then the next two bytes will be taken from the input stream to check whether it is a valid surrogate low. If it is proven to be a valid surrogate low, then the UTF-16 to GB18030 algorithmic transformation converts the pair of valid surrogates. Otherwise, the two-byte segment from the UTF-16 input stream is entered into the 2:4 binary table in an attempt to convert. If the bytes are successfully converted, a valid 4-byte output code is found, the conversion continues with the next two-byte input code point. This process continues until the output code is the defined ‘stop’ (x’FFFFFFFF’) or the end of the input steam is encountered or an error in the input stream is detected. If an end or error condition is encountered, execution will be terminated. When a ‘stop’ character is encountered, the *stop handling* logic will be executed and the process will loop back to the 2:1 mapping stage of the conversion operation.

Figure 1.3 shows the ‘stop handling’ logic in detail

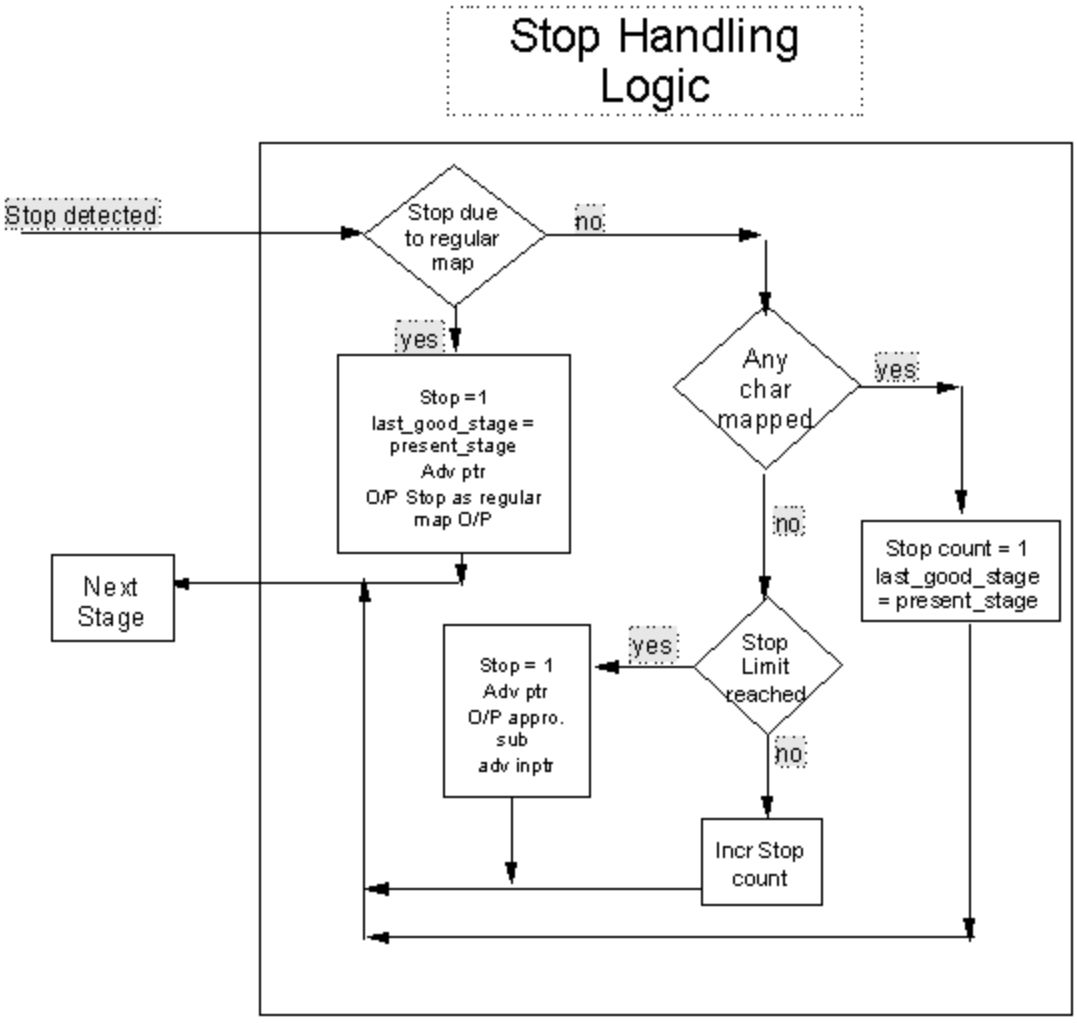

Figure 1.3 - **UTF-16 to GB18030 Conversion Method**

The stop handling logic is an integral part of the algorithm's ability to determine whether or not SUBs are to be output and detect the presence of regular mappings. This task is accomplished with the assistance of a 'stop limit' counter that effectively keeps track of the number of successive stops in the algorithm and 'last_good_stage' flag. It also remembers the last successful stage of conversion, and takes appropriate action.

In the stop handling logic, the logic first checks whether this stop is due to a regular mapping or not. If it is due to a regular mapping, the stop count will be reset to 1 and the last successful stage of operation is also set to the current mode of operation. The input pointer will be moved to the next position in the input stream while the stop (X'FFFF') character goes to the output as a target code point.

An important point to be aware of in this logic is that stop handling logic is stage dependent. The regular mapping could differ from stage to stage. So the stop handling logic in each stage should take care of the cases related to that particular stage.

If the stop is not due to the regular mapping, then the stop handling logic checks whether any successful conversion has taken place in this stage of operation. This

can be done by checking whether the input pointer has moved or not. If there is any successful conversion then the stop count will be reset to 1 and the last successful stage of operation is also set to the current stage of operation before we move on to the next stage of operation. If there is no mapping in this stage of operation then the logic checks whether the stop limit has been reached or not. The stop limit has been reached this indicates that each stage of the conversion operation (2:1, 2:2 and 2:4) has been executed without finding a successful mapping for the input 2 byte value. If this is the case then a sub is output as target according to the last successful stage of conversion. The stop count will be reset to 1 and the input pointer will be advanced to the next position.

If the stop limit is not reached, then the stop count will be incremented and the execution goes to the next stage of operation.

2-byte to 1-byte Conversion

File: 04B024E4.UG1-C0-A1

This table with its associated method is used to find the target code point when it is a single-byte.

Table format:

The binary conversion table is created by using the existing CDRA Method 6. Figure 1.4 illustrates the table format and the associated method.

The UCS-2 to single-byte table consists of a several 256-byte vectors (256 entries). The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector. The method takes each double-byte source code point and separates it into a first and second byte. The first byte is used as an offset into the index vector. The value found at this location "points" to the appropriate vector in the pool of vectors. The second byte is then used as an offset into the selected vector. The value found at this location is the single-byte target code point. Each target code point in the vector is one byte long. The binary conversion table is equivalent to existing CDRA US-R-D binary tables.

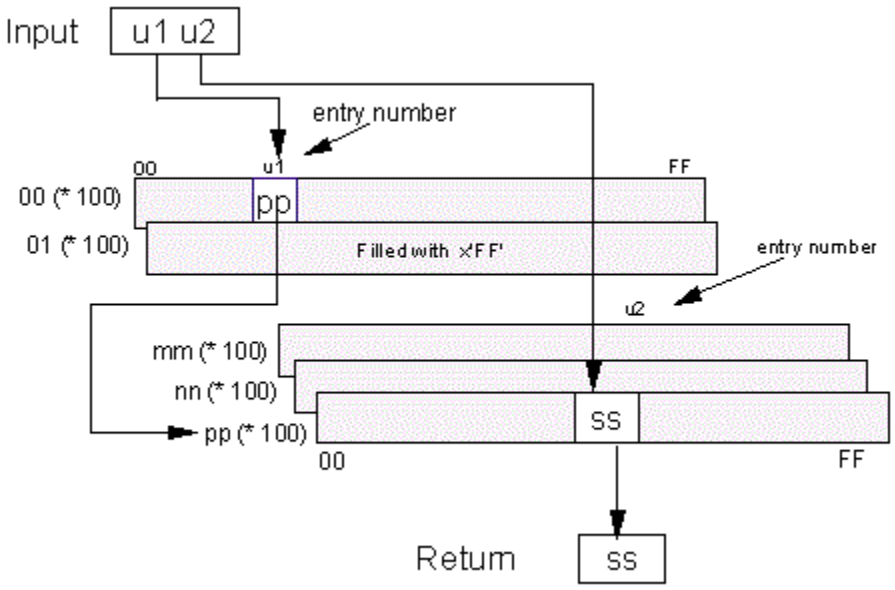

Figure 1.4 - UTF-16 to GB18030 Single-byte Binary Table & Conversion Method

2-byte to 2-byte Conversion

File: 04B02569.UG2-C0-A1

This table is used to find the target code point when the target code point is a double-byte.

Table format:

The binary conversion table is created by using the existing CDRA Method 2, a two-step vector lookup method. The conversion table and the associated method are illustrated in Figure 1.5 below.

This 2-byte to 2-byte table is made up a collection of 512-byte vectors. The first vector contains 256 single-byte pointers (vector numbers) into the rest of the table, followed by 256 unused bytes. The second vector, (the "substitute" vector) contains the mapping for code points in unassigned wards, and is filled with 256 2-byte SUB code points (X'FFFF' stop code point in this case). These are followed by additional 512-byte vectors, one for each populated ward in the source encoding.

The table is used as follows. The first byte of the input code point is used as a pointer into the index vector. The single-byte value found at the corresponding position in the index is the vector number in which to perform the second lookup. The second byte of the input code point is used as a pointer into the vector specified by the index vector.

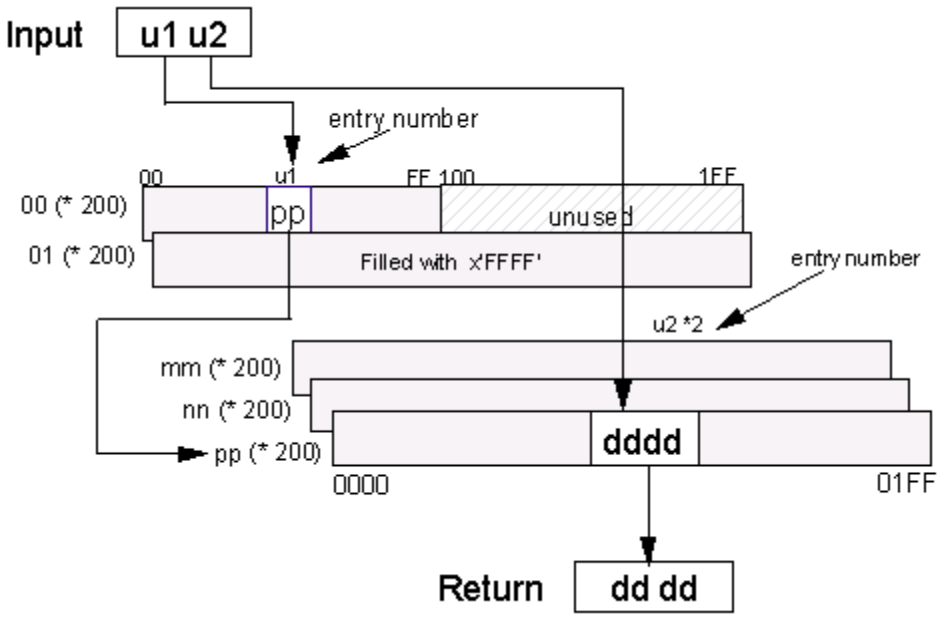

Figure 1.5 - UTF-16 to GB18030 Double-byte Binary Table & Conversion Method

In Figure 1.5 the input code point is X'u1u2'. The first byte, x'u1', is used as a pointer into the index to determine which vector in the table to use to obtain the output code point. In this case the output vector is x'pp'. The second byte, x'u2' is then used as a pointer into the specified vector to obtain the resultant double-byte output code point. Note that this value must be multiplied by 2 before the lookup is performed since each output entry is 2 bytes long.

The second vector mentioned for handling unassigned wards works as follows. All of the 256 double-byte code point values found in this vector (vector X'01') are those of the "Substitute (SUB)" character of the target encoding (in this case Stop character X'FFFF'). All the entries in the index vector for unassigned wards point to this "substitute" vector.

The binary conversion table is equivalent to existing CDRA UM-R-D binary tables.

2-byte to 4-byte conversion

File: 04B0156F.UG4-C0-A1

This binary table contains the mapping from UCS-2 to four-byte GB18030.

In this section before looking into the binary table, each UTF-16 input value is checked to see whether it is a valid high surrogate. If it is then the next two bytes segment will be taken from the input stream to check whether it is a valid low surrogate. If the pair is determined to be a valid surrogate pair, then the UTF-16 to GB18030 transformation will take place.

In the transformation the target code point will be calculated as follows:

(Note: * = multiplication; / = division; % = modular operation; - = subtraction; + = addition in the following equations)

index = (source_codepoint1 - X'd800')*1024+source_codepoint2-X'dc00';

b0=index/12600+X'90';

b1=index/1260%10+X'30';

b2=index/10%126+X'81';

b3=index%10+X'30';

target_code point=b0*X'1000000'+b1*X'10000'+b2*X'100'+b3;

For more details on transformation please see the transformations section of this document.

If the pair is not a valid surrogate pair, then the two bytes will be fed into the 2:4 binary table as described below.

Method 2X

The binary conversion table is similar to the tables used in existing CDRA Method 2, but extended to handle the GB18030 4-byte code. The input data stream consists of UTF-16 2-byte code points. The output from the conversion table will be 4-byte GB code points.

To describe the conversion method we first define the concept of a "ward". A ward is a section of a double-byte code page. It is equivalent to a "row" of code points in ISO/IEC 10646 and in Unicode. All of the code points contained in a specific ward begin with the same first byte. A ward is populated if there is at least one character in the double-byte code page (UTF-16 in this case) whose first byte is the ward value. There are 256 wards numbered X'00' to X'FF'.

This binary conversion table is made up of several 1024-byte vectors. The first vector acts as in index into the rest of the table. It contains 256 two-byte vector numbers (corresponding to each of the 256 wards for a total of 512 bytes) and the remaining 512 bytes are unused (filled with zeros). There is one 1024-byte vector for each populated ward in the source code page and one additional vector used for mapping all unassigned and invalid wards.

The method extracts two bytes at a time from the input data stream. The first byte is used as a pointer into the index vector (shown in Figure 1.6). Each vector number in the index vector is two bytes long. Therefore the first byte from the input code point is multiplied by two before calculating the offset into this index vector. The two-byte value found at the corresponding position in the index vector gives the vector number in which to perform the second lookup.

The second byte of the input code point is used as a pointer into the vector specified by the index vector. When calculating the offset into this vector there are two things to remember; first, the indexing starts at zero, and second, each entry is four bytes long (GB18030 code point).

In the example shown in Figure 1.6 the input code point is X'4D02'. Taking the first byte, x'4D' and multiplying by 2 you get x'9A' as the pointer value into the index vector. The index vector specifies x'004F' as the vector where the output code point will be found. Taking the second byte of the input code point, x'02' and multiplying by 4 (each output code point is 4-bytes long), indicates that the output code point will be found in the specified vector (x'4F') at location x'0008'. In the example, the resultant 4-byte output code point is x'8234F437'. This value would placed in the output data stream.

The one additional vector (X'0001') mentioned for handling code points from invalid or unassigned wards in the input data is used as follows. All of the 256 four-byte code point values found in this vector are those of the "Substitute" (SUB) character of the target code page (here in this case X'FFFFFFFF' stop character). All of the entries in the index vector for unused and invalid wards point to this "substitute" vector.

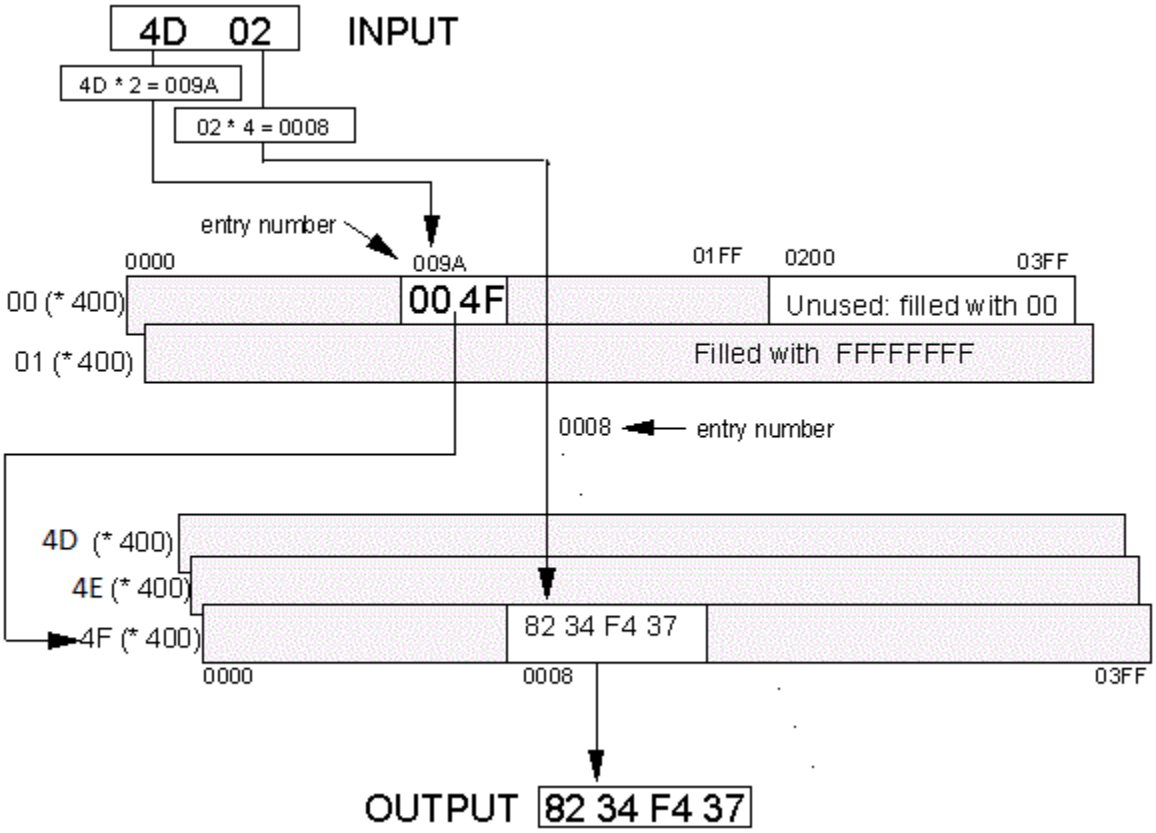

Figure 1.6 - UTF-16 to GB18030 Four-byte Binary Table & Conversion Method

In the binary table, target four-bytes could be found at the following positions.

