MechCommander Gold Campaign Builder's Guide

Part 2 – ABL Scripting Reference

Revision 0.3

By Cmunsta

Revision History

Rev 0.3
- Updated the function descriptions.
- Added various table of contents.
- Added quick links between sections.
- Added alphabetical and category function lists.
- Added links to functions and cross-references.
- Fixed some typos.

Rev 0.2
- Added description (very basic) of brain cells and how they work.
- Added information on the paths for the #include and #include_ directives
- Added section about discovered include file peculiarities.
- Added information about variable scope and lifetime.
- Added section about short-circuited logic, the differences with non short-circuited logic (as used in ABL) and the effect it can have on scripting.
- Added information on creating and using libraries.
- Added information on how modules work.
- Added information on the relationship between modules, libraries and functions.
- Added a proper beginning to the document.
- Added discovery of the ability to pass information from one mission to another within a campaign.
- Updated all built-in function definitions with information gleaned from the MechCommander 2 source and cross-referenced back with the game's executable.
- More proofreading.

Rev Preliminary Release 0.1
- Early edition. Internal functions are listed and defined as much as possible at this stage.
- Very quick proofreading done. (I'm sure many errors still exist).

Rev Pre-Release 0.1
- **Very** early edition of the document. Not all functions are defined.

Table of Contents

Introduction

This document is intended to be a reference guide to the MechCommander Gold ABL scripting language. But as the language must be worked out from the official scripts of the game as well as reverse engineering the game executable, this can be both difficult and time consuming.

Therefore, this reference should be considered a living document. That is, as new discoveries and functionality is discovered, the document will be updated to reflect these additions and changes.

As this is a reference guide about the ABL script language and its behavior, this document will not teach script programming itself. To learn that, there are many tutorials available on the internet and elsewhere.

It is hoped however, that many examples, samples and more can be added as this document grows.

Things To Do

This is just a reminder of some of the things that still need to be done before we can even consider this document relatively complete.

- Test code and functions, example code.
- Using the built-in ABL debugger
- Defined library functions, vars and constants

Discovered ABL Information

- There are maximums for the number of modules and the number of libraries that can be loaded. (unknown what these are though)
- The function search of a module seems to search ALL modules currently loaded.
- Seem to be able to reuse same name if have different parameter list (have to test this though). This implies function overloading.

Basic ABL Syntax and Workings

This section is a description of the basic syntax of the ABL language and how some of its workings. It is suggested that it be read through even if not all of it is clear as it will help in understanding later sections of this document.

There is a lot of information that needs to be presented in this section in order to better understand the sections that follow so bear with it. It'll all be worth while in the end.

The keywords are shown in **bold** text, optional parts are shown in square brackets, and parts that are either chosen by the user, or have user choices are shown in braces with *italic* text.

Case In-Sensitivity

The ABL scripting language is not case sensitive. This means that

AFunctionThatDoesStuff
afunctionthatdoesstuff
AFUNCTIONTHATDOESSTUFF
aFuNcTiOnThAtDoEsStUfF

all refer to the exact same thing.

Parser Directives

Directives are commands that are passed directly to the language parser and not the language interpreter. These can turn certain features on or off, but the most important is the #include directive. This allows us to include a different file as if we typed it in directly.

#include

Include an .ABI file into the current file. The #include directive looks for the given file starting in the game's root folder.

#include_

Include an .ABI file into the current file. The #include_ directive looks for the given file starting in the {MCX}\data\missions folder

#stringfuncs_on
#stringfuncs_off

Actual use unknown, but obviously has something to do with tuning on/off string functions. Likely affects the ABL debugger output.

#print_on
#print_off

Turns printing functions on/off. Likely for use with the ABL debugger. #print_on has been seen in one of the standard ABL files in the game.

#assert_on
#assert_off

Turns asserts on/off. This is a debug feature only, and likely for use with the ABL debugger. Asserts are used in debugging to verify certain values or conditions. If the verification fails, the assert forces the program to terminate. With a debugger, it is possible to catch these asserts and show the culprit that would cause the error.

Pre-defined Constants

The ABL language has a few constants already defined.

TRUE
FALSE

These two constants are defined for boolean variables. TRUE is defined as the value 1, and FALSE as the value 0.

Comments

Comments are notes left in the code to remind the programmer, or anyone else reading the program, what certain things do. It is considered good programming practice to comment your programs. This becomes even more important when you consider you may try editing or changing the program at a much later date when you have likely forgotten the details of how a piece of code works, or what it was trying to do and why.

The ABL language uses the same comment style as C and C++.

//

The double-slash begins a single line comment. All text after the double-slash up until the end of the line is considered a comment and is ignored by the language parser.

/*
*/

Block comments. The slash-star begins the comment block and the star-slash terminates the comment block. All text in between these symbols are considered comments and are ignored by the ABL parser.

Variable Types

There are four basic variable types.

boolean

Can only hold true or false values. (internally, the game stores these in byte size memory chunks).

integer

Holds a 32-bit signed integer value. The ABL language does not appear to have an unsigned modifier.

real

Holds a real or floating point value. Though it appears the game uses 32-bits to store these values, it does use the floating point processor for at least some of the arithmetic calculations.

char

Holds a single character. This type is also used for strings (built as an array of characters). It is uncertain if a char can be assigned to an integer or vice versa.

Array Declaration

To create an array (or list) of variables of a given type, we set the dimension (or size) of the array in square brackets *after* the variable type. For multi-dimension arrays, each dimension should be separated by commas between the square brackets.

Examples:
integer[9] tallyList**;**

This declaration will make the variable tallyList an array of 9 integers.

real[3,4] TwoDimArray**;**

This declaration will make the variable TwoDimArray into a two dimensional array of 3 rows of 4 columns of reals (12 reals in all). Note that it is not yet determined which would in fact be the rows, and which the columns, but this is somewhat academic.

Variable Modifiers

There are two types of variable modifiers.

static

The static modifier allow a variable to retain its value across multiple execution calls of a function, module or library.

eternal

The eternal modifier creates the variable in a global memory space. Only one such variable declaration is allowed since they are globally defined. Attempting to declare a second eternal variable of the same name will result in an error.

Variable Scopes and Lifetimes

The variable modifiers also dictate scope usage of a variable.

Variables without any modifiers (that is, a variable defined normally) will only be defined locally to the function, library or module that defines it. It also seems that such a variable cannot be used in sub-functions etc. Further, the variable will not retain its value when the function, library or module ends execution. Care does need to be done with these variables as the name will be allowed in sub-functions, but their use will cause the script to halt execution with an error as the script interpreter will attempt to access memory that is invalid within this new scope.

Variables with the static modifier are stored in their own memory space and so do not have the problem that normally declared variables do (as described above). This means that though a static variable will have the same localized scope as a normal variable, it will still be usable inside sub-functions because the variable itself is stored in its own memory space and so the interpreter is capable of accessing this memory correctly. This memory space also allows the variable to retain its value across function, library or module executions.

Variables with the eternal modifier are declared in a global memory space. This means that not only will the variable be accessible throughout all functions, libraries and modules, but that only one such variable declaration may exist in all functions, libraries and modules.

All variables, regardless of modifier type, are completely destroyed at mission end. This means that variables cannot retain values across missions within a campaign.

All variables appear to be initialized to 0 on creation.

User Defined Types

We can define our own variable types. User types can also have array sizes, this way we can define vectors and the like as a single type. It is not yet known if we can mix types within the same user type, or if we can use user types in declaring other user types..

{new type name} **=** *{base type}* [**[***{size}* [**,** *{size}*] **]**] **;**

Examples:

```
position = real[3];
path = real[30,2];
IntList = integer[255];
```

Arithmetic operators

No language would be very useful without some basic arithmetic operators, and ABL is no exception. These are the known operators.

+	Addition
-	Subtraction
*	Multiplication
/	Division
=	Assignment
mod	Modulus

Logic Operators

The ABL language defines certain operators for logical comparisons. All logical comparisons will result in a true or false value. These are the known logical operators.

<>	Not equal
>=	Greater or equal
<=	Less than or equal
==	Equal
>	Greater than
<	Less than
not	Logical inverse
and	Logical AND
or	Logical OR

Non Short-Circuited Logic

Short-circuited logic is a method by which logical operations are not completely evaluated if the result of the operation can be determined in advance. For instance, C and C++ uses short-circuited logic. When these languages are evaluating an OR statement, if the first parameter of the OR statement is true, then the second parameter need not be evaluated since the truth table of the logical OR dictates that the end-result will also be true regardless of the value of the second parameter. This fact is used by C and C++ programmers to purposely skip certain evaluations (a trick often used in error checking).

The ABL scripting language however, does not use short-circuited logic. This means that all parts of a logical statement are evaluated completely before performing the logical operation itself.

As the concept of short-circuited logic versus non short-circuited logic can affect how we write our scripts, here is an example (granted constructed to demonstrate the concept) that illustrates the difference.

Given the following function (the @ symbol in the parameters means that a value will be returned through that parameter):

```
function ChangeClip(@integer Clip) : boolean;
code
    Clip = 11;
    return (true);
endfunction;
```

Execution of the following code fragment will play different audio clips depending if short-circuited logic is present or not.

```
BettyClip = 5;

if ( false and ChangeClip(BettyClip)) then
    // some code here that isn't really important
    // as it never actually gets used
endif;

PlayBetty(BettyClip);
```

The PlayBetty function shown will play clip 5 (“New leader Selected”) if short-circuited logic is present, and play clip 11 (“Mission Failed”) if it is not.

The key to understanding this is in the if condition statement. First, is a constant value of false, while the second part of the AND is a function call that will change the value of BettyClip to 11 and return the boolean value true.

The AND statement in the 'if' works as follows (here, 'a' and 'b' are boolean values):

a AND b = result

We can create a so-called truth table for the AND statement by showing all possible values for 'a' and 'b' and the result value like this:

a	b	result
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

As can be seen, only if both 'a' and 'b' are true will the final result also be true.

The if condition statement of our example, as mentioned, has the 'a' part of the AND set to false. Since according to the above truth table having a value of false for 'a' always gives a result of false no matter the value of 'b', the 'b' part need not actually be evaluated as the result is already known.

So in a language with short-circuited logic, the call to the ChangeClip function never actually gets done. So the BettyClip variable will remain at 5.

However, in ABL scripting, all parts of the AND statement are always evaluated, so the function will get called and therefore, the value of BettyClip will be changed to 11. The end result is different than if short-circuited logic was being used.

This is an important thing to realize, especially if coming from a language that does used short-circuited logic such as C or C++, as the result you would expect will be different.

Operator Precedence

The operator precedence is not yet currently known, but we can usually safely assume that it follows that of most languages and arithmetic itself. If in doubt however, the use of parenthesis can enforce the desired precedence.

( ) Force statements inside the parentheses to be done first.

If-Else

Any language worth its salt will need to execute different blocks of code based on various conditions. The if-else statement allows this functionality.

if (*{condition}* **) then**
    // if code block
[**else**
    // else code block]
endif;

Please note that the “elsif” keyword appears in the ABL parser's keyword list, but does not work. Therefore, only if and if-else type statements can be used.

Switch

The switch statement acts much like the C and C++ versions (that is, like an overgrown if-else statement), but is somewhat diminished in ABL. There is no default case (as far as I can tell) so values that do not have a case associated with them cannot be caught by the switch statement itself and therefore will still need an if statement to catch these conditions.

All case statement blocks must end with an encase; statement. Therefore, cases cannot fall through to the next code block as in the C and C++ versions.

switch (*{variable}* **)**
    case *{constant}* **:**
        // case code block
    endcase;

[*{additional case statement blocks}*]

endswitch;

While Loops

Very often, we need to repeat an operation multiple times. Loops are the tools that allow us to do this. The ABL language has three types of loops. The first of these is the while loop.

while (*{condition}* **) do**
    // while code block
endwhile;

The condition of the while loop is always checked first. In this way, the code bock inside the loop can be completely skipped if the condition is false right from the start.

Repeat-Until Loops

The second type of loop we can use is the repeat-until loop. This works much like the while loop except that the condition is only checked at the end of the loop's code block meaning that the loop's code block will always be executed at least once.

repeat
    // repeat code block
until (*{condition}* **);**

For Loops

The third type of loop that the ABL language gives us if the for loop. These types of loops are very useful when we wish to iterate a variable over a range of values.

for *{variable}* = *{start value}* **to** *{end value}* **do**
    // for code block
endfor;

Maximum Loop Iterations

The ABL language has a built-in feature to stop run-away loops (that is, loops that never end). This is required as the script code is only meant to execute in order to update the AI of a unit or check the condition of the mission, then exit. If a loop is taking too long, something is obviously wrong. Therefore, the game actually counts the number of loop iterations and will force an error to happen if the iteration count reaches this maximum (default is 100001). See the built-in function SetMaxLoops for a way to change this number. However, it is not recommended that it be changed.

Return

Functions and modules can return a value to the calling code. This can be achieved by using the return statement. Note that the parentheses are required here.

return(*{value}* **);**

Functions

Functions in ABL are declared in the following format.

function *{funcname}* [**(** *{parameter list}* **)**] [**:** *{return value type}*]**;**
    [**var**
        {list of local variable declarations}]
    code
        // code goes here
endfunction;

Note: the language also defines the endcode and endvar keywords, but they appear to be optional as they are never seen in the official campaign scripts.

Some examples:

function FuncWithNoParams**;**
    code
        // function code goes here
endfunction;

This function has no parameters to pass to it, it doesn't return a value, and declares no local variables. To call this function from another piece of code, you would simply write:
    FuncWithNoParams;

function TwoParams**(integer** First, **real** Second**);**
    var
        integer AnInt**;**
        boolean aBool**;**

    code
        // some code
endfunction;

This function has two parameters. An integer called First, and a real called Second. It declares two local variables for use in the function. The first is an integer called AnInt and the second is a boolean called aBool. This function does not return a value. To call this function, you could write:
    TwoParams(10, 3.1415926);

```
function OneParamAndRetVal(integer Something) : boolean
        code
                // code goes here
endfunction;
```

This function has a single parameter called Something that is an integer, and will return a boolean type value to the calling function (this requires the use of the return statement). You could call it like this:

BoolRetVal = OneParamAndRetVal(42);

Pointer-Like Functionality for Function Parameters

Sometimes a function must be able to return multiple values to the code that called it. This can be achieved with the following function parameter modifier.

@

The @ symbol placed before the type and variable of a function parameter seems to act like a C++ reference call. That is, instead of the value being copied and passed into the function, the function will access the original variable as if it was local to that function, though calling it by the name in the function parameter list. The @ symbol only needs to be used in the declaration of the function parameters and need not be used in the code calling the function.

Example:
function SomeFunc**(integer** First**, @integer** Second**);**

The parameter Second would access the same variable memory that the calling code passed down.

Libraries

Libraries are what the game uses to allow extra functionality within the game scripts. We can look at them as a package of functions that we can use. Libraries are loaded on a per-mission basis. The libraries to load are listed in each mission map's .FIT file under the [ABLibraries] header. There is a maximum number of libraries that may be loaded, though it is unknown what that number is.

library *{library name}***;**
[**const**
{constant list}]

[**type**
{type list}]

[**var**
{variable list}]

{function declarations}

code
// main library code
endlibrary.

Note the period at the end of endlibrary instead of a semi-colon.

Libraries also need a function defined in them called init. This function is called once by the game when the library is first loaded and run. This allows a method for initializing variables for library use.

Calling Library Functions

Calling library functions from other code can be done in the following way. Though there are still questions about how this works as it seems that we don't always need to follow this method.

[*{variable}* =] *{libname}*.*{libfuncname}*[**(** *{parameters}* **)**]**;**

Modules

Modules are the base code that the game loads and executes for the mission etc. There is a maximum number of modules that may be loaded, though it is unknown what the value is at the moment.

module *{module name}* [**:** *{return type}*]**;**
    const
        // constant list

    type
        // type list

    var
        // variable list

    {local function declarations}

    code
        // code block
endmodule.

Again, note that the endmodule keyword ends with a period and not a semi-colon.

Modules, like libraries, also need a function called init, and is used in the same way. That is, the init function is called only once, automatically by the game. This is used to initialize variables for the module.

Relationship of Modules, Libraries and Functions

Modules are what the game loads and uses for its AI use. There is a module defined for running the logic for each mission, as well as a module defined for each and every unit within a mission.

