GPC is the scripting language used by the CronusMAX PLUS. Its commands have been designed to be self explanatory so anyone from a novice to expert programmer is able to pick it up quickly. The basic syntax of GPC has been borrowed from C language and the basic structure and keywords will be familiar to those whom have programmed in C before. There are of course a number of unique GPC functions that you will not find in C, however, these have also been designed in a way were they can be read and understood with ease.

GPC however is simply a human readable version of the code used by the CronusMAX PLUS. The CronusMAX actually uses a bytecode system. Whenever you program, build or compile a GPC script, the Compiler built in to Cronus Pro coverts the script in to bytecode prior to being sent to the device.

Bytecode is an instruction set for the virtual machine (VM or software interpreter) which the CronusMAX PLUS runs. Inputs for controllers are sent to the VM which then modifies them according the instruction set in the bytecode. This modified output is then sent to the console. The Virtual Machine is a hypothetical computer and its functions and architecture has been optimized specifically for the task of modifying and outputting controller commands to a console.

The CronusMAX VM is a stack machine and has been optimized so it is able to send data to a console without delay. Providing the CPU load is kept below 80%, the CronusMAX PLUS's CPU is able to carry out all the tasks assigned to it without delaying the output to the console. Part of the optimization process means error checks by the VM are kept to a minimum and to only what is necessary. This means it is important the programmer has a good understanding of the limitations and fundamentals of GPC when developing scripts.

### CronusMAX PLUS Specifications

- **Real CPU:** Atmel Microcontroller @ 16MHz
- **Real Memory:** 8184 bytes
- **Real Flash:** 130944 bytes
- **VM Type:** Stack Machine
- **VM Max Bytecode:** 4096 bytes
- **VM Stack Memory:** 128 words (256 bytes)
The contents of GPC Guide

Basic Syntax
A Simple Tutorial
Basic GPC Structure
Definitions
Data Section
Remapping
Variables
Init Section
Main Section
Combo Section
User Created Functions
I/O Functions
Internal Functions & Commands
Operators
List of GPC Keywords
Basic Syntax

As in C, GPC requires instructions to be terminated with a semicolon at the end of each statement. However, the closing tag of a block code automatically implies a semicolon, so a semicolon is not needed when terminating the last line of a GPC block.

```c
#define CronusMAX = 120;
int PLUS = 100;
main {
    if (get_val(XB1_A)) {
        combo_run(jump);
    }
}
combo jump {
    set_val(XB1_A, 100);
    wait(CronusMAX);
    set_val(XB1_A, 0);
    wait(PLUS);
}
```

Although the semicolon is not required in the final line of a block, it is considered good practice to use one so it is not missed should you expand on the code at a later date.
Nesting Code

Nesting code, or creating a logic block, binds code together. A Block starts with a { and end with a }. What this does is nest the code with the { and } meaning that the code is only executed when the statement before it is active.

In this example;

```c
main
{
    //Main Start
    if (get_val(PS4_R2))
    { //Block 1 Start
        if (get_val(PS4_L2))
        { //Block 2 Start
            combo_run(RAPID_FIRE_ADS);
        } //Block 2 End
        else
        { //Block 3 Start
            combo_run(RAPID_FIRE);
        } //Block 3 End
    } //Block 1 End
} //Main End
```

Blocks 2 & 3 are ignored unless Block 1 is active. So if the R2 button is not pressed, nothing happens. If R2 is pressed, then the CronusMAX looks at Block 2. If L2 is pressed, it will run the combo RAPID_FIRE_ADS and ignore Block 3. However, if L2 is not pressed, it will ignore Block 2 and instead execute the code in Block 3 and then run combo RAPID_FIRE.

Nesting is implied if you only have one line of code after a statement. As in this example;

```c
main {
    if (get_val(XB1_RT) > 95)
        combo_run(RAPID_FIRE);
}
```

When compiled, the line combo_run(RAPID_FIRE); will automatically be nested within the if statement. If you wish for more than one line of code to only be executed when the statement before them is active, then you must use { and }.
Commenting Code

A comment is text which is ignored by the compiler. Comments are usually used to annotate code for future reference or to add notes for others looking at the code. However, they can also be used to make certain lines of code inactive to aid when debugging scripts. If you have programmed in C before then GPC comments will be familiar to you as it uses the same style.

There are two types of comments, the Single Line Comment and the Multi Line Comment.

Single Line Comment

The // (two slashes) characters creates a single line comment and can be followed by any sequence of character. A new line terminates this form of comment. As shown below

```c
main {
    // A single line comment
    if(get_val(XB1_RT) > 95)
        // Another single line comment
        combo_run(RAPID_FIRE);
}
```

Multi Line Comment

The /* (slash, asterisk) characters starts a multi line comment and can also be followed by any sequence of characters. The multi line comment terminates when the first */ (asterisk, slash) is found. As shown below

```c
main {
    /* A multi line comment
    if(get_val(XB1_RT) > 95)
        combo_run(RAPID_FIRE);
    */
}
```

As the comment terminates when a */ (asterisk, slash) is found, this style of commenting cannot be nested. As shown below
main {
    /* A multi line comment
    if(get_val(XB1_RT) > 95)
        combo_run(RAPID_FIRE); /*This will cause a problem*/
*/
}
A Simple Tutorial

This basic tutorial will show you how to create your first script (A basic Rapid Fire Script) and how it could be expanded on.

Step 1

Load Cronus PRO and go to the File Dropdown menu and select New and then Empty File

Step 2

Either type or copy and paste the following code into the GPC code editor within Cronus PRO:

```c
main{
    if(get_val(XB1_RT)){
        combo_run(RAPID_FIRE);
    }
}

combo RAPID_FIRE{
    set_val(XB1_RT, 100);
    wait(40);
    set_val(XB1_RT, 0);
    wait(30);
    set_val(XB1_RT, 0);
}
```
Step 3

Compile the code to check for errors. To do this, either press **F7** on your keyboard or go to the Compiler dropdown menu in Cronus PRO and select compile:

The Output window below the GPC editor should give you this message:

```
------- GPC: Build started -------
> 1: New*: 
> Bytecode size: 148 bytes (3.6%) 
> Stack memory: 3 words (2.7%) 
Build succeeded with 0 warnings...
```

If your CronusMAX PLUS is connected via the PC PROG port and you have controller connected, you can see this script in action by using the Build and Run option. This is accessed by either pressing **F5** or selecting Build and Run in the Compiler Dropdown menu. This function will compile the code and then send it to your CronusMAX PLUS so you can test it.

**Script Breakdown**

What this script does is run the combo named RAPID_FIRE whenever the right trigger has a value or is pressed. If the Right Trigger is still held when the combo ends, it will be run again.

To analyse how the CronusMAX PLUS is told how to do this we must first break the script down into its two sections, the **main** and **combo** sections.
The Main Section

```c
main{
  if(get_val(XB1_RT)){
    combo_run(RAPID_FIRE);
  }
}
```

As explained [here](#), the `main` section is run in a loop by the CronusMAX PLUS. The Virtual Machine in the CronusMAX runs through the code in order and when it reaches the end of the code, data is sent to the console and then the Virtual Machine starts the next loop.

```c
if(get_val(XB1_RT))
```

The above line is telling the CronusMAX that if the statement is TRUE, run the nested code. In this case, if XB1_RT (Right Trigger) has a value greater than 0 (Zero) so is pressed or not at rest.

```c
{
  combo_run(RAPID_FIRE);
}
```

Above is the code nested within the `if` statement. Nesting code creates a hierarchical structure. An open curly bracket ( `{` ) starts the nesting and a closed curly bracket ( `}` ) ends it. By nesting code within the `if` statement we are telling the CronusMAX that we only wish for that code to be executed only when the `if` statement is TRUE. More information on nesting code can be found [here](#).

```c
combo_run(RAPID_FIRE);
```

This line simply tells the CronusMAX to run the `combo` named RAPID_FIRE. It is important to note that if the CronusMAX receives this instruction and the `combo` is already running it will not do anything. It will only run the `combo` again if it has finished. This means that if you hold down the Right Trigger with this code active, the CronusMAX start the `combo` and then run it again as soon as it has ended. Therefore running the `combo` in an indefinite loop or until such time as the Right Trigger is released.
The Combo Section

```c
combo RAPID_FIRE{
    set_val(XB1_RT, 100);
    wait(40);
    set_val(XB1_RT, 0);
    wait(30);
    set_val(XB1_RT, 0);
}
```

This is the **combo** which the CronusMAX is instructed to run when the Right Trigger is pressed. When run, a **combo** runs through the code until it gets to a **wait** statement. The **wait** statement instructs **combo** to execute the commands above it for a set amount of time which is expressed in milliseconds.

```c
set_val(XB1_RT, 100);
wait(40);
```

These lines instruct the **combo** to set the value of the Right Trigger to 100 (or fully pressed) for 40 milliseconds.

```c
set_val(XB1_RT, 0);
wait(30);
```

Once the 40 milliseconds has passed, these lines instruct the **combo** to set the Right Trigger to 0 (Release) for 30 milliseconds.

Additional detail on how a combo operates can be found here.

**Expanding the code**

Now that you understand how this script works, we will make it more complex and change when the **combo** is run.

Look at this line in the main section;

```c
if(get_val(XB1_RT))
```

and change it to;

```c
if(get_val(XB1_RT) && !get_val(XB1_LT))
```
by introducing && !get_val(XB1_LT) in to the if statement we are telling the CronusMAX to only run the combo if the Right Trigger has a value and the Left Trigger does not.

&& means 'and' in GPC and ! means not. So the if statement now reads 'if Right Trigger has a value and Left Trigger does not'. Which means when using this code in game the CronusMAX will only Rapid Fire your gun when you are not aiming down the sights.

You can use the build and run function to see this code in action.
A GPC script can be split into sections. There can be a total of 8 different sections and, in any user made script, they should be laid out in the order shown in the example below. Only the main section is mandatory as it is the heart of any script and is run in a constant loop.

You can download this script [here](#) or search 'Basic GPC' in the GPC library should you wish to load it in Cronus PRO.
Definitions Section (Optional)

```c
define zero = 0;
define one = 1;
define two = 2;
```

The definition section is used to assign values to words thus creating a named constant. Definitions must be placed before the `main` section.

In this example we define zero, one and two to the values 0, 1, 2 respectively. These are static values which cannot be altered during run time. So, should we use the word 'one' at any point in the script it is the same as typing the number 1.

Click [here](#) for more detail on Definitions.

Data Section (Optional)

```c
data (zero, one, two, 10, 128, 40);
```

The data section is an array of bytes (8 bit unsigned integer) which is placed at the start of the Virtual Address Space in GPC bytecode. The values are read only and cannot be altered at run time. The user can access these values via the zero based data array.

For example in this script we could do this to access the array:

```c
example4 = dbyte(4);  //example4 = 128
```

Click [here](#) for more detail on the data array.
Remapping Section (Optional)

\texttt{remap XB1_LB \rightarrow XB1_RB;  
remap XB1_RB \rightarrow XB1_LB;}

In this section we can alter the behavior of the controls. In this example we are telling the Virtual Machine that an value assigned to the Left Bumper should be sent to the Right Bumper instead and vice versa.

It is important to note that button remaps are applied once the main has finished and just before the output report is sent to the console. This means scripting should be programmed without considering the remapping.

For example, if at some point in this script we were to set the Left Bumper to 100 like so;

\texttt{set_val(XB1_LB, 100);}

When the main procedure finishes, the output report would initially contain a value of 100 for the left bumper. When the remaps are processed, this value would then instead be assigned to the Right Bumper and the output report would be modified. When the final output report is sent to the console, it would contain a value of 100 for the Right Bumper instead of the Left.

Click [here](#) for more detail on button Remapping.

---

Variable Initialization Section (Optional)

\texttt{int example1 = 10;  
int example2, example3;  
int example4 = 17;}

This is where the user can create variables. A variable is a point in the stack memory where a value can be placed and they can only be created before the main or init sections. These variables are global and GPC and can be accessed at any point within the script. Variables are not static and can be altered during run time.

If a variable is not assigned a value in this section, it is initialized with the value 0. Such as the variables example2 and example3 in this script.

Click [here](#) for more detail on variables.
GPC Initialisation Section (Optional)

```python
init {
    if(get_controller() == PIO_PS4) {
        example2 = 27;
    } else {
        example2 = 1;
    }
}
```

The `init` section is similar to the `main` with the exception that it is only run once when the script is loaded into the virtual machine. It can run the same commands and functions as the `main` section such as combos and user created functions.

It is generally used to populate variables/arrays and set up the script. In this example, if the CronusMAX PLUS has a PS4 controller connected when the script is first loaded, `example2` is assigned a value of 27. Otherwise it is assigned a value of 1.

Click [here](#) for more detail on the `init` section.
The main section is the heart and sole of any GPC script, all functions/combos are initially executed from this function. It is the only mandatory section and every GPC must have one. Unlike combos and user created functions, a GPC may only have one main section. It is run in a loop. The Virtual Machine runs through the code in order and generates an output report as it goes. When the Virtual Machine gets to the end of the main section, the output report is then ready to be sent to the console. Once the console requests new data, the output report is sent and the main starts another run.

As commands are run through in order, setting the value of a button in more than one place means that only the last command is sent.

For example, in this script;

```
main {
   set_val(XB1_LY, 100);
   set_val(XB1_LY, -100);
}
```

A value of -100 will be sent to the console. The console will not see the LY axis set to 100 because the output report for that control is modified again before it is sent to the console.

Click [here](#) for more detail on the main section.
A combo (which is short for Combination) is a function that will perform a set of instructions in order and for the amount of time assigned to the wait command directly after the commands.

In this script, when the combo is run, it will set identifier 20 (X on an Xbox controller or Square on a Playstation controller) to 100% (Fully pressed) for the time set in the variable example1 (10 milliseconds in this case) and then do nothing for 100 milliseconds.