Byte	Value	Position (hex)	Position (decimal)
first	82	X'004F' * X'400' + X'0008'	79 * 1024 + 8 = 80896
second	34	X'004F' * X'400' + X'0008' + 1	79 * 1024 + 8 + 1 = 80897
third	F4	X'004F' * X'400' + X'0008' + 2	79 * 1024 + 8 + 2 = 80898
fourth	37	X'004F' * X'400' + X'0008' + 3	79 * 1024 + 8 + 3 = 80899

Converting from GB18030 to UTF-16

Combined GB18030 Tables

Introduction

Following is a description of a GB to UCS conversion, as an alternative to following the CDRA's EUC normalization method and the resulting table structure. The normalization steps of detecting when it is a single, double, or four-byte is followed. However, instead of normalizing the data as in the EUC case, this method takes three branches in the logic and comes up with associated data structures. The data structures are linear arrays – one for each of the input types – single, double and four-bytes. The indexing operations get into these arrays only for valid ranges of code points. All other code points and broken multi-byte sequences are trapped and are dealt with separately in the logic.

The high level logic is:

- Detect valid single, double or four-byte code points (see Detecting Valid (Single, Double, and Four-byte) or Invalid Code Points for more details)

- Use single-byte to UCS-2 array for single-byte input (see 1-byte to 2-byte conversion for more details)

- Use double-byte to UCS-2 array for double-byte input (see 2-byte to 2-byte conversion for more details)

- Use four-byte to UCS-2 compact array for four-byte input (see 4-byte to 2-byte conversion for more details)

- All invalid and incomplete sequences dealt with separately (see 2-byte to 4-byte conversion for more details)

The main advantage of this method is to be able to keep the conversion tables as compact as possible (especially for the 4-byte to 2-byte part) and to be able to get at the converted UCS-2 code points in a relatively fast manner.

The single-byte to UCS-2 array consists of a single 512-byte vector (256 2-byte entries).

The double-byte GB to UCS-2 array is made up of several 512-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see 2-byte to 2-byte conversion for more details)

The four-byte to UCS-2 compact array is made up of:

- 256 bytes of flag values (followed by 3*256 bytes of unused bytes)
- Four 1024-byte long vectors, each containing 256 4-byte index values, and
- A long compact array containing target two-byte UCS-2 code points.

Detecting Valid (single, Double, and Four-byte) or Invalid Code Points

Refer to Figure 1.7 below. The method fetches one byte at a time from the input stream. The first byte, b0, is checked to see whether it is in the valid range of single-byte or start of a double-byte or four-byte code point. If it is not, then a SUB code point (X'001A' in this case) is inserted in the output data stream, and the pointer to the input stream is incremented by one.

If b0 is in the valid range of single-byte (X'00' - X'80'), then it is passed to the 1-byte to 2-byte conversion as a valid single-byte code point. The resultant 2-byte output is placed in the output buffer, and the pointer into the input stream is incremented by one.

(Note: X'80' is a valid but unassigned code point, per confirmation received through IBM China from Chinese sources. In one of the early conversion tables, it was mapped to the EURO SIGN (U+20AC) in UCS-2. In subsequent tables this has been removed.)

If b0 is in the range X'81' - X'FE' for a valid first byte of a double-byte or four-byte code point, then the next byte, b1, is fetched and checked to see whether it is in the valid range for a second byte of a double-byte (X'40' - X'7E' or X'80' - X'FE') or a second byte of a four-byte (X'30' - X'39') code point. If it is not, then the SUB code point X'001A' is inserted into the output stream and the pointer into the input stream is incremented by one. The pointer should point to byte b1 now. The first byte b0 is considered to be the start of a broken sequence of bytes in the input.

If b1 is within the valid range for a second byte of a double-byte code point (X'40' - X'7E' or X'80' - X'FE'), the sequence b0 b1 (or the input pointer) is passed to the 2-byte to 2-byte conversion (see 2-byte to 2-byte conversion). The resultant two-byte output is placed in the output stream, and the pointer into the input stream is incremented by two.

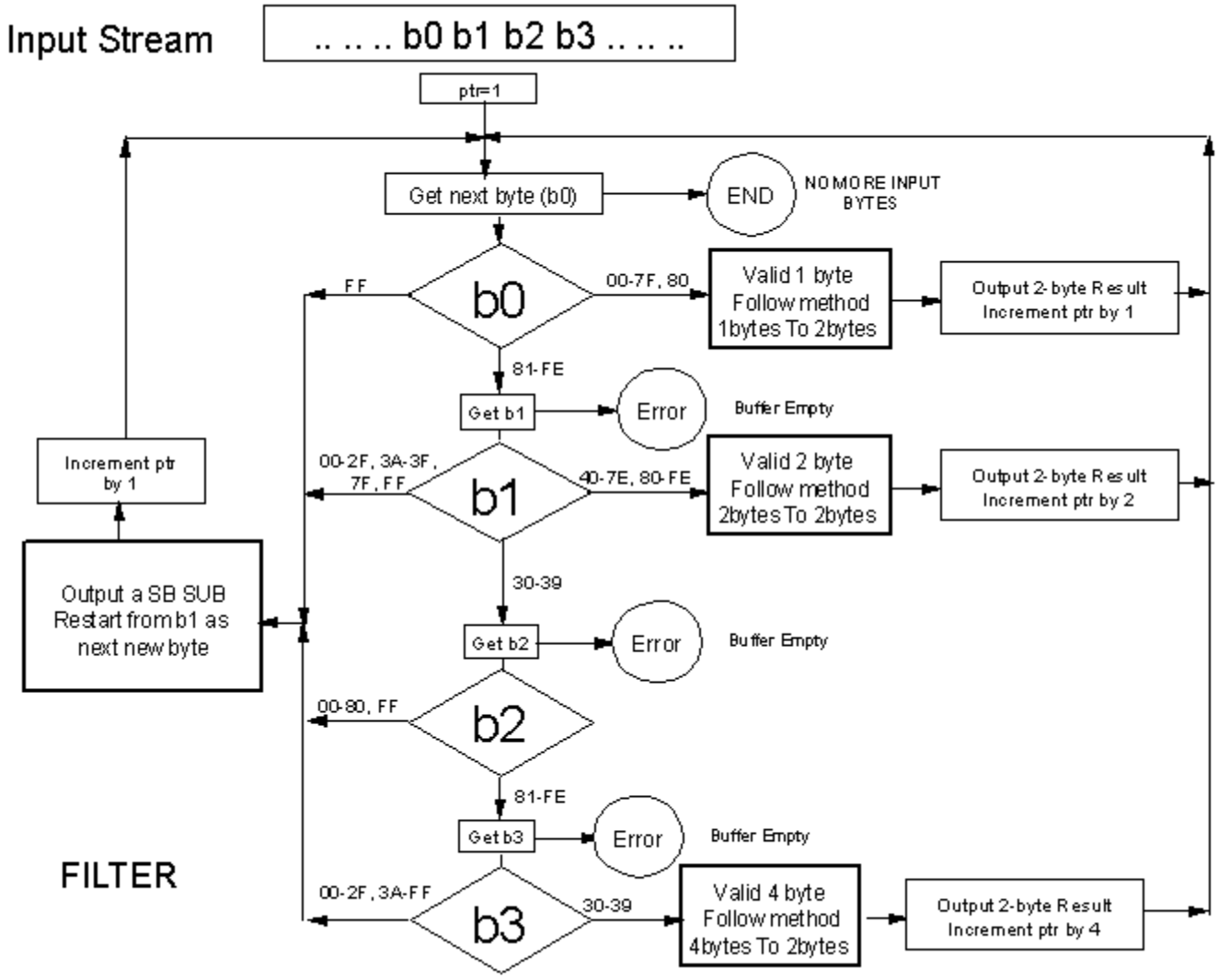

Figure 1.7 - Input Filter for GB18030-1 stream<

If b1 is within the valid range for a second byte of a four-byte code point (X'30' - X'39'), then the next byte in the input stream, b2, is checked whether its in the valid range for a third byte of a four-byte code point (X'81' - X'FE'). If it is not, then the SUB code point X'001A' is inserted into the output stream and the pointer into the input stream is incremented by one. The pointer should point to byte b1 now. The first byte b0 is considered to be the start of a broken sequence of bytes in the input.

If b2 is within the valid range for a third byte of four-byte code point, the next input byte, b3, is checked whether it is in the valid range for the fourth byte (X'30' - X'39'). If it is not, then the SUB code point X'001A' is inserted into the output stream and the pointer into the input stream is incremented by one. The pointer should point to byte b1 now. The first byte b0 is considered to be the start of a broken sequence of bytes in the input.

If b3 is within the valid range for the fourth byte then the sequence of bytes, b0 b1 b2 b3, (or the input pointer) is passed to the 4-byte to 2-byte conversion (see 4-byte to 2-byte conversion) as valid four-byte code point. The resultant two-byte output is

placed in the output stream, and the conversion pointer into the input stream is incremented by four.

The above steps are repeated until there is no more data in the input stream. If the input stream is exhausted during fetching of any of the second, third or fourth bytes in the above filtering logic, then the conversion logic cannot proceed and there will be some unconverted data in the input stream. The pointer into to the input stream indicates the first of the remaining bytes that have not been converted. An implementation may choose to deal with such a situation in any suitable manner.

The insertion of X'001A' for the first byte of a broken sequence permits locating where such a sequence might have been in the input by examining the output stream.

1-byte to 2-byte Conversion

File: 157004B0.G1U-R-D

This table with its associated method is used to find the target code point when the source code point is a valid single-byte GB18030 code.

Table format:

The binary conversion table is created by using the existing CDRA Method 5. Figure 1.8 illustrates the table format and the associated method.

The single-byte to UCS-2 table consists of one 512-byte vector (256 2-byte entries). The source code point is used as a pointer to determine which 2 bytes in the vector represent the target code point. Each target code point in the vector is two bytes long. Therefore the input code point is multiplied by two before calculating the offset into this vector. The binary conversion table is the same format as existing CDRA SU-R-D binary tables.

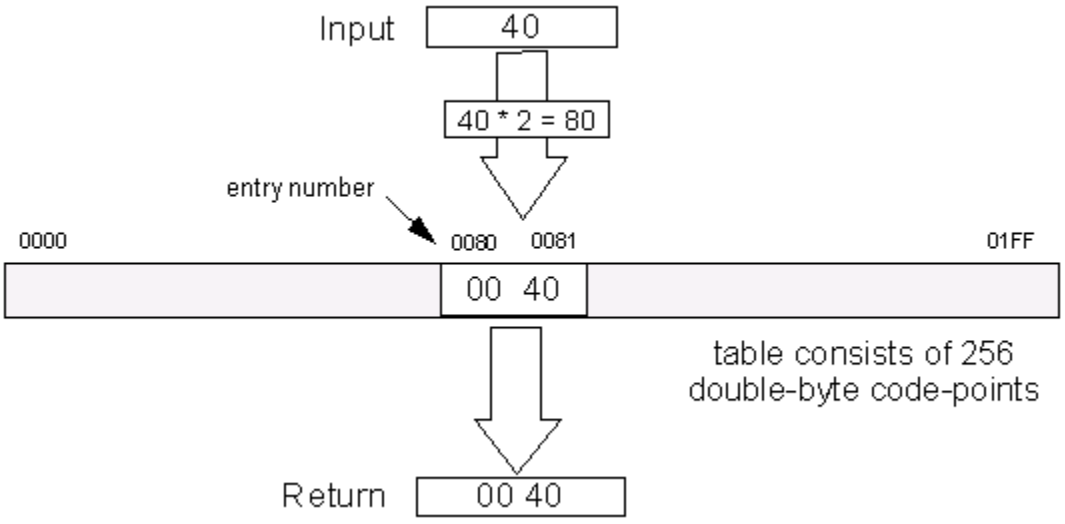

Figure 1.8 - GB 1-byte to UCS-2 2-byte Table and Method

2-byte to 2-byte Conversion

File: 157004B0.G2U-R-D

This table is used to find the target code point when the source code point is a valid double-byte GB18030 code.

Table format:

The binary conversion table is created by the existing CDRA Method 2, a two-step vector lookup method. This is similar to the one described in 2-byte to 4-byte conversion except here each vector is 512 bytes long (256 double-bytes) instead of 1024 bytes (256 four-bytes).

The conversion table and the associated method are illustrated in Figure 1.9 below.

This 2-byte to 2-byte table is made up of several 512-byte vectors. The first vector contains 256 single-byte indices (vector numbers) into the rest of the table, followed by 256 unused bytes. The second vector, the "substitute" vector contains the mapping for code points in unassigned wards, and is filled with 256 2-byte SUB code points (X'FFFD' in this case). These are followed by 512-byte vectors, one for each populated ward in the source encoding.

The table is used as follows. The first byte of the input code point is used as a pointer into the index vector. The single-byte value found at the corresponding position in the index is the vector number in which to perform the second lookup. The second byte of the input code point is used as a pointer into the vector specified by the index vector.

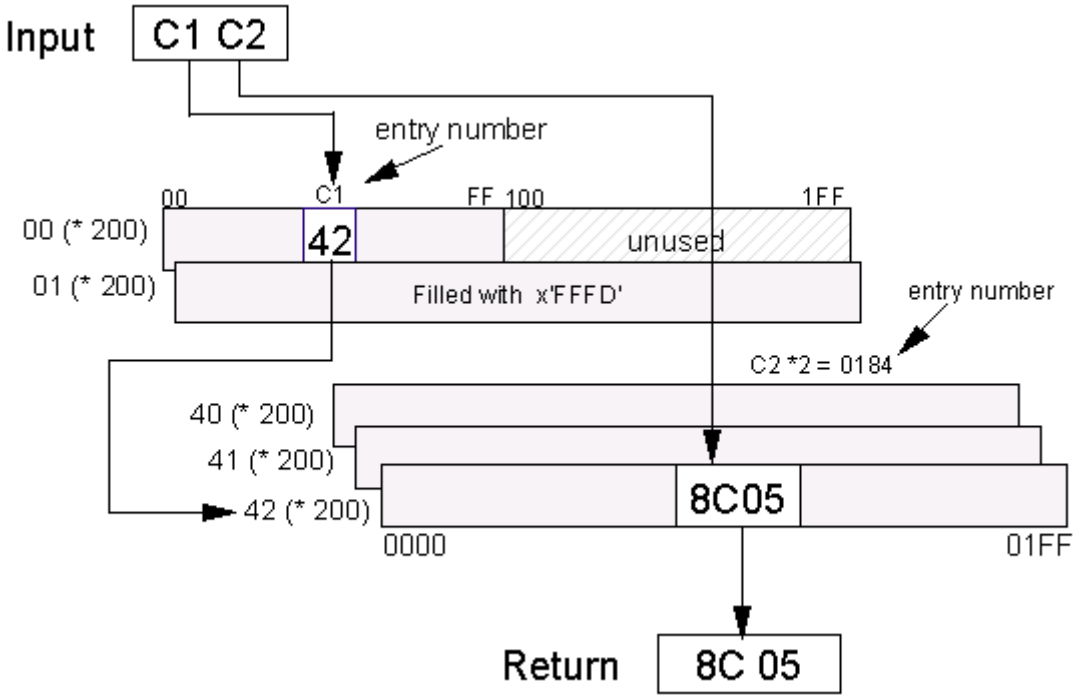

Figure 1.9 - GB 2-byte to UCS-2 2-byte Table and Method

For example (see Figure 1.9) if the input code point is X'C1C2', you would find a vector number at the X'C1' position in the index vector. The resultant double-byte output code point is found in the specified vector number (x'42') beginning at position X'0184' (which is two times X'C2'), counting into the vector starting at zero.

The second vector mentioned for handling unassigned wards works as follows. All of the 256 double-byte code point values found in this vector (vector X'01') are those of

the "Substitute (SUB)" character of the target encoding (X'FFFD' for UCS-2). All the entries in the index vector for unassigned wards point to this "substitute" vector.

The binary conversion table is the same format as existing CDRA MU-R-D binary tables.

4-byte to 2-byte Conversion

File: 157004B0.G4U-R-D

Table format:

The binary conversion table format and the associated method are detailed below.

This binary table consists of five 1024-byte vectors followed by a large linear array corresponding to the table structure as shown in Figure 1.10. There will be one three-dimensional sub array for each first byte (b0) value used in the table definition. Each sub array will contain the minimum number of cells - 12600 cells - needed to map the valid ranges of the second (b1, 10 values), the third (b2, 126 values) and the fourth (b3, 10 values) bytes of four-byte code points in GB. Each cell contains the corresponding 2-byte UCS-2 code point.