Functions are code segments that can be defined inside a module or library to help in the creation of the AI code. Functions allow us to re-use sections of code so that we don't need to re-type all the details every time we want this functionality. Function are never defined as stand alone. They are part of a library or module.

Libraries are somewhat like modules, though we can view them more like a collection of functions that are loaded at mission startup. This allows us to define sets of functions that are useful in many situations.

Both libraries and modules can have a local function named init that will be called automatically by the game when the module or library is first loaded and can be used to initialize variables for use by the module or function.

Scripting Modules

Modules are the foundation on which all the scripts are made. Modules are what are loaded and executed by the game in order to perform the required tasks for a mission or creating the AI of a unit. Clearly, a good understanding of the fundamentals of a module is important.

Thankfully, the mission editor creates a base module script for each map we create. Though this does mean that we don't have to do everything from scratch, it also means that we should examine what the editor will create for us in order to better understand things.

So to begin, lets create a simple solo mission and examine the code that it generates. To follow along, I suggest creating a mission as I describe here or things may get confusing later.

So start the mission editor and create a new solo mission. I started with a Port Arthur map, with grass, and all sliders set to the far left (the default settings).

Start by placing a single drop zone. I placed mine around x = 1725.6, y = -693.5 (though you don't have to be that precise with your drop zone).

Now place a Swiftwind on the map. I placed mine around x = 1352.9, y = -335.4.

Save the mission. I called mine “guide”.

Exit the editor and try the mission to make sure it works. Once satisfied it does (not really hard to test now is it), exit the game and open a text editor.

In the text editor, open the mission's main module by opening the .ABL file which has the same name as the mission. In this case, “guide.abl” in the {MCX}\data\missions folder.

The first line of code (after an initial comment line) is the definition for the module. Note how this module returns an integer. The game will use this returned value to determine the state of the mission.

Line 15 is the start of the definition for this module's constants. These are values that act like variables, but whose values cannot be changed. As they are being defined inside the module, their scope is limited to this module. Note how in this case all the constants that are being defined are actually held in another file, but are loaded here via the #include_ parser directive. This line tells the parser to process the contents of the file “misconst.abi”, and that this file can be found in the data\missions folder (this is the default folder for the #include_ directive). The misconst.abi file is actually held in mission.fst, but the parser scans these as well, so the file will be found.

At line 25 is the start of the user defined variable types section for this module. As there are none, this section is empty.

Starting at line 33 is the declaration of the module's variables. Note again the use of the #include_ directive to include extra variables that the editor created but placed in an external file.

Note how some of these variables are declared as static, others as eternal, and still more with no variable modifiers at all.

Lines 70 to 98 define the init function for this module. This is a special function that the game will call automatically when the module is first loaded. It is used to set initial values for variables. Note how it also uses a #include_ directive to add in some editor created variables.

Starting at line 105 and continuing to the end of the file is the actual code for this module. This is what the game will actually execute periodically. Note how there is also some code added by the editor via more #include_ directives (lines 132 and 137).

The comments for each section should be fairly clear as to what each part of this code is doing.

Starting at line 204 is an if statement that determines if the mission is over based on checks it did previously in the code. Within this if, it sets a variable called ScenarioResult.

Lastly, the last line of code (right before the endmodule line) is how the module returns the result to the game.

Lets try something here. Immediately before the return statement, lets reset this result variable to the value PLAYING. Add this line immediately before the return statement.

```
ScenarioResult = PLAYING;
```

Save the file and try the mission again. Note how the mission simply never ends now but the victory music still plays. The only way out is to either abort the mission or end the program.

This goes to show how much power this script really has over the mission. Without the script telling the game that the mission should end, the mission never will end. The game itself doesn't really know or care what the victory and/or defeat condition of a mission are, it merely runs the scripts until it is told to do otherwise.

You can go remove that line now. That was just a demonstration.

Using Built-in Functions

Though the module's script code is powerful, for it to be meaningful requires access to information that the game uses. However, the script doesn't have direct access to this data. But thankfully, there are a number of functions built into the language that are always available. They act sort of like a library of functions, but are in fact built right into the game's executable. Indeed, most of this document is merely a description of these built-in functions.

So as a simple example of how we can use these functions, we'll modify the code so that Betty (the female voice that says if the mission is complete etc) periodically plays one of her audio clips.

The built-in function we need to use is called PlayBetty (you can look this function up in the function list of this document). It requires a single integer parameter telling the function which audio clip to play. The table of valid values is shown in that function's description which we can see goes from 0 to 37 inclusive.

As we want to change which audio clip is played each time, we are going to need a variable to hold the clip number. Also, we are going to need to allow some time to pass between each call so that we can hear each clip, so we will also need to handle this.

So first, add a couple of variables to the module's variable block. We can add them starting at line 64. So add the following two variables.

```
static integer BettyClip;
static real ClipPlayTime;
```

The BettyClip variable will be used to say which audio clip we should play, and the ClipPlayTime variable will say when to start playing the clip.

Note how we use the static variable modifier here. This is needed as we want to keep the variable's value between calls to this module. Without this, the variable would simply be reset to 0 each iteration.

Now we need to set the initial values of these variables. We do this inside the init function for this module. Add these lines after the #include_ “guide_INIT.ABI” line.

```
BettyClip = 0;
ClipPlayTime = 7.0;
```

The initial value for the ClipPlayTime variable is to say we will start playing the audio clip after 7 seconds.

Now for actually making the code.

Around line 115 (depends how many blank lines you added above), you will find the first 'if' statement. This statement is actually keeping track of the passage of time in the game. As we want to also keep track of some time passage for playing our audio clips, we can do something similar. So immediately after this 'if' statement (around line 125 of so) add in the following after the “endif;” statement:

```
if (ClipPlayTime <= nextsecond) then
     ClipPlayTime = gametime + 7.0;

     PlayBetty(BettyClip);

     BettyClip = BettyClip + 1;
     if (BettyClip > 37) then
          BettyClip = 0;
     endif;
endif;
```

The first line is checking if the correct number of seconds have gone by (the previous if statement is handling the nextsecond and gametime variables for us, so we don't have to worry about them). If at least ClipPlayTime time has gone by, then we calculate when the next clip should be played, start the playing of the current audio clip, then figure out which audio clip to play next (we use another if statement in here to check if the next clip is beyond the value 37 and if so reset the value to 0. This is done because we know the valid values for the Betty audio clips go from 0 to 37).

With these changes, save the script and try the mission. Every 7 seconds, Betty will chime in with one of her sayings.

Creating Functions

So we've now seen how to add and change the script to do a few things. But if we have a block of code we want to use multiple times, it would be tedious to re-write this section of code multiple times. This is where functions come in.

Just like the built-in functions, we can actually make some of our own. As an example, even though we only use it once in the script, lets make a function that will handle playing the Betty audio clip. Further, lets make it so that this function handles the incrementing of the BettyClip variable so that the main code doesn't even have to know or deal with it.

First, we need to add the function prototype. That is, we need to tell the script what the function looks like, and what value to accept as parameters (if any) and what value to send back (if any).

So go back to the script file and add the following lines before line 104 or so (effectively we want this after the definition of the init function we saw before):

```
function PlayNextBettyClip;
code
endfunction;
```

This tells the script that we are defining a function called PlayNextBettyClip. It does not have any parameters, nor does it return a value. The “endfunction;” line is the end of this function definition. The line that says “code” is the command to tell the script where the code to execute is when this function needs to be executed.

Now, copy and paste the lines from the main script code to our new function so that our function looks like this:

```
function PlayNextBettyClip;
code
     PlayBetty(BettyClip);

     BettyClip = BettyClip + 1;
     if (BettyClip > 37) then
          BettyClip = 0;
     endif;
endfunction;
```

Now remove those same lines from the main script code and replace it with the single line “PlayNextBettyClip;” so that the if statement we added previously looks like this:

```
if(ClipPlayTime <= nextsecond) then
     ClipPlayTime = gametime + 7.0;

     PlayNextBettyClip;
endif;
```

Now we can use the function PlayNextBettyClip each time we want to play the next Betty audio clip, and we don't have to worry about maintaining the BettyClip variable either as it is done for us in our function. Try it out.

The PlayNextBettyClip function can only be accessed from within this module script however, because the function itself was only defined within the scope of this module. If we wanted access to this function in other scripts, we need to define it in a library instead.

Creating and Using Libraries

Is the previous section we saw how we can modify the mission's script to add some functionality to our mission, and how we could make some local functions to help in the writing of that script. But what if we wanted to have some of that functionality in multiple scripts. For that, we need a library.

Libraries are basically a collection of functions that the game loads for a mission. All the functions defined in the library are accessible from outside the library as well. In short, the functions in a library have a global scope.

So for this example, lets make our PlayNextBettyClip function part of a library.

Though we could add our function to one of the existing libraries, it is rather pointless to do so. Not only would we have to go extract the file and modify it (possibly creating problems for other missions that rely on those libraries) but as we will see, it is a simple matter to add our own libraries to a mission anyway. So in fact, we can get the best of both worlds. We don't need to modify any existing libraries, yet still allow our own libraries to be used (and of course still use the standard libraries as well).

As libraries are kept in their own files, we will need to create a new file for defining our library. So first, create a file in the {MCX}\data\missions folder called “libguide.abx” (libraries are assumed to have the .ABX file extension).

Libraries are defined much like modules are. So in this new file, add the following lines:

```
library LibGuide;
    const

    type

    var

function init;
code
endfunction;

code
endlibrary.
```

Here I'm defining a shell for all libraries. It has no functionality right now of course, but all the needed components are there. All that really needs to change here is the name of the library if we wanted to use this for yet another library.

You will notice that there is a code block defined for libraries, though it is not normally used for anything. It is not yet known what happens if code is placed in here, but it is expected to at least run once.

Now to add in our function. Cut the entire PlayNextBettyClip function from the mission script module and paste it in after the init function in our new library file.

Now that the function is defined in the library, we need to have it able to access the BettyClip variable. Remember that this variable is currently defined in the mission script file (guide.abl) and is no longer needed there. So first, remove the line

```
static integer BettyClip;
```

From guide.abl and add this same line to the variable list of the library instead.

Now as this variable still needs to be initialized in the library, in the library's init function add the following line after the code statement:

```
BettyClip = 0;
```

Finally, this same line should be removed from the mission script file since the variable no longer exists in that module.

Note how the function and variable declaration is identical to what was once in the mission script file, but now resides in the library instead. Meanwhile, the mission script module no longer has any reference to this variable and the only reference to the function is the function call in the main code itself.

The final version of libguide.abx looks like this:

```
library LibGuide;
    const

    type

    var
        static integer BettyClip;

function init;
code
    BettyClip = 0;
endfunction;
```

```
function PlayNextBettyClip;
code
    PlayBetty(BettyClip);

    BettyClip = BettyClip + 1;
    if (BettyClip > 37) then
        BettyClip = 0;
    endif;
endfunction;

code
endlibrary.
```

Now to get the library to be loaded by the game requires that we edit the mission's .FIT file. So open guide.fit into a text editor (you can find this file in the {MCX}\data\missions folder).

Look for the section heading called [ABLibraries] and add a new library entry like this:

```
st Library2 = "libguide"
```

So now this entire section becomes:

```
[ABLibraries]
st Library0 = "orders"
st Library1 = "miscfunc"
st Library2 = "libguide"
```

Save all the files and try this out. Our new library should be loaded along with the others, and our new library function can be accessed from the mission's code as before. However, we could also include this library in another mission script and use this same function there as well.

Try it out.

Unit AI Modules

Every unit in the game, be it a vehicle or mech, has a pilot associated with it. These pilots also have a pilot script module associated with them as well.

For example, if we load up the profile for the MechWarrior Lynx, we find this as the first entry in the [General] section:

```
st Brain = "pbrain"
```

This is defining which .ABL file the game will used to run the AI for Lynx. If we then opened the file pbrain.abl into an editor, we would then see the actual script module used by the game for preforming the AI.

It is beyond the scope of this document to delve into all the details of editing these files, but if we wanted to create our own MechWarrior profile, we could assign it our own script module file to use and create our own custom AI as well.

The AI module file used is similar to the mission's module script, and also has access to the library functions.

However, there is one major difference between a mission module script and an AI module script. Since the AI module is meant to control a unit, it can access what is called Brain Cells.

Brain cells are memory locations that the game actually uses for handling the behavior of the unit itself.

Brain cells are accessed via the functions GetIntegerMemory, SetIntegerMemory, GetRealMemory and SetRealMemory.

There are constants already defined to help find out which brain cell controls which function. You can find these constants in the include file OConst.ABI which is packaged in the mission.fst file.

The best way to learn about how the AI code works is to view and edit versions of the default AI files and study how they are put together.

Creating Include Files

As we've seen in the previous sections, many of the game's script files use the #include and #include_ parser directives to include external files into a script.

We can do the same thing by simply creating a file ourselves and including it at the appropriate location in our scripts.

But there are a few things that should be watched for when creating include files. First, it must be realized that the file should be seen as a continuation of the file we are including it into. That is, we shouldn't define both constants and variables in the same include file and expect to include this in the variable declaration block of a module or function as this will create a conflict.

The best way to view include files is to pretend that they are a copy and paste operation. That is, the contents of the file are copied into the current file and used directly there.

There is another little problem I discovered with include files. For some reason, the ABL language parser appears to have a problem with how the include file ends. The last line of an include file should be a blank line with no spaces on it (actually, it seems it can have some spaces, but it must be less than the number of spaces in the previous line if that previous line has more than 0 spaces). The last line of the file should also not have any script commands on it or the include file may not function, This is why I suggest to always have the last line of the include file be a blank line with no spaces at all on it.

For these examples, I will show spaces as a dot (.), carriage returns as ampersands (&) and the end of the file as a dollar sign ($)

This will work as an include file:

```
....static integer variable1;&
.....integer variable2;&
eternal integer variable3;&
$
```

This will not work as an include file (spaces on last line):

```
....static integer variable1;&
.....integer variable2;&
eternal integer variable3;&
...$
```

This also will not work as an include file (script code on last line):

```
....static integer variable1;&
.....integer variable2;&
eternal integer variable3;$
```

Passing Information Between Campaign Missions

Here is a little trick I discovered while researching the function details and descriptions. The trick will allow us to store and retrieve information even between missions within a campaign.

The trick effectively uses the SetGlobalValue and GetGlobalValue functions. These functions appear to be leftovers from previous code the developers worked on. It turns out that these functions store values in a global array called globalMissionValues. This is an array of 50 floats that, apart from the SetGlobalValue and GetGlobalValue functions, is never accessed anywhere else in the game executable (Note that this doesn't mean they are never used at all, other scripts may also use this space so test first!!!).

This means that whatever we store there will always be there during the execution of our campaign since the game never actually changes the contents of this array other than through these two functions. This also means however that the game never initializes this array, ever. So whatever values were in those memory locations the first time the game runs will remain there until something goes and changes it.

So to set a value we use the SetGlobalValue function by giving it an index to one of the array entries, and the value to store there. Now oddly, though the execution path seems to accept both integers and reals for the value parameter, but the function only appears to actually accept integers. For example, if we want to set the 14th array index to 5, we would use SetGlobalValue like so:

```
SetGlobalValue(13, 5);
```

Note that array indexes start counting at 0 not 1, so the 14th index is number 13.

To retrieve a value, we need to read it from the same index using the GetGlobalValue function. However, this function returns the value as a real, so we need to call the Trunc function as well in order to convert it back to a integer. So to read the value back, we would use:

```
Value = Trunc( GetGlobalValue(13) );
```

Note that the reading function does not need to be in the same script or even the same campaign mission as the code that set the value in the first place.

The one thing I have yet to test is that due to this conversion from integer to real and then back to integer, there may be an effective size limit on the possible values that can be stored here.

There is, of course, a rather limited amount of space for storing values here. There are only 50 reals for use (indexes 0 to 49) so we can't go hog wild here, but it does open all kinds of fun possibilities.

The ABL Debugger

***** Update *****

After further investigation, it is evident that many of the routines required by the ABL Debugger inside the game executable have been removed. Therefore, it will be impossible to get the ABL Debugger to function.

The remainder of this section is left intact since it also contains information on changing the game's screen resolution. However we will never be able to get the ABL debugger working any further than what is described here.

There appears to be an ABL script debugger built into the game. To get the debugger to appear, open system.cfg in the game's root folder and add/change the following.