You can assign multiple commands before a single wait statement. For example, in the following combo, both the Left Bumper and Right Trigger will be pressed for 500 milliseconds (half a second) when the combo is run. It will then do nothing for 500 milliseconds.

```c
combo LB_AND_RT{
    set_val(XB1_LB, 100);
    set_val(XB1_RT, 100);
    wait(500);
    wait(500);
}
```

Click here for more detail on Combos and Wait times.
**Function Section (Optional)**

```javascript
function example_function()
    if(get_val(example2)){
        example3 = 10;
        return 1;
    } else if(get_val(example4)){
        example3 = 19;
        return 1;
    }
    return 0;
}
```

A user created function is similar to the main section. Commands are processed in order and any GPC which is valid in the main section can be used here. Functions must be placed at the end of the of a GPC script.

The main difference with functions is they are only run when called and can return a value. When a value is returned from a function, the function is terminated and any code beyond that point is not executed.

GPC user functions are global, this means that can be called from the **init**, **main** and **combo** sections. A function can even be called from within another function.

Click [here](#) for more detail on User Created Functions.
Definitions

The sole purpose of a definition is assign a value to a word and therefore make a script easier for a human to read. They do not use utilize any bytecode space in a script as they are not sent to the CronusMAX PLUS. When a script is compiled the words are changed to their assigned value.

Syntax

```
define <name> = <value>;
```

- `<name>` The name of the constant.
- `<value>` The value assigned to the constant. Note only Integer values can be used.

Once a word is defined and given a value, that word can be used anywhere in the script where a value is valid, as shown below;

```
define my_value = 50;

int myvar;

main {
    set_val(XB1_RT, my_value);  //Set RT / R2 to 50
    myvar = my_value;           //myvar equals 50
    if(myvar >= my_value) {     //If myvar is greater than or equal to 50
        //Do Something
    }
    if(get_val(XB1_LT) > my_value) {  //If LT / L2 is greater than 50
        //Do Something
    }
}
```
A **define** is a static value and therefore cannot be changed during run time, as shown below;

```plaintext
#define my_value = 50;

main {
    my_value = 70;  //This will produce a compiler error
}
```

If you wished to assign a value to a word and change its value during runtime, you would use a **variable** instead of a **define**.
The **data** section is located at the first portion of the virtual address space within the GPC bytecode and contains static values which cannot be altered during run time.

The main purpose for the **data** section is to store static information and the size of it is determined by the values within it. The static values can be accessed in a GPC through the use of an indexer and definitions can also be used if they are placed before the data section, as shown below.

```c
#define myValue 255;

// Index No. 0 1 2 3 4 5 6 7 8 9 10
#define data ( 20, 42, 35, myValue, 1, 100, 0, 86, 255, 11, 2 );

int Var, var, VAR;

main {
    Var = dbyte(3); // Var = 255
    var = dchar(3); // var = -1;
    VAR = dword(3); // VAR = 511
}
```

The values placed within the data section are expressed in bytes (8 bit unsigned integer). The index is zero based. As you can see above, the first value is index point 0 (zero) and the 11th value would be index point 10.
dbyte function

```
Var = dbyte(3); //Var = 255
```

This function returns a byte value (8 bit unsigned integer) from the selected index within the array.

**Syntax**

dbyte (<Index>);

**Parameters**

<index> : The index of an element within the data section

---

dchar function

```
var = dchar(3); //var = -1;
```

This function returns a char value (8 bit signed integer) from the selected index within the array.

**Syntax**

dchar (<Index>);

**Parameters**

<index> : The index of an element within the data section
dword function

VAR = dword(3); //VAR = 511

This function returns a word value (16 bit signed integer) from the selected index within the array.

Syntax

dword ( <Index> );

Parameters

<index> : The index of an element within the data section
Remapping

Remappings are defined at the beginning of a script and must be done before the main procedure. They cannot be executed during run time.

Although they are defined at the beginning of a script, the Virtual Machine does not execute the remaps until the main section has finished its current iteration. This means that any scripting operations on the remapped entries for inputs should be programmed for the original buttons and without considering the remappings.

For example, if you used the remap command to swap the PS4_CIRCLE button with the PS4_TRIANGLE button and wished for the CIRCLE button on the controller to start a combo, you would still use the CIRCLE button in your code, like so;

```
remap PS4_CIRCLE -> PS4_TRIANGLE;
remap PS4_TRIANGLE -> PS4_CIRCLE;

main {
    if(get_val(PS4_CIRCLE)) { //Input commands are not affected by remaps.
        combo_run(My_Combo); //So to run this combo, you would physically press the CIRCLE button.
    }
}

combo My_Combo{
    set_val(PS4_L3, 100);
    wait(200);
    set_val(PS4_L3, 0);
    wait(190);
}
```

However, as remaps are evaluated after the main procedure is finished, any output commands should consider the remappings. So if you wished for the CIRCLE button to turbo the TRIANGLE button (which it is remapped to) when pressed. You would ignore the remaps for the get_val command but would consider them when creating the output, like so:
Remaps assign the value of the input to the selected output. So you can have one input controlling several outputs. This also means that unless something else is remapped to the input, it will still also output its original function.

For example, if you were to do this;

```c
remap PS4_CIRCLE -> PS4_TRIANGLE;
remap PS4_TRIANGLE -> PS4_CIRCLE;

main {
    if(get_val(PS4_CIRCLE)) {  //Input commands are not affected by remaps.
        combo_run(My_Combo);  //So to run this combo, you would physically press the CIRCLE button.
    }
}

combo My_Combo{
    set_val(PS4_CIRCLE, 100);  //When this combo is run, it will turbo the CIRCLE button.
    wait(40);  //However, when the main section has finished its current iteration,
    set_val(PS4_CIRCLE, 0);  //the Virtual Machine will evaluate the REMAPS and will reassign these
    wait(30);  //commands to the TRIANGLE button.
}
```

The CROSS button would output as CROSS, SQUARE and TRIANGLE simultaneously. The SQUARE and TRIANGLE buttons would not output anything.

**Commands and Syntax**

There are two remapping commands;
**remap**

Assigns the value of the input identifier to the output identifier

**Syntax**

```
remap <input identifier> -> <output identifier>;
```

Remaps are based on the output identifier so any subsequent remaps to the same output identifier will override the previous remap.

Due how remap alter the output report to the console, they do not consume any additional CPU runtime.

**unmap**

Disconnects an input from the output report. This means that although the Virtual Machine can still see the value of the button/axis on the input report, it will not pass its value onto the console in the output report. You can therefore still use an unmapped button to run code or start combos in your GPC script without worrying about its original function being sent to the console. For example;

```
unmap PS4_TRIANGLE;

main {
   if(get_val(PS4_TRIANGLE)){
      combo_run(MyCombo);
   }
}
```

**Syntax**

```
unmap <output identifier>;
```

You can also use the constant ALL_REMAPS to **unmap** all remappings in a single operation, like so;

```
unmap ALL_REMAPS;
```
All variables in GPC are 16 bit signed integers. An integer (from the Latin 'integer' which means 'whole') is a number which can be written without a fractional component. For example, 0, 20, 128 and -1000 are all integers while 4.2, 5.6 or -110.9 are not. Therefore, GPC does not support fractions and will round down any decimal to a zero. Meaning 3.4 would become 3.

16 bit signed means the variables can store an integer ranging from -32768 to +32767

The following sections on variables can be found within this page;

- Declaring Variables
- Boolean Variables
- Arrays
- Accessing any variable via an array

Declaring Variables

A variable is a place where data can be stored in the Virtual Machines memory. A variables name can start with either an underscore ( _ ) or a letter and can be followed by any combination of letters, digits or underscores. They are however case sensitive, so cronusmax, Cronusmax and CRONUSMAX would specify three different variables.

Variables defined this way in GPC are global, this means they can accessed and modified within the init or main sections as well as a combo or function. Only variables assigned to user created functions are local. Details of how variables operate within user functions can be found here.

Global variables must be declared before the main or init sections and therefore cannot be declared after or in either of those sections. As shown below;
Variables are always assigned a value. If no value is assigned when they are declared, then they are initialized with a value of 0 (zero). The value assigned to a variable can be altered during runtime, as shown below:

```
int myVar = 100, MYVar;
int MYVAR = -40;

init {
    int incorrect; //This will cause an error
}

int Incorrect; //This will also cause an error

main {

    int INCORRECT; //This will again cause an error
}
```

**Boolean Variables**

Unlike other programming languages, such as C# for example, GPC does not require a separate variable type for Boolean values. The integers in GPC have been designed to support Boolean expressions. The keywords TRUE and FALSE have been assigned the values 1 and 0 respectively as shown in the example below;

```
int myvar, MYVAR;

main {

    myvar = TRUE;   //myvar = 1
```
MYVAR = FALSE; //MYVAR = 0;

You can therefore use integers in your code to create a toggle switch which enables or disables sections of code, like so;

```c
int myvar;

main {
    if(event_press(XB360_A)) {  //If A / Cross is pressed on the controller
        myvar = !myvar;         //myvar equals not myvar. So if myvar is TRUE or has a value it will be
                                 //made FALSE / assigned the value of 0 (zero)
        //If myvar is FALSE or has a value of 0 (zero) it will be made TRUE / assigned
        the value of 1
    }
    if(myvar) {  //If myvar has a value other than 0 (zero)
        //Do something
    }
}
```

As seen in the above example, a variable will return TRUE in an if statement if it has any value other than 0 (zero). You can however use operators should you wish for your nested code to only be active if a variable is a certain value;

```c
int myvar;

main {
    if(event_press(XB360_A)) {  //If A / Cross is pressed on the controller
        myvar = myvar + 1;      //myvar equals myvar plus 1.
    }
    if(event_press(XB360_X)) {  //If X / Square is pressed on the controller
```
myvar = myvar - 1; //myvar equals myvar minus 1.

if (myvar == 2) { //If mvar has a value of 2.
    //Do something
}

if (myvar >= 4) { //If mvar has a value of greater than or equal to 4.
    //Do something
}

if (myvar < 2) { //If mvar has a value less than 2.
    //Do something
}

Arrays

An array is a method for creating multiple variables in a single command. Arrays are 0 (zero) based so creating a 5 dimensional array will create 5 variables which can be accessed with the indexes 0 through 4, as shown below;

```c
int myvar[5];

main {

    //int myvar[5] creates 5 variables
    myvar[0];  //1st Variable
    myvar[1];  //2nd Variable
    myvar[2];  //3rd Variable
    myvar[3];  //4th Variable
    myvar[4];  //5th Variable

```
Arrays are assigned a value of 0 (zero) when initialized and cannot be assigned a different value before the `init` section. They are also global and can be assigned a value or modified at any point during run time in your script, like so;

```plaintext
int myvar[5];

init {
    myvar[0] = 20;  //myvar[0] equals 20
}

main {
}

function add_two_variables(_1st_variable, _2nd_variable) {
    return _1st_variable + _2nd_variable;
}
```

### Accessing Any Variable via an Array

When a script is compiled, all global variables are placed within an array. You can therefore use an array call to access subsequent variables from a variable which has not be initialized as an array, as shown below;

```plaintext
int myvar;
int _1st = 10;
```
```c
int _2nd = 20;
int _3rd = 30;
int _4th = 40, _5th = 50;
int var_to_place_value_into;

main {
    var_to_place_value_into = myvar[1]; //var_to_place_value_into equals 10 (_1st)
    var_to_place_value_into = myvar[2]; //var_to_place_value_into equals 20 (_2nd)
    var_to_place_value_into = (myvar[4] + myvar[5]) - myvar[3]; //var_to_place_value_into equals 60 ((40 + 50) - 30)
    var_to_place_value_into = _2nd[1]; //var_to_place_value_into equars 30 (_3rd)
}
```
Init Section

The initialization or init section of a GPC is identical to the main section other than it is not run in a loop. The init section is run once before the first iteration of the main section when a script is first loaded and can run any combo or call any function. You can even modify the value of variables within it, just as you would in the main section.

The init section is used to setup your script. For example, say you wished to use the same Rapid Fire script on a Playstation 3 and Xbox One but your Playstation 3 game uses the bumper to shoot where as your Xbox One game uses the trigger, you could use the init section to automatically adjust your script on when it is loaded as shown in the following example;

```c
int FIRE_BTN;

init
{
    if(get_console() == PIO_PS3) { //If connected to a PS3 when loaded
        FIRE_BTN = 3; //3 = LB/L1
    } else { //If connected to any other console
        FIREBTN = 4; //4 = RT/R1
    }
}

main
{
    if(get_val(FIRE_BTN)) {
        combo_run(Rapid_Fire);
    }
}

combo Rapid_Fire {
    set_val(FIRE_BTN, 100);
    wait(50);
    set_val(FIRE_BTN, 0);
    wait(40);
    set_val(FIRE_BTN, 0);
}
```
The **init** section is very useful when you are using persistent variables as you do not want to constantly recall the values stored in them during run time. Information on Persistent Variables and how they can be recalled in the **init** section can be found here.
The **main** section is the heart and sole of any GPC script. It is run in a constant loop and any code executed during run time can be traced back to it. Therefore, the main section is mandatory and a GPC script is not valid without one.

Just as with any nested code, the **main** section start and finish points are denoted with a { or } respectively, like so;

```plaintext
main
{
   //Main Start
}
   //Main End
```

When the main section reaches the End point, the output report to the console is created, any **remaps** are evaluated and the report is sent to the console. The main section is then restarted from the beginning.

Code within the **main** section is executed in the order it is written. It is important to remember this when building your own GPC scripts as the output report to the console is created at the end of the **main** section, so it is possible to create code which would cancel each other out and result in a different output to the console than you expected. For example;

```plaintext
int press_lt;

main{

   if(get_val(XB1_RT)){        //If RT/R2 is pressed..
      press_lt = TRUE;        //Variable 'press_lt' is set to TRUE
   }

   press_lt = FALSE;           //Variable 'press_lt' is set to FALSE

   if(press_lt) {              //This line of code will never see 'press_lt'
      set_val(XB1_LT, 100);   //as TRUE so the code nested within the if
   }                          //Statement will never be active

}
```
As you can see above, the variable 'press_lt' is set to TRUE when the RT/R2 button is pressed but is immediately set to FALSE in the next line of code. Therefore the if(press_lt) statement will always be FALSE and the code nested within the if statement will never be executed.

