MICRO FOCUS®

UTF-8 標準の Linux 上での COBOL の活用手法

目次

はじめに 2
1 SJIS ロケールでの利用 2
1.1 COBOL をローカルに SJIS 稼動させる方法 2
1.2 メッセージのコード変換 4
1.3 COBOL が出力するファイルのコード変換 7
1.4 運用管理ツールでの対応 9
1.5 オンラインアプリケーションにおけるコード変換 9
1.6 サードパーティ製品との連携 13
2 UTF-8 ロケールへの書き換え 15
おわりに 19

はじめに

近年、グローバル化や常用漢字表の改定によりビジネス活動で扱う文字が増大しています。これに伴い UTF-8 をはじめたとした Unicode 文字を表現する符号化方式の採用を迫られるソフトウェアが増えてきています。しかし、従来のミッションクリティカルな基幹業務システムの多くは「JIS X 0208」の漢字コード規格をベースにした Shift JIS 系の文字コードを使用して開発されています。この符号化方式では日本語、英語以外の文字はもちろんのこと常用漢字表の改定等に伴い追加された文字を表現することができません。更に、Red Hat をはじめとする OS ベンダーがシステムロケールを UTF-8 以外に設定することをサポートしないと公表したことにより、アプリケーションの Unicode 化を検討する企業が増加の一途を辿っています。しかし、既に SJIS を前提にコーディングされているアプリケーションを Unicode 化するのは大きなインパクトを伴います。
本書では Unicode 前提のプラットフォーム上で COBOL アプリケーションを運用する方法について概説します。第 1 章では COBOL のみを SJIS ロケールで運用するテクニックについて解説します。モダナイゼーションを先に見据えつつも、まずは低コスト、低リスクで Unicode を前提としたプラットフォームをターゲットにリホスト をするのが短期的な目的であれば Unicode 化による影響は最小限に留めておきたいところです。Visual COBOL は SJIS に対応したソフトウェアであり、ユーザロケールを SJIS にしてビルド・実行するのであれば、SJIS を前提として開発された既存 COBOL 資産に影響が及びません。他のアプリケーションコンポーネントが UTF-8 環境で動いているのであれば、COBOL から入出力されるデータをプラットフォーム標準の方法で UTF-8 へ変換することで連携も可能です。
一方、システムを取り巻くテクノロジを刷新し、移行時にアプリケーションをモダナイズするというのであれば、アプリケーションの拡張性などを考慮しますと、この時点で多少の書き換えをしてでも COBOL の Unicode 化に取り組む価値は大いにあります。第 2 章ではこの場合の具体的な影響並びに対応策を紹介します。

1 SJIS ロケールでの利用

1.1 COBOL をローカルに SJIS 稼動させる方法

Red Hat Enterprise Linux はバージョン 4 以降はデフォルトのシステムロケールが「UTF-8 」に設定されています。また、このシステムロケールのデフォルト値を SJIS に変えた場合、Red Hat ではサポートの対象外とするとしています [1]。そのため、システムロケールを SJIS にして企業システムを運用するには各種ハードウェアベンダ等が提供する SJIS のサポートサービスを利用するのが一般的です。しかし、通常のサポートの範囲から外れないようシステムロケールはデフォルト値を維持しつつもユーザロケールを一時的に SJIS にして COBOL アプリケーションを運用する場合はどうでしょうか。ここでは、ユーザロケールを SJIS にする場合の影響を整理し、SJIS を前提として開発された COBOL アプリケーションを このユーザロケール下で運用する場合の影響を整理してみます。

Red Hat をはじめとした Linux ディストリビューションが SJIS ロケールをサポートしない主な理由としては以下のようなものが挙げられます [2]。

① Linux の文字処理ライブラリ関数は、Unicode を扱うことを基本としているため、本ライブラリ関数を使ってインプリメントされた Linux システムコマンドでは、ファイルデータの中の文字処理や、ファイル名の処理で、Unicode は正しく扱えても、Shift JIS は扱えないことがある。
② Shift JIS データの処理は、「特別」な扱いとなり、メールクライアント Thunderbird など、個々のミドルウェアに多大な開発負担を負わせている。
③ 特に、正統 Shift JIS ロケール sjis では、 0x5C=U+00A5 というマッピングのために、オープン系プログラム（C 言語、Java など）の動作が保証されない。cp932 などでは問題ない。

Visual COBOL 自体は内部的には OS のライブラリ関数を利用するため、①で問題としている関数が利用される可能性は排除できません。しかし、製品としては SJIS ロケール下での利用をサポートしています。Micro Focus では、SJIS ロケール下での利用を想定した品質管理プロセスを経て製品はリリースしています。仮に SJIS ロケール下固有の問題が発生した場合でも、製品に閉じられた範囲内での問題であればサポートの対象となります。た

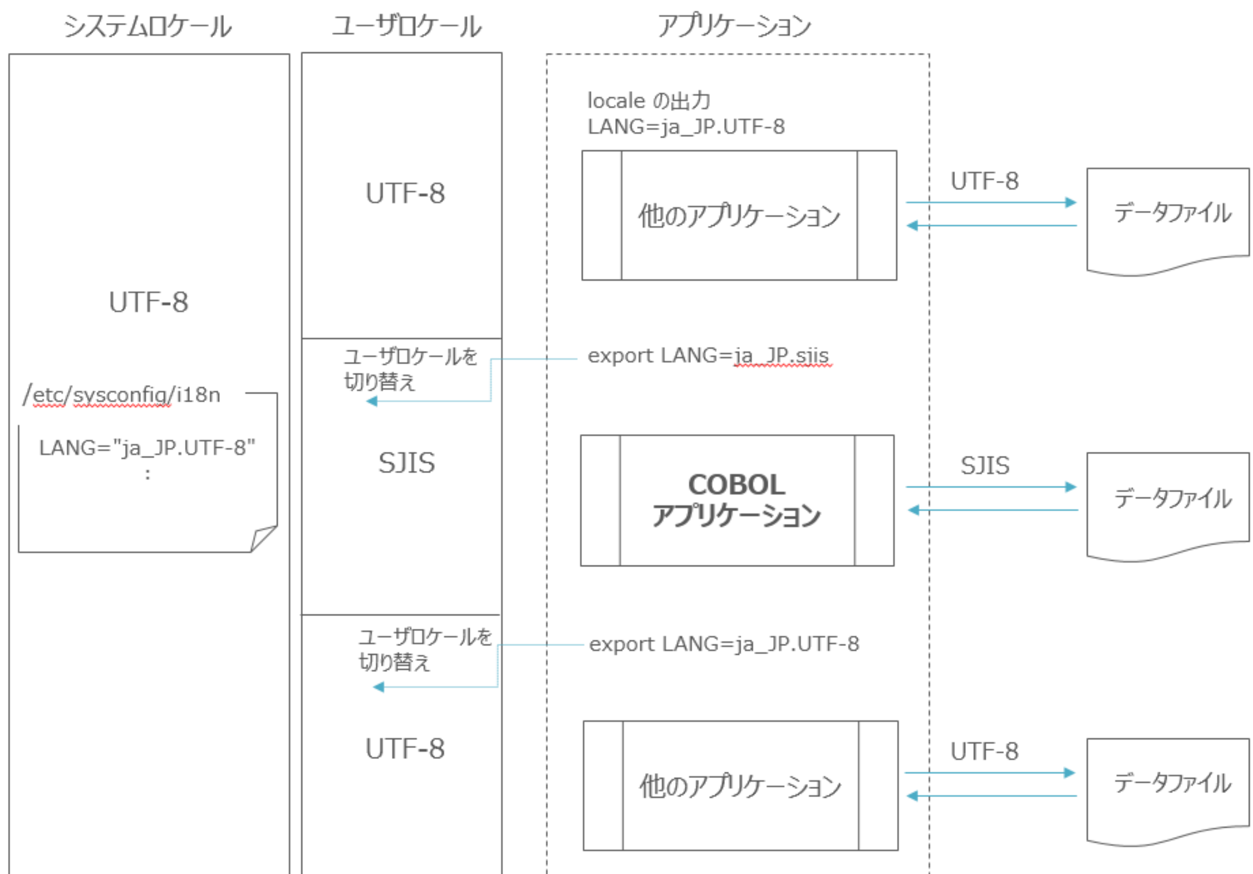
[1] 引用：日本 OSS 推進フォーラム プラットフォーム部会 マイグレーションタスクフォース(2009/7/10)
「Linux の Shift JIS サポート – 現状とその対応策 -」
http://www.ipa.go.jp/software/open/forum/download/Linux_SJIS_Support.pdf
(リンク確認: 2015/06/15)

