New Technical Notes
Macintosh

®

Developer Support

Apple Script Manager Q&As
Interapplication Communication

Revised by: Developer Support Center   May 1993
Written by: Developer Support Center   October 1990

This Technical Note contains a collection of Q&As relating to a specific topic—questions you've sent the Developer Support Center (DSC) along with answers from the DSC engineers. While DSC engineers have checked the Q&A content for accuracy, the Q&A Technical Notes don't have the editing and organization of other Technical Notes. The Q&A function is to get new technical information and updates to you quickly, saving the polish for when the information migrates into reference manuals.

Q&As are now included with Technical Notes to make access to technical updates easier for you. If you have comments or suggestions about Q&A content or distribution, please let us know by sending an AppleLink to DEVFEEDBACK. Apple Partners may send technical questions about Q&A content to DEVSUPPORT for resolution.

String2Date and Date2Secs conversion surprises
Date Written: 8/19/91
Last reviewed: 9/17/91

String2Date and Date2Secs treat all dates with the year 04 to 10 as 2004 to 2010 instead of 1904 to 1910.

___

This is correct; the Script Manager treats two-digit years less than or equal to 10 as 20xx dates if the current year is between 1990 and 1999, inclusive. Basically, it just assumes that you're talking about 1-20 years in the future, rather than 80-100 years in the past. The same is true of two-digit 9x dates, when the current year is less than or equal to xx10. Thus, in 2003, the date returned when 3/7/94 is converted will be 1994, not 2094. This is all documented in "Worldwide Development: Guide to System Software," available from APDA.

Using FormatXToStr and FormatStrToX with Pascal switches
Date Written: 12/10/90
Last reviewed: 1/16/91

Why do the FormatXToStr and FormatStrToX Script Manager routines stop working when I use the Pascal -MC68881 switch?

___

Regular SANE extended numbers are 10 bytes long while MC68881 extended numbers are 12 bytes long, and the extra two bytes are right in the middle of every 68881 extended number. Appendix G "The SANE Library" in the Macintosh Programmer's Workshop (MPW) Object Pascal version 3.1 manual goes into detail about this. The FormatX2Str and FormatStr2X parse the extended number you pass them directly, and they can only parse 10-byte extended numbers. Fortunately, you can still use the -mc68881 option with these routines as long as you convert any extended numbers to 80-bit extended numbers before passing them to FormatX2Str and FormatStr2X. The SANE.p unit has routines to do this called X96toX80 and X80toX96 (incorrectly documented as X96to80 and X80to96 in the MPW Object Pascal manual). Because the extended80 and extended96 types aren't equivalent to the extended type as far as Object Pascal is concerned, you have to redeclare FormatX2Str and FormatStr2X to take these types. You can do this as follows:

```
FUNCTION FormatX2Str80 (x:              extended80;
                        myCanonical:   NumFormatString;
                        partsTable:    NumberParts;
                        VAR outString:  Str255):  FormatStatus;
   INLINE $2F3C,$8210,$FFE8,$A8B5;

FUNCTION FormatStr2X80 (source:        Str255;
                        myCanonical:   NumFormatString;
                        partsTable:    NumberParts;
                        VAR x:         extended80):  FormatStatus;
   INLINE $2F3C,$8210,$FFE6,$A8B5;
```

Call these routines instead of the originals. To call FormatX2Str80, all you have to do is this:

```
VAR
  x:            extended80; {96-bit extended number}
  myCanonical:  NumFormatString;
  partsTable:   NumberParts;
  outString:    Str255

result := FormatX2Str80 (X96toX80 (x), myCanonical, partsTable, outString);
```

Calling FormatStr2X80 is just slightly more complicated because the extended number is passed by reference:

```
VAR
  x:            extended;   {96-bit extended number}
  x80:          extended80; {80-bit extended number}
  source:       Str255;
  myCanonical:  NumFormatString;
  partsTable:   NumberParts;

x80 := X96toX80 (x);
result := FormatStr2X80 (theString, realCanon, PartsTable, x80);
x := X80toX96 (x80);
```

You should find that these calls now work properly with the -mc68881 option set. This of course means that you'll need two versions of the source code; one with the calls to convert between 96-bit and 80-bit extended numbers for use with the -mc68881 option and another one which just uses plain old 80-bit extended numbers for use when the -mc68881 option is turned off.

FormatX2Str strings

Date Written: 10/9/91
Last reviewed: 11/6/91

Using the Script Manager to convert numbers to strings and vice versa, in any language, what's the best way to create the string to pass to FormatX2Str? Will strings using the characters: "#" or "0" or "." or "," work no matter what script is currently running, and if not, what can I do?

———

The number format string and canonical number format string mechanisms that you use with FormatX2Str and its kin is a strange design, for exactly the reason that you asked about. The number format string (the one with the characters such as "#" and "0") does not necessarily work right regardless of the current script. In fact, it doesn't even necessarily work right between localized versions within one script system. The canonical number format string does work between localized systems and between script systems. The strange thing is there's an easy way to store number format strings (usually in a 'STR ' resource), but no obvious way to store canonical number format strings. Here's what you can do when converting between numbers and strings:

When you convert a number format string to a canonical number format string with Str2Format on a U.S. system, it converts it from something like "###.###" to a canonical number format string that looks something like, "three digits, a decimal point, and three digits." On a German system, that same number format string would be converted to "three digits, a thousands separator, and three digits."

