Addendum

to the

Implementation Guide

Chapter 1. Contents

This addendum contains all desriptions of REALs and associated library modules, as well as some other new library modules. The page headers of this document reflect the chapters of the original manual to which the documents are associated.

The following parts are included:

Chapter 2. The Standard Library

Section 1. RealInOut

1. General Description

The purpose of this module is to allow REAL number input and output. It is one of the standard modules defined in **Programming in Modula-2**. The WriteRealOct procedure, which is hardly used by application programs, has been removed from this module. More versatile REAL output formatting can be found in the **ConvertReal** module. RealInOut internally relies upon this module.

2. REAL Input

REAL numbers are accepted nearly free-form. The EBNF notation of the accepted input looks like:

```
ReadableReal  = Sign Mantissa [('E'|'e') Sign Number].
Sign          = ['+'|'-'].
Mantissa      = Number ['.' Number]| '.' Number.
Number        = Digit {Digit}.
Digit         = '0'|..|'9'..
```

Examples of correct REAL numbers:

```
1.0    0    0.0000000007   1E38   1.978e-9
+15.793e-5       20000.9
```

The following conditions have to be fulfilled by your input:

- The maximum REAL value is about 1.7e38 (less than 2^{127}), the smallest non-zero, positive REAL is about 3e-39 (2^{-128}).

- The chosen format allows for about 7 digits of precision because it uses 24 binary mantissa digits internally. Numbers with more significant digits will be truncated; the digits that aren't representable are discarded (if they are behind the '.') or used to get the exponent.

- The maximum exponent of a number is 38. The minimum value that leads to non-zero numbers is -39.

3. REAL Output

The output formatting is rather limited, because it uses always the scientific format. The scientific format contains up to 7 mantissa digits. The minimum length of the output is 6 characters ('0.E+00'). Setting the format-parameter 'n' to more than 12 results in leading blanks in the output.

4. The Interface

```
DEFINITION MODULE RealInOut;

  EXPORT QUALIFIED
     ReadReal, WriteReal;

  PROCEDURE ReadReal( VAR r: REAL );
  PROCEDURE WriteReal( r: REAL; width: CARDINAL );

END RealInOut.
```

Section 2. MathLib

1. General Description

MathLib was originally postulated by Prof. Wirth in **Programming in Modula-2** (page 85) as **MathLib0.** In an attempt to standardize a larger library, MODUS (Modula-2 Users Society) has created a draft standard that adds the **'power'** function to the original MathLib0 and names it simply MathLib.

Please keep in mind that REALs have a very finite precision. The calculations allow for a theoretic maximum precision of about 7.2 decimal digits (24 binary mantissa bits --> $\log10(2^{24})$ = 7.2...). For numbers near the maximum representable REAL number, the difference between two adjacent representable numbers is $2^{(127-24)}$ = 2^{103} = 10^{31}. So, there is no use to print results of any calculations with more than 7 significant digits -- you just generate a "better" precision than the computer does.

The algorithms used were chosen because their **average** performance is sufficient and their execution times compare favorable to those of most competitors. None of the calculations are iterative. If series are evaluated, only a fixed number of components gets calculated. These components are spelled out instead of iterated.

If you are mathematically inclined or in need of better precision, D.E. Knuth offers a wealth of numerical algorithms in volume 2 of **The Art of Computer Programming, Seminumerical Algorithms.**

2. Error Handling

Errors during the calculation of functions are fatal. The program is halted with an appropriate message to the terminal using the 'Terminal' module. An exception hereto are 'sin' and 'cos', which may cause an overflow if too big arguments are passed to them.

3. Trigonometric Function Procedures

MathLib contains a set of trigonometric function procedures, namely the sine (sin), the cosine (cos) and the arctangent (arctan). Out of this set, every other trigonometric function can be calculated. In fact, even the cosine isn't necessary to do so.

WARNING - Do not use arguments greater than 32767.0 for sine and cosine. Due to the algorithm used for these procedures, an overflow could occur otherwise.

4. Exponential and Power Function Procedures

The set of exponential and power functions includes the natural logarithm (ln) to the base e = 2.7182818. Note that the precision of the calculations is limited as is the range. The largest argument of 'exp' leading to a successful calculation is about 88 (less than $\ln(2^{127})$). The natural logarithm works on positive numbers greater than 0 only.

The 'power' function allows to raise a REAL number 'x' to the REAL power 'y'.

