Table of Contents

Notices

"Python" is a registered trademark of the **Python** Software Foundation.

Introduction

This document describes how to execute the tests in the `asma/tests/chars` directory, particularly in your own environment.

Directory `asma/tests/chars` contains tooling to execute the ASMA testing of **Python** character handling. The directory itself includes the results of the tests as provided by the Stand Alone Tool Kit (SATK). To simply examine these results, no execution of the test scripts are required.

Only when observing these tests in your specific environment is there any value in actually executing the tests.

Before performing any of the actions on the local system, it is recommended that this manual be read in full. (Yes, RTFM.)

Establish a directory in which the local tests will be performed. Copy the scripts:

1. `src` – Creates ASMA source used to test the assembler (required),
2. `hex` – View the binary contents of the generated assembler source (optional),
3. `asm` – Perform ASMA test by assembling the source file from step 1 (required), and
4. `hex` – View the binary contents of the generated assembler listing (optional).

Perform local changes on each copy to conform with the test execution environment. Modify paths as required and statements compatible with the execution environment, in particular, the local operating system.

The following sections describes in more detail each step.

Execute each script in the above described order.

References

1. *ASMA – A Small Mainframe Assembler*, `doc/asma/asma.odt` or `doc/asma/asma.pdf`.

Step 1 – src – Create ASMA Source File (Required)

The `src` script uses the SATK supplied tool `asma/tests/chrstst.py` to create an assembler source file used by ASMA in this test.

Many of the assembler statements used by this test contain binary values that are not typically available with a text editor. The role of `chrstst.py` is to create the source used by the test incorporating these values.

`chrstst.py` performs two operations. First, it creates a binary work file that is then read as a **Python** text file in the same way as ASMA would read a text file (using **Python**). This "text" file as far as **Python** is concerned contains UTF-8 encoded text. UTF-8 only recognizes binary data containing values 0x00 through 0x7F inclusive. As a consequence, the work file is 128 bytes in length. By reading the work file, `chrstst.py` can determine how the UTF-8 characters are interpreted by **Python**, and, hence, what is received by ASMA when a line of text is read.

Second, from the interpretation of the UTF-8 characters by **Python**, `chrstst.py` can then create the ASMA assembler source file used by the test. The source file contains a DC assembler operation for each encountered character, enclosed in single quotes and followed by a comment identifying the hexadecimal value for which the constant is being generated.

chrstst.py Command-Line Options

The following command-line options are supported by the `chrstst.py` test tool.

-d, --debug

Causes debug messages to be generated when present.

-h, --help

Display the chrstst.py command-line options. When present all other options are ignored.

-s, --src FILEPATH

Identifies the path and file name of the generated ASMA assembler source file used by the test.

-w, --work FILEPATH

Identifies the path and file name of the generated **Python** work file.

Step 2 or 4 – hex – Examination of the ASMA Source File (Optional)

As an optional step, the binary contents of the generated ASMA source or listing file may be examined.

The `hex` script uses a locally available hexadecimal editor or display program. If a locally available hexadecimal editor is not installed, this step may be bypassed or a hexadecimal editor or display program may be acquired and installed.

The SATK provided `hex` script uses the `xxd` hexadecimal command-line editor. This editor is used simply because it is available on the author's Linux system. As provided, the `hex` script outputs the binary content of the ASMA source file as hexadecimal digits to the text file `chars.hex`. It assumes that the source file generated by `chrstst.py` is `chars.asm`. Adjust the paths and file names and tool to your local environment.

The `hex` script may also be used to create the hexadecimal content of the ASMA listing file, `listing.hex`. Modify the hex script for use with the listing file's date and time as observed within its file name.

Step 3 – asm – Assemble the Generated Test File

Assemble the generated test file using ASMA. The test file is assumed by the `asm` script to be `chars.asm`. The `asm` script creates the assembler output in the file:

```
asma-YYYYMMDD.HHMMSS.txt
```

where `YYYYMMDD.HHMMSS` is the date and time the assembler listing was created.

Test Results – Assembler Listing

Python defaults to UTF-8 character encoding of text files. As a consequence, ASMA converts, when assembling character data, from UTF-8 encoding to EBCDIC encoding.

The following line reflects this expected result. The initial **`F0`** is the EBCDIC encoding of the zero character assembled by ASMA's statement 71. The DC operation incorporates the character zero between its single nominal value surrounded by delimiting quotes. The final **`30`** is the UTF-8 encoding read by Python and passed to ASMA. It is the statement's comment.

```
000030  F0                                      71          DC    C'0'   30
```

The majority of lines in the assembly resemble this.

Unprintable Characters

Python recognizes as valid UTF-8 characters, but as unprintable those characters encoded by binary X'00'-X'08', X'0B', X'0C', X'0E'-X'1F', and X'7F' values.

