NWScript Basics

Before we can really being discussing the actual contents of the NWScript Compiled Script (NCS) file, we first must go over some of the basic concepts of the NWScript engine.

NWScript is a small instruction set byte code engine. This means that instead of compiling the script into x86 machine instructions, the compiler generates a series of platform independent commands. Languages such as Forth or Java use similar techniques to store their compiled source. When a script needs to be executed, current byte code is fetched from the compiled script and then depending on the value of the byte code, the script engine executes some predefined operation.

NWScript Stack Basics

In a real machine code program, local variables can be stored in specific memory locations, memory relative to a stack pointer, or even in the CPU registers. Byte code engines don't have the luxury of using CPU registers. Thus they are limited to variables being stored in specific memory locations or relative to a stack pointer. In the case of NWScript, all variables are accessed relative to a stack pointer. Global variables don't exist in the traditional sense.

Since variables are all accessed off the stack without CPU registers, then operators such as addition must operate differently. In machine code, if you wished to add two values, the values would be loaded into registers and then the operator is executed. (Note: That is a drastic simplification) With NWScript, operators always use the top most variable or variables on the stack. Once the operation is complete, the variables are removed and the result is placed on the stack. Thus, if you have a variable called "nValue" that you wished to negate but not lose "nValue" on the stack, you would first have make a copy of the variable onto the top of the stack and then invoke the operator.

Let us look at an example program:

```
void main ()

{

    int i = 12;

    int j = 1;

    i = i + j;

}
```

The first two lines of the program declare two variables "i" and "j". Once these two statements are complete, the stack looks as follows:

Top of Stack (SP)	
Offset	**Contents**
-4	j: 1
-8	i: 12
Bottom of Stack	

It is important to note that the NWScript stack builds up in increments of 4. Thus when accessing values on the stack, they must be referenced using negative offsets where "-4" points to the top most element on the stack.

The next step is to make a copy of "i" so that we can operate on it.

Top of Stack (SP)	
Offset	**Contents**
-4	i: 12
-8	j: 1
-12	i: 12
Bottom of Stack	

Next, we need to make a copy of "j".

Top of Stack (SP)	
Offset	**Contents**
-4	j: 1
-8	i: 12
-12	j: 1
-16	i: 12
Bottom of Stack	

Now that we have the two values on the top of the stack, we can invoke the operator to compute the results.

Top of Stack (SP)	
Offset	**Contents**
-4	results: 13
-8	j: 1
-12	i: 12
Bottom of Stack	

The final step is the assignment. To do this, we copy the top of the stack down to the variable and then remove the top of the stack.

Top of Stack (SP)	
Offset	**Contents**
-4	j: 1
-8	i: 13
Bottom of Stack	

The current top of stack is also known as the stack pointer (SP).

NWScript Global Variables

As stated previously, NWScript does not have global variables in the tradition sense where the values a stored in a known region of memory. In NWScript, global variables are placed onto the stack by a dummy shell routine. This routine wraps the "main" or "StartingConditional" routine. So when a script is executed with global variables, "main" and "StartingConditional" are not the first routines to be invoked. The "#globals" routine is invoked to place the globals onto the stack and then it invokes "main" or "StartingConditional".

However, placing global variables on the stack is only half the problem. Routines inside the script must be able to know how to reference the variables. For a routine such as "main" that is invoked directly from "#globals", it knows how deep down in the stack the global variables would be. If it had two local variables and needed to access the top global variable, it could use an offset of -12 (3 variables down time -4). However, subroutines called by "main" would have no idea how deep down the stack the global variables exist.

To solve this problem, Bioware created a second stack pointer called "BP" which is traditionally called base pointer for Intel processors. Inside the "#globals" routine just prior to invoking "main" or "StartingConditional", the current stack pointer (SP) is saved and becomes the new value of BP. Then when "main" or and subroutine needs to access a global variable, it just needs to access them relative to BP.

For simplicity, there are not many operations that can be done to a variable relative to BP. A copy of a variable can be placed on the top of the stack. The current top of stack can be assigned to a variable relative to BP. And a variable relative to BP can be incremented or decremented.

Calling Subroutines and Engine Routines (ACTIONS)

Invoking subroutines or engine routines is done basically in the same manner. Arguments are placed on the stack in reverse order. The call is then made and the callee removes all the arguments from the stack prior to returning.

However, return values are handled differently. In the case of a script subroutine that returns a value, space for the return value is reserved, then the arguments are placed on the stack and finally the subroutine is invoked. In the case of an engine routine, it is the job of the engine routine to place the return value on the stack after the calling arguments are removed.

