Inside Macintosh Interim Chapter Draft

Script Manager 2.0

Also refer to Inside Macintosh, Volume V, The Script Manager and Script Manager Hints & Recommendations (available from APDA)

Written by:	Mark Davis	March 15, 1988
Last Revision:	Sue Bartalo	October 3, 1988

This note describes the additional utility routines added to the Script Manager 2.0. It includes extended date & time utility routines, general-purpose number formatting routines, and additional text manipulation routines.

---

1. Overview

The Script Manager 2.0 release extends the tools and capabilities of developers on the Macintosh for three areas: text, dates and numbers. In addition, some minor bugs were fixed and performance enhancements incorporated.

The new text routines include: lexically interpreting different scripts (e.g. in macro languages); allotting justification to different format runs within a line; ordering format runs properly with bidirectional text (Hebrew & Arabic); quickly separating Roman from non-Roman text, and determining word-wrap in text processing. The international utilities text comparison routines were significantly improved in performance, in amounts ranging from 25% to 94% (see below).

The Macintosh date routines are extended to provide a larger range (roughly 35 thousand years), and more information. This allows programs that need a larger range of dates to use system routines rather than produce their own, which may not be internationally compatible. The programmer can also access the stored location (latitude and longitude) and time zone of the Macintosh from parameter RAM. The Map CDEV gives users the ability to change and reference these values.

The new number routines supplement SANE, allowing applications to display formatted numbers in the manner of Microsoft Excel or Fourth Dimension, and to *read* both formatted and simple numbers. The formatting strings allow natural display and entry of numbers and editing of format strings even though the original numbers *and* the format strings were entered in a language other than the final user's.

2. Implementation Notes

Some of the following routines have parameter blocks with reserved fields. ***These fields must be zeroed!***

In general, the additional routines are handled by the Script Manager rather than script interface systems. The three exceptions are FindScriptRun, PortionText, and VisibleLength which are handled by the individual script systems (such as Roman). The version of the Script Manager can be checked before using any of these routines, to make sure that it is Script Manager 2.0 (version is $0200 or greater).

For compatibility, all Script Systems test the version of the Script Manager, and do not initialize if the major version number (first byte) is greater than they expect. When the script systems are revised for the 6.0 System disk (and include the three additional script-specific routines), they will test for the next version number.

For testing only, the version number in INIT 2 can be changed in ResEdit or FEdit in the resource header to enable those systems to run; the header has the following format:

60xx	Branch
xxxx	Flags word
4943	Resource type (INIT)
4954	
0002	Resource number (2)
02xx	Script Manager version: change to 01FF for testing

For an old script, the three routines FindScriptRun, PortionText, and VisibleLength will not work at all. In addition, the itl4 resource (see below) for the script will not be present, so the IntlTokenize and number formatting routines will not work properly for the particular script's features.

The results returned from the new function calls are error and status codes which are listed in the interface section at the end of this document.

Note that in the following text, the term *language* generally refers to a natural language rather than a programming language!

3. Itl4 Resource

There is a new international resource, itl4, which contains information used by several of these routines and must be localized for each script (including Roman).

```
Itl4Rec   = RECORD
                flags:                  integer;
                resourceType:           longInt;
                resourceNum:            integer;
                version:                integer;
                resHeader1:             longInt;
                resHeader2:             longInt;
                numTables:              integer;        { one-based }
                mapOffset:              longInt;        { offsets are from record start }
                strOffset:              longInt;
                fetchOffset:            longInt;
                unTokenOffset:          longInt;
                defPartsOffset:                 longInt;
                resOffset6:             longInt;
                resOffset7:             longInt;
                resOffset8:             longInt;
                                        { the rest is data pointed to by offsets}
END;

Itl4Ptr =       ^Itl4Rec;
Itl4Handle =    ^Itl4Ptr;
```

4. Text

The new text routines include: lexically interpreting different scripts (e.g. in macro languages); allotting justification to different format runs within a line; ordering format runs properly with bidirectional text (Hebrew & Arabic); quickly separating Roman from non-Roman text, and determining word-wrap in text processing. The international utilities text comparison routines were significantly improved in performance, in amounts ranging from 25% to 94% (see below).

a. Parse Table

```
Type
          CharByteTable = Packed Array [0..255] of SignedByte;

Function ParseTable(table: CharByteTable): Boolean;
          INLINE $2F3C, $8204, $0022, $A8B5;
```

Double-byte characters have distinctive high (first) bytes, which allows them to be distinguished from single-byte characters. The ParseTable routine can be used to traverse double-byte text quickly. It does this by filling a table of bytes with values which indicate the extra number of bytes taken by a given character. This array can then be used instead of making function calls on each byte. As with the other script-specific routine calls, the values in the table will vary with the script of the current font in thePort, so you must make sure to set the font correctly.

An entry in the table is set to 0 for a single-byte character, and 1 for the first byte of a double-byte character. (With a single-byte script, the entries are all zero.) The return value from the routine will always be true. This routine has always been present in the Script Manager, but has not been documented until now. Also note that script systems will never require more than two bytes per character, so you can safely assume that there are only single and double byte characters.

For example, in the following code the reference to tablePtr[myChar] is functionally equivalent to a use of CharByte, but does not involve a trap call.

```
Var
          myChar:         Integer;
          i, max:         Integer;
          tablePtr:       CharWidthTable;
          s:              String [255];
          parseResult:    Boolean;
Begin
          parseResult := ParseTable(tablePtr);
          i := 1;
          max := length (s);
          While i <= max do Begin
                 myChar := ord(s[i]);                          {get byte}
                 i := i + 1;                                   {skip to start of next}
                 if (tablePtr[myChar] <> 0) then Begin{if double-byte}
                        myChar := myChar * $100 + ord(s[i]);{include next byte}
                        i := i + 1;                            {skip to start of next}
                 End;
                 {do something with myChar}
          End;
End;
```

Remember that the CharByteTable is specific to the script! There could be two or three scripts installed that are double-byte and have different CharByteTables.

b. IntlTokenize

```
Function IntlTokenize ( tokenParam : TokenBlockPtr  ): TokenResults;
```

The IntlTokenize routine is intended for use in macro expressions and similar programming constructs intended for general users. It allows the program to recognize variables, symbols and quoted literals without depending on the particular natural language (e.g. English vs. Japanese).
The routine is a mildly programmable regular expression recognizer for parsing text into tokens. The single parameter is a parameter block describing the text to be tokenized, the destination of the token stream, the itl4 resource handle, and the various programmable options. IntlTokenize will return a list of tokens found in the text.

```
        TokenBlock = RECORD
                source:  Ptr;                   {pointer to stream of characters}
                sourceLength: LongInt;          {length of source stream}
                tokenList: Ptr;                         {pointer to array of tokens}
                tokenLength: LongInt;           {maximum length of TokenList}
                tokenCount: LongInt;            {number of tokens generated by tokenizer}
                stringList: Ptr;                        {pointer to stream of identifiers}
                stringLength: LongInt;          {length of string list}
                stringCount: LongInt;           {number of bytes currently used}
                doString: Boolean;              {make strings & put into StringLIst}
                doAppend: Boolean;              {append to TokenList rather than replace}
                doAlphanumeric: Boolean;                {identifiers may include numeric}
                doNest: Boolean;                        {do comments nest?}
                leftDelims, rightDelims: ARRAY[0..1] OF TokenType;
                leftComment, rightComment: ARRAY[0..3] OF TokenType;
                escapeCode: TokenType;          {escape symbol code}
                decimalCode: TokenType;                 {decimal symbol code}
                itlResource: Handle;            {itl4 resource handle of current script}
                reserved: array [0..7] of Longint;      { must be zeroed! }
        END;

TokenType = Integer;            {see list of TokenType values at end of document}
TokenRec = RECORD
        theToken:       TokenType;
        position:       Ptr;            {ptr into original source}
        length:                 LongInt;                {length of text in original source}
        stringPosition:         StringPtr;      {Pascal/C string copy of identifier}
END;
```

For the *TokenBlock* record:

source is a pointer to the beginning of a stream of characters (not a Pascal string).

sourceLength is the number of characters in the source stream.

tokenList is a pointer to memory allocated by the application for the token stream. The tokenizer places the tokens it generates at and after the address in tokenList.

tokenLength is the number of tokens that will fit in the memory pointed to by tokenList (not the number of bytes).

tokenCount is the number of tokens that are currently occupying the space pointed to by tokenList. If the doAppend flag is true then tokenCount must be a correct number before calling the tokenizer. The tokenizer modifies this value to show how many tokens are in the token stream after tokenizing.

stringList is a pointer to memory allocated by the application for strings that the tokenizer generates if the doString flag is true. If the flag is false then stringList is ignored.

stringLength is the number of bytes of memory allocated for stringList.
stringCount is the number of bytes that are currently occupying the space pointed to by stringList. If the doAppend flag is true then stringCount must be a correct number before calling the tokenizer. The tokenizer modifies this value to show how many bytes are in the string stream after tokenizing.

doString is a boolean flag that instructs the tokenizer to create a sequence of even boundaried, null terminated Pascal strings. Each token generated by the tokenizer will have a string created to represent it if the flag is true. Each token record contains the address of the string that represents it.

doAppend is a boolean flag that instructs the tokenizer to append tokens to the space pointed to by tokenList rather than replace whatever is there. tokenCount must correctly reflect the number of tokens in the space pointed to by tokenList.

doAlphanumeric is a boolean flag that when true states that numerics may be mixed with alphabetics to create alphabetic tokens.

doNest is a boolean flag that instructs the tokenizer to allow nested comments of any depth.

leftDelims is an array of two integers, each of which corresponds to the class of the symbol that may be used as a left delimiter for a quoted literal. Double quotes, for instance, is class token2Quote. If only one left delimiter is needed, the other must be specified to be delimPad .

rightDelims is an array of two integers, each of which corresponds to the class of the symbol that may be used as the matching right delimiter for the corresponding left delimiter in leftDelims.

leftComment is an array of four integers. Each successive pair of two describes a pair of tokens that may be used as left delimiters for comments. These tokens are stored in reverse order. The 0th and 2nd tokens are the second tokens of the two-token sequences; the 1st and 3rd tokens are the first tokens of the two-token sequences.