[2] 同上

だし、アプリケーション内からこれらのライブラリ関数を直接利用するよう COBOL プログラムをコーディングしている場合は注意が必要となります。②に関しては、Visual COBOL はこの特別な扱いによる負担を背負ってでも SJIS ロケール 下での利用要求に応えるべく開発された製品であり、特に問題にはなりません。本製品と連携する各種ミドルウェアについても、多くは SJIS の扱いを想定して開発されているようです。この点については 1.6 で詳しくみていきます。COBOL 言語ではプログラムの構成要素である文字集合に③で問題としているバックスラッシュやチルダを含みません。「¥」を通貨記号として利用する場合も、JIS X 0201 の符号化方式に従い 0x5C を通貨記号としてコンパイラに認識させるための指令が用意されています。

以上のように、SJIS ロケール下で Visual COBOL でアプリケーションを開発し、それを運用する分には SJIS を利用する上で問題とされる内容は特段問題にはなりません。つまり、Red Hat のサポート内容から外れないようシステムロケールはデフォルト値を維持しつつも、COBOL を開発・運用する部分に関しては一時的にユーザロケールを SJIS に設定して運用して運用することが可能です。COBOL の部分に関しては、Micro Focus のサポート対象部分となり、例えば下図のように運用するのであれば全体のフローとしてメーカー未サポートのステップは発生しません。

図１－１
COBOL のみ SJIS ユーザロケールで運用する構成イメージ



上図では、COBOL アプリケーションの前段で実行されるアプリケーションはデフォルトから何も変更せず UTF-8 ロケール下で実行します。COBOL 部分については実行前にユーザロケールを SJIS に変更します。この変更はシステムロケールまでは及びません。Visual COBOL に付属するファイルハンドラは SJIS データも扱えるよう設計されているため、SJIS のユーザロケール環境下であれば SJIS で符号化されたデータを入出力できます。２バイト文字型の変数等、SJIS を前提としてコーディングされたプログラムから生成されたモジュールであっても、そのソースを同ユーザロケール下でコンパイルしていればこの環境下で正しく動作します。COBOL アプリケーションの実行後は、他のアプリケーションに影響が及ばぬようユーザロケールをデフォルト値に戻します。
以上のように COBOL アプリケーションの実行時のみユーザロケールを調整することで SJIS 上での運用を想定して開発されたリソースを UTF-8 標準の Linux 上でも有効活用することが可能です。

1.2 メッセージのコード変換

Visual COBOL より出力される各種メッセージは『<製品のインストールディレクトリ>/lang』の下に Visual COBOL がサポートするロケール毎にそれぞれ用意されています。例えば、本稿執筆時点で最新の Visual COBOL 2.2 Update 2 for x64/x86 Linux であれば下記のような構成でディレクトリが用意されています。

```
$ls -l $COBDIR/lang
合計 192
drwxr-xr-x. 2 root root 28672  5 月 29 13:11 C
lrwxrwxrwx. 1 root root     1  5 月 29 13:11 default -> C
lrwxrwxrwx. 1 root root     1  5 月 29 13:11 en_GB -> C
lrwxrwxrwx. 1 root root     1  5 月 29 13:11 en_GB.UTF-8 -> C
lrwxrwxrwx. 1 root root     1  5 月 29 13:11 en_US -> C
lrwxrwxrwx. 1 root root     1  5 月 29 13:11 en_US.UTF-8 -> C
lrwxrwxrwx. 1 root root    11  5 月 29 13:11 ja_JP -> ja_JP.eucjp
lrwxrwxrwx  1 root root    10  6 月 12 10:59 ja_JP.SJIS -> ja_JP.sjis
lrwxrwxrwx. 1 root root    10  5 月 29 13:11 ja_JP.UTF-8 -> ja_JP.utf8
lrwxrwxrwx. 1 root root    11  5 月 29 13:11 ja_JP.eucJP -> ja_JP.eucjp
drwxr-xr-x. 2 root root 28672  5 月 29 13:11 ja_JP.eucjp
drwxr-xr-x. 2 root root 28672  5 月 29 13:11 ja_JP.sjis
lrwxrwxrwx. 1 root root    11  5 月 29 13:11 ja_JP.ujis -> ja_JP.eucjp
drwxr-xr-x. 2 root root 28672  5 月 29 13:11 ja_JP.utf8
lrwxrwxrwx. 1 root root    11  5 月 29 13:11 japanese -> ja_JP.eucjp
$
```

X64/x86 Linux 版製品では上の出力中にあるディレクトリ名に対応するロケールをサポートしており、実行時はそれぞれのロケールに応じたディレクトリ配下に格納されているメッセージが出力される仕組みとなります。1.1 の要領でユーザロケールを SJIS にして実行すれば、SJIS ロケールの下にあるメッセージファイルが利用され SJIS で符号化されたメッセージが出力されます。しかし、図１－１のようなアプリケーションフローにて SJIS のメッセージを混在させるのではなく全て UTF-8 で統一した作業ログを生成したいという要望もあるかもしれません。この場合、SJIS ロケールに対応したディレクトリを UTF-8 のロケールに対応したディレクトリに置き換えることで、実行モジュールは SJIS ロケールに対応した動作をしつつもメッセージは UTF-8 に符号化されたものを得ることができます。以下にその例を記します。

