

Nippon Pulse Builder

Table of contents

Copyright	3
Cautions	3
Contact Information	3
NP Builder	4
1.0 AScript Editor	4
1.1 AScript Motion Control Command Set	9
1.2 AScript BASIC Commands	10
1.3 Create AScript	14
1.4 Compile and Run AScript	19
1.5 Additional AScript Options	22
2.0 BScript Editor	23
2.1 Parameter Descriptions	25
2.2 Terminal Display	29
2.3 BScript Tools	33
2.4 Plugin Designer	37
2.5 BScript Commands	39
2.5.1 BScript Extension Commands	41
2.5.2 Extended LOGIC BScript Commands	41
2.5.3 Other BScript Commands	42
2.5.4 Use BScript Commands	43
2.6 BScript Syntax	44
2.7 BScript Code Samples	45
2.7.1 General	46
2.7.2 IOControl	47
2.7.3 MotionControl	47
2.7.4 SystemConfig	50
3.0 Code Block Builder	52
3.1 Build a Code Block	54
PlayNote Sample	62
3.2 Run a Code Block	64

Copyright

Copyright

Copyright © 2019-2026 by NIPPON PULSE AMERICA, INC. All Rights Reserved.

First edition, October 2019

Current edition, August 2026

You may not reproduce or translate into any language in any form any part of this publication without written permission from NIPPON PULSE AMERICA, INC.

NIPPON PULSE AMERICA, INC. makes no representations or warranties regarding the content of this document. We reserve the right to revise this document any time without notice and obligation.

Cautions

- Copying any part of this manual without written approval is prohibited.
- The specifications of this controller may be changed without prior notice.
- If you find unclear, incomplete, or incorrect content, contact us using the [Page issue?](#) link at the bottom of the page to provide feedback.
- NIPPON PULSE AMERICA, INC. is not responsible for any results that occur from using this controller.
- The Commander core is designed for use in commercial apparatus (office machines, communication equipment, measuring equipment, and household appliances). If you use it in any device that requires high quality and reliability, or where faults or malfunctions may directly injure humans or endanger human survival (such as in nuclear power control devices, aviation devices or spacecraft, traffic signals, fire control, or various types of safety devices), NIPPON PULSE AMERICA, INC. is not liable for any problem that occurs, even if it was directly caused by the Commander core. Customers must provide their own safety measures to ensure appropriate performance in all circumstances.

Explanation of the descriptions in this manual

- The **X**, **Y**, **Z** and **U** in terminal names and bit names refer to the X-axis, Y-axis, Z-axis, and U-axis, respectively.
- Terminals with a / (ex. /RST) are negative logic. Their logic cannot be changed. Terminals without a / are positive logic. Their output logic can be changed.
- When describing the bits in registers, **n** refers to the bit position. A **0** means that the bit is in position 0 and that it is prohibited to write to any bit other than **0**. This bit always return a **0** when read out.

Contact Information

Nippon Pulse America, Inc.



4 Corporate Drive Street, Radford, VA 24141 USA

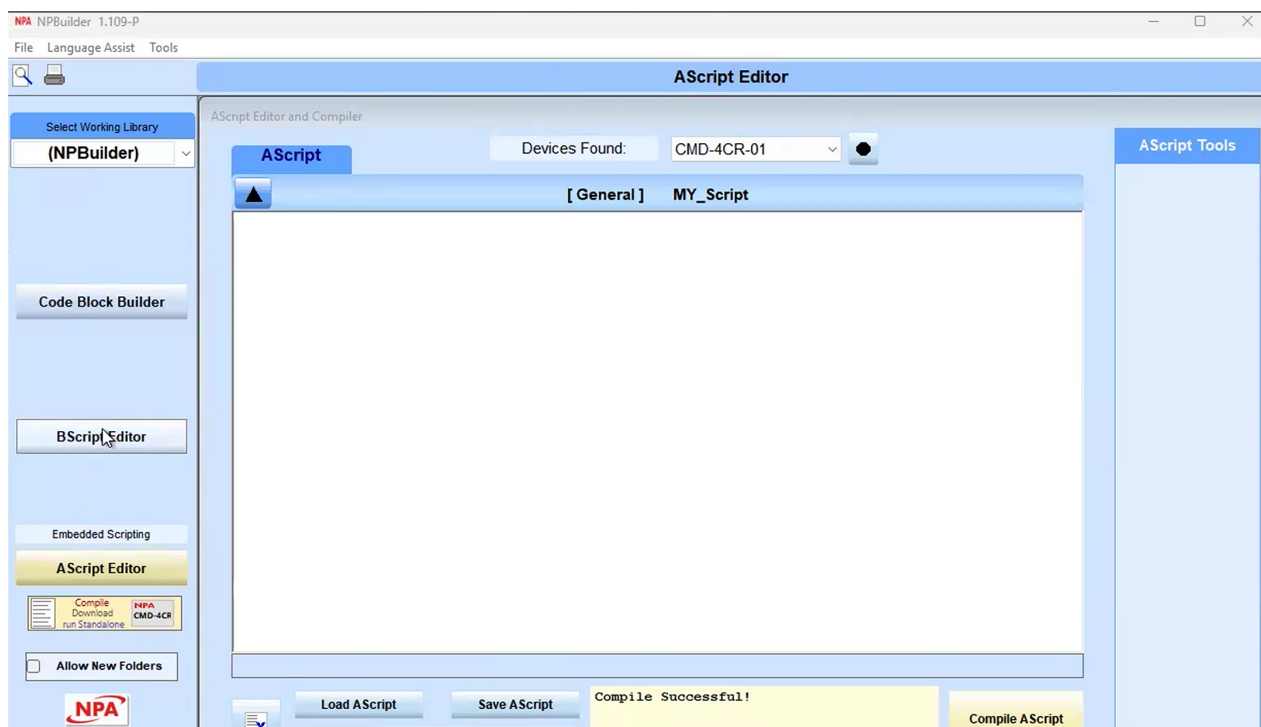
Phone: 1-540-633-1677

E-mail: info@nipponpulse.com

Web: <https://www.nipponpulse.com>

NP Builder

NP Builder



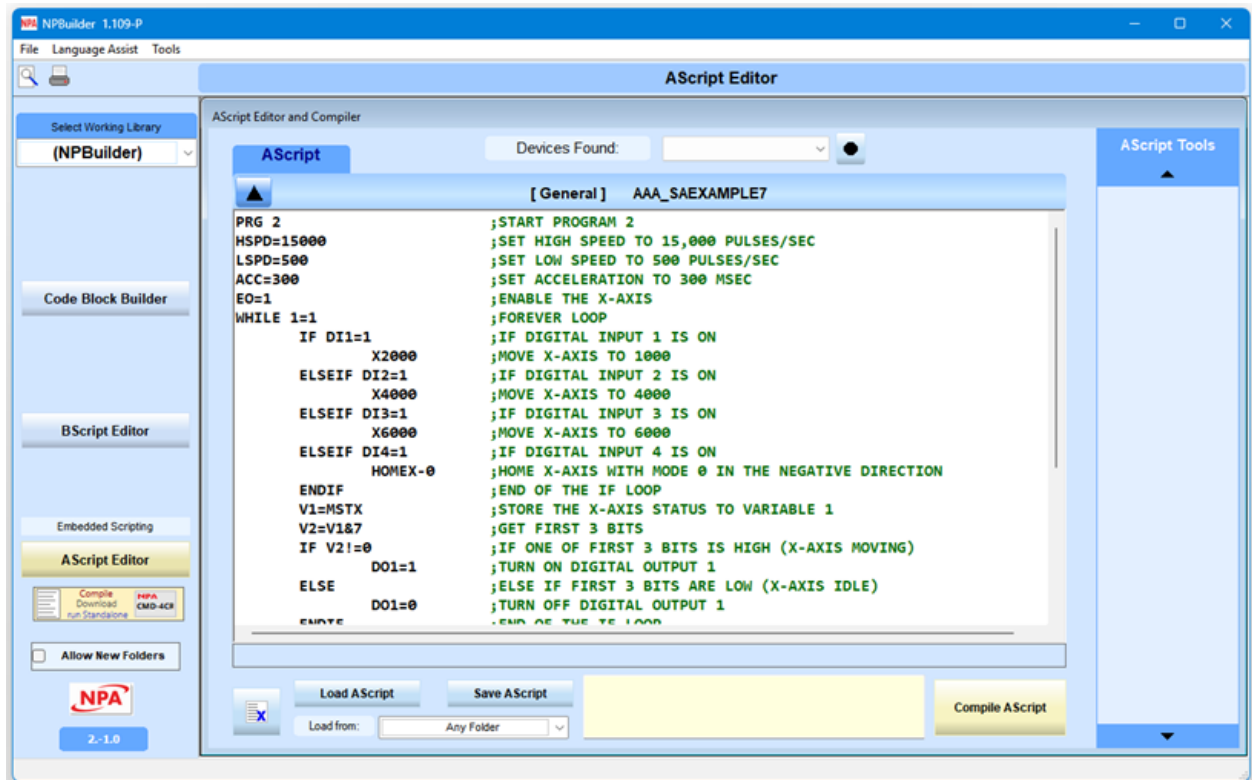
NPBuilder is a software tool for Windows, specifically designed for faster prototyping and development of motion control projects for the Commander ecosystem. The user interface (UI) provides an array of tools for building projects that can be run locally on a desktop environment or embedded onto the Commander to be run without the need for outside resources.

NPBuilder Core Features

- [AScript Editor](#)
- [BScript Editor](#)
- [Code Block Builder](#)

1.0 AScript Editor

AScript is a hybrid scripting language which can be compiled to Pcode. It includes [AScript Motion Control Commands](#), and a [BASIC Command Set](#).



Pcode is a machine code like language the Commander can interpret and use to execute a program faster than if it had to translate a program from a human readable script format one line at a time. When NPBBuilder compiles a script, a single line of code may become several lines of Pcode; still, it is faster for the Commander to process. When the Commander reports it can only store limited lines of code, it is referring to how many lines of Pcode the script is converted into. So, for example, if the Commander can store 6000 lines of Pcode, then scripts should only be around 1000 to 1500 lines of Ascript.

The Ascript Editor is where you write (and compile) Ascript programs.

Key functions:

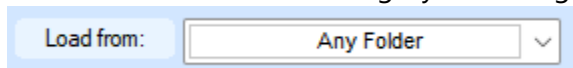
- Load and save scripts from the disk drives.



- Clear the script editor.



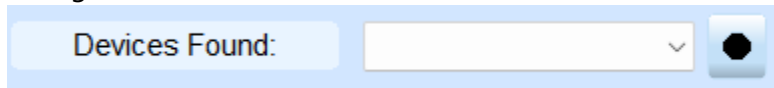
- Define the current folder category for saving and loading script files.



There are four categories you can use for saving script files:

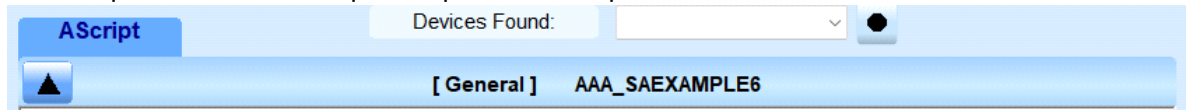
- **System Configuration**
 - **LibrariesMotion Control Libraries**
 - **IO Control Libraries**
 - **General Libraries**
- **Available Commander controllers**

Select a device are currently working on and to download a compiled script to. The button to the right is the **Refresh** button which will refresh the device list.



- **Script Title Bar**

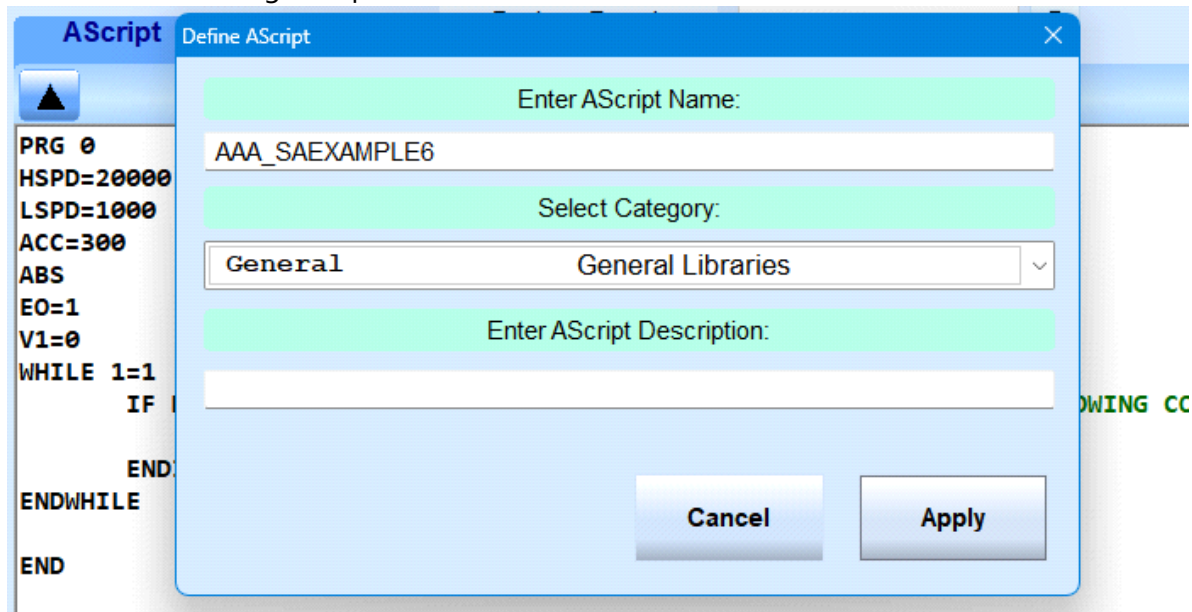
The name of your current script is in the Title Bar displays the name of the current script. The Arrow Up button on the top left opens the Script definition screen.



- **Script Definition**

- Name a script file
- Select a library folder
- brief description of the script.

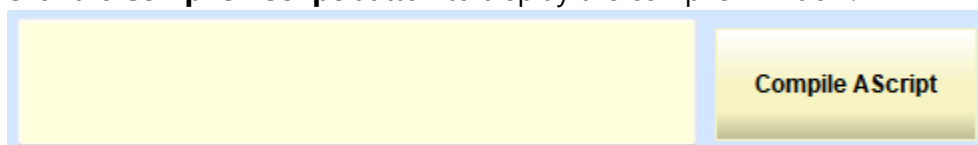
When you save a script this information is also included in the file. NPbuilder uses an XML like file format for storing Ascript files.



- Compiler message and Compile Button.

The yellow window is where the compiler returns error messages or a compile success message.

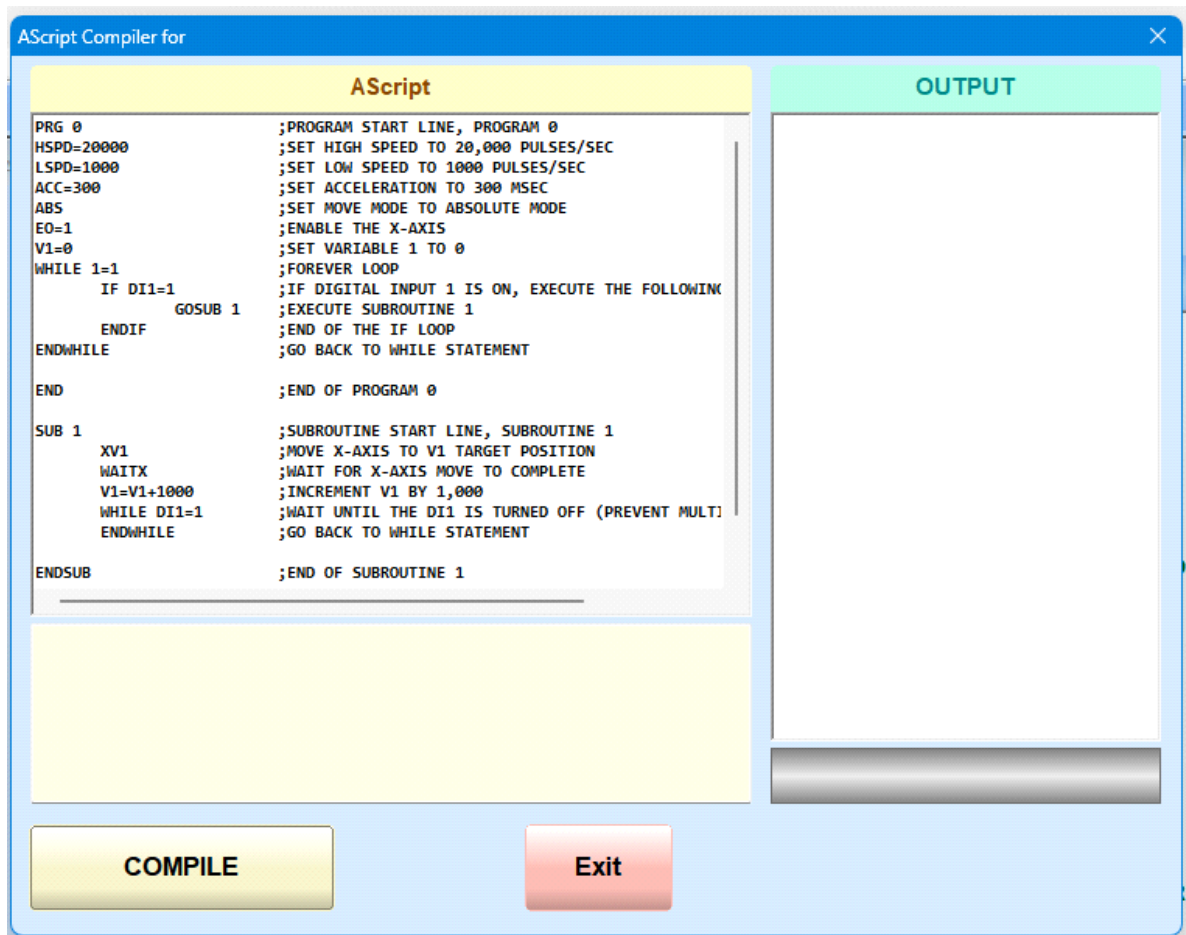
Click the **Compile AScript** button to display the compiler window.



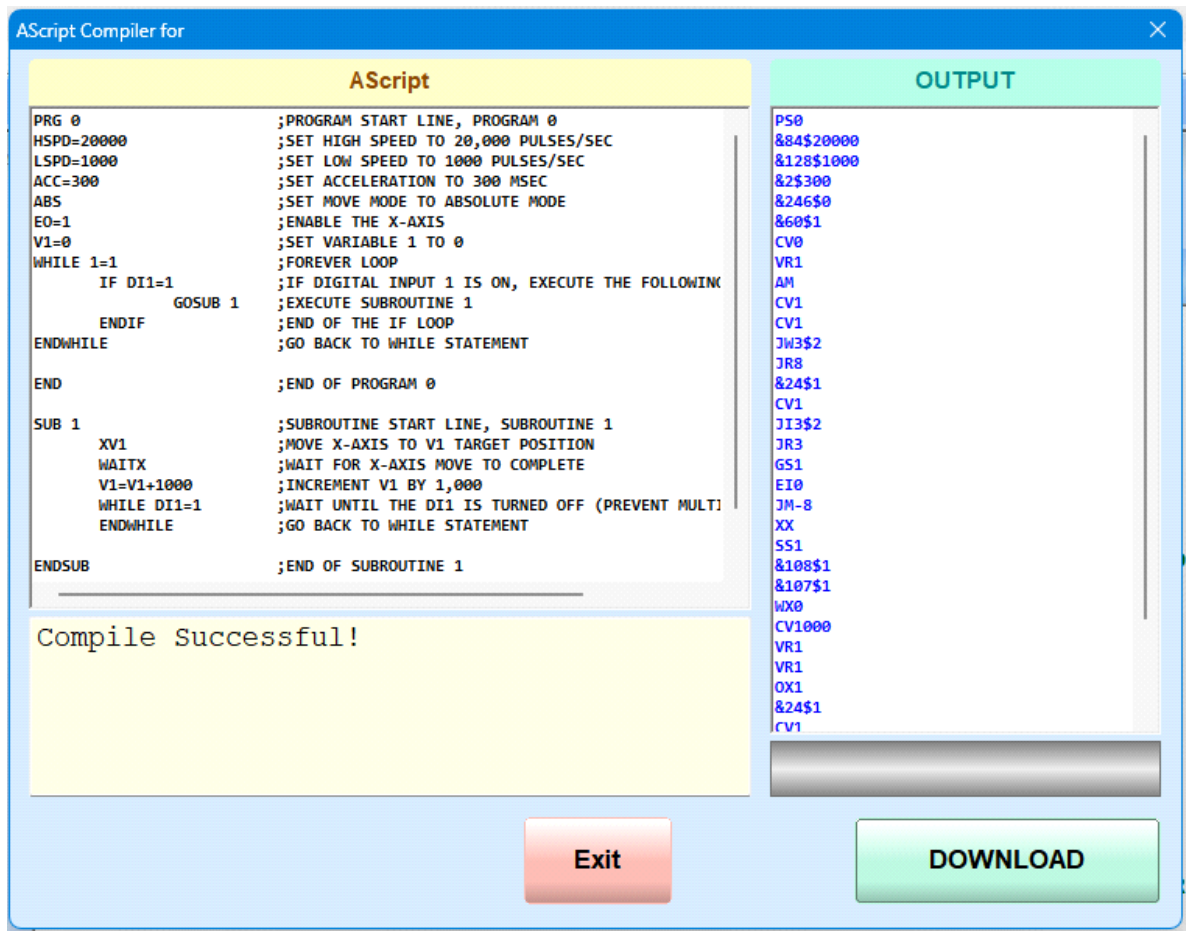
- Compiler screen

The script is displayed in the Compiler screen.

Click **Compile** to start.

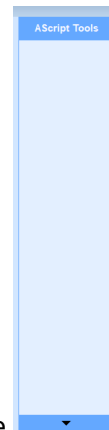


- Compiler results
- If there is an error, the compiler flags the problem script line. Click exit to fix the script.
- If the script compiles successfully, you can then click the Download button to download your compiled script to the Commander motion controller. The output box is the actual Pcode generated by the compiler. This will likely not make much sense to you, but it is there for diagnosis by Nippon. If you have problems with a script compiling properly and not working correctly, this output window may be useful to Nippon in finding any errors in the compiler itself.



- The AScript Editor has a Tools window on the right side of the editor. This Tools window is where any custom AScript Plugins will show up. NPbuilder has an Open API for writing plugins (32 bit Windows DLL) which it can load to add features to NPBuilder. You will be able to

create your own plugins and Nippon may also provide some they create.



1.1 AScript Motion Control Command Set

AScript motion control commands are similar to the Commander ASCII command set. There can be small differences in the syntax of AScript commands and ASCII commands (a command name may be slightly different or parameters treated differently).

Note: AScript motion control is only a *subset* of the entire ASCII command set, however, the AScript motion control command set does provide a feature set to handle most requirements.

Some motion control commands can also be used as functions to return a value, and can be used in **IF THEN** statements. **IF** statements can be nested to an unlimited depth.

Controllers in the Commander Ecosystem support four threads of AScript programs, which are labeled from 0 to 3. Each program supports up to 32 subroutines.

Note: Subroutines are not named but are referenced by an ID number from 0 to 31.

Comments are any characters after a semicolon ; character or a single quote ' character.

Programs support up to 100 variables, and indicated by the letter **V** followed by the variable ID, 1 to 100. They only hold 32-bit integer values. There is no floating point in AScript at this time.

NPBuilder provides a compiler which converts AScript to PCode that can be downloaded to the controller. One line of AScript may generate up to four or five lines of PCode, so the PCode line limit of the Commander is not the same as lines of AScript. If a script is too big to download, NPBuilder informs you. The compiler also has some error reporting features and indicates which line of code an error was found on.

Each motion control command is Write only, Read only, or both Read and Write.

- The Write syntax of a motion control command means it is a statement where you pass parameters with it to tell the Commander to do something. Ascript recognizes a write command by the command being the first word on a line, and the Commander expects the correct parametrs after reading the Write syntax.
- If a motion control command is used in Read syntax, then the command is acting like a function in BASIC. This syntax could be in the context of an IF statement, WHILE statement or after an = character in an assignment statement. When a motion control command is used as if it is a function, it always returns a value to the BASIC statement it is in.

Details about motion control commands can be found in the Commander reference document:

[ASCII Command Set](#)

1.2 AScript BASIC Commands

The BASIC part of Ascript can be broken down into different categories.