The following two lines are from the hexadecimal editor of statement 20 in the generated source file. Statement 20 is in **`bold`** font. The actual nominal value being assembled is **underlined**.

```
00000310: 300a2020 20202020 20202044 43202020  0.         DC
00000320: 20432701 27202020 30310a20 20202020   C'.'   01.
```

As can be seen, the nominal value between the two single quotes (as two UTF-8 characters encoded as X'27') is X'01'. The comment in this line is 01, indicating that X'01' is being assembled. This matches the binary content between the quotes. This is as expected. X'01' is a valid UTF-8 encoding, albeit, not a printable character.

The local tool used to examine content that contains unprintable characters will influence what is “seen” by the user. This single line illustrates this. The above display is from the Linux `xxd` utility. The unprintable character is displayed as a period: `'.'`.

This file line is a direct copy of the `chars.asm` source file copied into `LibreOffice` used for composition of this document:

```
                DC     C'#'   01
```

`LiberOffice` displays the unprintable X’01’ with the number sign (‘#’).

This line is from a display of the same line but using the commonly used Linux `Kate` text editor.

```
                DC     C''   01
```

`Kate` does not display the unprintable character. (However it is present within the `Kate` editor because when copied from `Kate` into the `LibreOffice` document, the number sign appeared between the quotes. The number sign was deleted in the above text to show what was observed within `Kate`.)

Another editor is `jEdit`. Here is its display of the same assembler source line:

```
                DC     C''   01
```

`jEdit` does the same thing as does `Kate`. (The number sign inserted by `LibreOffice` was removed.)

```
      020          DC     C''   01
```

The above line is from the debug message generated by `chrstst.py` itself to the console. In this case, the X’01’ contained between the two single quotes has been removed during the console display.

As the final example, the actual assembly by ASMA produces this line:

```
      000001  01                                    20          DC     C''   01
```

It actually contains the X’01’ between the two quotes, but it does not appear in the listing. X’01’ is certainly made available to ASMA, because it assembles X’01’ into the resulting object code (in **bold** font above). This is apparent from the statement in the listing as well as the object content line below, font adjusted to fit. The single X’01’ is (in **bold** font below).

```
000000   000   00010203 04050607 08090A0B 0C0D0E0F  10111213 14151617 18191A1B 1C1D1E1F   |................ ................|
```

The major result here is that what is displayed for an unprintable UTF-8 character differs depending upon the tool used. **Python** does not display the unprintable character but it is

present. If the listing is actually printed, unpredictable results may occur. The results depend upon how the printer reacts when such a character is received by it.

Printable Characters

In general most of the UTF-8 characters are printable, namely, those encoded by binary X'09' (horizontal tab), X'0A' (line feed), X'0D' (carriage return), and X'20'-X'7E' (standard characters).

An example of the test of a printable character is provided in the above section for the numeric zero character, encoded in UTF-8 as X'30'. All of the other standard characters result in similar lines in the test assembly.

Horizontal Tab (X'09')

The following line is copied from the `Kate` editor displaying the assembly listing of the horizontal tab. The horizontal tab assembles correctly. But the listing may be different depending upon the tool used to display the "tab". There is in fact a single horizontal tab character in `chars.asm`. As observed, three character positions are used to display the tab in the listing file.

```
000009  09                                      28          DC    C'   '   09
```

However, when this same line is copied into `LibreOffice` a different result occurs.

```
000009  09                                      28          DC    C'    '   09
```

The following lines are from the `listing.hex` file containing the binary contents of the listing file itself. Assembler statement 28, in **bold**, assembles a character containing a horizontal tab.

```
00000970: 20204327 08272020 2030380a 30303030    C'.'   08.0000
00000980: 30392020 30392020 20202020 20202020  09  09
00000990: 20202020 20202020 20202020 20202020
000009a0: 20202020 20202020 20203238 20202020             28
000009b0: 20202020 20204443 20202020 43270927         DC    C'.'
000009c0: 20202030 390a2020 20202020 20202020     09
```

As with the previous example, the character nominal value is underlined in both the hexadecimal interpretation and text equivalent. As with the source statement, there is a single horizontal tab contained between the two single quotes. However, `Kate` and `LibreOffice` produce different displays of the single hexadecimal character. The `LibreOffice` display likely results from a second interpretation of the horizontal tab present in the `Kate` listing.

Care should be taken when examining the results of assembling a horizontal tab within a character nominal value. While the object will be correct as intended, the source listing may not match the object.

Line Termination Characters – Line Feed (X'0A) and Carriage Return (X'0D')

ASMA design depends upon **Python**'s handling of line termination characters. **Python** uses these characters to separate lines. This separation occurs before control of the line is passed to ASMA itself. This causes some unexpected behavior.

```
                                     29          DC    C'
** [29:16] expected character string nominal value, found end of statement
                                     30 '   0A
** [30] @[30]-1 macro definition for 0A not found in MACLIB path as either 0A.mac or 0a.mac
```

The error encountered for assembly statement 29 is not too surprising. The line feed between the two single quotation marks is interpreted by **Python** as the end of statement 29. The ASMA parser is looking for a terminating single quote, but does not find it in statement 29.