If only one token is needed for a delimiter, the second token must be specified to be delimPad . If only one delimiter is needed then both of the tokens allocated for the other symbol must be delimPad . The first token of a two-token sequence is the higher position in the array. For example, the two left delimiters (* and { would be specified as

```
leftComment[0]:= tokenAsterisk      (*asterisk*)
leftComment[1]:= tokenLeftParen;    (*left parenthesis*)
leftComment[2]:= delimPad ;         (*nothing*)
leftComment[3]:= tokenLeftCurly;    (*curly brace*)
```

rightComment is an array of four integers with similar characteristics as leftComment. The positions in the array of the right delimiters must be the same as their matching left delimiters.

escapeChar is a single integer that is the class of the symbol that may be used for an escape character. The tokenizer considers the escape character to be an escape character (as opposed to being itself) only within quoted literals.

decimalCode is a single integer that is the tokenType that may be used for a decimal point. The tokenizer considers the decimal character to be a decimal character (as opposed to being itself) only when flanked by numeric or alternate numeric characters, or when following them. When the strings option is selected, the decimal character will always be transliterated to an ASCII period (and alternate numbers will be transliterated to ASCII digits).

itlResource is a pointer to the itl4 resource of the script in current use. The application must load the itl4 resource and place its handle here before calling the tokenizer. Everytime the script of the text to be tokenized changes, the pointer to the respective itl4 resource must be placed here.

reserved locations must all be zeroed.

For the *token* record:

theToken is the ordinal value of the token represented by the token record.

position points to the first character in the original text that caused this particular token to be generated.

length is the length in bytes of the original text corresponding to this token.
stringPosition points to a null-terminated, even-boundaried Pascal string that is the result of using the doString option. If doString is false then stringPosition is always set to NIL.

The available token types are: *whitespace*, *newline*, *alphabetic*, *numeric*, *decimal*, *endOfStream*, *unknown*, *alternate numeric*, *alternate decimal*, and a host of fixed token symbols, such as *(# @ : := .*

The tokenizer does not attempt to provide complete lexical analysis, but rather offers a programmable "pre-lex" function whose output should then be processed by the application at a lexical or syntactic level.

The programmable options include: whether to generate strings which correspond to the text of each token; whether the current tokenize call is to append to, rather than replace, the current token list; whether alphabetic tokens may have numerics within them; whether comments may be nested; what the left and right delimiters for comments are (up to two sets may be specified); what the left and right delimiters for quoted literals are (up to two sets may be specified); what the escape character is; and what the decimal point symbol is.

Some users may use two or more different scripts within a program. However, each script's character stream must be passed separately to the tokenizer because different resources must be passed to the tokenizer depending on the script of the text stream. Appending tokens to the token stream lets the application see the tokens generated by the different scripts' characters as a single token stream. Restriction: users may not change scripts within a comment or quoted literal because these syntactic units must be complete within a single call to the tokenizer in order to avoid tokenizer syntax errors.

The application may specify up to two pairs of delimiters each for both quoted literals and comments. Quoted literal delimiters consist of a single symbol, and comment delimiters may be either one or two symbols (including newline for notations whose comments automatically terminate at the end of lines). The characters that compose literals within quoted literals and comments are normally defined to have no syntactic significance; however, the escape character within a quoted literal does signal that the following character should not be treated as the right delimiter. Each delimiter is represented by a token, as is the literal between left and right delimiters.

If two different comment delimiters are specified by the application, then the doNest flag always applies to both. Comments may be nested if so specified by the doNest flag with one restriction that must be strictly observed in order to prevent the tokenizer from malfunctioning: ***nesting is legal only if both the left and right delimiters for the comment token are composed of two symbols each.*** In this version there is limited support for nested comments. When using this feature, test to ensure that it meets your requirements.

An escape character between left and right delimiters of a quoted literal signals that the following character is not the right delimiter. An escape character is not specially recognized and has no significance outside of quoted literals. When an escape character is encountered, the portion of the literal before the escape is placed into a single token, the escape character itself becomes a token, the character following the escape becomes a token, and the portion of the literal following the escape sequence becomes a token.

A sequence of whitespace characters becomes a single token.

Newline, or carriage return, becomes a single token.

A sequence of alphabetic characters becomes an alphabetic token. If the doAlphanumeric flag is set, then alphabetic characters include digits, but the first character must be alphabetic.

A sequence of numeric characters becomes a numeric token.

A sequence of numeric characters followed by a decimal mark, and optionally followed by more numeric characters, becomes a realNumber token.

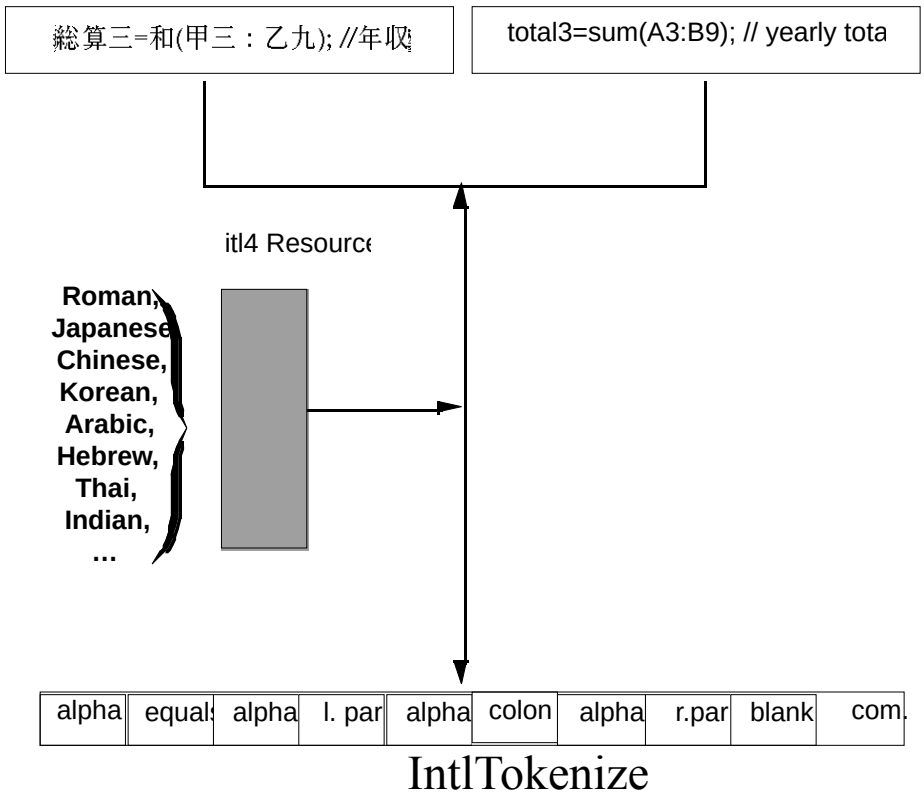
Some scripts have not only “English” digits, but also their own numeral codes, which of course will be unrecognizable to the typical application. A sequence of alternate digits becomes an alternate numeric token. If the strings option is selected then the digits will be transliterated to “English” digits. This includes the realNumber tokens, whose results become alternate real tokens.

The end of the character stream becomes a token.

A token record consists of a token code, a pointer into the source stream (signifying the first character of the sequence that generated the token), the byte length of the sequence of characters that generated the token, and space for a pointer to a Pascal string, explained next.

The application may instruct the tokenizer to generate null-terminated, even-boundaried Pascal strings corresponding to each token. In this case, if the token is anything but alphabetic or numeric then the text of the source stream is copied verbatim into the Pascal string. Otherwise, if the text in the source stream is roman letters or numbers then those characters are transliterated into Macintosh 8-bit ASCII and a string is created from the result, allowing users of other languages to transparently use their own script’s numerals or roman characters for numbers or keywords. Non-roman alphabetics are copied verbatim.

