UnoLib documentation

float32.pas

version 11/10/2025

The unit float32.pas contains types and routines emulating operations on floating-point numbers of single precision (soft-float), not supported directly by the Free Pascal Compiler for AVR. It can be used for potentially all types of AVR microcontrollers supported by the compiler. Please note that using soft-float is very resource-consuming and some AVRs may not be able to fit the compiled code into their flash memory.

TRawFloat32

TRawFloat32 is base type for floating point numbers of single precision. Internally it corresponds to type *single* and is mapped to *UInt32*. *TRawFloat32* is optimized for smaller compiled code size than *TFloat32* which is wrapper for *TRawFloat32*. To assign value for *TRawFloat32* using of binary representation of type *single* is recommended (e.g. $BD030000).

TRawFloat32 routines
`function` **`Float32Add`**`(const f1, f2: TRawFloat32): TRawFloat32;` Returns the sum of *f1* and *f2*.
`function` **`Float32Neg`**`(const f1: TRawFloat32): TRawFloat32;` Returns negative value of *f1*. It corresponds to use of operator -.
`function` **`Float32Sub`**`(const f1, f2: TRawFloat32): TRawFloat32;` Returns result of subtraction of *f2* from *f1*.
`function` **`Float32Mul`**`(const f1, f2: TRawFloat32): TRawFloat32;` Returns result of multiplication of *f1* by *f2*.
`function` **`Float32Comp`**`(const f1, f2: TRawFloat32): Int16;` Returns result of comparison of *f1* and *f2*. If they are equals the result is 0, if *f1*<*f2* the result is -1, if *f1*>*f2* then result is 1.
`function` **`Float32Div`**`(const f1, f2: TRawFloat32): TRawFloat32;` Returns result of division of *f1* by *f2*.
`function` **`Float32Mod`**`(const f1, f2: TRawFloat32): TRawFloat32;` Returns the remainder of division (modulo) of *f1* by *f2*.
`function` **`Float32ToInt`**`(const f: TRawFloat32): Int32;` Returns integer part of *f1* removing its fractional part.

```
function IntToFloat32(const value: Int32):TRawFloat32;
```

Converts *value* to floating-point number.

```
function Float32Sqrt(const f1: TRawFloat32): TRawFloat32;
```

Returns the square root of *f1*.

```
function Float32Abs(const f1: TRawFloat32): TRawFloat32;
```

Returns the absolute value (without sign) of *f1*.

```
function Float32Inv(const f1: TRawFloat32): TRawFloat32;
```

Returns the inverse value of *f1*.

Note: correct it so that it returns the result *1/f1*

```
function Float32InvSqrt(const f1: TRawFloat32): TRawFloat32;
```

Returns the inverse square root of *f1*.

```
function Float32Deg2Rad(const f1: TRawFloat32): TRawFloat32;
```

Converts *f1* from degrees to radians.

```
function Float32Rad2Deg(const f1: TRawFloat32): TRawFloat32;
```

Converts *f1* from radians to degrees.

```
function Float32Int(const f1: TRawFloat32): TRawFloat32;
```

Returns the integer part of a *f1*.

```
function Float32Sin(const f1: TRawFloat32): TRawFloat32;
```

Returns the sine of *f1*, where *f1* is an angle in radians.

```
function Float32Cos(const f1: TRawFloat32): TRawFloat32;
```

Returns the cosine of *f1*, where *f1* is an angle in radians.

```
function Float32Tan(const f1: TRawFloat32): TRawFloat32;
```

Returns the tangent of *f1*, where *f1* is an angle in radians.

```
function Float32Cotan(const f1: TRawFloat32): TRawFloat32;
```

Returns the cotangent of *f1*, where *f1* is an angle in radians.

```
function Float32Log2(const f1: TRawFloat32): TRawFloat32;
```

Returns the 2-base logarithm of *f1*.

```
function Float32Ln(const f1: TRawFloat32): TRawFloat32;
```

Returns the N-based logarithm of *f1*.

```
function Float32Log10(const f1: TRawFloat32): TRawFloat32;
```

Returns the 10-based logarithm of *f1*.

```
function Float32IntPow(x: TRawFloat32; n: Int32): TRawFloat32;
```

Returns *f1* to the power *n* (exponent), where exponent is an integer value.

```
function StrToFloat32(const s: PChar; const len: UInt8; out
rerror: boolean): TRawFloat32;
```

Converts a null-terminated string in decimal notation to floating-point number (TRawFloat32) .