- There is a command to create Loops. This is the **WHILE** command. Unlike more advanced BASICs which have the FOR NEXT loop, the only loop command in Ascript is the **WHILE** command. To create a loop with a certain number of iterations, you must define a counter for the loop. You can store a variable in a counter, and then increment it in the loop. The **WHILE** command tests for the variable counter.
- The Logic or Comparison command set, handled by the **IF** statement. There are a number of ways to nest the **IF** statement, so by using multiple **ELSEIF** commands you can implement something similar to the SELECT CASE command in other BASICs. The **IF** statement can consist of the **IF** command, multiple **ELSEIF** commands after it (as needed) and then only one **ELSE** command (per **IF**) and finally the **ENDIF** command to end the block.
- Subroutines are numbered (no name) from 0 to 31. Subroutine number 31 is a special routine which is called when there is an error. It is important to break up a program into smaller subroutines so it is easier to see the flow of code. Modular design makes code easier to work with, but also allows you to do more with less lines of code. If a subroutine can be used multiple times, the number of lines of code in your program is decreased. The Commander has a limit of how many lines of code (converted to Pcode) it can store.
- AScript allows up to 100 variables, numbered from 0 to 99 (for 4 axis Commander, but 0 to 64 for 2 axis Commander). Variables allow you to store a value for later comparison and to do calculations and store the results of the calculations.

Note: Math calculations can only be used during the setting of a variable (V# = something). During comparisons, such as in an **IF** statement you can not do a math calculation.

Available math calculations:

Operator	Description	Example
+	Integer Addition	V1=V2+V3
-	Integer Subtraction	V1=V2-V3
*	Integer Multiplication	V1=V2*V3
/	Integer Division (round	V1=V2/V3

	down)	
%	Modulus	V1=V2%5
>>	Bit Shift Right	V1=V2>>2
<<	Bit Shift Left	V1=V2<<2
&	Bitwise AND	V1=V2&7
	Bitwise OR	V1=V2 8
~	Bitwise NOT	V1=~V2

If you require a more complex calculation you will need to have multiple lines of code and do one calculation on each line of code.

- Motion control functions

Some motion control commands include a function syntax where they return *astate* or *value* from the Commander. When you use the motion control commands in their function syntax (READ, not WRITE) you can use them in BASIC statements as a numeric value. You can store a motion control state into a variable, or compare a motion control state in an IF statement. So in their function (READ) syntax motion control commands become part of the BASIC language to return state values back from the Commander.

List of BASIC commands

; [some text] Any text following semicolon in the same line appears as a comment
 ' [some text] Any text following single quote in the same line appears as a comment
 Note: comments also can come after any command or statement
DELAY Sets delay in ms
GOSUB[0-31] Calls and executes subroutine [0-31] and then returns to next line
IF [comparison]
ELSEIF [comparison]
ELSE
ENDIF

Executes the instruction following the **IF** statement when condition is true. Execute the instruction following the next **ELSE** or **ELSEIF** statement when the condition is false. When all conditions are false, continue executing program after **ENDIF**. The following operands are used in the conditional IF statement: =, >, <, >=, <=, !=, <>

The comparisons in an IF statement are always simple and not complex. They are simply one value compared to another value (no parantheses, no math). The operands are obvious but with one slight variation. The operand **!=** and **<>** both mean the same thing of not equal. The first one is typical of not equal as used in C and the second one is typical of not equal in BASIC. Both are acceptable.

Both the **IF** line and the **ELSEIF** lines can have a comparison. The **ELSE** and **ENDIF** lines do not. The new version of the Ascript compiler now has the ability of having unlimited nested **IF** and **WHILE** statements. The old version of the compiler was limited to just a few levels deep. The new compiler uses a two pass compiler which allows it to accomplish unlimited nesting. In the comparisons used in the IF statement, each side of the comparison can be one of the following: a variable (the letter V followed by a number), a numeric value (integer only) or any

motion control command which has a read syntax which returns a single value. Such motion control commands (read) act like a function and return a value.

Here is an example:

```

PRG 2           ' Program start line, program 2
HSPD=20000      ' Set high speed to 20,000 pulses/sec
LSPD=1000       ' Set low speed to 1000 pulses/sec
ACC=300        ' Set acceleration to 300 msec
EO=1          ' Enable the X-axis
V1=1          ' Set Variable # 1 to 1
WHILE 1=1      ' Forever loop
    IF DI1=V1   ' If digital input is on (= V1 which is 1) statement is TRUE
        X0      ' Move X-axis to zero
        WAITX   ' Wait for X-axis move to complete
        X1000   ' Move X-axis to 1,000
        WAITX   ' Wait for X-axis move to complete
    ENDIF       ' End of the IF loop
ENDWHILE      ' Go back to WHILE Statement
END          ' End program # 2

```

Note: The **THEN** keyword is allowed after the comparison in both the **IF** and **ELSEIF** commands. This is how many BASIC languages work, so Ascript can handle it either way (no**THEN** or use **THEN**). But when using **THEN** you can not put code after it like some BASIC's. You must put that code on the next line. So both of the following are acceptable:

```

IF DI1=V1      ' If digital input 1 is on, execute the following commands
    X0          ' Move X-axis to zero
    WAITX       ' Wait for X-axis move to complete
    X1000       ' Move X-axis to 1,000
    WAITX       ' Wait for X-axis move to complete
ENDIF         ' End of the IF loop

IF DI1=V1 THEN ' If digital input 1 is on, execute the following commands
    X0          ' Move X-axis to zero
    WAITX       ' Wait for X-axis move to complete
    X1000       ' Move X-axis to 1,000
    WAITX       ' Wait for X-axis move to complete
ENDIF         ' End of the IF loop

```

PRG[0-3] starts a program (thread) which is one of four possible programs
END ends current program

Ascript is multitasking (similar to threads, but technically it does time slicing and executes one line of code for each active program in a sequence and then starts over again so giving each program its own slice of the cycle). The **PRG / END** block defines the first and last line of a standalone program [0-3].

Example:

PRG 0	' Program start line, program 0
HSPD=20000	' Set high speed to 20,000 pulses/sec
LSPD=1000	' Set low speed to 1000 pulses/sec
ACC=300	' Set acceleration to 300 msec
EO1=1	' Enable the X-axis
V1=1	' Set Variable # 1 to 1
WHILE 1=1	' Forever loop
IF DI1=V1	' If digital input 1 is on, execute the following commands
X0	' Move X-axis to zero
WAITX	' Wait for X-axis move to complete
X1000	' Move X-axis to 1,000
WAITX	' Wait for X-axis move to complete
ENDIF	' End of the IF loop
ENDWHILE	' Go back to WHILE Statement
END	' End program # 0
 PRG 1	 ' Program start line, program 1
HSPD=20000	' Set high speed to 20,000 pulses/sec
LSPD=1000	' Set low speed to 1000 pulses/sec
ACC=300	' Set acceleration to 300 msec
EO2=1	' Enable the Y-axis
V1=1	' Set Variable # 1 to 1
WHILE 1=1	' Forever loop
IF DI2=V1	' If digital input 2 is on, execute the following commands
Y0	' Move Y-axis to zero
WAITY	' Wait for Y-axis move to complete
Y1000	' Move Y-axis to 1,000
WAITY	' Wait for Y-axis move to complete
ENDIF	' End of the IF loop
ENDWHILE	' Go back to WHILE Statement
END	' End program # 1

SUB[0-31]	Beginning of subroutine (31 is for errors)
ENDSUB	End of subroutine and then return to caller

The **SUB / ENDSUB** block is used to define the first and last line of a Subroutine [0-31]

Example:

PRG 0	' Program start line, Program 0
HSPD=20000	' Set high speed to 20,000 pulses/sec
LSPD=1000	' Set low speed to 1000 pulses/sec

ACC=300	' Set acceleration to 300 msec
ABS	' Set move mode to absolute mode
EO=1	' Enable the X-axis
V1=0	' Set variable 1 to 0
WHILE 1=1	' Forever loop
IF DI1=1 THEN	' If digital input 1 is on, execute the following commands
GOSUB 1	' Execute subroutine 1
ENDIF	' End of the IF block
ENDWHILE	' Go back to WHILE statement
END	' End of program 0
SUB 1	' Subroutine start line, subroutine 1
XV1	' Move X-axis to V1 target position
WAITX	' Wait for X-axis move to complete
V1=V1+1000	' Increment V1 by 1,000
WHILE DI1=1	' Wait until the DI1 is turned off (prevent multiple increments)
ENDWHILE	' Go back to WHILE statement
ENDSUB	' End of subroutine 1

V[*var*] (*read*) Get a variables value (acts like a function)

V[*var*] = *value* (*write*) Set a variables value.

Variables are defined with a number (0 to 99) for 4 axis controller and (0 to 64) for a 2 axis controller. Variables are 32 bit signed integers only. Ascript currently does not support floating point.

WHILE [<i>comparison</i>]	Start WHILE loop and test for comparison to exit loop
ENDWHILE	Return to WHILE (loop)

Example:

WHILE DI1=1	' Wait until the DI1 is turned off (prevent multiple increments)
ENDWHILE	' Go back to WHILE statement

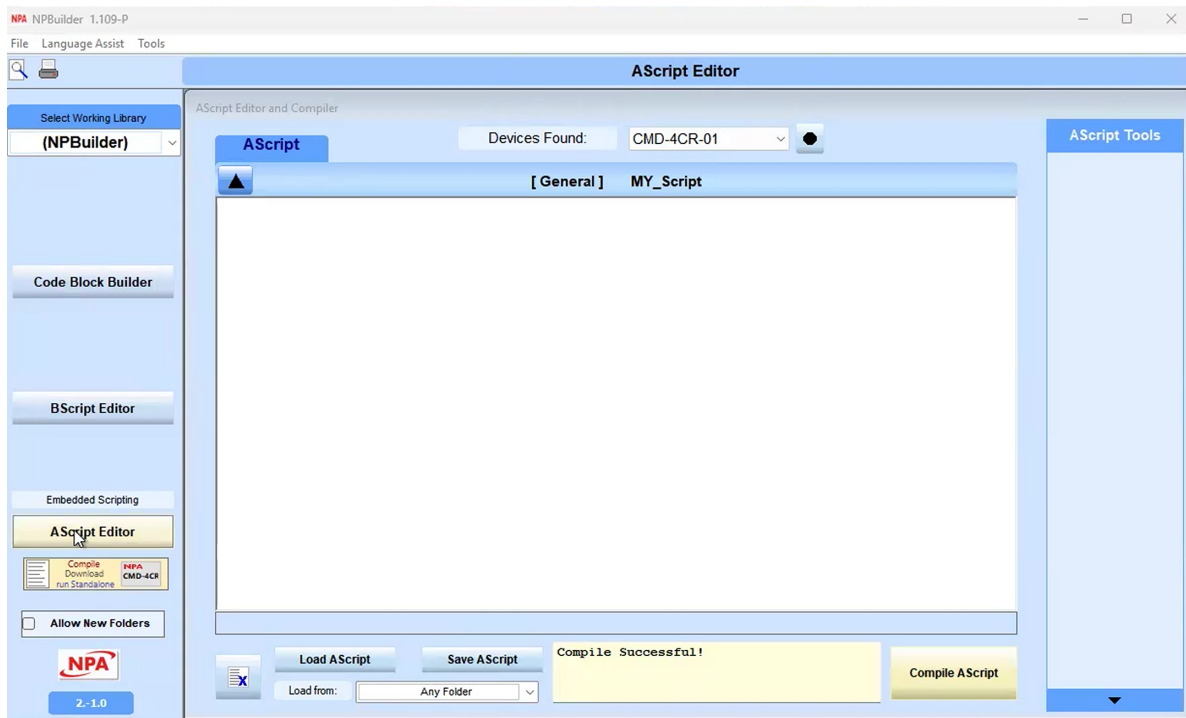
Continues executing the program after the **WHILE** statement when the conditional statement is true. Repeat until **WHILE** condition is false, and then execute program after **ENDWHILE**.

Conditional operands available include: =, >, <, >=, <=, !=, <>

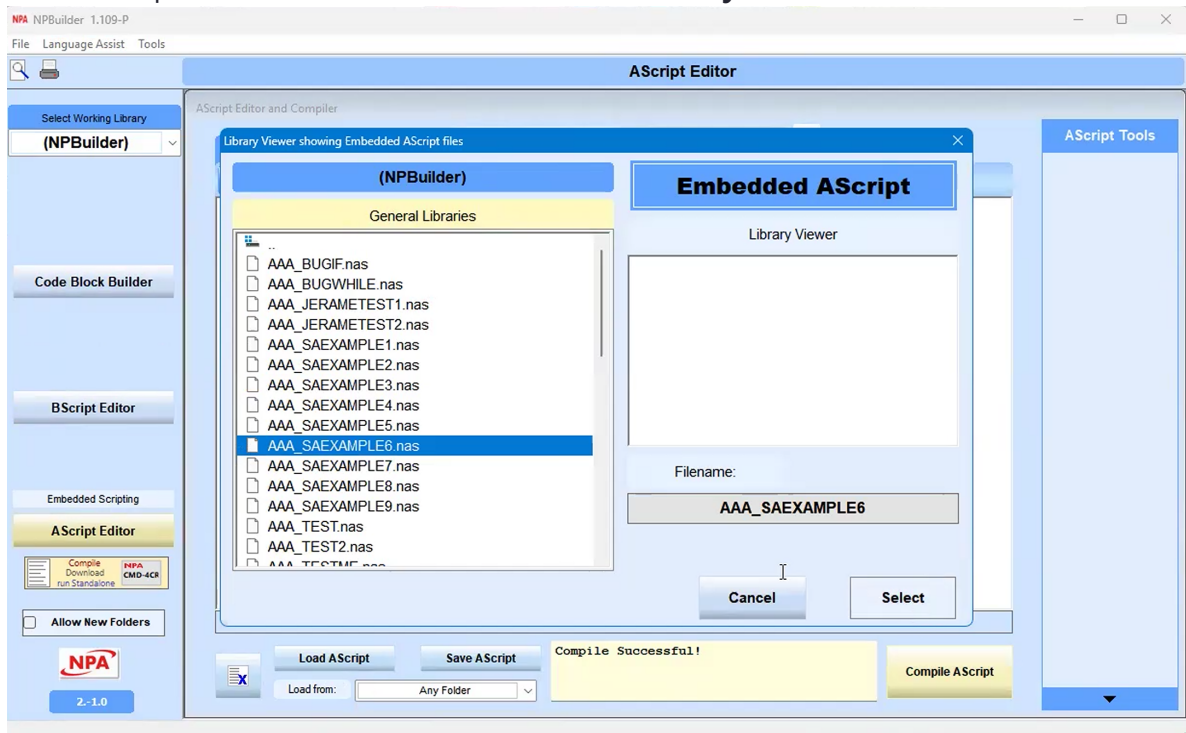
1.3 Create AScript

Example of an AScript Program

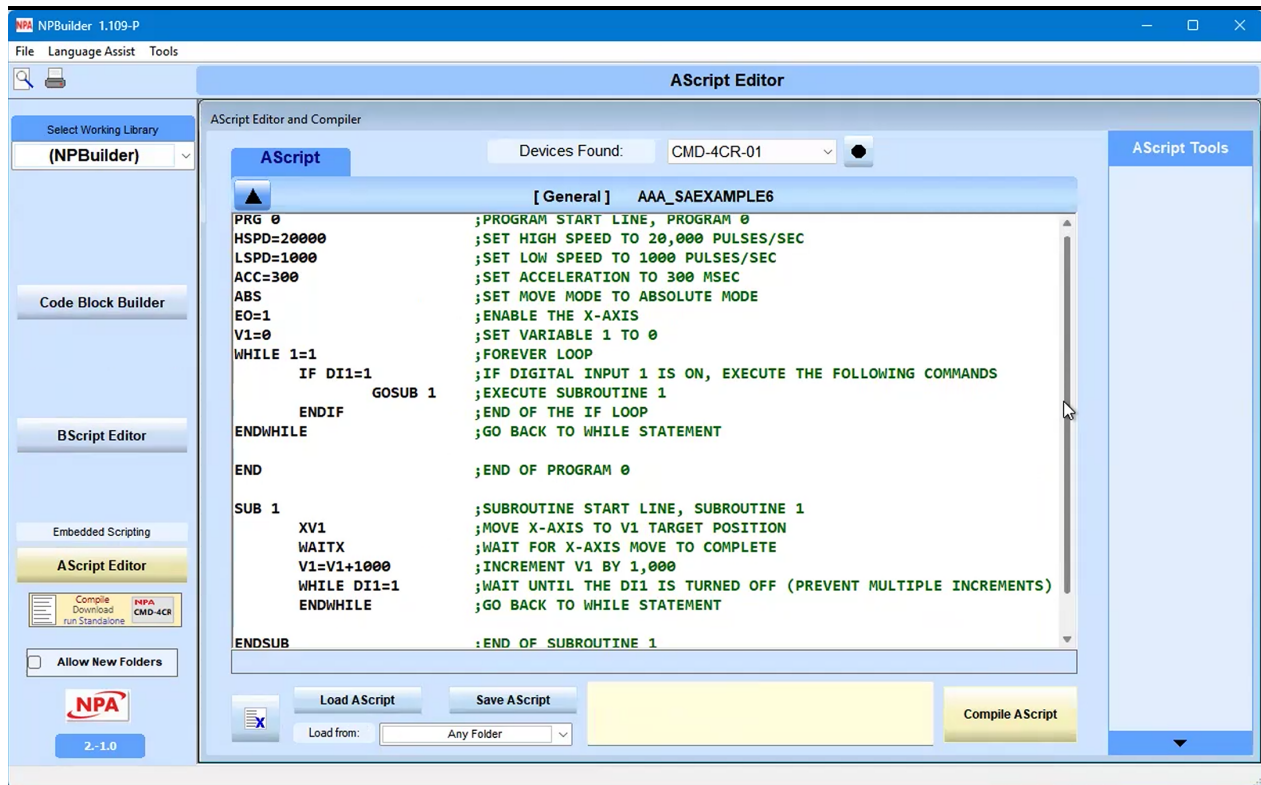
1. Click the **AScript Editor** button to open the editor.



2. Create a script or click the **Load AScript** button to open a script.
3. Select a script, then click the **Select** button under **Library Viewer**.



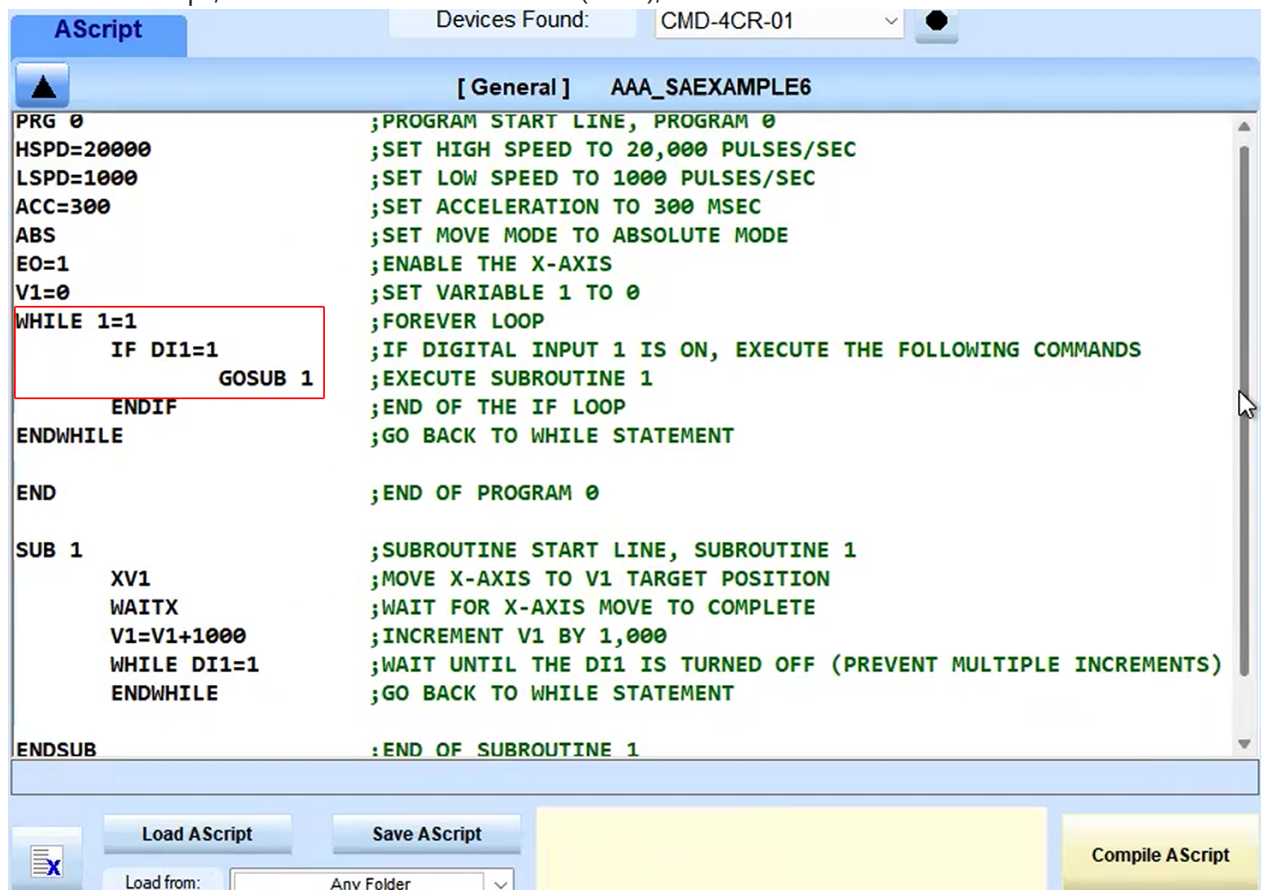
AScript example



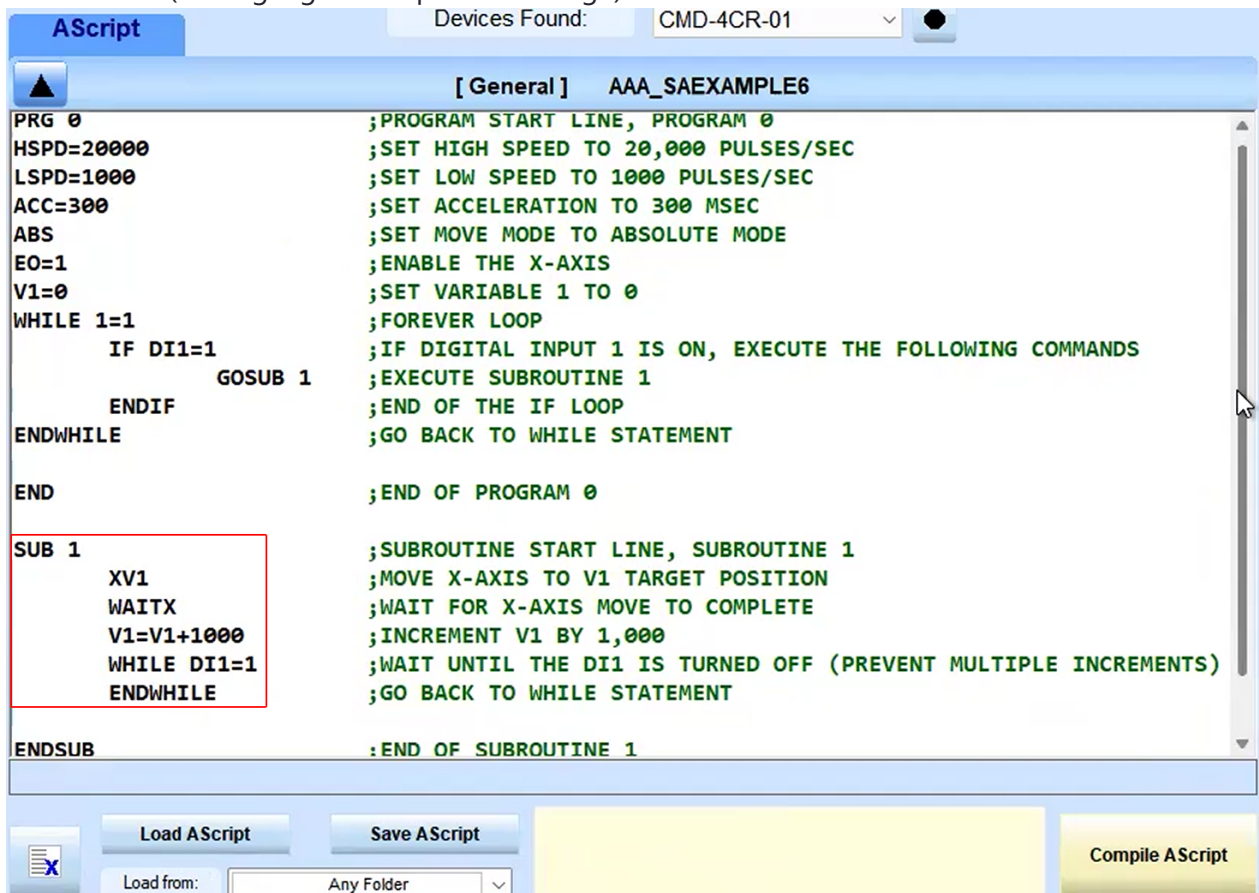
In this example:

WHILE statement, **IF** statement, **GOSUB** statement

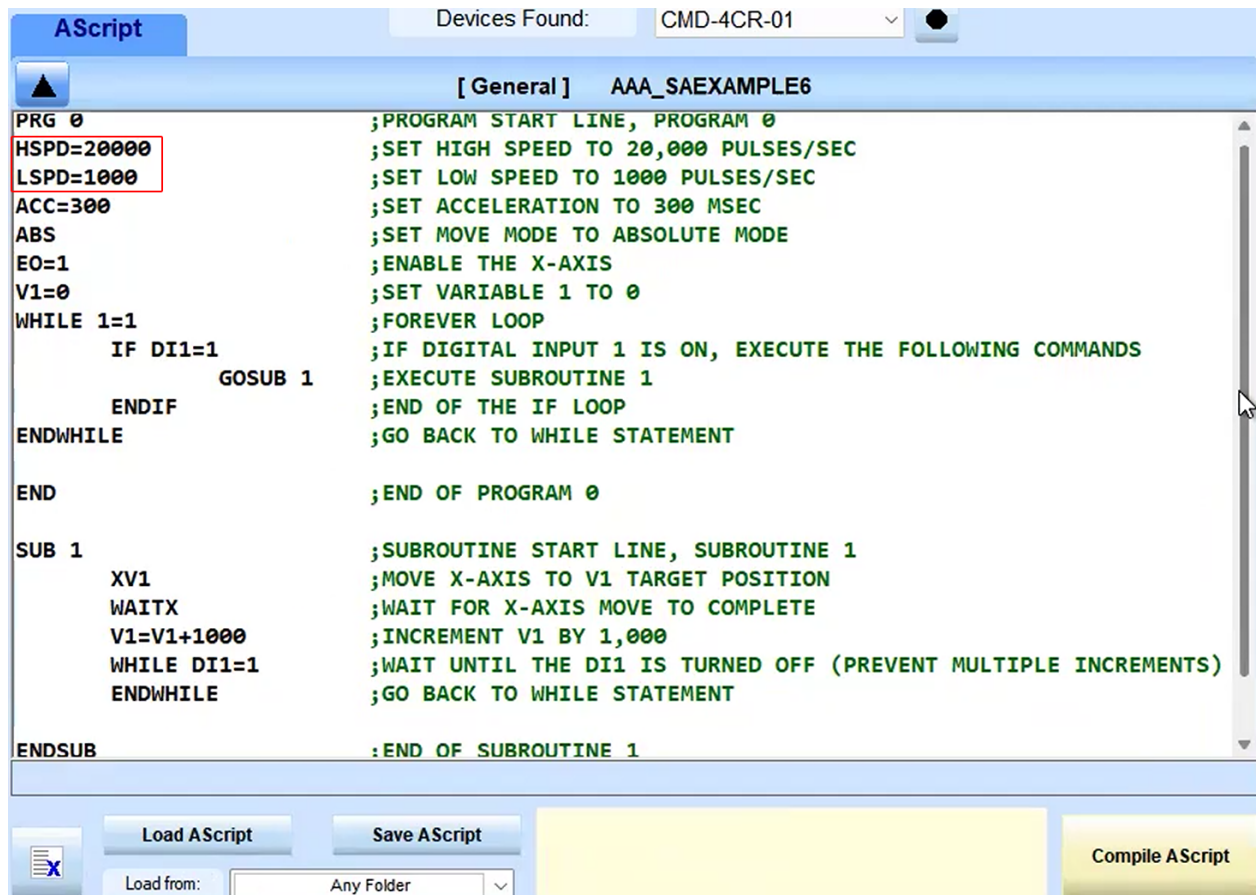
Note: In AScript, subroutines are numbered (0-32), not named.



SUB routine (call highlighted in previous image)

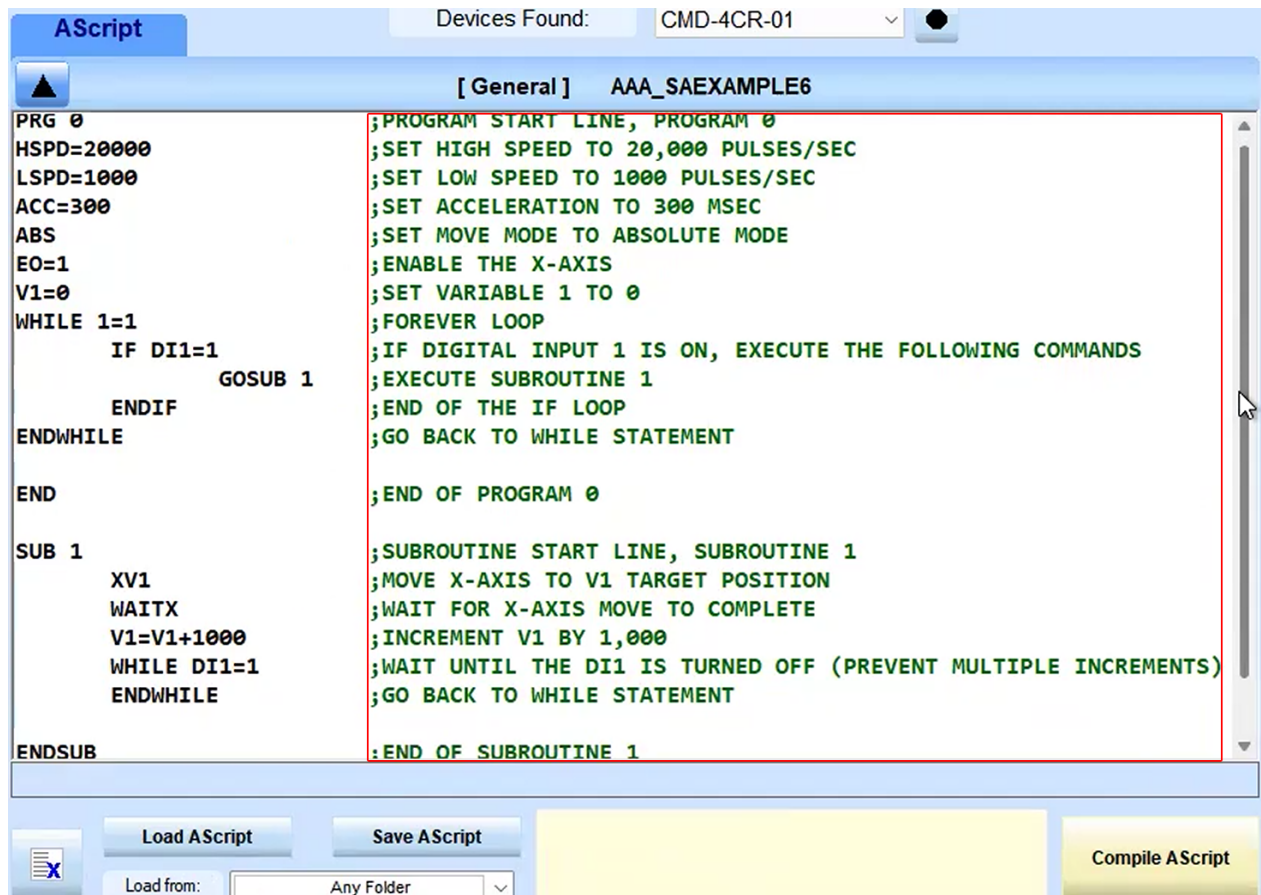


Motion control example of high speed (**HSPD**), setting low speed (**LSPD**), a command setting acceleration (**ACC**)



Commenting Ascripts

Comments are any characters after a semicolon ; (in this example) **or** character or a single quote ' character.



See the Commander Manual for [ASCII Language Specifications](#).

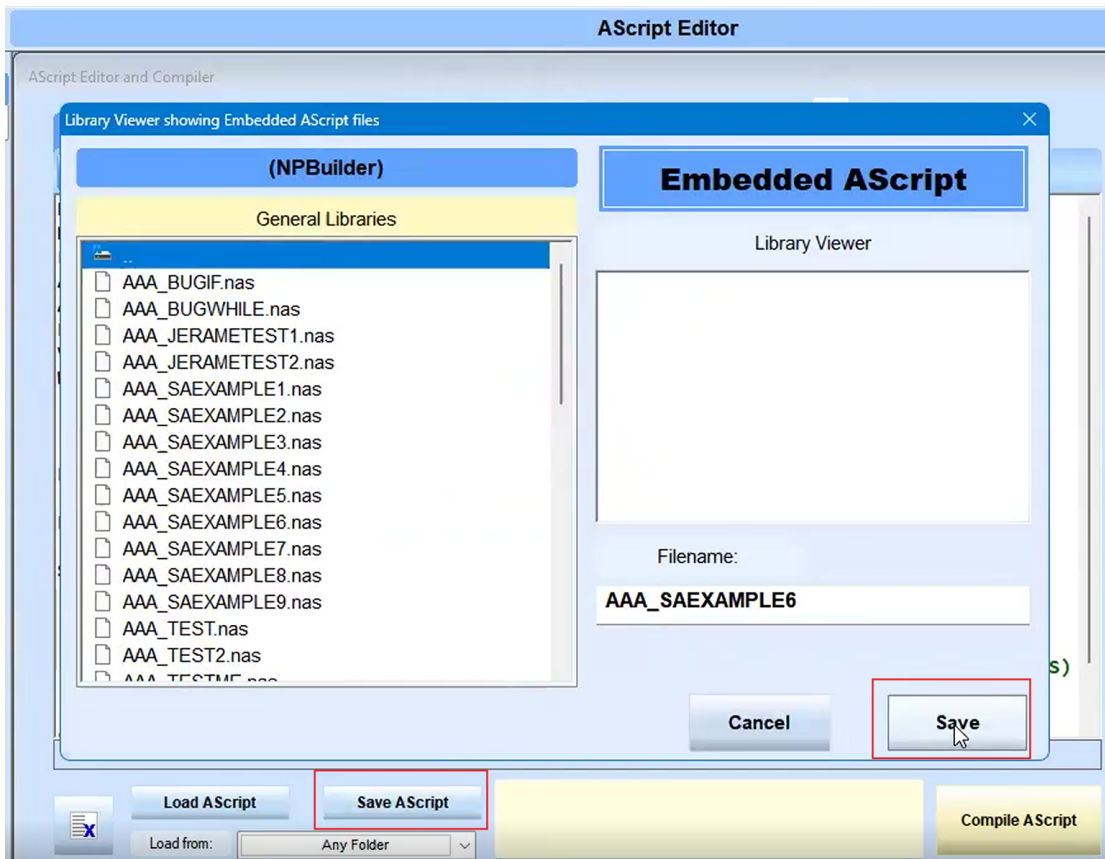
See also the [Commander Reference](#).

1.4 Compile and Run AScript

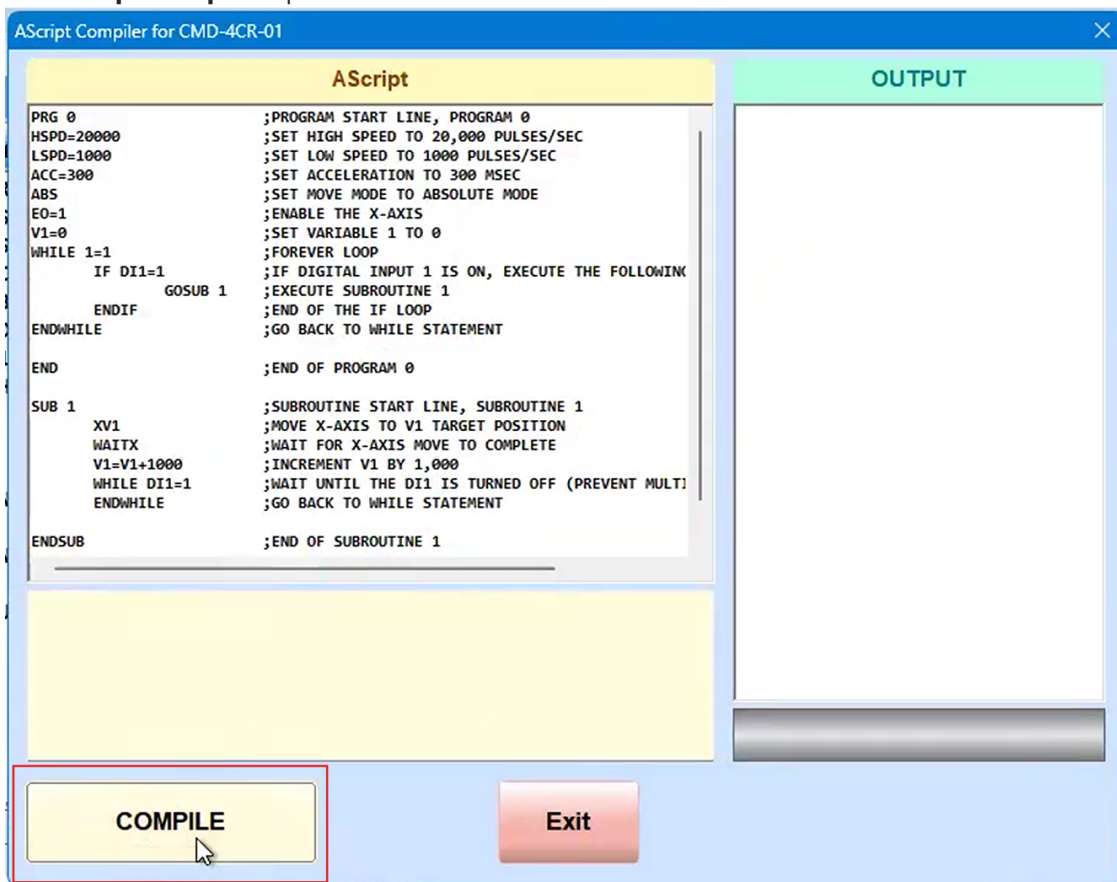
Compile and Run AScript

After your AScript program has been written, save and compile.

1. Click the **Save AScript** button.
The **Library Viewer** screen opens.
2. Type a Filename.
3. Click the **Save** button.

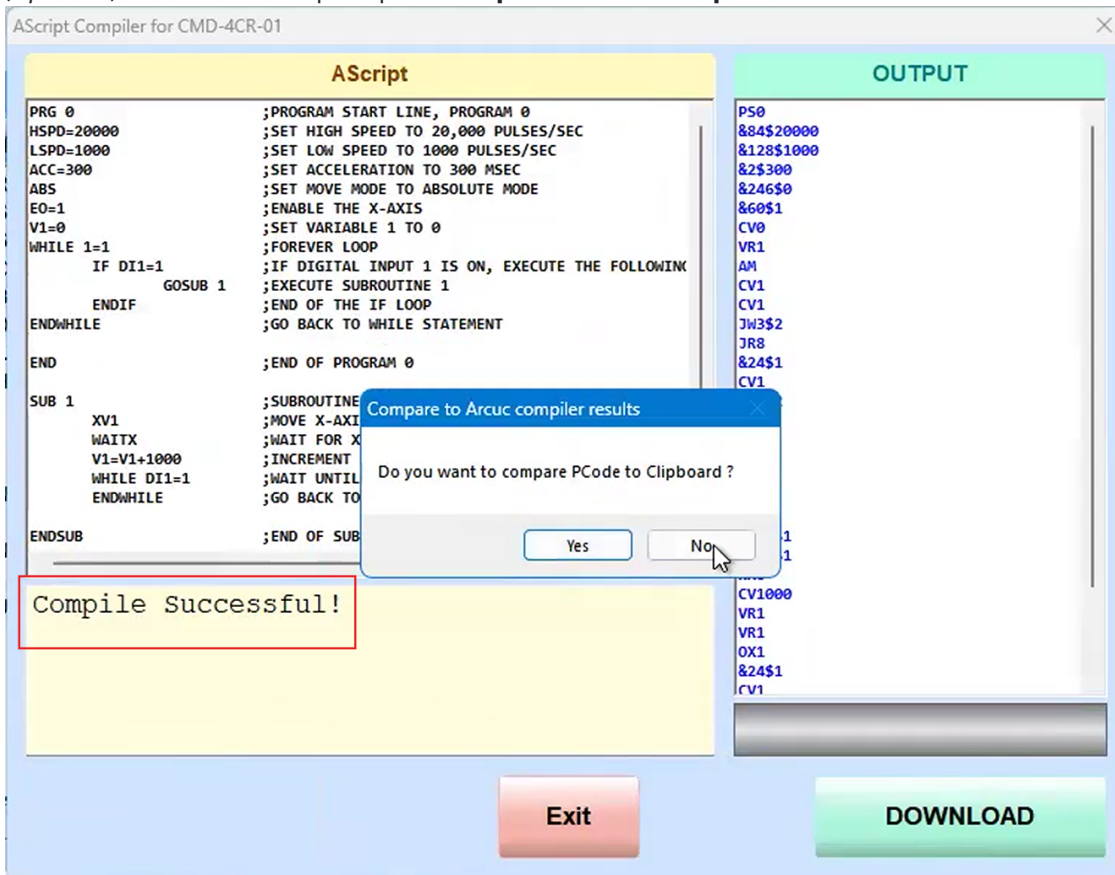


4. Click the **Compile AScript** button.
The **AScript Compiler** opens.



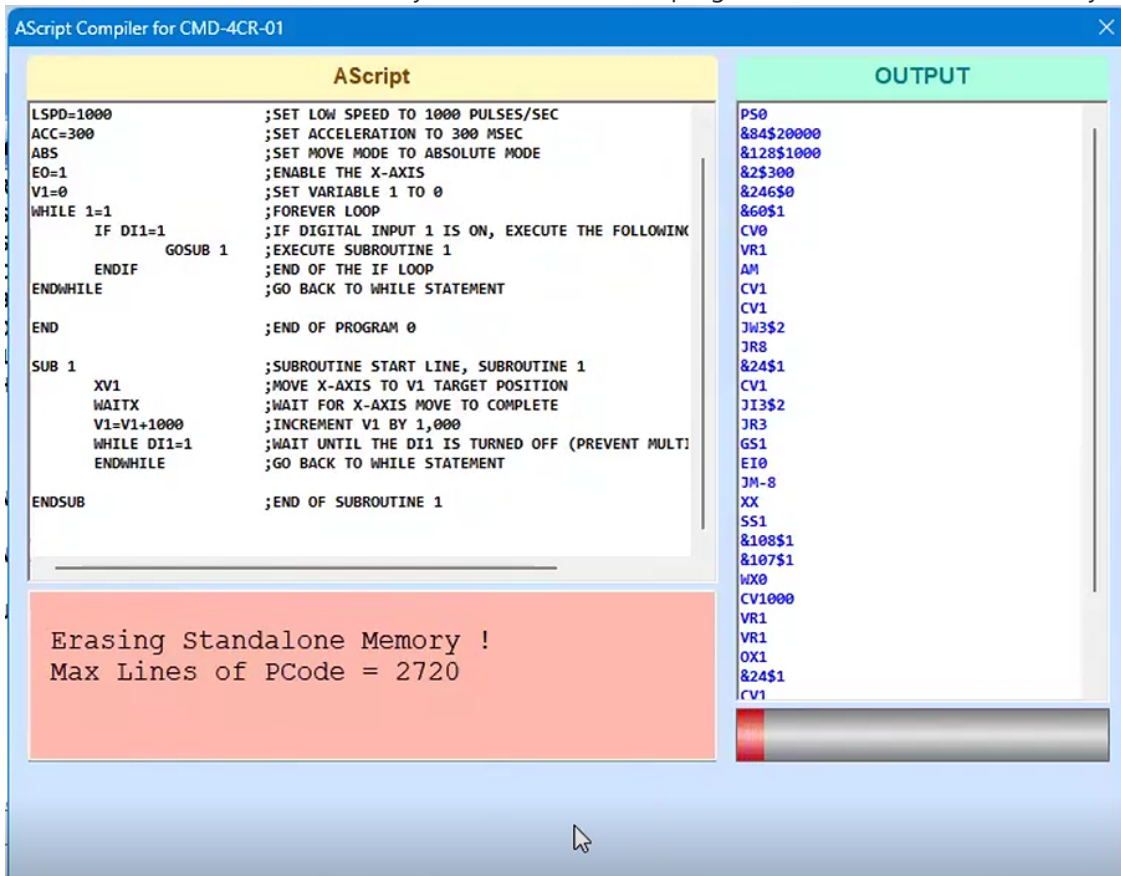
5. Click **Compile**.

6. (Optional) Click **Yes** on the prompt to **compare PCode to Clipboard**.

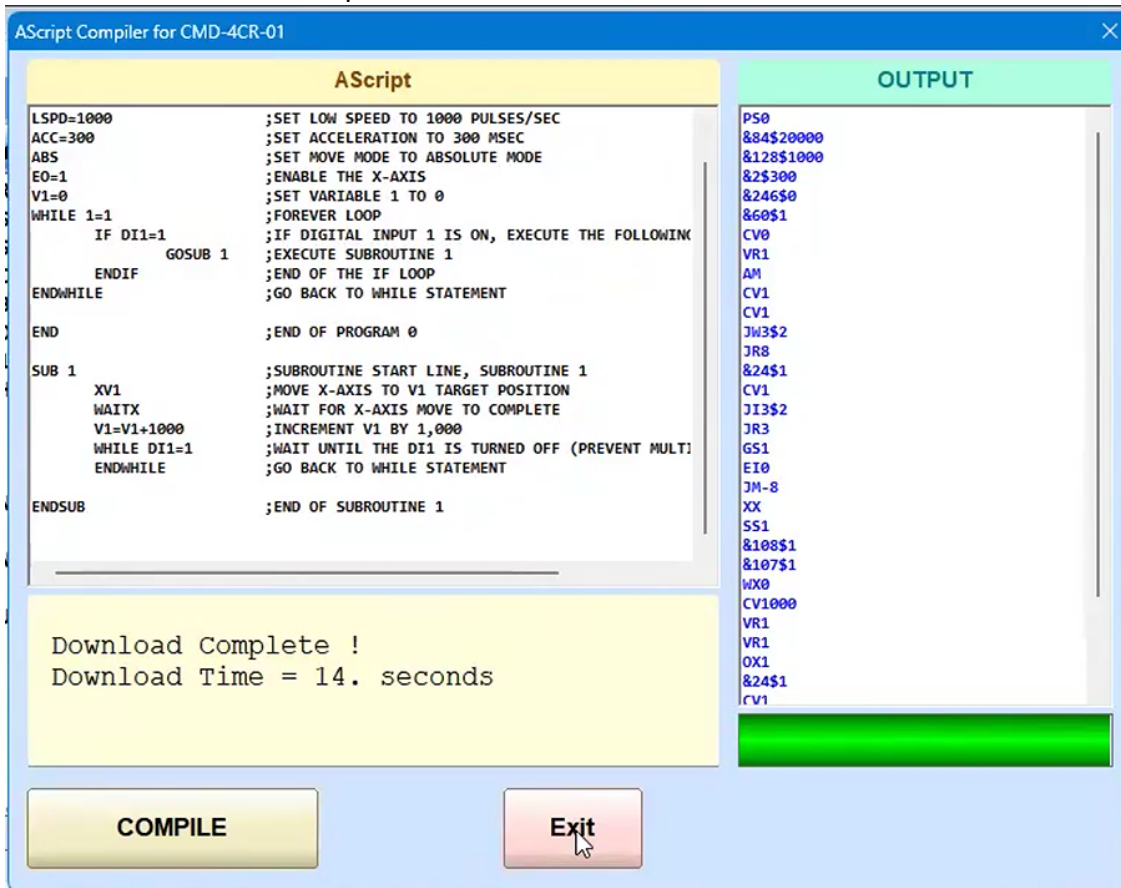


7. Click the **Download** button.

This erases the Commander memory then downloads the program to the Commander memory.



8. When the download is complete, click **Exit**.



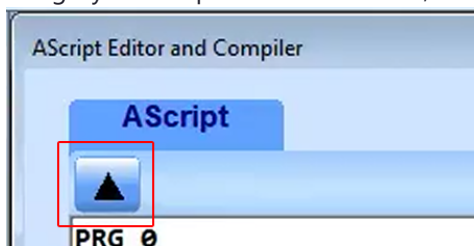
When the Commander starts up, it can automatically run this program.