Semantic attributes of byte codes vary from natural language to natural language. As an example, in the Macintosh character set code $81 is an Å, but in Kanji this code is the first byte of many double-byte characters, some of which are alphabetic, some numeric, and some symbols. This information is retrieved from the *itl4* resource, which also contains a canonical string format for the fixed tokens, so that the internal format of formulæ can be redisplayed in the original language.
Itl4 also holds a string copy routine which converts the native text to the corresponding English (except for alphanumerics). As with the other international resources, the choice of *itl4* depends on the script interface system in use.



The untokenTable in the itl4 resource contains standard representations for the fixed tokens, and can be used to display the internal format. An example of how a user might access this table and use the token information follows:

```
Type
        UntokenTable = Record
                len: Integer;
                lastToken: Integer;
                index: array [0..255] of Integer; {index table; last = lastToken}
                {list of pascal strings here. index pointers are from front of table}
        End;
        UntokenTablePtr = ^UntokenTable;
        UntokenTableHandle = ^UntokenTablePtr;

Function GetUntokenTable( Var x: UntokenTableHandle ): Boolean;
Var
        itl4: itl4Handle;
        p: UntokenTablePtr;
Begin
        GetUntokenTable := false;                       {assume error}
        itl4 := itl4Handle(IUGetIntl(4));               {get itl4 record}
        if itl4 <> nil then begin                       {if ok}
                HLock(Handle(itl4));                    {lock for safety}
                p := UntokenTablePtr(ord(itl4^)+itl4^^.untokenOffset);
                                                {untokenize parts subtable}
```

```
                With p^ Do Begin                                {using resource table}
                        x := UntokenTableHandle(NewHandle(len));
                                                        {make handle of proper size}
                        BlockMove(Ptr(p),Ptr(x^),len);          {copy contents}
                End;
                HUnlock(Handle(itl4));                  {free back up}
                GetUntokenTable := true;                        {no error}
        End;
End;


If (GetUntokenTable(myUntokenTable)) then
        With curToken^ Do Case theToken OF
                {. . .}
                tokenAlpha:
                        AppendString( myVariable[i] );
                Otherwise With myUntokenTable^^, curToken^ Do Begin
                        If theToken > lastToken Then Begin
                                AppendString( '?' );
                        End Else Begin
                                sPtr := pointer(ord(@len) + index[theToken]);
                                AppendString(sPtr^);
                        End; {if}
                End; {item}
        End; {case}
```

c. PortionText

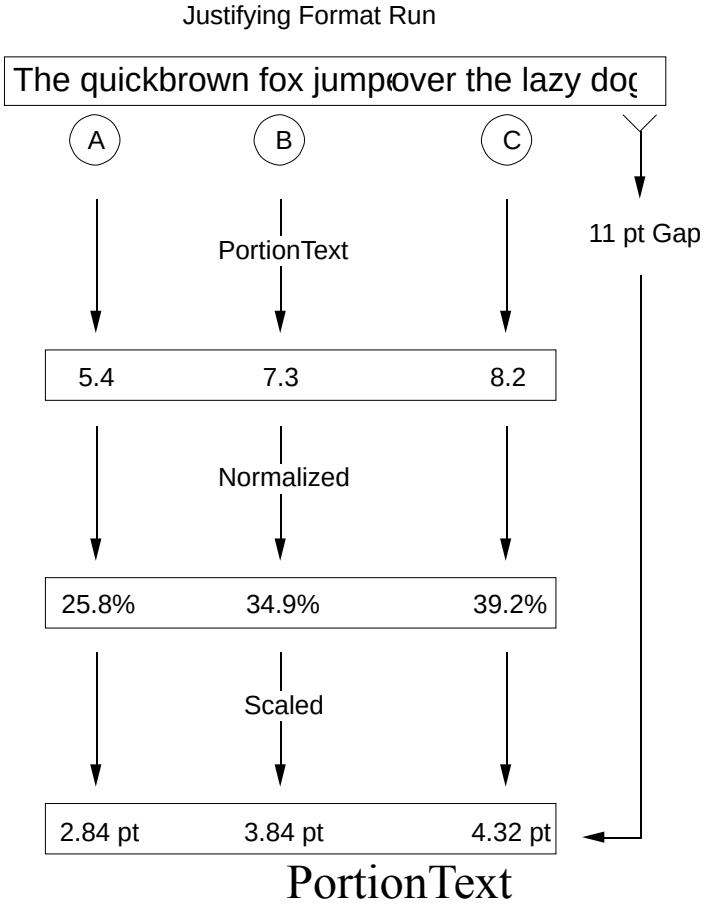
```
Function PortionText (  textPtr : Ptr; textLen : Longint): Fixed; {proportion}
```

This routine returns a result which indicates the proportion of justification that should be allocated to this text when compared to other text. It is used when justifying a sequence of format runs, so that the appropriate amount of extra width is apportioned properly among them. For example, suppose that there are three format runs on a line: A, B, and C. The line needs to be widened by 11 pixels for justification. Calling PortionText on these format runs yields the first row in the following table:

	A	B	C	Total
PortionText:	5.4	7.3	8.2	20.9
Normalized:	.258	.349	remainder	1.00
Pixels (p):	2.84	3.84	remainder	11.0
Rounded (r):	3	4	remainder	11

The proportion of the justification to be allotted to A is 25.8%, so it receives 3 pixels out of 11. In general, to prevent rounding errors, $r_n = \text{round}(\sum_{1..n} p) - \sum_{1..n-1} r$ (which can be computed iteratively); e.g., r_B is round(3.84+2.84) – 3, and r_C is round(11.0) – 7.

For normal Roman text, the result is currently a function of the number of spaces in the text, the number of other characters in the text, and the font size [the raw size, not ascent + descent + leading]. This may change in the future, so values should be compared at the time of execution.



d. Format Order

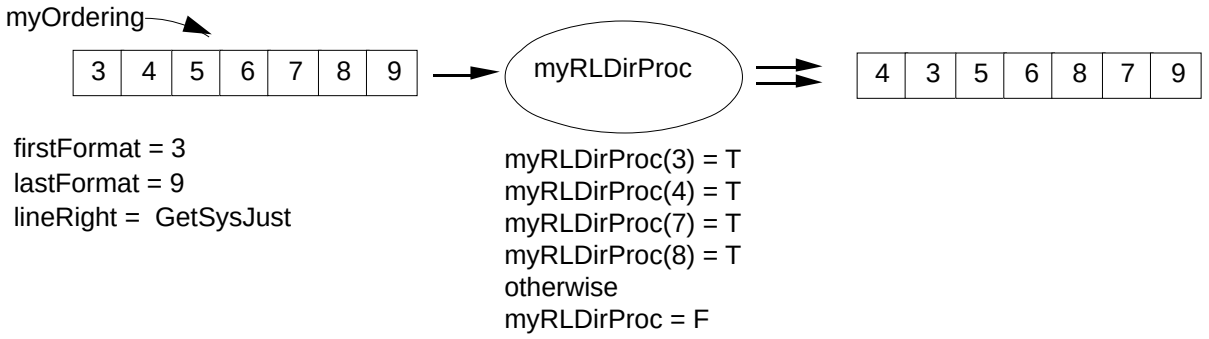
```
FormatOrder = array [0..0] of Integer;
FormatOrderPtr = ^FormatOrder;

Procedure GetFormatOrder (      ordering:      FormatOrderPtr;
                                firstFormat:   Integer;
                                lastFormat:    Integer;
                                lineRight:     Boolean;
                                RLDirProc:     Ptr;
                                dirParam:      Ptr);
```

This routine orders the text properly for display of bidirectional format runs. Word processing programs that use this procedure for multi-font text can be independent of script text-ordering in a line (e.g. Hebrew or Arabic right-left text). The ordering points to an array of integers, with (lastFormat – firstFormat + 1) entries. The GetFormatOrder routine retrieves the direction of each format by calling the direction procedure, RLDirProc, which has the following format:

```
Function MyRLDirProc (  theFormat :     Integer;
                        dirParam :      Ptr)
         :Boolean;
```

The RLDirProc is called with the values from firstFormat to lastFormat to determine the directions of each of the format runs. It returns true for right-left text direction, otherwise false. The parameter dirParam is available to provide other necessary information for the direction procedure (i.e., style number, pointer to style array, etc).

GetFormatOrder returns a permuted list of the numbers from firstFormat to lastFormat. This permuted list can be used to draw or measure the text. (For more detail, see the Script Manager developers' packet). The lineRight parameter is true if the text is right-left orientation, otherwise false.