What you can do to get around this is to save the canonical number format string in a resource instead of the number format string. The canonical string stores things in a language- and script-independent way. Create this resource by writing a trivial utility program that takes a number format string and calls Str2Format to convert it into a canonical number format string, and then copy this into a handle and save it as a resource of a custom type, like 'NUMF'. In your real program, load the 'NUMF' resource, lock it, and then pass the dereferenced handle to FormatX2Str and FormatStr2X.

You can see this done in the ProcDoggie Process Manager sample from the 7.0 Golden Master CD. Take a look at the SetUpProcessInfoItems procedure in UProcessGuts.inc1.p file. You'll see that the 'NUMF' resource is loaded, locked, and then passed to FormatX2Str. The result is displayed in the Process Information window.

If your program is localized by nonprogrammers, then you might want to provide the utility that converts a number format string to a canonical number format string resource just in case they have to change the entire format of the string. Then they can install the new 'NUMF' (or whatever you choose) resource as part of the localization process.

Code for truncating a multi-byte character string

Date Written: 1/24/92
Last reviewed: 4/7/92

I create a Macintosh file name from another file name. Since I am adding information to the name, I must make sure that it is within the 31 chars maximum allowed by the operating system. What I need is the equivalent of the TruncText command, except instead of dealing

with pixel width, I want the width to be number of characters (31). I can trunc myself, but I'd rather do a proper "smTruncMiddle" and have it nicely internationalized.

———

If you're going to be adding a set number of bytes to the end of a existing string and you don't want the localized ellipsis (from the 'itl4' resource) between the truncated string and your bytes, then you can use this routine:

```
PROCEDURE TruncPString (VAR theString: Str255; maxLength: Integer);
{ This procedure truncates a Pascal string to be of length maxLength or }
{ shorter. It uses the Script Manager charByte function to make sure }
{ the string is not broken in the middle of a multi-byte character. }
 VAR
   charType: Integer;
 BEGIN
   IF Length(theString) > maxLength THEN
     BEGIN
       charType := CharByte(@theString[1], maxLength);
       WHILE ((charType < 0) OR (charType > 1)) AND (maxLength <> 0) DO
         BEGIN
           maxLength := maxLength - 1;
           charType := CharByte(@theString[1], maxLength);
         END;
       theString[0] := chr(maxLength);
     END;
 END;
```

If you want the localized ellipsis (from the 'itl4' resource) between the truncated string and your bytes, or you want the localized ellipsis in the middle of the combined strings truncated to a specific length, then you can use this routine:

```
FUNCTION TruncPString (maxLength: Integer; VAR theString: Str255;
truncWhere: TruncCode): Integer;
{ This function truncates a Pascal String to be of length maxLength or }
{ shorter. It uses the Script Manager TruncString function which adds }
{ the correct tokenEllipsis to the middle or end of the string. See }
{ Inside Macintosh Volume VI, pages 14-59 and14-60 for more info. }
 VAR
   found: Boolean;
   first, midPoint, last: Integer;
   tempString: Str255;
   whatHappened: Integer;
 BEGIN
   found := FALSE;
   first := 0;
   last := TextWidth(@theString[1], 0, Length(theString));
   IF Length(theString) > maxLength THEN
     BEGIN
       WHILE (first <= last) AND NOT found DO
         BEGIN
           tempString := theString; { tempString gets destroyed every }
                                     { time through }
           midPoint := (first + last) DIV 2;
           whatHappened := TruncString(midPoint, tempString, truncWhere);
           IF whatHappened < smNotTruncated THEN
             BEGIN { ERROR, bail out now }
               TruncPString := whatHappened; { return error }
               Exit(TruncPString);
             END
           ELSE IF Length(tempString) = maxLength THEN
```

```
        found := TRUE
      ELSE IF Length(tempString) > maxLength THEN
        last := midPoint - 1
      ELSE
        first := midPoint + 1;
    END;
  theString := tempString;
  TruncPString := whatHappened; { will always be smTruncated }
                                { in this case }
    END
  ELSE
    TruncPString := smNotTruncated; { the string wasn't too long }
END;
```

Character type and subtype values within the Kanji system

Date Written: 11/17/89
Last reviewed: 12/17/90

What are the values of character type and subtype with the Macintosh Kanji system?

——

For Roman, these are the values of character type:

```
    Punctuation        0
    ASCII              1
    European           7
```

For KanjiTalk, the values are the same as Roman, with the addition of:

```
    Katakana           2
    Hiragana           3
    Kanji              4
    Greek              5
    Russian (Cyrillic) 6
```

In Roman, the subtype field is interpreted as:

```
    Normal punctuation 0
    Numeric            1
    Symbols            2
    Blanks             3
```

The KanjiTalk subtype values are the same as Roman except if the character type is Kanji, in which case the subtype field takes these values:

```
    JIS Level 1        0
    JIS Level 2        1
    JIS User Character 2
```

Finally, for KanjiTalk, the character direction field is replaced by the In-ROM field. It is 1 if the character is in the ROM card and 0 otherwise.