The first 256 bytes of this binary table, called the b0_used_array, is used to check if a particular value of b0 byte is used in the conversion table definition. The next 768 bytes (=3*256) are unused. These are followed by four 1024-byte vectors, K40, K41, K42, and K43. These vectors contain values used in computing an index into the rest of the table to determine the location of the output code point.

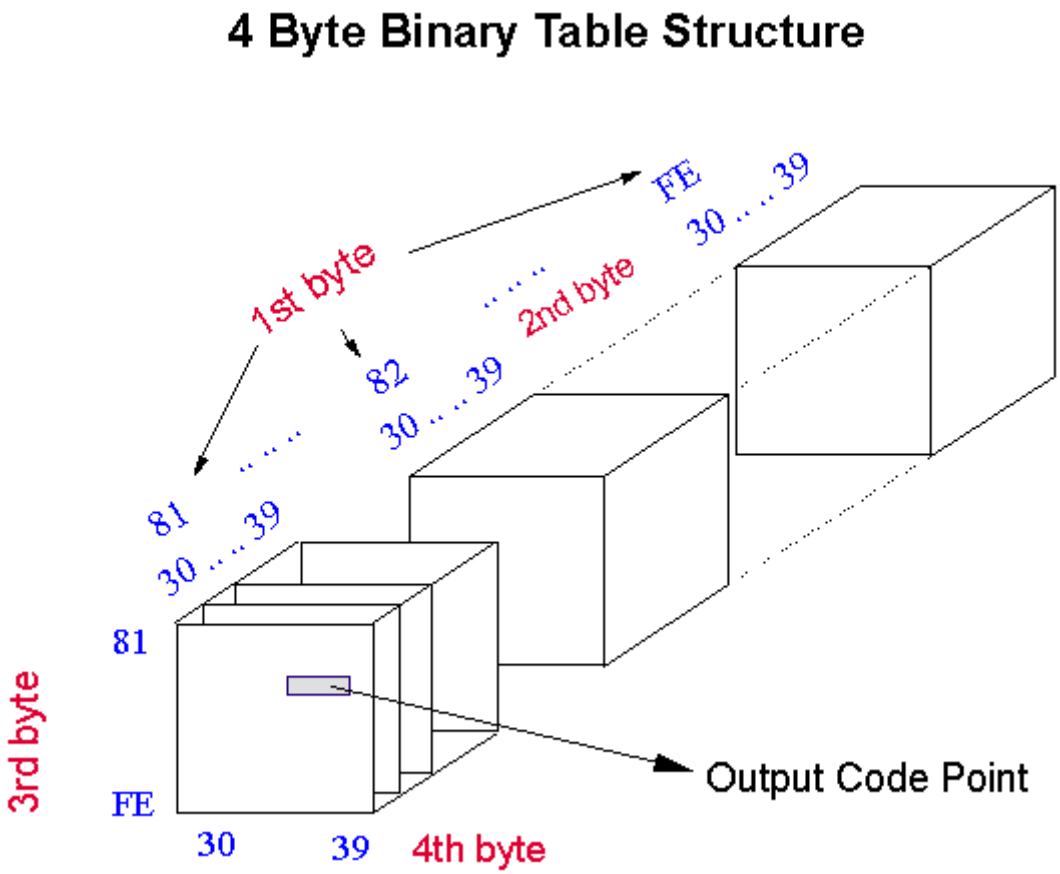

Figure 1.10 - GB 4-byte to UCS-2 2-byte table structure

For a given 4-byte code point (b0 b1 b2 b3), if the b0 is not used then there will not be a sub array populated for it. Therefore, before trying to find the target code point, the b0 value is checked to confirm that it has been used. This is checked by using the b0_used_array. As shown in Figure 1.11, b0 is used as an offset into b0_used_array. If the b0 value has been used, then the corresponding entry in the b0_used_array will be 01. If it has not been used the entry will be 00. If it is a 00, then the code point (b0 b1 b2 b3) is unassigned and its mapping is not defined in the conversion table. In this case a double-byte SUB code point (X'FEFE') is inserted by the conversion logic into the output data stream and the pointer into the input stream is incremented by 4.

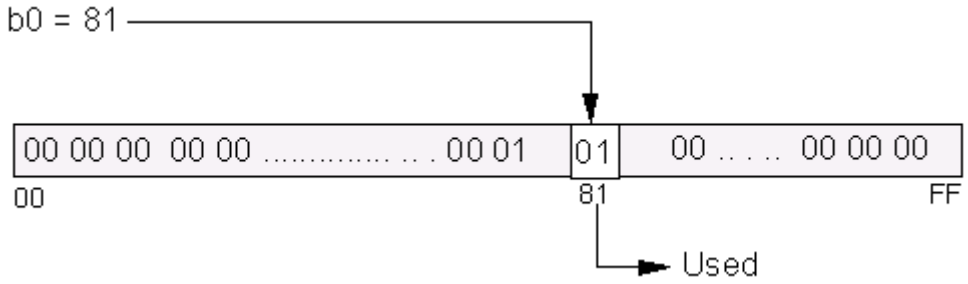

Figure 1.11 - b0_used_array

If b0 is equal to 01, then proceed with the steps to find target code point, as illustrated in Figure 1.12, and described below.

Get the computed index values from the four index vectors, K40, K41, K42, and K43 using b0, b1, b2, and b3 byte values of the input code point. Each byte of the valid four-byte source code point is used as an offset into the corresponding K4x table to get a part of the index value. When calculating the offset into these vectors there are two things to remember; first, you must begin counting at zero, and second, each entry is four bytes long. This means you have multiply the b0, b1, b2, or b3 by four before looking into the vectors. The resulting 4 values are added to get the location of the first byte of the target code point in the binary table.

The values in the index vectors are computed based on the following formulae:

K40(b0) = 25200 * (block number assigned to the b0 group)

K41(b1) = 2* (b1 - X'30') * (126 * 10)

K42(b2) = 2* (b2 - X'81') * 10

K43(b3) = 2* (b3 - X'30')

This eliminates the multiplication steps during conversion execution, improving the performance. Once these indices are added it will be the index or pointer value to the first byte of the cell containing the output code point in Figure 1.10. Figure 1.12 shows the mapping table of Figure 1.10 in as a linear structure.

Also, note that entries in these index arrays for illegal values of b0, b1, b2, or b3, are filled with zero-bytes. They will never be accessed if the filter logic has been followed correctly. Figure 1.12 also shows a possible entry for b0 = X'FD', Private Use Area, as a possible fifth block. The binary table currently defined contains mapping tables only for b0 values of X'81' to X'84', as defined in the conversion requirements received from Chinese Government sources.

The two bytes of the output code point are inserted into the output data stream. The pointer into the input data stream is incremented by four in the main filter logic described earlier in the section on detecting valid (single, double, and four-byte) or invalid code points.

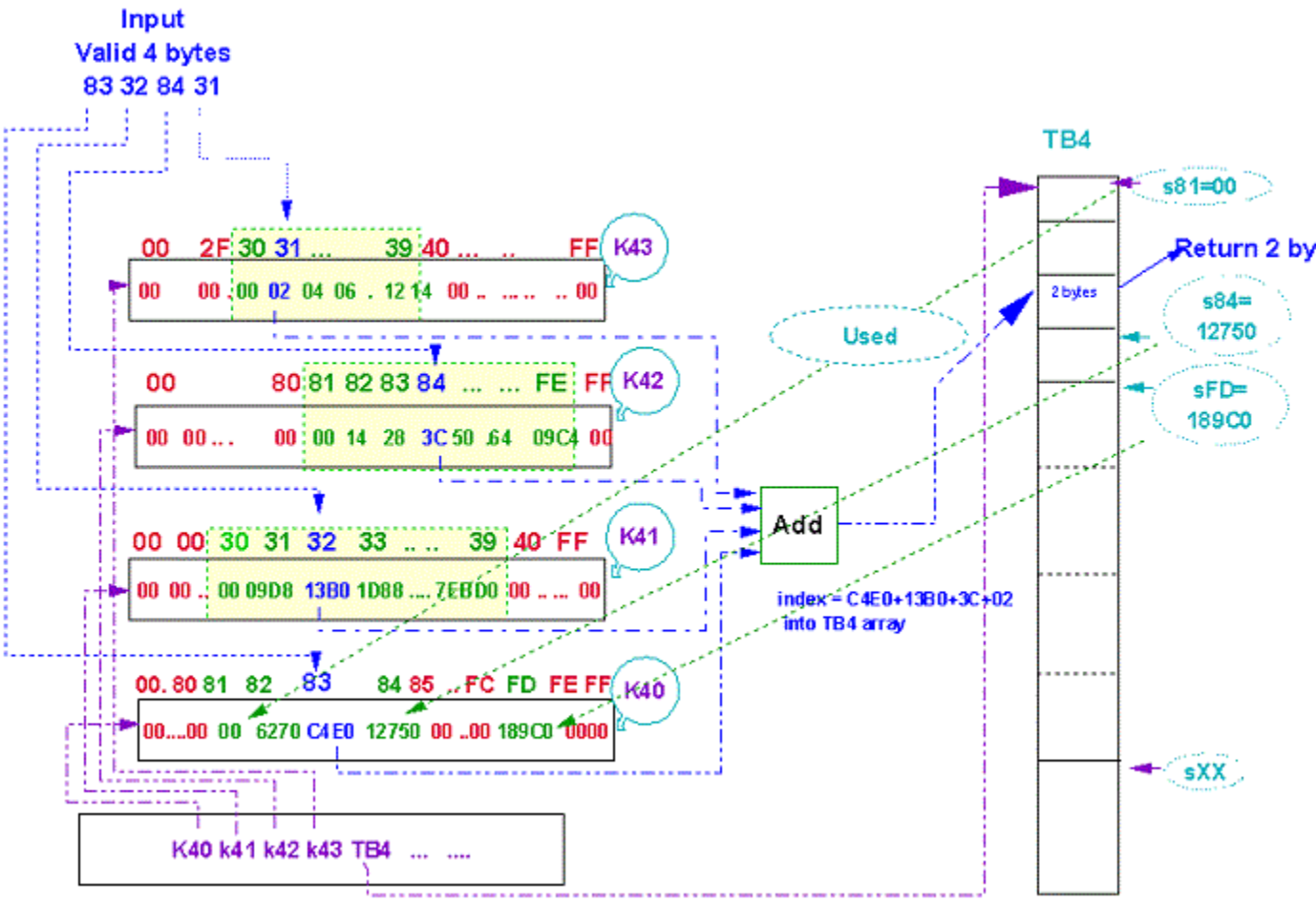

Figure 1.12 - Use of Index Arrays in GB 4-byte to UCS-2 2-byte conversion method

If b0 is not equal to 01 but the four-byte source code point is greater than X'90308130' and less than X'E3329A35', then proceed with the steps to find target code point using the GB18030 to UTF-16 transformation logic described below.

In this example i is a four-byte GB18030 code point.

(Note: * = multiplication; / = division; % = modular operation; - = subtraction; + = addition in the above equations)

b0= (((i >> 24)- 0x90) * 12600);

b1= ((((i >> 16) & 0xFF) - 0x30)*1260);

b2= ((((i >> 8) & 0xFF) - 0x81)*10) ;

b3= ((i & 0xFF)- 0x30);

temp=b0+b1+b2+b3;

utf_32 = temp + 0x10000;

surrogate high=(temp >> 10)+0xD800;

surrogate low=(temp & 0x3FF)+0xDC00;

For more detail see the section on Transformations.

Component GB18030 Tables

Introduction

Gb18030 contains 1, 2, and 4-byte code points and UTF-16 contains 2-byte code points. According to the z/OS logic, in order to do conversion from GB18030 to UTF-16, three binary tables are required. They are the following:

24E404B0.G1U-C0-A1 - Binary table from GB 1-byte part (9444) to UCS (1200)

(CDRA 1-byte to 2-byte conversion method, with X'FFFF' as STOP character)

256904B0.G2U-C0-A1 - Binary table from GB 2-byte part (9577) to UCS (1200)

(CDRA 2-byte to 2-byte conversion method, with X'FFFF' as STOP character)

156F04B0.G4U-C0-A1 - Binary table from GB 4-byte part (5487) to UCS (1200)

(CDRA 4-byte to 2-byte conversion method for GB18030, with X'FFFF' as STOP character)

GB18030 1-byte or 2-byte code points are mapped to the Unicode BMP (plane 0). The GB18030 4-byte code points that map beyond the BMP can be transformed into UTF-16 (surrogate high and low pair) or be represented as a 4-byte UTF-32 code using pure calculations which will be described in detail later.

The high level logic is:

First, the GB18030 input stream (sequence of 1, 2 and 4-bytes) will be fed into the conversion logic. Within the conversion logic, the GB18030 input stream will be passed through the following logic:

- 1:2 logic use GB18030 single-byte to UTF-16 binary table to get UTF-16 double-byte output (see 1-byte to 2-byte Conversion for more details)

- 2:2 logic use GB18030 double-byte to UTF-16 binary table to get UTF-16 double-byte output (see 2-byte to 2-byte Conversion for more details)

- 4:2 logic detects valid four-byte GB18030 code which can be transformed to a valid surrogate pair, using the transformation technique (see 4-byte to 2-byte Conversion for more details).

- 4:2 logic use GB18030 four-byte to UTF-16 binary table to get UTF-16 double-byte output (see 4-byte to 2-byte Conversion for more details)

- All invalid and incomplete sequences are dealt with separately.

The single-byte to UCS-2 array consists of a single 512-byte vector (256 2-byte entries).

The double-byte GB to UCS-2 array is made up of several 512-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see 2-byte to 2-byte Conversion for more details).

The four-byte to UCS-2 compact array is made up of:

- 256 bytes of flag values (followed by 3*256 bytes of unused bytes)
- Four 1024-byte long vectors, each containing 256 4-byte index values, and
- A long compact array containing target two-byte UCS-2 code points.

GB18030 to UTF-16 Conversion Logic

The following model is similar to the one described in the conversion logic for conversions from UTF-16 to GB18030 except the reverse direction binary tables are used in places where forward direction binary tables were used in the section mentioned above.

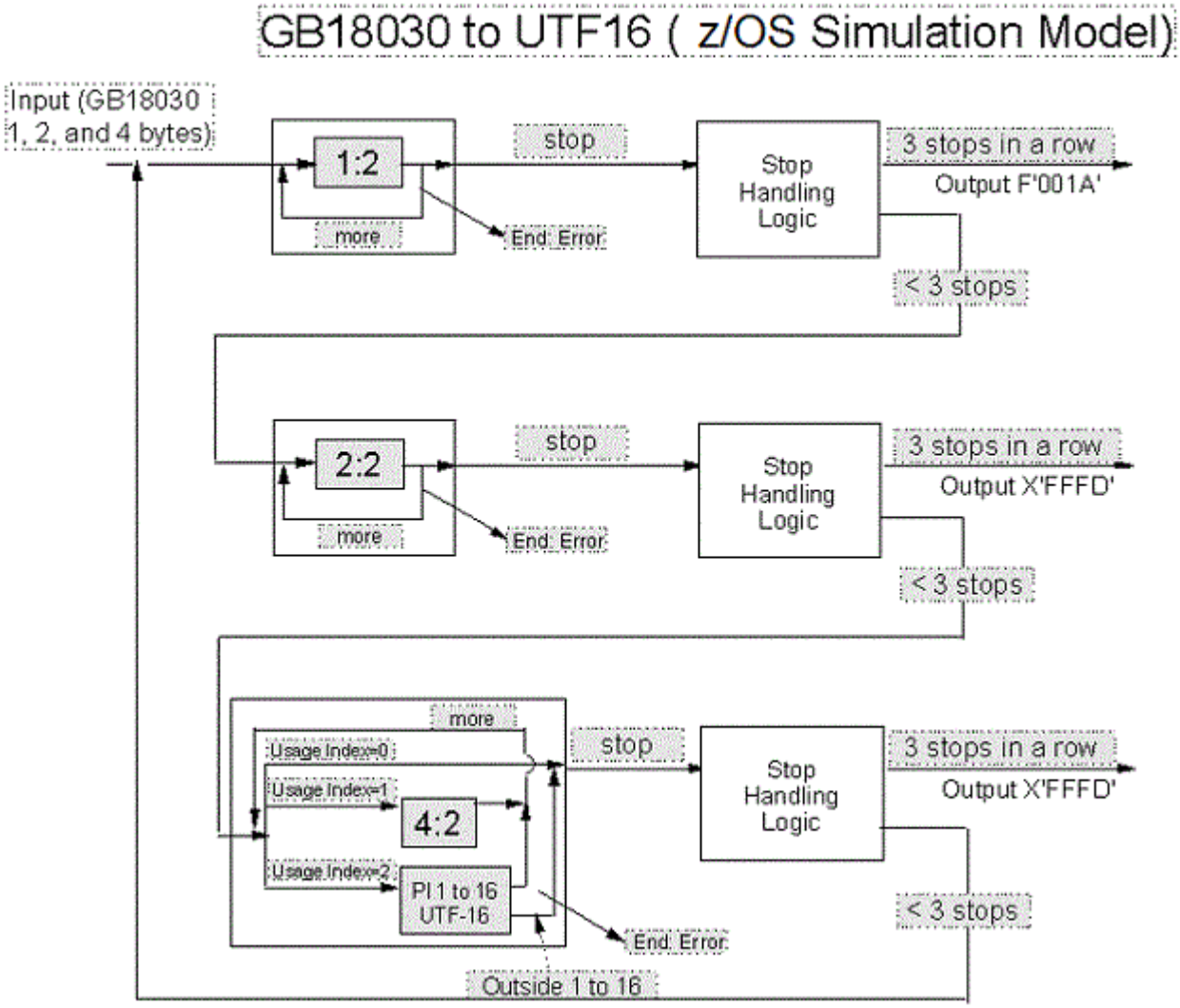

Figure 1.13 - GB18030 to UTF-16 Conversion Method

In understanding the GB18030 to UTF-16 z/OS conversion method, we refer to figure 1.13 above in order to obtain a graphical representation of the following description. Upon first glance at figure 1.13, note that there exists 3 binary tables for dealing with 1:2 byte, 2:2 byte and 4:2 byte mappings respectively. The presence of input that has no existing mapping within any of the three tables can be detected and

substituted upon passing input through the three stages. The precise means by which this is accomplished is described later in detail.