First, add the heading `[DebugGameSystem]` to the file. Typically I place it above the ABL section but it doesn't appear to matter where exactly it is placed. If this section is present, the game internally sets the game variable DebugGameSystem to 1. Otherwise, this variable is defaulted to the value 0. Commenting out this heading will disable the extra blank window that appears.

Next, under the [ABL] section, add/change the following entries:

```
ul IncludeDebugInfo = 1
ul DebuggerEnabled = 1
```

These changes will turn on the debugger window inside the game. Setting these to 0 will disable the debugger again (though if the [DebugGameSystem] heading is present, the extra window pane will still be present.

The ABL Debugger window can be dragged around by click-dragging the window from its title bar. The extra window however, cannot be moved.

To get a better view of the debugger window, it is recommended that you also set a higher resolution for the game. This can be done by adding or changing an entry in prefs.cfg.

Simply open prefs.cfg in a text editor and add or change the line:

```
l Resolution=2
```

The valid resolution numbers are as follows:

0 = 640x480 (default)
1 = 800x600
2 = 1024x768
3 = 1280x1024

Note that this last (resolution 3) has been known to sometimes crash the game.

Unfortunately, there doesn't appear to be a way to actually make the debugger accept input. It is not known if this is because this feature was disabled by the developers, or if we are merely missing a needed piece of information to actually get it to work.

Also, if the ABL debugger window is closed, it is unknown how to get it back again short of exiting and restarting the game.

The following appears to be the help text that is displayed by the ABL Debugger (extracted from the game executable).

?	help
??	current module info
t{+|-}	start/stop trace mode
s{+|-}	start/stop step mode
p <variable>	display current value of variable
w[f|s]{+|-}{.} <variable>	set/remove variable watch (fetch & store)
m [0 thru warrior #]	set current module (or list them)
b{+|-} <line#>	set/remove breakpt

Obviously, if we can get the debugger operating properly, these commands could help greatly in developing fun new scripts for campaigns and solo missions.

ABL Standard Functions

The standard functions are functions that are built into the game's executable. That is, we can view them as a library of functions that are always available for use.

Some of the functions listed may have been deprecated, while others still seem to exist but have been disabled (see GetModuleName for an example of a disabled function). The reason they are still being listed is that it is unclear at this point if it is possible to create our own functions using the same names. If this is not the case, then we will need to know what names the parser knows about simply to avoid them. Another reason is that they may not in fact be deprecated, but merely done in a different way. In which case, we need to know what functions to test.

The list of functions was discovered in three steps. First, a list of functions names that the parser can use was taken from the game executable, then a list of functions that the parser checks the syntax for was cross-referenced with the first and any missing functions added. Then finally, this new list was cross-referenced with the list of internal routines that the game uses to actually execute these functions. Again, any missing function names were added to the final list.

During this, attempts were made to discover the parameters of the functions and their use from the code. This was not always successful however.

Lastly, attempts were made to find example uses in the ABL scripts used in the official campaign. This will help in discovering how these functions work as there are known instances of their use. However, not all of the functions listed are used.

As an update to the function usage and parameters, All functions were also cross-referenced with the source code for MechCommander 2 and that usage reverified with the game executable.

Finally, be aware that much of what is presented here is guesswork. Many tests still need to be done, and parameter names and uses may be inaccurate.

Functions By Category
Functions in Alphabetical Order

Functions By Category

Functions in Alphabetical Order

(see next page for list of deprecated functions)

Deprecated Functions

AddPrisoner
AttackClosestTarget
AttackPerOrders
AttackThreat
FileExists
FireUponEnemyFireOnly
FireWeapon
GetAction
GetAttackOrder
GetFormation
GetGuardDistanceTo
GetGuardObjective
GetGuardPoint
GetGuardRadii
GetMode
GetModuleHandle
GetModuleName
GetMoveOrder
GetPhase
GetWeaponsInRange
GetWeaponsLocked
GetWeaponsReady
Handle
OpenFire
OrderFormation
OrderPatrolPath
OrderTraversePath
PlayWaveFile
Retreat
SetAction
SetGuardObjective
SetGuardPoint
SetGuardRadii
SetMode
SetModuleName
SetMoverBehavior
SetMoverOverlayWeight
SetPhase
SetUpdateTime
StartEnemyScan
StartFieldScan
StartFriendlyScan
StartMovePath
StartVehicleScan
TimeToImpact
UseFireOdds
UseFireRange
UseSpeed

Audio/Video Functions

PlayBetty
PlayDigitalMusic
PlaySmacker
PlaySoundEffect
PlaySpeech
PlayVideo
StopMusic

Data Functions

DistanceToObject
DistanceToPosition
GetAlarmTriggers
GetArmorPts
GetAttackerInfo
GetAttackers
GetChallenger
GetContactId
GetContactRelativePosition
GetContacts
GetContactStatus
GetCurrentBRValue
GetEnemyCount
GetFireRanges
GetFixed
GetGlobalValue
GetHomeTeam
GetId
GetIntegerMemory
GetLastTacOrder
GetMaxArmor
GetMessage
GetObjectActive
GetObjectDamage
GetObjectDmgPts
GetObjectMaxDmg
GetObjectPosition
GetPilotId
GetPilotWounds
GetRealMemory
GetRelativePositionToObject
GetRelativePositionToPoint
GetRepairState
GetSalvage
GetSensorsWorking
GetStrikes
GetTacOrder
GetTarget
GetTerrainObjectPartId
GetTime
GetTimeLeft
GetTimeWithoutOrders
GetUnitMates
GetUnitStatus
GetVehiclePartId
GetVisualRange
GetWarriorStatus
GetWeaponAmmo
GetWeaponRanges
GetWeapons
GetWeaponShots
HasMoveGoal
HasMovePath
InArea
IsCapturable
IsCaptured
IsContact
IsGateOpen
IsServer
IsTeamTargeting
SelectContact
SelectObject
SelectUnit
SelectWarrior
SendMessage
SetAnimation
SetAttackRadius
SetBuildingName
SetCaptureable
SetCaptured
SetChallenger
SetCurrentBRValue
SetExplosionDamage
SetExplosionRadius
SetGlobalValue
SetIntegerMemory
SetMoveGoal
SetObjectActive
SetObjectDamage
SetObjectivePos
SetObjectiveStatus
SetObjectiveTimer
SetObjectiveType
SetOrderMode
SetPilotWounds
SetPotentialContact
SetRadio
SetRealMemory
SetRevealed
SetSalvage
SetSalvageStatus
SetSensorRange
SetStrikes
SetTarget
SetTimer
SetTonnage
SetTrainSpeed
SortWeapons
WasEverCapturable

Deprecated Functions

AddPrisoner
AttackClosestTarget
AttackPerOrders
AttackThreat
FileExists
FireUponEnemyFireOnly
FireWeapon
GetAction
GetAttackOrder
GetFormation
GetGuardDistanceTo
GetGuardObjective
GetGuardPoint
GetGuardRadii
GetMode
GetModuleHandle
GetModuleName
GetMoveOrder
GetPhase
GetWeaponsInRange
GetWeaponsLocked
GetWeaponsReady
Handle
OpenFire
OrderFormation
OrderPatrolPath
OrderTraversePath
PlayWaveFile
Retreat
SetAction
SetGuardObjective
SetGuardPoint
SetGuardRadii
SetMode
SetModuleName
SetMoverBehavior
SetMoverOverlayWeight
SetPhase
SetUpdateTime
StartEnemyScan
StartFieldScan
StartFriendlyScan
StartMovePath
StartVehicleScan
TimeToImpact
UseFireOdds
UseFireRange
UseSpeed

Movement Functions

DistanceToObject
DistanceToPosition
GetObjectPosition
GetRelativePositionToObject
GetRelativePositionToPoint
HasMoveGoal
HasMovePath
InArea
ObjectInWithdrawal
OrderMoveTo
OrderMoveToContact
OrderMoveToObject
OrderWithdraw
SetMoveGoal
SetTrainSpeed

Multiplayer Functions

GetHomeTeam
GetMessage
IsServer
IsTeamTargeting
SendMessage
SetRevealed

Miscellaneous Functions

Object Functions

DamageObject
DistanceToObject
GetId
GetObjectActive
GetObjectDamage
GetObjectDmgPts
GetObjectMaxDmg
GetObjectPosition
GetPilotId
GetRelativePositionToObject
GetRepairState
GetSalvage
GetTerrainObjectPartId
GetUnitStatus
GetVehiclePartId
IsCapturable
IsCaptured
IsContact
IsTeamTargeting
ObjectChangeSides
ObjectClass
ObjectCommander
ObjectCreate
ObjectExists
ObjectInWithdrawal
ObjectSide
ObjectStatus
ObjectStatusCount
ObjectSuicide
ObjectTypeId
ObjectVisible
Repair
SelectContact
SelectObject
SelectUnit
SelectWarrior
SetAnimation
SetAttackRadius
SetBuildingName
SetCaptureable
SetCaptured
SetExplosionDamage
SetExplosionRadius
SetObjectActive
SetObjectDamage
SetSalvage
SetSalvageStatus
SetSensorRange
SetTonnage
WasEverCapturable

Objectives Functions

CheckObjectiveStatus
CheckObjectiveTimer
CheckObjectiveType
SetObjectivePos
SetObjectiveStatus
SetObjectiveTimer
SetObjectiveType

Orders Functions

GetLastTacOrder
GetTacOrder
GetTimeWithoutOrders
OrderAttackContact
OrderAttackObject
OrderCapture
OrderDeployElementals
OrderLoadElementals
OrderMoveTo
OrderMoveToContact
OrderMoveToObject
OrderPowerDown
OrderPowerUp
OrderRefit
OrderTest
OrderWait
OrderWithdraw

Selection Functions

SelectContact
SelectObject
SelectUnit
SelectWarrior

Time Functions

CheckObjectiveTimer
CheckTimer
EndTimer
GetTime
GetTimeLeft
GetTimeWithoutOrders
SetObjectiveTimer
SetTimer

Unit Functions

GetAttackerInfo
GetAttackers
GetChallenger
GetContactId
GetContactRelativePosition
GetContacts
GetContactStatus
GetCurrentBRValue
GetEnemyCount
GetFireRanges
GetFixed
GetId
GetLastTacOrder
GetMaxArmor
GetObjectActive
GetObjectDamage
GetObjectDmgPts
GetObjectMaxDmg
GetObjectPosition
GetPilotId
GetPilotWounds
GetRepairState
GetSalvage
GetSensorsWorking
GetTarget
GetTimeWithoutOrders
GetUnitMates
GetUnitStatus
GetVehiclePartId
GetWarriorStatus
GetWeaponAmmo
GetWeaponRanges
GetWeapons
GetWeaponShots
HasMoveGoal
HasMovePath
InArea
IsCapturable
IsCaptured
IsContact
IsTeamTargeting
Repair
SetAttackRadius
SetChallenger
SetCurrentBRValue
SetMoveGoal
SetOrderMode
SetPilotWounds
SetPotentialContact
SetRadio
SetSensorRange
SetTarget
SetTonnage

AddStrikes

Prototype

function AddStrikes(integer CommanderID, integer StrikeType, integer Num);

CommanderID	ID of commander to give artillery to 0 = Player (Inner Sphere) 1 = Enemy (Clan) 2 = Allies
StrikeType	Type of artillery to give 0 = Small strikes 1 = Large strikes 2 = Sensor drones 3 = Camera drones
Num	Number of strikes to add

Description

Adds artillery strikes to a given commander (side).

Returns

No return value

Remarks

The mission editor imposes a maximum of 5 artillery strikes per type. Though this is an artificial limit. This function can surpass this limit.

Seen In

Many of the multiplayer scripts.

Example use

This will add a single camera drone to the player.

```
AddStrikes(0, 3, 1);
```

See also

SetStrikes, GetStrikes

SetStrikes

Prototype

function SetStrikes(integer CommanderID, integer Artillery, integer Amount);

CommanderID	ID of commander to set the strikes on 0 = Player (Inner Sphere) 1 = Enemy (Clan) 2 = Allies
Artillery	Which artillery to give (0-3) 0 = Small strikes 1 = Large strikes 2 = Sensor drones 3 = Camera drones
Amount	Number of strikes to set

Description

Sets the number of artillery strikes.

Returns

No return value

Remarks

This function directly sets the artillery strikes. Any previous value is lost.

This function is not affected by the mission editors limit of 5 artillery strikes per type.

Seen In

Not used in the official scripts.

Example use

This will set the number of large artillery strikes the player has to 3.

```
SetStrikes(0, 1, 3);
```

See also

AddStrikes, GetStrikes

GetStrikes

Prototype

function GetStrikes(integer CommanderID, integer Artillery) : integer;

CommanderID	ID of commander to verify 0 = Player (Inner Sphere) 1 = Enemy (Clan) 2 = Allies
Artillery	Which artillery strike type to check (0-3) 0 = Small strikes 1 = Large strikes 2 = Sensor drones 3 = Camera drones

Description

Retrieves the number of remaining strikes of the given artillery type a commander has.

Returns

Number of strikes available

Remarks

none

Seen In

Many multiplayer scripts

Example use

This sets the variable NumSensorDrones with the number of sensor drones the player currently has.

```
NumSensorDrones = GetStrikes(0, 2);
```

See also

AddStrikes, SetStrikes

GetHomeTeam

Prototype

function GetHomeTeam : integer;

Description

Retrieve the ID of the home team.

Returns

ID of home team:

500 = IS
501 = CLAN
502 = ALLIED

Remarks

This function appears to be for multiplayer use, though it should be able to be used in solo and campaign missions as well.

Seen In

Many multiplayer scripts

GetMessage

Prototype

function GetMessage(integer Channel) : integer;

Channel Which channel to retrieve a message from

Description

Retrieves a message from a given message queue.

Returns

Message code

Remarks

This appears to be for multiplayer games only. The message code is effectively a user defined code that gets sent.

Though this function can be used in solo and campaign missions, it has no real use as its counterpart, SendMessage, does check for multiplayer mode.

Seen In

Many multiplayer scripts

See also

SendMessage

SendMessage

Prototype

function SendMessage(integer Channel, integer Message);

Channel	Channel to send message on
Message	Message code to be sent

Description

Sends a message code on a given message channel.

Returns

No return value

Remarks

This function is for multiplayer games only. This function will check for multiplayer mode and if it is not, the function merely returns without doing anything.

In fact, it also checks to ensure that it is the server of the multiplayer game as well. Therefore, only the game server can actually use this function, though it is safe to use in non-multiplayer or non-server situations.

Seen In

in many multiplayer and non-multiplayer script files

See also

GetMessage

IsTeamTargeting

Prototype

```
function IsTeamTargetting(integer TeamID, integer TargetID,
                          integer ExceptUnitID) : boolean;
```

TeamID	ID Team to check for targeting
TargetID	Unit ID to check if being targeted
ExceptUnitID	ID of unit to exclude from the check May be 0 for no exception unit

Description

Checks if any member of a team has the given unit actually targeted. If ExceptUnitID is non-zero, if will exclude this unit from the check.

Returns

TRUE is a team member is targeting the given target, FALSE if not.

Remarks

none

Seen In

orders.abx

Example use

```
if (isTeamTargeting(MyTeam,contactID,Me)) then
```

GetRepairState

Prototype

function GetRepairState(integer Unit) : integer;

Unit ID of unit to be checked

Description

Retrieve the percentage of repairable armor and internal structure points of the given unit.

Returns

Percentage of repairable armor and internal structure points.

Remarks

This function only works with “mover” type objects (mechs and vehicles etc.)

Seen In

not seen in any official scripts

See also

GetUnitStatus

GetFixed

Prototype

function GetFixed(integer Unit, integer Bay, integer Parameters) : integer;

Unit	ID of unit to be ordered
Bay	ID of bay to be used
Parameters	Unknown

Description

Orders a unit to go get fixed at the given repair bay.

Returns

One of the following codes:

-1	Unknown error
0	Initiating repair (it's all good)
1	Bay is out of repair points (probably means the bay is destroyed)
2	Wrong kind of bay
3	Bay is already repairing this unit
4	Bay is repairing something else
5	Unit is being repaired by something else
6	Unit doesn't need repair
7	Illegal unit object
8	Illegal bay object
9	Alignment mismatch (unit and bay are on different sides/teams)

Remarks

Vehicles should be sent to vehicle repair bays, mechs to mech repair bays.

A return value of 0 means everything functioned correctly.

Seen In

orders.abx

See also

OrderRefit

Repair

Prototype

function Repair(integer Unit, real Points);

Unit	ID of unit to repair
Points	Number of points to repair

Description

Repairs armor and internal structure of a unit.