GetFormatOrder

For example:

```
GetFormatOrder(myOrdering,firstFormat,lastFormat,GetSysJust,MyRLDirProc,nil);
for i := 0 to lastFormat-firstFormat do
         with MyFormat [myOrdering [i]], MyStyle [formatStyle] do begin
                  TextFont(styleFont);
                  {set up other text style features...}
                  case what of
                           drawing: DrawText(textStartPtr, formatStart, formatLength);
                           measuring: TextWidth(textStartPtr, formatStart, formatLength);
                           {and so on}
                  end; {case}
         end; {with}
end; {for}
```

e. FindScriptRun

```
Function FindScriptRun (        textPtr:        Ptr;
                                textLen:        Longint;
                        Var     lenUsed:        Longint)
         : ScriptRunStatus;

ScriptRunStatus = record
         script: SignedByte;
         Variant:       SignedByte;
End;
```

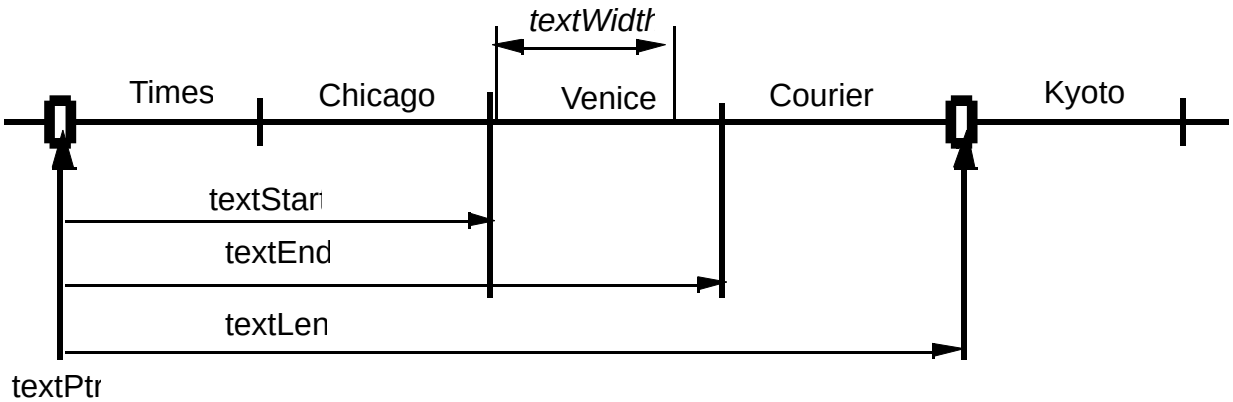
For compatibility, each script allows Roman text to be mixed in. This routine is used to break up mixed text (Roman & Native) into blocks. The lenUsed is set to reflect the length of the remaining text. The return value reflects the type of block: the upper byte is the script (0 being Roman text) and the lower byte being script-specific (script systems can return types of native sub-scripts, such as Kanji, Katakana and Hiragana for Japanese). For example, given that the capital letters represent Hebrew text:

```
myCharArray = 'abcDEFghi';
myCharPtr := @myCharArray;
blockType := FindScriptRun (myCharPtr, 9, lenUsed);
{lenUsed = 3, blockType = 0:  get remainder of text with: }
textPtr := ptr(ord(textPtr)+lenUsed);
textLen := textLen-lenUsed;
```

f. StyledLineBreak

```
Function StyledLineBreak(       textPtr:                Ptr;
                        textLen:        Longint;
                        textStart:      Longint;
                        textEnd:                Longint
                        flags:          Longint;
                Var     textWidth:      Fixed; {on exit, set if too long}
                Var     textOffset:     Longint)
         :StyledLineBreakCode;
```

This routine breaks a line on a word boundary. The user will loop through a sequence of format runs, resetting the *textPtr* and *textLen* each time the script changes; and resetting the *textStart* and *textEnd* for each format run. The *textWidth* will automatically be decremented by *StyledLineBreak* . *TextPtr* points to the start of the text, *textLen* indicates the maximum length of the text, and the *textWidth* parameter indicates the maximum pixel width of the rectangle used to display the text starting at the *textStart* and ending at the *textEnd*. The *flags* parameter is reserved for future expansion and must be zero.



StyledLineBreak

On input, a non-zero*textOffset* indicates whether this is the first format run (possibly forcing a character break rather than a word break: if textOffset is non-zero, at least one character will be returned if the line is not empty). On output it is the number of bytes from *textPtr* up to the point where the line should be broken. If the passed *textWidth* extended beyond the end of the text (i.e., is larger than the width from *textoffset* to *textLen*), then the width of the text is subtracted from the

textWidth and the result returned in the *textWidth* parameter. This can be used for the next format run.

The routine result indicates whether the routine broke on a word boundary, character boundary, or the width extended beyond the edge of the text.

When used with single-format text, the *textStart* can be zero, and the *textEnd* identical with the *textLen*. With multi-format text, the interval between *textStart* and *textEnd* specifies a format run. The interval between *textPtr* and *textLen* specifies a script run (a contiguous sequence of text where the script of each of the format runs is the same). Note that the format runs in *StyledLineBreak* **must** be traversed in back-end storage order, **not** display order (see GetFormatOrder).

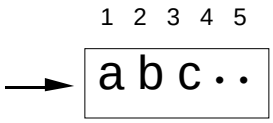
In other words, if the current format run is included in a contiguous sequence of other format runs of the same script, then the *textPtr* should point to the start of the first format run of the same script, while the *textLen* should include the last format run of the same script. This is so that word boundaries can extend across format runs; they will **never** extend across script runs.

Although the offsets are in longints and widths in fixed for future extensions, in the current version the longint values should be restricted to the integer range, and only the integer portion of the widths will be used.

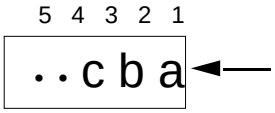
g. VisibleLength

```
FUNCTION VisibleLength (textPtr :      Ptr;
                        textLen:               Longint)
         : Longint;
```

This routine returns the length of the text excluding trailing white space, taking into account the script of the text. Trailing white space is only excluded if it occurs on the visible right side, in display order.



VisibleLength of this left-right example = 3.



VisibleLength of this right-left example = 5.

For example:

```
myVisibleLength := VisibleLength(myText,myOffset);
curSlop := myPixel - TextWidth(myText,0,myVisibleLength);
DrawJust(myText,myVisibleLength,curSlop);
```

h. Changing Text Case

```
Procedure UprText(textPtr: Ptr; len: Integer);
Procedure LwrText(textPtr: Ptr; len: Integer);
```

UprText provides a Pascal interface to the UprString assembly routine, which will uppercase text up to 32K in length. The LwrText routine provides the corresponding lowercase routine. Both of these routines will not change the number or position of characters in a string, but are faster and simpler than the Transliterate routine.

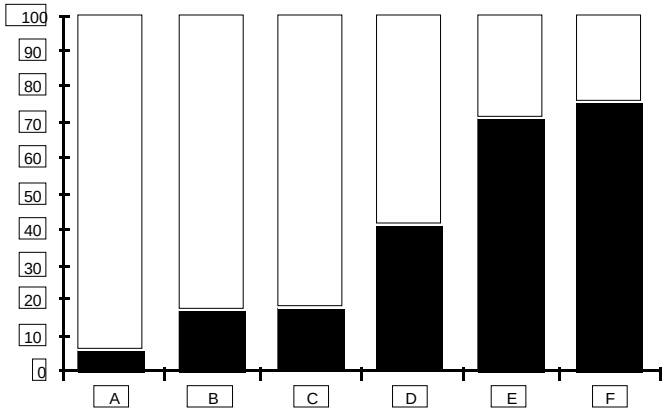
i. Text Comparison

We have done some performance analyses of Pack6 comparison routines, and based upon those, were able to increase performance by about 50% on average (see below). This results in a corresponding increase in 4th Dimension sorting performance, for example. Also, a long-standing bug in sorting *œ* and *Œ* has been corrected. A test program on the Mac SE comparing "The quick brown fox jumped over the lazy dog" to variants produced the following decreases in comparison time:

Identical text:	94%	
Last Character Unequal (g vs. X)	83%	
Last Character Weakly Equal (g vs. G):	82%	
First Character Unequal (T vs X):		59%
First Character Weakly Equal (T vs t):	29%	
All Characters Weakly Equal (T vs t…g vs. G):	25%	

Part of the performance increase results from internal caching of itl resources. Originally all itl resources (resulting from IUGetIntl of 0,1,2,4) were cached, but several programs do a ReleaseResource or DetachResource on itl0, rendering the cache invalid. Because of this, currently only itl2 and itl4 are cached. Developers must be sure not to release or detach these resources. Also, only the system file resources are used, so they cannot be overriden by copies in the application or document resource forks.

	"The quick brown fox jumped over the lazy dog"
A. Identical	"The quick brown fox jumped over the lazy dog"
Last Char	
B. Unequal	"The quick brown fox jumped over the lazy doX"
C. Similar	"The quick brown fox jumped over the lazy doG"
First Char	
D. Unequal	"Xhe quick brown fox jumped over the lazy dog"
E. Similar	"the quick brown fox jumped over the lazy dog"
All Chars	
F. Similar	"THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"



International Text Comparison

5. Dates

The Macintosh date routines are extended to provide a larger range (roughly 35 thousand years), and more information. This allows programs that need a larger range of dates to use system routines rather than produce their own, which may not be internationally compatible. The programmer can also access the stored location (latitude and longitude) and time zone of the Macintosh from parameter RAM. The Map CDEV gives users the ability to change and reference these values.

The long internal format of a date is as before, in seconds since 12:00 midnight, Jan 1, 1904, but is represented as a *signed* 64 bit integer (SANE Comp format), allowing a somewhat larger range (roughly 500 billion years). Short internal format dates (since they are unsigned) can be converted to long format by filling the top 32 bits with zero; long formats can be converted to short by truncating (assuming that they are within range). When storing in files, a five (or six) byte format can be used for a range of roughly 35 thousand years. This value should be sign-extended to restore it to a Comp format.