In this model, first the GB18030 input stream will be fed into the conversion logic. The conversion logic initially starts with the 1:2 stage of operation. In the step, the GB18030 input stream will be run through the 1:2 binary table by fetching one byte at a time from the input stream until it hits a stop character in the binary table or End/Error condition. In the case of an End/Error condition, execution will be terminated.

In the case that a stop character is encountered (X'FFFF' in this case) the stop handling logic will be executed. Then the execution goes to the next step in the process, which is 2:2 stage of operation.

In the 2:2 stage of operation, the remaining GB18030 input stream will be passed through the 2:2 binary table by fetching two bytes at a time from the input stream until it hits a stop character or End/Error condition. In the case of an End/Error condition, execution will be terminated.

In the case that a stop character is encountered (X'FFFF' in this case) the stop handling logic will be executed. Then the execution goes to the next step in the process, which is 4:2 stage of operation in this case.

In the 4:2 stage of operation, the remaining GB18030 input stream will be passed through the 4:2 conversion logic by fetching four bytes at a time from the input stream. Before attempting to find a target code point for an input 4 byte code point (b0 b1 b2 b3), the b0 value is checked to confirm that it has been used. This is checked by using the b0_used_array (see 4-byte to 2-byte conversion). If the entry corresponding to the b0 in b0_used_array is a 00, then the code point(b0 b1 b2 b3) is unassigned and its mapping is not defined in the conversion logic. This will be treated as hitting a stop character in the logic. If the entry corresponding to the b0 in b0_used_array is a 01, then the 4:2 binary table look up will take place by fetching 4 bytes at a time from the input stream. If the entry corresponding to the b0 in b0_used_array is a 02, then the GB18030 to UTF-16 transformation logic will be executed to output valid surrogate high and low pair. These three types of operations will take place until one of the three operations produce a stop character or End/Error condition. In the case of an End/Error condition, the execution will terminate.

In the case that a stop character is encountered (X'FFFF' in this case) the stop handling logic will be executed. Then the execution loops back to the starting step of the process, which is 1:2 stage of operation in this case. This loop of operations will go on until the end of input stream is reached.

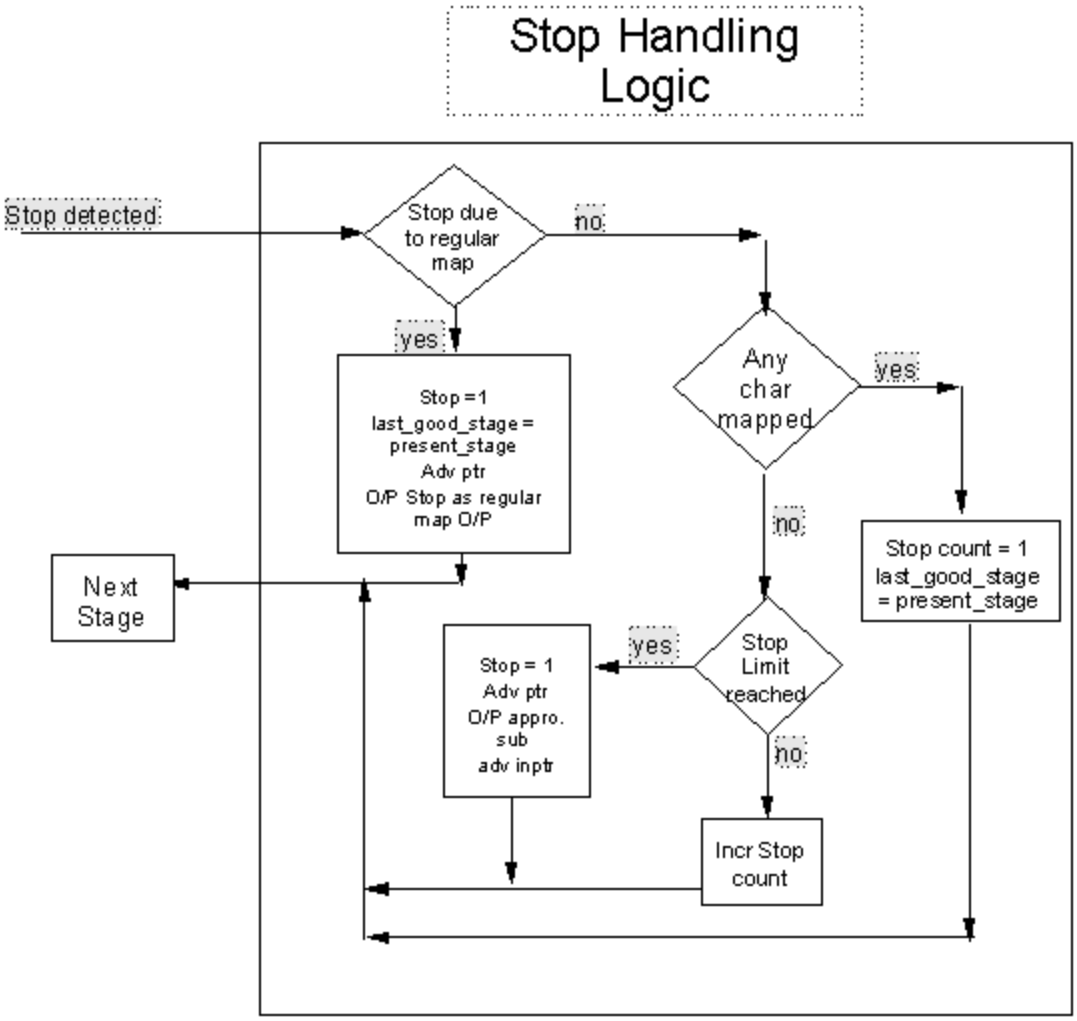

Figure 1.3 - **UTF-16 to GB18030 Conversion Methods**

The stop handling logic is illustrated in Figure 1.3 above (duplicate of Figure 1.3 presented here again for the reader's convenience). It is an integral part to the algorithm's ability to determine whether or not SUBs are to be output and to detect the presence of regular mappings. This task is accomplished with the assistance of a 'stop limit' counter that effectively keeps track of the number of successive stops in the algorithm and 'last_good_stage' flag. It also remembers the last successful stage of conversion, and takes appropriate action.

In the stop handling logic, the logic first checks whether this stop is due to a regular mapping or not. If it is due to a regular mapping, the stop count will be reset to 1 and the last successful stage of operation is also set to the present stage of operation. The input pointer will be moved to the next position in the input stream while the stop (X'FFFF') character goes to the output as a target code point.

An important point to be aware of in this logic is that stop handling logic is stage dependent. For example, GB18030 code point X'8437A439' maps to UTF-16 code point X'FFFF' which is same as the stop code point in this case. So the stop handling logic after 4:2 logic must check whether this stop is due to a regular mapping (for input X'8431A439') or not. This regular mapping could differ from stage to stage. This case has to be taken care of in the stop handling logic for each stage. (In the

OS390GB2U.C sample program this case has been taken care of in the 4:2 logic which is slightly different than what is described in here).

If the stop is not due to the regular mapping, then the stop handling logic checks whether any successful conversion has taken place in this stage of operation. This can be done by checking whether the input pointer has moved or not. If there is any successful conversion then the stop count will be reset to 1 and the last successful stage of operation is also set to the current stage of operation before we move on to the next stage of operation. If there is no mapping in this stage of operation then the logic checks whether the stop limit has been reached or not. If the stop limit is reached this indicates that each stage of the conversion operation (1:2, 2:2 and 4:2) has been executed without finding a successful mapping for the input code point. In this case a sub is output as target according to the last successful stage of conversion. The stop count will be reset to 1 now and input pointer also advanced to the next position.

If the stop limit is not reached, then the stop count will be incremented and the execution goes to the next stage of operation.

1-byte to 2-byte Conversion

The conversion method used for converting GB 1-byte to UTF-16 2-byte is identical to the previously mentioned method for converting GB 1-byte to UCS-2 2-byte. Please refer to 1-byte to 2-byte Conversion for GB to UCS-2 for details.

2-byte to 2-byte Conversion

The conversion method used for converting GB 2-byte to UTF-16 2-byte is identical to the previously mentioned method for converting GB 2-byte to UCS-2 2-byte. Please refer to 2-byte to 2-byte Conversion for GB to UCS-2 for details.

4-byte to 2-byte Conversion

The conversion method used for converting GB 4-byte to UCS-2 2-byte has previously been discussed in a prior section. Please refer to 4-byte to 2-byte Conversion for GB to UCS-2 for details.

Transformations

Transformation between GB18030 and UTF-32

In UTF-32 the first byte is always 00, and the second byte represents the plane number. Planes 0 through 16 are defined in UCS. GB18030 1-byte (all), 2-byte (all) and some of the 4-byte (X'81308130' to X'8431A439') code points fill all of the possible mapping points in UCS Plane 0. All of the UCS code points other than those found in BMP/Plane 0 (that is all the code points from Plane 1-16) are mapped to GB18030 4-byte code points only. No GB18030 1-byte or 2-byte code points are mapped to the Plane 1-16 code points.

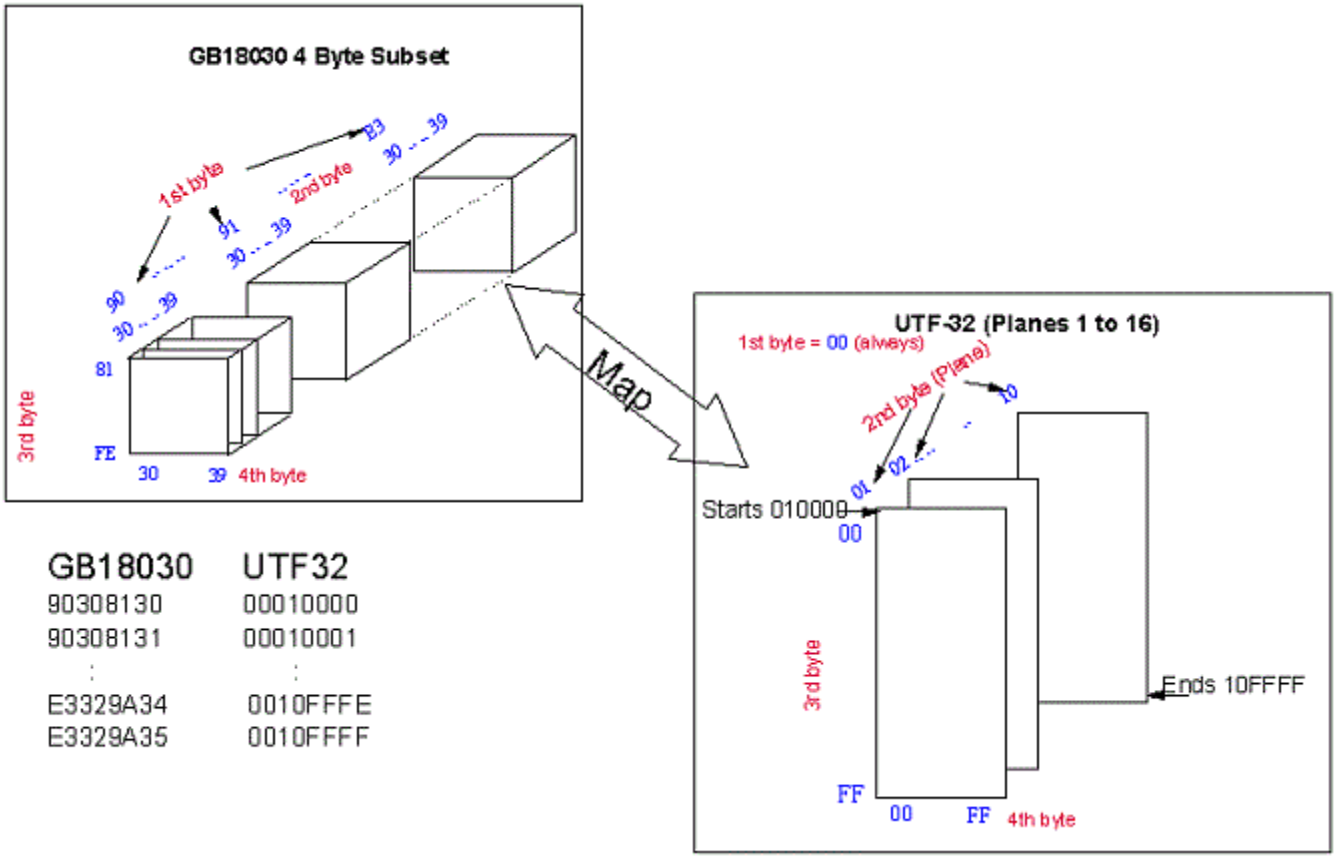

Figure 1.14: GB 4-byte to UTF-32 4-byte Mapping Structures

The mapping between GB18030 code points (4-byte subset) and the UCS code points from Plane 1-16 are done algorithmically. Starting from X'90308130' up to X'E3329A35', all the valid GB18030 4-byte code points are linearly mapped into the UCS code points in Planes 1-16 starting from X'00010000' up to X'0010FFFF' (Please refer to Figure 1.14).

GB18030	**UTF-32**
90308130	00010000
90308131	00010001
:	:
E3329A34	0010FFFE
E3329A35	0010FFFF

So given the valid GB18030 4-byte code point X'90308130', the corresponding UTF-32 value is calculated as follows:

Let 4 bytes of the GB18030 code point be b0 b1 b2 b3.

u0 = (b0 - X'90') * number of possible b1 * number of possible b2 * number of possible b3

u1 = (b1 - X'30') * number of possible b2 * number of possible b3

u2 = (b2 - X'81') * number of possible b3

u3 = (b3 - X'30')

UTF-32 code point = u0 + u1 + u2 + u3 + X'00010000'

Using the same principles for a UTF-32 code point in Plane 1 to Plane 16, the corresponding GB18030 value can be calculated as follows:

Let b0 b1 b2 b3 be the 4 bytes of GB18030 code point.

Let UTF-32 code point - X'00010000' = u0 u1 u2 u3

b0 = u0/(number of possible b1 * number of possible b2 * number of possible b3) + X'90'

b1 = u1/ (number of possible b2 * number of possible b3) % number of possible b1+ X'30'

b2 = u2/ (number of possible b3) % number of possible b2+ X'81'

b3 = u3 % number of possible b3 + X'30'

4-byte GB18030 = b0b1b2b3

(Note: * = multiplication; / = division; % = modular operation; - = subtraction; + = addition in the above equations)

Transformations between UTF-32 and UTF-16 Encoding Forms of Unicode

Each UTF-32 code point will be transformed into a UTF-16 surrogate pair as follows:

UTF-32	**UTF-16**
X'0000 0000'	X'0000'
:	:
X'0000 FFFF'	X'FFFF'
X'0001 0000'	X'D800 DC00' (see Figure 1.15 below)
:	:
X'0010 FFFF'	x'DBFF DFFF'
X'0011 0000' or greater	unmapped

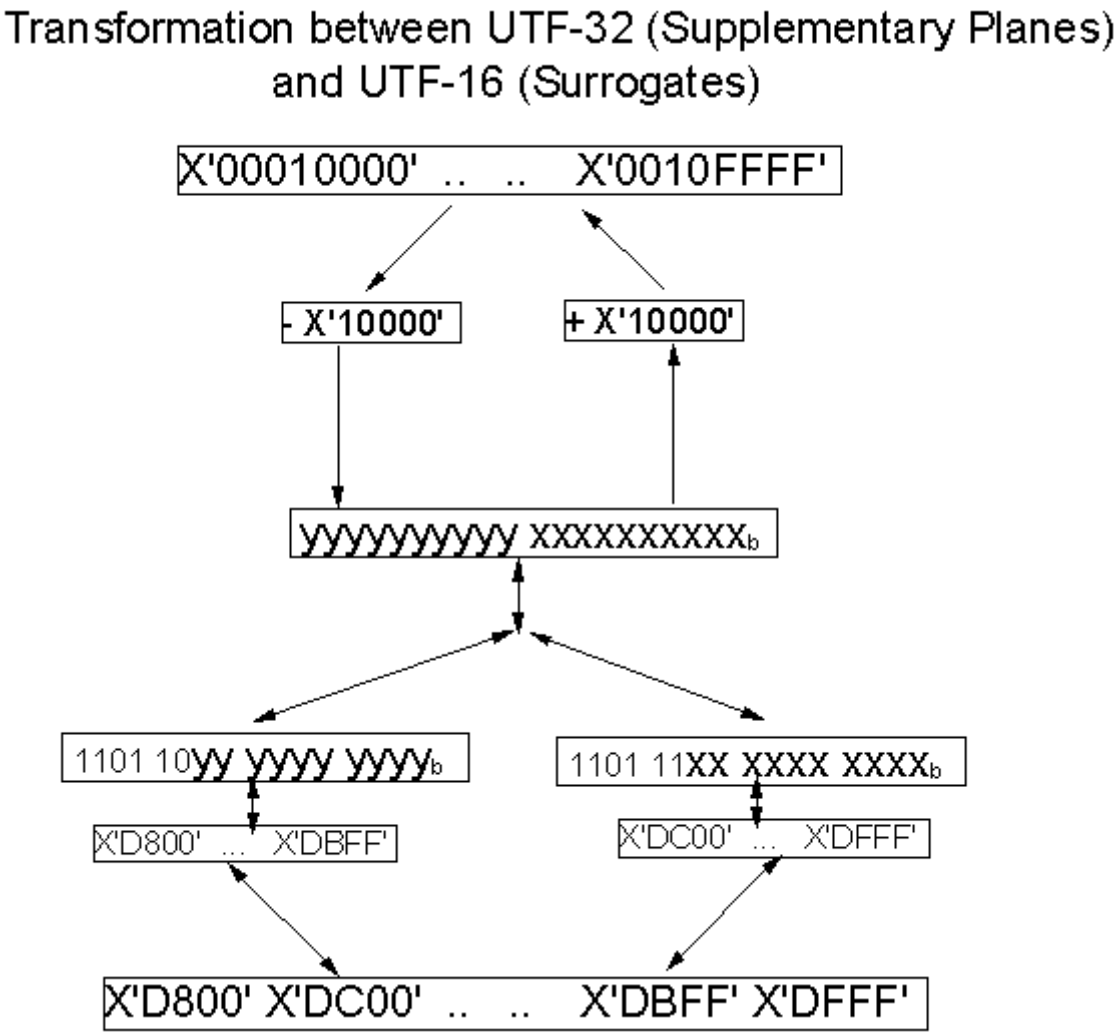

Figure 1.15 - UTF-32 (Supp. Planes) and UTF-16 (Surrogates)

Section 3: Conversions Between Unicode and Host Encodings

Conversion Between UCS-2 (CCSID 17584) <--> Host(1 and 2-byte, CCSID 1388)

This Section contains a description of the conversion tables and associated methods for converting data between S-Ch Host CCSID 1388 encodings and UCS-2 encodings.