Returns

no return value

Remarks

This function only works on mech units.

This function will not repair missing parts. That is, if an arm was blown off in combat, the unit will not get a new arm back.

The function will use up as many points as passed in. If the number of points does not cover all damage, then only that amount is repaired. If the number of points is greater than the damage, then all damage will be repaired and the remaining points are ignored. Since the number of points to repair is merely a value passed into this function, the Points parameter only makes sense as a cap for the repair since setting it to an incredibly high value will guarantee full repairs.

This function does not have any delay mechanism built into it. Therefore, all repair is instantaneous.

Seen In

not seen in any official scripts

GetUnitStatus

Prototype

function GetUnitStatus(integer Unit) : real;

Unit ID of unit to check

Description

Retrieves the percent health (0-100) of the given unit.

Returns

Percentile health of unit

Remarks

Unsure if the unit ID applies only to mechs, or if vehicles and buildings can also be checked in this way. There doesn't appear to be any checks for object classes built into this function however.

In fact, the internal code calls a function named getStatusRating which is a C++ member function defined for mechs, vehicles and buildings.

Seen In

orders.abx

Example use

```
if (getUnitStatus(Me) > 99.0) then
```

See also

GetRepairState

GetRelativePositionToObject

Prototype

function GetRelativePositionToObject(integer UnitID, real Angle,
real Distance, integer Flags, @real[3] Position);

UnitID	ID of object
Angle	Angle in degrees where point is to be at -180.0 <= Angle <= 180.0 + is left, - is right
Distance	Distance, in meters, from the object the new point is to be at
Flags	Optional flags 1 = Absolute 2 = Passable Start 4 = Passable Goal
Position	An array of three real values to receive the new point

Description

Calculates a point on the map that is at a given distance and angle relative to the given object.

Returns

No return value. The calculated point is passed back through the Position parameter instead.

Remarks

Though there are three possible values for the Flags parameter shown here, and as they are bit values, they could be OR'd (combined) together, only the Passable Start (2) value is actually checked for in the code. The other values are ignored. Therefore the Flags parameter should either be 2 or 0, as any other value will be irrelevant.

It is not clear what exactly the flags (or flag) value really does in the code.

Seen In

orders.abx

Example use

getRelativePositionToObject(GuardObjectID, Angle, Distance, 2, EscortPoint);

See also

GetRelativePositionToPoint

GetRelativePositionToPoint

Prototype

```
function GetRelativePositionToPoint(real[3] Point, real Angle,
        real Distance, integer Flags, @real[3] Position);
```

Point	Array of 3 reals containing the point coordinates
Angle	Angle in degrees where position will be -180.0 <= Angle <= 180.0 + is left, - is right
Distance	Distance from Point the position will be
Flags	Optional flags 1 = Absolute 2 = Passable Start 4 = Passable Goal
Position	Array of 3 reals that will contain the resulting map position

Description

Calculates a map position that is a given distance and angle away relative to the given map point.

Returns

No return value. The calculated position is passed back through the Position parameter instead.

Remarks

Though there are three possible values for the Flags parameter shown here, and as they are bit values, they could be OR'd (combined) together, only the Passable Start (2) value is actually checked for in the code. The other values are ignored. Therefore the Flags parameter should either be 2 or 0, as any other value will be irrelevant.

It is not clear what exactly the flags (or flag) value really does in the code.

Seen In

not seen in any official scripts

See also

GetRelativePositionToObject

CallStrikeEx

Prototype

function CallStrikeEx(integer StrikeType, integer TargetID,
real MapXCoord, real MapYCoord, real MapZCoord,
boolean FromClan, real TimeToImapct);

StrikeType	Type of strike 248 = Big Artillery 249 = Small Artillery 250 = Sensor Probe 507 = Big air strike 508 = Small air strike 509 = Sensor (air)
TargetID	ID of Target object 0 = No target, use map coords instead
MapXCoord	X coordinate where to send the strike Used only if TargetID is 0
MapYCoord	Y coordinate where to send the strike Used only if TagetID is 0
MapZCoord	Z coordinate where to send the strike This parameter is never used by the function
FromClan	Strike coming from the Clan? TRUE = yes, FALSE = no
TimeToImapct	Time, in seconds, until impact

Description

Calls down an air strike or artillery strike at the given coordinates or on a target, delayed by the given time.

Returns

No return value

Remarks

If the TargetID is 0 (or an invalid targetID given), then it will send the strike to the given coordinates. If the TargetID is not 0 and valid, then the given coordinate values are ignored and the location of the target object is used instead.

If the TimeToImpact parameter is negative, the code will treat it as 0.

The Z coordinate provided does not matter as the code will retrieve the elevation of the terrain at the map coordinates provided.

This function will check if it is running in a muliplayer game. If so, it internally calls the Fatal function with error code 86 and the message that air strikes are not allowed in multiplayer games. Because of this, this function cannot be used in any script intended for use in a multiplayer map.

A clue as to why this is can be seen in the source code for MechCommander 2. The following can be seen as a comment in the file ablmc2.cpp starting at line 3974:

```
// Since artillery strikes are commander(player) based, not team based,
// & current missions are coded to assume team based, we'll just
// fatal if we're not in a campaign mission. Then, we can decide if
// it's worth re-coding ABL scripts (since we'll have to change the
// parameters for this call to specify the player that's calling the
// strike)...
```

Guess they never did.

Seen In

m0201_Airstrike2.ABL

Example use:

```
CallStrikeEx(507,-1,-2036.0,-178.0,-1.0,TRUE,4.0);
```

See also

CallStrike

IsGateOpen

Prototype

function IsGateOpen(integer GateID) : boolean;

GateID ID of the gate object to be checked

Description

Determines if a gate is open.

Returns

TRUE if the gate is open, FALSE otherwise

Remarks

none

Seen In

Urbana_STR.abi

Example use

```
if (isGateOpen(95632)) then
```

See also

LockGateOpen, LockGateClosed, ReleaseGateLock

ReleaseGateLock

Prototype

function ReleaseGateLock(integer GateID);

GateID ID of gate object

Description

Releases the lock status of a gate.

Returns

no return value

Remarks

none

Seen In

Urbana_STR.abi

Example use

ReleaseGateLock(92888);

See also

IsGateOpen, LockGateOpen, LockGateClosed

LockGateClosed

Prototype

function LockGateClosed(integer GateID);

GateID ID of gate object

Description

Locks a gate in the closed position.

Returns

no return value

Remarks

none

Seen In

Urbana_MP.abi

Example use

```
LockGateClosed(92800);
```

See also

IsGateOpen, LockGateOpen, ReleaseGateLock

LockGateOpen

Prototype

function LockGateOpen(integer GateID);

GateID ID of gate object

Description

Locks a gate in the open position

Returns

no return value

Remarks

none

Seen In

Urbana_MP.abi

Example use

```
LockGateOpen(92800);
```

See also

IsGateOpen, LockGateClosed, ReleaseGateLock

SetTrainSpeed

Prototype

function SetTrainSpeed(integer TrainID, real Speed);

TrainID	ID of train object
Speed	Speed, in kilometers per second

Description

Sets the speed at which a train moves

Returns

no return value

Remarks

The speed value seems to be in kilometers per second according to the comment of the example found, but this is not guaranteed.

Seen In

MIS0205.ABL

Example use

```
SetTrainSpeed(512000,8.0); // ID, KPS
```

AddPrisoner

Prototype

function AddPrisoner(integer BuildingID, integer PrisonerID) : integer;

BuildingID	ID of building to add to
PrisonerID	ID of prisoner to add

Description

Adds the given prisoner to a given building.

Returns

0 if successful

Remarks

This function appears to have been deprecated.

The execution pipeline does not appear to exist, although the parser does appear to check the syntax of this function.

Based on the source code for MechCommander 2, the prisoner ID appears to be the ID of a Warrior.

Seen In

not seen in any scripts

OrderDeployElementals

Prototype

function OrderDeployElementals(integer OrderParams);

OrderParams	Order parameter. Typically 0	

Description

Orders an elemental carrier to deploy any loaded elementals.

Returns

no return value

Remarks

This function uses the current object as its target for the deployment order. Therefore, it should only be called in the AI code for the elemental carrier itself.

It is unknown how many elementals are deployed by this command. Nor where the elementals appear via this order.

Seen In

orders.abx

Example use

```
orderDeployElementals(0);
```

See also

OrderLoadElementals

OrderLoadElementals

Prototype

function OrderLoadElementals(integer CarrierID);

CarrierID ID of carrier to load into.

Description

Orders elementals of current group to load into the given elemental carrier.

Returns

no return value

Remarks

This function uses the current object as its target for the order. Therefore, it is likely that it must only be called in the AI code for an elemental (not the carrier).

It is unknown if a carrier must first be empty before issuing this order, or if the carrier can load and unload at any time.

Seen In

not seen in any scripts

See also

OrderDeployElementals

CallStrike

Prototype

```
function CallStrike(integer StrikeType, integer TargetID,
        real MapXCoord, real MapYCoord, real MapZCoord,
        boolean FromClan);
```

StrikeType	Type of strike 248 = Big Artillery 249 = Small Artillery 250 = Sensor Probe 507 = Big air strike 508 = Small air strike 509 = Sensor (air)
TargetID	ID of Target object 0 = No target, use map coords instead
MapXCoord	X coordinate where to send the strike Used only if TargetID is 0
MapYCoord	Y coordinate where to send the strike Used only if TagetID is 0
MapZCoord	Z coordinate where to send the strike This parameter is never used by the function
FromClan	Strike coming from the Clan? TRUE = yes, FALSE = no

Description

Calls down an air strike or artillery strike at the given map coordinates or on the given target.

Returns

no return value

Remarks

If the TargetID is 0 (or an invalid targetID given), then it will send the strike to the given coordinates. If the TargetID is not 0 and valid, then the given coordinate values are ignored and the location of the target object is used instead.

The Z coordinate provided does not matter as the code will retrieve the elevation of the terrain at the map coordinates provided.

This function will check if it is running in a muliplayer game. If so, it internally calls the Fatal function with error code 86 and the message that air strikes are not allowed in multiplayer games. Because of this, this function cannot be used in any script intended for use in a multiplayer map.

A clue as to why this is can be seen in the source code for MechCommander 2. The following can be seen as a comment in the file ablmc2.cpp starting at line 3917:

```
// Since artillery strikes are commander(player) based, not team based,
// & current missions are coded to assume team based, we'll just
// fatal if we're not in a campaign mission. Then, we can decide if
// it's worth re-coding ABL scripts (since we'll have to change the
// parameters for this call to specify the player that's calling the
// strike)...
```

Guess they never did.

Seen In

not seen in any scripts

See also

CallStrikeEx

SetBuildingName

Prototype

function SetBuildingName(integer BuildingID, integer NameID);

BuildingID	ID of building to be affected
NameID	ID of resource string to use

Description

Sets the name of a building to the name defined by the string resource ID.

Returns

no return value

Remarks

Valid values for NameID are those of the string IDs in the game's resource file.

Seen In

MP7_19INIT.ABI

Example use

```
setBuildingName(BaseAuxGen,LOCAL_TURRET_POWER);
```

WasEverCapturable

Prototype

function WasEverCapturable(integer ObjectID) : boolean;

ObjectID ID of object to be checked

Description

Check if an object has ever been flagged as capturable. This function will return the value even if the object is dead, currently captured, etc.

Returns

TRUE if the object was ever flagged as capturable, FALSE otherwise.

Remarks

This function only applies to vehicles. It is unknown what happens if a non-vehicle ID is given.

Seen In

miscfunc.abx

Example use

if (wasEverCapturable(x)) then

See also

IsCapturable, IsCaptured, OrderCapture, SetCaptureable, SetCaptured

IsCapturable

Prototype

function IsCapturable(integer ObjectID) : boolean;

ObjectID ID of object to be checked

Description

Check is an object can be captured.

Returns

TRUE is the object can be captured, FALSE otherwise.

Remarks

It is not known if the object ID is restricted to buildings and/or vehicles, or if other types of objects can also be checked. The code does not appear to specifically check for this however.

Seen In

orders.abx

Example use

```
if ((isCapturable(contactID)) OR (isCaptured(contactID) == 1)) then
```

See also

IsCaptured, OrderCapture, SetCaptureable, SetCaptured, WasEverCapturable

IsCaptured

Prototype

function IsCaptured(integer ObjectID) : integer;

ObjectID ID of the object to be checked

Description

Checks if the given object has been captured, and by which side.

Returns

The ID of the team who has captured the object, if any.

Remarks

It is unknown what the return value is if the object is not captured at all. Or even it indeed returns the team ID or just a true/false value saying it was captured.

Some of the code appears to actually return the number of units captured instead of a true/false value. More testing needs to be done.

Seen In

orders.abx

Example use

```
if ((isCapturable(contactID)) OR (isCaptured(contactID) == 1)) then
```

See also

IsCapturable, OrderCapture, SetCaptureable, SetCaptured, WasEverCapturable

OrderCapture

Prototype

function OrderCapture(integer UnitID, integer Params);

UnitID	ID of unit to be captured
Params	Order parameters

Description

Orders a unit to capture the given object.

Returns

no return value

Remarks

This function orders the current object, so it should be used in the object's AI code only.

The order parameter values are not known. Assuming this is typically 0.

Seen In

not seen in any scripts

See also

IsCapturable, IsCaptured, SetCaptureable, SetCaptured, WasEverCapturable

SetCaptureable

Prototype

function SetCaptureable(integer ObjectID, boolean CaptureState);

ObjectID	ID of object to be affected
CaptureState	State to set object to

Description

Sets the state of the capturable flag of an object.

Returns

no return value

Remarks

Setting the CaptureState flag to TRUE will allow an object to be captured, while setting it to FALSE disallows capturing.

If in a non-multiplayer game, only buildings and ground vehicles can be set with this function.

If this function is called from a multiplayer game, the object's captured flag is cleared.

This function actually appears to ignore the given state parameter and merely set the capturable flag (or clear the captured flag in multiplayer mode). Because of this, it is impossible to allow a building to be captured multiple times.

Seen In

1_1INIT.ABI

Example use

setCaptureable(LRMStack, TRUE);

See also

IsCapturable, IsCaptured, OrderCapture, SetCaptured, WasEverCapturable

SetCaptured

Prototype

function SetCaptured(integer ObjectID);

ObjectID ID of object to be captured

Description

Set the state of an object to captured.

Returns

no return value

Remarks

This function should immediately tag an object as being captured. It is likely that it will mark the object as being captured by the same team that is running the piece of code that calls this function.

The function does not appear to check the type/class of the given object.

Seen In

not seen in any scripts

See also

IsCapturable, IsCaptured, OrderCapture, SetCaptureable, WasEverCapturable

OrderRefit

Prototype

function OrderRefit(integer UnitID, integer Params);

UnitID	ID of unit to repair
Params	Order parameters. Typically 0

Description

Orders the current object to repair the given mech.

Returns

no return value

Remarks

This function checks for mechs and so will not work on anything other than on a mech.

This function assumes that the current object is the one to go refit the mech.

It is unknown what will happen if this function is called from the AI of a non-refit truck. The code appears to merely check if the current object has a pilot.

Seen In

not seen in any scripts

See also

GetFixed

GetSalvage

Prototype

function GetSalvage(integer ObjectID, integer ListSize,
@integer[] ItemList, @Integer[] QtyList);

ObjectID	ID of the object to check
ListSize	Size of list provided
ItemList	List of item IDs returned
QtyList	List of the number of items in ItemList

Description

Get the list of salvage items on an object

Returns

no return value. The list of items is placed in the ItemList parameter, and the number of each is returned in the QtyList

Remarks

The ListSize parameter is the size of the array that is being passed down. If the array is not large enough to hold the entire list, then only the first ListSize items will be returned.

The QtyList array holds the number of items corresponding to the item IDs held in ItemList. Therefore, ItemList[0] will be the ID of the first item and QtyList[0] will be the number of that item. Etc.

Seen In

not seen in any script files

See also

SetSalvage, SetSalvageStatus

SetRevealed

Prototype

function SetRevealed(integer Team, real Radius, real[3] Position);

Team	ID of team (side) to reveal for
Radius	Radius of area to be revealed
Position	Map coordinates for the center of the revealed area

Description

Reveals a circular area of the map for a given team/side.

Returns

no return value

Remarks

This function reveals the map for a team. The size of the area to be revealed is dictated by the radius, and the location of the area is specified by the position.