However, if we were to move the code around, then we would get the expected output;

```c
int press_lt;

main{

    press_lt = FALSE;           //Variable 'press_lt' is set to FALSE

    if(get_val(XB1_RT)){        //If RT/R2 is pressed..
        press_lt = TRUE;        //Variable 'press_lt' is set to TRUE
    }

    if(press_lt) {
        //This line of code will see if 'press_lt'
        set_val(XB1_LT, 100);   //is TRUE so the code nested within the if
    }                           //Statement could be activated

}
```

If RT/R2 is pressed then 'press_lt' is set to TRUE after it has been set to FALSE, the if(press_lt) statement will see that it is TRUE and the nested code will be run. If RT/R2 isn't being pressed then 'press_lt' is not set to TRUE after is has been set to FALSE and the nested code is not run.

If 'press_lt = FALSE;' was nested within an else statement, which is a better way to do it, then it could be placed after the 'press_lt = TRUE;' line, like so;

```c
int press_lt;

main{

    if(get_val(XB1_RT)){        //If RT/R2 is pressed..
        press_lt = TRUE;        //Variable 'press_lt' is set to TRUE
    }
    else {
        press_lt = FALSE;       //Variable 'press_lt' is set to FALSE
    }

    if(press_lt) {
        //This line of code will see if 'press_lt'
        set_val(XB1_LT, 100);   //is TRUE so the code nested within the if
    }                           //Statement could be activated
}
```
The above code will operate as expected because the `else` statement will only be TRUE if RT/R2 is not being pressed. Therefore, 'press_lt' is only set to false when RT/R2 is not pressed.

So, as you can see in the above examples, it is important to remember that code is executed in the order it is written and simply moving the placement of a line can have a significant effect of the output to the console.
A combo (short for combination) is a combination of preprogrammed instructions which are executed in sequence. Just like the main section, commands within a combo are performed in the order they are written. You can run any code you can run in the main section within a combo such as calling functions or setting variables, although this is generally unnecessary and usually results in nothing more than a waste of stack memory and bytecode space. A combo is ideal suited to setting the output of a button for a specific length of time which is done using the wait command, a command that is unique to combos and cannot be used elsewhere.

Just as with variables, a combos name can start with either an underscore (_) or a letter and can be followed by any combination of letters, digits or underscores.
**combo_run**

```java
combo_run (mycombc);
```

combo_run does precisely what the name suggests and runs a combo. However, unlike the `combo_restart` command, it has no effect if the combo is currently running. It will only start a combo if it is not already running.

**Syntax**

```
combo_run ( <Combo Name> );
```

**Parameters**

- `<Combo Name>` : The name assigned to a combo.

---

**combo_running**

```java
combo_running (mycombc)
```

combo_running is a function which can be used in your code to check if a combo is running or not. If the combo named in its parameter is running, then it will return TRUE. If not, it will return FALSE.

**Syntax**

```
combo_running ( <Combo Name> );
```

**Parameters**

- `<Combo Name>` : The name assigned to a combo.
Returns

TRUE if the combo is running, FALSE if it is not

combo_running is particularly useful if you only want certain lines of code executed when the combo has finished. For example, if you want to run a combo 5 times from a single button press, with a combination of combo_running and a variable you can do so as shown in the example below:

```c
int run_combo = 0;

main {
    if(event_press(19)) { //If A / Cross is pressed...
        run_combo = 5; //Variable 'run_combo' equals 5
    }

    if(run_combo && !combo_running(mycombo)) { //If 'run_combo' has a value and mycombo is not running...
        run_combo = run_combo - 1; //run_combo equals 'run_combo' minus 1
        combo_restart(mycombo); //restart mycombo
    }
}

combo mycombo {
    set_val(3, 100); //set RB / R1 to 100
    wait(200); //wait 200 milliseconds
    wait(200); //wait 200 milliseconds
}
```

Device monitor output from the above code;

![Device monitor output](image-url)
As you can see, the output from the above code will press the RB / R1 button 5 times from a single press of the A / Cross Button. As you can see, when using the combo_running function and a variable, it is possible to produce multiple button presses with a small amount of code. You could change the line 'run_combo = 5;' to any value above 0 (zero) and below 32767. The combo would be executed however many times you stipulated.

combo_stop

combo_stop(mycombo);

As the name suggests, combo_stop will stop a combo if it is currently running. As with combo_run, it has no effect if the combo is not currently running.

Syntax

combo_stop ( <Combo Name> );

Parameters

<Combo Name> : The name assigned to a combo.

combo_stop is particularly useful when you only wish for your combo to run when a button is held. To achieve this, you would use it in combination with an else statement as shown below;

```
int run_combo = 0;

main {
  if(get_val(19)) {  //If A / Cross is held..
    combo_run(mycombo);  //run mycombo
  } else {        //If A / Cross is not pressed/held...
    combo_stop(mycombo);  //stop mycombo
  }
}

combo mycombo {
  set_val(3, 100);  //set RB / R1 to 100
  wait(2000);    //wait 2000 milliseconds
  wait(2000);    //wait 2000 milliseconds
}
```
combo_restart

combo_restart(mycombo);

As the name suggests, combo_restart will restart a running combo. If the combo stated within it’s parameters is currently running, it will be restarted from the beginning. If the combo is not currently running, it will be run.

**Syntax**

combo_restart ( <Combo Name> );

**Parameters**

<Combo Name> : The name assigned to a combo.

---

**Wait Command**

wait(200);

The wait command instructs the Virtual Machine within the CronusMAX on how long the last set of commands should be executed for. The length of time they instruct the VM to execute the commands for is represented in milliseconds and can range from 10ms to 4000ms.

The commands executed during the wait time are those placed between the current wait and the previous wait time, the current wait time and previous call command or the start of the combo, whichever comes first. As shown in the example below;

```
combo mycombo {
    set_val(19, 100);//¯\_\_
    set_val(18, 100);//  | These two buttons will be held
    wait(1000);       //←_ for 1000 milliseconds (1 second)
```
```c
set_val(3, 100); // These two buttons will be held
wait(1500); // for 1500 milliseconds (1.5 seconds)

set_val(9, -100); // This axis will be held
wait(2000); // for 2000 milliseconds (2 seconds)
```

The `wait` command can only be used within a combo and must be at the first level of the combo block; it cannot be nested. The example below shows correct and incorrect usage of the `wait` command:

```c
main {
  wait(100); // Incorrect - Will produce an error as this command is
  // not permitted outside of a combo
}

combo mycombo {
  if(get_console() == PIO_PS4) {
    wait(200); // Incorrect - Although this wait command is within a combo,
    // is is not permitted beyond the first level of the combo block
  }
  wait(400); // Correct - This wait command is at the first level of the
  // combo block.
}
function myfunction() {
  wait(300); // Incorrect - Will again produce an error as this command is
  // not permitted outside of a combo
}
Syntax

wait ( <Time> );

Parameters

<Time> : The length of time the last commands should be executed for. Represented in milliseconds - range 10 ~ 4000

Call command

call (mycombo2);
Like the wait command, call is unique to combos and it is not permitted outside of the first level of a combo.

When the call command is used, the current combo it is placed in is paused and the combo within the call commands parameters is executed. Once the called combo has finished, the combo the call command was executed in is then resumed. For example;

```
combo _1st_combo {
    set_val(XB1_B, 100);
    wait(100);
    wait(200);
    call(_2nd_combo);  // _1st_combo is paused until _2nd_combo is finished
    set_val(XB1_Y, 100);
    wait(100);
    wait(200);
}

combo _2nd_combo {
    set_val(XB1_RB, 100);
    wait(100);
    wait(200);
}  //Once this combo has ended, the combo is was called from can resume
```
In the above example, when _1st_combo is run, the sequence of commands sent to the console would be;

- B / Circle is pressed for 100 milliseconds
- No buttons are pressed by the Virtual Machine for 200 milliseconds
- RB / R1 is pressed for 100 milliseconds
- No buttons are pressed by the Virtual Machine for 200 milliseconds
- Y / Triangle is pressed for 100 milliseconds
- No buttons are pressed by the Virtual Machine for 200 milliseconds

As you can see, using the call command injects a combo into a combo at a set point. This is particularly useful when you have two combos which perform identical actions at a point and wish to save space in your script.

**Syntax**

`call ( <Combo Name> );`

**Parameters**

`<Combo Name>`: The name of the combo to be called.
User Created Functions

As well as having a significant number of built in functions, GPC allows the user to create their own custom functions. A function can run any code valid in the main section and code is also executed in the order it is written.

Calling a function

myfunction(10, 20)

To call (or run) a function, you simply type its name and put any parameters it requires in between ( and ).

When a function is called, the code within it is executed and the return value is sent back to where it was called from.

User functions are what is known as global scope, this means they can be called from the init, main and combo sections. They can even be called from within another function, however, GPC does not support recursive calls of functions. This means a function cannot be called from within itself.
Function name and declaration

```function myfunction```

To declare a function type `function` followed by a name and `()`. Within the brackets `()` you place the names of any parameters you would like the function to have, if any. Function names and parameters follow the same rules as a variable, they can start with either an underscore `(_)` or a letter and can be followed by any combination of letters, digits or underscores.

Functions must be declared last in a GPC script though, after the `main` and any `combo` section.

Syntax

```function ( <Parameter(s)> );```  

Parameters

`<Parameter(s)>` : Optional parameters. You can use as many as you wish or none at all. Each one must be separated with a comma `(,)`

Function Parameters

`(_1stvalue, _2ndvalue)`

Function parameters can be thought of like local variables as they cannot be accessed outside of the function they are defined within. A value can be passed to them and they can be used within the function just like a variable could.

As GPC only supports the one data type (16bit Integers) you do not need to specify the data type of parameters within a function and the name of a parameter follows the same rules as a function or variable, they can start with either an underscore `(_)` or a letter and can be followed by any combination of letters, digits or underscores.

Function parameters are optional. You are not required to have any at all. For example, `function myfunction() { }` is perfectly valid.
Returning from a function

```c
return _1stvalue - _2ndvalue;
```

return is a command unique to functions. It is not mandatory for each user function to have a return value though. If there is no return in a function, then 0 (zero) will be automatically returned.

You can have multiple return points within a function. Once the first return command is executed, the function returns a value to where it was called and the function is terminated. The code beyond that point in the function will not be run.

Returning a value is one of the single most useful commands within a function as it can be used as a boolean value to enable or disable sections of code, to set parameters in other functions or to set a variable to a desired value. In the following example, you will see a couple of uses for the return command:

```c
int RF_HOLD = 40;
int RF_NULL = 30;

main {
    if(myfunction()) { //If myfunction returns a value other than 0(zero)...
        if(get_val(XB1_RT)) { //If we get a value from RT / R2 other than 0
            combo_run(Rapid_Fire); //Run combo Rapid_Fire
        }
    }
}

combo Rapid_Fire {
    set_val(XB1_RT, 100);
    wait(RF_HOLD);
    set_val(XB1_RT, 0);
    wait(RF_NULL);
    set_val(XB1_RT, 0);
}
```
function myfunction() {
    if(get_val(XB1_VIEW)) { //If we get a value from View other than 0
        if(get_val(XB1_A))
            RF_HOLD = adjust_speed(RF_HOLD, 10, 1000, 10);
        if(get_val(XB1_B))
            RF_NULL = adjust_speed(RF_NULL, 10, 1000, 10);
        set_val(XB1_A, 0);
        set_val(XB1_B, 0);
        set_val(XB1_LB, 0);
        set_val(XB1_RB, 0);
        set_val(XB1_VIEW, 0);
        set_val(TRACE_1, RF_HOLD / 10);
        set_val(TRACE_2, RF_NULL / 10);
        return 0; //Return 0
    }
    return 1; //If we do not get a value from View, return 1
}

function adjust_speed(var, min_value, max_value, adjustment_increment) {
    if(event_press(XB1_RB) && var < max_value)
        var = var + adjustment_increment;
    if(event_press(XB1_LB) && var > min_value)
        var = var - adjustment_increment;
    return var;
}
Understanding how the example works

When the GPC script is first loaded, the two variables RF_HOLD and RF_NULL are created with a value of 40 and 30 respectively.

The main section then starts its first iteration (run). When it gets to the line:

```c
if (myfunction()) {
```

the function 'myfunction()' is executed.

The code in 'myfunction()' is then run. If XB1_VIEW is not being pressed, the code nested in the statement:

```c
if (get_val(XB1_VIEW)) {
//If we get a value from View other than 0
```

is ignored as the if statement is FALSE. So the next line executed in the function is:

```c
return 1; //If we do not get a value from View, return 1
```

at which point the value of 1 is returned to the statement:

```c
if (myfunction()) {
```

Thus making the above statement TRUE and the code:

```c
if (get_val(XB1_RT)) {
//If we get a value from RT / R2 other than 0
    combo_run(Rapid_Fire); //Run combo Rapid_Fire
}
```

is executed. The main section carries on until it reaches its end point and loops.

However, if XB1_VIEW is being held when 'myfunction()' is executed, then the following code is run:

```c
if (get_val(XB1_A))
    RF_HOLD = adjust_speed(RF_HOLD, 10, 1000, 10);
if (get_val(XB1_B))
```
As you can see, if XB1_A or XB1_B are not also held down, the code set a few buttons to 0, writes the value of our two variables to TRACE values and then most importantly, reaches the line:

```
return 0; //Return 0
```

at which point a value of 0 is returned to the statement;

```
if(myfunction()) {
    if(get_val(XB1_RT)) { //If we get a value from RT / R2 other than 0
        combo_run(Rapid_Fire);  //Run combo Rapid_Fire
    }
}
```

which is nested within that statement is ignored and not executed.

If XB1_VIEW and XB1_A are both held when 'myfunction()' is executed, then the following line of code is reached and run;

