UFPI Scripting Language

User Manual

September 2020

Table Of Contents

Introduction....7
The script language....7
Data types....7
Primitives....7
Strings....8
Operators....9
Methods....10
Functions....11
Arrays....12
Operators....13
Methods....13
Objects and handles....15
Objects....15
Object handles....15
Function handles....16
UFPI Global Functions and Enums....19
error Function....19
errorString Function....19
icPower Function....19
icProbe Function....19
icPartition Function....19
icRead Function....19
icWrite Function....20
icVerify Function....20
icBlank Function....20
icErase Function....20
icID Function....20
icModel Function....20
icManufacturer Function....20
icPageSize Function....20
icSectorSize Function....21
icBlockSize Function....21
icSize Function....21
mode Function....21
msCount Function....21
msSleep Function....21
playSound Function....22
print Function....22
printa Function....22
progress Function....22
rand Function....22
reverseBits8 Function....22
reverseBits16 Function....22
reverseBits32 Function....22
reverseBits64 Function....22
reverseBytes16 Function....23
reverseBytes32 Function....23
reverseBytes64 Function....23
start Function....23
startDetached Function....23
taskBreak Function....23
taskString Function....23
verbose Function....23

version Function....24
waitForTask Fucntion....24
UFPI Objects....24
TBOX Object....24
buttonPressed Member....25
TBDM Object....25
TBUFFER Object....26
compare Member....26
copy Member....26
dump Member....26
fill Member....26
findBytes Member....27
findInt32 Member....27
findInt64 Member....27
findString Member....27
move Member....28
read8 Member....28
read16 Member....28
read32 Member....28
read64 Member....28
readString Member....28
resize Member....28
set Member....29
size Member....29
write8 Member....29
write16 Member....29
write32 Member....29
write64 Member....29
writeString member....29
TDIALOG Object....30
TDIR Object....31
createDir Member....32
createPath Member....32
current Member....32
exists Member (overloaded)....32
fromNative Member....32
home Member....32
list Member....32
next Member....33
path Member....33
removeDir Member....33
removePath Member....33
removeRecursively Member....34
setCurrent Member....34
setPath Member....34
temp Member....34
toNative Member....34
TFILE Object....34
open Member....34
close Member....35
path Member (overloaded)....35
read8 Member....35
read16 Member....35
read32 Member....35

read64 Member....36
readBuffer Member....36
readString Member....36
remove Member....36
rename Member....36
setFileName Member....36
seek Member....36
size Member....36
write8 Member....36
write16 Member....37
write32 Member....37
write64 Member....37
writeBuffer Member....37
writeString Member....37
TFILE2 Object....37
TGPIO Object....37
power Member....37
clear Member....37
dir Member....38
dirIn Member....38
dirOut Member....38
framesAdd Member....38
framesAnswer Member....39
framesClear Member....39
framesSend Member....39
mask Member....39
mread Member....39
mwrite Member....39
read Member....40
set Member....40
write Member....40
waitLow Member....40
waitHigh Member....40
waitFall Member....40
waitRise Member....40
TGUI Object....40
progressStart Member....41
progressAdd Member....41
progressValue Member....41
progressEnd Member....41
fileNameRead Member....41
fileNameWrite Member....41
filePathRead Member....41
filePathWrite Member....41
THASH Object....41
TI2C Object....43
TNAND Object....44
addr Member....44
cmd Member....44
eraseBlock Member....44
pageReadBuffer....44
pageWriteBuffer....44
read8 Member....44
read16 Member....44

read32 Member....45
read64 Member....45
readBuffer Member....45
write8 Member....45
write16 Member....45
write32 Member....45
write64 Member....45
writeBuffer Member....45
setBus Member....45
setCE Member....46
setSerialAccess Member....46
setWidth Member....46
TNOR Object....46
TOW Object....46
TPARTITIONS Object....47
add Member....47
clear Member....47
count Member....47
flags Member....47
partAddr Member....48
partFileName Member....48
partFilePos Member....48
partFS Member....48
partName Member....48
partSize Member....48
partUsed Member....48
remove Member....49
saveToUDEV Member....49
size Member....49
sizeUsed Member....49
setFlags Member....49
setPartAddr Member....49
setPartFileName Member....49
setPartFilePos Member....49
setPartFS Member....49
setPartName Member....50
setPartSize Member....50
setPartUsed Member....50
TSDMMC Object....50
TSPIFI Object....51
TUART Object....53
power Member....53
gpioClear Member....53
gpioDir Member....53
gpioDirState Member....53
gpioPins Member....53
gpioPulse Member....54
gpioSet Member....54
lineStatus Member....54
modemControl Member....54
modemStatus Member....55
rx8 Member....55
rxBuffer Member....55
rxClear Member....55

rxCount Member....55
rxSize Member....56
setBaud Member....56
setBits Member....56
setFlow Member....56
setModemControl Member....56
setParity Member....56
setRxDelay Member....56
setStop Member....56
setTxPin Member....57
tx8 Member....57
txBuffer Member....57

Introduction

The UFPI Scripting Language (USCR) it's powerful C-like scripting language. USCR primitives have direct matches in C++ (e.g., void, int8, int16, int (int32), int64, uint8, uint16, uint (uint32), uint64, float, double, bool). USCR has a registered string type and uses the standard std::string C++ string type. Script file can be encoded with ASCII or UTF-8. USCR functions uses internal memory buffer during execution. To access this buffer, use TBUFFER global object and his member functions. USCR does not have a built-in dynamic array type. Every USCR script must have main function, "hello world" script code:

```
int main()
{
     print( "Hello world!" );

     return 0;
}
```

Every script have global functions, objects and enum values. Such global objects as TSPIFI or TI2C defined only with connected SPIFI or I2C sockets.

The script language

Data types

Primitives

`void`

void is not really a data type, more like lack of data type. It can only be used to tell the compiler that a function doesn't return any data.

`bool`

bool is a boolean type with only two possible values: true or false. The keywords true and false are constants of type bool that can be used as such in expressions.

Integer numbers

type	min value	max value
int8	-128	127
int16	-32,768	32,767
int	-2,147,483,648	2,147,483,647
int64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint8	0	255
uint16	0	65,535
uint	0	4,294,967,295
uint64	0	18,446,744,073,709,551,615

As the scripting engine has been optimized for 32 bit data types, using the smaller variants is only recommended for accessing application specified variables. For local variables it is better to use the 32 bit

variant. int32 is an alias for int, and uint32 is an alias for uint.

Real numbers

type	range of values	smallest positive	max.digits
float	+/- 3.402823466e+38	1.175494351e-38	6
double	+/- 1.79769313486231e+308	2.22507385850720e-308	15

Note

These numbers assume the platform uses the IEEE 754 to represent floating point numbers in the CPU. Rounding errors may occur if more digits than the maximum number of digits are used. Real numbers may also have the additional values of positive and negative 0 or infinite, and NaN (Not-a-Number). For float NaN is represented by the 32 bit data word 0x7fc00000.

Strings

Strings hold an array of bytes or 16bit words depending on the application settings. Normally they are used to store text but can really store any kind of binary data.

There are two types of string constants supported in the script language, the normal quoted string, and the documentation strings, called heredoc strings.

The normal strings are written between double quotation marks (") or single quotation marks ('). Inside the constant strings some escape sequences can be used to write exact byte values that might not be possible to write in your normal editor.

sequence	value	description
\0	**0**	**null character**
****	**92**	**back-slash**
\'	**39**	**single quotation mark (apostrophe)**
\"	**34**	**double quotation mark**
\n	**10**	**new line feed**
\r	**13**	**carriage return**
\t	**9**	**tab character**
\xFFFF	**0xFFFF**	**FFFF** should be exchanged for a 1 to 4 digit hexadecimal number representing the value wanted. If the application uses 8bit strings then only values up to 255 is accepted.
\uFFFF	**0xFFFF**	**FFFF** should be exchanged for the hexadecimal number representing the unicode code point
\UFFFFFFFF	0xFFFFFFFF	FFFFFFFF should be exchanged for the hexadecimal number representing the unicode code point

```
string str1 = "This is a string with \"escape sequences" .";

string str2 = 'If single quotes are used then double quotes can be
included without "escape sequences".';
```

The heredoc strings are designed for inclusion of large portions of text without processing of escape sequences.

A heredoc string is surrounded by triple double-quotation marks ("""), and can span multiple lines of code. If the characters following the start of the string until the first linebreak only contains white space, it is automatically removed by the compiler. Likewise if the characters following the last line break until the end of the string only contains white space this is also removed.

```
string str = """
This is some text without "escape sequences". This is some text.
This is some text. This is some text. This is some text. This is
some text. This is some text. This is some text. This is some
text. This is some text. This is some text. This is some text.
This is some text.
""";
```

If more than one string constants are written in sequence with only whitespace or comments between them the compiler will concatenate them into one constant.

```
string str = "First line.\n"
             "Second line.\n"
             "Third line.\n";
```

The escape sequences \u and \U will add the specified unicode code point as a UTF-8 or UTF-16 encoded sequence depending on the application settings. Only valid unicode 5.1 code points are accepted, i.e. code points between U+D800 and U+DFFF (reserved for surrogate pairs) or above U+10FFFF are not accepted.

Supporting string object and functions

The string object supports a number of operators, and has several class methods and supporting global functions to facilitate the manipulation of strings.

Operators

= assignment

The assignment operator copies the content of the right hand string into the left hand string.

Assignment of primitive types is allowed, which will do a default transformation of the primitive to a string.

+, += concatenation

The concatenation operator appends the content of the right hand string to the end of the left hand string.

Concatenation of primitives types is allowed, which will do a default transformation of the primitive to a string.

==, != equality

Compares the content of the two strings.

<, >, <=, >= comparison

Compares the content of the two strings. The comparison is done on the byte values in the strings, which may not correspond to alphabetical comparisons for some languages.