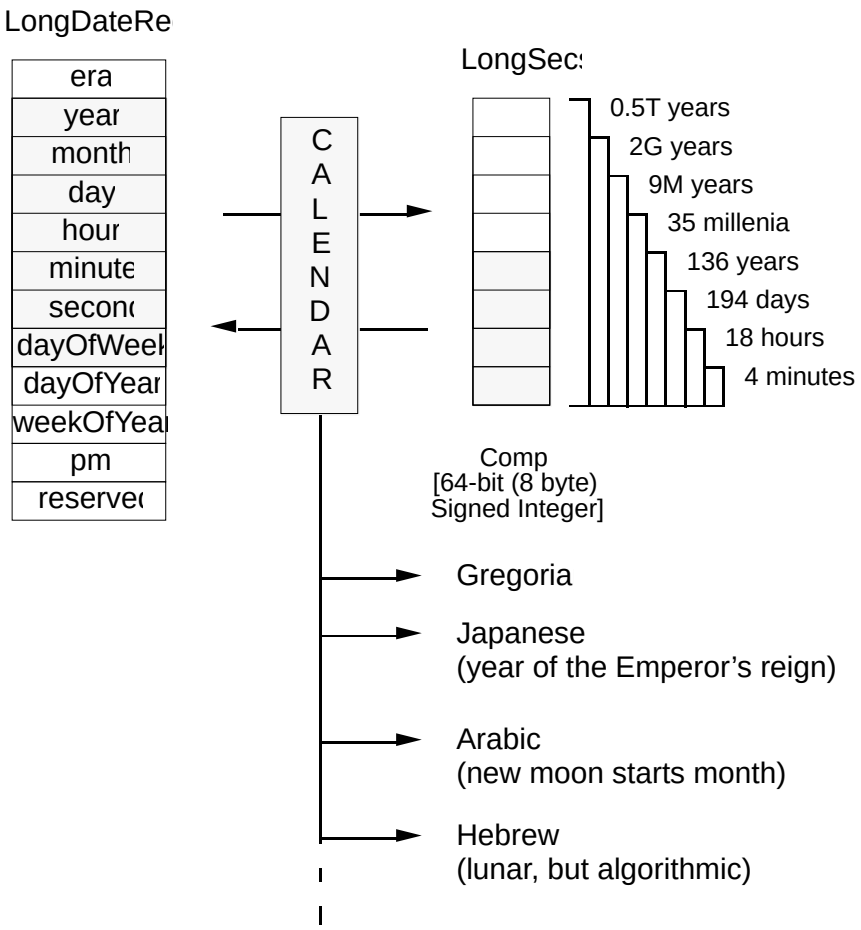
```
Type LongDateTime = Comp;
```

The standard date conversion record is extended using a new structure:

```
LongDateRec = Record
               case Integer of
               0:      (      era,year,month,day,hour,minute,second,
                              dayOfWeek,dayOfYear,weekOfYear,
                              pm,res1,res2,res3: Integer);
               1:      (      list: array [longDateField] of Integer);
               2:      (      eraAlt:Integer;
                              oldDate:       DateTimeRec);
        end;
```

The default calendar for converting to and from the long internal format is the Gregorian calendar. The era field for this calendar has values 0 for A.D., and -1 for B.C. (Note that the international date string conversion routines do not append strings for A.D. or B.C.) The current range allowed in conversion is roughly 30,000 BC to 30,000 AD.

(Note that in different countries the change from the Julian calendar to Gregorian calendar occurred in different years: in Catholic countries, it occurred in 1582, while in Russia it took place as late as 1917. Dates before these years in those countries should use the Julian calendar for conversion. The Julian calendar differs from the Gregorian by 3 days every 4 centuries.)



Long Date <——> String

a. InitDateCache

```
Function InitDateCache (theCache: DateCachePtr): OSErr;
```

This routine **must** be called before using the String2Date or String2Time routines (see below) to format the theCache record. Allocation of this record is the responsibility of the caller: it can either be a local variable, a Ptr or a locked Handle. By using this cache, the performance of the String2Date and String2Time routines are improved.

```
Procedure MyRoutine;
Var
        myCache: DateCacheRecord;
Begin
        InitDateCache (@myCache);
        {call the String2Date or Time routines}
End;
```

b. String2Date, String2Time

```
Function String2Date(   textPtr:              Ptr;
                        textLen:      longint;
                        theCache:     DateCachePtr;
                Var     lengthUsed:   Longint;
                Var     dateTime:     LongDateRec)
        : String2DateStatus;
```

```
Function String2Time(   textPtr:              Ptr;
                        textLen:      longint;
                        theCache:     DateCachePtr;
                Var     lengthUsed:   Longint;
                Var     dateTime:     LongDateRec)
         : String2DateStatus;
```

These routines expect a date/time at the beginning of the text. They parse the text, setting the lengthUsed to reflect the remainder of the text, and fill the dateTime record. They recognize all the strings that are produced by the international date and time utilities, and others. For example, they will recognize the following dates: September 1, 1987, 1 Sept 1987, 1/9/1987, and 1 1987 sEpT.

If the value of the input year is less than 100, then it is added to 1900; if less than 1000, then it is added to 1000 (the appropriate values are used from other calendars, gotten from the base date: LongDateTime = 0). Thus the dates 1/9/1987 and 1/9/87 are equivalent.

The routines use roughly the following grammar to interpret the date and time. The relevant fields of the international utilities resources are used for separators, month and week-day names, and the ordering of the date elements. The parsing is actually semantic-driven, so finer distinctions are made than those represented in the syntax diagram.

```
time      := number [tSep number [tSep number]] [mornStr | eveStr | timeSuff]
tSep      := timeSep | sep
date      := [dSep] dField [dSep dField [dSep dField [dSep dField [dSep]]]]
dField    := number | dayOfWeek | abbrevMonth | month
dSep      := dateSep | st0 | st1 | st2 | st3 | st4 | sep
sep       := <non-alphanumeric>
```

The date defaults are the current day, month and year. The time defaults to 00:00:00. The digits in a year are padded out on the left, using the base date (the date corresponding to zero seconds: Jan 1, 1904). This routine uses the tokenizer to separate the components of the strings. It depends upon the names of the months and weekdays used from international resources being single alphanumeric tokens.

Note that the date routine only fills in the year, month, day and dayOfWeek; the time routine fills in only the hour, minute and second. Thus the two routines can be called sequentially to fill complementary values in the LongDateRec.

The return from the routine is a set of bits that indicate confidence levels, with higher numbers indicating low confidence in how closely the input string matched what the routine expected. For example, inputing a time of 12.43.36 will work, but return a message indicating that the separator was not standard. This can also be used to parse a string containing both the date and time, by using the confidence levels to determine which portion comes first. The returned bits include:

```
        longDateFound         = 1;
        leftOverChars         = 2;
        sepNotIntlSep         = 4;
        fieldOrderNotIntl     = 8;
        extraneousStrings     = 16;
        tooManySeps           = 32;
        sepNotConsistnt               = 64;
        tokenErr                      = $8100;
        cantReadUtilities     = $8200;
        dateTimeNotFound              = $8400;
        dateTimeInvalid               = $8800;
```

c. LongDate Conversion

```
Procedure LongDate2Secs(lDate: LongDateRec;
                Var     lSecs: LongDateTime);

Procedure LongSecs2Date(lSecs: LongDateTime;
                Var     lDate: LongDateRec);
```

These extend the range of the Macintosh calendar as discussed above. Any fields that are not used should be zeroed. On input, the LongDate2Secs routine will use the day and month unless the day is zero; otherwise the dayOfYear is used unless it is zero; otherwise the dayOfWeek and weekOfYear are used.

Other fields are additive: if you supply a month of 37, that will be interpreted as adding 3 to the year, and using a month of 1. This latter property is subject to some restrictions imposed by the internal arithmetic: for example, | hour*60+minute | must be less than 32767.
Two new interfaces have been added to Pack6 for LongDate support:

```
IULDateString(          dateTime:      LongDateTime;
                        form:          DateForm;
                Var     Result:        Str255;
                        intlParam:     Handle);
Assembly selector:      20

IULTimeString(          dateTime:      LongDateTime;
                        wantSeconds:   BOOLEAN;
                Var     Result:        Str255;
                        intlParam:     Handle);
Assembly selector:      22
```

These routines take a LongDateTime, and return a formatted string. Only the old fields year..second,dayOfWeek are used. If the intlParam is zero, then the international resource 0 (itl0) is used. The output year is limited to four digits: e.g. from 1 to 9999 A.D.

d. ToggleDate, ValidDate

```
Function ToggleDate (Var mySecs:        LongDateTime
                         field:         LongDateField;
                         delta:         DateDelta;
                         ch:            Integer;
                         params:        TogglePB)
         :ToggleResults;

Function ValidDate (Var  date :         LongDateRec;
                         flags:         Longint;
                   Var   newSecs:               LongDateTime)
         : Integer;
```

The ToggleDate routine is used to modify a date/time record (see diagram below) by toggling one of the fields, up or down. The routine returns a valid date by performing two types of action. If the affected field over- or underflows, then it will wrap to the corresponding low or high value. If changing the affected field causes other fields to be invalid, then a close date is selected (which may cause other fields to change). For example, toggling the year upwards in Feb 29, 1980 results in Mar 1, 1981. Currently only the fields year..second, am can be toggled, although this should change in the future.