Here is an example of a call to a subroutine:

```
    int j = DoSomeScriptSubroutine (12, 14);
```

Prior to the call, the stack looks as follows:

Top of Stack (SP)	
Offset	**Contents**
-4	Arg1: 12
-8	Arg2: 14
-12	Return: ??
-16	j: ??

Bottom of Stack

After the call, the stack looks as follows:

Top of Stack (SP)	
Offset	**Contents**
-4	Return: ??
-8	j: ??
Bottom of Stack	

Here is an example of a call to an engine routine:

```
int j = DoSomeEngineRoutine (12, 14);
```

Prior to the call, the stack looks as follows:

Top of Stack (SP)	
Offset	**Contents**
-4	Arg1: 12
-8	Arg2: 14
-12	j: ??
Bottom of Stack	

After the call, the stack looks as follows:

Top of Stack (SP)	
Offset	**Contents**
-4	Return: ??
-8	j: ??
Bottom of Stack	

Byte Code Basics

All NWScript byte codes start with two bytes. The first byte is the instruction such as "RETN" or "JSR". The second byte is the type of the instruction such as an integer or floating point operation.

Following is a list of all the different types:

Unary Types	
Value	**Type**
3 (0x03)	Integer (I)
4 (0x04)	Float (F)
5 (0x05)	String (S)

6 (0x06)	Object (O)
16-31 (0x10-0x1F)	Engine Types 16 = Effect 17 = Event 18 = Location 19 = Talent
Binary Types	
Value	**Type**
32 (0x20)	Integer, Integer (II)
33 (0x21)	Float, Float (FF)
34 (0x22)	Object, Object (OO)
35 (0x23)	String, String (SS)
36 (0x24)	Structure, Structure (TT)
37 (0x25)	Integer, Float (IF)
38 (0x26)	Float, Integer (FI)
48-57 (0x30-0x39)	Engine Types 48 = Effect, Effect 49 = Event, Event 50 = Location, Location 51 = Talent, Talent
58 (0x3A)	Vector, Vector (VV)
59 (0x3B)	Vector, Float (VF)
60 (0x3C)	Float, Vector (FV)

The value listed in parenthesis next to the type name is the short hand name used to identify different byte codes. For example ADDII would add two integer values.

The TT opcode type is used to compare ranges of elements on the stack. More specifically, it is used for structures and vectors.

Byte Codes

Following is a list and description of all the known byte codes.

NOTE: All multi-byte values are stored in big endian order.

CPDOWNSP - Copy Down Stack Pointer

Copy the given number of bytes from the top of the stack down to the location specified.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x01	Byte Code

1	0x01	Type
2-5	Offset	Destination of the copy relative to the top of the stack.
6-7	Size	Number of bytes to copy

RSADDx- Reserve Space on Stack
RSADDI- Reserve Integer Space on Stack
RSADDF- Reserve Float Space on Stack
RSADDS- Reserve String Space on Stack
RSADDO- Reserve Object Space on Stack

Reserve space on the stack for the given variable type.

The value of SP is increased by the size of the type reserved. (Always 4)

Bytes	Value	Description
0	0x02	Byte Code
1	0x03	RSADDI type
	0x04	RSADDF type
	0x05	RSADDS type
	0x06	RSADDO type

CPTOPSP - Copy Top Stack Pointer

Add the given number of bytes from the location specified in the stack to the top of the stack.

The value of SP is increased by the number of copied bytes.

Bytes	Value	Description
0	0x03	Byte Code
1	0x01	Type
2-5	Offset	Source of the copy relative to the top of the stack.
6-7	Size	Number of bytes to copy

CONSTI - Place Constant Integer Onto the Stack

Place the constant integer onto the top of the stack.

The value of SP is increased by the size of the type reserved. (Always 4)

Bytes	Value	Description
0	0x04	Byte Code
1	0x03	Type
2-5	Integer	Integer value of the constant

CONSTF - Place Constant Float Onto the Stack

Place the constant float onto the top of the stack.

The value of SP is increased by the size of the type reserved. (Always 4)

Bytes	Value	Description
0	0x04	Byte Code
1	0x04	Type
2-5	Float	Float value of the constant

CONSTS - Place Constant String Onto the Stack

Place the constant string onto the top of the stack.