1.5 Additional AScript Options

Additional AScript Options

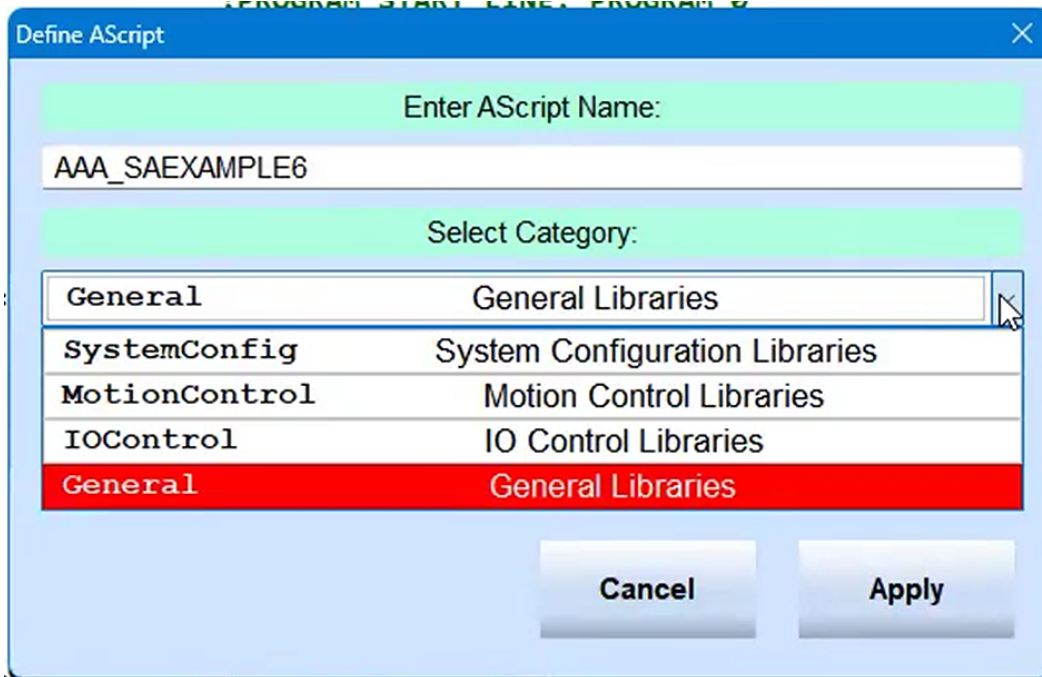
Other AScript Editor features.

This button displays an input window where you can define the script name, in which main folder or category the script should be stored, and provide a description.

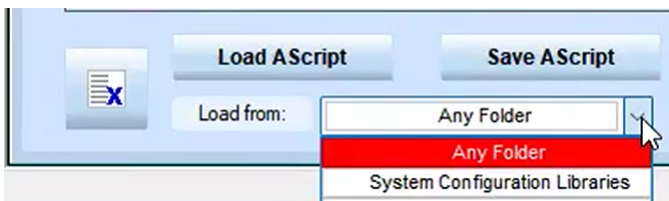


There are four key subfolders:

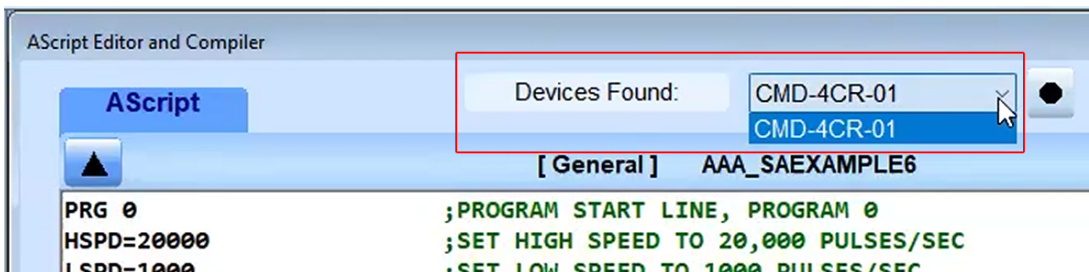
- System Configuration
- Motion Control
- IO Control
- General Libraries



When loading scripts, you can select any of the folders or select one folder to work directly from.



You can use this dropdown to see and select connected devices.

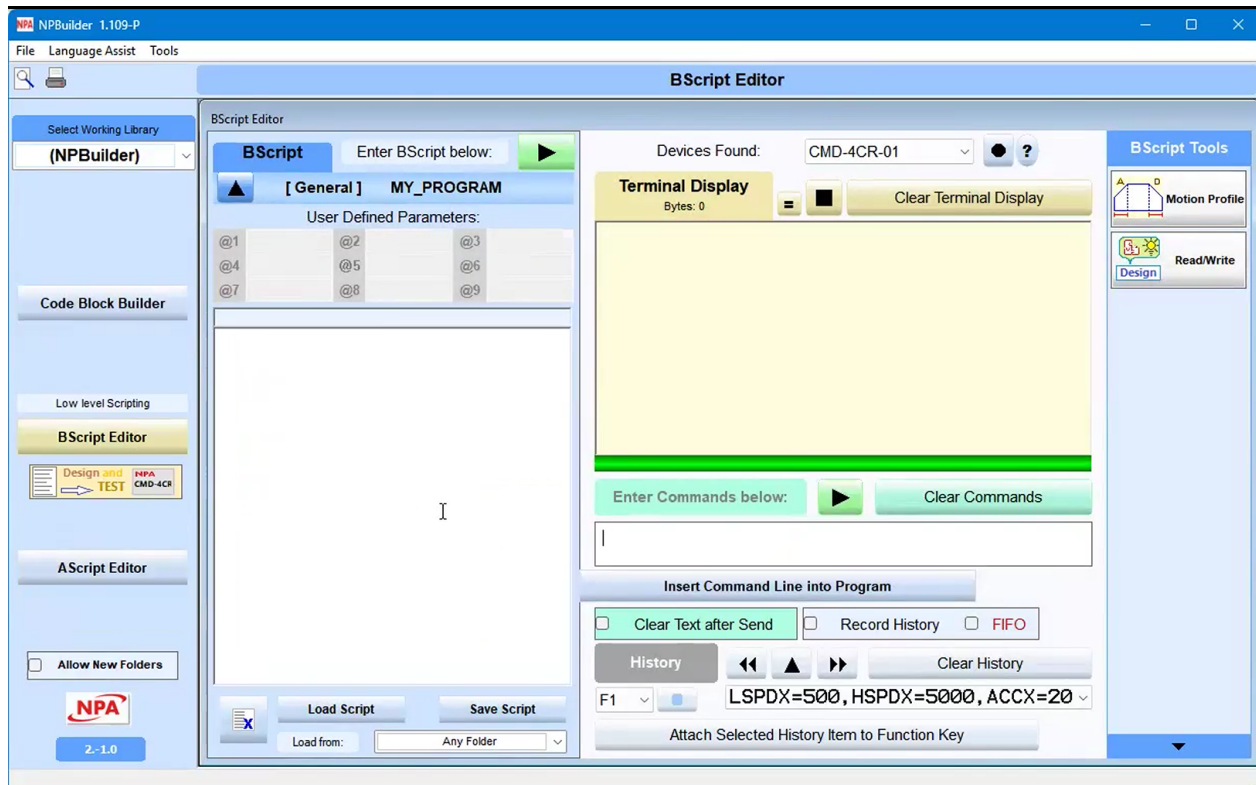


In this example, only one Commander is listed, but if multiple Commanders are connected all of them will be listed. The button next to the list is a **Reset** button to reset the Commander list to make sure it's up to date.

2.0 BScript Editor

BScript Editor

The BScript editor allows you to make more complex scripts than the terminal program, and BScripts can be saved, run, and tested.



BScript is a hybrid scripting language for external use (a PC sends commands via USB to the Commander). It consists of the entire Commander ASCII command set and a number of special BScript commands which are never sent to the Commander but talk to the Bscript interpreter. Commands like REPEAT and UNTIL are examples of BScript commands. This provides *minimal* logic capabilities when running an ASCII command script.

Bscript is not intended to be a programming language, since it is interpreted. It supplies missing pieces necessary to write a usable ASCII script which can be run in real time from NPBBuilder for testing purposes.

BScript has limitations. For example the REPEAT / UNTIL loops can only be nested up to 3 levels. This limit is imposed by the interpreter. The Bscript editor does some colorization of text to distinguish between BSCRIPT commands and ASCII commands and also comments. Comments are all text after the ' (single quote) character. While Ascript supports both the ; and ' characters for comments, Bscript only supports the single quote character. The BASIC language typically uses the single quote for comments.

BScripts support Parameter Macros. They are referenced as the @ character followed by a number, so there are only 9 Macros allowed (@1 to @9). The macros work with a simple text replacement method. If you define the macro @1 to be the text **JX+** then whenever the macro @1 is found in the script it will be replaced with the text **JX+** before the script is past to the Bscript interpreter.

BScripts are run by clicking the **Play** (or RUN) Green Button.

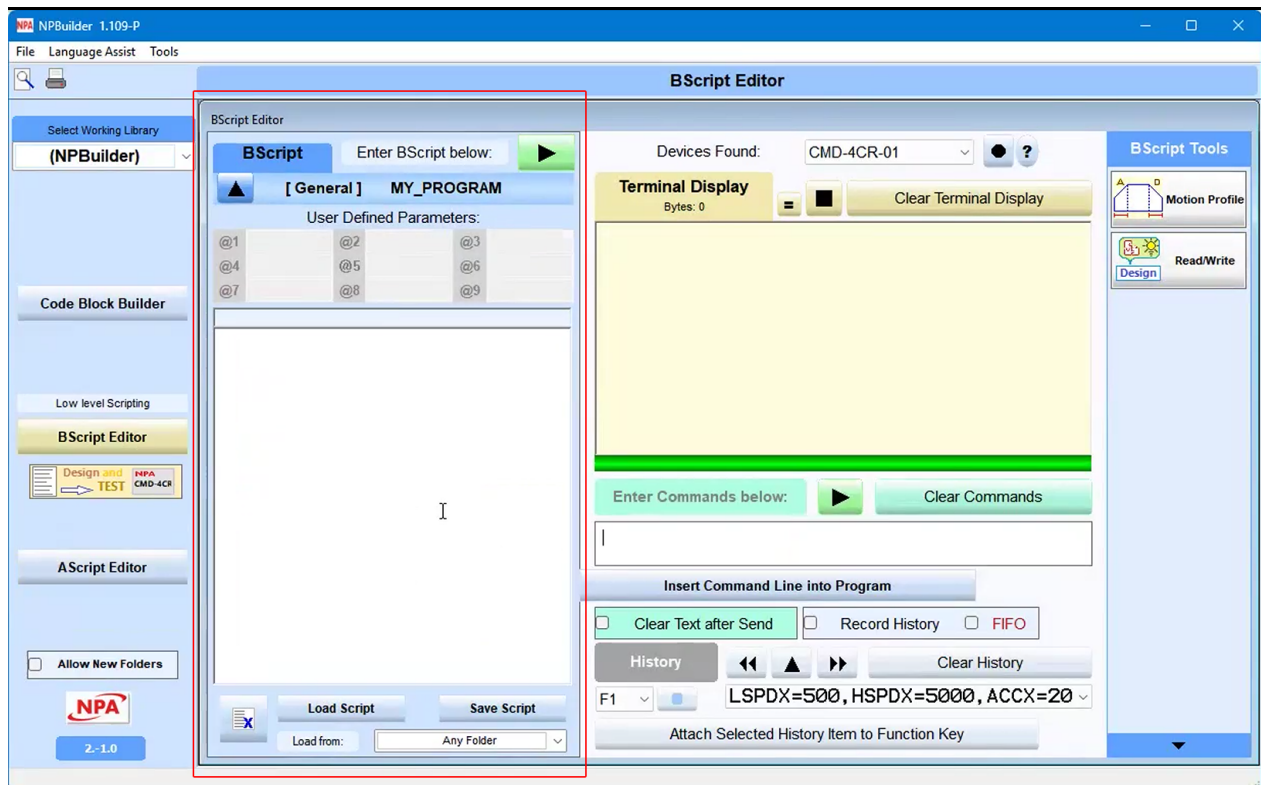
The right side of the Editor is similar to the Commander Terminal Program so you can enter ASCII commands via a terminal. Results of script execution are shown in the terminal window.

BScripts

The BScript Editor is ASCII commands sent from APC directly to the commander. An external device such as APC or even a stand alone system on a chip board like a Raspberry Pi could send

commands directly to the commander over Ethernet, USB, or serial port.

Many BScript Editor functions are the same as the [AScript Editor](#).



In this section

- [Parameter Descriptions](#)
- [Terminal Display](#)
- [BScript Tools](#)
- [BScript Commands](#)

See also

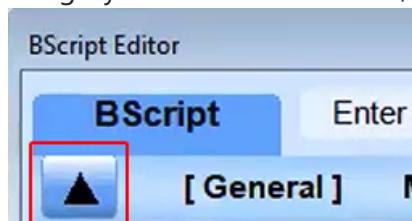
- [Commander Control Utility](#)
- [NPA Terminal Utility](#)

2.1 Parameter Descriptions

Parameter Descriptions

When a script has fixed values, it can only be used for one thing. Parameters can be changed and edited, so a single script can be used many times over with different values or parameters passed to it.

This button displays an input window where you can define the script name, which main folder or category it should be stored in, and provide a description.

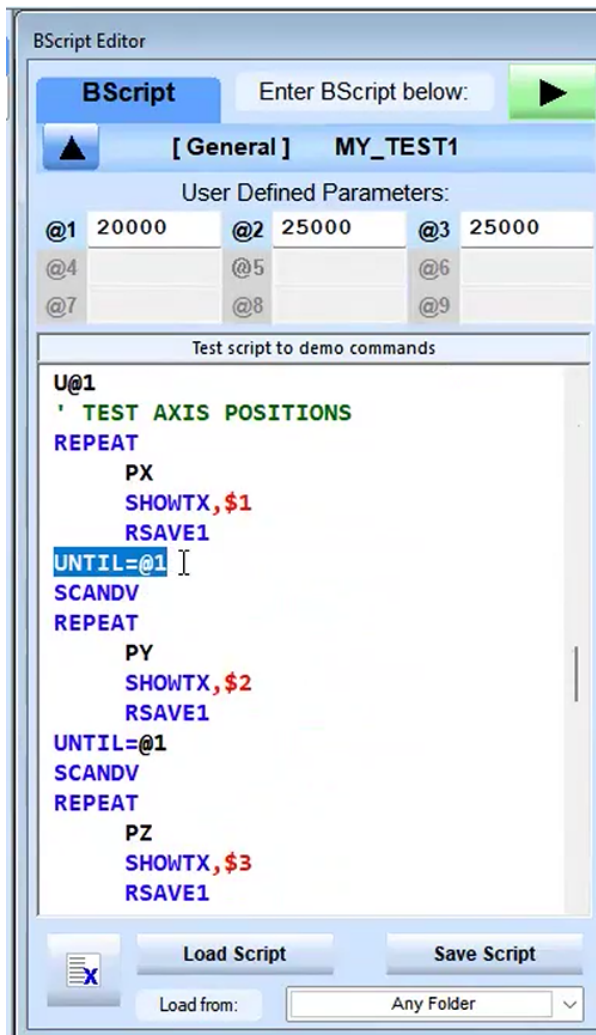


In the BScript Editor, the input window also includes Parameter Descriptions.

The screenshot shows a dialog box titled "BScript Procedure Info" with a sub-header "MY_TEST1". It contains several input fields and a table. The "Enter BScript Name:" field contains "MY_TEST1". The "Select Category:" field shows "General" selected in a red bar, with "General Libraries" also visible. The "Enter BScript Description:" field contains "Test script to demo commands". Below this is a "Parameter Descriptions:" section with a table of 9 rows. The first three rows are filled with parameter names: "@1 New Position", "@2 High Speed", and "@3 New XYZU Int Position". The remaining six rows are empty. At the bottom right are "Cancel" and "Apply" buttons.

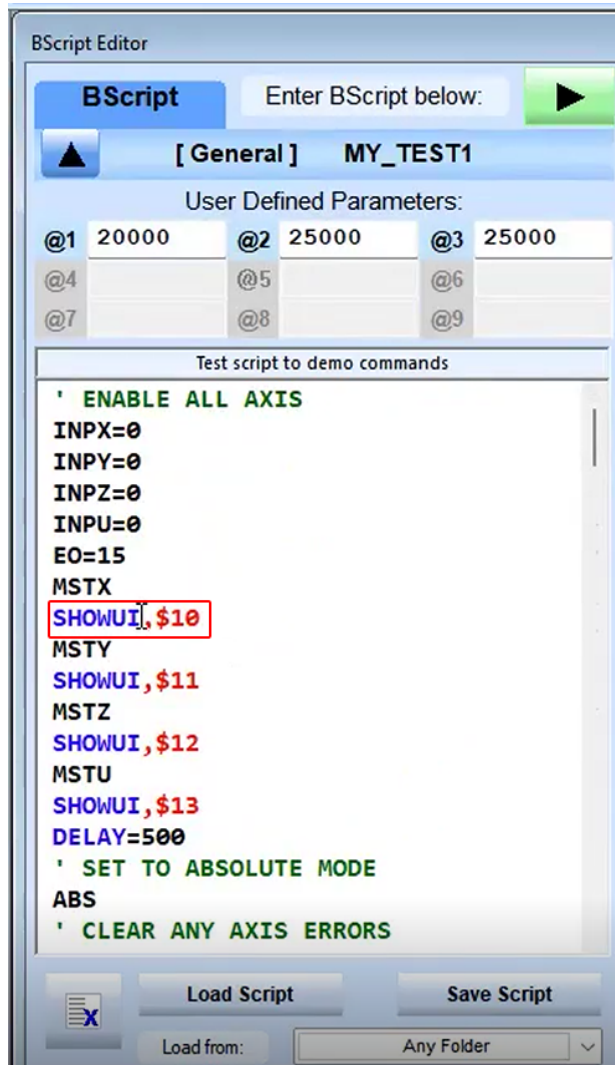
Parameter Descriptions:	
@1	New Position
@2	High Speed
@3	New XYZU Int Position
@4	
@5	
@6	
@7	
@8	
@9	

These are parameters that can be defined in a script; use the @ (at) character with a number from **1** to **9**. The value in the input box is pasted throughout the script. A parameter makes a script multi-usable. When the script is run, the @value in the script is replaced by the defined parameter, in this example, **20,000**.

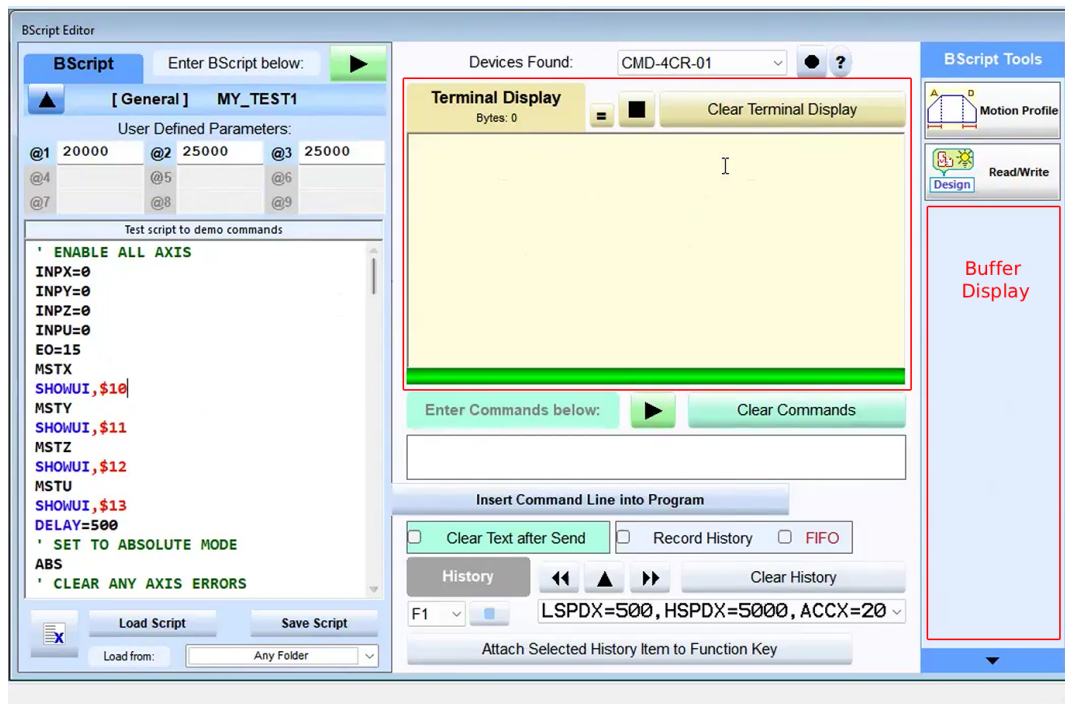


Dollar sign \$

If the command is in black, it's an assumed ASCII command, but BScript is a minimal scripting interpreter that runs your program. It has additional commands that are not Commander commands or ASCII commands for the Commander. These commands are never sent to the Commander; they tell the scripting engine to do something. In this example, it is a **SHOW** command.



This **SHOW** command says show UI, meaning show the the last reply value. The Commander will send a reply, **OK** or the current value. This command requests the last ASCII command sent be displayed. A display window opens as this program runs and shows a number of memory buffers, variables that might call them, and where a value can be put and seen in real time.



MSXT will return a value. This command shows the value converted to an unsigned integer. The **,\$10** tells where to put it. Without the **,\$10** it would be displayed only in the terminal window.

The terminal window is for commands that are executed and it shows it in real time. The format of the commands sent can be changed.

2.2 Terminal Display

The BScript Editor allows you to create programs by sending different commands and experimenting with them in real time, and to determine how your device is working with the commander.

On this page

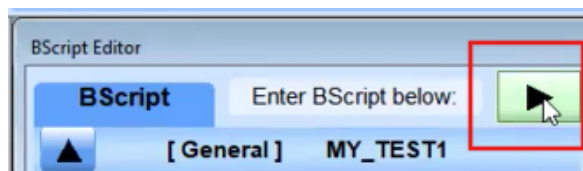
- [Terminal Display](#)
- [Change Format](#)
- [Enter Command Line](#)
- [Insert Command Line](#)
- [Additional Options](#)

Terminal Display

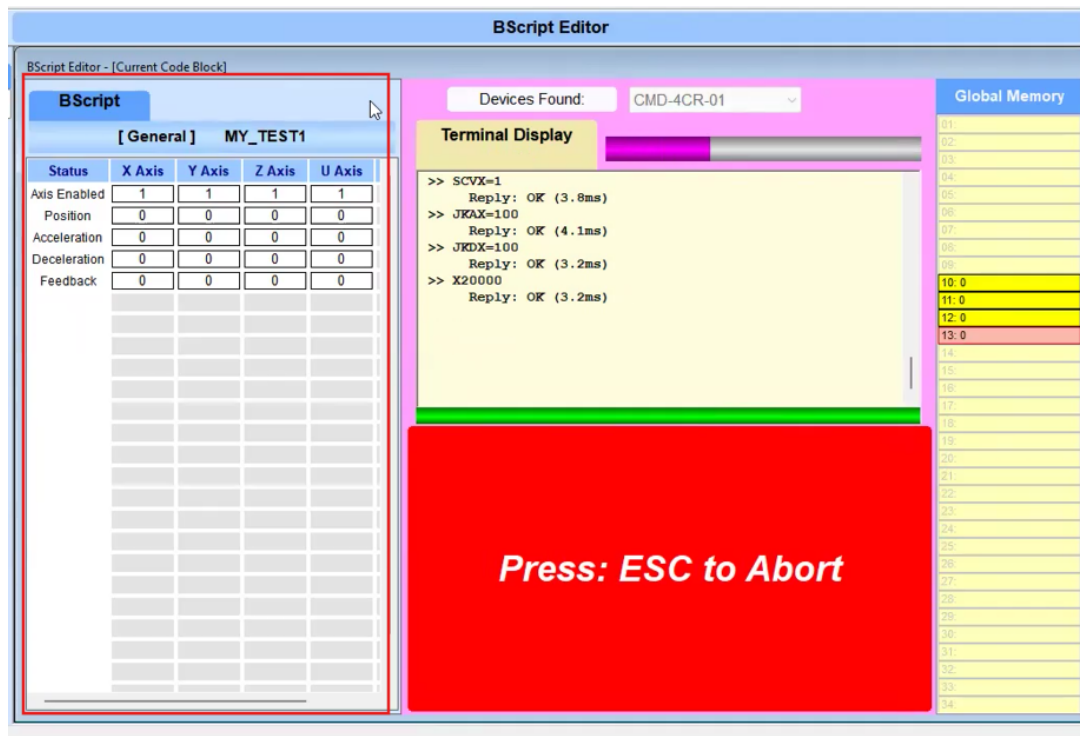
The terminal window displays commands executed in real time. The standard format shows everything sent from the script.