The routine will also toggle by character, if the delta = 0. The character will be used to change the field in the following way. If it is a digit, then it will be added to the end of the field, and the field will be then modified to be valid in a similar manner as in the alarm clock. For example, if the minute is ‘54’, then to replace it by ‘23’ by entering characters, first the minute will change to ‘42’, then to ‘23’. The AM/PM field will also use letters.

```
TogglePB = Record
                togFlags:       Longint;        {default $7E}
                amChars:        ResType;        {usually from intl0}
                pmChars:        ResType;        {ditto}
                reserved:       array [0..3] of Longint;
                                                {reserved: must be zeroed!}
    End;
```

The parameter block should be set up as follows. It should contain the uppercase versions of the AM and PM strings to match against (the defaults mornStr and eveStr can be copied from the international utilities using IUGetIntl, and uppercased with UprText).

The ToggleDate routine makes an internal call to ValidDate, which can also be called directly by the user. ValidDate checks the date record for correctness, using the TogglePB.flags value passed to it by ToggleDate. If any of the record fields are invalid, ValidDate returns a DateField value corresponding to the field in error. Otherwise, it returns a -1.

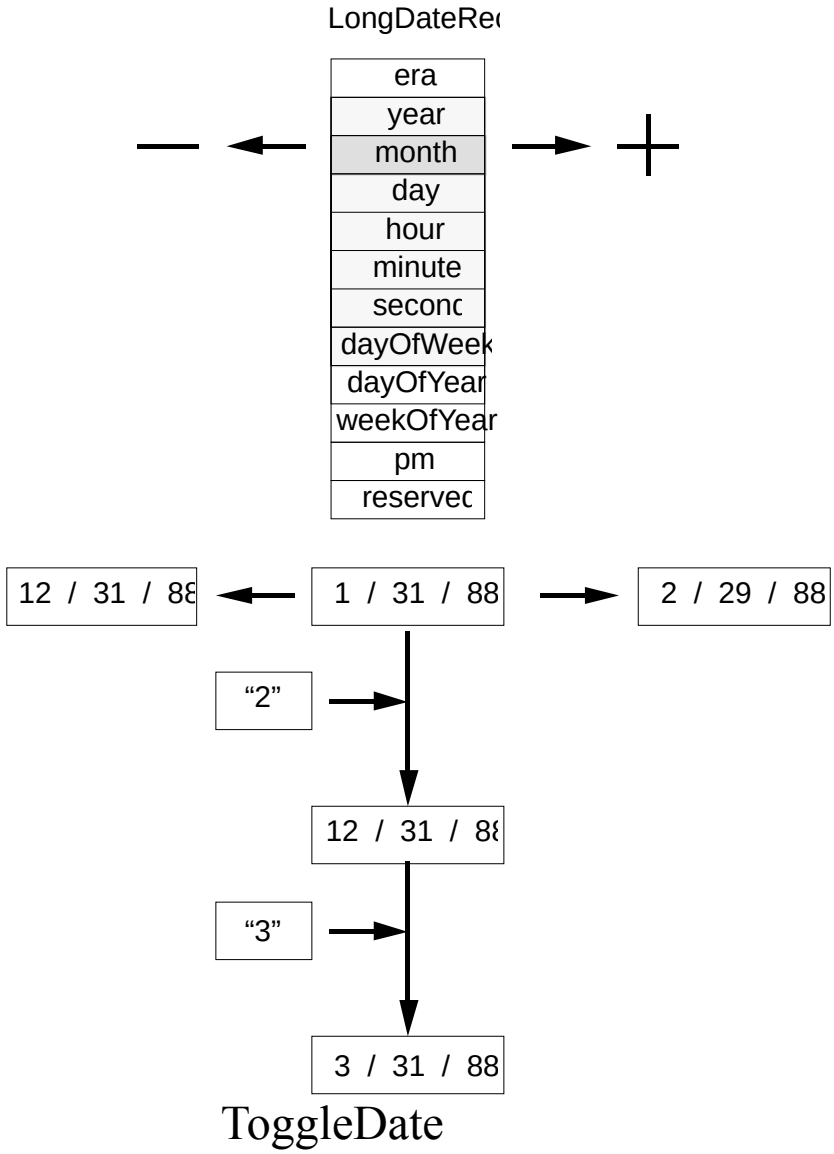
The TogglePB.flags value passed to ValidDate by ToggleDate are the same for ToggleDate and ValidDate. The low word bits correspond to the values in the enumerated type DateField. For example, to check the validity of the year field you can create a mask by doing the following:

```
yearFieldMask = 2**yearField;
```

The high word of the flags value can be used to set various other conditions. The only one currently used is a flag which can be set to restrict the range of valid dates to the short date format (smallDateBit = 31; smallDateMask = $80000000). All other bits are reserved, and should be set to zero. The reserved values should also be zeroed.

The ToggleDate routine should generally have the bits $007E turned on, which can be done by using the constants:

```
dateStdMask =          eraMask+yearMask+monthMask+dayMask+hourMask+minuteMask+secondMask;
```



e. Reading/Writing the Location

```
Procedure ReadLocation (Var loc : Location);
Procedure WriteLocation (loc : Location);
```

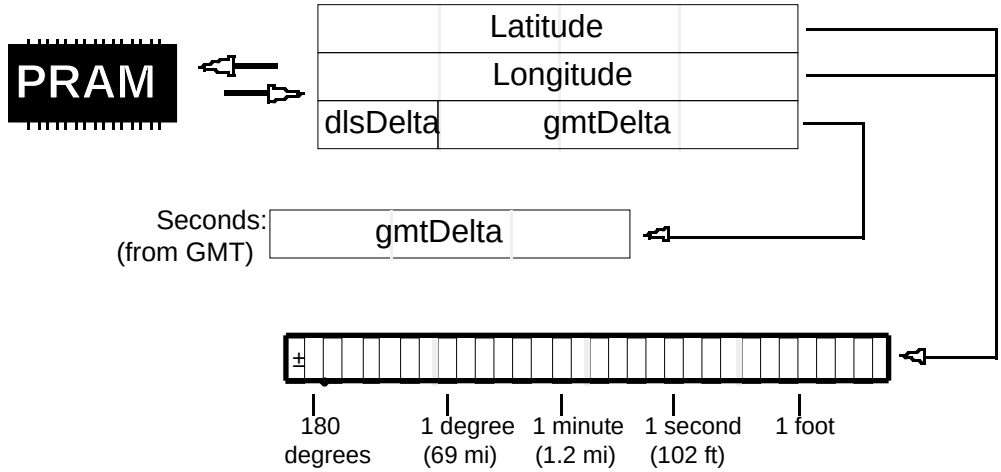
These routines allow the programmer to access the stored geographic location of the Macintosh from parameter RAM, and time zone information. For example, the time zone information can be used to derive the absolute time (GMT) that a document or mail message was created. With this information, when the document is received across time zones, the creation date and time are correct. Otherwise, documents can appear to be created “after” they are read (e.g., I can create a message in Tokyo on Tuesday, and send it to Cupertino, where it is received and read on Monday). Geographic information can also be used by applications that require it.

If the Location has never been set, then it should be <0,0,0>. The top byte of the gmtDelta should be masked off, and preserved when writing: it is reserved for future extension. The gmtDelta is in seconds east of GMT: e.g. San Francisco is at minus 28,800 seconds (8 hours * 3600 seconds/hour). The latitude and longitude are in fractions of a great circle, giving them accuracy to within less than a foot, which should be sufficient for most purposes. For example, fract values of 1.0 = 90°, -1.0 = -90°, -2.0 = -180°.

```
Type
        MachineLocation = RECORD
                        latitude, longitude : Fract;
                        case Integer of
                0:      (       dlsDelta: SignedByte);
                        1:      (       gmtDelta: Longint);
                END;
```

The gmtDelta is really a 3-byte value (see the diagram), so the user must take care to get and set it properly as in the following code examples:

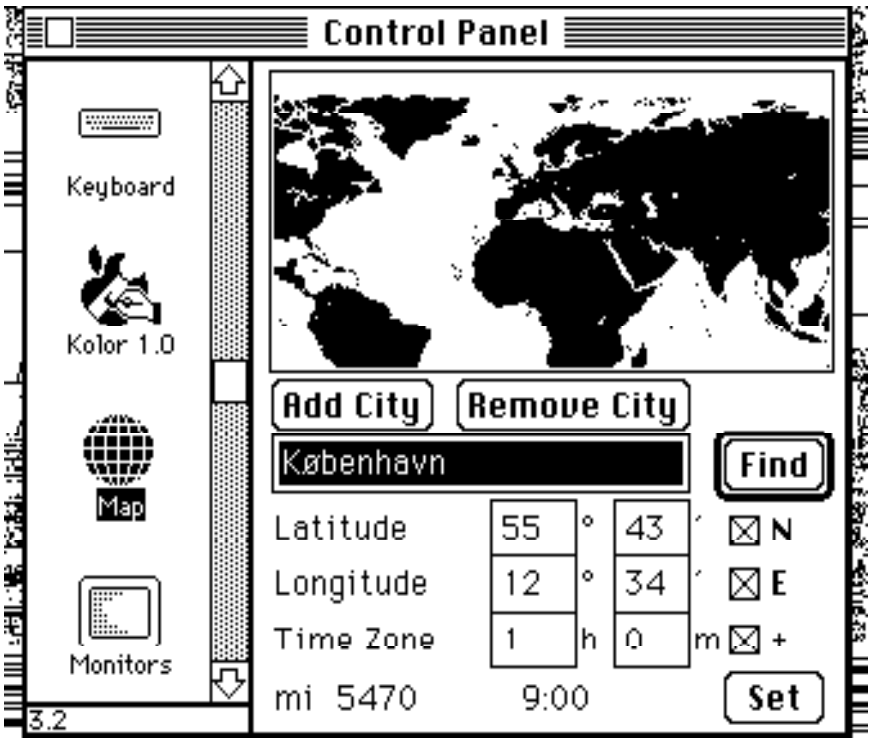
```
Function GetGmtDelta(Var myLocation: MachineLocation): Longint;
Var
        internalGmtDelta: Longint;
begin
        With myLocation Do Begin
                internalGmtDelta := BAnd(gmtDelta,$00FFFFFF);       {get value}
                If BTst(internalGmtDelta,23)                {sign extend}
                        Then internalGmtDelta := BOr(internalGmtDelta,$FF000000);
        GetGmtDelta := internalGmtDelta;
        End;
End;


Procedure SetGmtDelta(Var myLocation: MachineLocation; myGmtDelta: Longint);
Type
        LocationFlags = packed array [0..7] of Boolean;
Var
        tempSignedByte: LocationFlags;
Begin
        With myLocation Do Begin
                tempSignedByte := dlsDelta;          {save flags}
                gmtDelta := myGmtDelta;                     {set value}
                dlsDelta := tempSignedByte;          {restore flags}
        End;
End;
```



Locations

f. Setting Latitude/Longitude/Time Zone CDEV

This new control panel module on the utilities disk allows the user to set the latitude, longitude, and time zone. The values are stored in parameter RAM on the host machine. (See the Map CDEV documentation for more details).


Control Panel
Keyboard
Kolor 1.0
Map
Monitors
3.2
Add City
Remove City
København
Find
Latitude
55
°
43
′
N
Longitude
12
°
34
′
E
Time Zone
1
h
0
m
+
mi 5470
9:00
Set

Map

6. Numbers

The new number routines supplement SANE, allowing applications to display formatted numbers in the manner of Microsoft Excel or Fourth Dimension, and to *read* both formatted and simple numbers. The formatting strings allow natural display and entry of numbers and editing of format strings even though the original numbers *and* the format strings were entered in a language other than the final user's.

Number parsing is based on a NumberParts table that describes the essentials of numeric display for a particular language, including such components as thousands separator, decimal point, scientific notation, forced zeroes in the absence of significant digits, etc. A default NumberParts table for each locale's system resides in the itl4 resource for that system.

```
NumberParts = RECORD
          version:                      integer;
          data:                         array [tokLeftQuote..tokMaxSymbols] OF WideChar;
          pePlus, peMinus, peMinusPlus:         WideCharArr;
          altNumTable:                  WideCharArr;
          reserved:                     packed array [0..19] of Char;(must be zeroed!}
END;
```

Here is an example of how to access the itl4 default NumberParts table:

```
Function DefaultParts( Var x: NumberParts ): Boolean;
Var
          itl4: Itl4Handle;
Begin
          DefaultParts := false;                                {assume error}
          itl4 := itl4Handle(IUGetIntl(4));                     {get itl4 record}
          if itl4 <> nil then begin                             {if ok}
                  x := NumberPartsPtr(ord(itl4^)+itl4^^.defPartsOffset)^;    {number parts
                                                                             subtable}
                  DefaultParts := true;                         {no error}
          end;
End;
```

The user provides a format descriptor string very similar to Fourth Dimension's. This format string is translated by Str2Format into a canonical format which is transportable between different languages such as French, English, and Japanese.

The format descriptor string may be broken into as many as three parts: positive, negative, and zero. For example, the number 3456.713 used with the canonical format produced from "#,###.#;(#,###.#)" will produce the string representation "3,456.7" in the United States. In Switzerland the same canonical format would be displayed as "#.###,#;(#.###,#)", and the number displayed with this format would be "3.456,7".

The number formats include the following features (the defaults for the US are listed following):

Separators:

decimal separator (.), thousands separator (,)

Example:	format string: ###,###.0##,###
1	—> 1.0
1234	—> 1,234.0
3.141592	—> 3.141,592

Digits:

zero digit (0), skipping digit (#), padding digit (^), padding value (NBSP) Example:
format string: ###;(000);^^^

1	—>	1
-1	—>	(001)
0	—>	0

The number format routines always fill in digits from the right or from the left of the decimal point.

Example:	format string: ###'foo'###	
123foo456	—>	123foo456
22foo44	—>	2foo244
123foo	—>	123
Example:	format string: 0.###'foo'###	
0.foo123	—>	0.123
0.1foo456	—>	0.145foo6
0.1456	—>	0.145foo6

Formats using zero and skipping digit characters do not allow extension beyond the minimum number of digits specified to the right or left of the decimal place. For example: users must provide the desired maximum digits on the left: e.g. #,###,### instead of #,###. X2FormStr will return a result of formatOverflow when the number contains more digits to the left of the decimal point than specified in the format string. Input values with more digits to the right of the decimal point than there are digits allowed in the format string will be rounded on output.

Example:	format string: ###.###	
1234.56789	—>	formatOverflow on output
1.234999	—>	1.235

Control:

left quote (‘), right quote (’), escape quote (\), sign separator (;)

Example:	format string: ###'CR';###'DB';'\'zero\''	
1	—>	1CR
-1	—>	1DB
0	—>	'zero'

Marks:

plus (+), minus (-), percent (%)
positive exponent (E+), negative exponent (E-), mixed exponent (E)

Example:	format string: ##%	
0.1	—>	10%

There is a limitation creating format strings with exponential notation: the user must always place zero leaders immediately after the exponent marks and skipping digits before, when more than one digit must be represented between the exponent and the decimal point.

Example:	format string: ##.####E+0	
1.23E+3	—>	1.23E+3

The sign of exponents must be made explicit in the format string by using *ePlus* (E+) or *eMinus* (E-) format. *eMinusPlus* notation (E) is only used in the input number string to specify a positive exponent when the sign of the format string exponent is negative.

format	+ exponent sign	-
ePlus	ePlus(E+)	eMinus(E-)
eMinus	eMinusPlus(E)	eMinus(E-)

Use ePlus notation in the format string to specify negatively or positively signed exponents in the input number string:

```
Example:        ePlus format string: #.#E+#
1.2E-3          —>      1.2E-3
1.2E+3          —>      1.2E+3

Example:        eMinus format string: #.#E-#
1.2E-3          —>      1.2E-3
1.2E3           —>      1.2E3    (i.e., 1200)
```

Literals:

```
unquoted literals ([]$:(){}), literals requiring quotes (ABC...)
Example:        format string: [###' Million '###' Thousand '###]
300             —>      [300]
3000000         —>      [3 Million 000 Thousand 000]
```

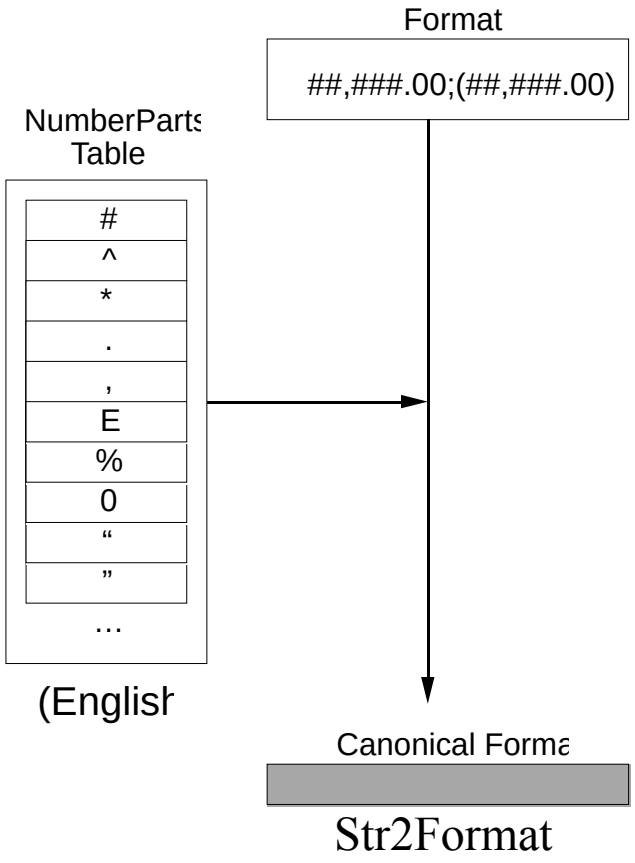
A typical scenario consists of the application reading the default NumberParts table from itl4. The user provides a format definition string, such as the string “#.###,#;(#.###,#)” of the above example, as a template for whatever field he is currently working in. The application submits that string to Str2Format, which returns a canonical format string corresponding to the user’s input. This canonical format, rather than the raw format definition string, is stored in the document. The program can convert the canonical format back to a user-editable string using the Format2Str routine.