```
RF_HOLD = adjust_speed(RF_HOLD, 10, 1000, 10);
```

what the above line means is the variable RF_HOLD equals the return value of the function 'adjust_speed' or you could say the return value from 'adjust_speed' is stored in RF_HOLD. So lets take a look at how that function returns a value.

As you can see above, four values are being sent to the function 'adjust_speed'. The value of RF_HOLD, 10, 1000 and 10. So lets take a look at the declaration of the function 'adjust_speed';
function adjust_speed(var, min_value, max_value, adjustment_increment) {

    function 'adjust_speed' requires 4 arguments, the variable to be adjusted, the minimum value you want it to be, the maximum value you wish for it to be and how much to adjust it by each increment.

    To manipulate the variable, the function executes the following code;

    ```
    if(event_press(XB1_RB) && var < max_value)
        var = var + adjustment_increment;

    if(event_press(XB1_LB) && var > min_value)
        var = var - adjustment_increment;

    return var;
    ```

    In the first part of this code, if XB1_RB is pressed and the variable value passed to the function is less than the maximum value allowed, the value passed in the forth parameter (10 in this case) is added to the value of var. The value of var is the returned to where the function is called. Therefore making RF_HOLD equal 10 more than it did before.

    If XB1_LB is pressed and the variable value passed to the function is greater than the minimum value allowed, the value passed in the forth parameter is subtracted from the value of var. The value of var is the returned to where the function is called. Therefore making RF_HOLD equal 10 less than it did before.

    An identical process is carried out if XB1_VIEW and XB1_B are pressed when 'myfunction()' is executed with the exception being that RF_NULL is adjusted rather than RF_HOLD.
I/O Functions

I/O stands for Input / Output. The two main sources of input into the virtual machine are from the connected controller and the console. As well as receiving inputs from these two sources, the Virtual Machine also outputs to both of them. Within this section of the manual you will find the GPC functions which either receive data from the outside world or affect the data being sent to it.

The contents of I/O Functions

- Controller I/O Functions
- Console I/O Functions
- LEDs
- Rumble
- Identifiers
A controller is mainly an input device, however, the CronusMAX PLUS also sends data to them to control the rumble motors and LEDs. Below are the GPC commands related to this data

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_val</td>
<td>Returns the current value of a controller entry</td>
</tr>
<tr>
<td>get_lval</td>
<td>Returns the previous value of a controller entry</td>
</tr>
<tr>
<td>get_ptime</td>
<td>Returns the elapsed time of a controller entries state change</td>
</tr>
<tr>
<td>get_controller</td>
<td>Returns the type of controller currently connected to the input port</td>
</tr>
<tr>
<td>get_battery</td>
<td>Returns the current status of the battery for a wireless controller</td>
</tr>
<tr>
<td>setLed</td>
<td>Sets the LED state of the controller connected to the input port</td>
</tr>
<tr>
<td>set_rumble</td>
<td>Sets the rumble motors to a set speed in %</td>
</tr>
<tr>
<td>event_press</td>
<td>Returns TRUE when a controller entry has been pressed</td>
</tr>
<tr>
<td>event_release</td>
<td>Returns TRUE when a controller entry has been released</td>
</tr>
<tr>
<td>block</td>
<td>Blocks the output of a controller entry for a set amount of time</td>
</tr>
<tr>
<td>swap</td>
<td>Swaps the output of two controller entries</td>
</tr>
</tbody>
</table>
**sensitivity** Adjusts the sensitivity of an analogue controller entry

**deadzone** Adjusts the deadzone of two controller entries

**stickize** Transforms the values of a mouse or Wiimote IR input to an analog stick

**ps4_touchpad** Returns detailed information on the Dualshock 4 touchpad state

**turn_off** Turns off a wireless controller connected to the input port

**wiir_offscreen** Returns TRUE if the IR sensor on a Wiimote is off screen

---

get_val

get_val returns the current value of a controller entry in the form of an **int**. This means it returns a number which represents a percentage %.

As GPC supports treating an **int** as a boolean value, the get_val command can be used to see if a controller entry simply has a value. For example;

```c
if(get_val(PS4_R1))
```

would return TRUE if R1 / RB had a value other than 0 (zero).

It can also be used to check the specific value if a controller entry. For Example;

```c
if(get_val(PS4_R2) > 50)
```

would return TRUE if the R2 / RT axis was at a value greater than 50 %

As get_val returns an **int**, you can also pass its return value in to a **variable**, like so;

```c
int myvar;

main {
```
myvar = get_val(XB1_LT);

}  

Syntax

get_val ( <identifier> );

Parameters

<identifier> : the identifier of a controller entry

Returns

The current value of the specified identifier. Can range from -100 to +100 depending on the entry type.

get_lval

get_lval is similar to get_val with the exception that it returns the value of the specified identifier in the previous iteration (run) of the main loop. This value is also returned as an int and represents a percentage %.

However, get_lval gets its data from the input report so, unlike get_val, it is not affected by any code before it. For example, if you were to use the set_val command to overwrite the output of an identifier, get_lval would still return the previous value of said identifier. You can see this in action by running the following command on your CronusMAX PLUS;

```c
main {
  set_val(XB1_RT, 0);  // Set the value of RT / R2 to 0
  set_val(TRACE_1, get_lval(XB1_RT));   // Even though RT / R2 has been set to 0 (zero) before this line of code
  // it will still return a value if the trigger is pressed
}  
```
set_val(TRACE_2, get_val(XB1_RT));  // This will always return 0 because RT / R2 has
been set to 0 (zero)

// before this line of code

Syntax

get_lval ( <identifier> );

Parameters

<identifier> : the identifier of a controller entry

Returns

The previous value of the specified identifier. Can range from -100 to +100 depending on the entry type.

get_ptime

get_ptime returns the value in milliseconds of an identifier's state change in the form of an int. What this means is when an identifier's value changes from FALSE to TRUE or vice versa, the counter for get_ptime on that identifier is reset to 0.

Therefore the clock is always running for this function so it should be used with another command, such as get_val.

For example, using get_ptime in an if statement on its own like so;

if(get_ptime(XB360_A) > 200)

would give an undesired result as the statement would be TRUE is the button was pressed or not for greater than 200 milliseconds.
Using `get_ptime` in combination with the command `get_val` would modify the statement to only return `TRUE` if the button had been pressed for longer than 200 milliseconds, as shown below:

```c
if(get_val(XB360_A) && get_ptime(XB360_A) > 200)
```

### Syntax

```c
get_ptime (<identifier>);
```

### Parameters

`<identifier>` : the identifier of a controller entry

### Returns

The elapsed time of a controller entries state change. Value returned is in milliseconds with a range of 0 ~ 4000.

---

**get_controller**

`get_controller` returns a value in the form of a `int` which represents the controller type currently connected to the input port of the CronusMAX PLUS.

0 (zero) is returned if no controller is connected and a value of 1 ~ 5 is returned if a controller is connected depending on the type of controller is connected.

To save you from remembering which value relates to which type of controller, 5 constants have been created. They are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIO_PS3</td>
<td>Playstation 3 Controller</td>
<td>1</td>
</tr>
<tr>
<td>PIO_XB360</td>
<td>Xbox 360 Controller</td>
<td>2</td>
</tr>
</tbody>
</table>
Example of usage:

```c
main {
    if(get_controller() == PIO_XB1) {
        // Do Something
    }
}
```

**Syntax**

`get_controller();`

**Parameters**

None

**Returns**

A value which represents which type of controller is currently connected
get_battery

get_battery returns the battery level, if applicable, of the connected controller in the form of an int ranging from 0 ~ 10. With 0 being discharged and 10 being fully charged.

If no battery is connected, for example a wired controller is connected, then it returns 10.

Example of usage;

```c
main {
    if(get_battery() <= 2) {
        // Do Something
    }
}
```

Syntax

get_battery();

Parameters

None

Returns

A value ranging from 0 (Discharged) ~ 10 (Fully Charged)
set_led

set_led sets the state of an LED on the controller.

The LEDs range from 0 ~ 3. Four constants have been created to make it easier to remember which value is assigned to which LED:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED_1</td>
<td>LED 1 / Xbox 360 Quadrant 1</td>
<td>0</td>
</tr>
<tr>
<td>LED_2</td>
<td>LED 2 / Xbox 360 Quadrant 2</td>
<td>1</td>
</tr>
<tr>
<td>LED_3</td>
<td>LED 3 / Xbox 360 Quadrant 3</td>
<td>2</td>
</tr>
<tr>
<td>LED_4</td>
<td>LED 4 / Xbox 360 Quadrant 4</td>
<td>3</td>
</tr>
</tbody>
</table>

An LED can be set to one of four states using this function which range from 0 ~ 3, as shown in the table below;

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LED Off</td>
</tr>
<tr>
<td>1</td>
<td>LED On</td>
</tr>
<tr>
<td>2</td>
<td>LED Blink Fast</td>
</tr>
<tr>
<td>3</td>
<td>LED Blink Slowly</td>
</tr>
</tbody>
</table>

Example of usage:

```c
set_led(LED_1, 3); // Make LED 1 blink slowly
```

**Syntax**

```c
set_led (<led_identifier>, <state>);
```
Parameters

<led_identifier> : the identifier of an LED  
<state> : Numeric value which represents the state, as shown in the table above

Click [here](#) for more information on setting LED states

7

set_rumble

set_rumble sets the speed of the chosen rumble motor on the controller.

The rumble motors are numbered 0 ~ 3. To make it easier to remember which motor is which, four constants have been created:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUMBLE_A</td>
<td>Strong Rumble Motor (Usually the Left Motor)</td>
<td>0</td>
</tr>
<tr>
<td>RUMBLE_B</td>
<td>Weak Rumble Motor (Usually the Right Motor)</td>
<td>1</td>
</tr>
<tr>
<td>RUMBLE_RT</td>
<td>Right Trigger Motor (Xbox One controllers only)</td>
<td>2</td>
</tr>
<tr>
<td>RUMBLE_LT</td>
<td>Left Trigger Motor (Xbox One controllers only)</td>
<td>3</td>
</tr>
</tbody>
</table>

Example of usage;

```c
set_rumble(RUMBLE_A, 50);
```

Syntax

set_rumble ( <rumble_identifier>, <speed as %> );
Parameters

<rumble_identifier> : the identifier of a Rumble Motor
<speed as %> : Numerical value, range 0 ~ 100

Click here for more information on setting Rumble Motors

---

**event_press**

event_press returns TRUE in the main iteration when a control changes from FALSE to TRUE. Therefore, even if a button is pressed and held down, event_press would only return TRUE at the moment it is pressed and not while the button was held. This makes it perfect for combos you only wish to run once when a button is pressed.

For example, if you were playing a first person shooter, using a sniper rifle and you wanted the CronusMAX PLUS to automatically scope when you aimed down you sights. You could do this;

```plaintext
main {
  if(event_press(XB1_LT)){ // If LT / L2 is pressed...
    combo_run(scope_in);    // Run combo scope_in
  }
}

combo scope_in {
  wait(400);               // Wait for the ADS animation to finish
  set_val(XB1_RS, 100);    // Press Right Stick to Zoom scope
  wait(200);               // Press RS / R3 for 200ms
}
```
Syntax

event_press (<identifier>);

Parameters

<identifier> : the identifier of a controller entry

event_release

event_release is the opposite of event_press, it returns TRUE in the main iteration when a control changes from TRUE to FALSE. This makes it ideally suited to run code you which you only want run once when a button is released.

For example, if you were playing a shooter game and wanted the gun to be automatically reloaded whenever you stopped shooting, you could do this;

```c
main {
    if(event_release(XB1_RT)){ // If RT / R2 is pressed...
        combo_run(reload); // Run combo reload
    }
}

combo reload {
    wait(200); // Wait for the gun animation to finished
    set_val(XB1_X, 100); // Press X / Square to reload
    wait(200); // Press X / Square for 200ms
}
```
Syntax

event_release ( <identifier> );

Parameters

<identifier> : the identifier of a controller entry

block

block prevents the forwarding of a controller entry for a set period of time which is set in milliseconds. This time can range from 20 to 4000 milliseconds. It is extremely useful when you wish to get two uses from a single button.

For example, if you were playing a platform game and wished for the CronusMAX PLUS to automatically perform a double jump for you if you held the button down, you could do this;

```c
main {
  if(get_val(XB1_A)){  // If A / Cross is held...
    if(event_press(XB1_A)){  // When first pressed....
      combo_run(single_jump); // Run combo single jump
    }
    block(XB1_A, 500); // Block the forwarding of A / Cross from the controller for 500 ms
  }
}
combo single_jump {
  set_val(XB1_A, 100);
  wait(300);
}```
The above code will run the **combo** single_jump as soon at the button is pressed. If the button is released within 500 milliseconds then there is no further output to the console. However, if the button is held for longer than 500 milliseconds then the normal output of that button resumes.

Output with a quick tap of the button;

Output when the button is held;

**Syntax**

```
block ( <identifier> , <milliseconds> );
```

**Parameters**

- `<identifier>`: the identifier of a controller entry
- `<milliseconds>`: Length of time in milliseconds to block forwarding for. Allowed range 20 ~ 4000
swap

swap does as the name implies, it swaps the values of two controller entries. This makes it useful for remapping buttons on the fly.

For example, if you were playing a shooter game which zooms the scope with a Right Stick Click when you Aim Down the Sights but you wished to move that function to the Right Bumper Button so it does not interfere with your aiming, you could do this;

```c
main {

    if(get_val(XB1_LT)) { // If LT / L2 is pressed...

        swap(XB1_RS, XB1_RB); // Swap RS /R3 and RB / R2

    }

}
```

With the above code, whenever LT / L2 is pressed, RB / R2 will press the Right Stick click and vice versa.

**Syntax**

swap ( <identifier1> , <identifier2> );

**Parameters**

<identifier1> : the identifier of a controller entry
<identifier2> : the identifier of a controller entry
sensitivity

sensitivity adjusts the sensitivity of an analogue controller entry, usually this is an axis.

The function takes three parameters, the control to be modified, the midpoint and the sensitivity multiplier.

Midpoint sets the midpoint value of the controller entry. The default value is 50%. By changing this value, you are setting two sensitivity ranges. With a value lower than 50% a high sensitivity range is created closer to the rest position and a low sensitivity range when far from the rest position. A value above 50% creates the opposite effect.