Text Format Tables

Combined GB18030 Host Tables

056C44B0.TPMAP100 - S-Ch Host (CCSID 1388) To UCS-2 (CCSID 17584)

44B0056C.RPMAP100 - UCS-2 (CCSID 17584) To S-Ch Host (CCSID 1388)

056C44B0.UPMAP100 - S-Ch Host (CCSID 1388) To and FROM UCS-2 (CCSID 17584)

Component GB18030 Host Tables

334444B0.TPMAP100 - Text table from Host 1-byte part (4933) to UCS-2 (17584)

44B03344.RPMAP100 - Text table from UCS-2 (17584) to Host 1-byte part (13124)

334444B0.UPMAP102 - Text table between Host 1-byte part (13124) and UCS-2 (17584)

134544B0.TPMAP100 - Text table from Host 2-byte part (13124) to UCS-2 (17584)

44B01345.RPMAP100 - Text table from UCS-2 (17584) to Host 2-byte part (13124)

134544B0.UPMAP102 - Text table between Host 2-byte part (13124) and UCS-2 (17584)

Table files with the extension RPMAPnnn, TPMAPnnn, and UPMAPnnn contain human readable formats. They have two columns containing the source code point value (in Hex), and the target code point value (in Hex). Each file contains a brief header and column descriptions. Please ensure you read the header files for values of SUB used and any special handling required.

Conversion From UCS-2 (CCSID 17584) to S-Ch Host Extended (CCSID 1388)

Combined S-Ch Host Conversion

To be able to indicate whether a double-byte UTF-16 code point is mapped to a single-byte or double-byte code point in Host, in a binary conversion table, some indication will be needed as to the width of the target code point. This is accomplished by using a normalized Host code point in the conversion tables. Each output code point entry will have two bytes, single-byte Host code point values will be normalized by inserting a leading 00 byte. This permits a fixed width 2-byte to fixed width 2-byte conversion method and associated table structure to be used. The conversion logic strips out the leading zero-byte and adds any necessary code extension controls (Shift-IN/Shift-Out) before placing the value in the output stream. The following sections give more details.

2-byte To 2-byte Conversion

File: 44B0056C.UM-E-D

This binary table maps UCS-2 to normalized S-Ch Host Extended.

Method 8:

The binary conversion table is same to the tables used in existing CDRA Method 8. The input data stream consists of UCS-2 2-byte code points. The output from the conversion table will be 2-byte normalized Host code points. The resultant code points are de-normalized before being inserted into the output data stream.

Assumed normalization:

Host Byte	*Normalized*	*Comment*
xx (Single byte)	00xx	Two bytes, with 1 leading zero byte; SO/SI removed.
xxxx (Double byte)	xxxx	Two bytes

To describe the conversion method we first define the concept of a "ward". A ward is a section of a double-byte code page. It is equivalent to a "row" of code points in ISO/IEC 10646 and in Unicode. All of the code points contained in a specific ward begin with the same first byte. A ward is populated if there is at least one character in the double-byte code page (UCS-2 in this case) whose first byte is the ward value. There are 256 wards numbered X'00' to X'FF'.

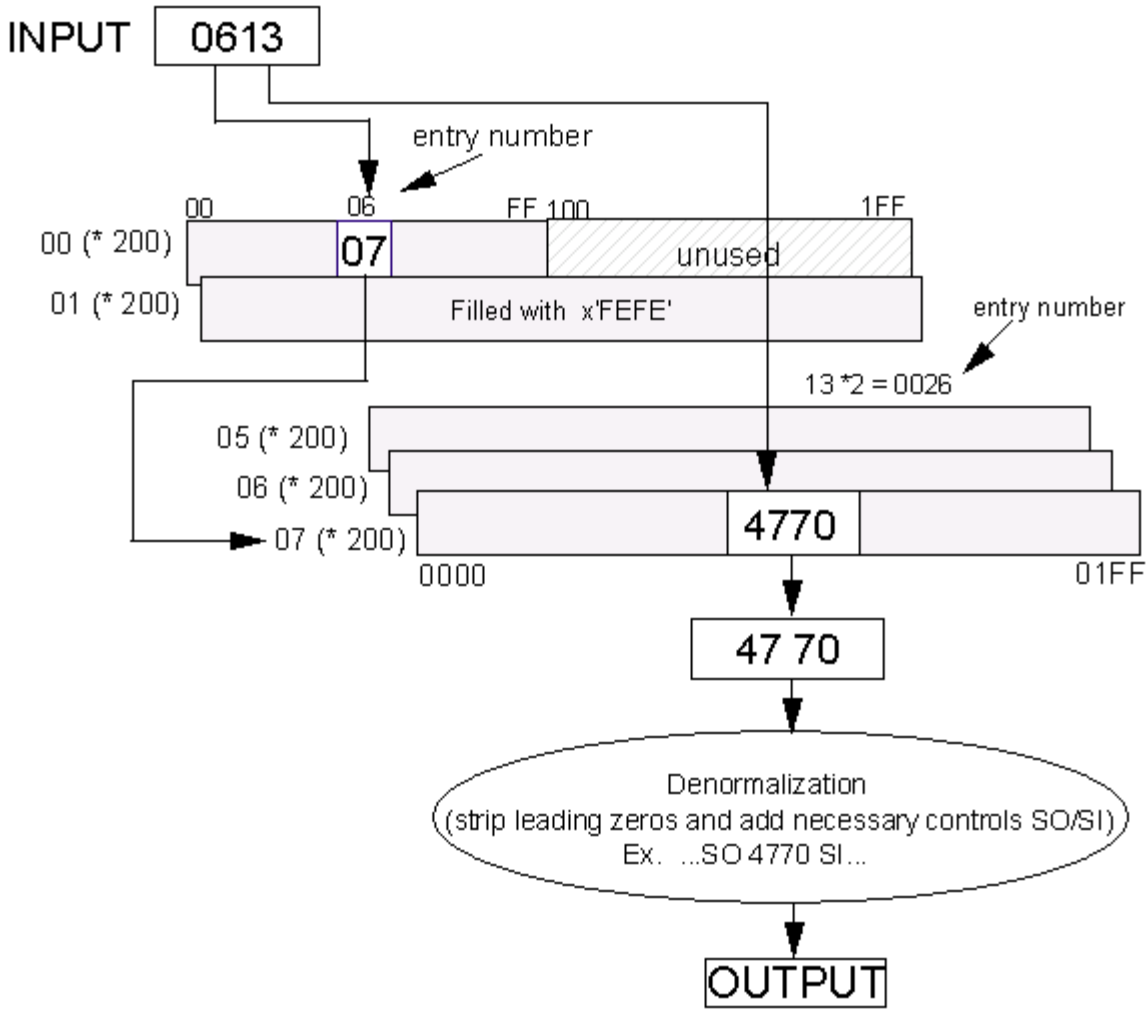

Figure 2.1 - **UCS-2 to S-Ch Host Extended - Binary Table and Conversion Method**

The double-byte binary conversion table is made up of several 512-byte vectors. The first vector contains 256 single-byte indices (vector numbers) into the rest of the table, followed by 256 unused bytes. The second vector, the "substitute" vector is used for mapping code points in unassigned wards, and is filled with 256 2-byte SUB code points (X'FEFE' in this case). These are followed by a collection of 512-byte vectors, one for each populated ward in the source code page.

The table is used as follows. The first byte of the input code point is used as a pointer into the index vector. The single-byte value found at the corresponding position in the index is the vector number in which to perform the second lookup. The second byte of the input code point is used as a pointer into the specified vector.

For example in Figure 2.1 the input code point is X'0613'. Using x’06’ as a pointer into the index vector, we find x’07’ as the specified vector in the table structure. Then the second byte, x’13’ is used to calculate the pointer value into the specified vector. The resultant double-byte output code point is found in vector number x'07' beginning at position X'0026' (which is two

times X'13'), counting into the vector starting at zero. In this example the output double-byte value is x'4770'.

The second vector mentioned for handling unassigned wards works as follows. All of the 256 double-byte code point values found in this vector (vector X'01') are those of the "Substitute (SUB)" character of the target code page (X'FEFE' for S-Ch Host Extended). All the entries in the index vector for unassigned wards point to this "substitute" vector.

The binary conversion table is the same format as the CDRA UM-E-D binary tables.

Denormalization:

Since resultant single-byte values in the table have been normalized with a leading zero-byte, when composing the output string the leading zero-bytes must be removed. In a properly formed Host Mixed data stream the single-byte strings and the double-byte strings must be bracketed with appropriate controls (SO/SI). The denormalization process must perform both tasks.

Example: SO 41 41 40 40 SI 41 SO 72 01 SI

Component S-Ch Host Conversions

Introduction

The z/OS converter does not use the combined tables. In the z/OS environment a call is made to CONVERT DATA tagged with CCSID 01388 to Unicode CCSID. The converter logic looks up 01388 information and gets the component conversion tables and sets up 2 to 1, 2 to 2 logic and resources and starts executing the conversion. The z/OS implementation needs the component tables not the combined ones. In this case two binary tables are required to perform the conversion. They are the following:

44B03344.US-C0-A1 - Binary table from UCS (17584) to Host 1-byte part (13124)
(CDRA 2-bytes to 1-byte conversion method, with X'3F' as SUB/STOP)

44B01345.UM-C0-A1 - Binary table from UCS (17584) to Host 2-byte part (4933)
(CDRA 2-byte to 2-byte conversion method, with X'FEFE' as SUB/STOP)

The high level logic is:

The UCS-2 input stream (sequences of double-bytes) will be fed into the conversion logic. Within the conversion logic, the UCS-2 input stream will be passed through the following logic:

- 2:1 logic use UCS-2 to S-Ch Host single-byte binary table to get S-Ch Host single-byte output (see the section on 2-byte to 1-byte Conversion for more details)

- 2:2 logic use UCS-2 to S-Ch Host double-byte binary table to get S-Ch Host double-byte output (see the section on 2-byte to 2-byte Conversion for more details)

The UCS-2 to single-byte S-Ch Host binary table consists of several 256-byte vectors. The first vector (00) acts as an index into the rest of the table and

the second vector (01) is the "substitute" vector (see the section on 2-byte to 1-byte Conversion for more details).

The UCS-2 to double-byte S-Ch Host binary table is made up of several 512-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see the section on 2-byte to 2-byte Conversion for more details).

UCS-2 to S-Ch Host conversion Logic

To understand the UCS-2 to S-CH Host z/OS conversion method, refer to figure 2.2 for a graphical representation of the following description. In figure 2.2, note that there exists 2 binary tables for dealing with 2:1 byte and 2:2 byte mappings respectively. The presence of input that has no existing mapping within either of the two tables can be detected and substituted upon passing input through the two stages. The precise means by which this is accomplished is described later in detail.

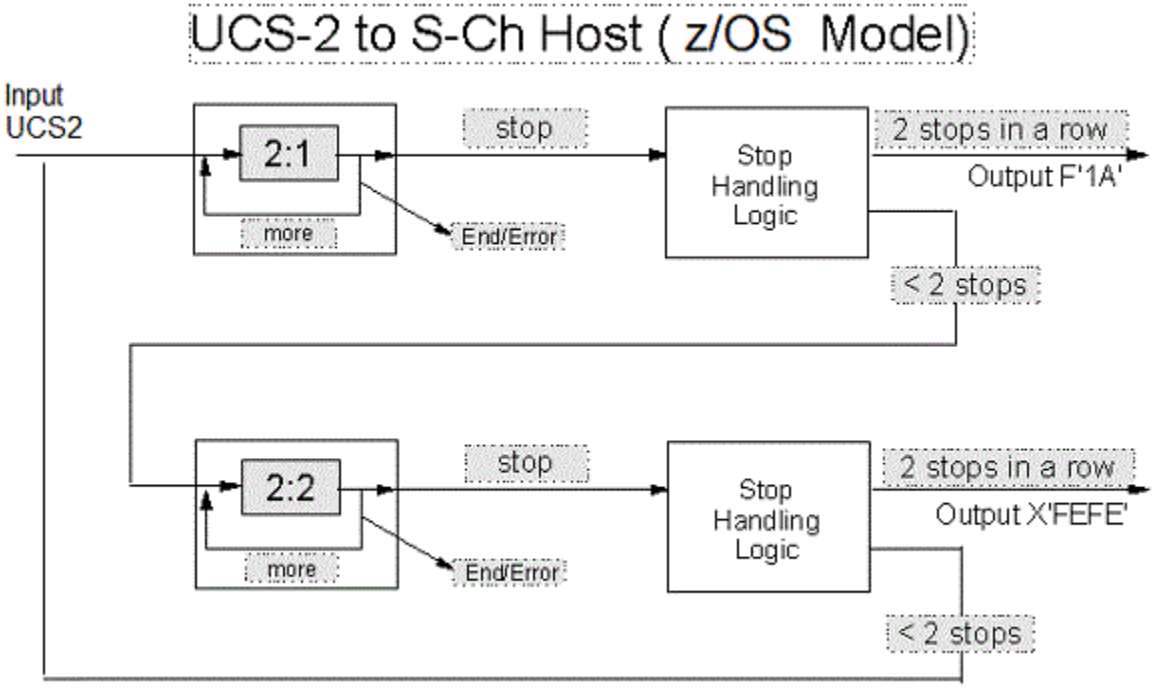

Figure 2.2 - UCS-2 to S-Ch Host Conversion Method

In this model, the UCS-2 input stream will first be fed into the conversion logic. The conversion logic initially starts with the 2:1 stage of operation. In the 2:1 stage of operation, the UCS-2 input stream will be run through the 2:1 binary table by fetching two bytes at a time from the input stream until it hits a stop character in the binary table or End/Error condition. In case of End/Error condition, execution will be terminated.

In the case that a STOP/SUB character is encountered (X'3F' in this case) the stop handling logic will be executed. Then the execution goes to the next stage of operation, which is 2:2 stage of operation in this case.

In the 2:2 stage of operation, the UTF-16 input stream will be run through the 2:2 binary table by fetching two bytes at a time from the input stream until it hits a stop character or End/Error condition. In case of End/Error condition, execution will be terminated.

In the case that a STOP/SUB character is encountered (X'FEFE' in this case) the stop handling logic will be executed. Then the execution goes to the next stage of operation, which is 2:1 stage of operation in this case.