When a number is to be displayed, the application passes the number and canonical format to FormatX2Str to produce a formatted number that the application then displays in that field. If the user types a string into the field, then FormatStr2X can be used with the canonical format for the field to read formatted numbers. That is, the user can type in “(3.678,9)”, and have the number interpreted correctly.

a. Converting to Canonical Formats

```
Function Str2Format (    inString :      str255;
                         partsTable:     NumberParts;
                  Var    outString:      NumFormatString )
          :FormatStatus;
```

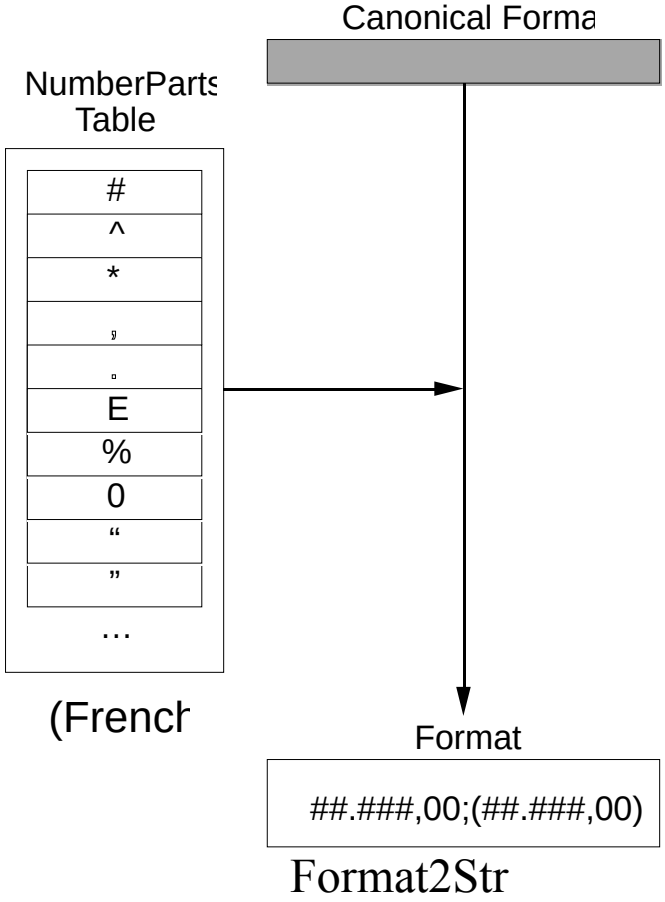
Str2Format converts a string typed by the user into a canonical format. It checks the validity of the format string itself and also that of the NumberParts table, because the NumberParts table is programmable by the application.



b. Displaying the Canonical Format String

```
Function Format2Str (    myCanonical :  NumFormatString;
                         partsTable:    NumberParts;
                 Var     outString:     str255;
                 Var     positions:     TripleInt))
         :FormatStatus;
```

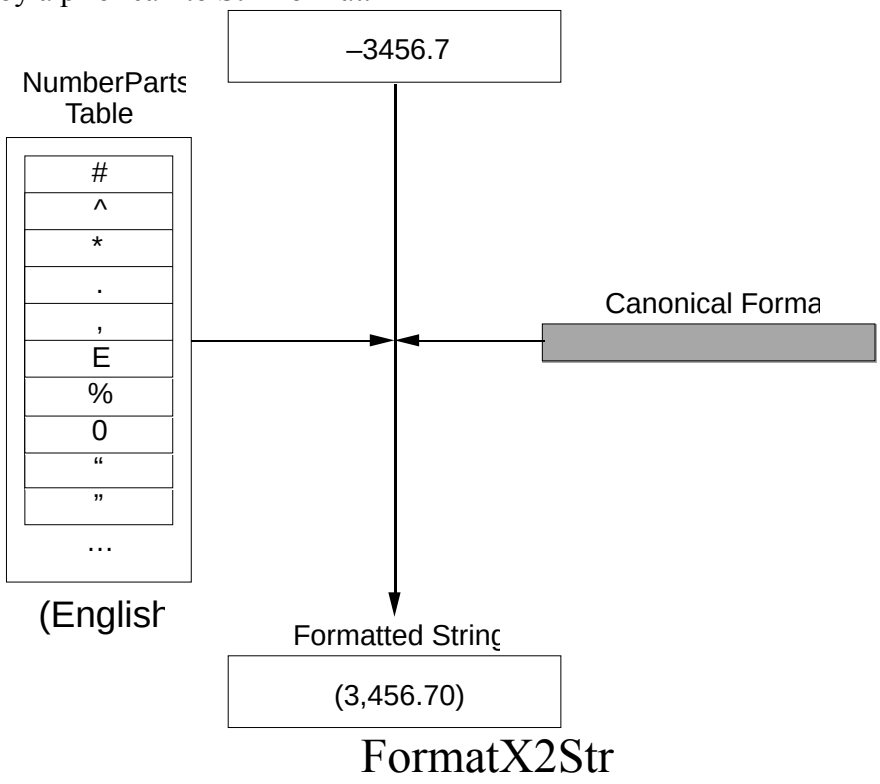
Format2Str creates the string corresponding to a format definition string which has been created by a prior call to Str2Format and according to the NumberParts table. It is the inverse operation of Str2Format. This allows programs to display previously entered formats for users to edit.



c. Formatting Numbers

```
Function FormatX2Str (   x :            Extended;
                         myCanonical:   NumFormatString;
                         partsTable:    NumberParts;
                  Var    outString:     Str255)
          :FormatStatus;
```

This routine creates a textual representation of a number according to a canonical format which has been created by a prior call to Str2Format.



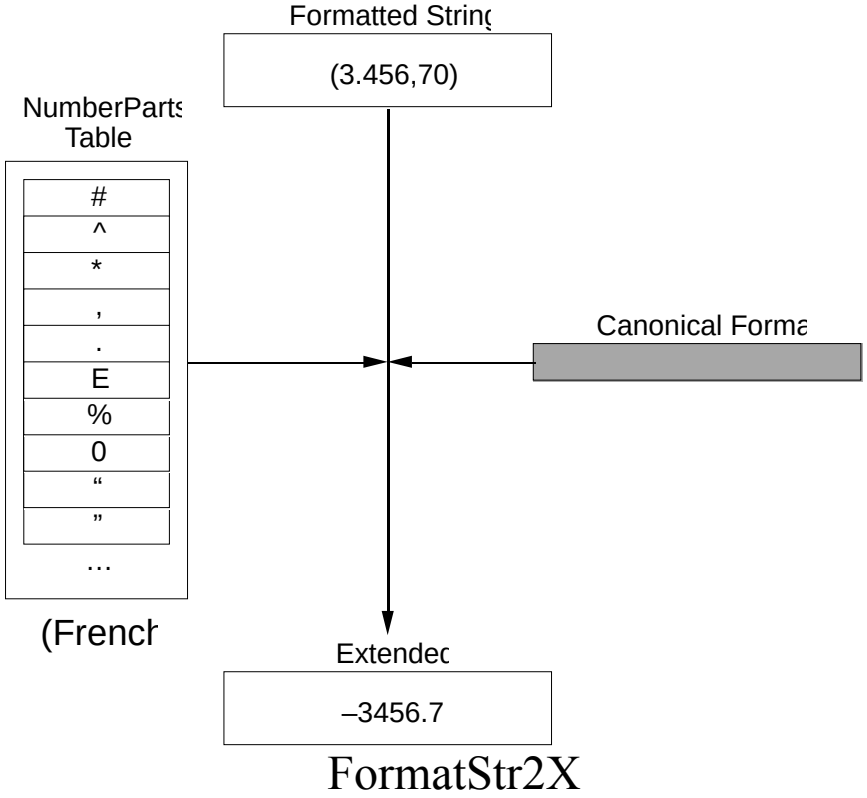
FormatX2Str

d. Reading Formatted Numbers

```
Function FormatStr2X (   source:        str255;
                         myCanonical:   NumFormatString
                  partsTable:    NumberParts
                  Var     x:             Extended)
          :FormatStatus;
```

This routine reads a textual representation of a number according to a canonical format which has been created by a prior call to Str2Format, and creates an extended floating point number which corresponds to that string.

Internally, the routine converts the string into a format acceptable to SANE, matching against the three possible patterns in the canonical format. If the input string does not match any of the patterns, then FormatStr2X parses the string as best it can returning the result. Currently it is converted to a simple form, stripping non-digits and replacing the decimal point, before calling SANE.



7. Summary of Routines

The updated interface files are available for MPW 3.0 on AppleLink on the Developer Services Bulletin Board and with the disk version of this file (available with the Macintosh Technical Notes) from the Apple Programmer’s and Developer’s Association (APDA).