Parameters
s: A pointer to a null-terminated string representing floating-point number.
len: The length of string *s*.
rerror: Output parameter indicating failure of the conversion.

Return value
The function returns the number converted. In case of a conversion error, the value returned is undefined.

```
function Float32ToStr(const s: PChar; const maxlen, decplaces:
UInt8; f: TRawFloat32):UInt8;
```

Converts a 32-bit floating-point number to a null-terminated string in decimal notation and stores it in a provided buffer. The function returns the number of characters in the resulting string.

Parameters
s: A pointer to a character buffer where the null-terminated string will be stored. The buffer should be declared as array[0..n] of char.
maxlen: The maximum length of the output string, including the null terminator. This value is typically the size of the *s* buffer. It must be at least 4 to accommodate the shortest possible output strings (e.g., "Inf", "NaN").
decplaces: The desired number of decimal places after decimal separator in the output string. This value should not exceed 8. Note that the accuracy of the conversion depends on the limitations of the TRawFloat32 type and is most effective with 0-6 decimal places.
f: The 32-bit floating-point number to be converted.

Return value
The number of characters in the output string excluding null character. Returns 0 if the s buffer is too small to contain the converted string or special values. For special values, this is the length of the corresponding string written to the buffer. The following special values may be written to the s buffer in specific scenarios:
DTL: “Decimal places too large” - the value for *decplaces* is greater than 8.
NaN: “Not a number” - the input number f is a NaN value.
Inf: “Infinity” - the input number f is Infinity.
Sub: “Subnormal” - the input number f is a subnormal number.

```
function Float32ToStrE(const s: PChar; const maxlen,
decplaces: UInt8; f: TRawFloat32):UInt8;
```

Converts a 32-bit floating-point number to a null-terminated string in scientific notation and stores it in a provided buffer. The output string has the format: 1 digit before the decimal point, followed by *decplaces* number of decimal places, then the character “E”, a plus or minus sign, and a three-digit exponent. A minus sign may precede the number if *f* is negative. The function returns the number of characters in the resulting string.

Parameters
s: A pointer to a character buffer where the null-terminated string will be stored. The buffer should be declared as array[0..n] of char.
maxlen: The maximum length of the output string, including the null terminator. This value is typically the size of the *s* buffer. It must be at least 4 to accommodate the shortest possible output strings (e.g., "Inf", "NaN").
decplaces: The desired number of decimal places after decimal separator (including “E” character) in the output string. This value should not exceed 8. Note that the accuracy of the conversion depends on the limitations of the TRawFloat32 type and is most effective with 0-6 decimal places.
f: The 32-bit floating-point number to be converted.

Return value
The number of characters in the output string excluding null character. If the *s* buffer is too small to contain the converted string, including any special values (e.g., if *maxlen* is less than 4), the function returns 0. The following special values may be written to the s buffer in specific scenarios:
DTL: “Decimal places too large” - the value for decimal places is greater than 8.
BTS: “Buffer too small” - when buffer size is lesser than *decplaces* + 8 (number of characters in scientific format) but is large enough for the "BTS" string.
NaN: “Not a number” - the input number f is a NaN value.
Inf: “Infinity” - the input number f is Infinity.
Sub: “Subnormal” - the input number f is a subnormal number.

```
function Float32Pow(const f1, f2: TRawFloat32): TRawFloat32;
```

Returns *f1* raised to the power *f2*.

```
function Float32Exp(const f1: TRawFloat32): TRawFloat32;
```

Returns the exponent of *f1*, i.e. the number *e* raised to the power *f1*.

TFloat32

TFloat32 represents floating-point number of single precision, based on *TRawFloat32*. It is optimized for simpler use, allowing to use operators like +, -, *, / etc., but generates bigger output code than using its base type.

TFloat32String is a string buffer of 21 chars for string representation of *TFloat32* numbers.

TFloat32 routines

`class operator + (f1, f2: TFloat32): TFloat32;`
`class operator - (f1, f2: TFloat32): TFloat32;`
`class operator * (f1, f2: TFloat32): TFloat32;`
`class operator / (f1, f2: TFloat32): TFloat32;`
`class operator mod (f1, f2: TFloat32): TFloat32;`
 	
Standard mathematical operations for TFloat32 numbers.
 	