[] index operator

The index operator gives access to a single byte in the string.

Methods

uint length() const

Returns the length of the string.

void resize(uint)

Sets the length of the string.

bool isEmpty() const

Returns true if the string is empty, i.e. the length is zero.

string substr(uint start = 0, int count = -1) const

Returns a string with the content starting at start and the number of bytes given by count. The default arguments will return the whole string as the new string.

void insert(uint pos, const string &in other)

Inserts another string other at position pos in the original string.

void erase(uint pos, int count = -1)

Erases a range of characters from the string, starting at position pos and counting count characters.

int findFirst(const string &in str, uint start = 0) const

Find the first occurrence of the value str in the string, starting at start. If no occurrence is found a negative value will be returned.

int findLast(const string &in str, int start = -1) const

Find the last occurrence of the value str in the string. If start is informed the search will begin at that position, i.e. any potential occurrence after that position will not be searched. If no occurrence is found a negative value will be returned.

int findFirstOf(const string &in chars, int start = 0) const

int findFirstNotOf(const string &in chars, int start = 0) const

int findLastOf(const string &in chars, int start = -1) const

int findLastNotOf(const string &in chars, int start = -1) const

The first variant finds the first character in the string that matches on of the characters in chars, starting at start. If no occurrence is found a negative value will be returned.

The second variant finds the first character that doesn't match any of those in chars. The third and last variant are the same except they start the search from the end of the string.

Note

These functions work on the individual bytes in the strings. They do not attempt to understand encoded characters, e.g. UTF-8 encoded characters that can take up to 4 bytes.

Functions

string join(const array<string> &in arr, const string &in delimiter)

Concatenates the strings in the array into a large string, separated by the delimiter.

int64 parseInt(const string &in str, uint base = 10, uint &out byteCount = 0)

uint64 parseUInt(const string &in str, uint base = 10, uint &out byteCount = 0)

Parses the string for an integer value. The base can be 10 or 16 to support decimal numbers or hexadecimal numbers. If byteCount is provided it will be set to the number of bytes that were considered as part of the integer value.

double parseFloat(const string &in, uint &out byteCount = 0)

Parses the string for a floating point value. If byteCount is provided it will be set to the number of bytes that were considered as part of the value.

string formatInt(int64 val, const string &in options = '', uint width = 0)

string formatUInt(uint64 val, const string &in options = '', uint width = 0)

string formatFloat(double val, const string &in options = '', uint width = 0, uint precision = 0)

The format functions takes a string that defines how the number should be formatted. The string is a combination of the following characters:

l = left justify

0 = pad with zeroes

+ = always include the sign, even if positive

space = add a space in case of positive number

h = hexadecimal integer small letters (not valid for formatFloat)

H = hexadecimal integer capital letters (not valid for formatFloat)

e = exponent character with small e (only valid for formatFloat)

E = exponent character with capital E (only valid for formatFloat)

Examples:

// Left justify number in string with 10 characters

```
string justified = formatInt(number, 'l', 10);
```

// Create hexadecimal representation with capital letters, right justified

```
string hex = formatInt(number, 'H', 10);
```

// Right justified, padded with zeroes and two digits after decimal separator

```
string num = formatFloat(number, '0', 8, 2);
```

Arrays

It is possible to declare array variables with the array identifier followed by the type of the elements within angle brackets.

Example:

```
array<int> a, b, c;
array<Foo@> d;
```

a, b, and c are now arrays of integers, and d is an array of handles to objects of the Foo type.

When declaring arrays it is possible to define the initial size of the array by passing the length as a parameter to the constructor. The elements can also be individually initialized by specifying an initialization list. Example:

```
array<int> a;  // A zero-length array of integers
array<int> b(3);  // An array of integers with 3 elements
array<int> c(3, 1);  // An array of integers with 3 elements, all set to 1 by default
array<int> d = {5,6,7};  // An array of integers with 3 elements with specific values
```

Multidimensional arrays are supported as arrays of arrays, for example:

```
array<array<int>> a; // An empty array of arrays of integers
array<array<int>> b = {{1,2},{3,4}}; // A 2 by 2 array with initialized values
array<array<int>> c(10, array<int>(10)); // A 10 by 10 array of integers with uninitialized values
```

Each element in the array is accessed with the indexing operator. The indices are zero based, i.e. the range of valid indices are from 0 to length - 1.

```
a[0] = some_value;
```

When the array stores handles the elements are assigned using the handle assignment.

// Declare an array with initial length 1

```
array<Foo@> arr(1);
```

// Set the first element to point to a new instance of Foo

```
@arr[0] = Foo();
```

Arrays can also be created and initialized within expressions as anonymous objects.

// Call a function that expects an array of integers as input

```
foo({1,2,3,4});
```

// If the function has multiple overloads supporting different types with

// initialization lists it is necessary to explicitly inform the array type

```
foo2(array<int> = {1,2,3,4});
```

Supporting array object and functions

The array object supports a number of operators and has several class methods to facilitate the manipulation of strings.

The array object is a reference type even if the elements are not, so it's possible to use handles to the array object when passing it around to avoid costly copies.

Operators

= assignment

The assignment operator performs a shallow copy of the content.

[] index operator

The index operator returns the reference of an element allowing it to be inspected or modified. If the index is out of range, then an exception will be raised.

==, != equality

Performs a value comparison on each of the elements in the two arrays and returns true if all match the used operator.

Methods

uint length() const

Returns the length of the array.

void resize(uint)

Sets the new length of the array.

void reverse()

Reverses the order of the elements in the array.

void insertAt(uint index, const T& in value)

```
void insertAt(uint index, const array<T>& arr)
```

Inserts a new element, or another array of elements, into the array at the specified index.

```
void insertLast(const T& in)
```

Appends an element at the end of the array.

```
void removeAt(uint index)
```

Removes the element at the specified index.

```
void removeLast()
```

Removes the last element of the array.

```
void removeRange(uint start, uint count)
```

Removes count elements starting from start.

```
void sortAsc()
void sortAsc(uint startAt, uint count)
```

Sorts the elements in the array in ascending order. For object types, this will use the type's opCmp method.

The second variant will sort only the elements starting at index startAt and the following count elements.

```
void sortDesc()
void sortDesc(uint startAt, uint count)
```

These does the same thing as sortAsc except sorts the elements in descending order.

```
void sort(const less &in compareFunc, uint startAt = 0, uint count =
uint(-1))
```

This method takes as input a callback function to use for comparing two elements when sorting the array.

The callback function should take as parameters two references of the same type of the array elements and it should return a bool. The return value should be true if the first argument should be placed before the second argument.

```
array<int> arr = {3,2,1};
arr.sort(function(a,b) { return a < b; });
```

The example shows how to use the sort method with a callback to an anonymous function.

```
int find(const T& in)
int find(uint startAt, const T& in)
```

These will return the index of the first element that has the same value as the wanted value. For object types, this will use the type's opEquals or opCmp method to compare the value. For arrays of handles any null handle will be skipped. If no match is found the methods will return a negative value.

int findByRef(const T& in)

int findByRef(uint startAt, const T& in)

These will search for a matching address. These are especially useful for arrays of handles where specific instances of objects are desired, and not just objects that happen to have equal value. If no match is found the methods will return a negative value.

Script example:

```
int main()
{
     array<int> arr = {1,2,3}; // 1,2,3
     arr.insertLast(0);        // 1,2,3,0
     arr.insertAt(2,4);        // 1,2,4,3,0
     arr.removeAt(1);          // 1,4,3,0
     arr.sortAsc();            // 0,1,3,4
     int sum = 0;
     for(uint n = 0; n < arr.length(); n++) sum += arr[n];
     return sum;
}
```

Objects and handles

Objects

There are two forms of objects, reference types and value types.

Value types behave much like the primitive types, in that they are allocated on the stack and deallocated when the variable goes out of scope. Only the application can register these types, so you need to check with the application's documentation for more information about the registered types.

Reference types are allocated on the memory heap, and may outlive the initial variable that allocates them if another reference to the instance is kept. All script declared classes are reference types. Interfaces are a special form of reference types, that cannot be instantiated, but can be used to access the objects that implement the interfaces without knowing exactly what type of object it is.

obj o; // An object is instantiated

o = obj(); // A temporary instance is created whose value is assigned to the variable

Object handles

Object handles are a special type that can be used to hold references to other objects. When calling methods or accessing properties on a variable that is an object handle you will be accessing the actual object that the handle references, just as if it was an alias. Note that unless initialized with the handle of an object, the handle is null.

```
obj o;
obj@ a;  // a is initialized to null
obj@ b = @o; // b holds a reference to o
b.ModifyMe(); // The method modifies the original object
if ( a is null ) // Verify if the object points to an object
{
      @a = @b;  // Make a hold a reference to the same object as b
}
```

Not all types allow a handle to be taken. Neither of the primitive types can have handles, and there may exist some object types that do not allow handles. Which objects allow handles or not, are up to the application that registers them.

Object handle and array type modifiers can be combined to form handles to arrays, or arrays of handles, etc.

Function handles

A function handle is a data type that can be dynamically set to point to a global function that has a matching function signature as that defined by the variable declaration. Function handles are commonly used for callbacks, i.e. where a piece of code must be able to call back to some code based on some conditions, but the code that needs to be called is not known at compile time.

To use function handles it is first necessary to define the function signature that will be used at the global scope or as a member of a class. Once that is done the variables can be declared using that definition.