The Sensitivity multiplier is the amount the input value is multiplied by. The parameter is passed in percentage. So 40 would mean multiply by 0.40, 100 means multiply by 1.00 and 140 means multiply by 1.40.

How these translate into the difference between the input into the CronusMAX PLUS and the output to the console can be seen below.

With a midpoint of 35% and a sensitivity of 100 using the command - sensitivity(XB1_LX, 35, 100);

With a midpoint of 70% and a sensitivity of 140 using the command - sensitivity(XB1_LX, 70, 140);
Syntax

sensitivity ( <identifier> , <midpoint> , <sensitivity> );

Parameters

<identifier> : the identifier of a controller entry
<midpoint> : sets the midpoint value
<sensitivity> : the ratio of the sensitivity adjustment

deadzone

deadzone adjusts the values of the output to alter the deadzone of two axis. The default deadzone programmed into consoles is 20%, this means a console will ignore any signal from an analogue stick which is below 20%. The CronusMAX PLUS can adjust the output signals relative to the input.

For example, if you wished to remove 10% of the deadzone, the CronusMAX PLUS will output and additional 10% on both axis with the command;

deadzone(XB1_LX, XB1_LY, 10, 10);

With the above command, a physical movement of 10% on the Left stick of either the Horizontal or Vertical Axis would output as 20%. An additional 10% is added to the output up until 90%. Any movement from 90% or greater will output 100%, as shown on the graph below;
The above code will create a square deadzone, that means 10% will be applied to both sticks at all times. However, the deadzone function can also be used to create a circular deadzone using the predefined constant, DZ_CIRCLE (which equals 101).

When you use DZ_CIRCLE (or 101) in the third parameter instead of a value, the forth parameter then sets the radius of the circle instead of the value for the Y axis.

Example when using the DZ_CIRCLE constant;

```c
deadzone(XB1_LX, XB1_LY, DZ_CIRCLE, 10);
```

**Syntax**

```c
deadzone ( <identifier_x> , <identifier_y> , <dzone_x> / DZ_CIRCLE , <dzone_y> / <radius> );
```

**Parameters**

- `<identifier_x>`: a controller entry which represents an X axis
- `<identifier_y>`: a controller entry which represents a Y axis
- `<dzone_x> / DZ_CIRCLE`: X axis deadzone value / DZ_CIRCLE constant
- `<dzone_y> / <radius>`: Y axis deadzone value / The radius value of the circle
stickize

stickize transforms the values of a Wiimote IR or mouse input to an analog stick. It does this by setting the radial output of the translation from their movements to the analogue stick, as shown in the diagram below;

<TODO>  Insert Diagram

As the PS3 has a square output for its analogue sticks, we recommend a value of 129 is used on that console. The Xbox 360, Xbox One and PS4 all use a radial output so we recommend a value of 113 is used on those consoles.

Syntax

stickize ( <identifier_x> , <identifier_y> , <radius> );

Parameters

<identifier_x> : a controller entry which represents an X axis
<identifier_y> : a controller entry which represents a Y axis
<dzone_y> / <radius> : The radius value of the circle

ps4_touchpad

ps4_touchpad returns detailed information on the current state of the touchpad. Like get_val, it returns an int. ps4_touchpad can give you information on where two fingers are positioned on the touchpad and their X / Y coordinates.

This is done via six constants;
<table>
<thead>
<tr>
<th>PS4T Constant</th>
<th>Value</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS4T_P1</td>
<td>1</td>
<td>TRUE if at least one finger is on the touchpad, FALSE if none</td>
</tr>
<tr>
<td>PS4T_P1X</td>
<td>2</td>
<td>X axis value of the first finger on the touchpad</td>
</tr>
<tr>
<td>PS4T_P1Y</td>
<td>3</td>
<td>Y axis value of the first finger on the touchpad</td>
</tr>
<tr>
<td>PS4T_P2</td>
<td>16</td>
<td>TRUE if two fingers are on the touchpad, FALSE if less than two.</td>
</tr>
<tr>
<td>PS4T_P2X</td>
<td>32</td>
<td>X axis value of the second finger on the touchpad</td>
</tr>
<tr>
<td>PS4T_P2Y</td>
<td>48</td>
<td>Y axis value of the second finger on the touchpad</td>
</tr>
</tbody>
</table>

To get the value of a one of the above constants, simply use it in the `ps4_touchpad` parameter. For example, to read the state of all the constants, you could do this;

```c
main {
    if(ps4_touchpad(PS4T_P1)) {  // If at least one finger is on touchpad...
        set_val(TRACE_1, 1); // Set TRACE_1 to 1
        set_val(TRACE_2, ps4_touchpad(PS4T_P1X)); // Set TRACE_2 to X coordinate of touchpoint 1
        set_val(TRACE_3, ps4_touchpad(PS4T_P1Y)); // Set TRACE_3 to Y coordinate of touchpoint 1
    } else { // If no fingers are on the touchpad...
        set_val(TRACE_1, 0); // Set TRACE_1 to 0
    }

    if(ps4_touchpad(PS4T_P2)) { // If two fingers is on touchpad...
        set_val(TRACE_4, 1); // Set TRACE_4 to 1
        set_val(TRACE_5, ps4_touchpad(PS4T_P2X)); // Set TRACE_5 to X coordinate of touchpoint 2
        set_val(TRACE_6, ps4_touchpad(PS4T_P2Y)); // Set TRACE_6 to Y coordinate of touchpoint 2
    } else { // If less than two fingers on the touchpad...
        set_val(TRACE_4, 0); // Set TRACE_4 to 0
    }
}
```
Syntax

```c
ps4_touchpad ( <PS4T_constant> );
```

Parameters

```c
<PS4T_constant> : A constant from the table above
```

Returns

An `int` value related to the PS4T_ constant used

```
turn_off
```

turn_off will switch off a wireless controller connected to the CronusMAX PLUS input port.

Example of usage;

```c
main {
    if(event_press(XB1_RS)) { // If RS / R3 is pressed...
        turn_off(); // Turn off wireless controller
    }
}
```

Syntax

```c
turn_off ( );
```
Parameters

None

wiir_offscreen

wiir_offscreen checks to see if the Wiimote controller is pointing off screen.

Example of usage;

```c
main {
    if(wiir_offscreen()) {
        // Do Something
    }
}
```

Syntax

wiir_offscreen ( );

Parameters

None

Returns

TRUE if the Wiimote IR is pointing off screen, FALSE if it is not
The CronusMAX PLUS mainly outputs data to a console, however, it does receive data such as rumble and led states which we can read in GPC scripts. Below are the GPC commands related to this data.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_val</td>
<td>Overwrites the current value of a controller entry</td>
</tr>
<tr>
<td>get_console</td>
<td>Returns the current type of console connected to the output port</td>
</tr>
<tr>
<td>output_reconnection</td>
<td>Disconnects and Reconnects the USB connection on the output port</td>
</tr>
<tr>
<td>ps4_authtimeout</td>
<td>Returns the PS4 authentication timeout state</td>
</tr>
<tr>
<td>get_led</td>
<td>Returns the current state of a specified LED</td>
</tr>
<tr>
<td>get_rumble</td>
<td>Returns the current value of a rumble motor</td>
</tr>
</tbody>
</table>
set_val

set_val overwrites the current value of a controller entry with the value that is specified in its second parameter. What this means is whatever the output is from the controller for the specified button/axis, the set_val command will overwrite that value with the value you specify.

It is mainly used in **compos** to set buttons in sequence, however, it can also be used in the **main** or user created **function**.

For example, if you were playing a shooter game and wished to create Hair Triggers. A **combo** would not be suitable for such a function as you would want the output constant for however long the triggers are held down. You could use the set_val command within the **main** section to achieve that function, like so;

```c
main {
    if(get_val(XB1_LT)) { // If LT / L2 is pressed
        set_val(XB1_LT, 100); // Set LT / L2 to 100 (Fully Pressed)
    }
    if(get_val(XB1_RT)) { // If RT / R2 is pressed
        set_val(XB1_RT, 100); // Set RT / R2 to 100 (Fully Pressed)
    }
}
```

**Syntax**

```c
set_val ( <identifier> );
```

**Parameters**

```c
<identifier> : the identifier of a controller entry
```
get_console

get_console returns a value in the form of a int which represents the type of console currently connected to the output port of the CronusMAX PLUS.

0 (zero) is returned if no console is connected and a value of 1, 2, 4 or 5 is returned if a console is connected, depending on the type of console connected.

To save you from remembering which value relates to which type of controller, 4 constants have been created. They are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIO_PS3</td>
<td>Playstation 3</td>
<td>1</td>
</tr>
<tr>
<td>PIO_XB360</td>
<td>Xbox 360</td>
<td>2</td>
</tr>
<tr>
<td>PIO_PS4</td>
<td>Playstation 4</td>
<td>4</td>
</tr>
<tr>
<td>PIO_XB1</td>
<td>Xbox One</td>
<td>5</td>
</tr>
</tbody>
</table>

Example of usage:

```c
main {
    if(get_console() == PIO_XB1) {
        // Do Something
    }
}
```

Syntax

get_console ( );
Parameters

None

Returns

A value which represents which type of console is currently connected.

output_reconnection

output_reconnection forces the CronusMAX PLUS to electronically disconnect the output port from the console and then reconnect again. This function was mainly used to reset the authentication timeout on the PS4, however, it is redundant now as there is no timeout when using a USB HUB on Firmware 1.20 and above. It has been left in for legacy CronusMAX Users.

Example of usage;

```c
main {

    if(get_val(XB1_MENU) && event_press(XB1_XBOX)) {  // If MENU / OPTIONS / START is held and XBOX / PS button is pressed...
        output_reconnection();  // Disconnect & Reconnect to console
        set_val(XB1_XBOX, 0); // Zero XBOX / PS button to prevent it interfering with game
    }
}
```

Syntax

output_reconnection (  );

Parameters

None
ps4_authtimeout

ps4_authtimeout returns the authentications timeout status on the PS4 in the form of an `int`. As with output_reconnection, this function is redundant since Firmware 1.20 and above as Partial PS4 cross over support is no longer required. However, the function has not been removed as legacy CronusMAX users and those not using a USB Hub would still require it.

The main function of ps4_authtimeout is to enable you to script a warning when the CronusMAX PLUS is close to automatically disconnecting and reconnecting to the console when the authentication times out on a PS4. This could be achieved using a script from our online library;

```c
int authcount;
int NOTIFY = 3; // 1 = 30 secs 2 = 1 min 3 = 1 min 30 secs and so on up to 16.

main{
    //PS4 Cross Over Gaming Section
    if(get_console() == PIO_PS4 && get_controller() != PIO_PS4){
        authcount = ps4_authtimeout();
        swap(1,27); // Swap Back/Select with Touchpad Press
        if(get_val(27) && get_val(5)) { // Press Back/Select and press RS/R3 to press share.
            set_val(27, 0);
            set_val(5, 0);
            set_val(1, 100);
        }
        if(get_val(27)) { // Hold Back/Select and Press A/Cross to reset Auth Timeout.
            if(event_press(19)){
                output_reconnection();
            }
            set_val(19, 0);
        }
        if(authcount <= NOTIFY + 1) {
            combo_run(notify);
        }
    }
    //PS4 Cross Over Gaming Section End
}
```
```c
combo notify{
    set_rumble(RUMBLE_A, 100);
    set_rumble(RUMBLE_B, 100);
    wait(150);
    reset_rumble();
    wait(250*authcount);
}
```

## Syntax

```c
ps4_authtimeout();
```

## Parameters

None

## Returns

The PS4 authentication timeout status. This is a decedent count down with 1 being the last value returned before an Automatic Reconnection is performed by the CronusMAX PLUS.

## get_led

`get_led` returns a value in the form of an `int` which represents the current state of the chosen LED.

The LEDs range from 0 ~ 3. Four constants have been created to make it easier to remember which value is assigned to which LED;
The return value from this function informs you of the current state of the selected LED. The function returns a value ranging from 0 ~ 3:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LED Off</td>
</tr>
<tr>
<td>1</td>
<td>LED On</td>
</tr>
<tr>
<td>2</td>
<td>LED Blinking Fast</td>
</tr>
<tr>
<td>3</td>
<td>LED Blinking Slowly</td>
</tr>
</tbody>
</table>

Example of usage:

```c
main {
    if(get_led(LED_2) == 1) { // If LED 2 is On
        // Do Something
    }
}
```

**Syntax**

```c
get_led ( <Led_Identifier> );
```

**Parameters**

*<Led_Identifier>* : The identifier of an LED
Returns

An int ranging from 0 ~ 3 which represents the current state

Click here for more information on LED states

get_rumble

get_rumble returns the speed of the chosen rumble motor on the controller in the form of an int. The value returned can range from 0 ~ 100 which represents the speed in a percentage ( % ).

The rumble motors are numbered 0 ~ 3. To make it easier to remember which motor is which, four constants have been created;

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUMBLE_A</td>
<td>Strong Rumble Motor (Usually the Left Motor)</td>
<td>0</td>
</tr>
<tr>
<td>RUMBLE_B</td>
<td>Weak Rumble Motor (Usually the Right Motor)</td>
<td>1</td>
</tr>
<tr>
<td>RUMBLE_RT</td>
<td>Right Trigger Motor (Xbox One controllers only)</td>
<td>2</td>
</tr>
<tr>
<td>RUMBLE_LT</td>
<td>Left Trigger Motor (Xbox One controllers only)</td>
<td>3</td>
</tr>
</tbody>
</table>

Example of usage;

```c
main {
    if(get_rumble(RUMBLE_A) > 50) {  // If Rumble Motor A is running greater than 50%
        // Do Something
    }
}
```
**Syntax**

```c
get_rumble ( <rumble_identifier> );
```

**Parameters**

<rumble_identifier> : the identifier of a Rumble Motor

**Returns**

An `int` ranging from 0 ~ 100 which represents the current speed of the chosen motor

Click here for more information on setting Rumble Motors
The CronusMAX PLUS allows you to control the LEDs on your controller if the controller supports it. Setting the LEDs is useful when you wish to create a visual indicator of an active function in your scripts.