1) ここでは下記のような２バイト文字項目で構成されたレコードをファイル出力するロジック及び、添え字の範囲外エラーを引き起こすロジックを含んだプログラムを例にとり考えてみます。

```
$cat MSGDEMO.cbl
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT AFILE ASSIGN TO "AFILE.txt".
       DATA DIVISION.
       FILE SECTION.
       FD AFILE.
       01 AREC                 PIC G(5).
       WORKING-STORAGE SECTION.
       01 A-VAR                PIC X(5) OCCURS 10.
       01 WK-IDX               PIC 9(3) VALUE 11.
       PROCEDURE DIVISION.
           OPEN OUTPUT AFILE.
           MOVE SPACE TO AREC.
           WRITE AREC.
           CLOSE AFILE.

           MOVE SPACE TO A-VAR(WK-IDX).
           GOBACK.
$
```

２バイト文字項目で構成されたレコードに空白を転記してファイル書き出しをします。

10 回の繰り返し項目に対して 11 を添え字に指定しています。

2) このプログラムを UTF-8 ロケール下で、UTF-8 に設定されたターミナルよりコンパイル実行すると出力メッセージは UTF-8 になるため、正しく表示できます。しかし、プログラムそのものは２バイト文字支援機能が有効にならず正しく処理されていないことがわかります。

```
$cob -u -C"NCHAR" MSGDEMO.cbl
$cobrun MSGDEMO.gnt

目的コード エラー：ファイル '/home/yoshihiro/test/wp_sjis/MSGDEMO.gnt'
エラーコード：153, pc=0, call=1, seg=0
153 添字が指定範囲外になっている (MSGDEMO.cbl 内, 18 行)

$od -x AFILE.txt
0000000 2020 2020 2020 2020 2020
0000012
$
```

エラーメッセージは UTF-8 で符号化されて出力されていることがわかります。

SJIS ロケール下で実行されていないため、全角スペース(X'8140') ではなく半角スペースが出力されています。

3) ユーザロケールを SJIS に変更し再度、コンパイル・実行します。今回は SJIS ロケール下で実行されているため、２バイト文字支援機能は正しく機能していますが、出力メッセージは SJIS で符号化されているため、UTF-8 に設定されたターミナルでは正しくメッセージを表示することができません。

```
$export LANG=ja_JP.sjis
$cob -u -C"NCHAR" MSGDEMO.cbl
$cobrun MSGDEMO.gnt

        'I▒R▒[▒h ▒G▒▒▒[: ▒t▒@▒C▒▒ '/home/yoshihiro/test/wp_sjis/MSGDEMO.gnt'▒
▒G▒▒▒[▒R▒[▒h: 153, pc=0, call=1, seg=0
153 ▒Y▒▒▒▒w▒▒O▒▒Ä▒Ä▒▒▒ (MSGDEMO.cbl ▒▒, 18 ▒s)

$od -x AFILE.txt
0000000 4081 4081 4081 4081 4081
0000012
$
```

ユーザロケールを SJIS に変更

メッセージはロケールに応じて SJIS で出力されるため、SJIS に対応していない環境では表示できません。

SJIS ロケール下で実行されているため、２バイト文字支援機能が正しく機能し全角スペース X'8140' が出力されています。

※ Linux の distribution によっては、SJIS のロケールが出荷時の状態では用意されていないものもあるようです。しかし、SJIS の charmap ファイルについては執筆時点で確認できている限りでは、SJIS ロケールがない distribution にも用意されているようです。この場合、下記例のようにして localedef コマンドによる SJIS ロケール追加が可能です。-f フラグに続いて指定する charmap ファイルは JIS X 0208 にない拡張文字の扱いに関する基盤管理の方針等に基づき適切なファイルを選択します。

```
# localedef -f WINDOWS-31J -i ja_JP ja_JP.sjis
#
```

4) SJIS ロケールで実行する際に利用されるメッセージファイルを UTF-8 用のものに置き換えます。

```
# mv ja_JP.sjis ja_JP.sjis_bk
# cp -r ja_JP.utf8 ja_JP.sjis
#
```

SJIS ロケールで利用されるメッセージファイルが格納されたディレクトリをバックアップ

UTF-8 ロケールで利用されるメッセージファイルが格納されたディレクトリを SJIS ロケールで実行される際にピックアップされるようコピー移動

5) この環境で再度実行するとランタイムは SJIS ロケールに応じて正しく処理をします。その一方でメッセージファイルに関しては UTF-8 用のものがピックアップされるようになっています。

```
$cob -uv -C"NCHAR" MSGDEMO.cbl
cob64 -C nolist -uv -CNCHAR MSGDEMO.cbl
* Micro Focus COBOL                    V2.2 revision 002          Compiler
* Copyright (C) Micro Focus 1984-2014. All rights reserved.
* 受諾 - verbose
* 受諾 - nolist
* 受諾 - NCHAR
* MSGDEMO.cbl のコンパイル中
* 合計メッセージ:     0
* データ:         872     コード:          313
* Micro Focus COBOL Code Generator
* Copyright (C) Micro Focus 1984-2014. All rights reserved.
* Accepted - verbose
* Generating MSGDEMO
* Data:           640     Code:          808      Literals:          32
$cobrun MSGDEMO.gnt

目的コード エラー: ファイル '/home/yoshihiro/test/wp_sjis/MSGDEMO.gnt'
エラーコード: 153, pc=0, call=1, seg=0
153  添字が指定範囲外になっている (MSGDEMO.cbl 内, 18 行)
$od -x AFILE.txt
0000000 4081 4081 4081 4081 4081
0000012
$
```

エラーメッセージだけでなく、コンパイラのメッセージも UTF-8 化されたものが出力されます。

1.3 COBOL が出力するファイルのコード変換

ユーザロケールを SJIS にした場合、ACCEPT/DISPLAY 文等によるコンソール入出力やファイルの入出力は SJIS で符号化されたデータとなります。

Linux には iconv という文字列の符号化を変換するコマンドが用意されています。コンソール入出力をリダイレクトしてログ管理しており、ここも UTF-8 で一括管理したいという場合は、COBOL からの出力を直接 iconv で SJIS から UTF8 に変換してリダイレクトすることが可能です。以下にその例を記します。

1) ユーザロケールが SJIS の環境下で DISPLAY "あいうえお" のみを持つ SJIS で符号化されたプログラムソースを用意します。

サンプルプログラム：

```
       PROCEDURE DIVISION.
           DISPLAY "あいうえお".
           GOBACK.
```

2) 1) で用意したプログラムをコンパイルして実行します。1.2 と同様にユーザロケールは SJIS にしていますが、UTF-8 に設定されたターミナルから実行しているため、DISPLAY 出力は正しく表示されません。

```
$echo $LANG
ja_JP.sjis
$cob -u DISPDEMO.cbl
$cobrun DISPDEMO.gnt

$
```

SJIS データが DISPLAY 出力されるため、UTF-8 に設定されたターミナルからは正しく内容を確認することができません。

3) DISPLAY による出力を直接 SJIS から UTF-8 に変換します。

```
$cobrun DISPDEMO.gnt

$cobrun DISPDEMO.gnt|iconv -f Shift_JIS -t UTF-8
あいうえお
$
```