Click the **Play** button to run the script.

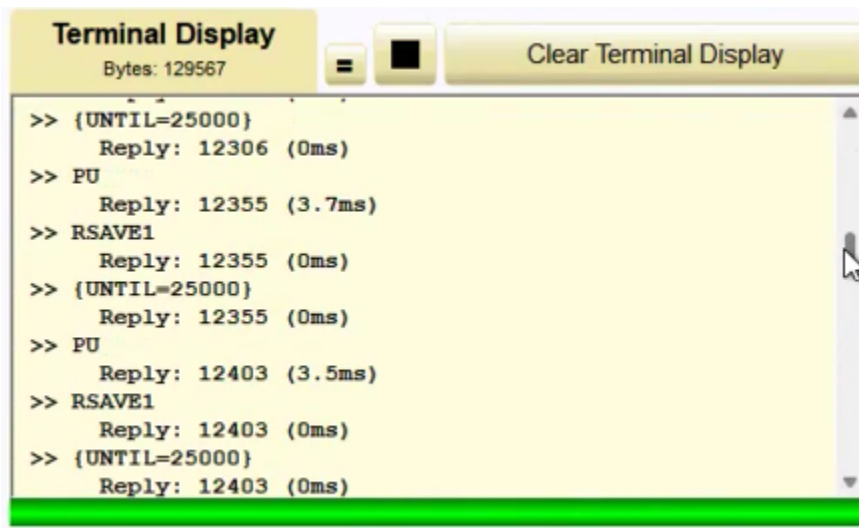


As it runs, a BScript command displays current values at certain points in this section.



This shows the buffer memory where you can send the variables.

When the run is complete, scroll to see the executed commands in the **Terminal Display**.



The display also shows the amount of memory used in storing the information in the terminal display.



Press **ESC** on the keyboard at any time to abort the run.

Change Format

The format determines what is sent to the terminal window. Use this button to change the format:



<format option details>

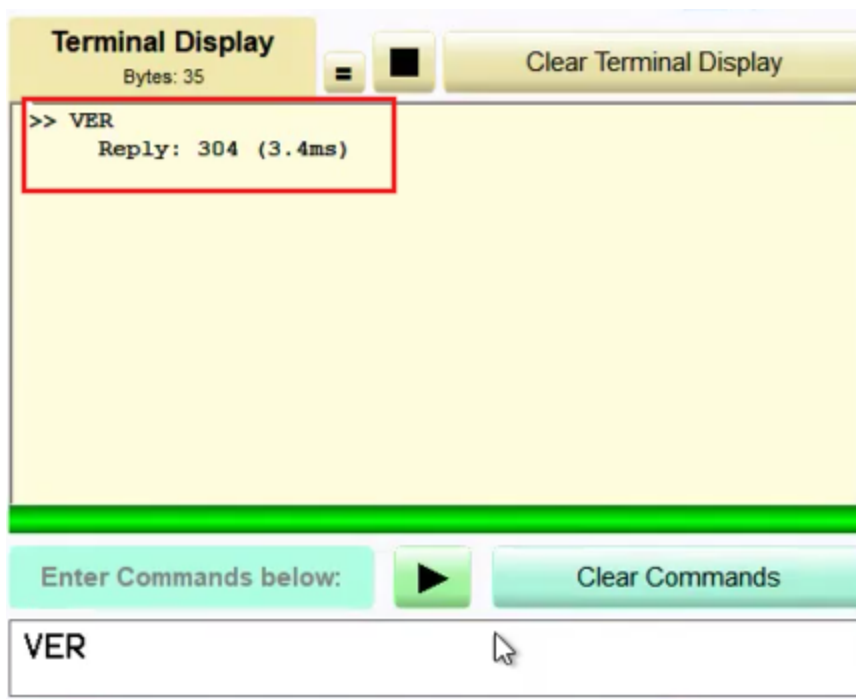
Enter Command Lines

You can send a single command, or a small group of commands by typing here:



Click the Play button, or press enter on the keyboard, and is sent, in this example, VER which requests a version.

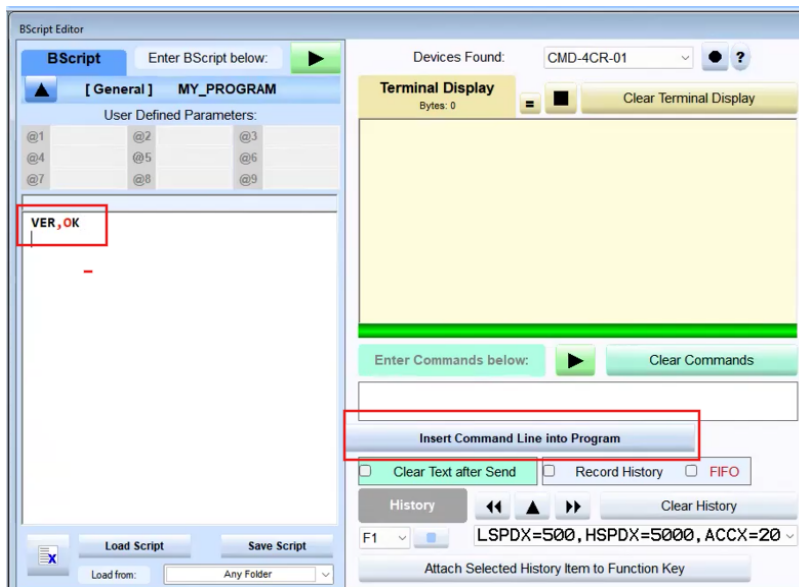
The Reply is shown in the Terminal Display, and indicates the amount of time it took to be executed (3.4 milliseconds).



Separate multiple commands with commas (,).

Insert Command Line into Program

Click the **Insert Command Line into Program** button to add the command to a script.

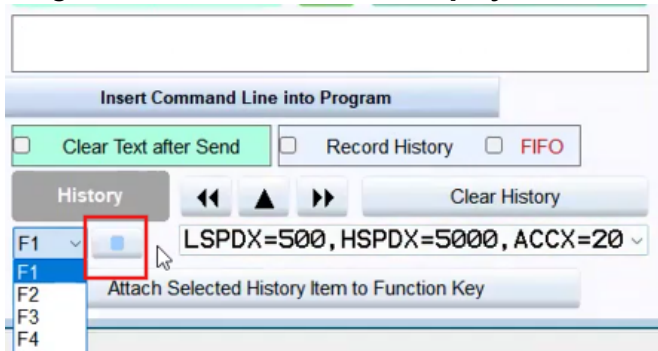


Additional Terminal Display Options

- Click the **Record History** button to record the history. Select the **FIFO** checkbox to record first in first out.
- Click the **Clear History** button to clear the history.
- Select different history command(s).



- Assign content in the **Terminal Display** to a function key.



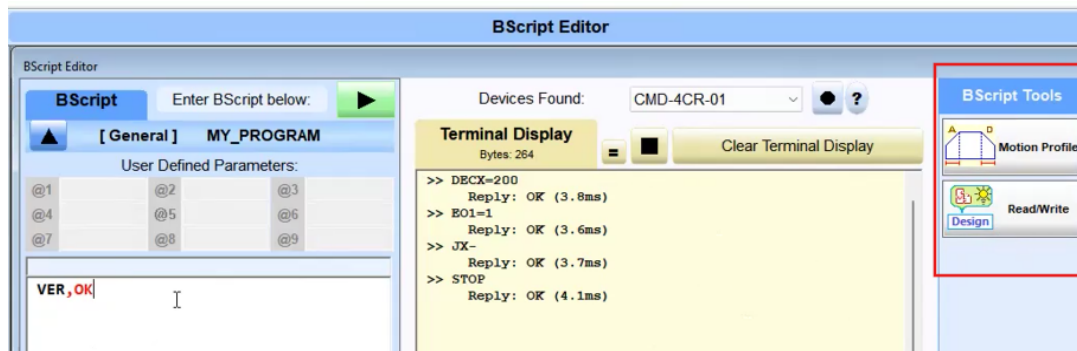
See [Commander Control Utility](#) and [NPA Terminal Utility](#) for additional details.

2.3 BScript Tools

BScript Editor Tools

NPBuilder uses open API plugins. Plugins are used to generate code and provide additional features.

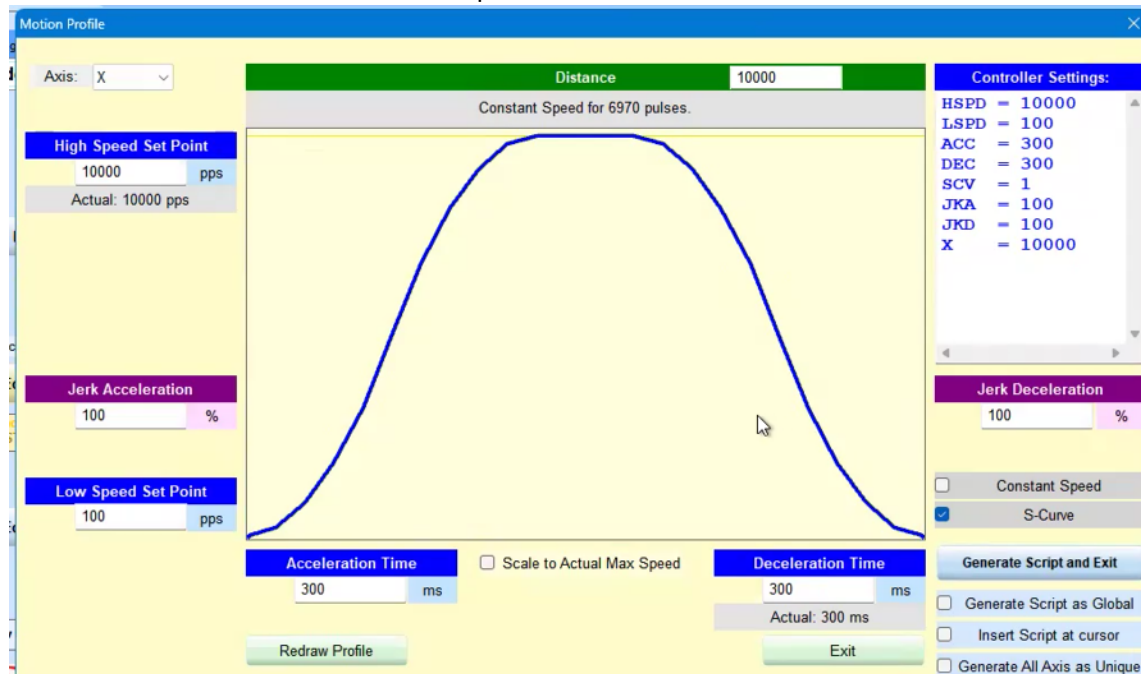
Note: NPA plans to develop plugins over time to improve code generation in NPBuilder. You can also create your own plugins to create customized code generators to meet your specific needs.



Motion Profile Tool Example

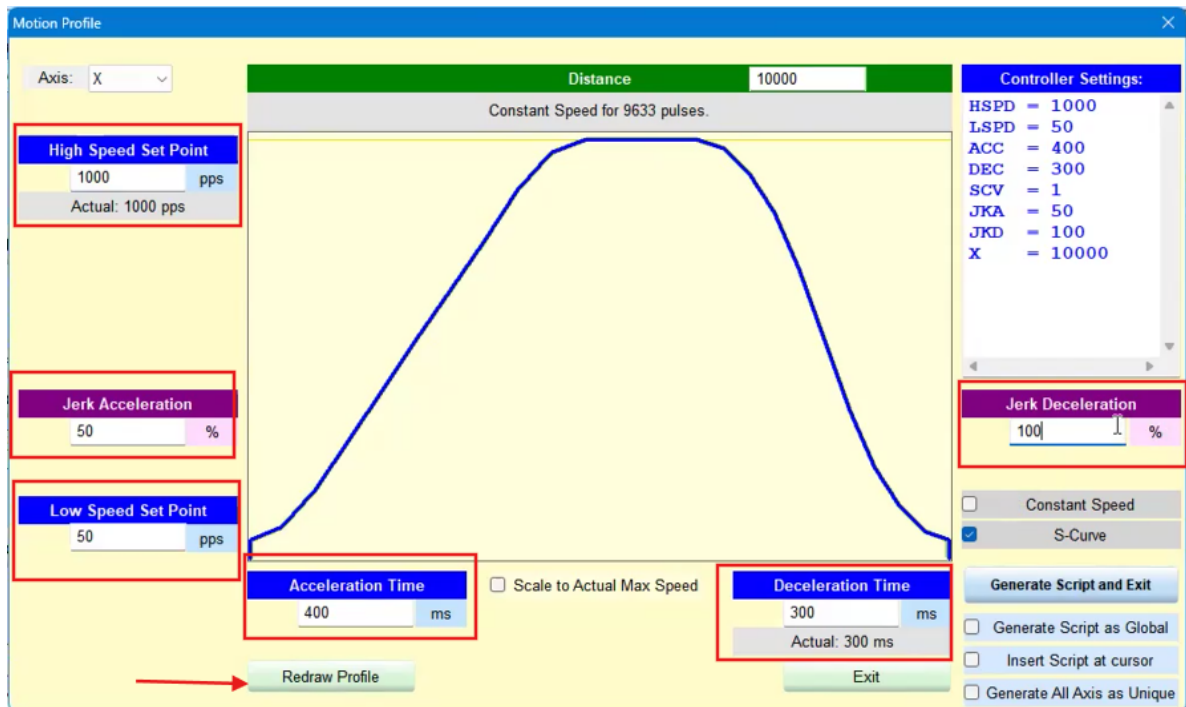
This Motion Profile tool is not a built in NPBuilder tool. It is a Dynamic Link Library (.dll) written with a plugin API. NPBuilder looks for plugins in the plugin folder when it starts, then loads available tools into the toolbar. This example illustrates use of the Motion Profile plugin.

1. Click the **Motion Profile** button to open.

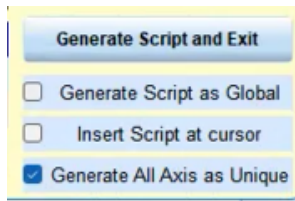


Use this plugin to set various parameters, such as **High Speed Set Point**, **Low Speed Set Point**, **Jerk Acceleration**, and others as available in the plugin.

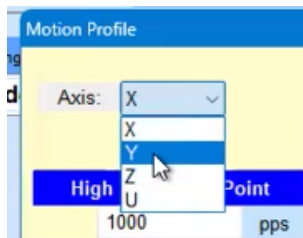
2. Click the **Redraw Profile** button after making changes.



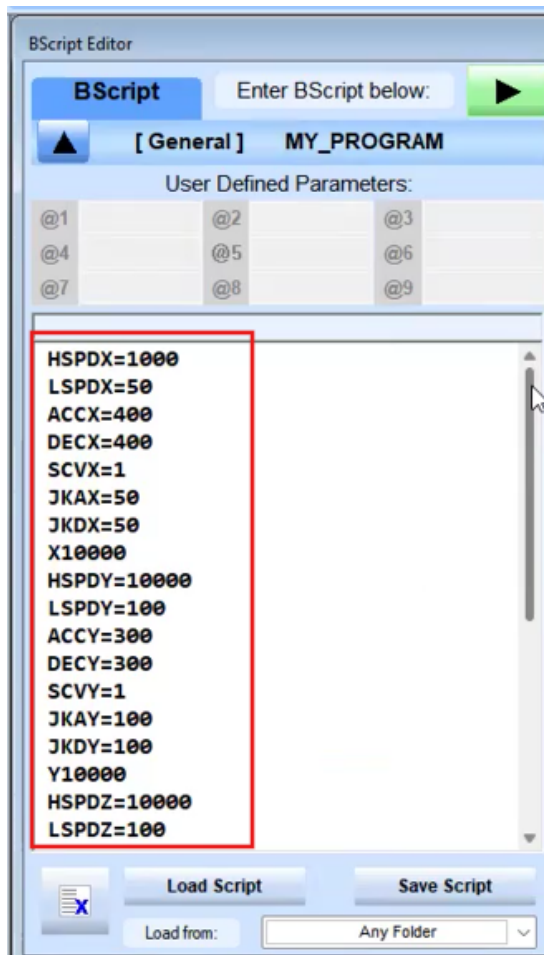
3. Select a **Generate Script** option; in this example **Generate All Axis as Unique**.



4. Select an **Axis**.



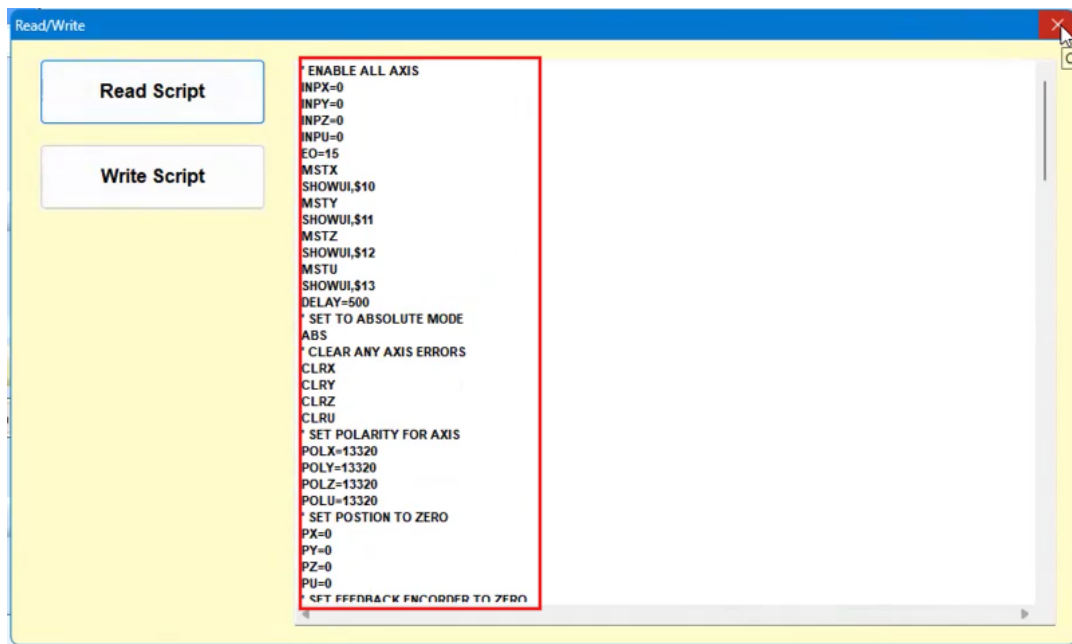
5. Click the **Generate Script and Exit** button.
The ASCII commands required to create the defined profile have been generated as a script.



Read/Write

This Read/Write tool uses commands in the API to read from the BScript Editor. This simple example is intended to illustrate the use of API plugins.

1. Open a script in the BScript Editor.
2. Click the **Read/Write** button.
3. Click the **Read Script** button in the **Read/Write** screen.
The script is displayed and can be edited in this view.

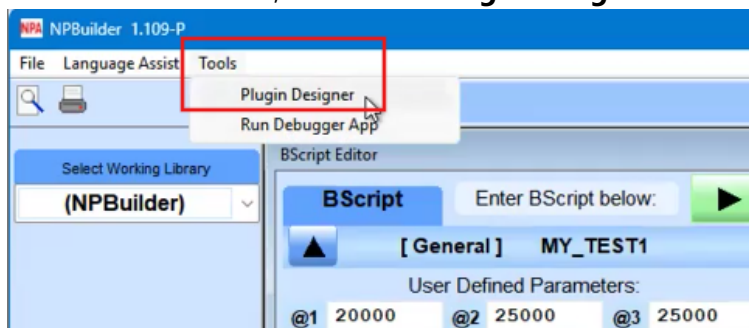


2.4 Plugin Designer

Plugin Designer

NPBuilder contains a Plugin Designer tool. It has the ability to generate code in power basic, which is A Win 32 native code compiler that can be used to write plugins. You can write plugins in a variety of different languages and design your form.

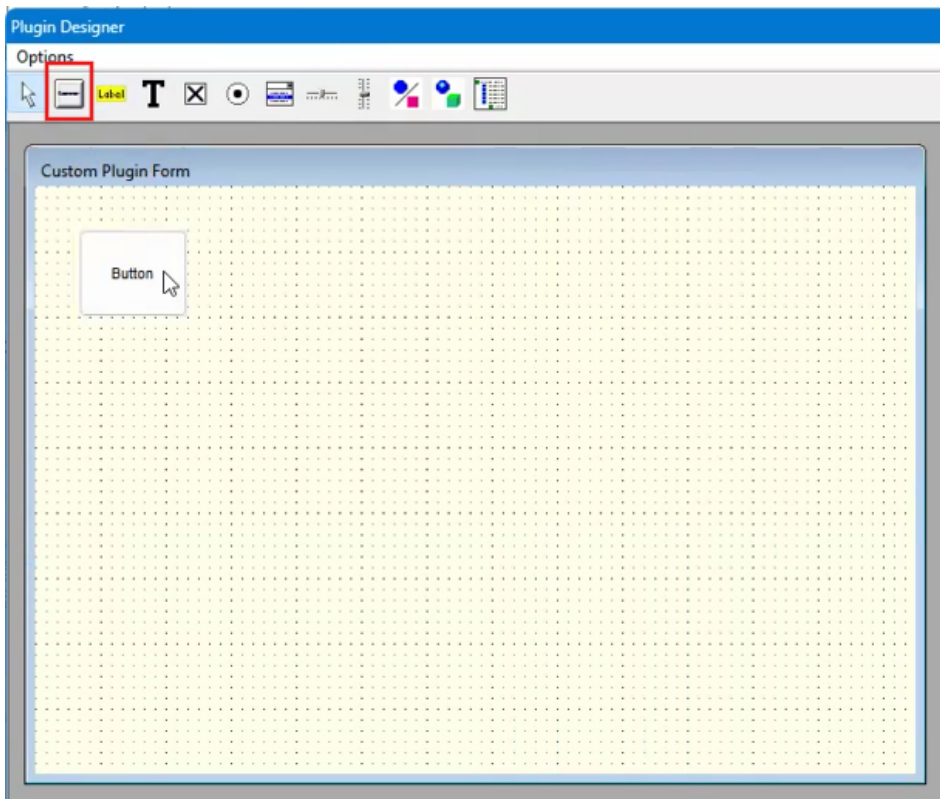
1. Click the **Tools** menu, then select **Plugin Designer**.



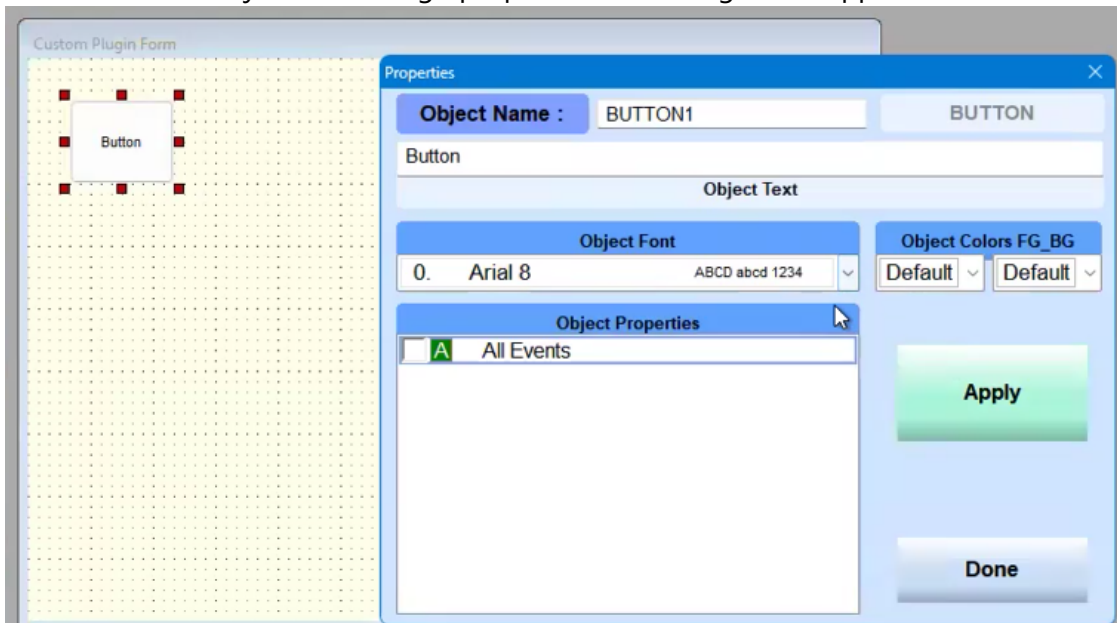
The Options menu has various elements you can use for design.