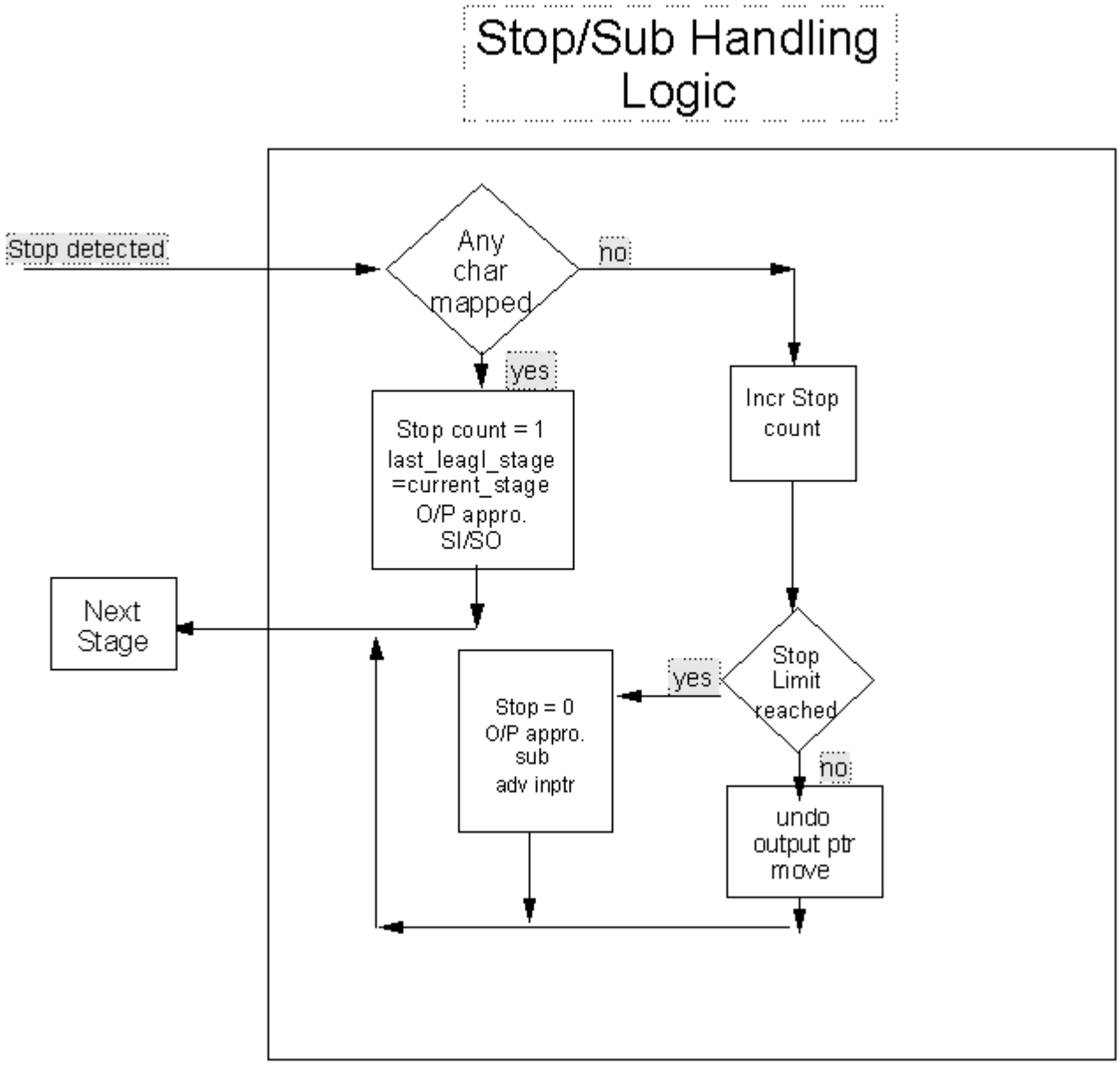

Figure 2.3 - **UCS-2 to S-Ch Host Conversion Methods**

The stop handling logic is illustrated in Figure 2.3 above. It is an integral part of the algorithm's ability to determine whether or not a SUB or an appropriate control character (SI/SO) is to be output when a STOP/SUB character is detected. This task is accomplished with the assistance of a 'stop limit' counter that effectively keeps track of the number of successive stops in the

algorithm and 'last_good_stage' flag which remembers the last successful stage of conversion.

In the stop handling logic, the logic first checks whether any successful conversion has taken place in this stage of operation. This can be done by checking whether the input pointer has moved or not. If there is any successful conversion, then the stop count will be reset to 1. The last successful stage of operation is also set to the current stage of operation and the appropriate control character (SI/SO) is placed in the output stream before we move on to the next stage of operation. If there is no mapping in this stage of operation then the stop count will be incremented and the logic checks whether the stop limit has been reached. If the stop limit has been reached then a sub is output as target according to the last successful stage of conversion. The stop count will be reset to 0 now and input pointer also advanced to the next position. If the stop limit is not reached, then and the execution goes to the next stage of operation after removing any inappropriate control characters (SI/SO).

2-byte to 1-byte Conversion

The conversion method used for converting UCS-2 to S-Ch single-byte is identical to the previously mentioned method for converting UTF-16 to GB18030 single-byte. Please refer back to the earlier section on 2-byte to 1-byte Conversion for details.

2-byte to 2-byte Conversion

The conversion method used for converting UCS-2 to S-Ch double-byte is identical to the previously mentioned method for converting UTF-16 to GB18030 double-byte. Please refer back to the earlier section on 2-byte to 2-byte Conversion for details.

Conversion From S-Ch Host Extended (CCSID 1388) to UCS-2 (CCSID 17584)

Combined S-Ch Host

This table is used to find the target code point when the source code point is single/double-byte. The method used to create this table requires that the input data is normalized such that each input code point is two bytes long. This is done by prefixing each single-byte code point with a zero-byte (X'00). Any code extension controls *(Shift-IN/Shift-Out)* are removed from the input data stream.

2-byte to 2-byte Conversion

File: 056C44B0.MU-R-D

Table format: The binary conversion table created by using the existing CDRA Method 7 - a two-step vector lookup method.

Method 7:

The conversion table and the associated method are illustrated in Figure 2.6 below.

The double-byte binary conversion table is made up of several 512-byte vectors. The first vector contains 256 single-byte indices (vector numbers) into the rest of the table, followed by 256 unused bytes. The second vector, the "substitute" vector is used for mapping code points in unassigned wards, and is filled with 256 2-byte SUB code points (X'FFFD' in this case). These are followed by 512-byte vectors, one for each populated ward in the source code page.

The table is used as follows. The first byte of the input code point is used as a pointer into the index vector. The single-byte value found at the corresponding position in the index is the vector number in which to perform the second lookup. The second byte of the input code point is used as a pointer into the vector specified by the index vector.

For example in Figure 2.6 the input code point is X'4760'. The first byte of the code point (x'47') is used as a pointer into the index vector, resulting in x'0A' being identified as the vector containing the output code point. The resultant double-byte output code point is found in the specified vector number (x'0A') by calculating the location from the second byte of the input code point. The second byte x'60' is multiplied by 2 (since each resultant code point is two bytes long). Thus the output code point will be found beginning at position X'00C0' (which is two time X'60'), in the specified vector. Following this process results in an output code point of x'0603'.

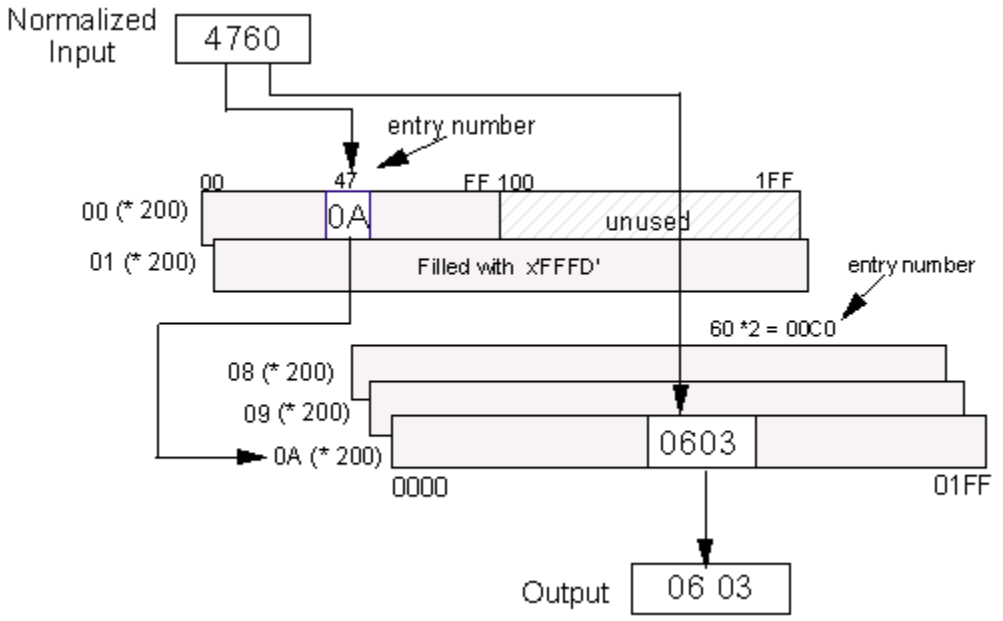

Figure 2.6: Normalized S-Ch Host Extended 2-byte to UCS-2 2-byte Table and Method

The second vector mentioned for handling unassigned wards works as follows. All of the 256 double-byte code point values found in this vector (vector X'01') are those of the "Substitute (SUB)") character of the target code page (X'FFFD' for UCS-2). All the entries in the index vector for unassigned wards point to this "substitute" vector.

The binary conversion table is the same format as CDRA MU-R-D binary tables.

Components S-Ch Host (z/OS Usage)

Introduction

S-Ch contains 1, and 2-byte code points and UCS-2 contains 2-byte code points. According to the z/OS logic, in order to do conversion from S-Ch to UCS-2, there are two binary tables needed. They are the following:

334444B0.MU-C0-A1 - Binary table from S-Ch Host 1-byte part (13124) to UCS-2 (17584)
(CDRA 1-byte to 2-byte conversion method, with X'FFFD' as STOP/SUB character)

134544B0.MU-C0-A1 - Binary table from S-Ch Host 2-byte part (4933) to UCS-2 (17584)
(CDRA 2-byte to 2-byte conversion method, with X'FFFD' as STOP/SUB character)

The high level logic is:

The S-Ch Host input stream (sequence of 1 and 2-bytes) will be fed into the conversion logic. Within the conversion logic, the S-Ch Host input stream will be passed through the following logic:

- 1:2 logic use S-Ch Host single-byte to UCS-2 binary table to get UCS-2 double-byte output

- 2:2 logic use S-Ch Host double-byte to UCS-2 binary table to get UCS-2 double-byte output

The single-byte to UCS-2 array consists of a single 512-byte vector (256 2-byte entries).

The double-byte GB to UCS-2 array is made up of several 512-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see section 1.3d for more details).

S-Ch to UCS-2 Conversion Logic

To understand the S-CH to UCS-2 Host z/OS conversion method, refer to figure 2.7 for a graphical representation of the conversion. Looking at figure 2.7, note that there exist 2 binary tables for dealing with 1:2 byte and 2:2 byte mappings respectively. The presence of input that has no existing mapping within any of the two tables can be detected and substituted upon passing input through the two stages. The precise means by which this is accomplished is described later in detail.

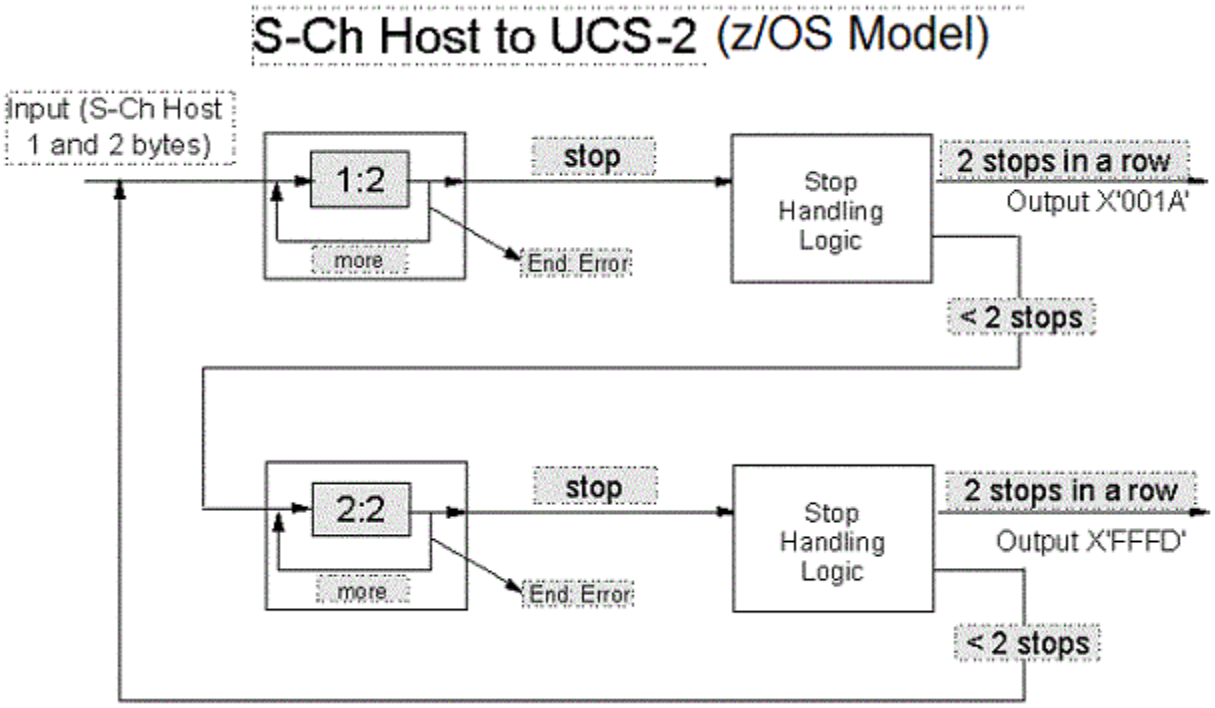

Figure 2.7 - S-Ch to UCS-2 Host Conversion Method

In this model, the S-Ch input stream will first be fed into the conversion logic. The conversion logic initially starts with the 1:2 stage of operation. In the 1:2 stage of operation, the S-Ch input stream will be run through the 1:2 binary table by fetching 1 byte at a time from the input stream until it hits a stop character in the binary table or End/Error condition. In case of End/Error condition, execution will be terminated.

In the case that a stop character is encountered (X'000E' in this case) the stop handling logic will be executed. Then the execution goes to the next stage of operation, which is 2:2 stage of operation in this case.

In the 2:2 stage of operation, the UTF-16 input stream will be run through the 2:2 binary table by fetching two bytes at a time from the input stream until it hits a stop character or End/Error condition. In case of End/Error condition, execution will be terminated.

In the case that a stop character is encountered (X'FFFD' in this case) the stop handling logic will be executed. Then the execution goes to the next stage of operation, which is 2:1 stage of operation in this case.

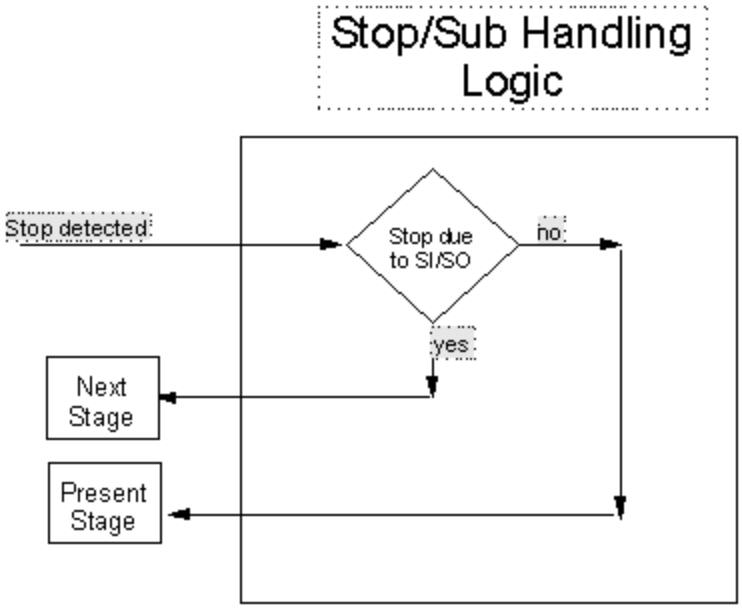

Figure 2.8 - S-Ch to UCS-2 Host Conversion Method

The stop handling logic is illustrated in Figure 2.8 above. Unlike the previous stop logic implementation, in performing UCS-2 to S-Ch conversions, the C-Sh to UCS-2 stop logic is much simpler. Its sole objective is to determine whether or not the stop was due to a control character (SI/SO). If it was due to a control character (SI/SO), then execution continues to the appropriate stage.

1-byte to 2-byte Conversion

The conversion method used for converting S-Ch Host 1-byte to UCS-2 2-byte is identical to the previously mentioned method for converting GB 1-byte to UCS-2 2-byte.

2-byte to 2-byte Conversion

The conversion method used for converting S-Ch Host 2-byte to UCS-2 2-byte is identical to the previously mentioned method for converting GB 2-byte to UCS-2 2-byte.

Section 4: Conversions Between GB18030 and Host Encodings

S-Ch Host Extended (CCSID 1388) <-> GB18030 (1,2 and 4-byte, CCSID 5488)

Introduction

This document contains description of conversion tables and associated methods for converting data between S-Ch Host extended CCSID 1388 encodings and GB 18030 (CCSID 5488) encodings. These (GB 18030 to and from Host) conversion tables use a normalized form of data. The Host code points are normalized by placing a leading zero-byte (X'00) in front of each single-byte to yield a two-byte form. Host data must have the SO-SI control characters deleted during normalization and reinserted afterwards during de-normalization. The GB code points are normalized to four-bytes by inserting leading zero-bytes for the single- and double-byte GB code point values. Figure 3.1 shows the use of the Host to and from GB18030 conversion tables.

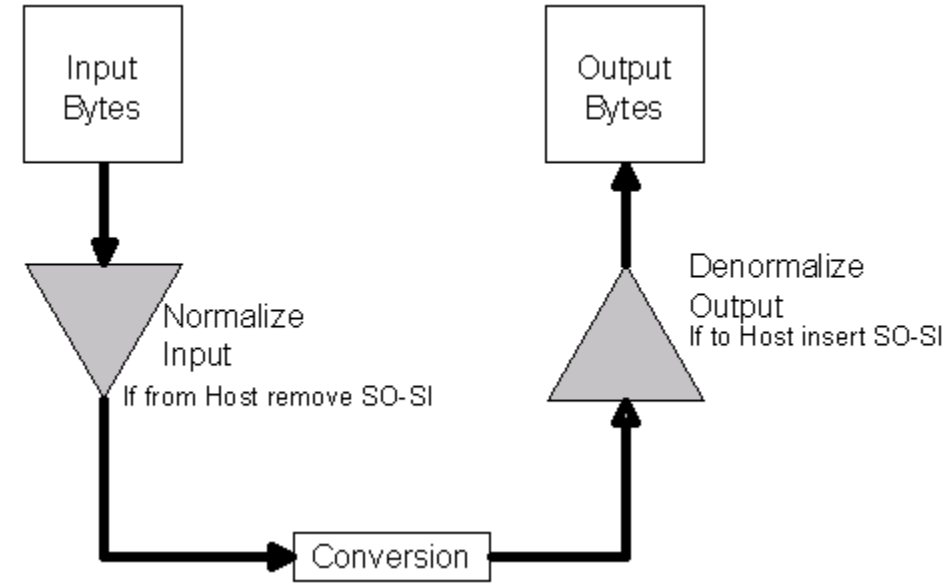

Figure 3.1 - Host and GB 18030 Conversion

The input byte or bytes, up to maximum of four bytes per code point, are first normalized and used as input to the conversion table. The output (again, four bytes per code point maximum) from the conversion table is also normalized data, which must be de-normalized prior to subsequent processing. Host data must have the SO-SI control characters deleted during normalization and reinserted afterwards during de-normalization.