The square root 'sqrt' works on positive arguments only.

WARNING - 'exp' arguments may be up to 88, 'power' allows for x = ln(MAX(REAL)) / ln(y) , maximum, 'ln' arguments must be greater than 0 and 'sqrt' doesn't accept negative argument values.

5. Conversion Function Procedures

The 'entier' and 'real' functions provide conversions between INTEGERs and REALs in both ways.

WARNING - 'entier' and 'real' work on the INTEGER range only. 'entier' returns the maximum or minimum INTEGER values for arguments that are out of the integer bounds.

6. The Interface

```
DEFINITION MODULE MathLib;

  EXPORT QUALIFIED
    sqrt, exp, ln, power,
    sin, cos, arctan,
    real, entier;


  PROCEDURE sqrt(x : REAL): REAL;
  PROCEDURE exp(x : REAL): REAL;
  PROCEDURE ln(x : REAL): REAL;
  PROCEDURE power(x, y : REAL): REAL;
  PROCEDURE sin(x : REAL): REAL;
  PROCEDURE cos(x : REAL): REAL;
  PROCEDURE arctan(x : REAL): REAL;
  PROCEDURE real(i: INTEGER): REAL;
  PROCEDURE entier(x : REAL): INTEGER;

END MathLib.
```

Section 3. ConvertReal

1. General Description

ConvertReal includes procedures that do conversions between STRING and REAL data. This module is part of a standard library proposed by MODUS.

A real number consists of two major parts: The **mantissa** and the **exponent.** Both can have a sign. The mantissa may be any floating point number, i.e. 1.8, -1567.04, 0.000002, etc. The exponent may be an integer number in the range -39 up to and including 38. Note that numbers that are too small to be different from zero, are automatically set to zero. No error is indicated in this case. On the other hand, if the upper limit of 2^{127} = 1.7..* 10^{38} is reached or surpassed in a calculation, a fatal runtime error occurs.

2. StrToReal

The input procedure StrToReal converts an ASCII string into the internal REAL format. A legal REAL number has to comply to the syntax:

```
AcceptedReal  = Sign Mantissa [('E'|'e') Sign Number].
Sign          = ['+'|'-'].
Mantissa      = Number ['.' Number] | '.' Number.
Number        = Digit {Digit}.
Digit         = '0'|..|'9'.
```

The maximum range of the REAL number format is +/- (5e-39..1.7014118e38). The exponent -the number after 'E' or 'e'- is to the decimal base. Thus, the REAL number is composed as mantissa * $10^{exponent}$.

NOTE - Differing from the compiler's real number syntax which allows it to determine the type of a constant, the dot may be omitted in input to this procedure.

Errors during a conversion are indicated by setting the BOOLEAN variable parameter 'success' to FALSE. Errors may occur because of numbers too big to be represented, or if illegal characters are in the string holding the number.

3. RealToStr

This procedure converts a REAL 'r' to the string 's' producing 'width' digits or blanks. The 'decPlaces' parameter serves different purposes: If 'decPlaces' is greater than zero, the REAL is converted to a fixed-point representation having 'decPlaces' decimal places (for example '1.50' for decPlaces = 2). If 'decPlaces' is zero, an integer representation of the number without a '.' (100000) is produced. If it is negative, the scientific notation (1.8E+10, etc.) is created. In any case, the string contains leading blanks if the number is not exactly 'width' characters long. If the scientific format has been chosen, at least 1 and at most 8 significant digits are output for the mantissa. The value of 'width' determines the actual number of digits according to the formula

digits = width - ExponentChars (4) - Dot (1) - sign (0 or 1).

'success' is set to FALSE if the conversion couldn't be accomplished due to too small a field width. If 'width' is greater than the size of the string, an error gets flagged, too.

The minimum field widths are:

Integer:	1	
Decimal:	2 + decPlaces	("0." + decPlaces)
Scientific:	6	("0.E+00")

For each representation, negative numbers require one more character.

NOTE - Due to the REAL number format, only about 7 digits are significant in a number.

4. The Interface

```
DEFINITION MODULE ConvertReal;

  EXPORT QUALIFIED
    RealToStr, StrToReal;

  PROCEDURE RealToStr((* converts *)     r          : REAL;
                      (* using    *)     width      : CARDINAL;
                      (* and      *)     decPlaces  : INTEGER;
                      (* into     *) VAR s          : STRING;
                      (* if       *) VAR success    : BOOLEAN);

  PROCEDURE StrToReal((* converts *)     s          : STRING;
                      (* into     *) VAR r          : REAL;
                      (* if       *) VAR success    : BOOLEAN);

END ConvertReal.
```

Section 4. Files

1. General Description

This is the third File System included with this Modula-2 System. It is capable of file positioning at the byte and record-level and therefore provides some previously unavailable features.