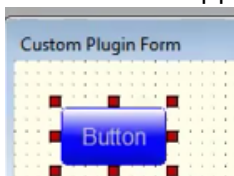
2. In this example, click the **Button** option to add a button to the form.



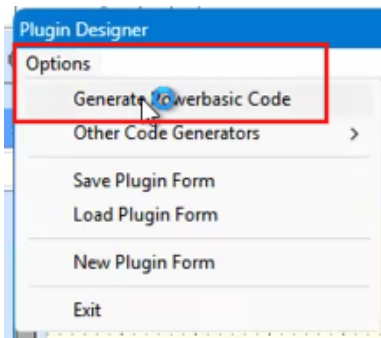
3. Double-click an object to to assign properties and change their appearance.



4. Click **Done** to apply changes.

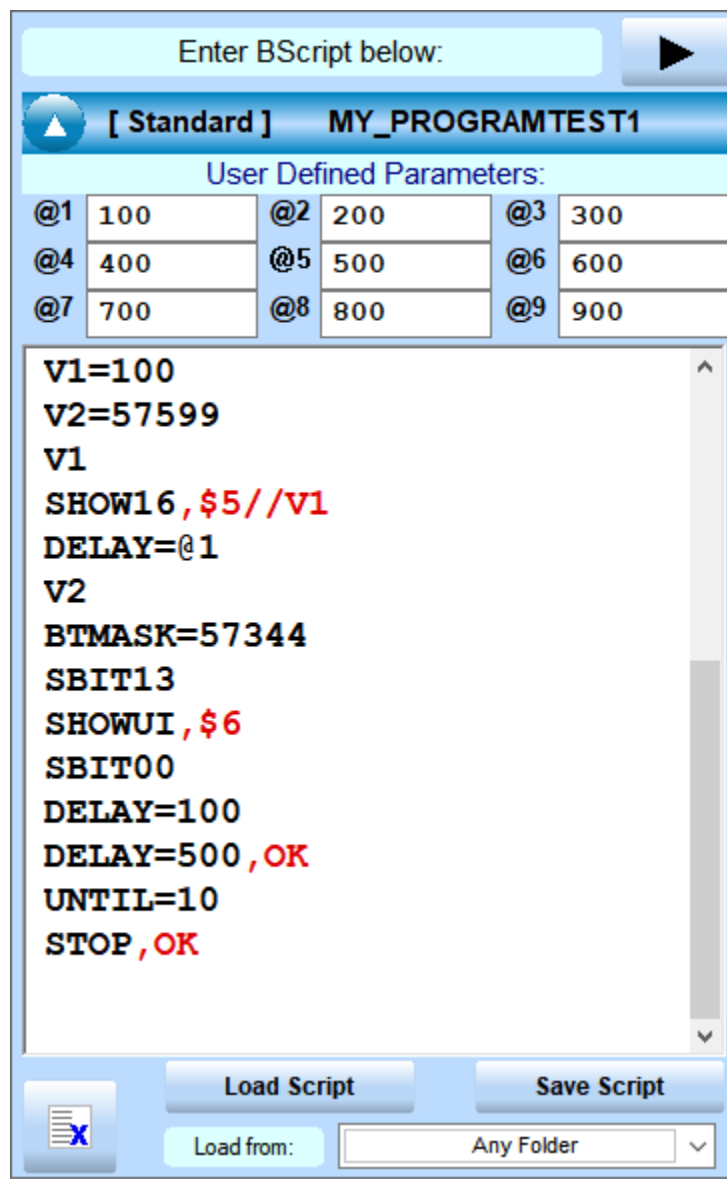


5. Click Options, then select Generate Powerbasic Code.
Note: NPA plans to add other coding languages.



2.5 BScript Commands

The full BScript mode requires the commands be entered in the Script Editor in the NPBuilder application:



BScript programs use a comma to separate a command with the expected reply value. Each command line ends with a carriage return (non-visible) character, instead of a comma like is done in direct mode.

- BScript or Controller Command, Expected reply
- BScript or Controller Command, Expected reply
- BScript or Controller Command, Expected reply
- BScript or Controller Command, Expected reply

Controller ASCII commands are sent to the motor controller module without modification (other than parameter macros which are swapped in before the commands are sent to the controller).

BScript extension commands are preprocessed by the BScript engine and most do not send an actual Controller ASCII command to the motor controller module. BScript commands do effect code logic (*i.e.* REPEAT , UNTIL loops) and can communicate with user interface components (in No-Code development) or may be passed to user written code (*i.e.* Low Code development). When running BScript command programs in the ASCII Script Editor directly in the NPBuilder application it may send data directly to the History window or the Global Memory Storage window.

You can ignore the expected reply if you don't want a Pass/Fail test on the command.

The expected reply is compared to the actual reply as a text string, character for character and not a math comparison.

To force the BScript engine to compare the reply with the expected reply using a math comparison, one of the following Math Macros must be used as a prefix to the expected reply:

```

<=###          reply is less than or equal to expected value
>=###          reply is greater than or equal to expected value
*=###|###      reply is in the range between the two expected values
(inclusive)
==###          reply is equal to expected value and force a math
comparison
&=###          reply ANDed with current Bit Mask (BTMASK command)
is equal to expected value

```

Multiple comparisons can be made using the : (colon) character as an item separator.

Normally the value after the comma (*i.e.* PX=0,OK) is compared as a string regardless of what the value is. The reply is compared to the expected value character for character as strings.

You can put a math comparison macro (Macros are always 2 characters long) in front of the expected reply value to force math comparisons. They are as follows:

```

<=###          reply is less than or equal to expected value
>=###          reply is greater than or equal to expected value
*=###|###      reply is in the range between the two expected values
(inclusive)
==###          reply is equal to expected value and force a math
comparison
&=###          reply ANDed with current Bit Mask (BTMASK command)
is equal to expected value

```

Multiple comparisons can be made using the : (colon) character as an item separator.

Command, <=100 : >=200

The above would compare the return value and if less than (or equal) to 100 and also greater than (or equal) to 200 it would pass, otherwise it would set a fail flag.

When a math comparison is done (any above) the values are tested to see if they are using either

BASIC or C notations for Binary or Hex numbers as well.

App test for the following notations:

0x for C++ Hexadecimal (just x can be used as well)
 0B for C++ Binary (just B can be used as well)
 &H for Basic Hexadecimal
 &B for Basic Binary

2.5.1 BScript Extention Commands

Extended Time/Delay BScript commands:

DELAY	Delay (or wait) for 2 seconds
DELAYS	Delay (or wait) for 1/2 second
DELAYM	Delay (or wait) for 1 and 1/2 seconds
DELAYL	Delay (or wait) for 3 seconds
DELAY=#	Delay (or wait) for a user defined number of milliseconds within
	range from 50 to 60000 milliseconds (don't use comma to define #)

These are logic commands you can use in the terminal programs.

2.5.2 Extended LOGIC BScript Commands

These are logic commands you can use in the terminal programs. They are not autocontrol script commands, but actual program commands. They can be used when typing in commands via the command textbox too, but you have to type in a complete REPEAT loop code (start with REPEAT and end with UNTIL) all on one line separated by commas. Programs loaded in from files are what it is primarily designed for though.

REPEAT (starts a repeat cycle or loop)
UNTIL= ### (ends loop or repeats it again based on comparison of equal)
UNTIL< ### (ends loop or repeats it again based on comparison of less than)
UNTIL> ### (ends loop or repeats it again based on comparison of greater than)
UNTILR ###:### (ends loop or repeats it again based on comparison of range values. Low value and high value separated by :)
UNTILM ### (ends loop when comparison value ANDed with BitMask equal value)

By default the repeat loop uses its own counter (kind of like a FOR NEXT loop) unless a math value is designated.

To designate a math comparison use the commands just after a command whose reply you want to compare in the UNTIL Logic command:

RSAVE1
RSAVE2
RSAVE3
RSAVE4

These commands save the last reply value of the last command executed in a math comparison variable. The reason there are four save commands is if the reply value is like this:

1000:2000:3000:2000

which some commands can return (ie. EP) you can save any one of the values based on the index. In this case:

RSAVE1 would save a value of 1000
 RSAVE2 would save a value of 2000
 RSAVE3 would save a value of 3000
 RSAVE4 would save a value of 2000

Now when the UNTIL command is executed, it uses this saved math value instead of the loop counter for the comparison.

You can nest up to 3 levels deep of repeat loops.

2.5.3 Other BScript Commands

SHOWDC (displays last Controller reply value of previous command as decimal)

SHOW32 (displays last Controller reply value of previous command as 32 bit Binary)

Both BTMASK and SBIT commands effect result

SHOW16 (displays last Controller reply value of previous command as 16 bit Binary)

Both BTMASK and SBIT commands effect result

SHOW08 (displays last Controller reply value of previous command as 8 bit Binary)

Both BTMASK and SBIT commands effect result

SHOW04 (displays last Controller reply value of previous command as 4 bit Binary)

Both BTMASK and SBIT commands effect result

SHOW03 (displays last Controller reply value of previous command as 3 bit Binary)

Both BTMASK and SBIT commands effect result

SHOW02 (displays last Controller reply value of previous command as 2 bit Binary)

Both BTMASK and SBIT commands effect result

SHOW01 (displays last Controller reply value of previous command as 1 bit Binary)

Both BTMASK and SBIT commands effect result

SHOWUI (displays last Controller reply value of previous command as unsigned integer)

Both BTMASK and SBIT commands effect result

SHOWHX (displays last Controller reply value of previous command as 32 bit Hexadecimal)

Both BTMASK and SBIT commands effect result

SHOWTX (displays the last Controller reply as actual text returned)

Note: The SHOW commands displays values in the Terminal History window and the Global Memory Storage (if defined) window when run in the ASCII Script editor.

Alternate syntax for above **SHOW32** and **SHOW16** commands:

SHOW32| ' breaks up Binary in multiple 8 bit blocks separated by | character

SHOW32|# ' breaks up Binary in multiple 8 bit blocks separated by | character and # represents specific item if reply is

####:####:####:####

The NPBuilder application allows you to add the following macros for the expected reply for the above SHOW commands:

SHOW??, \$## //some description

The \$ indicates a Global Memory variable. ## is a value from 1 to 99

SHOW32, \$99

This stores the generated SHOW value into the Global Memory storage index number 99.

The use of the // macro is optional. The text after the macro can be used to describe or name the Global Memory Slot.

This allows the BScript to set values in memory for use by a generated application built using the NPBuilder application. For example the data may be passed to visual components when building a No-Code application. The Visual Designer will allow one to access the Global Memory Storage of any BScript. Also in the Low-Code features of the NPBuilder application the Global Memory Storage will be accessible.

BTMASK (display current Bit Mask as Hexadecimal value)

BTMASK=#### (set current Bit Mask to new value)

Syntax for Bit Mask value #### can be:

32000 ' decimal

0xFFFF ' hexadecimal (C syntax)

&HFFFF ' hexadecimal Visual Basic syntax (and Powerbasic)

0b001010 ' Binary (C syntax)

&B001010 ' binary Visual Basic syntax

A value of zero will turn off Bit Mask comparisons for following commands.

The BTMASK command Effects the following commands:

UNTILM in a REPEAT loop.

SHOW?? commands will have a numeric value modified by the Bitmask.

To reset the BTMASK command so it will not effect numeric comparions set the value to a negative 1.

BTMASK=-1

SBIT## shift bits to right (low bit direction) a specific number of bits

from 1 to 15 bits. The leftside bits will be zeroed with a shift.

must be a value from 00 to 15 (always two digits). A value of zero (00) will turn off Bit shifting in the following SHOW commands.

The shifting of bits will only effect the SHOW commands which display a numeric value which is binary or hexadecimal.

2.5.4 Use BScript Commands

Example of two REPEAT loops (one nested) using just the default counter value for comparison:

```
REPEAT,
PX=0,OK
PY=0,OK
PZ=0,OK
PU=0,OK
```

```

REPEAT,
PX=0, OK
PZ=0, OK
UNTIL=3,
UNTIL=10,

```

The outside loop will loop 10 times. The inside loop will loop 3 times , but because the outside loop iterates 10 times it will actually run 30 times.

A math comparison using a reply value would look like this:

```

JX-, OK
REPEAT,
EP,
RSAVE1,
UNTILR1000:2000,
STOP, OK

```

Assuming the EP values start at 0000:0000:0000:0000 the program above should loop until the first value returned from EP is between 1000 and 2000 and then the loop ends and the STOP command is executed.

The UNTILR command is inclusive, meaning the value is valid if not only between the two range values but also including the range values themselves.

The RSAVE# command will return a reply of what the actual value it stores in memory from the previous reply value. I need to change what displays in the window, since it current says SREPLY, not RSAVE#. I originally had the command be SREPLY, but changed it to RSAVE.

2.6 BScript Syntax

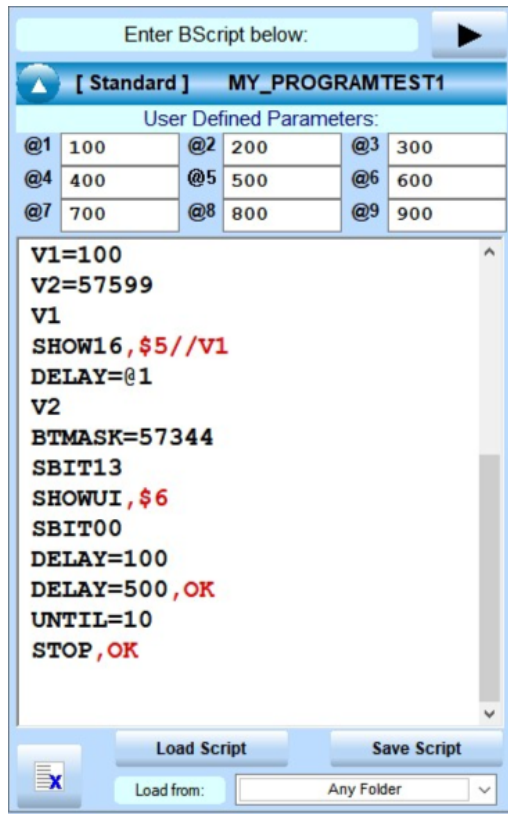
Write Programs for NPBuilder with BScript

BScript is an extended script command set beyond the normal Controller ASCII command language.

BScript is a dynamic extended scripting language which is translated by the BScript engine for Windows. The Controller ASCII command language can pass commands with no reply testing as normal or it can pass commands with an expected reply test (can set a Pass fail state). If requesting an expected reply, append the expected reply after the Controller ASCII command separated by a comma.

A subset of BScript is support in direct command mode in both the Nippon Terminal application and also the direct command mode in the NPBuilder application. In direct mode no expected replies are allowed, since each command is separated by commas so multiple commands can be sent to the controller from a single line of commands.

The full BScript mode requires the commands be entered in the Script Editor in the NPBuilder application:



2.7 BScript Code Samples

The Script Builder organizes scripts into categories:

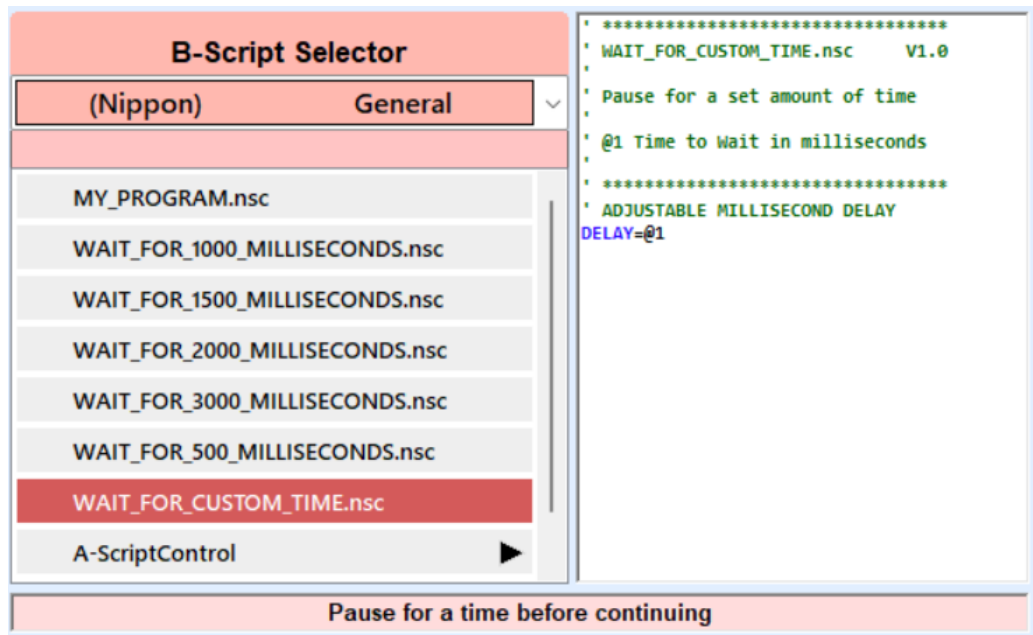
- **System Config**
- **Motion Control**
- **I/O Control**
- **General**

There are two Nippon-specific folders appear:

- **Nippon MP Builder** - editable scripts
- **Nippon** –Read Only library

B-Script Selector	
(Nippon)	General
(NPBuilder)	SystemConfig
(NPBuilder)	MotionControl
(NPBuilder)	IOControl
(NPBuilder)	General
(Nippon)	SystemConfig
(Nippon)	MotionControl
(Nippon)	IOControl
(Nippon)	General

The scripts can be found and read in the code box next to the selector



2.7.1 General

- **WAIT_FOR_1000_MILLISECONDS.nsc**
Pause for 1 seconds before continuing
- **WAIT_FOR_1500_MILLISECONDS.nsc**
Pause for 1.5 seconds before continuing
- **WAIT_FOR_2000_MILLISECONDS.nsc**
Pause for 2 seconds before continuing
- **WAIT_FOR_3000_MILLISECONDS.nsc**
Pause for 3 seconds before continuing
- **WAIT_FOR_500_MILLISECONDS.nsc**
Pause for 0.5 seconds before continuing
- **WAIT_FOR_CUSTOM_TIME.nsc**
Pause for a time before continuing

A-ScriptControl

- **Pause_A-Script.nsc**
Pause a running A-Script program
- **Resume_A-Script.nsc**
Resume a paused A-Script program
- **Run_A-Script_Subroutine.nsc**
Run an A-Script Subroutine once
- **Start_A-Script.nsc**
Start an A-Script program loaded on the Commander
- **Stop_A-Script.nsc**
Stop an A-Script program

ApplicationScripts

- **RASTER_INC_BIDIRECTIONAL_SCAN.nsc**

Performs an incremental raster scan with a scan in positive and negative direction

- **RASTER_INC_UNIDIRECTIONAL_SCAN.nsc**
Performs an incremental raster scan with a scan in positive direction negative direction is only for return
- **STEP_REPEAT_BIDIRECTIONAL_SCAN.nsc**
The scan axis moves a set distance, stops, system performs an action, and then repeats the process a set number of times in positive and negative direction
- **STEP_REPEAT_UNIDIRECTIONAL_SCAN.nsc**
The scan axis moves a set distance, stops, system performs an action, and then repeats the process a set number of times. After completing a full line of discrete stops along the scan axis, the step axis moves a set distance, then motion is repeated in the negative direction.

2.7.2 IOControl

- **GPIO_CHANGE_ALL_STATE.nsc**
Set state of all GPIO
- **GPIO_CHANGE_SINGLE_STATE.nsc**
Change a single GPIO state
- **GPIO_TOGGLE_STATE_OFF_ON.nsc**
Toggles an IO state off then on
- **GPIO_TOGGLE_STATE_ON_OFF.nsc**
Toggles an IO state on then off

2.7.3 MotionControl

SET_EXTERNAL_START.nsc

AxisEnableOperations

- **1AXIS_ENABLE_DISABLE_w_ERROR_CLEAR.nsc**
Axis enable with clear error status
- **2AXES_ENABLE_DISABLE_w_ERROR_CLEAR.nsc**
2 Axes enable with clear error status
- **3AXES_ENABLE_DISABLE_w_ERROR_CLEAR.nsc**
3 Axes enable with clear error status
- **4AXES_ENABLE_DISABLE_w_ERROR_CLEAR.nsc**
4 Axis enable with clear error status
- **ALL_AXES_ENABLE_DISABLE_w_ERROR_CLEAR.nsc**
All Axis enable with clear error status

AxisHomeingOperations

HOME_Z_INDEX_w_Return.nsc

Homes at Homing Speed to Z-index then returns at Return Speed

AxisMoves

- **1AxisMotion**

- **1AXIS_JOG_FOR_TIME.nsc**
@1 axis will jog for a set amount of time
- **1AXIS_JOG_WHILE_INPUT.nsc**
@1 axis will jog while input is enabled
- **1AXIS_MOVE.nsc**
Moves 1 axis a distance
- **1AXIS_MOVE_AWAIT_MOTION_TO_END.nsc**
Moves 1 axis a distance then awaits the motion of the selected axis to end
- **1AXIS_MOVE_WAIT_FOR_TIME.nsc**
Moves 1 axis a distance then waits for time

- **2AxesMotion**

- **2AXES_CIRCLE_MOVE.nsc**
Uses 2 axes for circle
- **2AXES_MOVE.nsc**
Moves 2 axes a distance
- **2AXES_MOVE_AWAIT_MOTION_TO_END.nsc**
Moves 2 axes a distance then awaits the motion of the selected axes to end
- **2AXES_MOVE_WAIT_FOR_TIME.nsc**
Moves 2 axes a distance then waits for time

- **3AxesMotion**

- **3AXES_MOVE.nsc**
Moves 3 axes a distance
- **3AXES_MOVE_AWAIT_MOTION_TO_END.nsc**
Moves 3 axes a distance then awaits the motion of the selected axes to end
- **3AXES_MOVE_WAIT_FOR_TIME.nsc**
Moves 3 axes a distance then waits for time
- **XYZ_ARC_WITH_HELIX.nsc**
Uses axis XY for arc and Z for tangent
- **XYZ_CIRCLE_WITH_HELIX.nsc**
Uses axis XY for circle and Z for tangent

4AxesMotion

- **4AXES_MOVE.nsc**
Moves 4 axes a distance
- **4AXES_MOVE_AWAIT_MOTION_TO_END.nsc**
Moves 4 axes a distance then awaits the motion of the selected axes to end
- **4AXES_MOVE_WAIT_FOR_TIME.nsc**
Moves 4 axes a distance then waits for time

AxisWaitOperations

- **Home(ORG)**

- **1AXIS_AWAIT_HOME(ORG).nsc**
Awaits the motor ORG (home) status on for the selected axis
- **1AXIS_AWAIT_HOME(ORG)_SET_PULSE_ENCODER_COUNTERS.nsc**
Awaits the motor home(ORG) input for the selected axis then sets Pulse and Encoder counters
- **2AXES_AWAIT_HOME(ORG).nsc**
Awaits the motor ORG (home) status on for the selected axis
- **2AXES_AWAIT_HOME(ORG)_SET_PULSE_ENCODER_COUNTERS.nsc**
Awaits the motor home(ORG) input for the selected axes then sets Pulse and Encoder counters
- **3AXES_AWAIT_HOME(ORG).nsc**
Awaits the motor ORG (home) status on for the selected axis
- **3AXES_AWAIT_HOME(ORG)_SET_PULSE_ENCODER_COUNTERS.nsc**
Awaits the motor home(ORG) input for the selected axes then sets Pulse and Encoder counters
- **4AXES_AWAIT_HOME(ORG).nsc**
Awaits the motor ORG (home) status on for the selected axis
- **4AXES_AWAIT_HOME(ORG)_SET_PULSE_ENCODER_COUNTERS.nsc**
Awaits the motor home(ORG) input for the selected axes then sets Pulse and Encoder counters