Example
F1, F2, F3: TFloat32;
 	
F1.Create('125.000');
F2.Create(4.000');
F3.Create('0.0');
 	
F3 := F1 + F2; // Result 129.000
F3 := F1 mod F2; // Result 1.000

`constructor` **`Create`**`(const f1: TRawFloat32);` Creates a new TFloat32 from *f1*, where *f1* is TRawFloat32 type.

`constructor` **`Create`**`(Str: TFloat32String);`
 	
Creates a new TFloat32 from *Str*.
 	
Example
 	
F1: TFloat32;
F1.Create('123.456');

`function` **`ToString`**`(DecPlaces: UInt8): TFloat32String;`
`function` **`ToStringE`**`(DecPlaces: UInt8): TFloat32String;`
 	
Converts a TFloat32 to a string.
 	
Example
F1: TFloat32;
Str : String;
 	
F1.Create('123.456');
Str := F1.ToString(6); //123.456000
Str := F1.ToStringE(6); //1.23456E+002

`function` **`StringToFloat32`**`(Str: TFloat32String): TFloat32;` Converts a string to a TFloat32 number.

Example
F1: TFloat32;
Str : String;
 	
F1 := StringToFloat32('123.456');

`function` **`ToInt32`**`(): Int32;` Converts a TFloat32 to an Int32 number.

`function` **`Sign`**`(): Integer;`
`function` **`Frac`**`(): Integer;`
`function` **`Exp`**`(): Integer;`
`function` **`Raw`**`(): Integer;`
Returns the sign, mantissa, exponent and raw value from a TFloat32 number (IEEE 754 standard).

`function` **`SqrtFloat32`**`(const f1: TFloat32): TFloat32;` Returns the square root value of *f1*.
`function` **`AbsFloat32`**`(const f1: TFloat32): TFloat32;` Returns the absolute value of f1.
`function` **`InvFloat32`**`(const f1: TFloat32): TFloat32;` Returns the inverse value of *f1*. Note: correct it so that it returns the result *1/f1*
`function` **`InvSqrtFloat32`**`(const f1: TFloat32): TFloat32;` Returns the inverse square root value of *f1*.
`function` **`Deg2RadFloat32`**`(const f1: TFloat32): TFloat32;` Converts *f1* from degrees to radians.
`function` **`Rad2DegFloat32`**`(const f1: TFloat32): TFloat32;` Converts *f1* from radians to degrees.
`function` **`IntFloat32`**`(const f1: TFloat32): TFloat32;` Returns the integer part of a *f1*.
`function` **`SinFloat32`**`(const f1: TFloat32): TFloat32;` Returns the sine of *f1*, where *f1* is an angle in radians.
`function` **`CosFloat32`**`(const f1: TFloat32): TFloat32;` Returns the cosine of *f1*, where *f1* is an angle in radians.
`function` **`TanFloat32`**`(const f1: TFloat32): TFloat32;` Returns the tangent of *f1*, where *f1* is an angle in radians.
`function` **`CotanFloat32`**`(const f1: TFloat32): TFloat32;`

Returns the cotangent of *f1*, where *f1* is an angle in radians.
`function` **`Log2Float32`**`(const f1: TFloat32): TFloat32;` Returns the 2-base logarithm of *f1*.
`function` **`LnFloat32`**`(const f1: TFloat32): TFloat32;` Returns the N-based logarithm of *f1*.
`function` **`Log10Float32`**`(const f1: TFloat32): TFloat32;` Returns the 10-based logarithm of *f1*.
`function` **`IntPowFloat32`**`(const f1: TFloat32; n: Int32): TFloat32;` Returns *f1* to the power *n* (exponent), where exponent is an integer value.
`function` **`Int32ToFloat32`**`(const value: Int32): TFloat32;` Converts a *value* to a TFloat32 number, where value is Int32 type.
`function` **`NegFloat32`**`(const f1: TFloat32): TFloat32;` Returns a negative value of *f1*.

Example

This test example for the TFloat32 floating-point number implementation validates the accuracy and functionality of floating-point arithmetic and conversion routines on an Arduino Uno board by sending the results over the serial port. It initializes the serial communication by starting the hardware serial port (the Serial object). The main loop repeats the tests every second, providing comprehensive validation of the TFloat32 type and sending the results to the serial port.

To run this example, a Serial Monitor application is needed, such as PuTTY or Tera Term. It can also be tested in the AVRPascal IDE, which includes a built-in Serial Monitor. The serial monitor should be configured as follows:

Baud Rate: 9600 bits per second.