Here's an example that shows the syntax for using function handles

```
// Define a function signature for the function handle
funcdef bool CALLBACK(int, int);
// An example function that shows how to use this
void main()
{
    // Declare a function handle, and set it
    // to point to the myCompare function.
    CALLBACK @func = @myCompare;
    // The function handle can be compared with the 'is' operator
    if( func is null )
    {
      print("The function handle is null\n");
      return;
    }
```

```
    // Call the function through the handle, just as if it was a normal
function
    if( func(1, 2) )
    {
      print("The function returned true\n");
    }
    else
    {
      print("The function returned false\n");
    }
}
  // This function matches the CALLBACK definition, since it has
  // the same return type and parameter types.
  bool myCompare(int a, int b)
  {
    return a > b;
  }
```

Delegates

It is also possible to take function handles to class methods, but in this case the class method must be bound to the object instance that will be used for the call. To do this binding is called creating a delegate, and is done by performing a construct call for the declared function definition passing the class method as the argument.

```
class A
{
     bool Cmp(int a, int b)
     {
       count++;
       return a > b;
    }
    int count = 0;
}

void main()
{
     A a;
    // Create the delegate for the A::Cmp class method
    CALLBACK @func = CALLBACK(a.Cmp);
    // Call the delegate normally as if it was a global function
```

```
    if( func(1,2) )
    {
      print("The function returned true\n");
    }
    else
    {
      print("The function returned false\n");
    }
    printf("The number of comparisons is "+a.count+"\n");
}
```

Auto declarations

It is possible to use 'auto' as the data type of an assignment-style variable declaration.

The appropriate type for the variable(s) will be automatically determined.

`auto i = 18;` // i will be an integer

`auto f = 18 + 5.f;` // the type of f resolves to float

`auto anObject = getLongObjectTypeNameById(id);` // avoids redundancy for long type names

Auto can be qualified with const to force a constant value:

`const auto i = 2;` // i will be typed as 'const int'

If receiving object references or objects by value, auto will make a local object copy by default. To force a handle type, add '@'.

```
obj getObject()
{
     return obj();
}
auto value = getObject(); // auto is typed 'obj', and makes a local copy
auto@ handle = getObject(); // auto is typed 'obj@', and refers to the returned obj
```

The '@' specifier is not necessary if the value already resolves to a handle:

```
obj@ getObject()
{
     return obj();
}
auto value = getObject();  // auto is already typed 'obj@', because of the return type of getObject()
auto@ value = getObject();  // this is still allowed if you want to be more explicit, but not needed
```

Auto handles can not be used to declare class members, since their resolution is dependent on the constructor.

UFPI Global Functions and Enums

error Function

`int error()`
Returns the last error code value.

errorString Function

`string errorString()`
`void errorString(string &out str)`
Returns string with the last error code verbose description.

icPower Function

`bool icPower(int value, int mv=-1)`
Sets power state, voltage and flags. Returns true if successful, otherwise returns false.

Enumerated power state values:
```
POWER_OFF
POWER_ON
POWER_FLAGS
```

Enumerated power flags values:
```
POWER_FLAG_SOFT_ON_SET
POWER_FLAG_SOFT_ON_CLEAR
POWER_FLAG_ALWAYS_ON_SET
POWER_FLAG_ALWAYS_ON_CLEAR
```

Examples:
```
icPower(POWER_ON,1800); // Power On with 1.8V voltage
icPower(POWER_ON|POWER_FLAG_SOFT_ON_SET); // Power On softly
icPower(POWER_FLAGS|POWER_FLAG_ALWAYS_ON_CLEAR); // Clear Always On flag
```

icProbe Function

`bool icProbe()`
Detects IC ID and parameters. Returns true if successful, otherwise returns false.

icPartition Function

`bool icPartition(int idx)`
Sets active eMMC partition to *idx*. Returns true if successful, otherwise returns false.

icRead Function

`bool icRead(string fname, int64 addr=0, int64 size=0)`
Reads IC into the file with filename *fname*. If *fname* value is equal "buffer" internal BUFFER will be used. Returns true if successful, otherwise returns false.

Examples:
```
icRead("buffer"); // Read entire IC into BUFFER
```

```
icRead("out.bin", 0x20000); // Read IC from 0x20000 to end
icRead("out.bin", 0x20000, 0x30000); // Read IC from 0x20000, size 0x30000
```

icWrite Function

`bool icWrite(string fname, int64 addr=0, int64 size=0)`
Writes IC using data from file with filename *fname*. If *fname* value is equal "buffer" internal BUFFER will be used. Returns true if successful, otherwise returns false.

icVerify Function

`bool icVerify(string fname, int64 addr=0, int64 size=0)`
Verify IC using data from file with filename *fname*. If *fname* value is equal "buffer" internal BUFFER will be used. Returns true if successful, otherwise returns false.

icBlank Function

`bool icBlank(int64 addr=0, int64 size=0)`
IC Blank check Returns true if successful, otherwise returns false.

icErase Function

`bool icErase(int64 addr=0, int64 size=0)`
Erase IC. Returns true if successful, otherwise returns false.

icID Function

`string icID()`
Returns IC identification string.

icModel Function

`string icModel()`
Returns IC model.

icManufacturer Function

`string icManufacturer()`
Returns IC manufacturer.

icPageSize Function

`int icPageSize(int param=0)`
Returns IC page size.

Example:
```
int page = icPageSize();
int spage = icPageSize(1); // Page with spare for NAND IC's
```

icSectorSize Function

```
int icSectorSize(int idx=0)
```
Returns IC sector size with the index *idx*.

icBlockSize Function

`int icBlockSize(int param=0)`
Returns IC block size.

icSize Function

`int64 icSize(int param=0)`
Returns IC size.

Example:
```
int64 size=icSize(); // USER size for eMMC
int64 size=icSize(2); // BOOT2 size for eMMC
```

mode Function

`int mode()`
Returns UFPI operating mode.

Enumerated mode values:
```
MODE_DISCONNECTED
MODE_UFPI
MODE_UPDATE
MODE_NAND
MODE_ONENAND
MODE_SNAND
MODE_SDMMC
MODE_SPIFI
MODE_NOR
MODE_JTAG
MODE_BDM
MODE_I2C
MODE_1W
MODE_SPIEE
MODE_MWIRE
MODE_UART
MODE_LOGGER
```

msCount Function

`int64 msCount()`
Returns the number of milliseconds since 1970-01-01T00:00:00 Universal Coordinated Time.

msSleep Function

`void msSleep(int ms)`
Sleeps *ms* milliseconds.

playSound Function

`void playSound(const string &in fname)`
Starts playing the sound specified by file with the filename *fname*. If *fname* value equal “done” or “error” will be used filenames specified in the Settings, Media. The function returns immediately.

print Function

`void print(const string &in text)`
Add string *text* to log.

printa Function

`void printa(const string &in text)`
Appends string *text* to the last line in the log.

progress Function

`void progress(bool val)`
Sets progress calculation ON/OFF for high-level UFPI functions calls.

rand Function

`int rand()`
Returns a value between 0 and RAND_MAX (32767), the next number in the current sequence of pseudo-random integers.

Examples:
```
int value=rand()%100; // generate value in range from 0 to 99
int value=rand()%200+1; // generate value in range from 1 to 200
int value=rand()%30+1991; // generate value in range from 1991 to 2020
```

reverseBits8 Function

`uint8 reverseBits8(uint8 val)`
Returns 8-bit value *val* with reversed bits order.

reverseBits16 Function

`uint16 reverseBits16(uint16 val)`
Returns 16-bit value *val* with reversed bits order.

reverseBits32 Function

`uint32 reverseBits32(uint32 val)`
Returns 32-bit value *val* with reversed bits order.

reverseBits64 Function

`uint64 reverseBits64(uint64 val)`
Returns 64-bit value *val* with reversed bits order.

reverseBytes16 Function

`uint16 reverseBytes16(uint16 val)`
Returns 16-bit value *val* with reversed byte order, converting little-endian values to big-endian (and vice versa).

reverseBytes32 Function

`uint32 reverseBytes32(uint32 val)`
Returns 32-bit value *val* with reversed byte order, converting little-endian values to big-endian (and vice versa).

reverseBytes64 Function

`uint64 reverseBytes64(uint64 val)`
Returns 64-bit value *val* with reversed byte order, converting little-endian values to big-endian (and vice versa).

start Function

`int start(const string cmd, int msec=30000)`
Starts the command *cmd* in a new process and waits until the process has finished, or until msecs milliseconds have passed. Command is a single string of text containing both the program name and its arguments. The arguments are separated by one or more spaces. Returns -1 if process hasn't started, -2 on timeout; otherwise returns the exit status of the last process that finished.

Example:
```
int res=start("\"C:\\Program Files\\WinRAR\\WinRAR.exe\"");
```

startDetached Function

`bool startDetached(const string cmd)`
Starts the command *cmd* in a new process and detaches from it. Command is a single string of text containing both the program name and its arguments. The arguments are separated by one or more spaces. Returns true on success, otherwise returns false.

Example:
```
int res=startDetached("aplay sound.wav");
```

taskBreak Function

`bool taskBreak()`
Retuns true if stop button is pressed.

taskString Function

`string taskString()`
`void taskString(string &out str)`
Returns the task string parameter. For task like TASK_READ this is a filename.

verbose Function

`void verbose(bool val)`

Sets verbose mode for high-level UFPI functions calls.

version Function

`int version()`
Returns 32-bit value with the information about application version and build date.
Bits [7-0] Build day (1-31).
Bits [11-8] Build month (1-12).
Bits [19-12] Build year (year-2000).
Bits [23-20] Version Build.
Bits [27-24] Version Minor.
Bits [30-28] Version Major.

waitForTask Fucntion

`int waitForTask(int timeout = -1)`
Waits for the user task with timeout in milliseconds.

Example:
```
for (;;)
{
     int task = waitForTask();

     if (task == TASK_EXIT) break;

     switch(task)
     {
          case TASK_READ: funcRead(); break;

          case TASK_WRITE: funcWrite(); break;

          default: funcDefault();
     }
}
```

Enumerated return values:
```
TASK_BREAK
TASK_EXIT
TASK_ID
TASK_INFO
TASK_READ
TASK_WRITE
TASK_ERASE
TASK_VERIFY
TASK_BLANK
TASK_POWER
```

UFPI Objects

TBOX Object

This object defined in any script. Only one TBOX object with name BOX defined.

