Strings

String Types The following discussion does not religiously follow each languages peculiar use of the terms "string", "character", "array of characters", etc. The term "string" is simply used to denote whatever data type corresponds to the primary one used by a language to manipulate blocks of ASCII characters. The three most common types of strings encountered when working with FaceWare modules correspond to the 3 major languages:

1. C: character array of any length terminated by a null byte
2. Fortran: character array terminated by spaces (space-padded)
3. Pascal: ≤ 255-character array preceeded by a leading length byte

Macintosh compilers often support more than one of the above types of strings. They will also often provide string-conversion functions that convert one type to another. The UtilIt module also supports a CnvStr command that can be used to interconvert any of the above types. With respect to strings in shared records, a module will either force you to use a particular type of string, or will allow you to use your language's native string type and do the conversions itself.

Each string type has its advantages and disadvantages. In most cases, the disadvantages of each type are balanced by built-in functions that do the dirty work for you: finding the length, inserting a substring, etc.

Fortran strings have the advantage of having no "undefined" characters since spaces are used to pad the end of strings. This makes it easy to manipulate substrings within Fortran strings since you never have to worry about where the end of the string is. On the negative side, spaces at the end of a string are never significant, and a search must be made to determine string length.

C strings also suffer from there being no simple way to determine their size without searching for the end of the string (a null byte). Another disadvantage is that the position of the null byte must be managed when changing the size of the string. On the plus side, any non-null character can be used in the string, and C strings are not limited in size.

Pascal strings have a leading length byte which solves the problem of quickly determining the length of a string. The length byte also supports packing Pascal strings together into lists, providing a very efficient way to store strings. On the down side, the length byte (like C's null byte) must be adjusted each time the size of the string is changed, and limits the size of the string to 255 characters.

String-To-Number Interconversion Most applications that work with numbers will become involved with interconverting numbers and strings. Toolbox, SANE, and UtilIt routines are available to all Mac programmers to interconvert numbers and strings: StringToNum and NumToString for integers, str2num and num2str for reals, and the commands NumToS and SToNum. Each language also has built-in support for interconverting numbers and strings:

Fortran Fortran's support for "internal files" makes conversion of numbers to strings almost trivial, and brings all the power of format statements to this operation. For example, suppose an existing program writes two integers to unit 3:

```
    integer alpha,beta
    ...
    alpha = 5
    beta = 10
    write(3,5) alpha,beta
5   format (2I5)
```

The equivalent statements for writing the same variables to the fRec's uName string (referred to as an "internal file") would be,

```
    write(uName,5) alpha,beta
5   format (2I5)
```

or simply,

```
    write(uName,'(2I5)') alpha,beta
```

Reading a Fortran string is very similar to writing a string. The following code reads the

variables alpha and beta from the string uName:

```
    read(uName,5) alpha,beta
5   format (2I5)   !2 int.s occupying 5 characters each
```

One drawback to using a string as an internal file in a read statement is that, with some Fortran compilers, you cannot use list-directed formats ("*" formatting) when reading more than one variable from a single string. There is usually no problem, however, using "*" formatting when writing multiple values to a string.

Pascal The THINK Pascal functions "StringOf" and "ReadString" provide most of the functionality of the equivalent Fortran "write" and "read". Translating from the above Fortran examples,

```
  alpha,beta : longint;
  ...
  alpha := 5;
  beta := 10;
  uName := StringOf(alpha:5,beta:5);
```

One potential problem using the "StringOf" function, however, is that it will use more characters than the designated field width ("5" in the above example) if it finds that the output string is larger than the space given. So be certain to make your field-widths large enough.

Reading a string with "ReadString" is a lot like a list-directed read with Fortran. Unfortunately, you cannot always guarantee that the values in a string will be properly delimited (there may be no space between one item and the next, or a single item may be broken up by spaces). For this reason, the safest way to use ReadString when reading multiple values from a single string is to call it for each substring that must be evaluated. For example, to read the variables alpha and beta from the string uName used in previous examples,

```
  ReadString(copy(uName,1,5),alpha);
  ReadString(copy(uName,6,5),beta);
```

If your particular implementation of Pascal doesn't support functions equivalent to "StringOf" and "ReadString" (or if you are developing a code resource in which StringOf and ReadString are not supported), then use the UtilIt, toolbox, or SANE routines mentioned above.

C C provides an extensive set of string-handling functions. From the perspective of other languages, many of these functions are necessary because C does not provide more direct ways of accomplishing the same task. For example, even the simplest of operations such as comparing and assigning strings are function based:

```
  strcpy(uName,"EditIt.Rsrc");    /* assign string */
  if (strcmp(uName,"Red") == 0)   /* compare strings */
```

On the bright side, the C functions "sprintf" and "sscanf" provide nearly all of the functionality of the equivalent Fortran "write" and "read" to internal files. Translating from the above Fortran examples,

```
long   alpha,beta;
  ...

alpha = 5;

beta = 10;

sprintf(uName,"%5ld%5ld",alpha,beta); /* "l" for long */
```

will write alpha and beta to uName using 5 characters for each number.

The use of "sscanf" looks much like the equivalent Fortran formatted "read". In many cases, however, sscanf does not prove to be a reliable way of reading multiple values from a single string. One problem is that the format which you enter as part of the sscanf call is

not strictly followed if the function encounters spaces at the beginning of a field. In this case the spaces are skipped over until a non-space character is found before reading the next variable with the next format, thereby putting the rest of the scan "out of synch". This function also has problems if the contents of a field contain spaces between non-space characters.

The bottom line is that sscanf should not be used to read multiple values from a string in one statement unless you are certain that the non-space characters in the substrings being read are left-justified in each field. In all other cases you should read each substring with a separate statement (as with Pascal). For example, to read the variables alpha and beta from the string uName, use

```
sscanf(uName,"%5ld",&alpha);
sscanf(&uName[5],"%5ld",&beta);
```

where "&uName[5]" means that the second scan should begin at the location in memory which corresponds to the sixth character position of uName (0-based array indexing).