Data Format: 8 data bits, no parity, 1 stop bit (8-N-1).

```
program TestFloat32;

{$IF NOT (DEFINED(atmega328p) or DEFINED(arduinouno) or
DEFINED(arduinonano) or DEFINED(fpc_mcu_atmega328p) or
DEFINED(fpc_mcu_arduinouno) or DEFINED(fpc_mcu_arduinonano))}
 {$Fatal Invalid controller type, expected: atmega328p, arduinouno, or
arduinonano}
{$ENDIF}

{$mode ObjFPC}{H+}
{$MODESWITCH ADVANCEDRECORDS}
```

```pascal
uses
  float32, hardwareserial, timer, stringutils;

var
  StrA, StrB, StrC: string[32];
  F32a, F32b, F32c: TFloat32;
  i: UInt32;
begin

   //type your setup code here
  Serial.Start(9600);
  Delay(1000);

  //program main loop
  //type your code here
  while true do
  begin
    Serial.WriteLn('BEGIN TEST');
    Serial.Flush();

    StrA := 'Hello ' + 'How '+ 'Are ' + 'You';
    Serial.WriteLn(StrA);
    Serial.Flush();

    StrA := 'Very ' + 'Well '+ 'Thank ' + 'You';
    Serial.WriteLn(StrA);
    Serial.Flush();

    StrA := '';
    F32a.RawData.Raw := $44201062; // 640.256
    StrA := F32a.ToString(3);
    Serial.Write('TFloat32 : $44201062 = 640.256  result: ' + StrA);
    Serial.Writeln('');
    Serial.Flush();

    StrA := '';
    F32a.RawData.Raw := $40490FDB; // 3.14159
    StrA := F32a.ToString(5);
    Serial.Write('TFloat32 : $40490FDB = 3.14159  result: ' + StrA);
    Serial.Writeln('');
    Serial.Flush();

    StrA := '';
    F32a.RawData.Raw := $3F49FBE7; // 0.789
    StrA := F32a.ToString(3);
    Serial.Write('TFloat32 : $3F49FBE7 = 0.789  result: ' + StrA);
    Serial.Writeln('');
    Serial.Flush();
    Serial.Flush();

    StrA := '';
    F32a.RawData.Raw := $C2029DB2; // -32.654
    StrA := F32a.ToString(3);
    Serial.Write('TFloat32 : $C2029DB2 = -32.654  result: ' + StrA);
    Serial.Writeln('');
    Serial.Flush();

    StrA := '';
    F32a := TFloat32.Create('789.123');
```

```
StrA := F32a.ToString(3);
Serial.Write('TFloat32.Create(''798.123'') result: ' + StrA);
Serial.Writeln('');
Serial.Flush();

StrA := '';
F32b := TFloat32.Create('3.14159');
F32c := Rad2DegFloat32(F32b);
StrB := '';
StrC := F32c.ToString(5);
Serial.Write('Rad2DegFloat32(3.14159) result: ' + StrC);
Serial.Writeln('');
Serial.Flush();

StrC := '';
F32b := TFloat32.Create('3.14159');
F32c := SinFloat32(F32b);
StrC := F32c.ToString(5);
Serial.Write('SinFloat32(3.14159) result: ' + StrC);
Serial.Writeln('');
Serial.Flush();

StrC := '';
F32b := TFloat32.Create('3.14159');
F32c := CosFloat32(F32b);
StrC := F32c.ToString(5);
Serial.Write('CosFloat32(3.14159) result: ' + StrC);
Serial.Writeln('');
Serial.Flush();

StrA := '';
StrB := '';
StrC := '';
F32a := TFloat32.Create('10.0');
F32b := TFloat32.Create('21.0');

F32c := F32a + F32b;
StrC := F32c.ToString(6);
StrA := F32A.ToString(6);
StrB := F32B.ToString(6);
Serial.Write(StrA + ' + ' + StrB + ' = ' + StrC);
Serial.Writeln('');
Serial.Flush;

StrA := '';
StrB := '';
StrC := '';
F32a := TFloat32.Create('125.0');
F32b := TFloat32.Create('4.0');
F32C := F32a mod F32b;
StrC := F32c.ToString(6);
StrA := F32A.ToString(6);
StrB := F32B.ToString(6);
Serial.Write(StrA + ' mod ' + StrB + ' = ' + StrC);
Serial.Writeln('');
Serial.Flush;

Serial.WriteLn('END TEST');
Serial.Flush();
```

```
    Delay(1000);
  end;
end.
```