It only reveals the map for a given team/side. This makes it safe for use in a multiplayer mission.

Seen In

1_1MXLP.ABI

Example use

```
aPoint[0] = 6700;
aPoint[1] = -300;
aPoint[2] = 0.0;
setRevealed(1,125.0,aPoint);
```

SetAnimation

Prototype

function SetAnimation(integer ObjectID, integer AnimState, integer SubState);

ObjectID	ID of object
AnimState	Unknown
SubState	Unknown

Description

Sets the animation state and substate of an object.

Returns

no return value

Remarks

It is unknown what happens if this is called on an object that doesn't have animations.

Valid values for AnimState and SubState are not known

Seen In

1_4ACT.ABI

Example use

```
setAnimation(getTerrainObjectPartID(21,373),0,-1);
```

SetSalvageStatus

Prototype

function SetSalvageStatus(int ObjectID, boolean Status) : boolean;

ObjectID	ID of object to be added as salvage
Status	Status to set

Description

Adds an object as salvage.

Returns

TRUE if function succeeds. FALSE otherwise.

Remarks

If the object specified is a building or a vehicle, the list of salvage items associated with the building or vehicle is added to the salvage items the player receives. If the object is a mech, the mech itself is added to the salvage list.

Specifying FALSE as the status will remove all salvage items the object is giving the player from the player's salvage list.

Seen In

not seen in any scripts.

See also

GetSalvage, SetSalvage

SetSalvage

Prototype

function SetSalvage(integer ObjectID, integer ItemID, integer Qty) : boolean;

ObjectID	ID of object to be changed
ItemID	ID of component to add as salvage
Qty	Number of items to add

Description

Adds the type and quantity of salvage items to an object.

Returns

TRUE if successful, FALSE otherwise.

Remarks

Only one item type can be set at a time, though multiple calls to this function will allow adding more items.

Seen In

1_1MXIN.ABI

Example use

```
setSalvage(salv1,117,2);
```

See also

GetSalvage, SetSalvageStatus

SetExplosionRadius

Prototype

function SetExplosionRadius(integer ObjectID, real Radius);

ObjectID	ID of object to be affected
Radius	New radius to set, in meters

Description

Sets the explosion radius of an exploding object.

Returns

no return value

Remarks

This function will have no effect if the object provided is not an exploding object.

Seen In

not seen in any scripts

See also

SetExplosionDamage

SetExplosionDamage

Prototype

function SetExplosionDamage(integer ObjectID, real Damage);

ObjectID	ID of object to be affected
Damage	Amount of damage the explosion should do In points of damage.

Description

Sets the amount of damage that an exploding object can do.

Returns

no return value

Remarks

This function will have no effect if the object provided is not an exploding object.

The valid range of values for the damage parameter is unknown, but as they are damage points, it may not truly matter. Though obviously, having this value too high will cause instant death.

Seen In

not seen in any scripts

See also

SetExplosionRadius

PlayWaveFile

Prototype

function PlayWaveFile(integer Unknown1, real Unknown2);

Unknown1	Unknown
Unknown2	Unknown

Description

Plays a wave file.

Returns

no return value

Remarks

The first parameter is likely an index indicating which wave file to play, though where this list of files may be is unknown.

The second parameter is unknown, but may be a volume setting, though this is merely speculation.

This function appears to have been deprecated.

Seen In

not seen in any scripts

See also

PlayBetty, PlaySpeech, PlaySoundEffect, PlayDigitalMusic, StopMusic

SetTonnage

Prototype

function SetTonnage(integer ObjectID, real Tonnage);

ObjectID	ID of object to affect
Tonnage	Tonnage value to use

Description

Sets the tonnage of an object.

Returns

no return value

Remarks

Not sure what exactly this is supposed to do. It is not known what effect this will have.

Seen In

not seen in any scripts

SetSensorRange

Prototype

function SetSensorRange(int ObjectID, real Range);

ObjectID	ID of object to be changed
Range	Range value to use

Description

Sets the sensor range of an object.

Returns

no return value

Remarks

It is unknown what effect this has on a unit, if any, or if it is affected by the sensor equipment on the unit.

Seen In

not seen in any scripts

SetPotentialContact

Prototype

function SetPotentialContact(integer ObjectID, integer Unknown) : integer;

ObjectID	ID of object
Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

The purpose of this function is unclear, though it appears to allow objects and or units to be seen as potential threats by the AI scripts.

Seen In

1_1INIT.ABI

Example use

```
setPotentialContact(CampHQ1,0);
```

SetMoverOverlayWeight

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This function appears to have been deprecated.

Seen In
not seen in any scripts

SetMoverBehavior

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This function appears to have been deprecated

Seen In
not seen in any scripts

SetObjectivePos

Prototype

function SetObjectivePos(integer Objective, real XCoord, real YCoord,
real ZCoord);

Objective	Objective to be changed
XCoord	X coordinate to use
YCoord	Y coordinate to use
ZCoord	Z coordinate to use

Description

Sets the position of a mission objective.

Returns

no return value

Remarks

Unlike most other functions, this one will also accept integer values for the coordinate parameters. This is actually checked for by the language parser.

Seen In

1_1INIT.ABI

SetGlobalValue

Prototype

function SetGlobalValue(integer Global, real Value);

Global	Number of global variable to change	
Value	Value to set	(can be an integer or real)

Description

Sets the value of a global variable.

Returns

no return value

Remarks

Unlike most other functions, this function appears to accept an integer or real value as the second parameter.

There are a maximum of 50 globals (numbered 0-49).

This function actually accesses and stores the value into an internal game array of 50 floats called globalMissionValues. However, this array does not appear to ever be used elsewhere in the game executable. If this is in fact the case, then this function and its counterpart GetGlobalValue could be used to pass information from one mission to the next.

Seen In

CPL0203A.abl

Example use

```
setGlobalValue(CAPTUREDCONVOYA+(Me-CLAN_VEHICLE0_STAR0), YES);
```

See also

GetGlobalValue

GetGlobalValue

Prototype

function GetGlobalValue(integer Global) : real;

Global Global mission variable to be read

Description

Gets the value of a global mission variable.

Returns

The value of the global mission variable.

Remarks

Unlike most other functions, this one appears to allow either an integer or real value as its return value type. This appears to be checked for by the language parser.

There are a maximum of 50 globals (numbered 0-49).

This function actually accesses and reads the value into an internal game array of 50 floats called globalMissionValues. However, this array does not appear to ever be used elsewhere in the game executable. If this is in fact the case, then this function and its counterpart SetGlobalValue could be used to pass information from one mission to the next.

Seen In

E3_0203.ABL

Example use

```
if (getGlobalValue(CAPTURETRUCKS + x) == YES) then
```

See also

SetGlobalValue

GetObjectMaxDmg

Prototype

function GetObjectMaxDmg(integer ObjectID) : integer;

ObjectID ID of object to verify

Description

Retrieves the damage level of an object.

Returns

Damage value

Remarks

none

Seen In

abrain.abl

Example use

GuardObjectCF=getObjectMaxDmg(GuardObjective);

See also

GetObjectDamage, GetObjectDmgPts, SetObjectDamage

GetObjectDmgPts

Prototype

function GetObjectDmgPts(integer ObjectID) : integer;

ObjectID ID of object to be checked

Description

Retrieves the number of damage points of an object.

Returns

The damage points of the object.

Remarks

none

Seen In

not seen in any scripts

See also

GetObjectDamage, GetObjectMaxDmg, SetObjectDamage

SetObjectDamage

Prototype

SetObjectDamage(integer ObjectID, integer Value);

ObjectID	ID of object to be affected
Value	Amount of damage to set

Description

Sets the damage level of an object. Note that damage can only be given, not taken away.

Returns

no return value

Remarks

This function can be used to partially damage a building or object at the beginning of a mission. This allows doing things like partially used repair bays for example.

The damage value is handled as a percentile.

Seen In

1_2MXIN.ABI

Example use

setObjectDamage(RepairBay,50);

See also

GetObjectDamage, GetObjectMaxDmg, GetObjectDmgPts

GetObjectDamage

Prototype

function GetObjectDamage(integer ObjectID) : integer;

ObjectID ID of object to check

Description

Returns the current damage of an object.

Returns

The amount of damage as a percentile.

Remarks

none

Seen In

1_4STR.abi

Example use

if (getObjectDamage(CB1) > 0) then

See also

GetObjectMaxDmg, GetObjectDmgPts, SetObjectDamage

GetObjectActive

Prototype

function GetObjectActive(integer ObjectID) : integer;

ObjectID ID of object to check

Description

Retrieves the state of the active flag of an object.

Returns

Number of active objects

Remarks

If the object ID given is a group ID, then this function will return a count of all active objects in that group.

Seen In

Catk05011.abl

Example use

```
if (getObjectActive(PLAYER_LANCE0) == 0) then
```

See also

SetObjectActive

SetPilotWounds

Prototype

function SetPilotWounds(integer ObjectID, integer Wound);

ObjectID	ID of object whose pilot is to be affected
Wound	Amount of wounds to set

Description

Sets the amount of wounds that a pilot has.

Returns

no return value

Remarks

Maximum wound value is 6. Values above this are capped to 6.

The object ID given must be of an object with a pilot, as the code will retrieve the pilot of the given object.

Though untested, it seems that a pilot could also be healed via this function.

Interestingly, the Wound value is checked and capped at 6 but never checked or capped for values below 1. It is unknown what effect a value of 0 or a negative value will do (it is presumed a value of 0 will kill the pilot).

Seen In

not seen in any scripts

See also

GetPilotWounds

GetPilotWounds

Prototype

function GetPilotWounds(integer ObjectID) : real;

ObjectID ID of unit whose pilot is to be checked

Description

Retrieves the amount of wounds a unit's pilot has.

Returns

Wound value

Remarks

The maximum wound value appears to be 6.

The ObjectID must be that of a unit that has a pilot, otherwise the function merely returns a value of 0.

Note that the wound value is returned as a real, even thought the SetPilotWounds function uses an integer.

Seen In

not seen in any scripts

See also

SetPilotWounds

GetPilotId

Prototype

function GetPilotId(integer UnitID) : integer;

UnitID ID of unit to check

Description

Retrieves the ID of the pilot of a given unit.

Returns

ID of the pilot

Remarks

The unit ID must be a “mover” unit. That is, a mech or vehicle unit as these are the only ones with a pilot.

Seen In

not seen in any scripts

GetMaxArmor

Prototype

function GetMaxArmor(integer ObjectID) : integer;

ObjectID ID of object to check

Description

Retrieves the maximum armor point value of an object

Returns

Armor value

Remarks

It is unknown what the valid value range for a unit's armor is.

The object ID must be that of a mover object (mech or vehicle).

Seen In

not seen in any scripts

See also

GetArmorPts

GetArmorPts

Prototype

function GetArmorPts(integer ObjectID) : integer;

ObjectID ID of object to be checked

Description

Get the current armor points of an object

Returns

Armor value

Remarks

It is unknown what the valid range for armor points are.

The object ID must be that of a mover object (mech or vehicle).

Seen In

not seen in any scripts

See also

GetMaxArmor

SetCurrentBRValue

Prototype

function SetCurrentBRValue(integer UnitID, integer BRValue);

UnitID	ID of unit to affect
BRValue	Battle rating value to set.

Description

Sets the current battle rating of a unit.

Returns

no return value

Remarks

It is unknown what the valid range of values for a battle rating is.

It is unknown what effect this will have on a unit.

The function does not appear to check the object type passed in.

Seen In

not seen in any scripts

See also

GetCurrentBRValue

GetCurrentBRValue

Prototype

function GetCurrentBRValue(integer UnitID) : integer;

UnitID ID of unit to be checked

Description

Gets the current battle rating of a unit.

Returns

battle rating value

Remarks

It is unknown what range of values a battle rating can take.

The function does not appear to check the type of object passed in.

Seen In

abrain.abl

Example use

myBR = getCurrentBRValue(CUR_OBJECT_ID);

See also

SetCurrentBRValue

GetSensorsWorking

Prototype

function GetSensorsWorking(integer ObjectID) : integer;

ObjectID Id of object to check

Description

Checks if the sensors of a unit are working or not.

Returns

TRUE if working, FALSE otherwise

Remarks

The function seems to return a boolean like value, though it is returned in an integer.

Seen In

not seen in any scripts

InArea

Prototype

function InArea(integer UnitID, real[3] Point, real Radius, integer Min) : boolean;

UnitID	ID of unit, side or team to check
Point	Array of 3 reals describing map coordinates
Radius	Radius or area to check
Min	Minimum number needed to pass the check. -1 means all must be in.

Description

Check if a unit, side or team is in a given area defined by the map position and radius.

Returns

TRUE if unit, team, or side is in the given area, FALSE otherwise

Remarks

This function can check for a single unit, multiple units, or even a full team. In the case of a team, the Min value can specify the minimum number of units in the team/group that must be in the area for the check to return a value of TRUE.

Giving a Min value of -1 indicates that all units in the group must be in the area for the check to pass. If the ID is that of a single unit or object, then the Min value has no effect.

Seen In

miscfunc.abx

Example use

```
return(inArea(Unit, Point, Radius, -1));
```

GetWeaponAmmo

Prototype

function GetWeaponAmmo(integer UnitID, integer WeaponID) : integer;

UnitID	ID of unit to check
WeaponID	ID of weapon part to be checked

Description

Returns the number of rounds left for a given weapon

Returns

Ammo count of weapon

Remarks

none

Seen In

not seen in any scripts

GetVehiclePartId

Prototype

function GetVehiclePartId(integer UnitID, integer Part) : integer;

UnitID	ID of unit to check (should be a vehicle)
Part	Part to check (?)

Description

Gets the ID of a vehicle part (?)

Returns

Vehicle part ID

Remarks

Not really sure what exactly this function does. It appears to return the ID of a vehicle part on a given vehicle, but the code is unclear.

Seen In

not seen in any scripts

GetTerrainObjectPartId

Prototype

function GetTerrainObjectPartId(integer Block, integer Vertex) : integer;

Block Map row of terrain object
Vertex Map column of terrain object

Description

Retrieves the ID of a terrain object.

Returns

ID of terrain object

Remarks

The Block and Vertex values to use for a given terrain object can be retrieved from the map editor.

Seen In

1_1INIT.ABI

Example use

```
CampHQ1 = getTerrainObjectPartId(15,162);
```

ObjectTypeId

Prototype

function ObjectTypeId(integer UnitID) : integer;

UnitID ID of unit to check

Description

Get a unit's Type.

Returns

Type ID code

Remarks

This function will return the type of the object given. That is, if the UnitID is of a mech, it returns a type code indicating a mech, if unit was a vehicle, it returns a type indicating a vehicle etc.

It is unknown what would happen if calling this function with a building.

The valid type ID codes are not known.

Seen In

orders.abx

Example use

```
VehicleType = objectTypeId(Me);
```

ObjectInWithdrawal

Prototype

function ObjectInWithdrawal(integer UnitID) : integer;

UnitID ID of unit or group to check

Description

Checks if a unit or group is currently withdrawing.

Returns

TRUE if in withdrawal, FALSE otherwise

Remarks

The return value appears to simply be a TRUE/FALSE value, but the return type is an integer not a boolean.

The ID passed in can be that of a single unit or of a group.

Seen In

E3_0203.ABL

Example use

```
if (objectInWithdrawal(CLAN_VEHICLE1_STAR0) == 1) then
```

See also

OrderWithdraw

SetObjectActive

Prototype

function SetObjectActive(integer ObjectID, boolean Active) : integer;

ObjectID	ID of object (or group) to affect
Active	State to set

Description

Turn on/off an object's AI.
(Sets the active state flag of an object)

Returns

Number of units activated/deactivated..

Remarks

Allows setting the active flag of an object to either TRUE or FALSE, effectively turning the object's AI on (TRUE) or off (FALSE). When an object's AI is turned off, the game will no longer call the “brain” module for that unit's AI.

The unit ID may also be a group.

Seen In

1_1MXST.ABI

Example use

```
setObjectActive(getterrainobjectpartid(14,107),FALSE);
```

See also

GetObjectActive

PlayBetty

Prototype

function PlayBetty(integer BettyClip) : integer;

BettyClip Audio clip number to play

Description

Plays one of Betty's many, many little sayings. Such as "MechWarriors, prepare for combat", and of course her greatest hit "Enemy Components Captured!".

Returns

Result (values unknown)

Remarks

The return value of this function is not known, but likely an error code of some kind.