- **InPosition**

- **1AXIS_AWAIT_IN_POSITION_OFF.nsc**
Awaits the selected axis in-position status to turn off
- **1AXIS_AWAIT_IN_POSITION_ON.nsc**
Awaits the selected axis in-position status to turn on
- **2AXIS_AWAIT_IN_POSITION_OFF.nsc**
Awaits the selected axis in-position status to turn off
- **2AXIS_AWAIT_IN_POSITION_ON.nsc**
Awaits the selected axis in-position status to turn on
- **3AXIS_AWAIT_IN_POSITION_OFF.nsc**
Awaits the selected axis in-position status to turn off
- **3AXIS_AWAIT_IN_POSITION_ON.nsc**
Awaits the selected axis in-position status to turn on
- **4AXIS_AWAIT_IN_POSITION_OFF.nsc**
Awaits the selected axis in-position status to turn off
- **4AXIS_AWAIT_IN_POSITION_ON.nsc**
Awaits the selected axis in-position status to turn on

- **MotionToEnd**

- **1AXIS_AWAIT_MOTION_TO_END.nsc**
Awaits the motion of the selected axis to end
- **2AXES_AWAIT_MOTION_TO_END.nsc**
Awaits the motion of the selected axes to end
- **3AXES_AWAIT_MOTION_TO_END.nsc**
Awaits the motion of the selected axes to end
- **4AXES_AWAIT_MOTION_TO_END.nsc**
Awaits the motion of the selected axes to end
- **Time**
 - **WAIT_FOR_1000_MILLISECONDS.nsc**
Pause for 1 seconds before continuing
 - **WAIT_FOR_1500_MILLISECONDS.nsc**
Pause for 1.5 seconds before continuing
 - **WAIT_FOR_2000_MILLISECONDS.nsc**
Pause for 2 seconds before continuing
 - **WAIT_FOR_3000_MILLISECONDS.nsc**
Pause for 3 seconds before continuing
 - **WAIT_FOR_500_MILLISECONDS.nsc**
Pause for 0.5 seconds before continuing
 - **WAIT_FOR_CUSTOM_TIME.nsc**
Pause for a time before continuing

MotionProfileSetup

- **GLOBAL_HSPD_CHANGE.nsc**
Changes the global HSPD for motion profile
- **GLOBAL_LSPD_CHANGE.nsc**
Changes the global LSPD for motion profile
- **GLOBAL_MOTION_PROFILE.nsc**
Sets motion profile for all axis on a single Commander
- **LOCAL_HSPD_CHANGE.nsc**
Changes the HSPD for a single axis motion profile
- **LOCAL_LSPD_CHANGE.nsc**
Changes the local LSPD for motion profile
- **LOCAL_MOTION_PROFILE.nsc**
Sets motion profile for single axis

SET_EXTERNAL_START.nsc

Sets the External start function

2.7.4 SystemConfig

COMMANDER_SETUP.nsc**SOFT_LIMITS_SINGLE_AXIS.nsc****SetPositionEncoderCounters**

- **1AXIS_SET_ENCODER_COUNTER.nsc**
Sets the Encoder Counters for 1 Axis
- **1AXIS_SET_POSITION_COUNTER.nsc**
Sets the Position Counters for 1 Axis
- **1AXIS_SET_POSITION_ENCODER_COUNTER.nsc**
Sets the Position and Encoder Counters for 1 Axis
- **2AXES_SET_ENCODER_COUNTER.nsc**
Sets the Encoder Counters for 2 Axes
- **2AXES_SET_POSITION_COUNTER.nsc**
Sets the Position Counters for 2 Axes
- **2AXES_SET_POSITION_ENCODER_COUNTER.nsc**
Sets the Position and Encoder Counters for 2 Axes
- **3AXES_SET_ENCODER_COUNTER.nsc**
Sets the Encoder Counters for 3 Axes
- **3AXES_SET_POSITION_COUNTER.nsc**
Sets the Position Counters for 3 Axes
- **3AXES_SET_POSITION_ENCODER_COUNTER.nsc**
Sets the Position and Encoder Counters for 3 Axes
- **4AXES_SET_ENCODER_COUNTER.nsc**
Sets the Encoder Counters for 4 Axes
- **4AXES_SET_POSITION_COUNTER.nsc**
Sets the Position Counters for 4 Axes
- **4AXES_SET_POSITION_ENCODER_COUNTER.nsc**
Sets the Position and Encoder Counters for 4 Axes

UserUnitSetup

- **LINEAR_UNITS_SETUP.nsc**
Sets the linear unit conversion for B-Script
- **ROTARY_UNITS_SETUP_DEGREES.nsc**
Sets the rotary, degrees per step unit conversion for B-Script
- **ROTARY_UNITS_SETUP_ROTATIONS.nsc**
Sets the rotary unit conversion, steps per rotation for B-Script

COMMANDER_START_UP.nsc

Sets the Commander into a known state

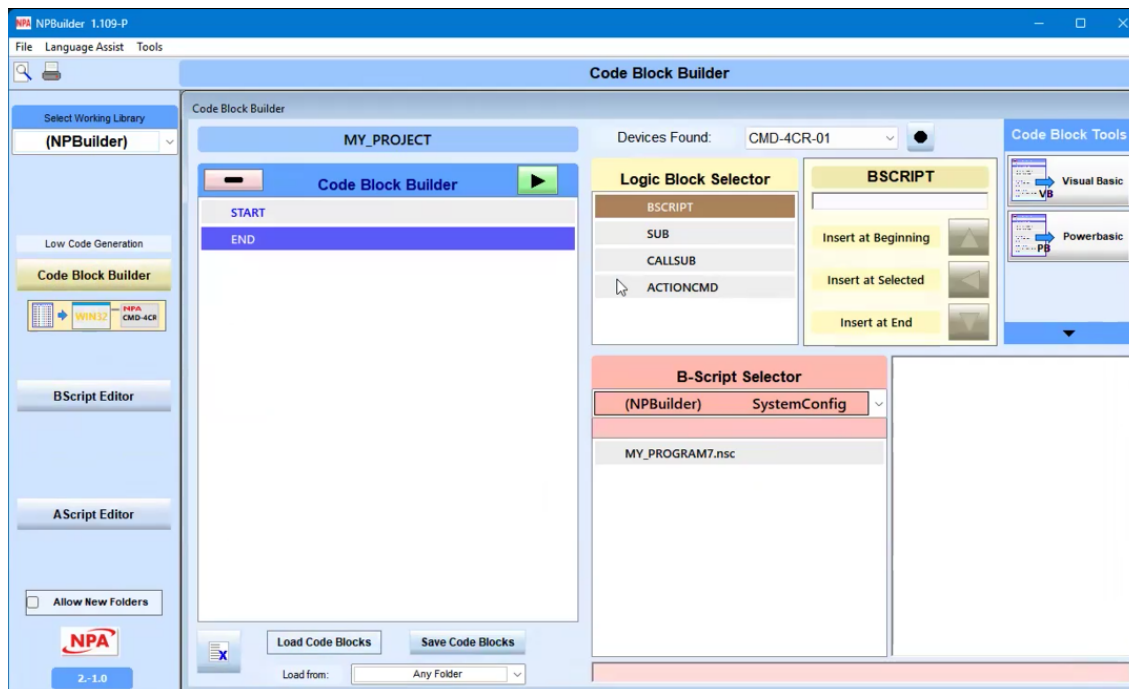
SOFT_LIMITS_SINGLE_AXIS.nsc

AxisLetter (X,Y,Z,U,A,B,C,D)

3.0 Code Block Builder

The Code Block Builder is a project level editor for building a complex project by combining smaller Bscripts . There is a limited user interface engine, but this engine is for help in making prototyping easier and not for a final UI for a product. A project is created using the provided Logic Blocks such as a Bscript (use a Bscript with unique parameters passed to it), SUB (define a new subroutine), CALLSUB (call an existing subroutine and some logic such as comparisons can be done) and ACTIONCMD (non motion control tasks such as UI definition, device selection, keyboard input).

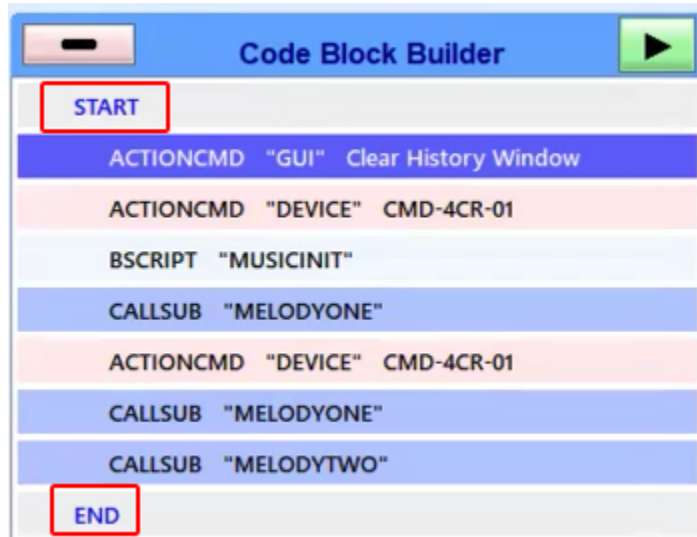
Projects can be run in real time by clicking the Play / RUN button. This is for development and prototyping or even running a Demo machine. For a final product one would convert the project to usable source code for a target programming language. This is what the Plugins are for. NPBuilder will come with a few code generation plugins for some languages, but the Plugins are built upon an open API which will allow you to create your own custom code generation tools for any programming of your choice.



When you create BScripts, you can [define parameters](#). Code Block builder is used to combine BScripts.

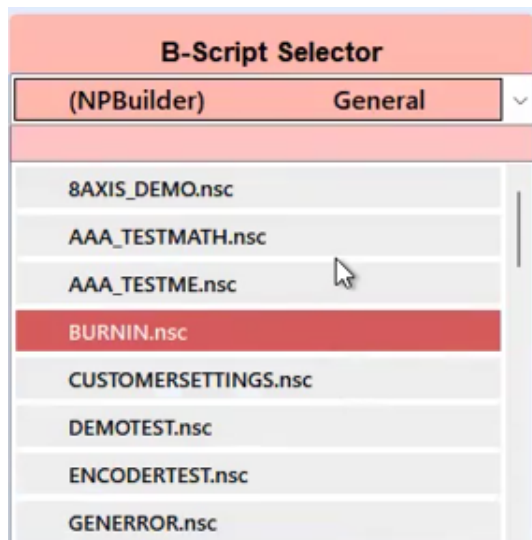
The Code Block Builder screen contains these sections:

Code Block Builder



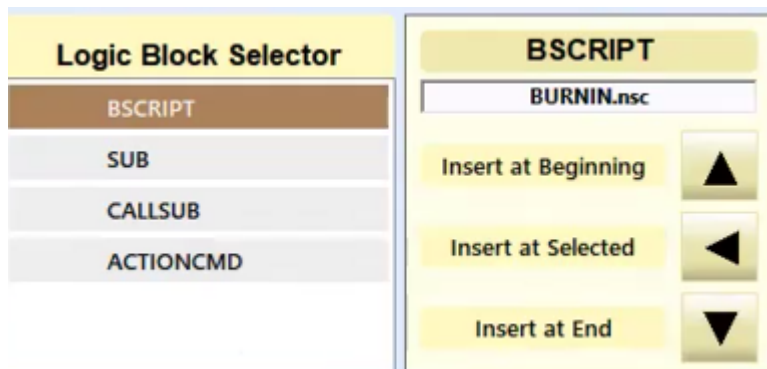
Start and **End** define where the projects begins and finishes. The script commands are within the the block.

B-Script Selector



Use this section to select the BScripts that will be added to the Code Block.

Logic Block Selector



When you have selected a Bscript, use the **Logic Block Selector** to add the script to the **Code Block Builder**. You can define the block, and choose where in the builder to insert it.

BScript Code

```

'BURN-IN TESTING
EO=0,OK
HSPD=40000,OK
' TESTING @ 40,000PPS 10 TIMES
ACC=500,OK
DEC=500,OK
EO=7,OK
REPEAT,
  ABS,OK
  DELAY=100,
  X0,OK
  BTMASK=7,
  REPEAT,
    MSTX,
    RSAVE1,
  UNTILM0,
  DELAY=100,
  Y0,OK
  BTMASK=7,
  REPEAT,
    MSTY,
    RSAVE1,
  UNTILM0,
  DELAY=100,
  Z0,OK

```

The Bscript code is displayed in the section next to the selector once a selection has been made.

3.1 Build a Code Block

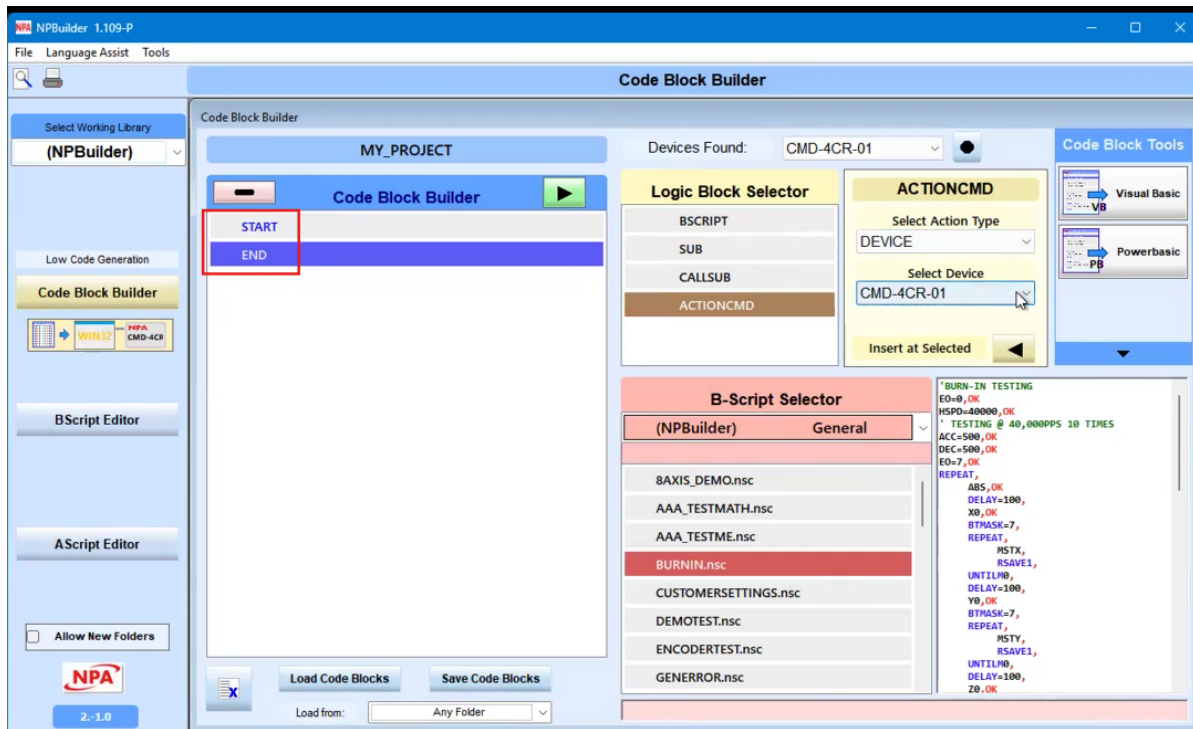
Use the Code Block Builder to generate code for selected programming languages and to prototype complex projects.

This video provides a walk-through of the steps documented below.

On this page

- [Add an Action Command](#) (00:13)
- [Add a GUI Command](#) (00:39)
- [Add a Wait Action](#) (01:50)
- [Create a Subroutine](#) (02:22)
- [Create an Endless Loop](#) (03:18)
- [Call Subroutines Conditionally](#) (04:22)
- [Script Categories](#)

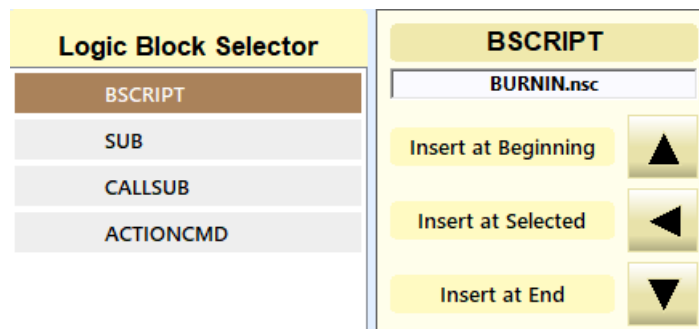
Code Block Builder opens with only **Start** and **End** in the Block Builder.



In the Logic Block Selector, select the command type to add to the builder.

- BSCRIPT
- SUB (subroutine)
- CALLSUB (call a subroutine)
- ACTIONCMD (action command)

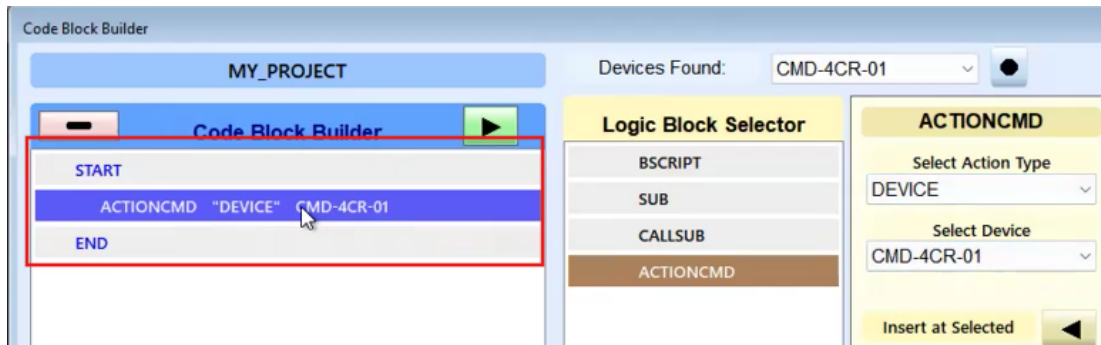
Note: The Section next to the block selector changes depending on the selection.



Following is an example of building a simple UI script.

Add an Action Command

1. Click **ACTIONCMD** in the Logic Block Selector.
2. Choose an **Action Type** (*Device*).
3. Select the device or object to interact with.
4. Click the **Insert at Selected** arrow button at the selected point in the script.
This creates an executable line.

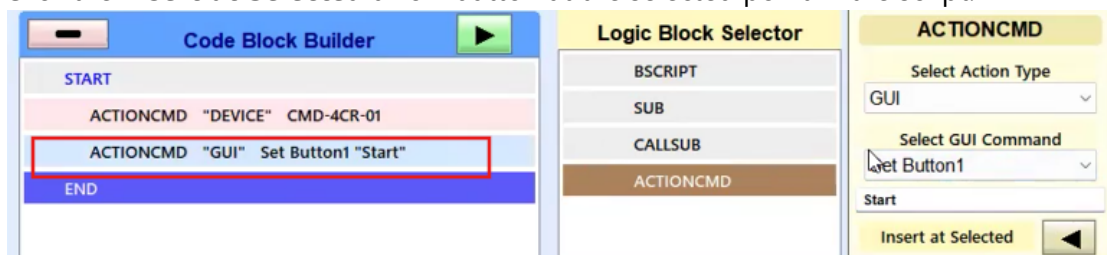


[Return to Top](#)

Add a GUI Command

Add GUI actions such as:

- o **Clear History Window** – clears the execution log
 - o **Clear Window** – clears the main display panel where text and UI elements appear
 - o **Display text, buttons, other UI elements**
1. Click **ACTIONCMD** in the Logic Block Selector.
 2. Select **GUI** as the action type.
 3. Select a GUI Command (*SetButton1*).
 4. Click the **Insert at Selected** arrow button at the selected point in the script.



Up to five buttons are supported. Each button has a numeric ID.

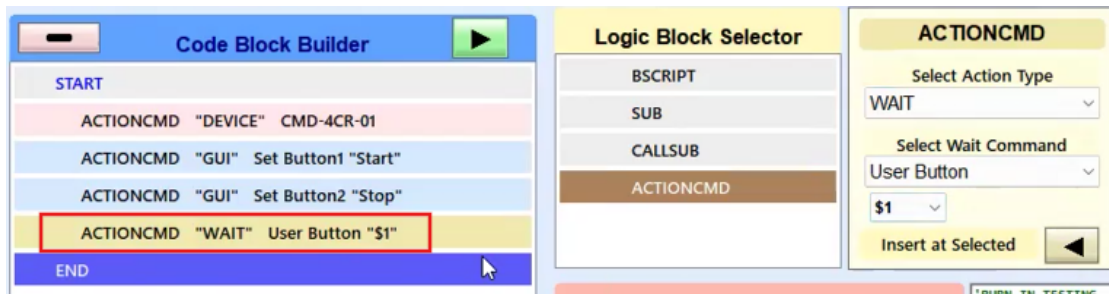
[Return to Top](#)

Add a Wait Action

Use a **Wait Action** to pause execution until a button is pressed.

Note: ACTIONCMD "GUI" Set Button2 "Stop" has been added.

1. Click **ACTIONCMD** in the Logic Block Selector.
2. Select **Wait** as the action type.
3. Select a Wait Command (*User Button*).
In this example, the User Button is going to test for certain value, so if a user button is pressed, the number of the button is put in this value (\$1), and memory location.
4. Click the **Insert at Selected** arrow button at the selected point in the script.



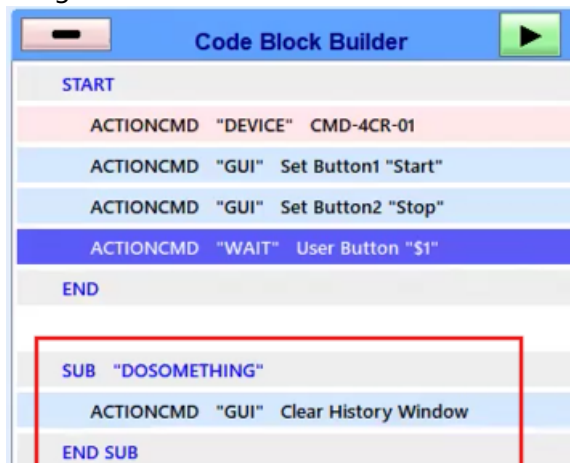
In this example, If a User Button is pressed, store the button number in global variable #1. This allows the script to detect which button the user selected.

[Return to Top](#)

Create a Subroutine

Subroutines allow you to organize reusable logic.

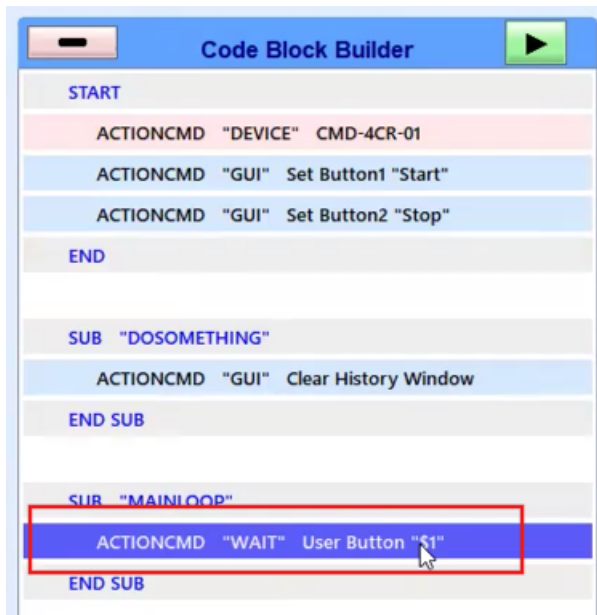
1. Click **SUB** in the Logic Block Selector.
2. Type a Subroutine Name (*DOSOMETHING*).
3. Click the **Insert at End** arrow button.
4. Add commands to the subroutine, such as clearing the history window or showing text using **ACTIONCMD**.



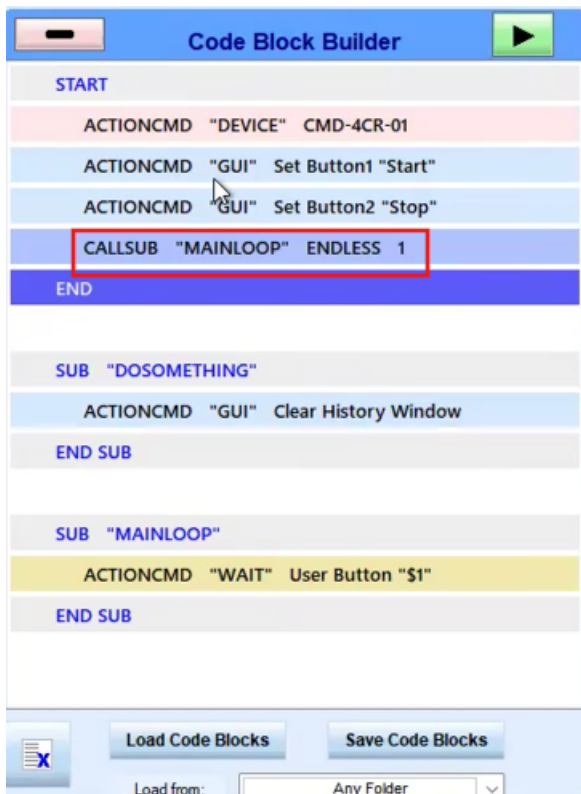
[Return to Top](#)

Create an Endless Loop

1. Click **SUB** in the Logic Block Selector.
2. Type a Subroutine Name (*MAINLOOP*).
3. Click the **Insert at End** arrow button.
4. Add commands to repeat continuously using **ACTIONCMD**, or as in this example, drag and drop the **ACTIONCMD "Wait" User Button "\$1"** line to the **MAINLOOP** subroutine.