ここでは、DISPLAY 出力を直接 iconv で SJIS から UTF-8 に変換しているため、上の結果と異なり、UTF-8 に変換された DISPLAY 出力を確認することができます。これを更にログファイル等にリダイレクトすれば、UTF-8 に符号化されたログファイルとして管理ができます。

上で例示した方法は SJIS ユーザロケール下で iconv コマンドを実行しています。仮に本コマンドが文字処理ライブラリ関数を使ってインプリメントされたものであれば、1.1 の①で述べた懸念が危惧されます。この点も更にクリアしたフローを確立したいというのであれば、下例のように iconv を UTF-8 ロケール下で実行するという手段もあります。

① ユーザロケールを SJIS に変更
② COBOL アプリケーションを実行(この際、コンソール入出力の結果はファイルに SJIS データのままファイルリダイレクトします。)
③ ユーザロケールを UTF-8 に戻す
④ iconv コマンド実行し、②でリダイレクトしたファイル中のデータを UTF-8 へ変換
⑤ UTF-8 で管理する作業ログファイルへ④で変換したファイルをアペンド

COBOL アプリケーションから入出力されるファイルも COBOL アプリケーションの実行前後で UTF-8 ⇔ SJIS 変換をし、他の UTF-8 ロケール下で動作するアプリケーションと連携させるというのであれば、上のようなフローで管理することが可能です。以下は iconv を使って COBOL アプリケーションからの出力ファイルを変換する例となります。

1) SJIS データを出力するサンプルプログラムを用意します。

SJIS で「かきくけこ」のレコードデータを BFILE.txt へ書き出します：

```
$cat CONVDEMO.cbl
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT BFILE ASSIGN TO "BFILE.txt".
       DATA DIVISION.
       FILE SECTION.
       FD BFILE.
       01 BREC                   PIC G(5).
       WORKING-STORAGE SECTION.
       01 BVAL                   PIC X(10) VALUE X'82A982AB82AD82AF82B1'.
       PROCEDURE DIVISION.
           OPEN OUTPUT BFILE.
           WRITE BREC FROM BVAL.
           CLOSE BFILE.
           GOBACK.
$
```

2) 1) で用意したプログラムをこれまでと同じ環境でコンパイル・実行し、出力されたファイルを確認します。

```
$cob -u CONVDEMO.cbl
$cobrun CONVDEMO.gnt
$cat BFILE.txt
▒▒▒▒▒▒▒▒▒▒$
$
```

SJIS で符号化されたデータが出力されます。

3) ユーザロケールを UTF-8 に戻します。

4) iconv を実行し、COBOL アプリケーションから出力されたファイル中のデータを UTF-8 に変換します。

```
$iconv -f Shift_JIS -t UTF-8 BFILE.txt > BFILE_UTF8.txt
$cat BFILE_UTF8.txt
かきくけこ$
$od -x BFILE_UTF8.txt
0000000 81e3 e38b 8d81 81e3 e38f 9181 81e3 0093
0000017
$od -x BFILE.txt
0000000 a982 ab82 ad82 af82 b182
0000012
$
```

BFILE.txt 中のデータを UTF-8 へ変換し、結果を BFILE_UTF8.txt へリダイレクトしています。

正しく変換できていることが確認できます。

1.4 運用管理ツールでの対応

運用管理ツールは、分散化されたコンピュータシステム上で稼働する様々なアプリケーションの正常な稼働状態を維持し、それらの運用と保守を支援するものです。昨今のシステム構築ではこれらを活用して運用を自動化・効率化することが一般的です。
Linux 上で利用可能な代表的な運用管理ツールとしては、ユニリタ A-AUTO、日立 JP1、富士通 System Walker、IBM Tivoli Workload Scheduler、HP OpenView、野村総研 Senju などがあり、それぞれの特長と強化分野を持っています。
運用管理ツールの多くは、管理対象の各アプリケーションが出力するログ・メッセージを一元的に監視してヘルスチェックを行う機能を装備しています。Linux 上では UTF-8 ロケールが標準となっているため、多くのツールはログ・メッセージが UTF-8 で出力されることを前提としていますが、これまで 1.2、1.3 で述べてきたような方法によって、SJIS ロケール下で稼働する COBOL アプリケーションのログ出力も UTF-8 で行うことができますので、この点で問題はありません。
また、A-AUTO のような運用監視ツールはこのようなケースを想定して設計されており、ログのビューワーがエンコーディングを動的に選択できるようになっています。このため、COBOL アプリケーション部分が SJIS のままでログ出力しても問題なく対応可能です。

1.5 オンラインアプリケーションにおけるコード変換

バッチアプリケーションについては、1.1 で例示したようなフローで COBOL の実行部分のみを SJIS とするよう確立できますが、UTF-8 前提で稼動するオンラインアプリケーションにて COBOL のみを SJIS で稼動させるよう組み込むことは可能でしょうか。Linux 版の Visual COBOL が提供するオンラインアプリケーション開発で有用な主な機能としては、COBOL 専用のアプリケーションサーバーである「Enterprise Server」に COBOL モジュールをディプロイし EJB や Web Service として利用させる方法、並びに COBOL を javabyte コードへコンパイルし JVM アプリケーションに組み込む方法があります。本項ではこれらの技法を用いてそれぞれどのように既存の SJIS で開発された COBOL 資産を活用できるか見ていきます。

- Enterprise Server の活用
  Enterprise Server にディプロイされた COBOL アプリケーションは Web Service コンシューマや EJB クライアントと別の独立したプロセスにて動作します。この Enterprise Server についても SJIS ロケール上での運用がサポートされた製品コンポーネントであり、ここにディプロイされた COBOL アプリケーションはクライアント側の実行ロケール等に関わらず SJIS のアプリケーションとして運用することが可能です。

  また、クライアントアプリケーションとのデータ授受についても COBOL との前後で適切な変換処理が入るため、クライアント側及び COBOL については一切コード変換を意識する必要がありません。Enterprise Server に COBOL をディプロイする際は、IMTK(Interface Mapping ToolKit) を使って COBOL のデータ型と SOAP や Java のデータ型との変換マッピングを定義します。この定義情報を COBOL アプリケーションと併せてディプロイすることで実行時にサービス実行プロセス(Enterprise Server 上でアプリケーションコンテナ等として機能するプロセス)における Request Handler コンポーネントが COBOL とのパラメータ入出力の前後でこの定義情報に基づいた変換処理を行います。

  ここでは、UTF-8 で渡された SOAP リクエストを SJIS 環境上で稼動する Enterprise Server が正しくハンドリングし、COBOL アプリケーションが SJIS のアプリケーションとして正しく動作するようすを見てみます。

  1) SJIS のデータファイルを読み込み、２バイト文字項目を使って部分参照等をするプログラムを用意します。

     データファイル：

```
$ od -x AAA.dat
0000000 b382 b582 b782 b982 bb82 bd82 bf82 c282
0000020 c482 c682
0000024
$ cat AAA.dat
さしすせそたちつてと$
```