On a PS3, Wii and Xbox 360 controller, you have 4 leds which can be controlled independently although note that Xbox 360 controllers only allow one led to be lit at any time. The PS4’s native controller, the DS4, has one lightbar which supports an array of colors. The Xbox One official controller does not support LED functions at this time.

Functions related to LEDs

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get Led</td>
<td>Returns the current value of a LED state</td>
</tr>
<tr>
<td>set Led</td>
<td>Set the state of a LED</td>
</tr>
<tr>
<td>set LedX</td>
<td>Blinks a LED a certain number of times</td>
</tr>
<tr>
<td>get LedX</td>
<td>Checks if a LED is being blinked by the set LedX function</td>
</tr>
<tr>
<td>reset LEDs</td>
<td>Reset the LEDs state to what was set by the console</td>
</tr>
</tbody>
</table>

Additional Instructions in this section

Setting DS4 lightbar

LED Constants

Controllable LEDs range from 0 ~ 3. To make it easier to remember which value relates to which LED, the following Constants are available;
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED_1</td>
<td>LED 1 / Xbox 360 Quadrant 1</td>
<td>0</td>
</tr>
<tr>
<td>LED_2</td>
<td>LED 2 / Xbox 360 Quadrant 2</td>
<td>1</td>
</tr>
<tr>
<td>LED_3</td>
<td>LED 3 / Xbox 360 Quadrant 3</td>
<td>2</td>
</tr>
<tr>
<td>LED_4</td>
<td>LED 4 / Xbox 360 Quadrant 4</td>
<td>3</td>
</tr>
</tbody>
</table>

**get_led**

`get_led` returns a value in the form of an **int** which represents the current state of the chosen LED.

The return value from this function informs you of the current state of the selected LED. The function returns a value ranging from 0 ~ 3:

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LED Off</td>
</tr>
<tr>
<td>1</td>
<td>LED On</td>
</tr>
<tr>
<td>2</td>
<td>LED Blinking Fast</td>
</tr>
<tr>
<td>3</td>
<td>LED Blinking Slowly</td>
</tr>
</tbody>
</table>

**Example of usage;**

```c
main {
    if(get_led(LED_2) == 1) { // If LED 2 is On
        // Do Something
    }
}
```
Syntax

get_led ( <Led_Identifier>  );

Parameters

<Led_Identifier> : The identifier of an LED

Returns

An int ranging from 0 ~ 3 which represents the current state

set_led

set_led sets the state of an LED on the controller. When an led state is set, it remains set until such time as the LEDs are reset, it is set again in your GPC script or the script is unloaded.

An LED can be set to one of four states using this function which range from 0 ~ 3, as shown in the table below;

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LED Off</td>
</tr>
<tr>
<td>1</td>
<td>LED On</td>
</tr>
<tr>
<td>2</td>
<td>LED Blink Fast</td>
</tr>
<tr>
<td>3</td>
<td>LED Blink Slowly</td>
</tr>
</tbody>
</table>

Example of usage;

```plaintext
set_led(LED_1, 3); // Make LED 1 blink slowly
```
Syntax

set_led ( <led_identifier> , <state> );

Parameters

<led_identifier> : the identifier of an LED
[state] : Numeric value which represents the state, as shown in the table above

set_ledx

setLedX is used to Blink and LED a set amount of times. You can blink and led from 0 to 255 times. 0 sets the LED to on.

Example of usage:

```
main {
    if(!get_ledx()) { // If the leds are not blinking...
        set_led(LED_1, 10); // Blink LED 1 10 times
    }
}
```

Syntax

set_ledx ( <led_identifier> , <no_of_blinks> );
Parameters

<led_identifier> : the identifier of an LED
<no_of_blinks>   : The number of times to blink the LED

get_ledx

get_ledx checks to see if an LED is currently being blinked by the set_ledx function.

Example of usage;

```c
main {
    if(get_ledx()) { // If the leds are blinking...
        // Do Something
    }
}
```

Syntax

get_ledx ( );

Parameters

None

Returns

TRUE is the LEDs are being blinked by the set_ledx function, FALSE if they are not
reset_leds

reset_leds returns control of the LEDs to the console and disables any current LED states which are being set by the Virtual Machine.

Example of usage;

```c
main {
    if(event_press(XB1_A)) { // If A / Cross is Pressed...
        reset_leds(); // Reset Leds
    }
}
```

**Syntax**

```c
reset_leds();
```

**Parameters**

None

---

**Setting DS4 lightbar**

The Dualshock 4 controller has one lightbar instead of four LEDs. The color of the lightbar can be controlled by setting all four led states simultaneously.
For example, the following code will set the lightbar to green:

```c
main {
    if(event_press(PS4_CROSS)) {
        set_led(LED_1, 0);
        set_led(LED_2, 0);
        set_led(LED_3, 2);
        set_led(LED_4, 0);
    }
}
```

To save you from remembering all the different combinations, we have created this script which simplifies the task of setting the color with a custom function. You can add your own code to this and use the function to create visual notifications in your GPC script;

```c
// Dualshock 4 lightbar custom function
//
// Written by the CronusMAX Team
//
// For the CronusMAX PLUS manual www.cronusmax.com/manual/leds.htm#id_6

define Off = 0;
define Dim_Blue = 1;
define Dim_Red = 2;
define Dim_Green = 3;
define Dim_Pink = 4;
define Dim_SkyBlue = 5;
define Dim_Yellow = 6;
define Dim_White = 7;
define Blue = 8;
define Red = 9;
define Green = 10;
define Pink = 11;
define SkyBlue = 12;
define Yellow = 13;
define White = 14;
define Bright_Blue = 15;
define Bright_Red = 16;
```
```
define Bright_Green = 17;
define Bright_Pink = 18;
define Bright_SkyBlue = 19;
define Bright_Yellow = 20;
define Bright_White = 21;

data {
    0,0,0,0, // Off
    1,0,0,0, // Dim Blue
    0,1,0,0, // Dim Red
    0,0,1,0, // Dim Green
    0,0,0,1, // Dim Pink
    1,0,1,0, // Dim SkyBlue
    0,1,1,0, // Dim Yellow
    1,1,1,1, // Dim White
    2,0,0,0, // Blue
    0,2,0,0, // Red
    0,0,2,0, // Green
    2,0,2,0, // SkyBlue
    0,2,2,0, // Yellow
    2,2,2,2, // White
    3,0,0,0, // Bright Blue
    0,3,0,0, // Bright Red
    0,0,3,0, // Bright Green
    0,0,0,3, // Bright Pink
    3,0,3,0, // Bright SkyBlue
    0,3,3,0, // Bright Yellow
    3,3,3,3 // Bright white
}

main {

    // Examples of usage
    if(event_press(PS4_CROSS)) { // If Cross is pressed...
        set_ds4_led(Green); // Set the lightbar to Green
    }
}
```
if(event_press(PS4_CIRCLE)) { // If Circle is pressed...

    set_ds4_led(Red); // Set the lightbar to Red
}

if(event_press(PS4_OPTIONS)) { // If Options is pressed...

    reset_leds(); // Reset the LEDs
}

function set_ds4_led(colour) {

    set_led(LED_1, dbyte(colour * 4));
    set_led(LED_2, dbyte((colour * 4) + 1));
    set_led(LED_3, dbyte((colour * 4) + 2));
    set_led(LED_4, dbyte((colour * 4) + 3));
}
The CronusMAX PLUS allows you to completely control the Rumble Motors on your controller, including the Trigger Rumble motors on an Xbox One controller. Below are the GPC commands relating to these motors.

### Rumble Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_rumble</td>
<td>Returns the current value of a Rumble Motor</td>
</tr>
<tr>
<td>set_rumble</td>
<td>Set the speed of a Rumble Motor</td>
</tr>
<tr>
<td>block_rumble</td>
<td>Blocks any rumble signals from the console</td>
</tr>
<tr>
<td>reset_rumble</td>
<td>Resets the rumble state and returns control of the motors to the console</td>
</tr>
</tbody>
</table>

### Rumble Constants

Rumble motors are numbered from 0 ~ 3. To make it easier to remember which Motor relates to which number, the following Constants are available:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUMBLE_A</td>
<td>Strong Rumble Motor (Usually the Left Motor)</td>
<td>0</td>
</tr>
<tr>
<td>RUMBLE_B</td>
<td>Weak Rumble Motor (Usually the Right Motor)</td>
<td>1</td>
</tr>
<tr>
<td>RUMBLE_RT</td>
<td>Right Trigger Motor (Xbox One controllers only)</td>
<td>2</td>
</tr>
<tr>
<td>RUMBLE_LT</td>
<td>Left Trigger Motor (Xbox One controllers only)</td>
<td>3</td>
</tr>
</tbody>
</table>
get_rumble

get_rumble returns the speed of the chosen rumble motor on the controller in the form of an int. The value returned can range from 0 ~ 100 which represents the speed in a percentage (%).

Example of usage;

```c
main {
    if(get_rumble(RUMBLE_A) > 50) { // If Rumble Motor A is running greater than 50%
        // Do Something
    }
}
```

Syntax

```c
get_rumble ( <rumble_identifier> );
```

Parameters

```c
<rumble_identifier> : the identifier of a Rumble Motor
```

Returns

An int ranging from 0 ~ 100 which represents the current speed of the chosen motor
set_rumble

set_rumble sets the speed of the chosen rumble motor on the controller. Once a rumble has been activated by a script, it will remain at the speed set until such time as the script sets it again, rumble is reset or the script is unloaded.

Example of usage;

```c
main {
    if(event_press(XB1_A)) {  // If A / Cross is pressed...
        set_rumble(RUMBLE_A, 50);  // Set rumble motor A to 50% speed
    }
}
```

Syntax

```c
set_rumble ( <rumble_identifier> , <speed as %> );
```

Parameters

- `<rumble_identifier>` : the identifier of a Rumble Motor
- `<speed as %>` : Numerical value, range 0 ~ 100

block_rumble

block_rumble does as it implies and blocks any rumble signals to the controller. Once this function is used, it remains active until such time as it is reset in the script or the script is unloaded.
Example of usage;

```c
main {
    if(event_press(XB1_A)) {  // If A / Cross is pressed...
        block_rumble();  // Block rumble signals to the controller
    }
}
```

Syntax

```
block_rumble();
```

Parameters

None

reset_rumble

reset_rumble returns control of the rumble motors to the console. It also deactivates block_rumble if it is active.

Example of usage;

```c
main {
    if(event_press(XB1_A)) {  // If A / Cross is pressed...
        reset_rumble();  // Reset the rumble state
    }
}
```
Syntax

reset_rumble ( );

Parameters

None
## Identifiers

The table below lists all the controller identifiers in GPC, the index value they relate to and the readable range from each one:

<table>
<thead>
<tr>
<th>Index</th>
<th>PS4</th>
<th>PS3</th>
<th>XB1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PS4_PS</td>
<td>PS3_PS</td>
<td>XB1_XBOX</td>
</tr>
<tr>
<td>1</td>
<td>PS4_SHARE</td>
<td>PS3_SELECT</td>
<td>XB1_VIEW</td>
</tr>
<tr>
<td>2</td>
<td>PS4_OPTIONS</td>
<td>PS3_START</td>
<td>XB1_MENU</td>
</tr>
<tr>
<td>3</td>
<td>PS4_R1</td>
<td>PS3_R1</td>
<td>XB1_RB</td>
</tr>
<tr>
<td>4</td>
<td>PS4_R2</td>
<td>PS3_R2</td>
<td>XB1_RT</td>
</tr>
<tr>
<td>5</td>
<td>PS4_R3</td>
<td>PS3_R3</td>
<td>XB1_RS</td>
</tr>
<tr>
<td>6</td>
<td>PS4_L1</td>
<td>PS3_L1</td>
<td>XB1_LB</td>
</tr>
<tr>
<td>7</td>
<td>PS4_L2</td>
<td>PS3_L2</td>
<td>XB1_LT</td>
</tr>
<tr>
<td>8</td>
<td>PS4_L3</td>
<td>PS3_L3</td>
<td>XB1_LS</td>
</tr>
<tr>
<td>9</td>
<td>PS4_RX</td>
<td>PS3_RX</td>
<td>XB1_RX</td>
</tr>
<tr>
<td>10</td>
<td>PS4_RY</td>
<td>PS3_RY</td>
<td>XB1_RY</td>
</tr>
<tr>
<td>11</td>
<td>PS4_LX</td>
<td>PS3_LX</td>
<td>XB1_LX</td>
</tr>
<tr>
<td>12</td>
<td>PS4_LY</td>
<td>PS3_LY</td>
<td>XB1_LY</td>
</tr>
<tr>
<td>13</td>
<td>PS4_UP</td>
<td>PS3_UP</td>
<td>XB1_UP</td>
</tr>
<tr>
<td>14</td>
<td>PS4_DOWN</td>
<td>PS3_DOWN</td>
<td>XB1_DOWN</td>
</tr>
<tr>
<td>15</td>
<td>PS4_LEFT</td>
<td>PS3_LEFT</td>
<td>XB1_LEFT</td>
</tr>
<tr>
<td>16</td>
<td>PS4_RIGHT</td>
<td>PS3_RIGHT</td>
<td>XB1_RIGHT</td>
</tr>
<tr>
<td>17</td>
<td>PS4_TRIANGLE</td>
<td>PS3_TRIANGLE</td>
<td>XB1_Y</td>
</tr>
<tr>
<td>18</td>
<td>PS4_CIRCLE</td>
<td>PS3_CIRCLE</td>
<td>XB1_B</td>
</tr>
<tr>
<td>19</td>
<td>PS4_CROSS</td>
<td>PS3_CROSS</td>
<td>XB1_A</td>
</tr>
<tr>
<td>20</td>
<td>PS4_SQUARE</td>
<td>PS3_SQUARE</td>
<td>XB1_X</td>
</tr>
<tr>
<td>21</td>
<td>PS4_ACCX</td>
<td>PS3_ACCX</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>PS4_ACCY</td>
<td>PS3_ACCY</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>PS4_ACCZ</td>
<td>PS3_ACCZ</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>PS4_GYROX</td>
<td>PS3_GYRO</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>PS4_GYROY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>PS4_GYROZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>PS4_TOUCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>PS4_TOUCHX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>PS4_TOUCHY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XB360</td>
<td>WII/NUNCHUCK</td>
<td>CLASSIC CTRL PRO</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>XB360_XBOX</td>
<td>0</td>
<td>100</td>
<td>WII_HOME</td>
</tr>
<tr>
<td>XB360_BACK</td>
<td>0</td>
<td>100</td>
<td>WII_MINUS</td>
</tr>
<tr>
<td>XB360_START</td>
<td>0</td>
<td>100</td>
<td>WII_PLUS</td>
</tr>
<tr>
<td>XB360_RB</td>
<td>0</td>
<td>100</td>
<td>WII_RT</td>
</tr>
<tr>
<td>XB360_RT</td>
<td>0 ~ 100</td>
<td>WII_ZR</td>
<td>0 ~ 100</td>
</tr>
<tr>
<td>XB360_RS</td>
<td>0</td>
<td>100</td>
<td>WII_ONE</td>
</tr>
<tr>
<td>XB360_LB</td>
<td>0</td>
<td>100</td>
<td>WII_C</td>
</tr>
<tr>
<td>XB360_LT</td>
<td>0 ~ 100</td>
<td>WII_Z</td>
<td>0</td>
</tr>
<tr>
<td>XB360_LS</td>
<td>0</td>
<td>100</td>
<td>WII_TWO</td>
</tr>
<tr>
<td>XB360_RX</td>
<td>-100 ~ 100</td>
<td>WII_RX</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td>XB360_RY</td>
<td>-100 ~ 100</td>
<td>WII_RY</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td>XB360_LX</td>
<td>-100 ~ 100</td>
<td>WII_NX</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td>XB360_LY</td>
<td>-100 ~ 100</td>
<td>WII_NY</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td>XB360_UP</td>
<td>0</td>
<td>100</td>
<td>WII_UP</td>
</tr>
<tr>
<td>XB360_DOWN</td>
<td>0</td>
<td>100</td>
<td>WII_DOWN</td>
</tr>
<tr>
<td>XB360_LEFT</td>
<td>0</td>
<td>100</td>
<td>WII_LEFT</td>
</tr>
<tr>
<td>XB360_RIGHT</td>
<td>0</td>
<td>100</td>
<td>WII_RIGHT</td>
</tr>
<tr>
<td>XB360_Y</td>
<td>0</td>
<td>100</td>
<td>WII_X</td>
</tr>
<tr>
<td>XB360_B</td>
<td>0</td>
<td>100</td>
<td>WII_B</td>
</tr>
<tr>
<td>XB360_A</td>
<td>0</td>
<td>100</td>
<td>WII_A</td>
</tr>
<tr>
<td>XB360_X</td>
<td>0</td>
<td>100</td>
<td>WII_Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_ACCX</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_ACCY</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_ACCZ</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_ACCNX</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_ACCNY</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_ACCNZ</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_IRX</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WII_IRY</td>
<td>-100 ~ 100</td>
</tr>
<tr>
<td>DEBUG FIELDS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE_1</td>
<td>-128 ~ 127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE_2</td>
<td>-128 ~ 127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE_3</td>
<td>-128 ~ 127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE_4</td>
<td>-128 ~ 127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE_5</td>
<td>-128 ~ 127</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE_6</td>
<td>-128 ~ 127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