Source mapping file between GB18030 and Host

File: HOST2GB.ALL (2000-12-07)

File: GB2HOST.ALL (2000-12-07)

Conversion From S-Ch Host (CCSID 1388) to GB18030 (CCSID 5488)

Combined GB18030 Tables

The GB code points are normalized to four-bytes by inserting leading zero-bytes for the single and double-byte GB code point values. The Host code points are normalized to two-bytes by inserting a leading zero-byte for the single-byte Host code point values.

2-byte to 4-byte Conversion

File 056C1570.MGN-R-D

This binary table contains the mapping from normalized Host to normalized GB18030.

Method 2x:

The binary conversion table is similar to the tables used in existing CDRA Method 2, but extended to handle the normalized GB18030-1 4-byte code. The input data stream consists of normalized Host 2-byte code points. The output from the conversion table will be 4-byte normalized GB code points which are de-normalized before being inserted into the output data stream.

Assumed normalization for GB18030-1:

GB Byte	*Normalized*	*Comment*
xx (Single byte)	000000xx	Four byte, with 3 zero byte leading zeros
xxxx (Double byte)	0000xxxx	Four byte, with 2 zero byte leading zeros
xxxxxxxx (Four byte)	xxxxxxxx	Four byte

To describe the conversion method we first define the concept of a "ward". A ward is a section of a double-byte code page. It is equivalent to a "row" of code points in ISO/IEC 10646 and in Unicode. All of the code points contained in a specific ward begin with the same first byte. A ward is populated if there is at least one character in the double-byte code page (normalized Host in this case) whose first byte is the ward value. There are 256 wards numbered X'00' to X'FF'.

This binary conversion table is made up of several 1024-byte vectors. The first vector acts as in index into the rest of the table. It contains 256 two-byte vector numbers (corresponding to each of the 256 wards) and the remaining 512 bytes are unused (filled with zeros). There is one 1024-byte vector for each populated ward in the source code page and one additional vector used for mapping all unassigned and invalid wards.

The method fetches two bytes at a time from the input data stream. The first byte is used as a pointer into the index vector -- as shown in Figure 3.2. Each vector number in the index vector is two bytes long. Therefore the first byte from the input code point is multiplied by two before calculating the offset into this index vector. The two-byte value found at the corresponding position in the index vector gives the vector number in which to perform the second lookup.

The second byte of the input code point is used as a pointer into the vector specified by the index vector. When calculating the offset into this vector

there are two things to remember -- first, the indexing starts at zero, and second, each entry is four bytes long (normalized GB18030 code point).

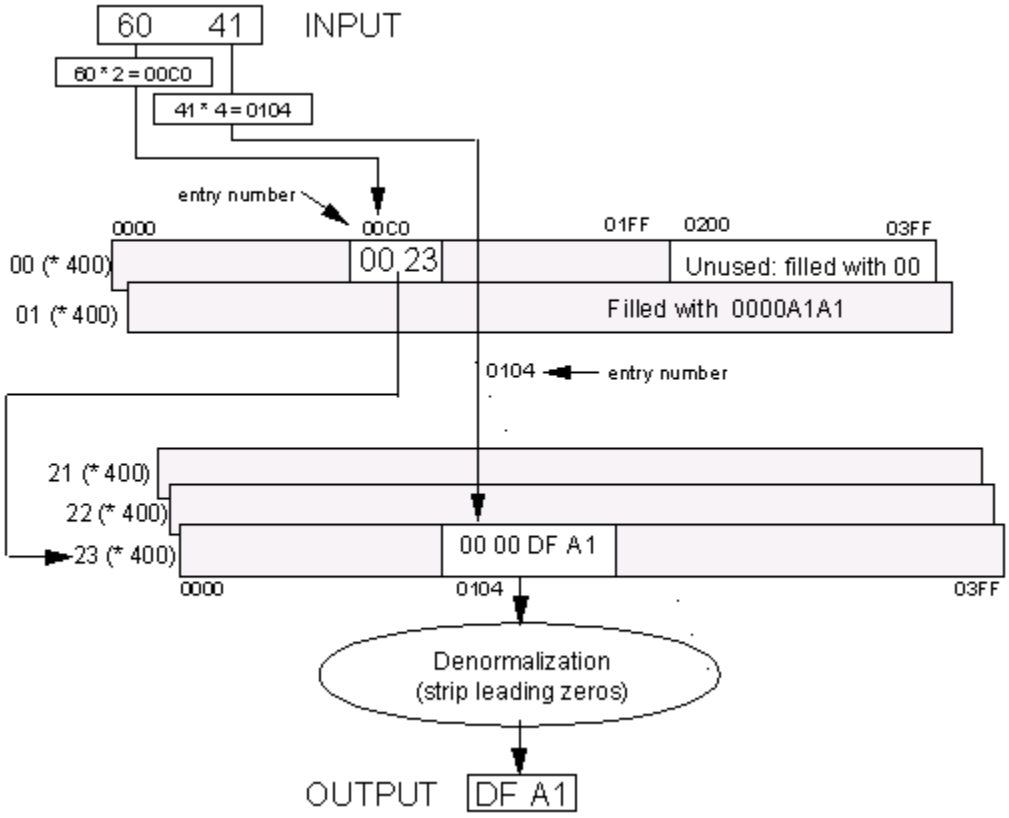

Figure 3.2 -S-Ch Host to GB18030-1 - Binary Table & Conversion Method

For example in Figure 3.2 the input code point is X'6041' you would find the two-byte vector number (X'0023') starting at byte position X'00C0' (which is two times X'60') in the index vector. The resultant four-byte output code point would be found in the specified vector number (X'0023') beginning at byte position X'0104' (which is four times X'41').

The one additional vector (X'0001') mentioned for handling code points from invalid or unassigned wards in the input data is used as follows. All of the 256 four-byte (normalized) code point values found in this vector are those of the "Substitute" (SUB) character of the target code page (X'0000A1A1' for GB18030-1). All of the entries in the index vector for unused and invalid wards point to this "substitute" vector.

In the binary table, target four-bytes could be found at the following positions.

Byte	Value	Position (hex)	Position (decimal)
first	0	X'0023' * X'400' + X'0104'	35 * 1024 + 260 = 36100
second	0	X'0023' * X'400' + X'0104' + 1	35 * 1024 + 260 + 1 = 36101

third	DF	X'0023' * X'400' + X'0104' + 2	35 * 1024 + 260 + 2 = 36102
fourth	A1	X'0023' * X'400' + X'0104' + 3	35 * 1024 + 260 + 3 = 36103

Denormalization:

Since the resultant single-bytes and double-bytes in the table have been prefixed with leading zero-bytes, when composing the output string the leading zero-bytes must be removed from the 4-byte output (three zero-bytes for single-byte and two zero bytes for double-byte).

Conversion From GB18030 (CCSID 5488) to S-Ch Host (CCSID 1388)

Combined GB18030 Tables

Following is a proposal for GB to HOST conversion, as an alternative to following the CDRA's EUC normalization method and the resulting table structure. The normalization steps for detecting when input is a single, double, or four-byte are followed. However, instead of normalizing the data as in the EUC case, the assumption here is to take three branches in the logic and come up with associated data structures. The data structures are linear arrays, one for each of the input types; single, double and four-byte. The indexing operations get into these arrays only for valid ranges of code points. All other code points and broken multi-byte sequences are trapped and are dealt with separately in the logic.

The high level logic is:

- Detect valid single, double or four-byte code points

- Use single-byte to HOST array for single-byte input

- Use double-byte to HOST array for double-byte input

- Use four-byte to HOST compact array for four-byte input

- All invalid and incomplete sequences dealt with separately

The main advantage of this method is to be able to keep the conversion tables as compact as possible (especially for the 4-byte to 2-byte part) and be able to get at the converted HOST code points in a relatively fast manner.

The single-byte to HOST array consists of a single 512-byte vector (256 Single-byte entries with a leading zero byte).

The double-byte GB to HOST array is made up of several 512-byte vectors. The first vector (00) acts as an index into the rest of the table and the second vector (01) is the "substitute" vector (see section 1.3d for more details).

The four-byte to HOST compact array is made up of:

- One vector, 1024 bytes long, first 256 bytes contain flag values, remaining 768 bytes are unused.

- Four vectors, 1024 bytes long, each containing 256 4-byte index values, and
- A long compact array containing all of the target two-byte HOST code points.

Detect a Valid Single-, Double-, Four-byte or Invalid code point

Refer to Figure 3.3 below. The method fetches one byte at a time from the input stream. The first byte, b0, is checked to see whether it is in the valid range of single-byte or start of a double-byte or four-byte code point. If it is not, then a SUB code point (X'003F' in this case) is inserted in the output data stream, and the pointer to the input stream is incremented by one.

If b0 is in the valid range of single-byte (X'00' to X'80'), then it is passed to the 1-byte to 2-byte conversion as a valid single-byte code point. The resultant two-byte output is placed in the output buffer, and the pointer into the input stream is incremented by one.

If b0 is in the range X'81' to X'FE' for a valid first byte of a double-byte or four-byte code point, then the next byte, b1, is checked to see whether it is in the valid range for a second byte of a double-byte (X'40' to X'7E' or X'80' to X'FE') or a second byte of a four-byte (X'30' to X'39') code point. If it is not, then the SUB code point X'003F' is inserted into the output stream and the pointer into the input stream is incremented by one. The pointer should point to byte b1 now. The first byte b0 is considered to be the start of a broken sequence of bytes in the input.

If b1 is within the valid range for a second byte of a double-byte code point (X'40' to X'7E' or X'80' to X'FE'), the sequence b0 b1 (or the input pointer) is passed to the 2-byte to 2-byte conversion. The resultant two-byte output is placed in the output buffer, and the pointer into the input stream is incremented by two.

If b1 is within the valid range for a second byte of a four-byte code point (X'30' to X'39'), then the next byte in the input stream, b2, is checked to determine whether it is in the valid range for a third byte of a four-byte code point (X'81' to X'FE'). If it is not, then the SUB code point X'003F' is inserted into the output stream and the pointer into the input stream is incremented by one. The pointer should point to byte b1 now. The first byte b0 is considered to be the start of a broken sequence of bytes in the input.

If b2 is within the valid range for a third byte of four-byte code point, the next input byte, b3, is checked to determine whether it is in the valid range for the fourth byte (X'30' to X'39'). If it is not, then the SUB code point X'003F' is inserted into the output stream and the pointer into the input stream is incremented by one. The pointer should point to byte b1 now. The first byte b0 is considered to be the start of a broken sequence of bytes in the input.

If b3 is within the valid range for the fourth byte then the sequence of bytes, b0 b1 b2 b3, (or the input pointer) is passed to the 4-byte to 2-byte conversion as valid four-byte code point.

The resultant two-byte output is placed in the output buffer, and the conversion pointer into the input stream is incremented by four.

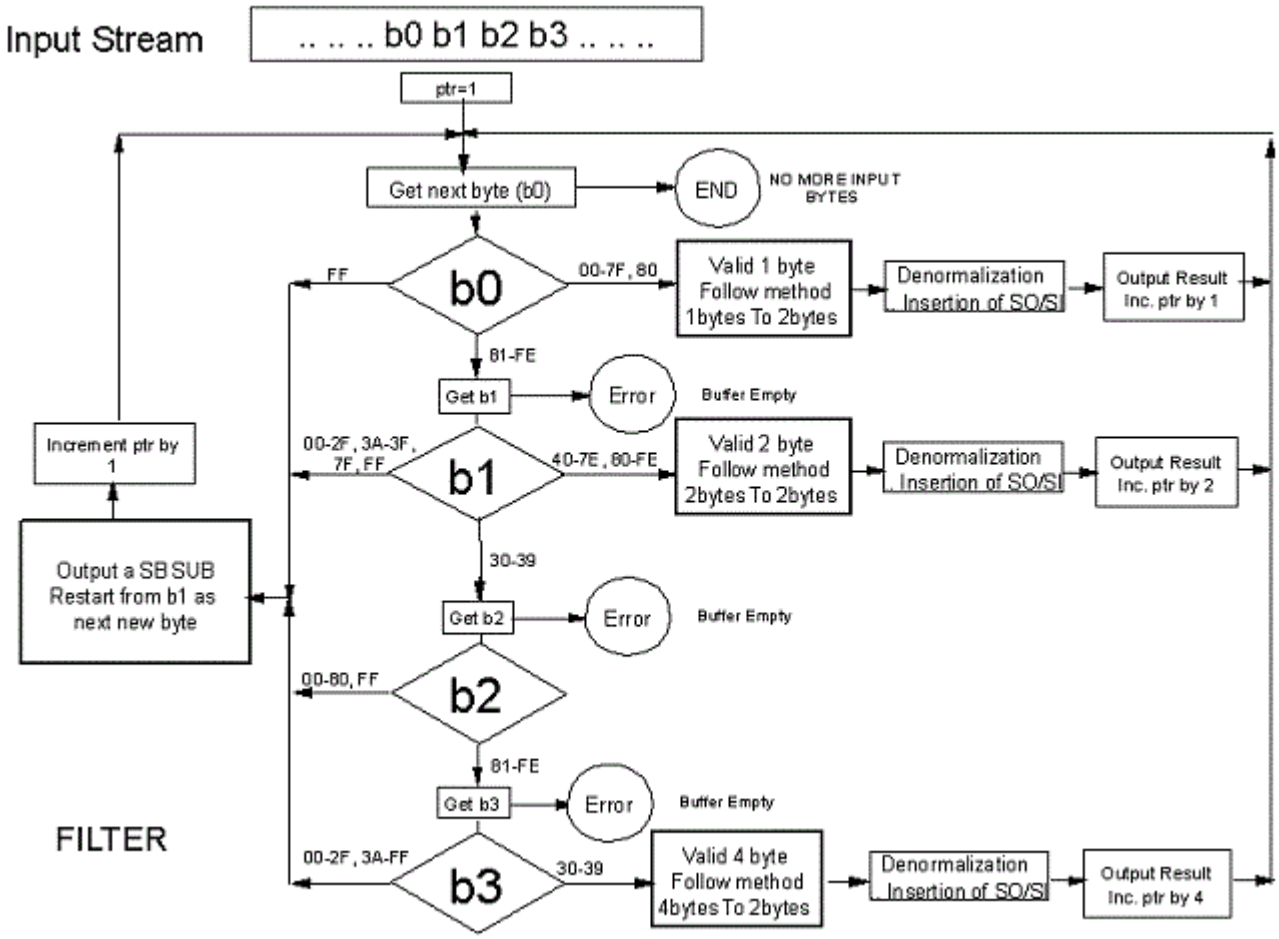

Figure 3.3: Input Filter for GB18030-1 stream

The above steps are repeated until there is no more data in the input stream. If the input stream is exhausted during fetching of any of second, third or fourth bytes in the above filtering logic, the conversion logic cannot proceed, and there will be some unconverted data in the input stream. The pointer into the input stream indicates the first of the remaining bytes that have not been converted. An implementation may choose to deal with such a situation in any suitable manner.

The insertion of X'003F' for the first byte of a broken sequence permits locating where such a sequence might have been in the input by examining the output stream.

Denormalization:

Since the resultant single-bytes in the table have been prefixed with a leading zero-byte, when composing the output string the leading zero-

byte must be removed. Also, all the single-byte sub-strings and the double-byte sub-strings must be bracketed with appropriate controls (SO/SI).

Example: SO 41 41 40 40 SI 41 SO 72 01 SI

1-byte to 2-byte Conversion

File: 1570056C.G1M-R-D

This table with its associated method is used to find the target code point when the source code point is a valid single-byte.

Table format:

The binary conversion table is created by using the existing CDRA Method 5. Figure 3.4 illustrates the table format and the associated method.