     SJIS で「さしすせそたちつてと」を符号化したデータが格納されています。

COBOL プログラム：

```
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
           SELECT AFILE ASSIGN TO "AAA.dat".
       DATA DIVISION.
       FILE SECTION.
       FD AFILE.
       01 AREC           PIC G(10).
       LINKAGE SECTION.
       01 PARMA          PIC X(10).
       01 PARMB          PIC X(10).
       PROCEDURE DIVISION USING PARMA
                                PARMB.
           IF PARMA = "あいうえお" THEN
               DISPLAY "COBOL に正しくわたりました" UPON CONSOLE
               DISPLAY "PARMA: 「" PARMA "」" UPON CONSOLE
               DISPLAY "PARMB: 「" PARMB "」" UPON CONSOLE
           END-IF.
           OPEN INPUT AFILE.
           READ AFILE.
           MOVE AREC(6:5) TO PARMA.
           MOVE AREC(1:5) TO PARMB.
           CLOSE AFILE.
           GOBACK.
```

2) COBOL モジュール及びデータファイルをデフォルトマッピングでディプロイします。(PARMA 及び PARMB は string 型にマッピングされています。)

3) ディプロイしたアプリケーションを呼び出します。

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsd="http://tempuri.org/WSDEMO">
   <soapenv:Header/>
   <soapenv:Body>
      <wsd:COBSUB>
         <wsd:PARMA_in>あいうえお</wsd:PARMA_in>
         <wsd:PARMB_in>かきくけこ</wsd:PARMB_in>
      </wsd:COBSUB>
   </soapenv:Body>
</soapenv:Envelope>[EOF]
```

リクエスト

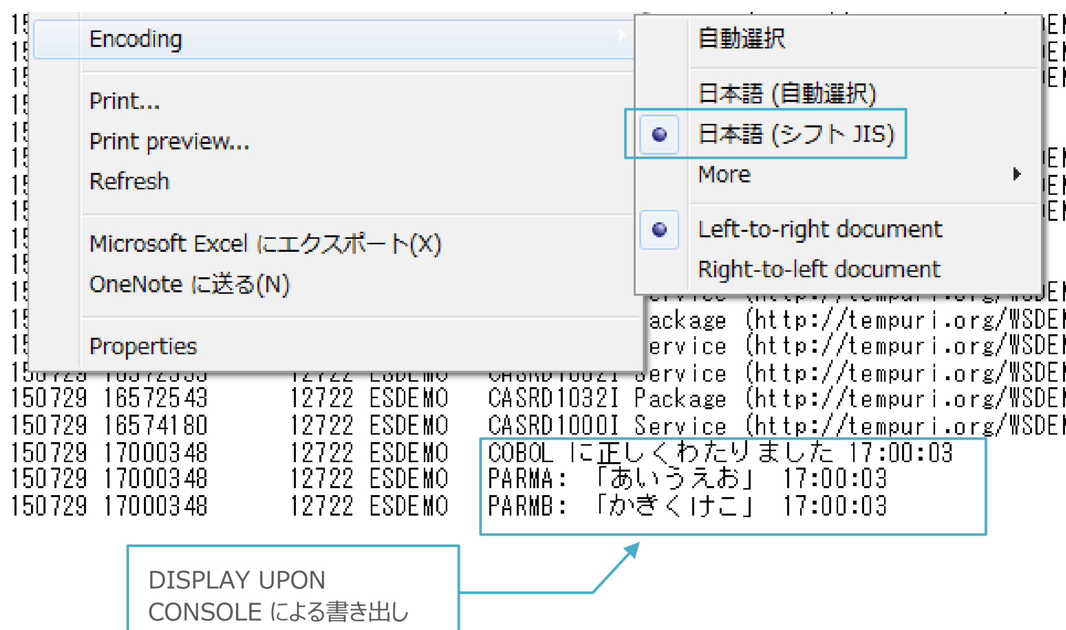
20桁 標準 [80] UTF-8N CRLF 挿入

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Body>
      <ns0:COBSUBResponse xmlns:ns0="http://tempuri.org/WSDEMO">
         <ns0:PARMA_out>たちつてと</ns0:PARMA_out>
         <ns0:PARMB_out>さしすせそ</ns0:PARMB_out>
      </ns0:COBSUBResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>[EOF]
```

レスポンス

1行: 1桁 標準 [80] UTF-8N CRLF 挿入

Web Service コンシューマは UTF-8 で符号化された「あいうえお」及び「かきくけこ」を入力パラメータとして設定し Enterprise Server 上にホストされた Web Service を呼び出しています。Enterprise Server に渡った UTF-8 のデータは SJIS に変換され、COBOL に渡されます。SJIS データとして正しく受け取っているのであれば、13 行目の条件式の評価は真となり DISPLAY 文が実行されます。Enterprise Server 上の COBOL アプリケーションが DISPLAY UPON CONSOLE とすると console.log に DISPLAY 文の内容を書き出します。下図はこの console.log をブラウザ表示したものとなりますが、DISPLAY 文が３つ実行されていることが確認できます。このうち２つは COBOL に渡されたデータをそのまま出力しており、SJIS のデータとして渡っていることがここからも確認できます。
COBOL アプリケーションは続いて、２バイト文字の変数でレコード定義された SJIS のデータファイルを読み出し、SJIS のデータとしてパラメータにセットしています。Enterprise Server 自体が SJIS ロケール上で稼動しているため、２バイト文字項目の部分参照のように SJIS 環境でのみ利用できる機能もローカルアプリケーションと同様に利用できます。ここからレスポンスメッセージを組み立てる際も、Enterprise Server は適切な変換をした上でセットします。上の実際のレスポンスを確認すると正しく UTF-8 で符号化された値を戻り値として返していることが確認できます。



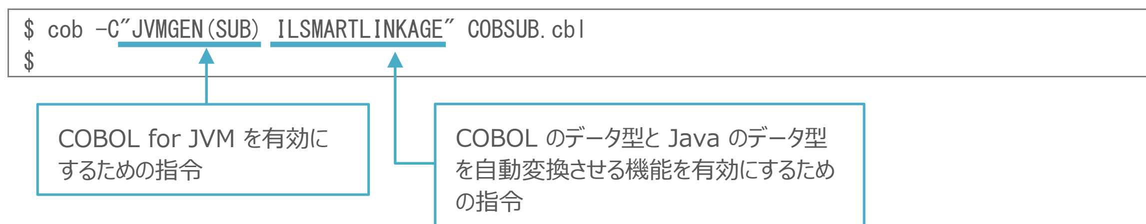
オンラインアプリケーションの場合、バッチアプリケーションで見たようにロケールを適宜切り替えるのは現実的ではありません。しかし、SJIS で運用したい COBOL 資産だけを切り出し Enterprise Server 上で部品化して運用すれば既存の SJIS 前提で開発された COBOL 資産を UTF-8 上で稼動するオンラインアプリケーションから利用することが可能です。Enterprise Server 上で COBOL アプリケーションが SJIS ロケール配下で稼動していようとも独立したプロセスとして動作し、適切なコード変換も Enterprise Server 側で処理されるため、その外側の Web Service コンシューマや EJB クライアントは COBOL 側のロケールや文字コードは意識する必要がありません。