Access to members: `BOX.someFunc();`

buttonPressed Member

`bool buttonPressed()`
Returns true if Button on the UFPI Box is pressed, otherwise returns false.

TBDM Object

This object defined only with connected BDM socket.
`Access to members: BDM.someFunc();`

`bool ack(int on)`
Enables/Disables Handshake. If *on* value is zero ACK_DISABLE command will be executed. This command does not issue an ACK pulse. If *on* value is non-zero ACK_ENABLE command will be executed. ACK pulse after this command is executed. Returns true if successful, otherwise returns false.

`bool connect()`
Sync, enables ACK, check status register and enables BDM. Returns true if successful, otherwise returns false.

`bool halt()`
Executes BACKGROUND command. Device will enter background mode if firmware is enabled. If enabled, an ACK will be issued when the part enters active background mode. Returns true if successful, otherwise returns false.

`bool power(int on, int mv=-1)`

`bool read8(uint addr, uint8 &value)`
Reads the byte from memory at address *addr* to *value* with the READ_BYTE opcode (BDM firmware lookup table out of map). Returns true if successful, otherwise returns false.

`bool read8bd(uint addr, uint8 &value)`
Reads the byte from memory at address *addr* to *value* with the READ_BD_BYTE opcode (BDM firmware lookup table in map). Returns true if successful, otherwise returns false.

`bool read16(uint addr, uint16 &value)`
Reads the word from memory at address *addr* to *value* with the READ_WORD opcode (BDM firmware lookup table out of map). Returns true if successful, otherwise returns false.

`bool read16bd(uint addr, uint16 &value)`
Reads the byte from memory at address *addr* to *value* with the READ_BD_WORD command (BDM firmware lookup table in map). Returns true if successful, otherwise returns false.

`bool reset(int mode=BDM_RESET_SPECIAL)`
`BDM_RESET_SPECIAL`
`BDM_RESET_NORMAL`

`bool setCore(int core)`
`BDM_CORE_HCS08`
`BDM_CORE_HCS12`

`bool setCPU(const string model)`
Supported CPU models:
`MC9S08AC32`

…
MC9S12XF512

bool sync()
Executes BDM Target-to-Host Serial Bit Timing detection protocol. Returns true if successful, otherwise returns false.

bool write8(uint addr, uint8 value)
Writes the byte with *value* to memory at address *addr* with the WRITE_BYTE command (BDM firmware lookup table out of map). Returns true if successful, otherwise returns false.

bool write8bd(uint addr, uint8 value)
Writes the byte with *value* to memory at address *addr* with the WRITE_BD_BYTE command (BDM firmware lookup table in map). Returns true if successful, otherwise returns false.

bool write16(uint addr, uint16 value)
Writes the word with *value* to memory at address *addr* with the WRITE_WORD command (BDM firmware lookup table out of map). Returns true if successful, otherwise returns false.

bool write16bd(uint addr, uint16 value)
Writes the word with *value* to memory at address *addr* with the WRITE_BD_WORD command (BDM firmware lookup table in map). Returns true if successful, otherwise returns false.

TBUFFER Object

This object defined in any script. Only one TBUFFER object with name BUFFER defined.
Access to members: BUFFER `.someFunc();`

compare Member

int compare(int pos1, int pos2, int size)
Compares two blocks of the memory in TBUFFER with the *size* bytes. Returns negative value (-1, -2) if *pos1*, *pos2* and *size* values are out of BUFFER range. Returns 0 if the contents of booth memory blocks are equal. If block are not equal returns index+1 of the first not matching byte.

copy Member

bool copy(int dest, int src, int size)
Copies size bytes from the memory block with the offset *src* in the BUFFER to the memory block with the offset *dest* in the BUFFER. Memory blocks can't overlap. Returns true if successful, otherwise returns false.

dump Member

void dump(int size, int pos=0, int64 addr=-1)
Prints the buffer *size* bytes at index position *pos* in the BUFFER.

fill Member

bool fill(uint8 value, int size=-1)
Sets every byte in the BUFFER to value. If size is different from -1 (the default), the byte array is resized to size *size* beforehand. Returns true if successful, otherwise returns false.

findBytes Member

int findBytes(int pos, int size, int b0, int b1=-1, int b2=-1, int b3=-1, int b4=-1, int b5=-1, int b6=-1, int b7=-1)
Find the first occurrence of the given bytes in the BUFFER from offset *pos* with the *size* bytes. If *size* is equal zero the search will to the end of the BUFFER. If no occurrence is found a negative value will be returned.

Example:

```
//Search 0x11, 0x22, 0x33 bytes in first 4K of the buffer
int found = BUFFER.findBytes(0, 4096, 0x11, 0x22, 0x33);
```

findInt32 Member

int findInt32(int pos, int size, int32 value)
Find the first occurrence of the int32 *value* in the BUFFER from offset *pos* with the *size* bytes. If *size* is equal zero the search will to the end of the BUFFER. If no occurrence is found a negative value will be returned.

Examples:

```
//Search 0x11, 0x22, 0x33, 0x44 bytes in entire buffer
int found = BUFFER.findInt32(0, 0, 0x44332211);
```

findInt64 Member

int findInt64(int pos, int size, int64 value)
Find the first occurrence of the int64 *value* in the BUFFER from offset *pos* with the *size* bytes. If *size* is equal zero the search will to the end of the BUFFER. If no occurrence is found a negative value will be returned.

findString Member

int findString(const string str, int pos=0, int size=0, int enc=ENCODING_ASCII)
Find the first occurrence of the string *str* with the *enc* encoding in the BUFFER from offset *pos* with the *size* bytes. If *size* is equal zero the search will to the end of the BUFFER. If no occurrence is found a negative value will be returned.

Encoding enums:

```
ENCODING_ASCII
ENCODING_UTF8
ENCODING_UNICODE16
```

Examples:

```
// Search ASCII "1234" string in entire BUFFER
int found = BUFFER.findString("1234");

// Search UNICODE-16 "1234" string in BUFFER region
int found = BUFFER.findString("1234", 0x1000, 4096, ENCODING_UNICODE16);
```

move Member

bool move(int dest, int src, int size)
Copies size bytes from the memory block with the offset *src* in the BUFFER to the memory block with the offset *dest* in the BUFFER. Memory blocks may overlap. Returns true if successful, otherwise returns false.

read8 Member

bool read8(int8 &value, int pos=0)
Reads the byte to value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

read16 Member

bool read16(int16 &value, int pos=0)
Reads the word to value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

read32 Member

bool read32(int32 &value, int pos=0)
Reads the dword to value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

read64 Member

bool read64(int64 &out value, int pos=0)
Reads the qword to value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

readString Member

bool readString(string &str, int pos=0, int len=0, int enc=ENCODING_ASCII)
Reads the string with the *enc* encoding from the BUFFER at index position *pos* to string variable *str*. Returns true if successful, otherwise returns false.

Encoding enums:

```
ENCODING_ASCII
ENCODING_UTF8
ENCODING_UNICODE16
```

Example:

```
// Read 8 UNICODE-16 symbols from the BUFFER at 0x1000
string str = "";
bool ok = BUFFER.readString(str, 0x1000, 8, ENCODING_UNICODE16);
if (ok) print("string:" + str);
```

resize Member

bool resize(int size)
Sets the size of the BUFFER to *size* bytes.

set Member

`bool set(uint8 value, int pos=0, int size=0)`
Sets the *size* bytes of the memory block with offset *pos* in the BUFFER to the specified value.

size Member

`int size()`
Returns the number of bytes in the BUFFER.

write8 Member

`bool write8(int8 value, int pos=0)`
Writes the byte with value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

write16 Member

`bool write16(int16 value, int pos=0)`
Writes the word with value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

write32 Member

`bool write32(int32 value, int pos=0)`
Writes the dword with value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

write64 Member

`bool write64(int64 value, int pos=0)`
Writes the qword with value at index position *pos* in the BUFFER. Returns true if successful, otherwise returns false.

writeString member

`bool writeString(const string str, int pos=0, int enc=ENCODING_ASCII)`
Writes the string *str* with the *enc* encoding into the BUFFER at index position *pos*. Returns true if successful, otherwise returns false.

Encoding enums:

```
ENCODING_ASCII
ENCODING_UTF8
ENCODING_UNICODE16
```

Examples:

```
// Write ASCII "1234"
int found = BUFFER.writeString("1234");
```

```
// Write UNICODE-16 "1234" string into the BUFFER at 0x1000
int found = BUFFER.writeString("1234", 0x1000, ENCODING_UNICODE16);
```

TDIALOG Object

This object defined in any script. Only one TDIALOG object with name DIALOG defined.
Access to members: DIALOG.`someFunc();`

Members

`bool accepted()`

Returns true if executed dialog has been executed and accepted, otherwise returns false.

`int getInt32(const string msg)`

Returns an int32 value entered by the user. If the user presses Cancel, it returns 0. The function creates a modal input dialog with the *msg* message.

`int64 getInt64(const string msg)`

Returns an int64 value entered by the user. If the user presses Cancel, it returns 0. The function creates a modal input dialog with the *msg* message.

`string getOpenFileName(const string dir="", const string filter="")`

Returns an existing file selected by the user. If the user presses Cancel, it returns an empty string. The function creates a modal file dialog. The file dialog's working directory will be set to *dir*. If *dir* includes a file name, the file will be selected. Only files that match the given filter are shown. The filter selected is set to *filter*. The parameters *dir* and *filter* may be empty strings. If you want multiple filters, separate them with ';;', for example: "Binary files (*.bin *.dat);;Text files (*.txt);;Script files (*.uscr)"

`string getSaveFileName(const string dir="", const string filter="")`

Returns a file name selected by the user. The file does not have to exist. It creates a modal file dialog. The file dialog's working directory will be set to *dir*. If *dir* includes a file name, the file will be selected. Only files that match the filter are shown. The filter selected is set to *filter*. The parameters *dir* and *filter* may be empty strings. Multiple filters are separated with ';;'. For instance: "Binary files (*.bin *.ubin);;Text files (*.txt);;XML files (*.xml)".