This file system uses internal buffers of 1kByte in the current implementation. As with most other modules, you can change this size to customize the system to your needs.

Its advantage over most other random I/O implementations lies in the fact that it is able to position a file at the byte level, ignoring CP/M's 128 byte sectors. The price for this flexibility is an increased complexity and increased size, but it is very simple to operate by the user.

It is also remarkable to say that you can read and write to a file without closing and re-opening it. In fact, you can sequentially read a character, write the next, read the third, etc.

NOTE - Upon each read action, a file's internal buffer gets flushed to disk, if it has been written to since the last read operation.

Errors may be detected by watching the 'EOF' and 'FileStatus'-functions. A function that gives literal (string) messages is also provided.

After the introduction of the variable types used in Files, each major operation on files is explained in "shorthand notation", i.e. without attempting to do sufficient error checking.

2. File Names

File names are compatible with the 'FileNames' Module, i.e. they are internally checked by the 'StringToFCBFile' procedure.

3. File Variables

The 'FILE' type is hidden from the user. To make a file acessible, you have either to open or to create it using the 'Open' and 'Create' procedures. To end a file's processing, you have to 'Close' it. If you want that the file also gets deleted on the disk, use the 'Release' procedure. This is especially useful for locally used files.

A normal CP/M 2.2 file may contain up to 65536 records (2^{16}). This results in 8 MBytes maximum file size. 'Files' is laid out to allow for this maximum number of records. For large CP/M 3.0 files (up to 32 MBytes, and 2^{18} records (262'144)), 'Files' does not work correctly. The record calculation procedure used by it, however, calculates 24 bit quantities.

4. File Position Variables

'FilePos' variables contain all the information necessary to position a file to a given point. A position can either be calculated using 'CalcPos' or it can be retrieved by calling 'GetPos' or 'GetEOF'.

The only way to position a file is given by the 'SetPos' procedure.

Users of Volition System's implementation may notice the absence of the 'SetEOF' procedure: Only CP/M 3.x does allow for file truncation. Since CP/M Plus isn't very widespread yet, such a procedure has not been included in the normal form of this module.

5. Reading From Files

Although 'Files' supports random access, a file is usually accessed sequentially. So, a read session may look as follows:

```
VAR
  f              : FILE;
  fileName, msg: STRING;
  inputState     : FileState;
  ch             : CHAR;
BEGIN
  inputState := Open(f, fileName);

  IF inputState = FileOK THEN

    WHILE NOT EOF(f) DO
      Read(f, ch);
    END;

    inputState := Close(f);
    IF inputState # FileOK THEN
      StatusMsg(inputState, msg);
      WriteString(msg);
      HALT;
    END;

  ELSE
    StatusMsg(inputState, msg);
    WriteString(msg);
  END;
```

NOTE - Even when a file is only read from, a call to 'Close' is mandatory if you want to reclaim the heap space used by the file descriptor.

Multibyte-records or arrays may be read using the 'ReadBytes' procedure. You pass the address of your structure, and its size. 'ReadBytes' returns the number of bytes that where actually read from the file.

6. Writing to Files

A typical write command sequence skeleton for sequential access is:

```
outputState := Create(f, fileName);

IF outputState = FileOK THEN

  REPEAT
    Write(f, ch);
  UNTIL allWritten;

  outputState := Close(f);

  IF outputState # FileOK THEN
    StatusMsg(outputState, msg);
    WriteString(msg);
  END;

ELSE
  StatusMsg(outputState, msg);
  WriteString(msg);
END;
```

Multibyte-records or arrays may be written using the 'WriteBytes' procedure. You pass the address of your structure, and its size. 'WriteBytes' returns the number of bytes that where actually written to the file.

7. Positioning Files

It is assumed that a file contains a given number of **fixed size records.** To access a given record, you can calculate its position by using 'CalcPos'. 'CalcPos' uses the record number and the record size which you provide, and calculates thereof the CP/M logical record number and the offset of the start of your logical record, in this record. Having calculated this position, you position the file by using 'SetPos'. So, the scheme looks like:

```
CalcPos(recordNumber, SIZE(record), pos);
SetPos(f, pos);
```

If you want to append data to the end of a file, use the 'GetEOF' procedure instead of 'CalcPos'.