> COBOL for JVM による Java 連携

COBOL for JVM を使って COBOL プログラムをコンパイルすると javabyte コードを生成し、JVM クラスとしてその COBOL プログラムを扱えるようになります。Enterprise Server を使ったソリューションと異なり、別のプロセスではなく JVM 上で Java と直接連携が可能です。パラメータ授受については、COBOL 側で英数字項目や２バイト文字項目に SJIS データを格納していても Java とやりとりする際は String 型のオブジェクトとしてやりとりが可能です。String 型のオブジェクトに格納されるデータは UTF-16 形式で値を保持しますが、Visual COBOL がこの間の変換を行うため、COBOL、Java ともにこの間の文字コード変換を意識する必要がありません。

以下にてこの変換のようすを確認します。

1) ユーザロケールを SJIS にセットします。[3]
2) Enterprise Server の例で使用したプログラムを javabyte コードへコンパイルします。



3) COBOL を呼び出す Java プログラムを用意します。[4]

```
$ cat CallCob.java
class CallCob{
    public static void main(String[] args){
        COBSUB cobclass = new COBSUB();
        Parma   cobparm1  = new Parma();
        Parmb   cobparm2  = new Parmb();
        cobparm1.setParma("あいうえお");
        cobparm2.setParmb("かきくけこ");
        cobclass.COBSUB(cobparm1, cobparm2);
        String chkStr = "¥u305F¥u3061¥u3064¥u3066¥u3068";
        if (cobparm1.getParma().equals(chkStr)){
            System.out.println("java に正しく値が戻されています");
            System.out.println("PARMA: 「" + cobparm1.getParma() + "」(java)");
            System.out.println("PARMB: 「" + cobparm2.getParmb() + "」(java)");
        }
    }
}
$
```

4) Java のプログラムをコンパイルして実行します。

```
$ javac CallCob.java
$ java CallCob
COBOL に正しくわたりました
PARMA: 「あいうえお」
PARMB: 「かきくけこ」
java に正しく値が戻されています
PARMA: 「たちつてと」(java)
PARMB: 「さしすせそ」(java)
$
```

英数字項目で定義された COBOL のパラメータは String の setter と getter を持つクラスにエクスポーズされます。Java 側ではこの自動生成された setter に String 形式で「あいうえお」及び「かきくけこ」をセットします。続いてこのクラス参照を COBOL のメソッドの引数にセットして呼び出すことで、Visual COBOL が COBOL に渡す前に String 型から SJIS の英数字項目に変換します。これによ

[3] 上述のように SJIS を前提として COBOL アプリケーションを運用するにはユーザロケールを SJIS に設定する必要があります。Enterprise Server の例と異なり、JVM 上で COBOL も動かすこととなるため、この COBOL を SJIS で動かすのであれば JVM も SJIS ユーザロケール上で稼動させる必要があります。
[4] SMARTLINKAGE を使ったパラメータ授受の仕組みに関しては別文書「Micro Focus による COBOL と Java の融合」
http://www.microfocus.co.jp/resources/whitepaper/pdf/white_paper_057.pdf
や製品マニュアルをご参照してください。

り、上の結果にあるように COBOL 側で SJIS のデータを Enterprise Server の例と同様正しくハンドリングしています。Java 側に結果が戻ると UTF-16 形式で直接「たちつてと」をセットした String の変数と値比較します。この結果、真となり Java 側に渡ったデータを出力させています。このように COBOL から戻る際も String 型に正しく変換されていることが確認できます。

SJIS 上での利用を前提として開発された COBOL 資産であっても、上で見たように Unicode でデータを内部表現する Java の String とシームレスにやりとりができることが確認できます。Enterprise Server を使う場合と異なり、このケースで JVM 自体を UTF-8 で稼動させることは難しいですが、このような SJIS を前提とした COBOL 資産も JVM 上で有効活用することが可能です。

1.6 サードパーティ製品との連携

COBOL 単体で動作するアプリケーションであればこれまでに紹介してきた方法で運用が可能ですが、サードパーティ製品と連携するようなアプリケーションを運用する際は、それらが SJIS に対応できているか確認する必要があります。オープン化されたアプリケーションの多くは RDB でデータを管理します。Linux 上で広く利用されている商用 RDB の Oracle を例にとりますと、NLS_LANG という環境変数を指定することで Database Server 上のデータベース・キャラクタ・セットと Client の間の符号を変換してくれるようです [5]。つまり、NLS_LANG に SJIS が指定された環境で実行すれば、SJIS に符号化されたデータで RDB と入出力処理できるようになります。本環境変数を指定して Oracle Client にコード変換させることにより、データベース上では Unicode で管理しつつも COBOL アプリケーションがデータベースとデータ授受する際は、SJIS でやりとりすることを可能とします。

ここでは、NLS_LANG 環境変数を構成して COBOL アプリケーションが実行する時のみ SJIS でやりとりができることを確認します。

1) データベース・キャラクタ・セットが UTF-8 の Database Server 上に UTF-8 で格納された日本語データがあることを確認します。

① データベース・キャラクタ・セットが UTF-8 であることを確認

```
SQL> SELECT value$ FROM sys.props$ WHERE name = 'NLS_CHARACTERSET';

VALUE$
--------------------------------------------------------------------------------
AL32UTF8

SQL>
```

② 日本語データが UTF-8 で格納されていることを確認

```
SQL> select COL2, RAWTOHEX(COL2) FROM WPDEMO;

COL2                           RAWTOHEX(COL2)
------------------------------ ------------------------------------------------------------
あいうえお                     E38182E38184E38186E38188E3818A
かきくけこ                     E3818BE3818DE3818FE38191E38193
さしすせそ                     E38195E38197E38199E3819BE3819D

SQL>
```

[5] 参考 : Oracle Database Online Documenttation 12c Release 1(12.1)
「Database Globalization Support Guide – 3. Setting Up a Globalization Support Environment」
http://docs.oracle.com/database/121/NLSPG/ch3globenv.htm#NLSPG003
(リンク確認: 2015/06/22)

2) 用意したテーブルから３番目のレコードを取り出し、そのままファイル書き出しするプログラムを用意します。

Oracle Database 中にて UTF-8 のコードで「さしすせそ」に相当するデータが格納されたレコードを読み込み、CFILE.txt に直接書き出します：

```
$ cat NLSDEMO.cbl
                   :
        SELECT CFILE ASSIGN TO "CFILE.txt".
    DATA DIVISION.
    FILE SECTION.
    FD CFILE.
    01 CREC                PIC X(10).
    WORKING-STORAGE SECTION.
        EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01  HV-COL2            PIC X(10) VARYING.
                   :
    PROCEDURE DIVISION.
                   :
        EXEC SQL
            SELECT COL2 INTO :HV-COL2 FROM WPDEMO WHERE COL1 = 3
        END-EXEC.
                   :
        WRITE CREC FROM HV-COL2-ARR.
        CLOSE CFILE.
        GOBACK.
$
```