5. Click **CALLSUB** in the Logic Block Selector.
6. In the CALLSUB popup:
 1. Select MAINLOOP as the **Subroutine to CALL**
 2. Select **Endless** for how many time to call.
7. Click the **Call Selected SUB** button.



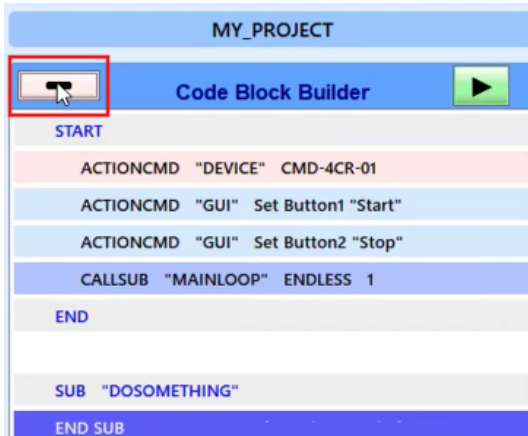
This creates a continuous loop that checks for button presses.

[Return to Top](#)

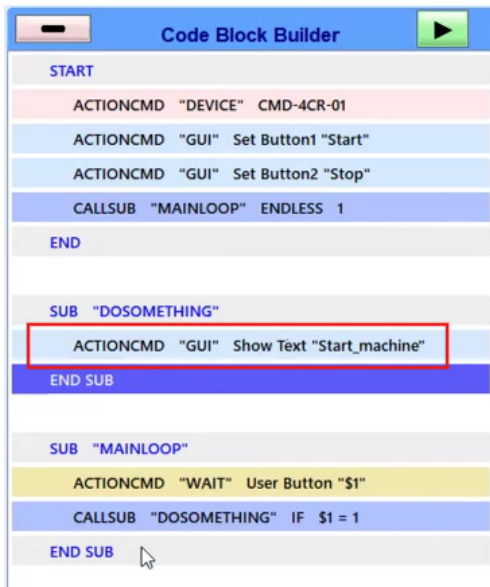
Call Subroutines Conditionally

Evaluate global variables in the MAINLOOP subroutine:

1. Click **CALLSUB** in the Logic Block Selector.
2. In the CALLSUB popup:
 1. Select DOSOMETHING as the **Subroutine to CALL**
 2. Select **IF** for how many time to call.
 1. For **Global Variable**, select **\$1**.
 2. For **IF Comparison**, select **= (Equal)**.
 3. For **Value**, type **1**.
 3. Click the **Call Selected SUB** button.
3. In the Code Block Builder Section, select the ACTIONCMD line in the DOSOMETHING subroutine.



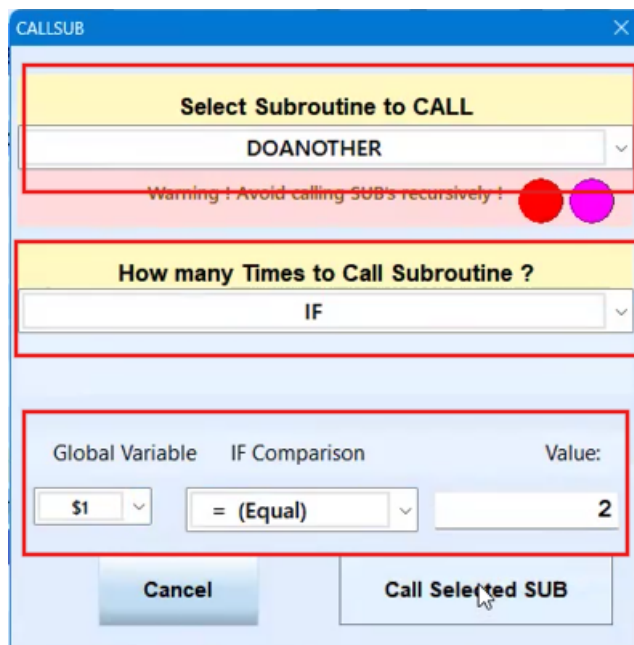
4. Add an **ACTIONCMD**, type **GUI**, with GUI Command **Show Text** to the DOSOMETHING subroutine.
5. Type **Start Machine**.
6. Click the **Insert at Selected** arrow button.



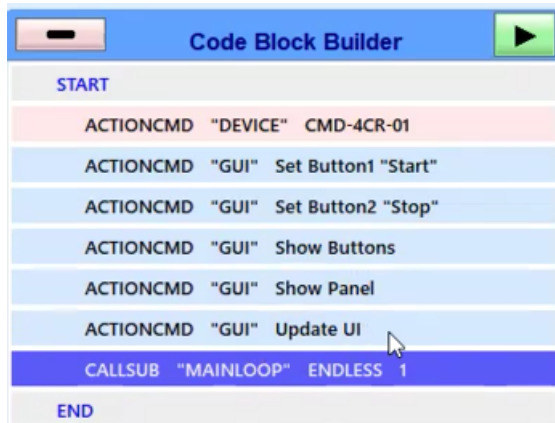
7. Create another subroutine (*DOANOTHER*).
8. Add an **ACTIONCMD**, type **GUI**, with GUI Command **Show Text** to the DOANOTHER subroutine with text **Stop machine**.



9. Add a **CALLSUB** command.
 1. For **Global Variable**, select **\$1**.
 2. For **IF Comparison**, select **= (Equal)**.
 3. For **Value**, type **2**.
 4. Click the **Call Selected SUB** button.

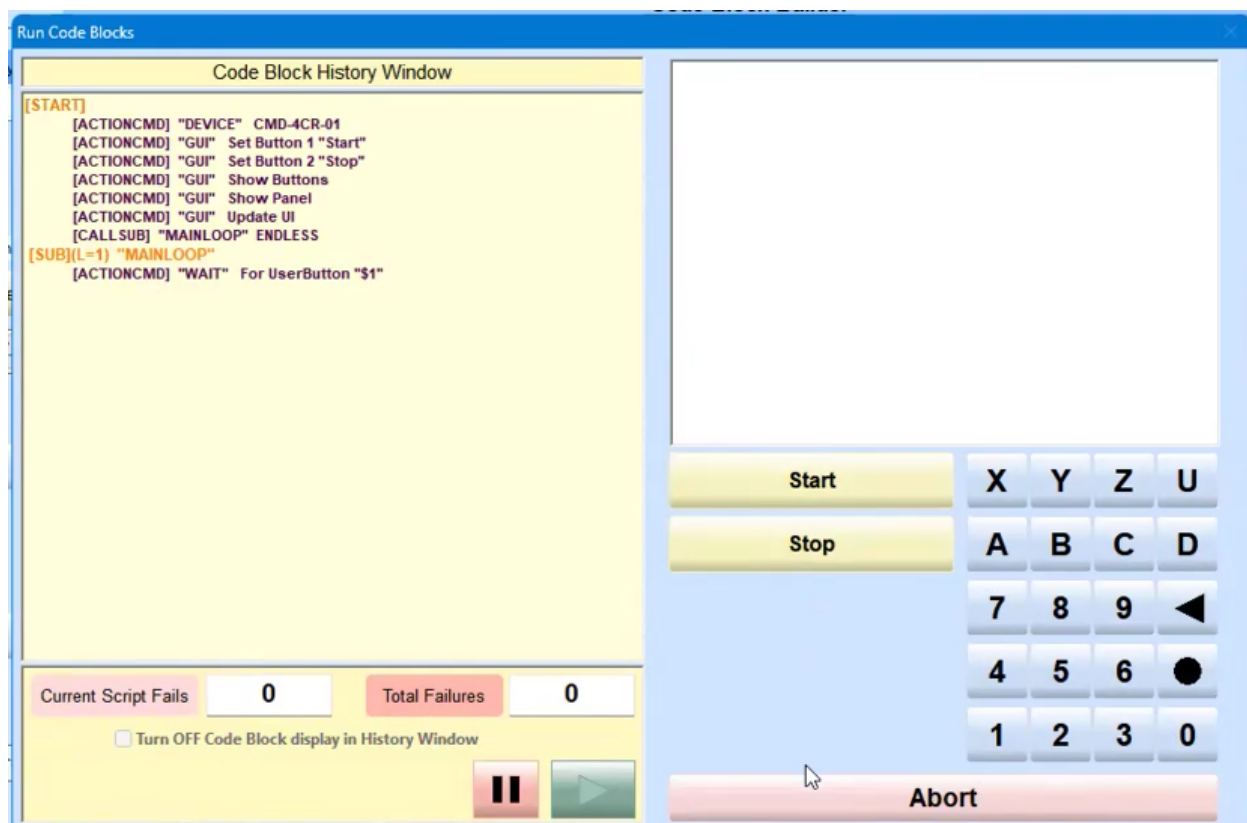


10. Add an **ACTIONCMD**: type **GUI**, GUI command **Show Panel**.
11. Add an **ACTIONCMD**: type **GUI**, GUI command **Update UI**.



When you press the Play button, the script opens in the Run Code Blocks screen. On the right side of the screen:

- Pressing **Start** prints *Start Machine*.
- Pressing **Stop** prints *Stop Machine*.
- The main loop continues checking for button presses.
- Subroutines execute based on the global variable values.



Expand this structure to create menus, run different script sections, or build more complex user interfaces.

[Return to Top](#)

Script Categories

The Script Builder organizes scripts into categories:

- **System Config**

- **Motion Control**
- **I/O Control**
- **General**

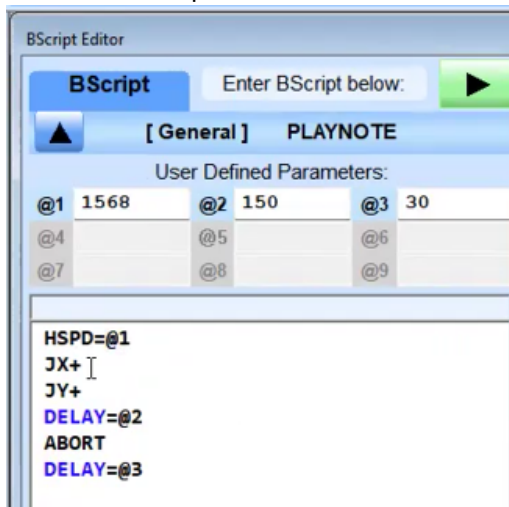
There are two [Nippon-specific folders](#):

- **Nippon NP Builder** - editable scripts
- **Nippon** –Read Only library

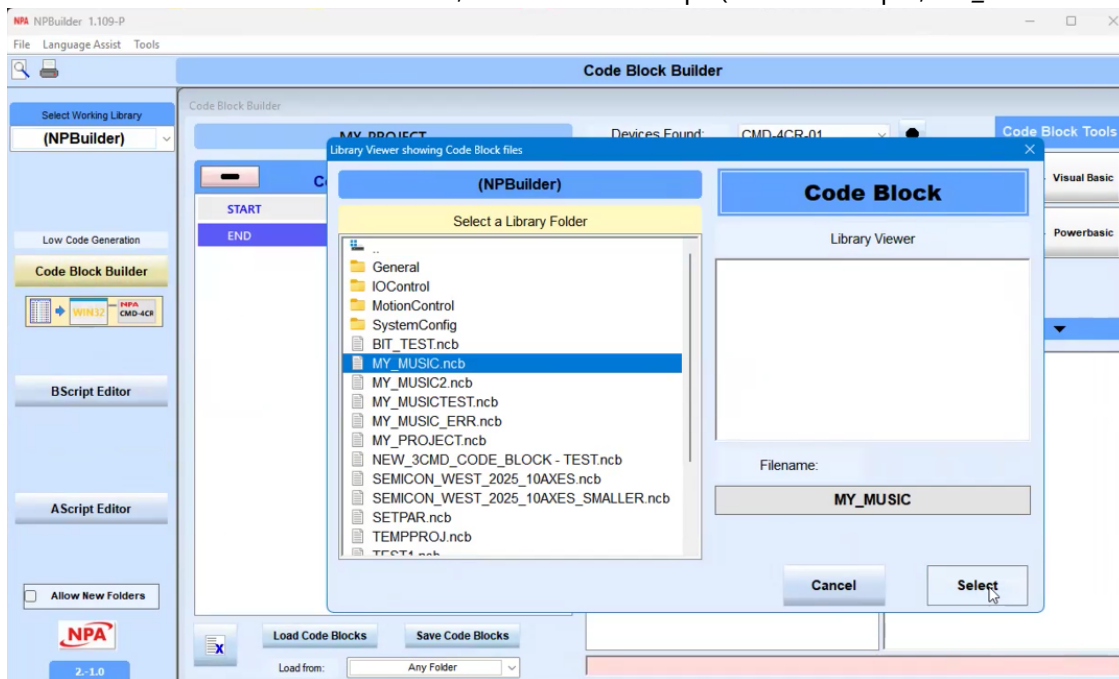
PlayNote Sample

In the following example, we illustrate Code Block Builder using a script called *PLAYNOTE* (a small, simple script which turns on or jogs a stepper motor to play a sound).

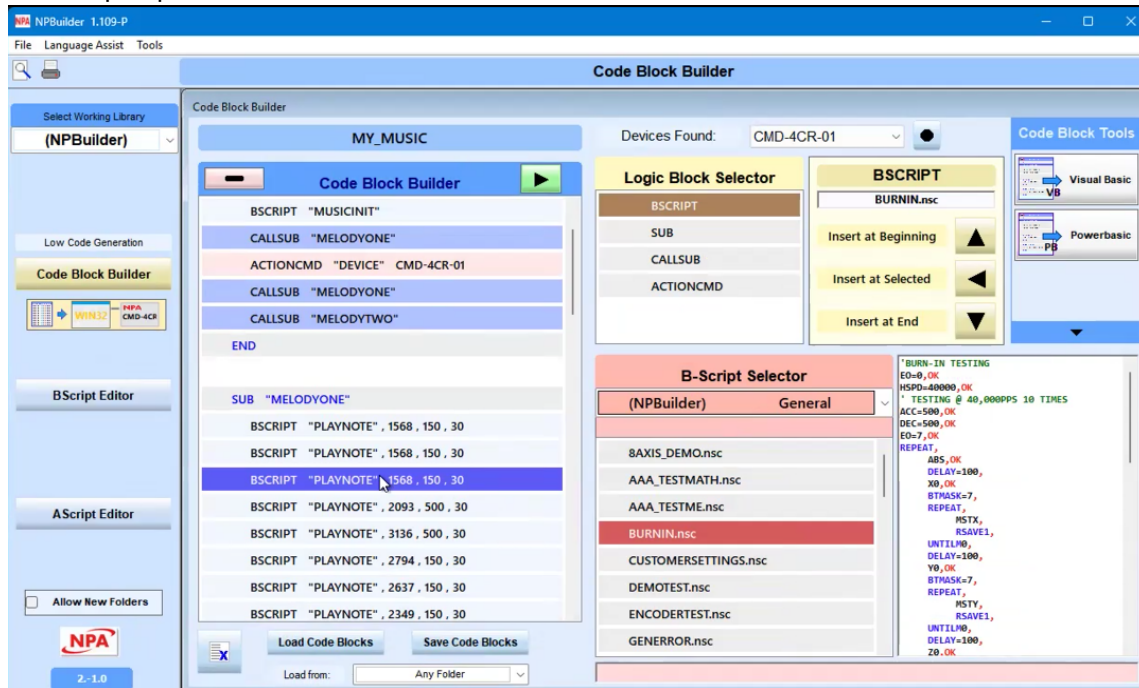
PLAYNOTE script:



1. Open the BScript Editor.
2. Click the **Code Block Builder** button.
3. Click the **Load Code Blocks** button, then select a script (in this example, *MY_MUSIC.ncb*).



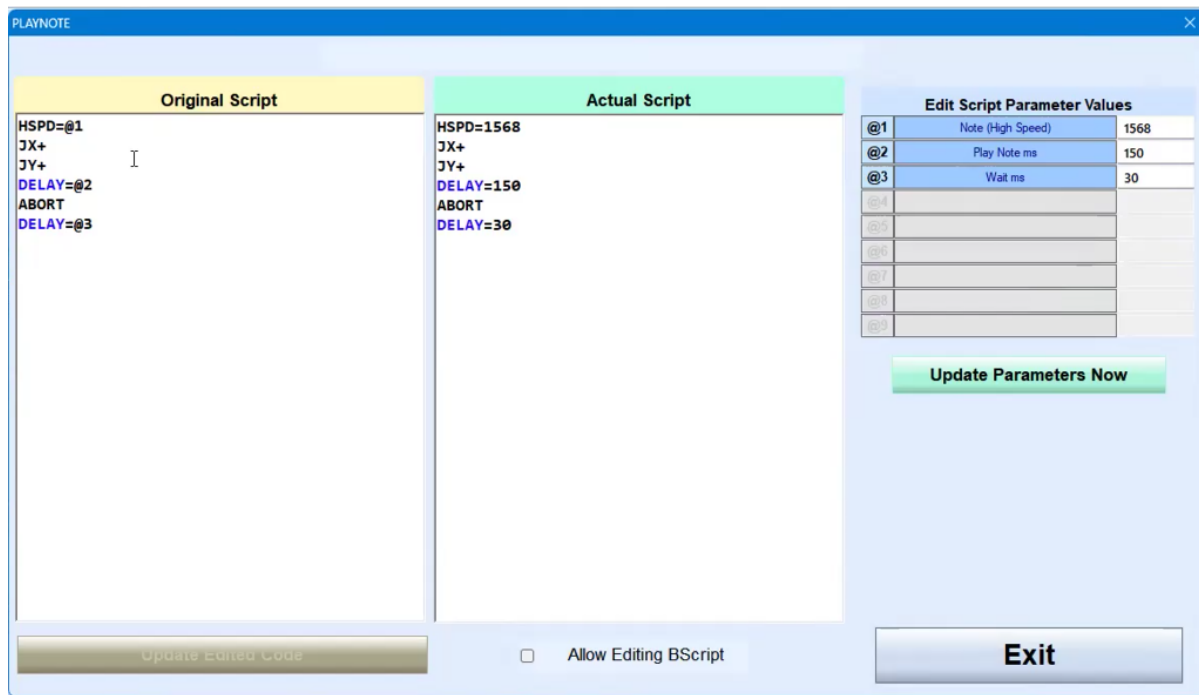
The script opens in the Code Block Builder



The subroutine Melodyne reuses the same script for multiple lines but uses different parameters.

SUB "MELODYONE"	
BSCRIPT	"PLAYNOTE", 1568, 150, 30
BSCRIPT	"PLAYNOTE", 1568, 150, 30
BSCRIPT	"PLAYNOTE", 1568, 150, 30
BSCRIPT	"PLAYNOTE", 2093, 500, 30
BSCRIPT	"PLAYNOTE", 3136, 500, 30
BSCRIPT	"PLAYNOTE", 2794, 150, 30
BSCRIPT	"PLAYNOTE", 2637, 150, 30
BSCRIPT	"PLAYNOTE", 2349, 150, 30

Double-click a line to view the Original/Actual script. This screen also displays the scrip parameter values. These values can be edited here. Click **Update Parameters Now** to save changes.



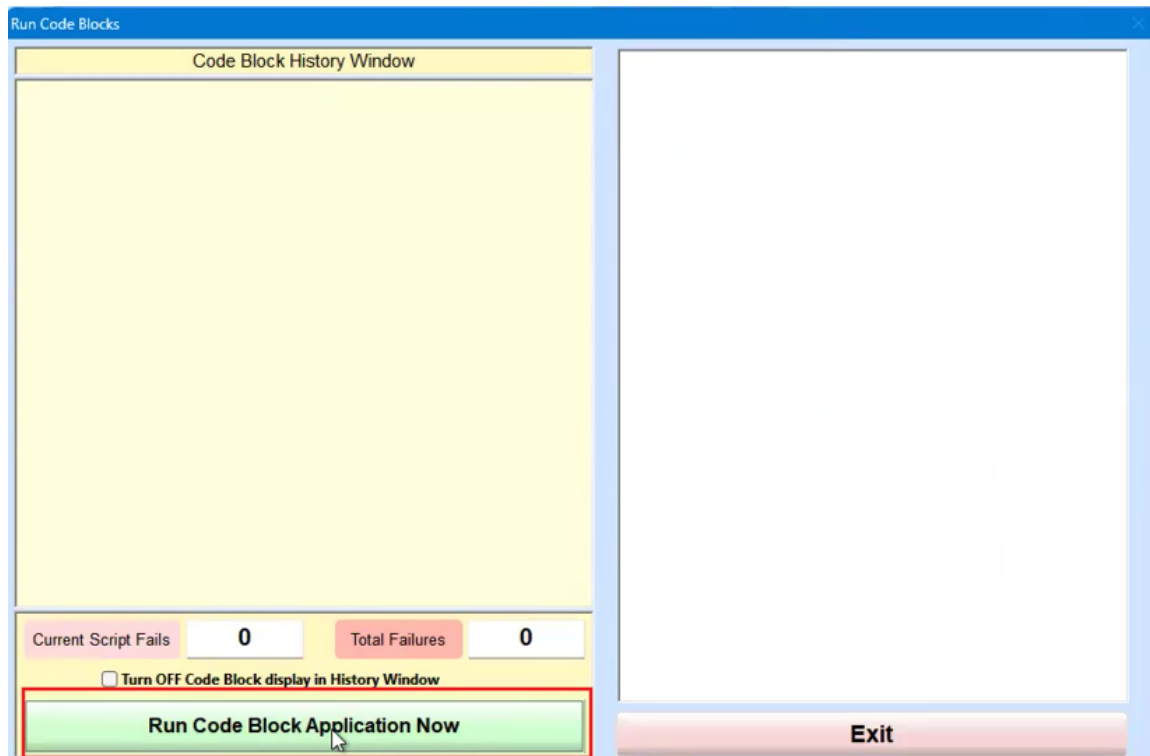
Click the **Exit and Save** button to close this screen.

3.2 Run a Code Block

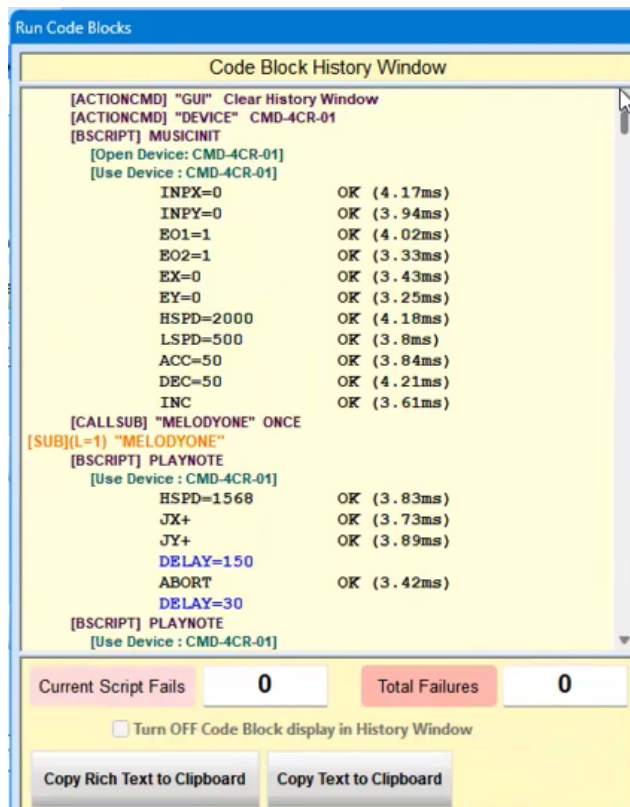
When your Code Block is complete, click the Play button to run the code.



Click **Run Code Block Application Now** in the **Run Code Blocks** screen.



The commands being executed will be displayed in the Code Block History Window. It also tracks any failures as the script runs.



Code Block Builder includes a temporary interface option, which can be used to test what is being sent to the Commander.