NOTE - 'GetEOF' cannot account for your logical records. If your file doesn't end at a CP/M sector end, you have to know where the exact end of file is, either by a mark or by recalculating a record position.

NOTE - Setting the position of a file causes no immediate disk access. This access is delayed until the next read or write operation occurs. This means that positioning a file multiple times between read or write operations doesn't cause your computer to "fiddle around" on the disks.

8. Renaming Files

The renaming operation works straight forward. You give the old and the new file name as strings, and the rename operation is carried out. Success of operation can be determined by watching Rename's return value.

```
IF Rename('XX.OLD', 'YY.NEW') # FileOK THEN
  WriteString('Error in Renaming XX.OLD');
END;
```

9. Deleting Files

You can delete files by using unambigous or ambigous file names. As renaming, deletion is done by name, directly.

```
IF Delete('*.BAS') = FileOK THEN
  WriteString('All line numbers destroyed');
END;
```

10. The Interface

```
DEFINITION MODULE Files;

  FROM Strings IMPORT
    STRING;

  FROM SYSTEM IMPORT
    ADDRESS;


  EXPORT QUALIFIED
    FILE, EOF, FileState, FileStatus, StatusMsg,
    Open, Create, Close, Release,
    Rename, Delete,
    FilePos, SetPos, GetPos, GetEOF, CalcPos,
    Read, Write, ReadBytes, WriteBytes;


  TYPE
    FILE;
    FilePos;

    FileState = (   FileOK,
                    UseError,
                    StatusError,
                    DeviceError,
                    EndError);
```

```
  PROCEDURE EOF(f: FILE): BOOLEAN;
  PROCEDURE FileStatus(f: FILE): FileState;
  PROCEDURE StatusMsg(status: FileState; VAR msg: STRING);

  PROCEDURE Open(VAR f: FILE; name: STRING): FileState;
  PROCEDURE Create(VAR f: FILE; name: STRING): FileState;

  PROCEDURE Close(VAR f: FILE): FileState;
  PROCEDURE Release(VAR f: FILE): FileState;

  PROCEDURE Delete(name: STRING): FileState;
  PROCEDURE Rename(currentName, newName: STRING): FileState;

  PROCEDURE GetPos(f: FILE; VAR pos: FilePos);
  PROCEDURE GetEOF(f: FILE; VAR pos: FilePos);

  PROCEDURE SetPos(f: FILE; pos: FilePos);

  PROCEDURE CalcPos(recNum, recSize: CARDINAL; VAR pos: FilePos);

  PROCEDURE Read(f: FILE; VAR ch: CHAR);
  PROCEDURE ReadBytes(f:FILE; buf: ADDRESS; nBytes: CARDINAL): CARDINAL;

  PROCEDURE Write(f: FILE; ch: CHAR);
  PROCEDURE WriteBytes(f:FILE; buf: ADDRESS; nBytes: CARDINAL): CARDINAL;

END Files.
```

Section 5. CmdLin

1. General Description

The CmdLin module provides a procedure that reads the CP/M command line into a string.

If the commmand line isn't empty, it is checked for correctness. If a space is entered before the program's name, CP/M puts the program name into the command line, too. The default file control blocks, however, are set up correctly. Using this fact, CmdLin finds the correct beginning of the command line. Simply, if you enter (your input underlined)

A> copy alpha beta <CR>

or

A>copy alpha beta <CR>,

CmdLin will return ' alpha beta' as command line.

If the CP/M command line (stored at 80H .. 0FFH) is empty, the ReadCommandLine procedure displays a star prompt and awaits the user's input.

ReadCommandLine can be called multiple times. It returns the CP/M command line but on the first call, if the command line isn't empty at that time. Afterwards, the star prompt is displayed and user input awaited. This input is read with the aid of Terminal.ReadLn, so all the editing features possible there are also present in CmdLin, including program abortion by ^C.

The command line returned by ReadCommandLine is guaranteed to be non-empty.

2. The Interface

```
DEFINITION MODULE CmdLin;

  FROM Strings IMPORT
    STRING;

  EXPORT QUALIFIED
    ReadCommandLine;

  PROCEDURE ReadCommandLine(VAR commandLine: STRING);

END CmdLin.
```