`string getString(const string msg)`

Returns a text string entered by the user. If the user presses Cancel, it returns an empty string. The function creates a modal input dialog with the *msg* message.

`bool hexEditor(const string fileName="")`

Creates a modal HexEditor dialog with the specified *fileName* or internal BUFFER if the *fileName* not specified

`int messageBox(const string msg, int flags)`

Executes a modal dialog for informing the user or for asking the user a question and receiving an answer. A message box can also display an icon and standard buttons for accepting a user response.

Defined standard button flags:

MB_OK

MB_SAVE

MB_SAVE_ALL

MB_OPEN

MB_YES

MB_YES_TO_ALL

MB_NO

MB_NO_TO_ALL

MB_ABORT

MB_RETRY

MB_IGNORE

MB_CLOSE

MB_CANCEL

MB_DISCARD

MB_HELP

MB_APPLY

MB_RESET

MB_RESTORE_DEFAULTS

Defined icon flags:

MB_ICON_INFORMATION

MB_ICON_WARNING

MB_ICON_ERROR

MB_ICON_QUESTION

Example:

```
int res;
res = DIALOG.messageBox("Continue?", MB_YES|MB_NO|MB_ICON_WARNING);
if (res == MB_YES)
{
     doSomething();
}
```

TDIR Object

This object defined in any script. Only one TDIR object with name DIR defined.

Access to members: DIR.`someFunc();`

createDir Member

`bool createDir(const string dir)`
Creates a sub-directory called *dir*. Returns *true* on success, otherwise returns *false*. If the directory already exists when this function is called, it will return *false*.

createPath Member

`bool createPath(const string dirPath)`
Creates the directory path *dirPath*. The function will create all parent directories necessary to create the directory. Returns *true* if successful, otherwise returns *false*. If the path already exists when this function is called, it will return *true*.

current Member

`string current()`
Returns the application's current directory.

exists Member (overloaded)

`bool exists()`
Returns *true* if the directory exists, otherwise returns *false*. (If a file with the same name is found this function will return *false*). The overload of this function that accepts an argument is used to test for the presence of files and directories within a directory.

`bool exists(const string path)`
Returns *true* if the file called name exists, otherwise returns *false*. Unless name contains an absolute file path, the file name is assumed to be relative to the directory itself, so this function is typically used to check for the presence of files within a directory.

fromNative Member

`string fromNative(const string path)`
Returns *path* using '/' as file separator. On Windows, for instance, fromNative("c:\\Windows") returns "c:/Windows". The returned string may be the same as the argument on some operating systems, for example on Unix.

home Member

`string home()`
Returns the user's home directory.

list Member

`int list(const string nameFilter="", int filter=-1)`
Builds a list of the names of all the files and directories in the directory, ordered according to the *nameFilter* and attribute filters *filter*. List is empty if the directory is unreadable, does not exist, or if nothing matches the specification. Returns number of items in list.

Enumerated Dir filter values:

```
DIR_FILTER_DIRS
DIR_FILTER_FILES
DIR_FILTER_DRIVES
DIR_FILTER_READABLE
DIR_FILTER_WRITABLE
DIR_FILTER_EXECUTABLE
DIR_FILTER_MODIFIED
DIR_FILTER_HIDDEN
DIR_FILTER_SYSTEM
DIR_FILTER_NO_DOT
DIR_FILTER_NO_DOTDOT
DIR_FILTER_NO_SYMLINKS
DIR_FILTER_ALL_DIRS
DIR_FILTER_CASE_SENSITIVE
```

next Member

string next();
Advances the pointer to the next entry in list, and returns the file path of this new entry.

Example:

```
// Print Dirs and system files on drive C:
int i;
DIR.setPath("c:\\");
i=DIR.list("", DIR_FILTER_DIRS|DIR_FILTER_NO_DOT|DIR_FILTER_NO_DOTDOT);
while((i--)>0) print(DIR.next());
i=DIR.list("*.sys", DIR_FILTER_FILES|DIR_FILTER_NO_SYMLINKS|
DIR_FILTER_SYSTEM|DIR_FILTER_HIDDEN);
print("System files:");
while((i--)>0) print(DIR.next());
```

path Member

`string path()`
Returns the path. This may contain symbolic links, but never contains redundant ".", ".." or multiple separators. The returned path can be either absolute or relative (see **`setPath()`**).

removeDir Member

`bool removeDir(const string dir)`
Removes the directory specified by *dir*. The directory must be empty to succeed. Returns *true* if successful, otherwise returns *false*.

removePath Member

`bool removePath(const string dirPath)`
Removes the directory path *dirPath*. The function will remove all parent directories in *dirPath*, provided that they are empty. Returns *true* if successful, otherwise returns *false*.

removeRecursively Member

`bool removeRecursively()`
Removes the directory, including all its contents. Returns *true* if successful, otherwise *false*. If a file or directory cannot be removed, removeRecursively() keeps going and attempts to delete as many files and sub-directories as possible, then returns *false*. If the directory was already removed, the method returns *true* (expected result already reached).

setCurrent Member

`bool setCurrent(const string path)`
Sets the application's current working directory to *path*. Returns *true* if the directory was successfully changed, otherwise returns *false*.

setPath Member

`void setPath(const string path)`
Sets the path of the directory to *path*. The path is cleaned of redundant ".", ".." and of multiple separators. No check is made to see whether a directory with this path actually exists, but you can check for yourself using exists(). The *path* can be either absolute or relative. Absolute paths begin with the directory separator "/" (optionally preceded by a drive specification under Windows). Relative file names begin with a directory name or a file name and specify a path relative to the current directory.

temp Member

`string temp()`
Returns the system's temporary directory.

toNative Member

`string toNative(const string path)`
Returns *path* with the '/' separators converted to separators that are appropriate for the underlying operating system. On Windows, toNative("c:/Windows") returns "c:\Windows". The returned string may be the same as the argument on some operating systems, for example on Unix.

TFILE Object

This object defined in any script. Only one TFILE object with name FILE defined.
Access to members: FILE.`someFunc();`

open Member

`bool open(const string mode)`
Opens the file using *mode*, returning true if successful, otherwise false. The mode must be “r”, “w”, or “rw”. It may also have additional flags, such as “b” and “a”. In WriteOnly or ReadWrite mode, if the relevant file does not already exist, this function will try to create a new file before opening it.

Example:
`open("rwb");` // Open file for reading and writing in binary mode. File content will be cleared.
`open("rb+");` // Open file for reading and writing in binary mode. File content will be updated.

Mode flags:
r – Read access
w – Write access
b – Binary mode
a – Append mode
+ – Update mode

bool open(const string filename, const string mode)
Opens the file with filename in the given mode. Returns true if successful, otherwise returns false.

Examples:
open("test.bin", "rwb"); // Open file for reading and writing in binary mode

Mode flags:
r – Read access
w – Write access
b – Binary mode
a – Append mode
+ – Update mode

close Member

void close()
Closes the FILE.

path Member (overloaded)

string path()
Returns a FILE filename's path absolute path. This doesn't include the file name. On Unix the absolute path will always begin with the root, '/', directory. On Windows this will always begin 'D:/' where D is a drive letter, except for network shares that are not mapped to a drive letter, in which case the path will begin '//sharename/'.

string path(const string fname)
Returns a *fname* file's path absolute path.

read8 Member

bool read8(int8 &value)
Reads the byte to value from the FILE. Returns true if successful, otherwise returns false.

read16 Member

bool read16(int16 &value)
Reads the word to value from the FILE. Returns true if successful, otherwise returns false.

read32 Member

bool read32(int32 &value)
Reads the dword to value from the FILE. Returns true if successful, otherwise returns false.

read64 Member

bool read64(int64 &value)
Reads the byte to value from the FILE. Returns true if successful, otherwise returns false.

readBuffer Member

bool readBuffer(int size, int pos=0)
Reads *size* bytes from the FILE into BUFFER with index position *pos*. Returns true if successful, otherwise returns false.

readString Member

bool readString(string &str, int len=0, int enc=ENCODING_ASCII)
Reads the string with the *enc* encoding from the FILE to the string variable *str*. Returns true if successful, otherwise returns false.

remove Member

bool remove()
Removes the file. Returns true if successful, otherwise returns false. The file is closed before it is removed.

rename Member

bool rename(const string newName)
Renames the file currently specified to *newName*. Returns true if successful, otherwise returns false. If a file with the name *newName* already exists, rename() returns false (i.e., TFILE will not overwrite it). The file is closed before it is renamed. If the rename operation fails, TFILE will attempt to copy this file's contents to *newName*, and then remove this file, keeping only *newName*. If that copy operation fails or this file can't be removed, the destination file *newName* is removed to restore the old state.

setFileName Member

void setFileName(const string name)
Sets the *name* of the file. The name can have no path, a relative path, or an absolute path. Do not call this function if the file has already been opened. If the file name has no path or a relative path, the path used will be the application's current directory path at the time of the open() call.

seek Member

bool seek(int64 pos)
Sets the current position in the FILE to *pos*. Returns true if successful, otherwise returns false.

size Member

int64 size()
Returns size of the FILE.

write8 Member

bool write8(int8 value)

Writes the byte with *value* to the FILE. Returns true if successful, otherwise returns false.

write16 Member

`bool write16(int16 value)`
Writes the word with *value* to the FILE. Returns true if successful, otherwise returns false.

write32 Member

`bool write32(int32 value)`
Writes the dword with *value* to the FILE. Returns true if successful, otherwise returns false.

write64 Member

`bool write64(int64 value)`
Writes the qword with *value* to the FILE. Returns true if successful, otherwise returns false.

writeBuffer Member

`bool writeBuffer(int size, int pos=0)`
Writes *size* bytes from the BUFFER with index position *pos* into the FILE. Returns true if successful, otherwise returns false.

writeString Member

`bool writeString(const string str, int enc=ENCODING_ASCII)`
Writes the string *str* with the *enc* encoding to the FILE. Returns true if successful, otherwise returns false.