The second quote and the comment of `0A` are moved by **Python** to another text line. Assembler processing must determine what operation is being performed in this new text line to determine what type of label and the operand formats are valid for the operation. In this case, not finding an instruction or an assembler directive of '`0A`', the assembler tries finding the operation in a macro library. This too fails, resulting in the error for statement 30.

The only way to assemble into a program's object a binary X'0A' or X'0D' is to do so as hexadecimal (or binary) values. The following assembler statement, 31, shows this using a hexadecimal value.

```
00000A  0A                           31          DC    X'0A'  0A
```

Identical results occur in statements 34-36 illustrating what occurs when a carriage return character is encountered. In fact, **Python** will convert the carriage return character into a line feed character before presenting the text line to ASMA.

Horizontal Tab Outside of Character Nominal Value

A horizontal tab immediately following the operands field of the assembler statement generates an error. This is illustrated by these assembly listing lines:

```
                                    151          BALR  2,0       <--- tab is here
** [151] @[151]-1 expected right parenthesis or operator, found " <"
```

These listing lines in hexadecimal are:

```
00002fb0: 7f272020 2037460a 20202020 20202020  .'   7F.
00002fc0: 20202020 20202020 20202020 20202020
00002fd0: 20202020 20202020 20202020 20202020
00002fe0: 20202020 20313531 20202020 20202020       151
00002ff0: 20204241 4c522020 322c3009 3c2d2d2d    BALR  2,0.<---
00003000: 20746162 20697320 68657265 0a202020   tab is here.
00003010: 20202020 202a2a20 5b313531 5d20405b       ** [151] @[
00003020: 3135315d 2d312065 78706563 74656420  151]-1 expected
00003030: 72696768 74207061 72656e74 68657369  right parenthesi
00003040: 73206f72 206f7065 7261746f 722c2066  s or operator, f
00003050: 6f756e64 20220903c 220a0c41 534d4120  ound ".<"..ASMA
```

The horizontal tab character in assembly statement 151 is underlined in the previous hexadecimal listing. The tab immediately follows the final zero character in the instruction. The following error message identifies the tab as the start of the failure. It too appears in its hexadecimal display. The tool displaying the listing obscures the real error by inserting some number of spaces for the tab. This same error would be generated regardless of the character when it is not a space. It is just much harder to observe when the error is caused by a tab.

The error statement is modified in statement 152 with a space inserted following the zero.

```
      000080  0520                            152         BALR  2,0      <--- tab is here
```

This statement assembles without error. The listing in hexadecimal for this statement confirms the content:

```
00003110: 540a0a30 30303038 30202030 35323020  T..000080  0520
00003120: 20202020 20202020 20202020 20202020
00003130: 20202020 20202020 20202020 20202020
00003140: 31353220 20202020 20202020 2042414c  152          BAL
00003150: 52202032 2c302009 3c2d2d2d 20746162  R  2,0 .<--- tab
00003160: 20697320 68657265 0a202020 20202020   is here.
```

As the hexadecimal listing shows, the inserted space followed by the tab, both underlined, does not create an error.

The actual error message in this situation may vary depending upon the parsing process used for different operands. But the underlying cause is the same, namely, the missing space causes the assembler to believe additional content is part of the operands field, but the parser being used expects an end of the operands field, identified by the presence of a space character.

This is implemented by design. While nominal character values should support all characters, in this case allowed by **Python**, outside of the character nominal value, a tab is only allowed within the comment field. The comment field is separated from the operands field by a space. The space fixes the previous error and in this situation is required.

Non-UTF-8 Characters (UnicodeDecodeError Exception)

All single byte binary values between X'80' and X'FF' inclusive are outside the range of valid UTF-8 characters. As far as **Python** is concerned, these characters do not occur in UTF-8 text data. When encountered, **Python** treats these values as an error condition, the `UnicodeDecodeError` exception.

There is nothing that ASMA can do when reading text files to recover when this error occurs. If encountered, this exception will prematurely abort ASMA with an internal error.

Usually ASMA reads text files created by a local host text editor. In this case the `UnicodeDecodeError` will not be encountered. The author has never encountered it.

If the error is triggered, ASMA is most likely reading a file that contains binary data. Examine the files used within the assembly for possible binary content, perhaps an incorrect file name.

From this perspective, most EBCDIC characters will trigger this error. ASMA does not support the direct reading of EBCDIC card data. Card-based text data transferred to the local host from an EBCDIC host must be converted to UTF-8 characters with local host line-termination characters to be read by ASMA using **Python**. See Reference 1 for other considerations while moving EBCDIC source files for ASMA usage. On the other hand, all binary files created by ASMA inherently use EBCDIC character encoding. While such data can not be read by ASMA, it can be processed by programs or systems using EBCDIC.

Because `chrstst.py` is written in **Python** it operates under the same rules as does ASMA. For this reason only UTF-8 characters are tested by the tool. Anything else triggers the `UnicodeDecodeError` exception.