The following table gives the valid values for the clip number.

Clip	Saying
0	Mission drop weight exceeded.
1	Warning: Enemy has air superiority.
2	Enemy components captured.
3	Mission objective cannot be completed.
4	Mission objective complete.
5	New leader selected.
6	Enemy resources captured.
7	Mission critical : Mission time will expire in 30 seconds.
8	Mission time will expire in 2 minutes.
9	Perimeter breached
10	Commendation for low drop weight.
11	Mission failed.
12	Primary objective one : Complete
13	Primary objective two : Complete
14	Primary objective three : Complete
15	Secondary objective one : Complete
16	Secondary objective two : Complete
17	Secondary objective three : Complete

Clip	Saying
18	Mission successful
19	Commencing deployment. MechWarriors, prepare for combat.
20	Repair complete
21	Repair incomplete
22	Artillery standing by
23	Sensor probe available
24	Camera drone available
25	Command interface initiated
26	Incoming transmission
27	Forces do not meet mission parameters.
28	New equipment available.
29	New component available.
30	New battlemech available.
31	New pilot available.
32	New MechWarriors available.
33	New vehicle available.
34	Sensor tower reports contact.
35	Sensor tower destroyed.
36	Perimeter alarm destroyed.
37	Secondary objective cannot be completed.

Seen In

Achilles_MP.abi

Example use

```
playBetty(BETTY_CANNOT_COMP_OBJ);
```

See also

PlayWaveFile, PlaySpeech, PlaySoundEffect, PlayDigitalMusic, StopMusic

PlaySpeech

Prototype

function PlaySpeech(integer UnitID, integer Clip) : integer;

UnitID	ID of unit to say speech
Clip	Audio clip number to say

Description

Play a MechWarrior's audio speech clip.

Returns

returns 0

Remarks

The function does not appear to return any value other than 0 regardless of the success or failure of the function.

The list of valid speech clips a warrior can have is not known.

Seen In

pbrain.abl

Example use

```
playSpeech(CUR_OBJECT_ID,RADIO_UNDER_AIRSTRIKE);
```

See also

PlayWaveFile, PlayBetty, PlaySoundEffect, PlayDigitalMusic, StopMusic

SetRadio

Prototype

function SetRadio(integer WarriorID, boolean RadioState) : integer;

WarriorID	ID of the warrior pilot to affect
RadioState	State of the radio to set

Description

Turns a MechWarrior pilot's radio on or off.

Returns

none

Remarks

The radio chatter is all the radio messages that MechWarriors say during a mission such as “Beast here... Oh, oh. New contact.” and the like. Setting the RadioState to FALSE will turn these messages off, while setting it to TRUE will turn it on again.

Since the player can only ever hear radio chatter from their own team, this function has no real use with enemy units.

Seen In

E3_0101.abl

Example use

The following will turn off radio chatter for the player's mech or vehicle in the first slot of the first drop zone.

```
SetRadio(GetPilotID(GetVehicleID(PLAYER_FORCE,0,0)), FALSE);
```

PlayVideo

Prototype

function PlayVideo(integer VideoClip) : integer;

VideoClip Video clip number to play

Description

Starts playback of a MechWarrior video clip (as seen on the TAC map in game).

Returns

returns 0

Remarks

The valid range of values for the video clip is unknown.

This function appears to play one of the miniature video clips on the TAC map. Such as when a MechWarrior sends a radio message etc. These videos have no audio, so any audio would need to be played immediately before or after the call to this function.

This function appears to always return 0.

Seen In

not seen in any scripts

See also

PlaySmacker

PlaySoundEffect

Prototype

function PlaySoundEffect(integer SfxClip) : integer;

SfxClip Sound effect audio clip number to play

Description

Plays a sound effect audio clip.

Returns

returns 0

Remarks

The list of valid sound effect clips that can be used is unknown.
(values 0-81 appear to be usable, see the text file SOUND.SND)

The return value appears to always be 0, regardless of the success of this function.

Seen In

Achilles.abl

Example use

```
playSoundEffect(PERIMETER_ALARM_SOUND);
```

See also

PlayWaveFile, PlayBetty, PlaySpeech, PlayDigitalMusic, StopMusic

StopMusic

Prototype

function StopMusic : integer;

Description

Stops playback of the background music.

Returns

returns 0.

Remarks

This function always appears to return 0.

Seen In

not seen in any scripts

See also

PlayWaveFile, PlayBetty, PlaySpeech, PlaySoundEffect, PlayDigitalMusic

PlayDigitalMusic

Prototype

function PlayDigitalMusic(integer MusicNum) : integer;

MusicNum Music number to play

Description

Starts playback of background music.

Returns

returns 0

Remarks

The return value always appears to be 0.

The list of music files seems to be defined in the text file SOUND.SND, though the music entries 25-48 get remapped to 0-23 if in a Cermak map. Music file #24 does not exist and should not be used.

Values larger than 24 can be used with this function, though as the files are remapped when on a Cermak map, the Port Arthur music cannot be accessed. While the Cermak music can be if using a Port Arthur map.

The remapping is done internally and is dependent on the “Setting” value defined in the mission's .FIT file.

Seen In

Achilles_MP.abi

Example use

```
PlayDigitalMusic(VICTORY_MUSIC);
```

See also

PlayWaveFile, PlayBetty, PlaySpeech, PlaySoundEffect, StopMusic

CheckObjectiveType

Prototype

function CheckObjectiveType(integer Objective) : integer;

Objective Objective number to check

Description

Retrieve the type of a given objective.

Returns

Type code.

Remarks

The valid type code values this function returns is not known.

Seen In

not seen in any scripts

See also

CheckObjectiveStatus, CheckObjectiveTimer, SetObjectiveStatus, SetObjectiveTimer, SetObjectiveType

SetObjectiveType

Prototype

function SetObjectiveType(integer Objective, integer Type) : integer;

Objective	Objective to affect
Type	Objective type to set

Description

Sets the type of an objective.

Returns

result code.

Remarks

The return value of this function is not known. Though appears to be an error code.

The valid objective types are not known.

It is unknown what will happen if performing this function on an existing or completed objectives.

Seen In

not seen in any scripts

See also

CheckObjectiveStatus, CheckObjectiveTimer, CheckObjectiveType, SetObjectiveStatus, SetObjectiveTimer

CheckObjectiveStatus

Prototype

function CheckObjectiveStatus(integer Objective) : integer;

Objective Objective number to check

Description

Get the status of a mission objective.

Returns

Objective status code (incomplete, complete, failed).

Remarks

The objective number start counting at 0 and not 1. Therefore, the first objective is 0, the second is 1, etc.

Seen In

Achilles.abl

Example use

```
if (checkObjectiveStatus(0) == INCOMPLETE) then
```

See also

CheckObjectiveTimer, CheckObjectiveType, SetObjectiveStatus
SetObjectiveTimer, SetObjectiveType

SetObjectiveStatus

Prototype

function SetObjectiveStatus(integet Objective, integer State) : integer;

Objective	Objective to affect
State	State code to set

Description

Sets the status (complete, incomplete, failed) of an objective.

Returns

result code.

Remarks

The actual return value is not known, but appears to be the result code for the success of the function.

The objective numbers start counting at 0 not 1.

It is not known what happens if attempting to set the status of a non-existent objective.

Seen In

Achilles_MP.abi

Example use

```
setObjectiveStatus(0, FAILED);
```

See also

CheckObjectiveStatus, CheckObjectiveTimer, CheckObjectiveType, SetObjectiveTimer, SetObjectiveType

CheckObjectiveTimer

Prototype

function CheckObjectiveTimer(integer Objective) : real;

Objective Objective number to check

Description

Get the timer value for a mission objective.

Returns

Timer value, in seconds.

Remarks

The objective numbers starts at 0, not 1.

Seen In

not seen in any scripts

See also

CheckObjectiveStatus, CheckObjectiveType, SetObjectiveStatus, SetObjectiveTimer, SetObjectiveType

SetObjectiveTimer

Prototype

function SetObjectiveTimer(int Objective, real Time) : integer;

Objective	Objective to set
Time	Time value to set, in seconds

Description

Sets the timer value for an objective.

Returns

result code

Remarks

The return value is not known, though though appears to be a success/failure code.

Objectives start counting at 0, not 1.

Seen In

not seen in any scripts

See also

CheckObjectiveStatus, CheckObjectiveTimer, CheckObjectiveType, SetObjectiveStatus, SetObjectiveType

EndTimer

Prototype

function EndTimer(integer TimerID);

TimerID ID of the timer to end.

Description

Stops a timer and removes it from the internal timer list.

Returns

no return value

Remarks

This function stops and removes the timer identified by TimerID.

Only scenario timers can be used as TimerIDs. There are 8 scenario timers in all ranging in value from 7 to 14. If the timer ID given is not one of these values, this function does nothing.

Seen In

E3_0203.abl

Example use

endTimer(SCENARIO_TIMER_1);

See also

CheckTimer, SetTimer

CheckTimer

Prototype

function CheckTimer(integer TimerID) : real;

TimerID Id of the timer to check

Description

Gets the current time value, in seconds, of the given timer.

Returns

Time value of timer, in seconds.

Remarks

If the value of the timer is negative, this function returns 0.

Timers always count down, not up.

This function can handle up to 32 timers, ranging in value from 0 to 31, which is odd as the other timer functions only allow values 7 to 14 which is the range of the scenario timers.

Seen In

E3_0203.abl

Example use

```
if ((ExitTimerSet) AND (CheckTimer(EXIT_TIMER) == 0.0)) then
```

See also

EndTimer, SetTimer

SetTimer

Prototype

function SetTimer(integer TimerID, real Time) : integer;

TimerID	ID of timer to be set
Time	Time to set for this timer, in seconds.

Description

Sets the time for a timer and starts it ticking.

Returns

ID of timer created.

Remarks

The return value of this function is not clear. Merely seems to return the ID of the timer that was passed to it.

This function checks the TimerID value given. If it is not in the range of 7 to 14 (one of the scenario timers) this function does nothing and returns 0.

Seen In

Achilles_MP.abi

Example use

```
setTimer(EXIT_TIMER,2.0);
```

See also

CheckTimer, EndTimer

ObjectClass

Prototype

function ObjectClass(integer ObjectID) : integer;

ObjectID ID of object to check

Description

Get the class of object the given object is.

Returns

class ID of object

Remarks

A list of valid classes can be found in `misconst.abi` though there may be more class values than are listed there.

Seen In

abrain01.abl

Example use

```
if (objectClass(colliderId) <> BUILDING) then
```

ObjectCommander

Prototype

function ObjectCommander(integer ObjectID) : integer;

ObjectID ID of object to check

Description

Get the commander of an object

Returns

Commander of object

Remarks

The actual use of this function is still somewhat obscure, though it appears to be the Commander ID where 0 is the player side, 1 is the clan side, and 2 is the allies.

If the ObjectID given is a group, this function does nothing.

Seen In

Achilles_STR.abi

Example use

```
sendMessage(51,(((objectCommander(Artillery_Crate_014122000)+1)*10)+X));
```

ObjectSide

Prototype

function ObjectSide(integer ObjectID) : integer;

ObjectID ID of object to be checked

Description

Get which side (or team) an object belongs to.

Returns

Side/team code.

Remarks

The side/team codes can be found in `misconst.abi` and are defined as INNER_SPHERE, CLAN, and NEUTRAL.

Seen In

1_1MXST.abi

Example use

objectSide(TurCon1) <> TurCon1Alignment;

ObjectVisible

Prototype

function ObjectVisible(integer UnitID, integer ObjectID) : integer;

UnitID	ID of unit that is “looking”
ObjectID	ID of unit to check

Description

Check if the object is visible from the unit's perspective.

Returns

TRUE if object is visible, FALSE otherwise.

Remarks

Though this function returns a TRUE/FALSE value, the return type is in fact an integer.

If the UnitID provided is that of a group, this function will check if any member of the group can see the object.

Seen In

not seen in any scripts

ObjectStatusCount

Prototype

function ObjectStatusCount(integer UnitID, int[9] List);

UnitID	ID of unit to check
List	Array of integers to receive the status counts

Description

Get status counts of an object, unit or team.

Returns

no return value. Values are passed back through the List parameter.

Remarks

The purpose of this function remains a mystery. It is difficult to tell what a status count is or why you would want to know. The MechCommander 2 source does not shed any light on this either.

Seen In

miscfunc.abx

Example use

objectStatusCount(Unit, tallyList);

ObjectStatus

Prototype

function ObjectStatus(integer ObjectID) : integer;

ObjectID ID of object to be checked

Description

Gets the status of an object. (disabled, destroyed, etc)

Returns

The status of the object.

Remarks

The list of possible return values are not known.

Seen In

abrain01.abl

Example use

```
switch (objectStatus(issuedTarget))
```

ObjectExists

Prototype

function ObjectExists(integer ObjectID) : integer;

ObjectID ID of object to check.

Description

Checks if an object exists.

Returns

TRUE if object exists, FALSE otherwise.

Remarks

Though the return value is a TRUE/FALSE value, the return type is an integer.

This function appears to accept a group ID as well.

Seen In

not seen in any scripts

ObjectCreate

Prototype

function ObjectCreate(integer ObjectID) : integer;

ObjectID ID of object type to be created.

Description

Creates an object.

Returns

ID of object created

Remarks

Creates or allow an object to appear on the mission map that had been it's exist flag set to 0 in the .FIT file. (See guide part 3 for details)

Seen In

123_2CRE.abi

Example use

```
objectCreate(VehicleID(CLAN_STAR3,1,0));
```

ObjectSuicide

Prototype

function ObjectSuicide(integer UnitID);

UnitID ID of unit to be destroyed

Description

Cause an object to destroy itself.

Returns

no return value

Remarks

If a group ID is given, then the entire group will be destroyed.

Seen In

not seen in any scripts

DistanceToPosition

Prototype

function DistanceToPosition(integer UnitID, real[3] Point) : real;

UnitID	ID of unit or team to check
Point	Array of three reals containing the coordinates to check

Description

Retrieve the distance of a unit or team to a given map point.

Returns

Distance in meters.

Remarks

If the unit ID is a team, then the function will return the distance of the closest member of the team.

Seen In

1_1MXLP.abi

Example use

```
if (distanceToPosition(PLAYER_FORCE, aPoint) < 10000) then
```

See also

DistanceToObject

DistanceToObject

Prototype

function DistanceToObject(integer UnitID, integer ObjectID) : real;

UnitID	ID of unit or team to check distance for
ObjectID	ID of object to check distance to

Description

Calculate the distance of a team to a given object.

Returns

Distance in meters

Remarks

If the ID given is that of a team, the distance of the closest member is returned.

Seen In

3_5ACT.abi

Example use

```
if (distancetoobject(PLAYER_FORCE,VehicleID(CLAN_STAR4,0,0)) < 350) then
```

See also

DistanceToPosition

ObjectChangeSides

Prototype

function ObjectChangeSides(integer ObjectID, integer Side);

ObjectID	ID of object to change
Side	Side to assign to object

Description

Forces an object to be marked as belonging to a given side (IS, Clan or Allies).

Returns

no return value

Remarks

none

Seen In

1_1INIT.ABI

Example use

objectChangeSides(CampHQ1,CLAN);

FileExists

Prototype

function FileExists(char[] Name) : boolean;

Name String containing name of file to check.

Description

Checks for the existence of a file.

Returns

TRUE if the file exists, FALSE otherwise.

Remarks

This function appears to have been deprecated.

It is not known in which folder a file is checked, nor if it scans the FSTs as well.

It is not known if a path can be given with the file's name.

Seen In

not seen in any scripts

PlaySmacker

Prototype

function PlaySmacker(integer VideoNum) : integer;

VideoNum Index of the video to be played.

Description

Plays a smacker video file.

Returns

Unknown, likely an error code.

Remarks

It is not known if this plays a full screen video, or if it plays one of the briefing videos.

It is unknown where the list of valid video files is stored.

Seen In

not seen in any scripts

See also

PlayVideo

OrderTest

Prototype

function OrderTest(integer Unknown) : integer;

Unknown Unknown

Description

Orders current unit to perform a test (?)

Returns

Unknown

Remarks

It is not known what exactly this function does. May only be for debugging purposes.

Seen In

abrain01.abl

Example use

```
ordertest(10);
```

SetAttackRadius

Prototype

function SetAttackRadius(real Radius) : real;

Radius	The attack radius to set (in meters)

Description

Sets the radius for which the current unit will break off attack from its last target.