The value of SP is increased by the size of the type reserved. (Always 4)

Bytes	Value	Description
0	0x04	Byte Code
1	0x05	Type
2-3	String Length	Length of the string
4-n	String Data	Text of the string

CONSTO - Place Constant Object ID Onto the Stack

Place the constant object ID onto the top of the stack.

The value of SP is increased by the size of the type reserved. (Always 4)

Bytes	Value	Description
0	0x04	Byte Code
1	0x06	Type
2-5	Object	When the type is an object, these bytes contain the OID

ACTION - Call an Engine Routine

Invoke the engine routine specified. All arguments must be placed on the stack in reverse order prior to this byte code. The arguments will be removed by the engine routine and any return value then placed on the stack.

The value of SP is increased by the size of the return value and decreased by the total size of the arguments. It is important to note that the total size of the arguments might be different than the number of arguments. Structures and vectors are take up more space than normal types.

Bytes	Value	Description

0	0x05	Byte Code
1	0x00	Type
2-3	Routine #	Number of the action routine. NWSCIPT.NSS lists engine routines in order starting at 0
4	Arg Count	Number of arguments

LOGANDII - Logical AND Two Integers

Compute the logical AND of two integer values.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x06	Byte Code
1	0x20	Type

LOGORII - Logical OR Two Integers

Compute the logical OR of two integer values.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x07	Byte Code
1	0x20	Type

INCORII - Bitwise Inclusive OR Two Integers

Compute the inclusive OR of two integer values.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x08	Byte Code
1	0x20	Type

EXCORII - Bitwise Exclusive OR Two Integers

Compute the exclusive OR of two integers.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description

0	0x09	Byte Code
1	0x20	Type

BOOLANDII - Boolean or Bitwise AND Two Integers

Compute the boolean AND of two integers.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0A	Byte Code
1	0x20	Type

EQUALxx - Test for Logical Equality
EQUALII - Test for Logical Equality Two Integers
EQUALFF - Test for Logical Equality Two Floats
EQUALSS - Test for Logical Equality Two Strings
EQUALOO - Test for Logical Equality Two Object IDs

Test the two operand for logical equality. This operator supports the comparison or all the basic types and then engine types as long as both operands have the same type.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0B	Byte Code
1	0x20 0x21 0x22 0x23 0x30-0x39	EQUALII Type EQUALFF Type EQUALOO Type EQUALSS Type For engine types

EQUALTT - Test for Logical Equality Two Structures

Test the two operand for logical equality. This operator supports the comparison or all the basic types and then engine types as long as both operands have the same type.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0B	Byte Code
1	0x24	Type
2-3	Size	Size of structure

NEQUALxx - Test for Logical Inequality

NEQUALII - Test for Logical Inequality Two Integers
NEQUALFF - Test for Logical Inequality Two Floats
NEQUALSS - Test for Logical Inequality Two Strings
NEQUALOO - Test for Logical Inequality Two Object IDs

Test the two operand for logical inequality. This operator supports the comparison or all the basic types and then engine types as long as both operands have the same type.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0C	Byte Code
1	0x20 0x21 0x22 0x23 0x30-0x39	EQUALII Type EQUALFF Type EQUALOO Type EQUALSS Type For engine types

NEQUALTT - Test for Logical Inequality Two Structures

Test the two operand for logical inequality. This operator supports the comparison or all the basic types and then engine types as long as both operands have the same type.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0C	Byte Code
1	0x24	Type
2-3	Size	Size of the structure

GEQxx - Test for Greater Than or Equal
GEQII - Test for Greater Than or Equal Two Integers
GEQFF - Test for Greater Than or Equal Two Floats

Test the two operand for logically greater than or equal.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0D	Byte Code
1	0x20 0x21	GEQII Type GEQFF Type

GTxx - Test for Greater Than
GTII - Test for Greater Than Two Integers
GTFF - Test for Greater Than Two Floats

Test the two operand for logically greater than.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0E	Byte Code
1	0x20 0x21	GTII Type GTFF Type

LTxx - Test for Less Than
LTII - Test for Less Than Two Integers
LTFF - Test for Less Than Two Floats

Test the two operand for logically less than.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x0F	Byte Code
1	0x20 0x21	LTII Type LTFF Type

LEQxx - Test for Less Than or Equal
LEQII - Test for Less Than or Equal Two Integers
LEQFF - Test for Less Than or Equal Two Floats

Test the two operand for logically less than or equal.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x10	Byte Code
1	0x20 0x21	LEQII Type LEQFF Type

SHLEFTII - Shift the Integer Value Left

Shift the value left be the given number of bits. Operand one is the value to shift while operand two is the number of bits to shift.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x11	Byte Code
1	0x20	Type