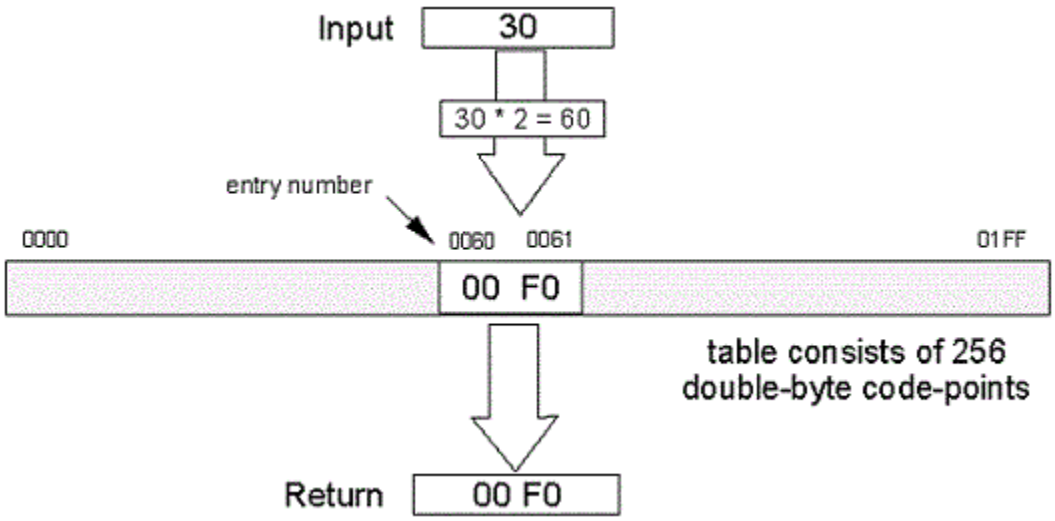

Figure 3.4: GB 1-byte to normalized S-Ch HOST 2-byte Table and Method

The single-byte to normalized HOST table consists of a single 512-byte vector (256 2-byte entries). The source code point is used as a pointer to determine which 2 bytes in the vector represent the target code point. Each target code point in the vector is two bytes long. Therefore the input code point is multiplied by two before calculating the offset into this vector. The binary conversion table is equivalent to existing CDRA SU-R-D binary tables.

2-byte to 2-byte Conversion

File: 1570056C.G2M-R-D

This table is used to find the target code point when the source code point is a valid double-byte.

Table format:

The binary conversion table is created by using CDRA Method 2, a two-step vector lookup method.

The conversion table and the associated method are illustrated in Figure 3.5 below.

This 2-byte to 2-byte table is made up of several 512-byte vectors. The first vector contains 256 single-byte indices (vector numbers) into the rest of the table, followed by 256 unused bytes. The second vector, the "substitute" vector contains the mapping for code points in unassigned wards, and is filled with 256 2-byte SUB code points (X'FEFE' in this case). These are followed by one 512-byte vector for each populated ward in the source encoding.

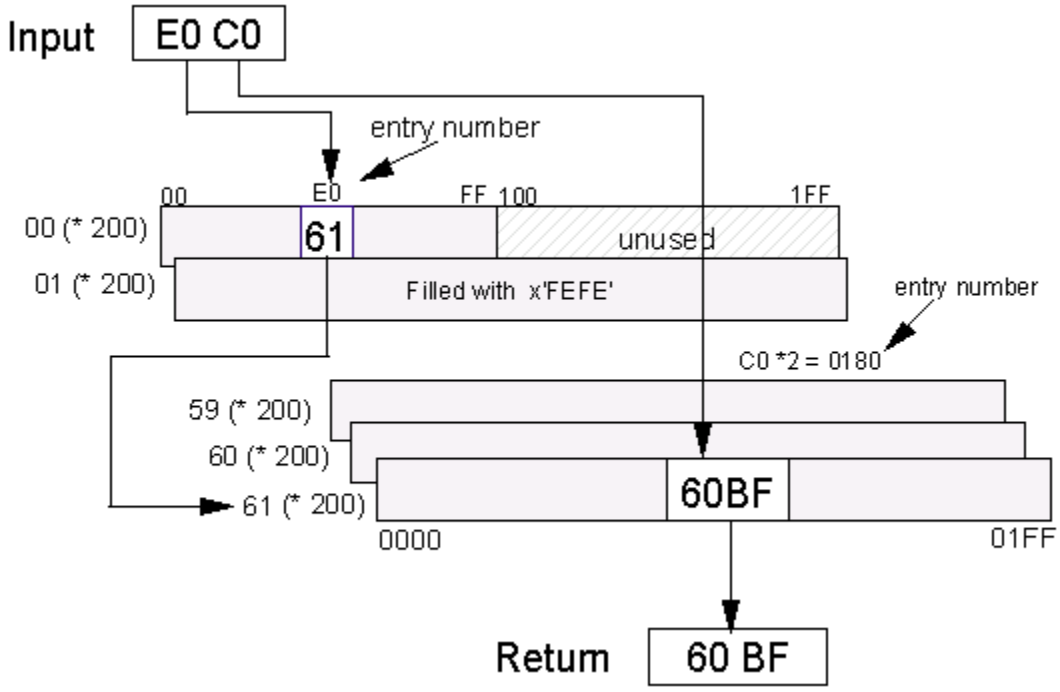

Figure 3.5: GB 2-bytes to normalized S-Ch HOST 2-bytes Table and Method

The table is used as follows. The first byte of the input code point is used as a pointer into the index vector. The single-byte value found at the corresponding position in the index is the vector number in which to perform the second lookup. The second byte of the input code point is used as a pointer into the vector specified by the index vector.

For example in Figure 3.5 the input code point is X'E0C0', you would find a vector number at the X'E0' position in the index vector. The resultant double-byte output code point is found in the specified vector number (x'61') beginning at position X'0180' (which is two times X'C0'), counting into the vector starting at zero.

The vector for handling unassigned wards works as follows. All of the 256 double-byte code point values found in this vector (vector X'01') are those of the "Substitute (SUB)" character of the target encoding (X'FEFE' for S-Ch Host). All the entries in the index vector for unassigned wards point to this "substitute" vector.

The binary conversion table is equivalent to existing CDRA MU-R-D binary tables.

4-byte to 2-byte Conversion

File: 1570056C.G4M-R-D

Table format:

The binary conversion table format and the associated method are detailed below.

This binary table consists of five 1024-byte vectors followed by a large linear array corresponding to the table structure as shown in Figure 3.6. There will be one three-dimensional sub array for each first byte (b0) value used in the table definition. Each sub array will contain the minimum number of cells -- 12600 cells -- needed to map the valid ranges of the second (b1, 10 values), the third (b2, 126 values) and the fourth (b3, 10 values) bytes of four-byte code points in GB. Each cell contains the corresponding 2-byte normalized HOST code point.

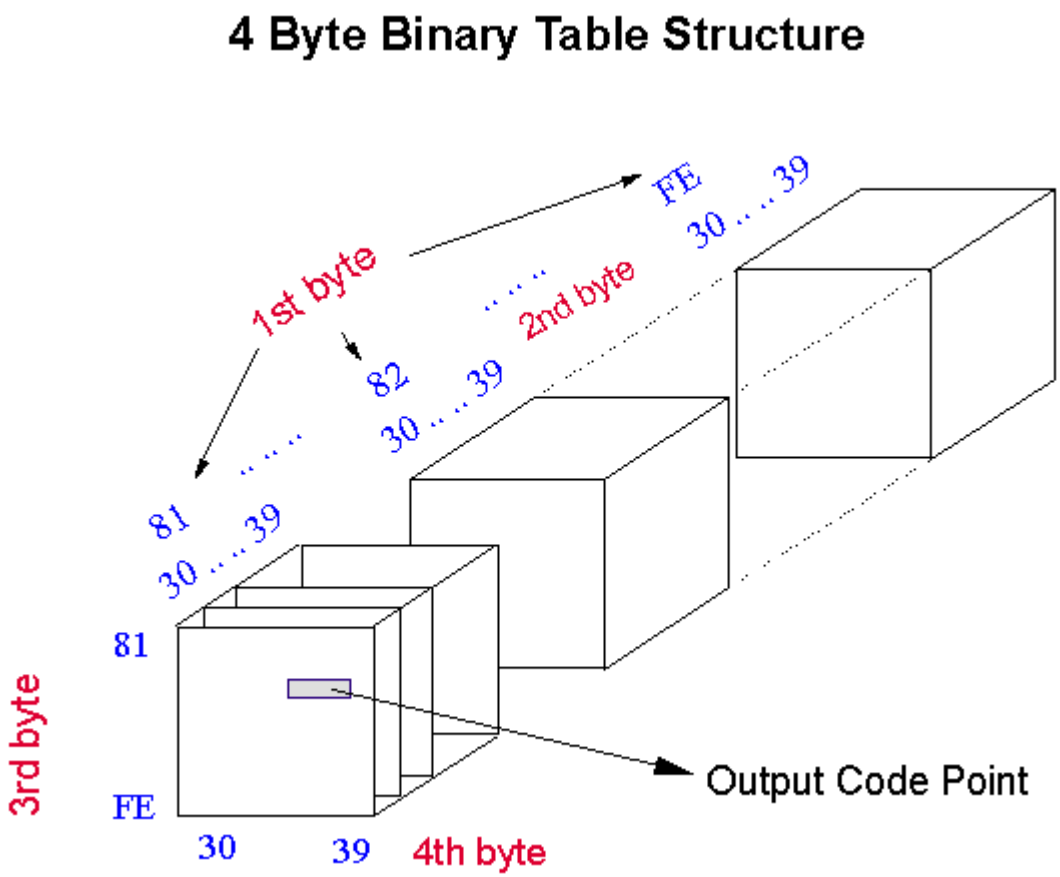

Figure 3.6: GB 4-byte to S-Ch HOST 2-byte table structure

The first 256 bytes of this binary table, called the b0_used_array, is used to check if a particular value of b0 byte is used in the conversion table definition. The next 768 bytes (=3*256) are unused (for now, may change later). These are followed by four 1024-byte vectors -- K40, K41, K42, and K43. These vectors contain values used in computing an index into the rest of the table to get at the output code point.

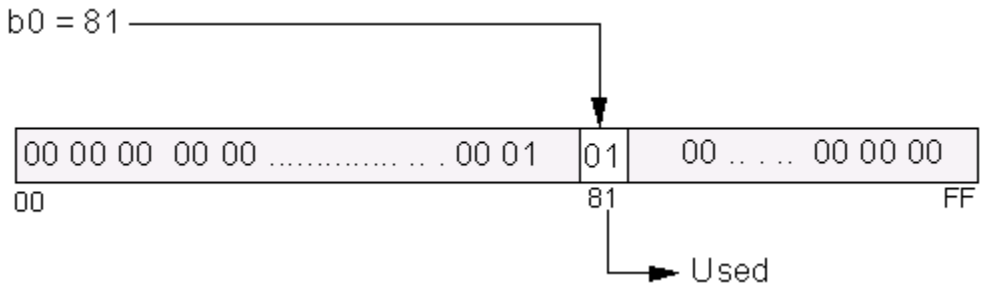

Figure 3.7: b0_used_array

For a given 4-byte code point (b0 b1 b2 b3), if the b0 is not used then there will not be a sub array populated for it. Therefore, before trying to find the target code point, the b0 value is checked to confirm that it has been used. This is done using the b0_used_array shown in Figure 3.7. If the entry at the appropriate location in the b0_used_array is 01 it is valid, if it is 00 it is not used. If it is a 00, then the code point (b0 b1 b2 b3) is unassigned and its mapping is not defined in the conversion table, a double-byte SUB code point (X'FEFE') is inserted into the output data stream. The pointer into the input stream is incremented by 4 in the main filter logic described earlier

If b0 has been used, then proceed with the steps to find target code point, as illustrated in Figure 3.8, and described below.

Get the computed index values from the four index vectors, K40, K41, K42, and K43 using b0, b1, b2, and b3 byte values of the input code point respectively. Each byte of the valid four-byte source code point is used as an offset into the corresponding K4x table to get a part of the index value. When calculating the offset into these vectors there are two things to remember; first, you must begin counting at zero, and second, each entry is four bytes long. This means you have multiply the b0, b1, b2, or b3 by four before looking into the vectors. The resulting 4 values are added to get the location of the first byte of the target code point in the binary table.

The values in the index vectors are computed based on the following formulae:

K40(b0) = 25200 * (block number assigned to the b0 group)

K41(b1) = 2* (b1 - X'30') * (126 * 10)

K42(b2) = 2* (b2 - X'81') * 10

K43(b3) = 2* (b3 - X'30')

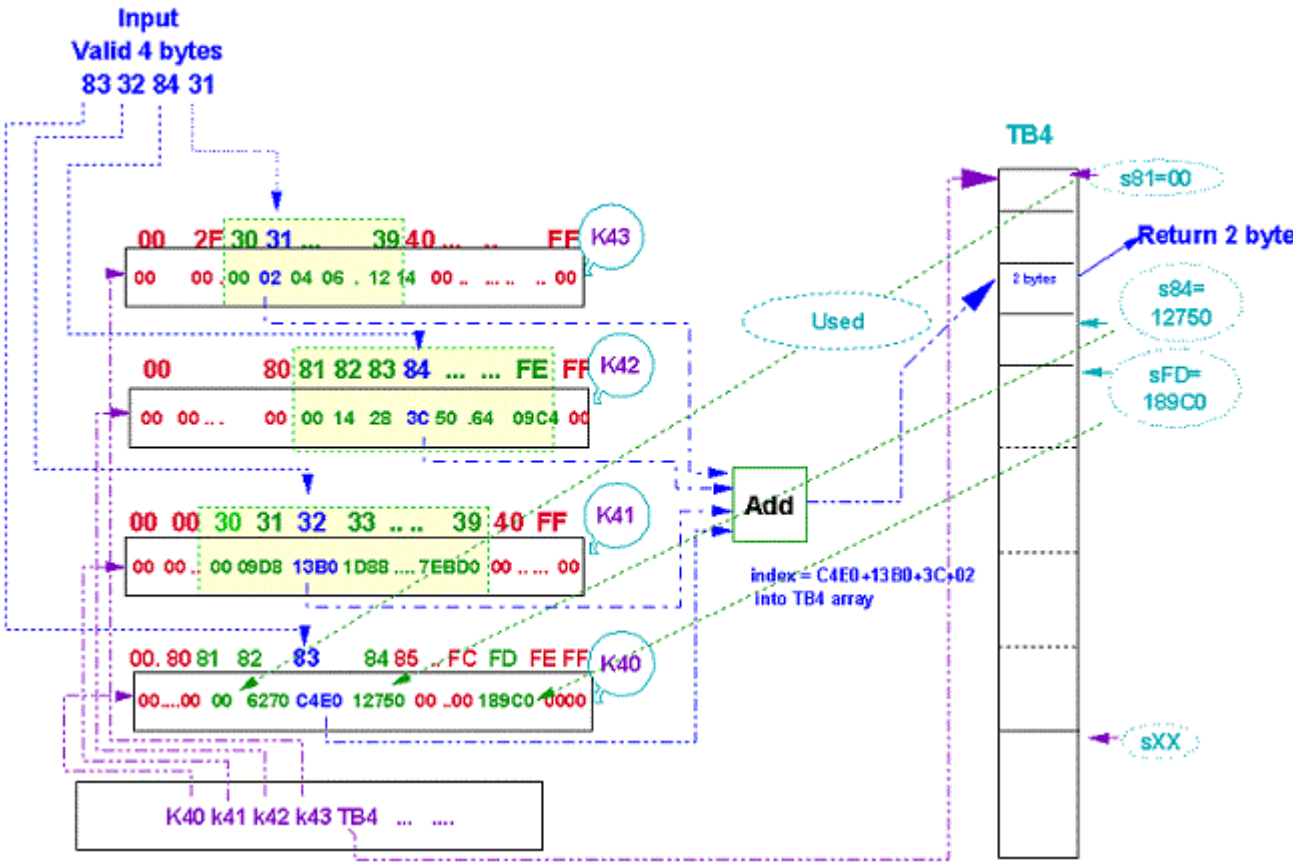

Figure 3.8: Use of Index Arrays in GB 4-byte to normalized S-Ch HOST 2-byte conversion method

This eliminates the multiplication steps during conversion execution, improving the performance. These values are added together to calculate the index (or pointer) to the first byte of the cell containing the output code point (see Figure 3.6). Figure 3.8 shows the mapping table of Figure 6 in a linearized structure.

Also, note that entries in these index arrays for illegal values of b0, b1, b2, or b3, are filled with zero-bytes. They will never be accessed if the filter logic has been followed correctly. Figure 3.8 also shows a possible entry for b0 = X'FD' -- Private Use Area - as a possible fifth block. The binary table currently defined contains mapping tables only for b0 values of X'81' to X'84', as defined in the conversion requirements received from Chinese Government sources.

The two bytes of the output code point are inserted into the output data stream. The pointer into the input data stream is incremented by four in the main filter logic described earlier.

Section 5: Annexes

ANNEX A - CCSIDs for Phase 1 and Phase 2 (as of 2001-06-14)

Note that complete definitions for all CCSIDs can be found in the CCSID repository.

GB 18030 - Phase 1

CCSID		ESID	Comments
Decimal	**Hex**		
05488	1570	2A00	S-ch PC Data mixed for GB 18030
05487	156F	2900	S-ch 4 byte part PC Data for GB 18030(Fixed UCS2 Subset)
09577	2569	2200	S-ch double-byte PC Data double-byte part of GB 18030 (Fixed UCS2 Subset) (*Four-byte SUB to be used)
09444	24E4	4105	S-ch single-byte part of GB 18030

GB 18030 - Phase 2

CCSID		ESID	Comments
Decimal	**Hex**		
01392	0570	2A00	S-ch PC Data mixed for GB 18030
01391	056F	2900	S-ch PC Data 4-byte part of GB 18030(Includes UCS Plane 1-16, and Plane 0 subset)

Host Mixed S-Ch Extended (for GB18030 support)

S-Ch DBCS-Host Data GBK mixed, all GBK character set and other growing chars(GB18030 / Unicode 3.0)

CCSID		ESID	Comments
Decimal	**Hex**		
01388	056C	1301	S-Ch Host mixed for GB 18030-1
13124	3344	1100	S-Ch Host single-byte part of GB 18030-1
04933	1345	1200	S-Ch Host double-byte part of GB 18030-1

09580	256C	1301	S-Ch Host mixed for GBK
00836	0344	1100	S-CH Host single-byte part of GBK
13125	3345	1200	S-Ch Host double-byte part of GBK