TFILE2 Object

This object defined in any script. The same as TFILE object.

TGPIO Object

This object defined only with connected GPIO socket.
Access to members: GUI`.someFunc();`

power Member

`bool power(int on, int mv=-1)`

clear Member

`bool clear(uint pins)`
Clears selected GPIO pins according bits in the *pins* variable. Output bits can be cleared by writing ones. Returns true if successful, otherwise returns false.

Example:
```
GPIO.clear(0x800001); // Set IO0 and IO23 low
```

dir Member

bool dir(uint value)
Configures the GPIO pins as inputs or outputs by bits in the *value* variable. Pins can be configured as outputs by writing ones. Returns true if successful, otherwise returns false.

Example:
```
GPIO.dir(0x1); // Config IO0 as output, IO1-IO23 as inputs
```

dirIn Member

bool dirIn(uint pins)
Configures selected GPIO pins as inputs according bits in the *pins* variable. Pins will be configured as inputs by writing ones. Returns true if successful, otherwise returns false.

Example:
```
GPIO.dirIn(0x81); // Config IO0 and IO7 as inputs
```

dirOut Member

bool dirOut(uint pins)
Configures selected GPIO pins as outputs according bits in the *pins* variable. Pins will be configured as outputs by writing ones. Returns true if successful, otherwise returns false.

Example:
```
GPIO.dirOut(0xFF00); // Config IO8-IO15 as outputs
```

framesAdd Member

bool framesAdd(int frame, uint value=0, uint value2=0)
Adds new frame data into the frame buffer. Returns true if successful, otherwise returns false.

Example:
```
GPIO.framesClear(); // Clear all frames
GPIO.framesAdd(GPIO_WRITE, 0x55); // Write 0x55 to IO0-7
GPIO.framesAdd(GPIO_DELAY_US, 1000); // 1000us delay
GPIO.framesAdd(GPIO_WRITE, 0xAA); // Write 0xAA to IO0-7
GPIO.framesAdd(GPIO_WAIT_FALL, 8, 500); // Wait for falling edge on IO8
GPIO.framesAdd(GPIO_READ); // Read #1
GPIO.framesAdd(GPIO_WAIT_FALL, 8, 500);
GPIO.framesAdd(GPIO_READ); // Read #2
if (!framesSend())
{
     // handle error
     return;
}
uint read1= framesAnswer(); // Get read #1 value
uint read2= framesAnswer(1); // Get read #2 value
```

Enumerated frame values:
```
GPIO_DIR
GPIO_IN
```

```
GPIO_OUT
GPIO_CLEAR
GPIO_SET
GPIO_READ
GPIO_WRITE
GPIO_MASK
GPIO_MREAD
GPIO_MWRITE
GPIO_NOP
GPIO_DELAY_US
GPIO_WAIT_LOW
GPIO_WAIT_HIGH
GPIO_WAIT_FALL
GPIO_WAIT_RISE
```

framesAnswer Member

`uint framesAnswer(int idx=0)`
Returns received answer with the index *idx*.

framesClear Member

`void framesClear()`
Clears frames counter.

framesSend Member

`bool framesSend(int timeout=3000)`
Sends all frames and executes in a single command. Returns true if successful, otherwise returns false.

mask Member

`bool mask(uint val)`
Sets the IO pins mask for mread() and mwrite() functions. Returns true if successful, otherwise returns false.

Example:

```
uint val;
GPIO.dir(0xF000); // Set IO8-IO11 as outputs
GPIO.mask(0xFF00); // Mask IO8-IO15 pins
GPIO.mwrite(0xF000); //Set IO8-IO11 to zeros, IO12-IO15 to ones
GPIO.mread(val); // Read IO8-IO15 pins
```

mread Member

`bool mread(uint &value)`
Reads masked GPIO pins state into the *value* variable. Returns true if successful, otherwise returns false.

mwrite Member

`bool mwrite(uint val)`
Writes masked GPIO pins state using bits in the *value* variable. Returns true if successful, otherwise returns false.

read Member

`bool read(uint &value)`
Reads the GPIO pins state into the *value* variable. Returns true if successful, otherwise returns false.

set Member

`bool set(uint pins)`
Sets to one selected GPIO pins according bits in the *pins* variable. Output bits can be set by writing ones. Returns true if successful, otherwise returns false.

write Member

`bool write(uint value)`
Writes the GPIO pins state using bits in the *value* variable. Returns true if successful, otherwise returns false.

waitLow Member

`bool waitLow(int pin, int timeout=1000)`
Waits until the selected GPIO *pin* state becomes low (zero). Values from 0 to 23 are valid pin number values. Timeout value in microseconds (us). Returns true if successful, otherwise returns false.

Example:
```
bool x=GPIO.waitLow(23, 50000); // Wait for zero on IO23, timeout 50ms
```

waitHigh Member

`bool waitHigh(int pin, int timeout=1000)`
Waits until the selected GPIO *pin* state becomes high (one). Values from 0 to 23 are valid pin number values. Timeout value in microseconds (us). Returns true if successful, otherwise returns false.

waitFall Member

`bool waitFall(int pin, int timeout=1000)`
Waits for the falling edge on the selected GPIO *pin*. Values from 0 to 23 are valid pin number values. Timeout value in microseconds (us). Returns true if successful, otherwise returns false.

waitRise Member

`bool waitRise(int pin, int timeout=1000)`
Waits for the rising edge on the selected GPIO *pin*. Values from 0 to 23 are valid pin number values. Timeout value in microseconds (us). Returns true if successful, otherwise returns false.

TGUI Object

This object defined in any script.

Access to members: `GUI.someFunc();`

progressStart Member

`void progressStart(int64 size)`
Sets the progress bar's minimum value to 0 and maximum value to *size* respectively.

progressAdd Member

`void progressAdd(int64 val)`
Adds value *val* to the progress bar's current value.

progressValue Member

`void progressValue(int64 val)`
Sets the progress bar's current value to *val*. Attempting to change the current value to one outside the minimum-maximum range has no effect on the current value.

progressEnd Member

`void progressEnd()`
Resets the progress bar.

fileNameRead Member

`string fileNameRead()`
Returns value of the current "Read to:" field.

fileNameWrite Member

`string fileNameWrite()`
Returns value of the current "Write from:" field.

filePathRead Member

`string filePathRead()`
Returns an absolute path of the current "Read to:" field. This doesn't include the file name.

filePathWrite Member

`string filePathWrite()`
Returns an absolute path of the current "Write from:" field. This doesn't include the file name.

THASH Object

This object defined in any script.

`void CRC16Init(uint16 poly=0x8005, uint16 init=0, bool refin=true, bool refout=true, uint16 xorout=0)`

Initializes the CRC16 hash.

void CRC16Restart()

Restarts the CRC16 hash.

void CRC16Update8(uint8 val)

Updates a CRC16 with byte value *val*.

bool CRC16UpdateBuffer(int size, int pos=0)

Updates a CRC16 with *size* bytes from the BUFFER with index position *pos*. Returns true if successful, otherwise returns false.

uint16 CRC16Final()

Calculates a CRC16 and restarts. Returns calculated CRC16 value.

void CRC32Init(uint32 poly=0x04C11DB7,uint32 init=0xFFFFFFFF, bool refin=true, bool refout=true, uint32 xorout=0xFFFFFFFF)

Initializes the CRC32 hash.

void CRC32Restart()

Restarts the CRC32 hash.

void CRC32Update8(uint8 val)

Updates a CRC32 with byte value *val*.

bool CRC32UpdateBuffer(int size, int pos=0)

Updates a CRC32 with *size* bytes from the BUFFER with index position *pos*. Returns true if successful, otherwise returns false.

uint32 CRC32Final()

Calculates a CRC32 and restarts. Returns calculated CRC32 value.

void SHA1Restart()

Restarts the SHA-1 hash.

void SHA1Update8(uint8 val)

Updates a SHA-1 with byte value *val*.

bool SHA1UpdateBuffer(int size, int pos=0)

Updates a SHA-1 with *size* bytes from the BUFFER with index position *pos*. Returns true if successful, otherwise returns false.

string SHA1Final()

Calculates a SHA-1 and restarts. Returns string with calculated SHA-1 value.

void SHA1FinalBuffer(int pos=0)

Calculates a SHA-1 and restarts. Puts a SHA-1 hash (20 bytes) into the BUFFER with index position *pos*.

TI2C Object

This object defined only with connected I2C socket.
Access to members: I2C`.someFunc();`

Members

bool setClk(uint clk)
Sets the I2C clock to the *clk* value in Hz. Returns true if successful, otherwise returns false.

bool start()
Sets the start condition. Returns true if successful, otherwise returns false.

bool stop()
Sets the stop condition. Returns true if successful, otherwise returns false.

bool read8(uint8 &out val, uint8 &out ack)
Reads the byte and acknowledge values from the I2C bus to *val* and *ack* variables. Returns true if successful, otherwise returns false.

bool read8Ack(uint8 &out)
Reads the byte value with the set acknowledge bit from the I2C bus to *val* variable. Returns true if successful, otherwise returns false.

bool readBuffer(int size, int pos=0)
Reads *size* bytes from the I2C bus into BUFFER with index position *pos*. Returns true if successful, otherwise returns false.

bool write8(uint8 val, uint8 &out ack)
Writes the byte with *val* to the I2C bus and reads acknowledge bit to *ack*. Returns true if successful, otherwise returns false.

bool write8Ack(uint8 val)
Writes the byte with *val* to the I2C bus with the set acknowledge bit. Returns true if successful, otherwise returns false.

bool writeBuffer(int size, int pos=0)
Writes *size* bytes from the BUFFER with index position *pos* to the I2C bus. Returns true if successful, otherwise returns false.

bool ispEnter(int mode)
Puts the MCU to the ISP programming mode. Returns true if successful, otherwise returns false.