3) NLS_LANG 環境変数を指定します。

```
$ export LANG=Japanese_Japan.JA16SJISTILDE
$
```

charset = SJIS に設定しています。

4) プログラムをコンパイルして実行します[6]。

```
$ cob -u -C"INITCALL(orainit) p(cobsql) CSQLT==ORACLE8 end-c endp" NLSDEMO.cbl

* Cobsql Integrated Preprocessor
* CSQL-I-018: Oracle プリコンパイラトランスレータを起動します。
* CSQL-I-020: Oracle プリコンパイラの出力を処理中。
* CSQL-I-001: COBSQL：チェッカへの引き渡しを完了しました。
$ cobrun NLSDEMO.gnt
$
```

COBSQL の技術を利用し、Pro*COBOL を内部的にキックさせています。

[6] Oracle 連携をする際のコンパイルオプションの内容や技法等については別の White Paper「Pro*COBOL による Oracle データベース 12.1c アクセス動作検証結果報告書」や製品マニュアル等を参照してください。

5) COBOL が出力したファイルを確認します。

NLS_LANG の設定により Oracle Client がデータを取得する際に設定に基づいた変換処理をしているため、COBOL アプリケーションがデータを受け取る時点では、SJIS のデータが渡されており，COBOL 側では特にコードを意識することなく処理できます：

```
$ od -x CFILE.txt
0000000 b382 b582 b782 b982 bb82
0000012
$ cat CFILE.txt
さしすせそ$
```

2 UTF-8 ロケールへの書き換え

SJIS の文字体系でアプリケーション要件を全て満たせるのであれば、書き換えに伴うコストを考えると第 1 章で紹介したような既存資産をそのまま活かす方法は効果的です。しかし、グローバル化等の理由によりアプリケーション内で SJIS の範囲外の文字も扱う要件が出てくれば、プログラム自体も UTF-8 化を検討せざるを得ません。Visual COBOL は従来より提供される SJIS や EUC 等の２バイト環境における支援機能は維持しつつも、UTF-8 環境とも親和性の高い製品へと進化させています。影響箇所を正しく洗い出しメンテナンスすれば、既存資産のロジックを活かしつつ UTF-8 化が可能です。

まずは、UTF-8 化にあたり影響しうる主なポイントを以下に列挙します。

- ２バイト文字支援機能の利用
  - ＞ ２バイト文字項目の使用

PIC G, N USAGE DISPLAY-1 で定義する２バイト文字項目及び PIC N USAGE JAPANESE/NCHAR で定義する Micro Focus 拡張の２バイト文字項目は SJIS のような２バイト文字を扱うための項目であり、UTF-8 の文字を格納することができません。UTF-8 化した後もこれまでのように文字数で項目長を定義する型を利用し続ける必要があれば実行ロケールに関わらず内部表現を UCS2 とする PIC N USAGE NATIONAL という各国語データ項目を利用できます。例えば、現行

```
        01      DOUBLE-BYTE-ITEM     PIC N(5) VALUE N'あいうえお'.
```

のように記述しているコードに対しては、ソースを UTF-8 化した上で「USAGE NATIONAL」句を加えることで内部表現 USC2 として５文字格納させることも可能です。さらに、NSYMBOL(NATIONAL) コンパイル指令を指定するとプログラムの記述を変更することなく暗黙的に USAGE NATIONAL 句を指定することが可能です。現行のソースで USAGE DISPLAY-1 がコード書かれていれば、この部分は USAGE NATIONAL に書き換えるか、USAGE 句を削除し、NSYMBOL(NATIONAL) を指定します。このように UTF-8 のような可変の文字コードを持つ環境下でも日本語を文字数単位で扱うことが可能になります。ただし、サロゲートペア文字のように UCS2 でも４バイトで表現される文字については２文字分、つまり PIC N(2) 分のサイズが割り当てられます。

SJIS の２バイト文字項目を各国語の２バイト文字項目に切り替えることで、現行の２バイト文字項目に対する MOVE 文、STRING 文、UNSTRING 文、INSPECT 文のような文字列操作関連の COBOL 文のロジックを維持することが可能です。

各国語データ項目中で UCS2 で内部表現されるデータについては、DISPLAY-OF 組み込み関数を使うことでその実行時ロケールに合わせた符号へ変換することが可能です。例えば、下記のようなプログラムがあるとします。

```
       WORKING-STORAGE SECTION.
       01 DOUBLE-BYTE-ITEM PIC N(5) VALUE N'あいうえお'.
       PROCEDURE DIVISION.
           DISPLAY FUNCTION DISPLAY-OF(DOUBLE-BYTE-ITEM).
```

これを UTF-8 ロケール配下で実行すると、
DOUBLE-BYTE-ITEM には UCS2 における「あいうえお」 X' 3042 3044 3046 3048 304A'
が格納されていますが、DIPLAY 文中で FUNCTION DISPLAY-OF によりこれを UTF-8 における表現
X'E38182 E38184 E38186 E38188 E3818A' に変換しているため、UTF-8 環境下で正しく「あいうえお」が出力されます。

この逆方向の変換については NATIONAL-OF 組み込み関数で実現が可能です。

> N, G リテラルの利用

SJIS 環境では２バイト文字定数を表現する際、N もしくは G リテラルを使用して表現しますが、UTF-8 環境下ではこれらは利用できません。上述の NSYMBOL(NATIONAL) を指定してコンパイルすると、N リテラルは Unicode の定数、つまり UCS2 で表現する定数として扱われるようになります。従いまして、上の例中の VALUE 句「VALUE N'あいうえお'」では UCS2 で表現された文字定数として扱われています。これにより、 PIC N USAGE NATIONAL で定義した項目に正しくデータが移送されました。

> NCHAR 編集項目

PICTURE 句を N, B, /, 0 で構成して表現する NCHAR 編集項目については、SJIS や EUC のような２バイト文字環境を前提とした機能となります。この項目をプログラム中で利用している場合は、項目を分ける等の何らかの対応が必要となります。

> 表意定数 SPACE

２バイト文字項目に対して、DBSPACE 指令が有効な状態(Visual COBOL ではデフォルトで有効)で表意定数 SPACE を移送すると SJIS 環境では PIC N USAGE DISPLAY-1 に全角スペース(X'8140') が転記されます。USC2 で内部表現する各国語の変数 PIC N USAGE NATIONAL に対して表意定数 SPACE を転記しても半角スペースが格納されるため、全角スペースを転記したい場合は、例えば下記のようにして対応します。