Returns

Previous attack radius (in meters).

Remarks

The function affects the current pilot.

Seen In

not seen in any scripts

DamageObject

Prototype

function DamageObject(integer TargetID, integer AttackerID,
integer WeaponID, real DmgPoints, integer HitLocation,
real HitRoll, real EntryAngle) : integer;

TargetID	ID of target object or group
AttackerID	ID of unit doing the attack
WeaponID	ID of weapon to use for the attack
DmgPoints	Amount of damage points to apply
HitLocation	Location of hit
HitRoll	Unknown. This parameter does not appear to be used.
EntryAngle	Angle at which the attack hit is coming from.

Description

Directly damages an object as if it were shot by a weapon.

Returns

0 = Success
-1 = Bad target
-2 = Bad attacker

Remarks

The attacker ID must be valid, but it does not need to have the weapon that is supposedly firing.

If the target ID given is that of a group, the entire group will be hit.

This function will work in multiplayer. The function checks for and acts appropriately for multiplayer games, though only the game server will activate the code properly.

Seen In

not seen in any scripts.

FireUponEnemyFireOnly

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

OpenFire

Prototype
function OpenFire;

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

Retreat

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

OrderWithdraw

Prototype

function OrderWithdraw : integer;

Description

Orders the current unit to withdraw.

Returns

0 = Success
-2 = Invalid warrior

Remarks

This function affects the current pilot.

There are actually two versions of this function. The first, as described here, is defined in the game EXE, the other is actually defined in the Orders library. Note that the library version takes a parameter that is the object to be ordered.

Seen In

CGD0000.abl

Example use

orderWithdraw;

See also

ObjectInWithdrawal

AttackPerOrders

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

AttackClosestTarget

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

AttackThreat

Prototype

```
function AttackThreat(integer Unknown1, integer Unknown2,
    integer Unknown3, boolean Unknown4, integer Unknown5,
    integer Unknown6) : integer;
```

Unknown1	Unknown
Unknown2	Unknown
Unknown3	Unknown
Unknown4	Unknown
Unknown5	Unknown
Unknown6	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

OrderAttackContact

Prototype

function OrderAttackContact(integer AtkType, integer AtkMethod,
integer AtkRange, integer Pursue) : integer;

AtkType	Type of attack to perform 0 = none 1 = destroy 2 = disable
AtkMethod	Method of attack 0 = ranged 1 = dfa (??) 2 = ramming
AtkRange	Range code to attack from 0 = short 1 = medium 2 = long
Pursue	Pursue the contact? 0 = No 1 = Yes

Description

Orders the current unit to direct all weapons fire at current contact.

Returns

1 = Success (or no errors)
0 = Tactical order failed.
-2 = Bad contact

Remarks

This function affects the current pilot.

Seen In

abrain01.abl

Example use

```
orderAttackContact(ATTACK_TO_DESTROY,ATTACK_RANGED,
                   FIRERANGE_MEDIUM, TRUE);
```

See also

OrderAttackObject

OrderAttackObject

Prototype

function OrderAttackObject(integer ObjectID, integer AtkType,
integer AtkMethod, integer AtkRange, boolean Pursue) : integer;

ObjectID	ID of object to attack
AtkType	Type of attack to perform 0 = none 1 = destroy 2 = disable
AtkMethod	Method of attack 0 = ranged 1 = dfa (??) 2 = ramming
AtkRange	Range code to attack from 0 = short 1 = medium 2 = long
Pursue	Pursue the contact? 0 = No 1 = Yes

Description

Orders the current unit to attack the given object.

Returns

1 = Success (or no errors)
0 = Tactical order failed.
-2 = Bad contact

Remarks

Unlike OrderAttackContact, the last parameter of this function **is** a boolean.

This function affects the current pilot.

Seen In

cat0000.abl

Example use

```
orderAttackObject(HKList[0],ATTACK_TO_DESTROY, ATTACK_RANGED,
                  FIRERANGE_OPTIMAL, TRUE);
```

See also

OrderAttackContact

OrderPowerUp

Prototype

function OrderPowerUp;

Description

Orders the current MechWarrior to power up their mech.

Returns

no return value

Remarks

This function uses the current mech/mechwarrior.

Seen In

dredambush01.abl

Example use

orderPowerUp;

See also

OrderPowerDown

OrderPowerDown

Prototype
function OrderPowerDown;

Description
Orders the current MechWarrior to power down their mech.

Returns
no return value

Remarks
This function uses the current mech/mechwarrior.

Seen In
dredambush01.abl

Example use
orderPowerDown;

See also
OrderPowerUp

OrderPatrolPath

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

OrderTraversePath

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

OrderMoveToContact

Prototype

function OrderMoveToContact(boolean Run) : integer;

Run Should the unit run?
TRUE = yes
FALSE = no

Description

Orders the current unit to move to its current contact.

Returns

0 = tactical order failed
1 = success

Remarks

This function uses the current unit as the unit that will receive the orders.

This function may be problematic as all calls to this function found in the official scripts have been commented out and replaced with OrderMoveToObject calls.

Seen In

orders.abi

Example call

orderMoveToContact(MoveSpeed);

See also

OrderMoveToObject, OrderMoveTo

OrderMoveToObject

Prototype

function OrderMoveToObject(integer ObjectID, boolean Run) : integer;

ObjectID	ID of object to move to
Run	Flag to indicate if should walk or run to object TRUE = run FALSE = walk

Description

Orders the current unit to move to the given object.

Returns

1 = success
0 = tactical order failed

Remarks

This function sends its orders to the current unit.

Seen In

orders.abi

Example use

```
orderMoveToObject(getContactId,MoveSpeed);
```

See also

OrderMoveToContact, OrderMoveTo

OrderMoveTo

Prototype

function OrderMoveTo(real[3] Point, boolean Run) : integer;

Point	An array of 3 reals containing the map coordinate to move to
Run	Should unit should walk or run. TRUE = run FALSE = walk

Description

Orders the current unit to move to the specified location.

Returns

0 = tactical order failed
1 = success

Remarks

This function places its orders on the current unit.

Seen In

abrain01.abl

Example use

```
orderMoveTo(StartPoint,true);
```

See also

OrderMoveToContact, OrderMoveToObject

OrderWait

Prototype

function OrderWait(real WaitTime, boolean ClearLastTarget) : integer;

WaitTime	Time, in seconds to wait
ClearLastTarget	Should unit clear its last target TRUE = yes FALSE = no

Description

Order the current unit to wait for a specified amount of time.

Returns

0 = tactical order failed
1 = success

Remarks

This function issues its orders to the current unit.

This function can also be used to clear the current unit's target (see example use below).

Seen In

cfunc.abi

Example use

The following call will clear any target the current unit has.

```
OrderWait(0.0, TRUE);
```

SetOrderMode

Prototype

function SetOrderMode(integer UnitID) : integer;

UnitID ID of unit to affect

Description

Sets a unit to order mode.

Returns

unknown, likely an error code.

Remarks

This function sets a unit into order mode (whatever that actually means). It is unclear if this is a toggle or always sets this mode.

Seen In

not seen in any script files

GetLastTacOrder

Prototype

function GetLastTacOrder(integer UnitID, @real OrderTime,
@integer[] OrderParams) : integer;

UnitID	ID of object to check
OrderTime	Time order was given
OrderParams	Order parameters (size varies by tactical order)

Description

Retrieves the last tactical order the current unit was issued. Also retrieves the time the order was given at, and the parameter list of the order.

Returns

Last tactical order code. Time order was issued (via function parameters) and the order's parameters (via the function parameters).

Remarks

It is not known what the valid values for tactical orders are.

The size of the OrderParams array is not known, though it is dependent on the actual tactical order.

A value of -1 for the unit ID appears to make the function use the current unit.

Seen In

sbrain01.abl

Example use

```
lastTacCode = getLastTacOrder(-1, time, tacOrderParams);
```

See also

GetTacOrder

GetTacOrder

Prototype

```
function GetTacOrder(integer UnitID, @real OrderTime,
                     @integer[] OrderParams) : integer;
```

UnitID	ID of object to check
OrderTime	Time order was given
OrderParams	Order parameters (size varies by tactical order)

Description

Retrieves the current tactical order the current unit was issued. Also retrieves the time the order was given at, and the parameter list of the order.

Returns

Tactical order code. Time order was issued (via function parameters) and the order's parameters (via the function parameters).

Remarks

It is not known what the valid values for tactical orders are.

The size of the OrderParams array is not known, though it is dependent on the actual tactical order.

A value of -1 for the unit ID appears to make the function use the current unit.

Seen In

sbrain01.abl

Example use

```
curTacCode = getTacOrder(-1, time, tacOrderParams);
```

See also

GetLastTacOrder

GetUnitMates

Prototype

function GetUnitMates(integer UnitID, @integer[12] List) : integer;

UnitID	ID of unit or group to get
List	Array of integers to receive the list of mates

Description

Retrieves the list of mates for a given unit.

Returns

Number of mates in the list, the list itself is returned through the parameter list and should receive the IDs of all the mates.

Remarks

The maximum size of the list is believed to be 12.

The actual list of mates can be found in the group definitions in the mission's .FIT file.

Seen In

mis0103.abl

Example use

```
numforce = getunitmates(player_force,teamlist);
```

GetVisualRange

Prototype

function GetVisualRange(integer UnitID) : real;

UnitID ID of unit to check (-1 for self)

Description

Gets the visual range of a unit.

Returns

Distance, in meters, a unit can see visually. Also takes into account the Beagle Probe.

Remarks

It seems that if the unit ID is set to -1, the function uses the current unit as the unit to check.

Seen In

orders.abi

Example use

```
Range = getVisualRange(CUR_OBJECT_ID);
```

FireWeapon

Prototype

function FireWeapon(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

TimeToImpact

Prototype

function TimeToImpact(integer Unknown1, integer Unknown2) : real;

Unknown1	Unknown
Unknown2	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SortWeapons

Prototype

```
function SortWeapon(@integer[] WeaponList, integer ListSize,
                    integer SortType);
```

WeaponList	List of weapon IDs to sort
ListSize	Size of weapon list
SortType	Type of sort to perform

Description

Sorts the weapons on the current mech (?)

Returns

Unknown

Remarks

What this function actually does is not known, nor why it is needed.

Seen In

not seen in any scripts

HasMovePath

Prototype

function HasMovePath : boolean;

Description

Checks if the current unit has a movement path.

Returns

TRUE is unit has a movement path, FALSE otherwise.

Remarks

This function checks the current unit.

Seen In

not seen in any scripts

HasMoveGoal

Prototype

function HasMoveGoal : boolean;

Description

Checks if the current unit has a movement goal.

Returns

TRUE if unit has a movement goal, FALSE otherwise.

Remarks

This function performs its checks on the current unit.

Seen In

not seen in any scripts

StartMovePath

Prototype

```
function StartMovePath(integer Unknown1, integer Unknown2,
                       integer Unknown3) : integer;
```

Unknown1	Unknown
Unknown2	Unknown
Unknown3	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

StartFriendlyScan

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

StartEnemyScan

Prototype

function StartEnemyScan(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

StartVehicleScan

Prototype

```
function StartVehicleScan : integer;
```

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

StartFieldScan

Prototype

function StartFieldScan(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SetRealMemory

Prototype

function SetRealMemory(integer MemoryCell, real Value);

MemoryCell	Brain cell memory to set
Value	Value to set

Description

Sets a brain cell variable with a real value

Returns

no return value

Remarks

This function sets what is known as a brain cell. Brain cells are used in the game for dealing with AI and such.

Seen In

abrain01.abl

Example use

setRealMemory(MI_COORD_X, StartPoint[0]);

See also

GetIntegerMemory, GetRealMemory, SetIntegerMemory

SetIntegerMemory

Prototype

function SetIntegerMemory(integer MemoryCell, integer Value);

MemoryCell	Brain cell to be set
Value	Value to set

Description

Sets a brain cell variable with an integer value

Returns

no return value

Remarks

This function sets what is known as a brain cell. Brain cells are used in the game for dealing with AI and such.

Seen In

abrain01.abl

Example use

setIntegerMemory(MI_MODE, COMBAT_MODE_GUARD);

See also

GetIntegerMemory, GetRealMemory, SetRealMemory

SetUpdateTime

Prototype

function SetUpdateTime(real Unknown);

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SetMoveGoal

Prototype

function SetMoveGoal(integer UnitID, real[3] Point);

UnitID	ID of unit
Point	Array of three reals containing the map coordinate of the goal

Description

Sets the movement goal of a unit to a given location.

Returns

no return value

Remarks

Seems to set the destination point for a given unit, but this is not clear.

The parameters to this function need to be rechecked.

Seen In

not seen in any scripts

SetPhase

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

SetAction

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

SetMode

Prototype

function SetMode(integer Unknown);

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SetChallenger

Prototype

function SetChallenger(integer UnitID, integer ChallengerID) : integer;

UnitID	ID of unit to be challanged (victim)
ChallengerID	ID of unit to be challenger (challenger)

Description

Make a unit challenge another unit.

Returns

0 = No error
-1 = ChallengerID cannot be a group ID
-2 = Invalid UnitID or Unit ID is not a mover (mech or vehicle)

Remarks

The effect of having one unit challenge another is not known.

Seen In

abrain01.abl

Example use

```
setChallenger(targetId, CUR_OBJECT_ID);
```

See also

GetChallenger

GetAttackerInfo

Prototype

function GetAttackerInfo(integer UnitID) : real;

UnitID ID of unit to be checked

Description

Returns the time, in seconds, since the given unit last attack the current pilot.

Returns

Time since unit last attacked current pilot, -1.0 if never.

Remarks

This function uses the current pilot.

If the unit given has ever attacked the current pilot, this function returns the time, in seconds, since the last attack. If the unit has never attacked the current pilot, the time returned is -1.

Seen In

ebrain01.abl

Example use

```
firing = (getAttackerInfo(contactId) <= 5.0);
```

GetAttackers

Prototype

function GetAttackers(@integer[] List, real TimeFrame) : integer;

List	An array of integers to receive the attackers.
TimeFrame	Time frame in seconds to check

Description

Gets all attackers of the current pilot within the last time frame given (i.e. the number of attackers in the last 'x' seconds).

Returns

Number of attackers, the list of attackers itself is passed back through the List parameter.

Remarks

This function uses the currently selected pilot.

The size of the List array is not known, but the comment in the MechCommander 2 source code says “make it big!”. As it is possible to have 40 or more enemy units on a map at a time, and they could all theoretically have attacked the same target within the given time frame, it is easy to see why that comment has been made.

Seen In

ebrain01.abl

Example use

```
numAtts = getAttackers(attackers, 3.0);
```

GetFireRanges

Prototype

function GetFireRanges(@real[4] Ranges);

Ranges Array of reals to receive the list of ranges.

Description

Get the current settings for short, medium and long weapon ranges, as well as the max weapon range.

Returns

no return value

Remarks

Though untested, it is believed that the values returned are the minimum and maximum ranges as given by all the weapons on the current unit.

This function uses the currently selected unit.

Seen In

abrain01.abl

Example use

getFireRanges(FireRange);

GetTimeWithoutOrders

Prototype

function GetTimeWithoutOrders : real;

Description

Get the amount of time that the current unit has been without orders.

Returns

Time, in seconds, since last orders.

Remarks

It is not known if this value is the amount of time passed since the unit last had orders, or if it is a time stamp from when the unit finished its last order.

This function uses the currently selected unit.

Seen In

sbrain01.abl

Example use

time = getTimeWithoutOrders;

GetChallenger

Prototype

function GetChallenger(integer UnitID) : integer;

UnitID ID of unit to check

Description

Retrieves the ID of the unit that is challenging the given unit.

Returns

Unit ID of challenger.

Remarks

none

Seen In

abrain01.abl

Example use

```
if (getChallenger(contactId) == 0) then
```

See also

SetChallenger

GetAlarmTriggers

Prototype

function GetAlarmTriggers(@integer[] Triggers) : integer;

Triggers Array of integers to receive the list of triggers

Description

Get the number of alarm triggers and the list of trigger IDs

Returns

Number of triggers in the list. The trigger IDs are returned through the Triggers parameter.

Remarks

It is not known what the maximum size of the trigger list is.

Seen In

abrain01.abl

Example use

```
numTriggers = getAlarmTriggers(triggers);
```

GetRealMemory

Prototype

function GetRealMemory(integer MemoryCell) : real;

MemoryCell Brain cell to read from

Description

Retrieves the value of a brain memory cell containing a real value.