Enumerated mode values
ISP_MODE_WT61P8

ISP_MODE_WT61P802

bool ispExit(int mode=0)
Exits from the MCU ISP programming mode. Returns true if successful, otherwise returns false.

TNAND Object

This object defined only with connected NAND socket.
Access to members: NAND.`someFunc();`

Members

addr Member

bool addr(int addr)
Sends an address *addr* cycle to the NAND device, ALE signal high. Returns true if successful, otherwise returns false.

cmd Member

bool cmd(int cmd)
Sends a command *cmd* cycle to the NAND device, CLE signal high. Returns true if successful, otherwise returns false.

eraseBlock Member

bool eraseBlock(int block)
Erases a *block* in the NAND device. Returns true if successful, otherwise returns false.

pageReadBuffer

bool pageReadBuffer(int page, int col=0, int size=0, int pos=0)
Reads *size* bytes from the NAND device *page* data from the column *col* into the BUFFER with index position *pos.* Returns true if successful, otherwise returns false.

pageWriteBuffer

bool pageWriteBuffer(int page, int col=0, int size=0, int pos=0)
Writes *size* bytes from the BUFFER with index position *pos* to the NAND device *page* from the column *col*. Returns true if successful, otherwise returns false.

read8 Member

bool read8(int8 &val)
Reads the byte from the NAND device bus into the *val* variable. Returns true if successful, otherwise returns false.

read16 Member

bool read16(int16 &val)
Reads the word from the NAND device bus into the *val* variable. Returns true if successful, otherwise returns false.

read32 Member

`bool read32(int32 &val)`
Reads the dword from the NAND device bus into the *val* variable. Returns true if successful, otherwise returns false.

read64 Member

`bool read64(int64 &val)`
Reads the qword from the NAND device bus into the *val* variable. Returns true if successful, otherwise returns false.

readBuffer Member

`bool readBuffer(int size, int pos=0)`
Reads *size* bytes from the NAND device bus into the BUFFER with index position *pos.* Returns true if successful, otherwise returns false.

write8 Member

`bool write8(int8 val)`
Writes the byte with *val* to the NAND device bus. Returns true if successful, otherwise returns false.

write16 Member

`bool write16(int16 val)`
Writes the word with *val* to the NAND device bus. Returns true if successful, otherwise returns false.

write32 Member

`bool write32(int32 val)`
Writes the dword with *val* to the NAND device bus. Returns true if successful, otherwise returns false.

write64 Member

`bool write64(int64 val)`
Writes the qword with *val* to the NAND device bus. Returns true if successful, otherwise returns false.

writeBuffer Member

`bool writeBuffer(int size, int pos=0)`
Writes *size* bytes from the BUFFER with index position *pos* to the NAND device bus. Returns true if successful, otherwise returns false.

setBus Member

`bool setBus(int bus)`
Sets the IO Bus mode *bus* for the NAND device. Returns true if successful, otherwise returns false.

Enumerated NAND bus values:
NAND_BUS_X8
NAND_BUS_X16_ONFI

Example:
```
NAND.setBus(NAND_BUS_X8);
```

setCE Member

bool setCE(int ce)
Sets the current Chip Enable to *ce* for the NAND device. Returns true if successful, otherwise returns false.

setSerialAccess Member

bool setSerialAccess(int ns)
Sets the Serial Access Time to *ns* for the NAND device. Returns true if successful, otherwise returns false.

Example:
```
NAND.setSerialAccess(25); // Set Serial Access Time to 25ns
```

setWidth Member

bool setWidth(int width)
Sets the IO Bus *width* for the NAND device. Returns true if successful, otherwise returns false.

Example:
```
NAND.setWidth(16); // Set IO Bus width to 16 bit
```

TNOR Object

This object defined only with connected NOR socket.
Access to members: NOR`.someFunc();`

Members

bool read16(uint addr, uint16 &val)
Reads the word from the NOR device at the address *addr* into the *val* variable. Returns true if successful, otherwise returns false.

bool write16(uint addr, uint16 val)
Writes the word at the address *addr* with *val* to the NOR device. Returns true if successful, otherwise returns false.

TOW Object

This object defined only with connected 1-Wire socket.
Access to members: OW`.someFunc();`

Members

bool power(int on, int mv=-1)

bool reset()
Resets the 1-Wire bus. Returns true if the 1-Wire device has been found, otherwise returns false.

bool read1(uint8 &value)
Reads the bit from the 1-Wire bus into the *value* variable. Returns true if successful, otherwise returns false.

bool read8(uint8 &value)
Reads the byte from the 1-Wire bus into the *value* variable. Returns true if successful, otherwise returns false.

bool write1(uint8 value)
Writes the bit with *value* to 1-Wire bus. Returns true if successful, otherwise returns false.

bool write8(uint8 value)
Writes the byte with *value* to 1-Wire bus. Returns true if successful, otherwise returns false.

bool touch8(uint8 data, uint8 &value)
Writes the byte with *data* to 1-Wire bus and reads the byte from the 1-Wire bus into the *value* variable. Returns true if successful, otherwise returns false.

TPARTITIONS Object

This object defined in any script. Only one TPARTITIONS object with name PARTITIONS defined.
Access to members: PARTITIONS`.someFunc();`

Members

add Member

int add(string name, int64 addr=0, int64 size=0, int part=0, bool used=true)
Adds the new partition with *name*, *addr*, *size*, *part* and *used* variables values. Returns partition index if successful, otherwise returns negative value.

clear Member

void clear()
Clears all PARTITIONS data and sets PARTITIONS count to 0.

count Member

int count()
Returns count of the PARTITIONS.

flags Member

int flags()
Returns current flags value of the PARTITIONS.

Enumerated bit-masks values:
PARTS_FLAG_RAW_ADDR
PARTS_FLAG_RAW_SIZE
PARTS_FLAG_FILE_POS_EQU_ADDR

Example:
```
if (PARTITIONS.flags() & PARTS_FLAG_FILE_POS_EQU_ADDR)
{
     someFunc(...);
}
```

partAddr Member

`int64 partAddr(int idx)`
Returns the address value of the partition with the index *idx*. Returns negative (-1) value if partition not exits.

partFileName Member

`string partFileName(int idx)`
Returns a file name of the partition with the index *idx*.

partFilePos Member

`int64 partFilePos(int idx)`
Returns a file pointer value of the partition with the index *idx*.

partFS Member

`int partFS(int idx)`
Returns a File System type value of the partition with the index *idx*. Returns FS_UNKNOWN value if File System type not set or unknown.

Enumerated FS type values:
```
FS_UNKNOWN
FS_EXT4
FS_FAT16
FS_FAT32
FS_NTFS
FS_SQUASHFS
```

partName Member

`string partName(int idx)`
Returns a name of the partition with the index *idx*.

partSize Member

`int64 partSize(int idx)`
Returns the Size value of the partition with the index *idx*. Returns negative (-1) value if partition not exits.

partUsed Member

`bool partUsed(int idx)`
Returns the Used value of the partition with the index *idx*.

remove Member

`bool remove(int idx)`
Removes partition with the index *idx*. Returns true if successful, otherwise returns false.

saveToUDEV Member

`bool saveToUDEV(const string fileName)`
Saves the partitions data to the UDEV (UFPI Device) file with the filename *fileName*. Returns true if successful, otherwise returns false.

size Member

`int64 size()`
Returns size of the all PARTITIONS.

sizeUsed Member

`int64 sizeUsed()`
Returns size of the all used (checked) PARTITIONS.

setFlags Member

`void setFlags(int val)`
Sets the flags bits to *val*. See *flags()*.

setPartAddr Member

`bool setPartAddr(int idx, int64 val)`
Sets the partition address value to *val* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

setPartFileName Member

`bool setPartFileName(int idx, const string fileName)`
Sets a partition file name to *fileName* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

setPartFilePos Member

`bool setPartFilePos(int idx, int64 val)`
Sets the partition file offset value to *val* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

setPartFS Member

`bool setPartFS(int idx, int val)`
Sets the partition File System type to *val* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

setPartName Member

bool setPartName(int idx, const string name)
Sets the partition Name value to *val* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

setPartSize Member

bool setPartSize(int idx, int64 val)
Sets the partition Size value to *val* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

setPartUsed Member

bool setPartUsed(int idx, bool val)
Sets the partition Used value to *val* for the partition with the index *idx*. Returns true if successful, otherwise returns false.