-- Values Range --

(0 | 100) means 0 or 100
(0 ~ 100) means any value between 0 and 100
(-100 ~ 100) means any value between -100 and 100

Accelerometers: 25% represents 1G; 100% == 4G
Internal Functions & Commands

Within this section of the manual you will find the functions and commands the Virtual Machine uses to handle data in GPC scripts.

The contents of Internal Functions & Commands

Flow Control Commands
Math Functions
Bit Operations
Persistent Variables
Miscellaneous Functions
Flow control commands are integral to most GPC scripts as they allow you control the flow of executed code and tell the GPC how and when to make decisions. Without any flow control commands in place, all your combos would run at once and you would have probably end up with a useless script. Below are the GPC commands you can use to control the flow of code;

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if</strong></td>
<td>Evaluates an expression to its boolean value</td>
</tr>
<tr>
<td><strong>else</strong></td>
<td>Executes alternative code when an if statement is FALSE</td>
</tr>
<tr>
<td><strong>else if</strong></td>
<td>Executes alternative code when an if statement is FALSE and its expression is TRUE</td>
</tr>
<tr>
<td><strong>while</strong></td>
<td>Loops code until its expression is FALSE or the break command is used</td>
</tr>
</tbody>
</table>
The **if** command is one of the single most useful tools for your GPC scripting needs. It allows you to control when certain blocks of code are run and it takes any expression, function or code which returns or has a value. It can also be used in the **init**, **main**, **combo** and **function** sections of your code.

The **if** command works by evaluating the expression within it to its boolean value.

An expression is anything which has a value. Such as a function call which returns a value, a variable, a mathematical sum. literal values and comparisons.

In computer programming languages, a Boolean is a data type with only the two possible values, TRUE or FALSE. In GPC, FALSE means 0 and TRUE is a value other than 0 be it a negative or positive value.

If the expression placed within an **if** commands brackets is TRUE, then the code nested within it is executed, as shown below;

```plaintext
// | --------------- |
// | If this is    |
// ↓ true ↓
if ( expression )
{
  //<--------- |
  //          | Run code placed here
} //<--------- |
```

If the expression is FALSE, then the code contained within the block is ignored.

You can also nest an **if** command within another **if** command in as many times as the bytecode limit will allow in your script. As shown below with other examples of **if** command usage;

```plaintext
int x = 5, y = 10, z = 20;

init {
  if(get_console() == PIO_PS4)
    x = 30;
}
```c
main {

    if(get_val(7)) {
        if(x < y) {
            if(get_val(9) < -50)
                x = y + z;
            if(get_val(9) > 50)
                x = y + 20;
        }
    }
    if(x <= (z - y)) {
        z = y;
        x = 6;
    }
}
```

**else**

Should you wish to execute a different block of code when an if command does not return TRUE then you would use the else command.

An else command must have an if command preceding it. The code block contained within an else command will be executed if the expression in the if statement returns FALSE, as shown below;

```c
int ADS;

main {

    if(get_val(XB1_LT)) // if LT / L2 is pressed...
        ADS = TRUE;
    else // else (If LT / L2 is not pressed)
        ADS = FALSE;
}
```

The above code will set the variable ADS to TRUE if LT / L2 is pressed and will set it to FALSE if it is not.
else if

**else if** is a combination of **else** and **if**. Just like the **else** command it allows for a different block of code to be executed when the statement within the **if** command returns FALSE. However, it will only execute the code block when the statement within its parameter returns TRUE.

For example, if you were playing a first person shooter game and wished for rapid fire to be disabled when you aim down the sights but wanted the CronusMAX PLUS to automatically hold breath in the game to steady your aim while aiming down the sights, the following code would do this for you;

```c
main {
    if(get_val(XB1_LT)) // if LT / L2 is pressed...
        combo_run(Auto_HB);
    else if(get_val(XB1_RT)) // else if LT / L2 is not pressed and RT / R2 is....
        combo_run(Rapid_Fire);
}

combo Auto_HB {
    wait(400);
    set_val(XB1_LS, 100);
    wait(100);
    wait(2000);
}

combo Rapid_Fire {
    set_val(XB1_RT, 100);
    wait(40);
    set_val(XB1_RT, 0);
    wait(30);
    set_val(XB1_RT, 0);
}
```

You can also extend an **else if** command with an **else** command or another **else if** statement.
if(get_val(XB1_LT)) // if LT / L2 is pressed...
    combo_run(Auto_HB);
else if(get_val(XB1_RT)) // else if LT / L2 is not pressed and RT / R2 is....
    combo_run(Rapid_Fire);
else {
    sensitivity(XB1_RX, 50, 125);
    sensitivity(XB1_RY, 50, 125);
}
}

while

Other than the main section, the while command is the only loop function in GPC. As the main function runs in a loop, there really isn't a need to use this command. If you do choose to use it, although it can be used in the main section, it is recommended to only use it within a function so as not to temporarily halt the execution of a main iteration.

The operation of the while command is straight forward, it will execute code nested within it until the expression contained in its parameter is no longer TRUE;

main {

    if(event_press(XB1_A))
        _myfunction(10);
}

function _myfunction(var) {

    while(var) {
        var = var - 1;
    }
}
Should you wish to be able to end the while loop early, the break command has been created to make this possible. You would simply create a condition, which when TRUE, would call the break command thus ending the while loop, like so:

```c
main {
    if(event_press(XB1_A))
        _myfunction(10);
}

function _myfunction(var) {
    while(var) {
        var = var - 1;

        if(event_press(XB1_B))
            break;
    }
}
```
Math Functions

In this section are the GPC functions used to perform specific mathematical operations. Two important things to note when working with perform math task in GPC is that it is a signed 16bit environment so all operations must work within that range which is -32768 to +32767 and that GPC only supports integer values which means any fractions will be rounded down to a whole value. For example 7 / 3 = 3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Returns an absolute value</td>
</tr>
<tr>
<td>inv</td>
<td>Returns an inverted value</td>
</tr>
<tr>
<td>pow</td>
<td>Raise and value to the specified power</td>
</tr>
<tr>
<td>isqrt</td>
<td>Calculates an integer square root</td>
</tr>
</tbody>
</table>

abs

The abs command returns the absolute value of an expression. An absolute value is a number without regard for its sign, for example the absolute value of 8 and -8 is 8. An absolute value can also be thought of as its distance from zero which is always a positive value.

One of the popular uses for the abs command is when working with axis to start combos. Such as in fighting games where many users tend to use the right stick to start a combo. So if you wished for the same combo to be run if for example the right stick was pushed left or right, instead of using an or ( || ) operator or using two if commands you could just do this;

```c
1 int a, b;
2 main {
3     if( abs(get_val(XB1_RY)) <25 ) // if the RY axis has an absolute value less than 25...
4         set_val(XB1_RY, 0); // Set RY to zero
5     set_val(XB1_LY, inv(get_val(XB1_LY)) ); // Set LY to the inverse of its current value
6     a = pow(4, 3); // a = 64 (4^3 or 4 x 4 x 4)
7     b = isqrt(25); // b = 5 (5 x 5 = 25)
8 }
9
10
11
12
13
14
```
if( abs(get_val(XB1_RX)) > 70 ) // if RX is greater than 70 or less than -70...
    combo_run(my_combo);
}

combo my_combo {
    //Do something
}

### Syntax

```plaintext
abs( <expression> );
```

### Parameters

- `<expression>`: any expression which has a value

### Returns

The absolute value of the expression

inv

inv returns the inverted value of an expression or number. This means a positive value will be turned in to a negative value and vice versa, which is the same as multiplying the value by -1. For example;

```plaintext
int a = 100;
int b = -50;

main {
    a = inv(a); // a = -100
    b = inv(b); // b = 50
}
```
One of the popular uses for this command is to invert the right sticks Y axis. For example, if you prefer to play with an inverted aim but come across a game which doesn't support it, with one line of code you can have the CronusMAX PLUS invert the axis for you. Like so;

```c
main {
    set_val(XB1_RY, inv(get_val(XB1_RY))); // Set RY to the inverse of its current value
}
```

### Syntax

```c
inv ( <expression> );
```

### Parameters

- `<expression>`: any expression which has a value

### Returns

The inverted value of the expression

---

**pow**

`pow` raises the given value to the power specified. What this means is a value will be multiple by itself X amount of times. For example;

```c
int a;

main {
    a = pow(5, 3); // a = 125 (5³) or (5 * 5 * 5)
}
```
This function must be used with caution as there is a risk of an integer overflow when using it. This would occur when the function attempts to return a value greater than 32767 which is the maximum value for a signed 16 bit integer.

**Syntax**

\[ \text{pow}(\text{expression1}, \text{expression2}); \]

**Parameters**

<expression1> : any expression which has a value
<expression2> : any expression which has a value

**Returns**

The inverted value of the expression

\[ \text{isqrt} \]

isqrt returns the square root of a given value. The square root of a value is the value which when multiplied by itself equals the given value. For example, the square root of 25 is 5 (5 * 5 = 25). The return value is an integer which means any fractions will be dropped. As shown below;

```c
int a;

main {
    a = isqrt(10);  // a = 3. The square root of 10 is 3.16...
    // GPC supports integer values only so the fraction is dropped.
}
```
**Syntax**

isqrt ( <expression> );

**Parameters**

<expression> : any expression which has a value

**Returns**

The square root of the given expression
GPC allows you to manipulate the bits of a given variable. Bit operations are quite complicated, however, there is not really much call for them in the GPC environment and most users will never need them. Therefore, this section will assume you have an understanding of bits, bit masks, how they correlate with bytes and the binary system.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_bit</td>
<td>Sets one bit</td>
</tr>
<tr>
<td>clear_bit</td>
<td>Clears one bit</td>
</tr>
<tr>
<td>test_bit</td>
<td>Tests a bit</td>
</tr>
<tr>
<td>set_bits</td>
<td>Stores a value into a bit index</td>
</tr>
<tr>
<td>get_bits</td>
<td>Gets a value from a bit index</td>
</tr>
</tbody>
</table>
set_bit

set_bit(a, 5);
Sets one bit of a variable based on its bit index

Syntax

set_bit ( <variable>, <bit_index> );

Parameters

<variable> : any defined variable
<bit_index> : index point of the bit to be set, range 0~15

clear_bit

clear_bit(a, 5);
Clears one bit of a variable based on its bit index

Syntax

clear_bit ( <variable>, <bit_index> );

Parameters

<variable> : any defined variable
<bit_index> : index point of the bit to be set, range 0~15
test_bit

```c
test_bit(b, 2);
```
Tests a bit index point in a variable to check if it is TRUE or FALSE (1 or 0)

**Syntax**

```c
test_bit (<variable>, <bit_index>);
```

**Parameters**

- `<variable>`: any defined variable
- `<bit_index>`: index point of the bit to be set, range 0~15

**Returns**

TRUE is the bit is set, FALSE if it is not.

set_bits