SJIS 環境のコード：

```
       01 DOUBLE-BYTE-ITEM PIC N(5) VALUE SPACE.
```

UTF-8 移行後のコード：

```
       01 DOUBLE-BYTE-ITEM PIC N(5) VALUE ALL N'　'.
```

- <u>バイト数の変化による影響</u>

> 項目の定義

PIC X(n) 等の要領で定義する英数字項目や PIC A(n) 等の要領で定義する英字項目はバイト単位で長さを定義します。そのため、例えば日本語の「あ」を格納にするには SJIS では X'82A0' で割り当てられているため、２バイトの長さを用意すれば保持できます。一方、UTF-8 の場合は X'E38182' のように３バイトで割り当てられているため、３バイト分の長さを用意する必要があります。半角カナになると SJIS では１バイトで表現できていたところを UTF-8 になると３バイト分の長さが必要となります。移行後も同じデータをこれらの項目で保持させるには最大３倍の長さの項目定義が必要となります。

Visual COBOL は UTF-8 で符号化されたデータを英数字項目に、文字数でカウントしたり任意の文字位置で部分抽出する等、UTF-8 化を意識した特別な組み込み関数を幾つか提供しています [7]。これらを活用し UTF-8 化後の文字操作をシンプルなロジックで表現することも可能です。

> 部分参照

バイト単位で長さを定義する英字、英数字項目に日本語データを格納し、それを部分参照する処理がある場合は、文字のバイト長の変化を考慮する必要があります。一方、２バイト文字項目については上で紹介した USAGE NATIONAL の項目にすることで文字数単位による管理を維持できるため、開始位置や長さのパラメータを変更する必要はありません。

例えば、現行下記のような「あいうえお」から「うえ」だけを抽出する部分参照ロジックがあるとします。

```
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 AAA          PIC X(20) VALUE 'あいうえお'.
       01 BBB          PIC N(5)  VALUE N'あいうえお'.
       PROCEDURE DIVISION.
           DISPLAY AAA(5:4).
           DISPLAY BBB(3:2).
```

これを UTF-8 化する場合、下記のように英字・英数字項目については、UTF-8 の符号体系に合わせて開始位置、長さのパラメータを調整します。２バイト文字項目についてはコンパイラ指令 NSYMBOL(NATIONAL) を指定してプログラムに手を入れずに各国語の２バイト文字項目にすることでパラメータ部分への影響を回避することが可能です。

```
      $SET NSYMBOL(NATIONAL)
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 AAA          PIC X(20) VALUE 'あいうえお'.
       01 BBB          PIC N(5)  VALUE N'あいうえお'.
       PROCEDURE DIVISION.
           DISPLAY AAA(7:6).
           DISPLAY FUNCTION DISPLAY-OF(BBB(3:2)).
```

> 特殊レジスタ LENGTH OF

特殊レジスタ LENGTH OF は同特殊レジスタが指すデータ項目のバイト数を格納します。例えば、下記のようなコードについて考えてみますと、SJIS 環境下では 10 が DISPLAY 出力されますが、UTF-8 環境下では 15 が出力されます。

```
           DISPLAY LENGTH OF "あいうえお".
```

> LENGTH、LENGTH-AN 組み込み関数

ソース中の文字定数もソースの符号化に準じたバイト長で内部処理されます。LENGTH や LENGTH-AN 組み込み関数に下記のように文字定数で引数渡しすると、SJIS 環境下で実行すれば 10 が、UTF-8 環境下で実

[7] 詳細については製品マニュアルや別文書「COBOL による Unicode データ処理」
http://www.microfocus.co.jp/resources/whitepaper/pdf/white_paper_059.pdf
をご参照ください。

行すれば 15 が返ってきます。２バイト文字項目については、上と同様に USAGE NATIONAL で各国語の２バイト文字項目にすることで UTF-8 環境に移行後も同様に LENGTH 組み込み関数に文字数単位で計数させることが可能です。

```
          MOVE FUNCTION LENGTH("あいうえお") TO AAA.
```

> INSPECT 文

バイト長が直接影響する COBOL 文についても文字のバイトの変化による影響を考慮する必要があります。例えば、下記のような INSPECT 文を考えてみます。同ロジックは英数字項目 AAA に対して「お」が出てくるまでの文字数を数える処理を表したものとなります。検査対象は英数字項目となるため、「お」が出てくるまでのバイト数が BBB に格納されます。つまり、SJIS 環境下で実行すれば「あいうえ」のバイト数 8 が、UTF-8 環境下で実行すれば「あいうえ」のバイト数 12 が BBB に格納されます。

```
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 AAA           PIC X(20)  VALUE "あいうえお".
       01 BBB           PIC 9(4)   VALUE ZERO.
       PROCEDURE DIVISION.
           INSPECT AAA TALLYING BBB FOR CHARACTERS BEFORE INITIAL "お".
```

- その他

> 大小比較

SJIS と UTF-8 では文字に割り当てたコードも異なれば、コードの大小関係も異なるようです。例えば、漢数字「一」、「三」、「五」、「七」 は下表のように割り当てられています。

表２－１

	一	三	五	七
SJIS	88EA	8E4F	8CDC	8EB5
UTF-8	E4B880	E4B889	E4BA94	E4B883

この中から「三」と「五」に着目すると SJIS では「三」の方が「五」よりも割り当てられたコードは大きいですが、 UTF-8 になるとその逆となります。現行 SJIS に符号化された日本語文字を PIC X に格納し、IF 文等の条件式中で大小比較している場合、このような文字に該当すれば結果は異なります。ソート順についても同様となるため注意が必要です。例中の文字を SJIS に符号化されたデータとして昇順で並べると

一　五　三　七

となりますが、UTF-8 となりますと

一　七　三　五

の順となります。

> ソースのエンコード

プログラムソースのエンコード、コンパイル時のロケール、実行時のロケールは全て共通にする必要があります。そのため、現行 SJIS でエンコードされたソースを利用している場合は、コンパイル前に UTF-8 へ変換する必要があります。

おわりに

以上、Red Hat をはじめとした UTF-8 をシステムロケールの前提としている Linux OS 上で、SJIS のプラットフォーム上で開発された COBOL 資産をどのように有効活用できるか見てきました。Visual COBOL はプロジェクトの方針に応じて適切な手段を講じることができるよう、COBOL 資産を SJIS に符号化されたまま継続利用する方法、UTF-8 化する方法のいずれにも対応できるよう設計された製品です。アプリケーションは Linux へ移行はするが、COBOL アプリケーション自体は SJIS で表現できる文字の範囲内で継続運用できるのであれば、無理に COBOL 資産を UTF-8 化することは不要なリスクを招きます。この場合、第１章で紹介したような低リスク・低コストな手段は検討の価値が大いにあります。一方、現行 SJIS 上で運用されていても移行のタイミングでアプリケーション自体の国際化等も検討する必要があるのであれば第２章で紹介したポイントを整理し影響箇所を洗い出します。
近年のマーケットにおける競争激化等のあおりを受けて、TCO の削減は各企業にとって大きなテーマとなっています。クラウド環境と親和性の高い Linux OS の導入は今後も進むものと想定されます。このタイミングでハードウェアだけでなくアプリケーションをも正しく移行できればその効果は一層高まります。この COBOL アプリケーション移行の成功に際して、本書がその一助を担えれば幸いです。