TSDMMC Object

This object defined only with connected SD/eMMC socket.
Access to members: SDMMC`.someFunc();`

Members

int clk()
Returns current clock value in Hz.

bool cmd(uint32 cmd, uint32 arg=0)
Sends the command *cmd* with argument *arg* to the SD/eMMC device. Returns true if successful, otherwise returns false.

bool cmdApp(uint32 cmd, uint32 arg=0)
Sends the application-specific command *cmd* with argument *arg* to the SD/eMMC device. Returns true if successful, otherwise returns false.

bool cmdData(uint32 cmd, uint32 arg, uint size)
Sends the command *cmd* with argument *arg* to the SD/eMMC device and sets controller Byte Count Register value to size. Number of bytes to be transferred should be integer multiple of Block Size for block transfers. For undefined number of byte transfers, byte count should be set to 0. When byte count is set to 0, it is responsibility of host to explicitly send stop/abort command to terminate data transfer. Returns true if successful, otherwise returns false.

bool cmdSwitch(uint32 arg=0)
Sends the SWITCH Command (CMD6) with argument *arg* to the SD/eMMC device and checks status with the SEND_STATUS Command (CMD13). Returns true if SWITCH Command successful, otherwise returns false.

bool fifoReadBuffer(int size, int pos=0)
Reads *size* bytes from the SD/eMMC controller FIFO into the BUFFER with index position *pos.* Returns true if successful, otherwise returns false.

bool fifoWriteBuffer(int size, int pos=0)
Writes *size* bytes from the BUFFER with index position *pos* to the SD/eMMC controller. Returns true if

successful, otherwise returns false.

bool fifoReset()
Resets FIFO of the SD/eMMC controller. Returns true if successful, otherwise returns false.

bool power(uint on)
Turns ON/OFF power of the SD/eMMC socket. Returns true if successful, otherwise returns false.

uint32 response(int idx=0)
Returns the response received for the previously sent SD/eMMC command.

bool setBus(uint bus)
Sets the bus of the SD/eMMC controller to the *bus* value and sets the EXT_CSD BUS_WIDTH byte with the corresponding value. Only 1 and 4 are valid *bus* values. Returns true if successful, otherwise returns false.

bool setClk(uint clk)
Sets the SD/eMMC clock to the *clk* value in Hz. Returns true if successful, otherwise returns false.

bool setMedia(uint value)
Sets the current SD/eMMC media type to the *value*. Returns true if successful, otherwise returns false.

Enumerated media type values:
`MEDIA_SDC`
`MEDIA_MMC`

bool waitBusy(int ms=-1)
Waits in milliseconds *ms* until SD/eMMC card data busy signal active. Returns true if successful, otherwise returns false.

TSPIFI Object

This object defined only with connected SPIFI (SPI Flash Interface) socket.
Access to members: SPIFI`.someFunc();`

Members

bool setClk(uint clk)
Sets the SPI clock to the *clk* value in Hz. Returns true if successful, otherwise returns false.

bool cmd(uint32 cmd, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Sends the command *cmd* with or without *addr* and *dummy* values to the SPI bus. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

Example:

```
cmd(0x06); // Send Write Enable command
cmd(0x01, 0, 1); // Write 0 to the Status register
```

bool cmdRead8(uint32 cmd, uint8 &out val, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Reads the byte from the SPI bus into the *val* variable using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt*

value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdRead16(uint32 cmd, uint16 &out, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Reads the word from the SPI bus into the *val* variable using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdRead32(uint32 cmd, uint32 &out, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Reads the dword from the SPI bus into the *val* variable using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdRead64(uint32 cmd, uint64 &out, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Reads the qword from the SPI bus into the *val* variable using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdReadBuff(uint32 cmd, int size, int pos=0, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Reads *size* bytes from the SPI bus into BUFFER with index position *pos* using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

Example:
`cmdReadBuff(0x9F, 8);` // Read 8 bytes to the BUFFER with JEDEC ID instruction

bool cmdWrite8(uint32 cmd, uint8 val, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Writes the byte *val* to the SPI bus using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdWrite16(uint32 cmd, uint16 val, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Writes the word *val* to the SPI bus using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdWrite32(uint32 cmd, uint32 val, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Writes the dword *val* to the SPI bus using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdWrite64(uint32 cmd, uint64 val, uint32 addr=0, int acnt=0, uint32 dummy=0, int dcnt=0)
Writes the qword *val* to the SPI bus using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

bool cmdWriteBuff(uint32 cmd, int size, int pos=0, uint32 addr=0, int

acnt=0, uint32 dummy=0, int dcnt=0)
Writes *size* bytes from the BUFFER with index position *pos* to SPI bus using the command *cmd* with or without *addr* and *dummy* values. Address value *addr* will be sent in MSB order. *acnt* value defines number of the address bytes. *dcnt* value defines number of the dummy bytes. Returns true if successful, otherwise returns false.

TUART Object

This object defined only with connected UART socket.
Access to members: UART`.someFunc();`

Members

power Member

bool power(int on, int mv=-1)

gpioClear Member

bool gpioClear(uint8 value)
Clears selected UART GPIO pins according bits in the *value* variable. Output bits can be cleared by writing ones. Returns true if successful, otherwise returns false.

Example:
```
UART.gpioClear(0x41); // Clear GP6 and GP0 pins
```

gpioDir Member

bool gpioDir(uint8 value)
Configures the UART GPIO pins as inputs or outputs by bits in the *value* variable. Pins can be configured as outputs by writing ones. Returns true if successful, otherwise returns false.

Example:
```
UART.gpioDir(0x41); // Config GP6, GP0 pins as outputs, others as inputs
```

gpioDirState Member

bool gpioDirState(uint8 &value)
Reads the UART GPIO pins direction state into the *value* variable. Returns true if successful, otherwise returns false.

gpioPins Member

bool gpioPins(uint8 &value)
Reads the UART GPIO pins state into the *value* variable. Returns true if successful, otherwise returns false.

Example:
```
uint8 pins;

if (!UART.gpioPins(pins))
```

```
{
     // handle error
     return -1;
}

if ((pins & 0x02)==0)
{
     // GP1 pin input = 0
}
```

gpioPulse Member

bool gpioPulse(uint8 pins, int duration)
Sends negative or positive pulse with the duration in microseconds (us). Sends negative pulse if state of the selected pins are not equal zero. Returns true if successful, otherwise returns false.

gpioSet Member

bool gpioSet(uint8 value)
Sets to one selected UART GPIO pins according bits in the *value* variable. Output bits can be set by writing ones. Returns true if successful, otherwise returns false.

lineStatus Member

bool lineStatus(uint8 &value)
Reads the UART Line Status Register into the *value* variable. Returns true if successful, otherwise returns false.

Line Status Register bits Enums:
UART_LSR_RX_DATA_READY
UART_LSR_RX_OVERRUN_ERROR
UART_LSR_RX_PARITY_ERROR
UART_LSR_RX_FRAMING_ERROR
UART_LSR_BREAK_INTERRUPT
UART_LSR_TX_REGISTER_EMPTY
UART_LSR_TRANSMITTER_EMPTY
UART_LSR_RX_FIFO_ERROR

modemControl Member

bool modemControl(uint8 &value)
Reads the UART Modem Control Register into the *value* variable. Returns true if successful, otherwise returns false.

Modem Control Register bits Enums:
UART_MCR_DTR_CTRL
UART_MCR_RTS_CTRL
UART_MCR_LOOPBACK_MODE
UART_MCR_RTS_ENABLE
UART_MCR_CTS_ENABLE

modemStatus Member

bool modemStatus(uint8 &value)
Reads the UART Modem Status Register into the *value* variable. Returns true if successful, otherwise returns false.

Modem Status Register bits Enums:
```
UART_MSR_CTS_CHANGE
UART_MSR_DSR_CHANGE
UART_MSR_RI_CHANGE
UART_MSR_DCD_CHANGE
UART_MSR_CTS_STATE
UART_MSR_DSR_STATE
UART_MSR_RI_STATE
UART_MSR_DCD_STATE
```

rx8 Member

bool rx8(uint8 &value)
Reads the byte from the UART RX Ring buffer the *value* variable. Returns true if successful, otherwise returns false.

rxBuffer Member

bool rxBuffer(int size, int pos=0)
Reads *size* bytes from the UART RX Ring buffer into BUFFER with index position *pos.* Returns true if successful, otherwise returns false.

rxClear Member

void rxClear()
Clears the UART RX Ring buffer.

rxCount Member

int rxCount()
Returns the number of bytes received in the UART RX Ring buffer.

Example:
```
int len;

BUFFER.fill(0xFF, UART.rxSize()); // fill Buffer and resize

rx = UART.rxCount();

if (rx > 0)
{
     UART.rxBuffer(rx);

     BUFFER.dump(rx);
}
```

rxSize Member

`int rxSize()`
Returns the size of the UART RX Ring buffer.

setBaud Member

`bool setBaud(uint value)`
Sets the UART baud rate to the *value* variable. Returns true if successful, otherwise returns false.

setBits Member

`bool setBits(uint8 value)`
Sets the UART character length to the *value* variable. Values from 5 to 8 are valid character length values. Returns true if successful, otherwise returns false.

setFlow Member

`bool setFlow(uint8 value)`
Sets the UART flow control to the *value* variable. Returns true if successful, otherwise returns false.

setModemControl Member

`bool setModemControl(uint8 value)`
Sets the UART Modem Control Register to the *value* variable. Returns true if successful, otherwise returns false.

setParity Member

`bool setParity(uint8 value)`
Sets the UART Parity to *value*. Returns true if successful, otherwise returns false.

Defined Parity values:

```
UART_PARITY_NONE
UART_PARITY_EVEN
UART_PARITY_ODD
```

Example:

```
setParity(UART_PARITY_NONE);
```

setRxDelay Member

`bool setRxDelay(uint value)`
Sets RX delay counter to *value*.Values from 0 to 65535 are valid RX delay counter values. Returns true if successful, otherwise returns false.

setStop Member

`bool setStop(uint8 value)`
Sets the UART Stop Bit to *value*. Only 1 (1 stop bit) and 2 (2 stop bits) are valid Stop Bite values. Returns true if successful, otherwise returns false.

setTxPin Member

`bool setTxPin(uint value)`
Sets the UART TX pin state to *value*. Returns true if successful, otherwise returns false.

Defined state values:

```
UART_TX_PIN_LOW
UART_TX_PIN_HIGH
UART_TX_PIN_INPUT
UART_TX_PIN_TX
```

tx8 Member

`bool tx8(uint8 value)`
Writes the byte with *value* to the UART Transmit Holding Register. Returns true if successful, otherwise returns false.

txBuffer Member

`bool txBuffer(int size, int pos=0)`
Writes *size* bytes from the BUFFER with index position *pos* to the UART Transmit Holding Register. Returns true if successful, otherwise returns false.