Returns

Value retrieved from brain cell

Remarks

Brain cells are used for controlling unit AIs and such.

Seen In

abrain01.abl

Example use

```
GuardEngageRadius = getRealMemory(MI_ENGAGE_RADIUS);
```

See also

GetIntegerMemory, SetIntegerMemory, SetRealMemory

GetIntegerMemory

Prototype

function GetIntegerMemory(integer MemoryCell) : integer;

MemoryCell Brain cell to read from

Description

retrieves the value of a brain memory cell that contains an integer value.

Returns

Value retrieved from brain cell

Remarks

Brain cells are used for controlling unit AIs and such.

Seen In

abrain01.abl

Example use

issuedTarget = getIntegerMemory(MI_OBJECT);

See also

GetRealMemory, SetIntegerMemory, SetRealMemory

GetObjectPosition

Prototype

function GetObjectPosition(int ObjectID, @real[3] Point) : integer;

ObjectID	ID of object to check
Point	Array of three reals to receive the coordinates

Description

Get the map position of the given object.

Returns

no return value. The coordinates of the object are passed back through the Point array.

Remarks

It is not known if any object (such as mechs and vehicles) can also be gotten in this way. Though the code does not appear to check for any special object types.

Seen In

1_1INIT.ABI

Example use

```
getobjectposition(x,aPoint);
```

GetWeaponRanges

Prototype

function GetWeaponRanges(int UnitID, @real[3] Ranges);

UnitID	ID of unit to check
Ranges	Array of reals to receive the list of ranges

Description

Gets the list of ranges, in meters, of a unit's shortest, optimal and longest weapon ranges.

Returns

no return value. The list of ranges is passed back through the Ranges array.

Remarks

If the unit is not a mover (mech or vehicle) then the ranges are always returned as 0.0.

Seen In

orders.abx

Example use

```
getWeaponRanges(CUR_OBJECT_ID,myWeaponRanges);
```

GetWeaponShots

Prototype

function GetWeaponShots(integer WeaponIdx) : integer;

WeaponIdx	Index of weapon to check

Description

Returns the number of shots (ammo count?) the given weapon index has for the current unit.

Returns

Number of shots.

Remarks

It is uncertain is a weapon index is the same as a weapon ID, though it appears likely.

This function uses the current unit.

Seen In

not seen in any scripts

GetWeaponsInRange

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

GetWeaponsLocked

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

GetWeaponsReady

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

SetTarget

Prototype

function SetTarget(integer UnitID, integer TargetID);

UnitID	ID of unit to be set
TargetID	ID of unit to be targeted

Description

Sets a unit's target

Returns

no return value

Remarks

TargetID must be a single unit, not a group.

If the UnitID set is a group ID, then all units in the group will have there targets set to the TargetID.

There is a second version of this function defined in orders.abi that takes a target unit ID and a real.

Seen In

orders.abx

Example use

```
SetTarget(CUR_OBJECT_ID,TargetID);
```

See Also

GetTarget

GetTarget

Prototype

function GetTarget(integer UnitID) : integer;

UnitID ID of unit to check

Description

Retrieve the ID of a unit's target.

Returns

ID of target

Remarks

If the UnitID given is that of a group, this function merely returns 0.

UnitID given must contain a pilot. If not, this function will force an error to occur with the message "execHbGetTarget:No pilot in mover!".

Seen In

abrain01.abl

Example use

curTarget = getTarget(CUR_OBJECT_ID);

See Also

SetTarget

GetGuardDistanceTo

Prototype

function GetGuardDistanceTo(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

GetGuardRadii

Prototype

function GetGuardRadii(real Unknown1, real Unknown2) : integer;

Unknown1	Unknown
Unknown2	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

GetGuardPoint

Prototype

function GetGuardPoint(real[3] Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

GetGuardObjective

Prototype

function GetGuardObjective : integer;

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SetGuardRadii

Prototype

function SetGuardRadii(real Unknown1, real Unknown2) : integer;

Unknown1	Unknown
Unknown2	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SetGuardPoint

Prototype

function SetGuardPoint(real[3] Point) : integer;

Point	Point to be set

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

SetGuardObjective

Prototype

function SetGuardObjective(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

GetContactRelativePosition

Prototype

function GetContactRelativePosition(@real Range, @real Angle) : integer;

Range	Distance from contact (in meters)
Angle	Angle from contact (in degrees)

Description

Get the range and angle of the current unit to its current contact.

Returns

0 = success
1 = invalid contact or invalid current object

The range and angle are passed back through the parameters.

Remarks

This function uses the currently selected unit.

Seen In

abrain01.abl

Example use

```
getContactRelativePosition(range, angle);
```

GetContactStatus

Prototype

function GetContactStatus(@integer TagState) : integer;

TagState	Tag state (if any)

Description

Get the status of the current contact.

Returns

Status of the contact

Remarks

This function uses the currently selected unit.

The status appears to be the type of sensor contact that is detected. In the example code from abrain01.abl below, the return value checked for a little after the example use line is SENSOR_TRACE.

The TagState parameter always seems to be set to 0.

Seen In

abrain01.abl

Example use

```
contactStatus = getContactStatus(contactTag);
```

IsContact

Prototype

```
function IsContact(integer UnitID, integer Criteria,
                   integer Select) : integer;
```

UnitID	ID of unit to check
Criteria	Unknown
Select	Unknown

Description

Checks if a unit is a sensor contact. (?)

Returns

Handle to the contact. Or 0 if no contact.

Remarks

This function uses the currently selected unit.

It is not quite known what this function actually does. The name suggests that it should say if the unit is a contact, but the return value appears to actually be a kind of handle.

The Criteria parameter is unknown

The Select parameter does not appear to be used by the function.

Seen In

abrain01.abl

Example use

```
targetContactHandle = isContact(colliderId, CT_ENEMY
                                + CT_NOT_CHALLENGED, TRUE);
```

GetContactId

Prototype

function GetContactId : integer;

Description

Get the object ID of the current contact.

Returns

ID of the contact

Remarks

This function uses the current unit.

Seen In

abrain01.abl

Example use

contactId = getContactId;

SelectContact

Prototype

function SelectContact(integer Type, integer ContactHandle) : integer;

Type	Unknown, not used in function
ContactHandle	Handle to a contact

Description

Sets the specified contact as the current contact. The contact handle must be in the current units contact list.

Returns

0 = success
1 = not a contact
-1 = current unit is not a mover (mech or vehicle)

Remarks

The first parameter of this function does not appear to be used within the function.

The function uses the current unit.

Seen In

abrain01.abl

Example use

```
selectContact(1, contactHandle);
```

GetEnemyCount

Prototype

function GetEnemyCount(integer UnitID) : integer;

UnitID ID of unit to check

Description

Count the number of enemies sighted by the given unit. If the unit is a team or group, then it counts the number sighted by all team members.

Returns

Enemy count
-1 = given unit ID is invalid

Remarks

none

Seen In

not seen in any scripts

GetContacts

Prototype

```
function GetContacts(@integer[] List, integer Criteria,
                     integer SortType) : integer;
```

List	List of contacts
Criteria	Unknown
SortType	Unknown

Description

Gets a sorted list of sensor contacts for the current unit's pilot.

Returns

Number of contacts in the list. The actual list is passed back through the first parameter.

Remarks

This function uses the current unit.

The sort type and selection criteria values are not known.

The maximum size of List is unknown.

Seen In

abrain01.abl

Example use

```
numTargets = getContacts(enemyList, CT_ENEMY +
                         CT_LOS + CT_NOT_CHALLENGED, CS_CV);
```

GetWarriorStatus

Prototype

function GetWarriorStatus(integer WarriorID) : integer;

WarriorID	ID of the warrior to check

Description

Retrieves the status of a MechWarrior.

Returns

The status value
-1 = invalid warrior ID

Remarks

It is unknown what the return value actually represents.

Seen In

not seen in any scripts

SelectWarrior

Prototype

function SelectWarrior(integer WarriorID) : integer;

WarriorID ID of the warrior

Description

Set the given warrior ID as the current pilot.

Returns

Previous pilot's warrior ID.
-1 = not a pilot

Remarks

This sets the given pilot ID as the currently selected pilot. All functions that reference the current pilot will now be referencing this new pilot.

Seen In

not seen in any scripts

SelectUnit

Prototype

function SelectUnit(integer UnitID) : integer;

UnitID ID of unit to be selected

Description

Sets the given unit as the currently selected unit.

Returns

Previous unit ID
-1 = not an object

Remarks

This sets the currently selected unit to that given by UnitID. After this, all functions that reference the current unit will be referring to this unit.

Seen In

not seen in any scripts

SelectObject

Prototype

function SelectObject(integer ObjectID) : integer;

ObjectID	ID of object to be selected

Description

Set the given object as the current object.

Returns

Previous current object
-1 = not an object

Remarks

After this function, all functions that reference the current object will refer to this object given here.

Seen In

not seen in any scripts

GetTimeLeft

Prototype

function GetTimeLeft : real;

Description

Get the scenario time remaining.

Returns

time remaining in seconds.
-1 = infinite time (the scenario doesn't have a time limit)

Remarks

This function actually return the difference between the scenario time and the internal time variable actualTime. If the value would go negative, it returns 0.

Seen In

Achilles.abl

Example use

```
if (gettimeleft == 0.0) then
```

See also

GetTime

GetTime

Prototype

function GetTime : real;

Description

Gets the current game time.

Returns

Time value in seconds

Remarks

This function returns the value of the internal variable actualTime. The time returned increases from 0 from the start of the mission scenario.

Seen In

abrain01.abl

Example use

attackerTimeList[numAttackers] = getTime;

See also

GetTimeLeft

GetId

Prototype
    function GetId : integer;

Description
    Retrieves the ID of the current object.

Returns
    ID of selected object.
    0 = current object is not valid

Remarks
    none

Seen In
    abrain01.abl

Example use
    Me = getId;

GetPhase

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

GetAction

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

GetMode

Prototype
Unknown

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

Assert

Prototype

function Assert(boolean Test, integer ErrCode, char[] ErrMsg);

Test	Boolean value to test
ErrCode	Error code to use
ErrMsg	Error message

Description

Verifies if a condition is valid. If the test is false, then an error is generated using the error code and error message provided.

Returns

no return value if Test is TRUE, this function does not return if Test is FALSE.

Remarks

This function is for debugging purposes. Use to ensure that certain values do not become illegal.

If the condition provided is false, this function internally calls Fatal.

Example use

```
assert((CurrentGO <> -1),86,"Design Error.  All guard objects dead.");
```

See also

Fatal

Fatal

Prototype

function Fatal(integer ErrCode, char[] ErrMsg);

Description

Stops the program and generates an error condition with the given error code and message.

Returns

This function never returns.

Remarks

This function is for debugging purposes. The function generates the error and terminates the program.

Example use

Fatal(86,"Bad Unit Passed to NumCaptureDeadorFled");

See also

Assert

SetMaxLoops

Prototype

function SetMaxLoops(integer MaxValue);

MaxValue New maximum value to set

Description

Sets the maximum number of iterations the game will allow a loop to perform.

Returns

no return value

Remarks

This function will set the game's internal variable MaxLoopIterations. This variable is used by the language interpreter to set the number of loop iterations the game will allow before it decides that the loop is infinite and aborts with an error.

It is likely safest not to use this function.

The default value for MaxLoopIterations is 100001

Seen In

never seen in any scripts

SetModuleName

Prototype

function SetModuleName(char[] Name);

Name Name to set the module to

Description

Sets the name of the module

Returns

no return value

Remarks

This function appears to have been disabled. The entire execution of this function appears to merely ignore all the parameters and return immediately.

See also

GetModuleName

GetModuleName

Prototype

function GetModuleName : char[];

Description

Get the name of the module.

Returns

NULL

Remarks

This function has been completely disabled and does nothing whatsoever.

Here is a disassembly of this function's execution from within the game.

```
.text:0062F550 struct _Type * __cdecl execStdGetModName(struct _SymTableNode *):
.text:0062F550                 xor     eax, eax
.text:0062F552                 retn
```

As can be seen, the function merely ends up returning 0.

See also

SetModuleName

GetModuleHandle

Prototype

```
function GetModuleHandle : integer;
```

Description

Get a handle to the module.

Returns

Handle to the module

Remarks

This function appears to have been deprecated

Trunc

Prototype

function Trunc(real Value) : integer;

Value	Floating point value to truncate

Description

Converts a floating point value to an integer by simply truncating (removing) all decimal values.

Returns

Value as an integer

Remarks

This function uses the floating point processor.

Example use

```
IntVal = Trunc(3.14159265);
```

See also

Round

Sqrt

Prototype

function Sqrt(real Number) : real;

Number Number whose square root is to be calculated

Description

Calculates the square root of a number

Returns

Square root of the number

Remarks

This is one of the few functions that specifically checks for and allows it's parameter to be either a real or integer value.

This function uses the floating point processor

Round

Prototype

function Round(real Number) : real;

Number Number to be rounded

Description

Converts a real value into an integer value. The number is first rounded up.

Returns

Rounded number.

Remarks

This function adds 0.5 to the number provided, then internally calls Trunc.

This function uses the floating point processor.

See also

Trunc

Random

Prototype

function Random(integer Range) : integer;

Range Range of values to randomly choose from.

Description

Gets a random number.

Returns

Random integer number.

Remarks

The value returned is in the range of 0 to (Range-1).

Abs

Prototype

function Abs(real Number) : real;

Number Number whose absolute value is to been known

Description

Get the absolute value of a number.

Returns

Absolute value of the number

Remarks

This function only works with real values.

This function uses the floating point processor.

Concat

Prototype

function Concat(@char [] DestinationString, char[] InputString) : integer;

DestinationString	String to be concatenated to
InputString	String to concatenate

Description

Concatenates a value to a string

Returns

Uncertain. Possibly the length of the resulting string.

Remarks

This function can also use an integer or real value as its second parameter. If this is the case, then the value given is first converted to a string before concatenation.

It is uncertain if there is a return value. If so, it could be the length of the resulting string.

This function is likely only meant for debugging purposes as there doesn't seem to be any other uses for strings in the game.

Example use

```
Concat(dstring," is ");
Concat(dstring, 10);
Concat(dstring, 3.14159265);
```

See also

Print

Print

Prototype

function Print(char[] String);

String	String to be printed

Description

Prints the string to the debug console.

Returns

no return value

Remarks

This function either prints to the debug console, or to a log file. It is unclear which (or perhaps both).

The print function is also capable of accepting integers and reals as the parameter. The function specifically checks for these and handles them appropriately.

This function has no real use as the game's debugging console has been disabled.

Example use

```
Print(“I'm a message”);
Print( 10 );
Print( 3.14159265 );
```

See also

Concat

Handle

Prototype
    function Handle : integer;

Description
    Get a handle

Returns
    Unknown

Remarks
    This functions appears to have been deprecated.

Seen In
    not seen in any scripts

GetWeapons

Prototype

function GetWeapons(@integer[] List, integer InfoType) : integer;

List	Array of integers to receive the weapon list
InfoType	Type of list to retrieve. 0 = All weapons 1 = Only weapons that are functional and have ammo

Description

Get the weapons list of the current unit

Returns

Number of weapons in list
returns 0 if the current object is not a mover (mech or vehicle)

Remarks

The maximum size of the list array is not known.

This function may have been deprecated. The execution code does appear in the game's executable, but it doesn't appear to ever be called by the language parser.

Seen In

not seen in any scripts

GetMoveOrder

Prototype

function GetMoveOrder : integer;

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

GetAttackOrder

Prototype
```
function GetAttackOrder : integer;
```

Description
Unknown

Returns
Unknown

Remarks
This functions appears to have been deprecated.

Seen In
not seen in any scripts

OrderFormation

Prototype

function OrderFormation(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

GetFormation

Prototype

function GetFormation(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

UseSpeed

Prototype

function UseSpeed(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

UseFireRange

Prototype

function UseFireRange(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

UseFireOdds

Prototype

function UseFireOdds(integer Unknown) : integer;

Unknown	Unknown

Description

Unknown

Returns

Unknown

Remarks

This functions appears to have been deprecated.

Seen In

not seen in any scripts

IsServer

Prototype
function IsServer : boolean;

Description
Check if this is the game server for a multiplayer game.

Returns
TRUE if it is, FALSE otherwise

Remarks
none

Seen In
not seen in any scripts