SHRIGHTII - Shift the Integer Value Right

Shift the value right be the given number of bits. Operand one is the value to shift while operand two is the number of bits to shift.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x12	Byte Code
1	0x20	Type

USHRIGHTII - Unsigned Shift the Integer Value Right

Shift the value right be the given number of bits as if it was an unsigned integer and not a signed integer. Operand one is the value to shift while operand two is the number of bits to shift.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x13	Byte Code
1	0x20	Type

ADDxx - Add Two Values
ADDII - Add Two Integer Values
ADDIF - Add an Integer and Float Values
ADDFI - Add a Float and Integer Values
ADDFF - Add Two Float Values
ADDSS - Add Two String Values
ADDVV - Add Two Vector Values

Add the two operands.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x14	Byte Code
1	0x20 0x25 0x26 0x21 0x23 0x3A	ADDII Type ADDIF Type ADDFI Type ADDFF Type ADDSS Type ADDVV Type

SUBxx - Subtract Two Values
SUBII - Subtract Two Integer Values
SUBIF - Subtract an Integer and Float Values

SUBFI - Subtract a Float and Integer Values
SUBFF - Subtract Two Float Values
SUBVV - Subtract Two Vector Values

Subtract the two operands.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x15	Byte Code
1	0x20 0x25 0x26 0x21 0x3A	SUBII Type SUBIF Type SUBFI Type SUBFF Type SUBVV Type

MULxx - Multiply Two Values
MULII - Multiply Two Integer Values
MULIF - Multiply an Integer and Float Values
MULFI - Multiply a Float and Integer Values
MULFF - Multiply Two Float Values
MULVF - Multiply a Vector and Float Values
MULFV - Multiply a Float and Vector Values

Multiply the two operands.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x16	Byte Code
1	0x20 0x25 0x26 0x21 0x3B 0x3C	MULII Type MULIF Type MULFI Type MULFF Type MULVF Type MULFV Type

DIVxx - Divide Two Values
DIVII - Divide Two Integer Values
DIVLIF - Divide an Integer and Float Values
DIVFI - Divide a Float and Integer Values
DIVFF - Divide Two Float Values
DIVVF - Divide a Vector and Float Values

Divide the two operands.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x17	Byte Code
1	0x20 0x25 0x26 0x21 0x3B	DIVII Type DIVIF Type DIVFI Type DIVFF Type DIVVF Type

MODII- Compute the Modulus of Two Integer Values

Computes the modulus of two values.

The value of SP is increased by the size of the result while decreased by the size of both operands.

Bytes	Value	Description
0	0x18	Byte Code
1	0x20	Type

NEGx - Compute the Negation of a Value
NEGI - Compute the Negation of an Integer Value
NEGF - Compute the Negation of a Float Value

Computes the negation of a value.

The value of SP remains unchanged since the operand and result are of the same size.

Bytes	Value	Description
0	0x19	Byte Code
1	0x03 0x04	NEGI Type NEGF Type

COMPI - Compute the One's Complement of an Integer Value

Computes the one's complement of a value.

The value of SP remains unchanged since the operand and result are of the same size.

Bytes	Value	Description
0	0x1A	Byte Code
1	0x03	Type

MOVSP - Adjust the Stack Pointer

Add the value specified in the instruction to the stack pointer.

The value of SP is adjusted by the value specified.

Bytes	Value	Description
0	0x1B	Byte Code
1	0x00	Type
2-5	Offset	Value to add to the stack pointer.

STORE_STATEALL - Store the Current State of the Stack (Obsolete)

Obsolete instruction to store the state of the stack and save a pointer to a block of code to later be used as an "action" argument. This byte code is always followed by a JMP and then a block of code to be executed by a later function such as a DelayCommand.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x1C	Byte Code
1	0x08	Offset to the block of code for an "action" argument

JMP - Jump to a New Location

Change the current execution address to the relative address given in the instruction.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x1D	Byte Code
1	0x00	Type
2-5	Offset	Offset to the new program location from the start of this instruction

JSR - Jump to Subroutine

Jump to the subroutine at the relative address given in the instruction. If the routine returns a value, the RSADDx instruction should first be used to allocate space for the return value. Then all arguments to the subroutine should be pushed in reverse order.

The value of SP remains unchanged. The return value is NOT placed on the stack.