```c
set_bits(c, 3, 4, 15);
```
Stores a value in to a variable based on its bit index and a bit mask.

**Syntax**

```c
set_bits (<variable>, <value>, <bit_index>, <bit_mask>);
```
Parameters

<variable> : any defined variable
<value> : anything that has a value (constants, variables, functions, expressions, etc...)
<bit_index> : index point of the bit to be set, range 0~15
<bit_mask> : bit mask corresponding to the size, in bits, of the value to store (without shifting)

get_bits

get_bits(c, 4, 15)

Extracts a value from a variable based on a bit index and bit mask

Syntax

get_bits ( <variable>, <bit_index>, <bit_mask> );

Parameters

<variable> : any defined variable
<bit_index> : position of the less significant bit in the value to extract, range 0~15
<bit_mask> : bit mask corresponding to the size, in bits, of the value to store (without shifting)

Returns

An integer value.
Persistent Variables

Persistence in programming terms means a state which remains after the process that created it has ended. A word processor or Paint application achieves this by saving the document to a file for example. The CronusMAX PLUS does this by writing variable values to its EEPROM (Electrically Erasable Programmable Read-Only Memory). This allows you to save the value of a variable so it can be recalled the next time the script is loaded. There is space on the CronusMAX PLUS for a total of 160 persistent variables, 16 Global Variables and 144 Private Variables (16 for each slot).

Global variables can be accessed and set from any slot. They are generally used to pass values between scripts. 16 constants have been created for use with the get and set commands for persistent variables, they are:

PVAR_1, PVAR_2, PVAR_3, PVAR_4,
PVAR_5, PVAR_6, PVAR_7, PVAR_8,
PVAR_9, PVAR_10, PVAR_11, PVAR_12,
PVAR_13, PVAR_14, PVAR_15, PVAR_16,

Private variables are private to a specific slot, each of the 9 slots has 16 private variables which are used to save specific values for one script. For example Slot 1 has 16 private variables which no other Slot can access, you cannot read or set the value of Slot 1’s variables from Slot 4 and setting the private variables in Slot 1 have any affect on the private variables in any other slot. 16 constants have been created for use with the get and set commands for persistent variables, they are;

```c
int a, b, c;

int init {
    a = get_pvar(SFVAR_1, 0, 10, 5);
    b = get_pvar(SFVAR_2, 20, 40, 30);
    c = get_pvar(PVAR_3, 0, 400, 100);
}

main {
    if(event_press(XB1_VIEW))
        save_pvars();
}

function save_pvars() {
    set_pvar(SFVAR_1, a);
    set_pvar(SFVAR_2, b);
    set_pvar(PVAR_1, c);
}
```
To retrieve the value stored in a persistent variable or to set the value of one, the following functions are available:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_pvar</td>
<td>Returns the value stored within a Persistent Variable</td>
</tr>
<tr>
<td>set_pvar</td>
<td>Stores a value into a Persistent Variable</td>
</tr>
</tbody>
</table>

get_pvar

get_pvar returns the value stored in a Persistent Variable while allowing you to specify the minimum and maximum permissible value and a default value should the value stored be outside of that range. The min, max and default parameters are mainly intended for when you are retrieving values from a Global variable, however, they must still be specified when reading the value of a private variable.

Example of usage:

```c
int a, b, c;

init {
    a = get_pvar(SPVAR_1, 0, 10, 5);
    b = get_pvar(SPVAR_2, 20, 40, 30);
    c = get_pvar(PVAR_1, -30, 400, 100);
}

main {
}
```

Syntax

get_pvar ( <pvar_constant>, <min_value>, <max_value>, <default_value> );
Parameters

<pvar_constant> : A global or private persistent variable constant

<min_value> : The minimum permissible value

<max_value> : The maximum permissible value

<default_value> : The default value to return should the retrieved value be less than the min_value or greater than the max_value

Returns

The stored value or the default value if the stored one is out of range

set_pvar

set_pvar stores the specified value in to a Persistent Variable.

It is important to remember that each time this function is called, data is written to the EEPROM on the CronusMAX PLUS. The life of an EEPROM is typically rated in the amount of read/write cycles that can be performed and although the EEPROM in the CronusMAX is rated for 1000's of these, you should still ensure that this function is NOT going to called in every iteration of the main loop. Never use it at the base level of the main and always ensure it is nested within an if statement which will only return TRUE for one iteration, such as event_press;

```c
int a, b, c;

init {
  a = get_pvar(SPVAR_1, 0, 10, 5);
  b = get_pvar(SPVAR_2, 20, 40, 30);
  c = get_pvar(PVAR_1, -30, 400, 100);
}

main {
```
```
if(event_press(XB1_VIEW)){
    set_pvar(SPVAR_1, a);
    set_pvar(SPVAR_2, b);
    set_pvar(PVAR_1, c);
}

Syntax

get_pvar ( <pvar_constant> , <value> );

Parameters

<pvar_constant> : A global or private persistent variable constant
<value> : A value to be stored
In this section are the commands which do not fit elsewhere.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_rtime</td>
<td>Returns the elapsed time in milliseconds between main iterations</td>
</tr>
<tr>
<td>get_slot</td>
<td>Returns the active slot number</td>
</tr>
<tr>
<td>load_slot</td>
<td>Loads a specified slot</td>
</tr>
<tr>
<td>get_ctrlbutton</td>
<td>Returns the identifier of the controller button</td>
</tr>
<tr>
<td>vm_tctrl</td>
<td>Sets the vm timeout for the next iteration</td>
</tr>
</tbody>
</table>
get_rtime

get_rtime returns the elapsed time between the current and previous iteration of the main function. The value returned is in milliseconds.

You can see this function in action by using this counter script;

```c
int days;
int hours;
int minutes;
int seconds;
int milliseconds;

main {
    milliseconds = milliseconds + get_rtime();

    if(milliseconds >= 1000) {
        milliseconds = milliseconds - 1000
        seconds = seconds + 1;
        if(seconds == 60) {
            seconds = 0;
            minutes = minutes + 1;
            if(minutes == 60) {
                minutes = 0;
                hours = hours + 1;
                if(hours == 24) {
                    hours = 0;
                    days = days + 1;
                }
            }
        }
    }
    set_val(TRACE_1, days);
    set_val(TRACE_2, hours);
    set_val(TRACE_3, minutes);
    set_val(TRACE_4, seconds);
```
set_val(TRACE_5, milliseconds / 10);

}\n
Syntax

get_rtime ( );

Parameters

None

Returns

The elapsed time, in milliseconds, between main iterations

get_slot

get_slot returns an int which represents the number of the currently active slot.

Example of usage;

```c
int _currentSlot;

init {
    _currentSlot = get_slot();
}

main {
}
```
Syntax

get_slot (  );

Parameters

None

Returns

The number of the currently active slot

load_slot

load_slot will attempt to load the slot number specified within its parameter. If there is no script current stored in the specified slot, then it will unload the current slot and load slot 0.

Example of usage;

```plaintext
main {
    if(event_press(XB1_RB))  // if RB / R2 is pressed...
        load_slot(5);        // Load slot 5
    if(event_press(XB1_LB))  // if LB / L2 is pressed...
        load_slot(0);        // Unload current slot and load slot 0
}
```
load_slot ( <slot_number> );

Parameters

<slot_number>: A value while represents a slot number to load, range 0~9

get_ctrlbutton

get_ctrlbutton returns the current control button.

The control button is set in the Device tab within Cronus PRO's Options window. The enable remote control of slow switch on device dictates which button it is set to.

If enable remote control is disabled, then the control button is 0 (The Xbox / Guide / PS button)

If enable remote control is enabled and G8 Alternate is disabled, then the control button is 1 (Back / View / Select / Share)

If both remote control and G8 Alternate are enabled, then the control button is 8 (LS / L3)
Syntax

get_ctrlbutton (  );

Parameters

None

Returns

0, 1 or 8 depending on the Remote slot settings in device options

---

vm_tctrl

vm_tctrl sets the virtual machine timeout for the next iteration. By default, the virtual machine runs the main loop every 10 milliseconds as it aids stability. You can however adjust how often each main iteration is run. Just be aware than changing this setting may cause instability within your script.

Example of usage:

```c
main {
    vm_tctrl(-5); // Run the VM every 5ms
}
```

Syntax

vm_tctrl ( <Value> );

Parameters

<Value> : Numeric value to add to the VM base time. Range -9 ~ 10
An operator is a symbol which tells the interpreter to perform specific mathematical, relational or logical operation and produce final result. This section details the operators available in GPC.
Assignment

'=' is the assignment operator. Think of this as get sets to rather than equal to. When '=' is used, the left operand gets set to the value of the right operand.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Sets the left operand to the value of the right operand</td>
<td>a = 5; //a is set to 5</td>
</tr>
</tbody>
</table>

Arithmetic

It is often necessary to perform arithmetic on two values. The following table lists the arithmetic operators available in GPC.

In the examples, assume 'a' holds a value of 10 and 'b' holds a value of 5.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Adds two operands</td>
<td>a + b will give a value of 15</td>
</tr>
<tr>
<td>-</td>
<td>Subtracts right operand from the left operand</td>
<td>a - b will give a value of 5</td>
</tr>
<tr>
<td>*</td>
<td>Multiples both operands</td>
<td>a * b will give a value of 50</td>
</tr>
<tr>
<td>/</td>
<td>Divides left operand by right operand</td>
<td>a / b will give a value of 2</td>
</tr>
<tr>
<td>%</td>
<td>Modulus, gives the remainder of an integer division</td>
<td>a % b will give a value of 0</td>
</tr>
</tbody>
</table>

Note, GPC does not support fractions so the division operator ( / ) will drop any fractions. For example, 10 / 3 = 3 as the fraction is dropped. It also does not round, so 3 / 4 = 0 and not 1.
Logical

Logical operators are important in any programming language as they allow to tell the interpreter to make decisions based on certain conditions. The following table lists the logical operators within the GPC language.

In the examples, assume 'a' holds a value of 1 and 'b' holds a value of 0.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>AND operator, if both operators are TRUE then the condition becomes TRUE</td>
<td>(a &amp;&amp; b) is FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^^</td>
<td>XOR operator, if either operand is TRUE but not both then the condition becomes TRUE</td>
<td>(a ^^ b) is TRUE</td>
</tr>
<tr>
<td>!</td>
<td>NOT operator. Reverses the logical state of an operand.</td>
<td>!b is TRUE !a is FALSE</td>
</tr>
</tbody>
</table>
Relational operators produce boolean results (TRUE or FALSE) while comparing two operands. The following tables list the relational operands which are available in GPC.

In the examples, assume 'a' holds a value of 30 and 'b' holds a value of 10.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to, if the left operand holds the same value as the right then the condition becomes TRUE</td>
<td>(a == b) is FALSE</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to, if the left operand does not hold the same value as the right then the condition becomes TRUE</td>
<td>(a != b) is TRUE</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than, if the left operand holds a value greater than the left then the condition becomes TRUE</td>
<td>(a &gt; b) is TRUE</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than, if the left operand holds a value less than the left then the condition becomes TRUE</td>
<td>(a &lt; b) is FALSE</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to, if the left operand holds a value which is greater than or equal to the right then the condition becomes TRUE</td>
<td>(a &gt;= b) is TRUE</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to, if the left operand holds a value which is less than or equal to the right then the condition becomes TRUE</td>
<td>(a &lt;= b) is FALSE</td>
</tr>
</tbody>
</table>
List of GPC Keywords

Below is a list of the keywords within GPC. You cannot use any of the following words as constants, variables, combo or function names.

1. abs -- Math Functions
2. block -- I/O Functions
3. block_rumble -- Rumble
4. break -- Flow Control
5. call -- Combos
6. clear_bit -- Bit Operations
7. combo -- Combos
8. combo_restart -- Combos
9. combo_run -- Combos
10. combo_running -- Combos
11. combo_stop -- Combos
12. data -- Data Segment
13. dbyte -- Data Segment
14. dchar -- Data Segment
15. deadzone -- I/O Functions
16. define -- Basic Syntax
17. dword -- Data Segment
18. else -- Flow Control
19. event_press -- I/O Functions
20. event_release -- I/O Functions
21. get_bits -- Bit Operations
22. get_battery -- Misc Functions and Constants
23. get_console -- Misc Functions and Constants
24. get_controller -- Misc Functions and Constants
25. get_ctlbutton -- Misc Functions and Constants
26. get_led -- LEDs
27. get_ledx -- LEDs
28. get_lval -- I/O Functions
29. get_ptime -- I/O Functions
30. get_pvar -- Persistent Variables
31. get_rtime -- Misc Functions and Constants
32. get_rumble -- Rumble
33. get_slot -- Misc Functions and Constants
34. get_val -- I/O Functions
35. if -- Flow Control
36. `init` -- Initialization
37. `int` -- Variables, Types and Arrays
38. `inv` -- Math Functions
39. `isqrt` -- Math Functions
40. `load_slot` -- Misc Functions and Constants
41. `main` -- A Simple Tutorial
42. `output_reconnection` -- Misc Functions and Constants
43. `pow` -- Math Functions
44. `ps4_authtimeout` -- Misc Functions and Constants
45. `ps4_touchpad` -- Misc Functions and Constants
46. `remap` -- Remapping
47. `reset_leds` -- LEDs
48. `reset_rumble` -- Rumble
49. `sensitivity` -- I/O Functions
50. `stickize` -- I/O Functions
51. `set_bit` -- Bit Operations
52. `set_bits` -- Bit Operations
53. `set_led` -- LEDs
54. `set_ledx` -- LEDs
55. `set_pvar` -- Persistent Variables
56. `set_rumble` -- Rumble
57. `set_val` -- I/O Functions
58. `swap` -- I/O Functions
59. `test_bit` -- Bit Operations
60. `turn_off` -- Misc Functions and Constants
61. `unmap` -- Remapping
62. `vm_tctrl` -- Misc Functions and Constants
63. `wait` -- Combos
64. `while` -- Flow Control
65. `wiir_offscreen` -- Misc Functions and Constants