Bytes	Value	Description
0	0x1E	Byte Code
1	0x00	Type
2-5	Offset	Offset to the new program location from the start of this instruction

JZ - Jump if Top of Stack is Zero

Change the current execution address to the relative address given in the instruction if the integer on the top of the stack is zero.

The value of SP is decremented by the size of the integer.

Bytes	Value	Description
0	0x1F	Byte Code
1	0x00	Type
2-5	Offset	Offset to the new program location from the start of this instruction

RETN - Return from a JSR

Return from a JSR. All arguments used to invoke the subroutine should be removed prior to the RETN. This leaves any return value on the top of the stack. The return value must be allocated by the caller prior to invoking the subroutine.

The value of SP remains unchanged. The return value is NOT placed on the stack.

Bytes	Value	Description
0	0x20	Byte Code
1	0x00	Type

DESTRUCT - Destroy Element on the Stack

Given a stack size, destroy all elements in that size excluding the given stack element and element size.

The value of SP decremented by the given stack size minus the element size.

Bytes	Value	Description
0	0x21	Byte Code
1	0x01	Type
2-3	Size	Total number of bytes to remove off the top of the stack
4-5	Offset	Offset from the start of the bytes to remove to the element not to destroy
6-7	Size	Size of the element not to destroy

NOTI - Compute the logical NOT of an Integer Value

Computes the logical not of the value.

The value of SP remains unchanged since the operand and result are of the same size.

Bytes	Value	Description
0	0x22	Byte Code
1	0x03	Type

DECISP - Decrement Integer Value Relative to Stack Pointer

Decrements an integer relative to the current stack pointer.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x23	Byte Code
1	0x03	Type
2-5	Offset	Offset of the integer relative to the stack pointer

INCISP - Increment Integer Value Relative to Stack Pointer

Increments an integer relative to the current stack pointer.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x24	Byte Code
1	0x03	Type
2-5	Offset	Offset of the integer relative to the stack pointer

JNZ - Jump if Top of Stack is Non-Zero

Change the current execution address to the relative address given in the instruction if the integer on the top of the stack is non-zero.

The value of SP is decremented by the size of the integer.

Bytes	Value	Description
0	0x25	Byte Code
1	0x00	Type
2-5	Offset	Offset to the new program location from the start of this instruction

CPDOWNBP - Copy Down Base Pointer

Copy the given number of bytes from the base pointer down to the location specified. This instruction is used to assign new values to global variables.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x26	Byte Code
1	0x01	Type

2-5	Offset	Destination of the copy relative to the base pointer
6-7	Size	Number of bytes to copy

CPTOPBP - Copy Top Base Pointer

Add the given number of bytes from the location specified relative to the base pointer to the top of the stack. This instruction is used to retrieve the current value of global variables.

The value of SP is increased by the number of copied bytes.

Bytes	Value	Description
0	0x27	Byte Code
1	0x01	Type
2-5	Offset	Source of the copy relative to the base pointer
6-7	Size	Number of bytes to copy

DECIBP - Decrement Integer Value Relative to Base Pointer

Decrements an integer relative to the current base pointer. This instruction is used to decrement the value of global variables.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x28	Byte Code
1	0x03	Type
2-5	Offset	Offset of the integer relative to the base pointer

INCIBP - Increment Integer Value Relative to Base Pointer

Increments an integer relative to the current base pointer. This instruction is used to increment the value of global variables.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x29	Byte Code
1	0x03	Type
2-5	Offset	Offset of the integer relative to the base pointer

SAVEBP - Set a New Base Pointer Value

Save the current value of the base pointer and set BP to the current stack position.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x2A	Byte Code
1	0x00	Type

RESTOREBP - Restored the BP

Restore the BP from a previous SAVEBP instruction.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x2B	Byte Code
1	0x00	Type

STORE_STATE - Store the Current Stack State

Store the state of the stack and save a pointer to a block of code to later be used as an "action" argument. This byte code is always followed by a JMP and then a block of code to be executed by a later function such as a DelayCommand.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x2C	Byte Code
1	0x10	Offset to the block of code for an "action" argument
2-5	Size	Size of the variables to save relative to BP. This would be all the global variables.
6-9	Size	Size of the local routine variables to save relative to SP.

NOP - No-operation

Perform no program function. This opcode is used as a placeholder for the debugger.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x2D	Byte Code
1	0x00	Type

T - Program Size

This byte code isn't a real instruction and is always found at offset 8 in the NCS file.

The value of SP remains unchanged.

Bytes	Value	Description
0	0x42	Byte Code
1-4	Size	Size of the NCS file