
HP Prime Programming: An Introduction - Han - **12-23-2013 06:24 AM**

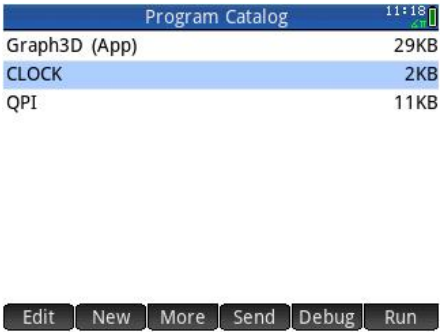
Last Update: JAN-24-2014 Fixed major error regarding "non-exported variables"

This introduction will assume that the user does all programming on the calculator. That said, any source file may be written completely on a computer and transferred to the calculator or emulator, completely bypassing the built-in *Program Editor*.

Part I: Editor, Compiler, Program Execution

1. Program Editor and Compiler

Press the [Shift] key followed by the [1] key to view the *Program Catalog*. At the very top will always the source file for the current app. Below the source file for the current app are the source files of user programs. To create a new program, press *New* from the menu at the bottom of the screen. You will then be presented with a new input screen titled "New Program" and be required to enter in the name of the "new program."



For this example, type in *MYPROG*. After pressing *OK* you will be presented with the *Program Editor* with a default template as shown below:
Code:

```
EXPORT MYPROG()
BEGIN

END;
```

To exit the *Program Editor*, simply press the [ESC] key. All changes are automatically saved. Additionally, the source code is parsed to check for syntax errors and, if no errors are found, compiled. If an error was found, the first error is displayed in a message box after exiting (and no compilation occurs). To validate the source code without exiting the *Program Editor*, press *Check* on the menu at the bottom of the screen. If an error is found, the cursor (and screen) will be moved to the location of the first error found. If no errors are found, a message box saying "No errors in the program" will be displayed. Moreover, the source code is also compiled (even while still inside the *Program Editor*).

When source code is compiled, a binary is created separate from the source code along with a pointer to the binary and a header consisting of exported variable/function names and argument information. At runtime, the header is used to enable access to exported functions or variables. Functions are called by their pointers. The header remains even after a reboot (warmstart) whereas the binary does not. If a function is called and its binary does not exist, then the binary will be compiled as needed without requiring user intervention.

Remark: The phrase "new program" as used by the *New* menu option in the *Program Catalog* is misleading. The name specified in the input screen is the name of the source file. It is used by the system to create a template *procedural function* (i.e. a program, similar to -- but different from -- a mathematical function). However, the procedural function need not have the same name as the source file. That is, the user is free to change *MYPROG()* to something completely different from the source file name *MYPROG*. Moreover, one may even create several procedural functions within the same source file. In fact, one may even use a single source file to create all the programs and variables that will ever be used on the calculator! So "new program" should really be "new project" (conceptually speaking).

2. A word about upper and lower case

One of the underlying principles in programming the HP Prime is that programs are essentially written as if commands will be executed in the Home view. One of the features of the Home view is that the command line parses commands without distinguishing upper and lower case. For example, *print("Hello");* is parsed as *PRINT("Hello");* -- even *pRiNt("Hello");* is parsed the same way. However, inconsistent upper/lower case usage in source code is bad practice. Moreover, there are commands which behave differently when typed in upper case vs. lower case (i.e. they are different commands). That said, since programs are essentially a sequence of commands executed in the Home view, one may type the source code completely in lower case (while accounting for the few instances in which *COMMAND()* has counterpart named *command()*). This feature enables program creation on a computer without having to have CAPS LOCK enabled.

3. Adding lines of code

In the code block above, we have a procedural function (i.e. a program) named *MYPROG* that does absolutely nothing. Rename the program *PROG1*, so that the source file *MYPROG* is distinct from the program contained within. Programs are simply sequences of commands separated by a semicolon (;) -- assuming default settings. Commands may be separated by carriage returns for legibility, though one is free to place all commands on a single line. Spaces and carriage returns are ignored by the compiler. That said, be careful as not all spaces are the same! The HP Prime uses Unicode, which means what appears as a space may not be the same character as an actual space! For this introduction, we will create a simple "hello world" program and then modify it to explore other programming options. A simple method for displaying a string is to use the *MSGBOX* command.
Code:

```
EXPORT PROG1()
BEGIN
  MSGBOX("Hello world!");
END;
```

Exit the *Program Editor* to compile the program.

4. Comments

Comments may be inserted using two forward slashes (basically, two "division" symbols). Comments are ignored by the compiler. We may add comments to either annotate the source code or to debug the program by temporarily "disabling" a particular line of code.
Code:

```
// This is an example of a "Hello world" in HP PPL
EXPORT PROG1()
BEGIN
  MSGBOX("Hello world!");
END;
```

5. Running the program.

There are two options for running a program. One is to use the *Run* menu option from the *Program Catalog*. Highlight the source name *MYPROG* and press *Run*. The other is to type the program's name into the command line. Press the [Home] key and type: *PROG1()* into the command line to see what the program does. A shortcut to actually typing out the program name is available through the toolbox key. Press the Toolbox key, and then select "User" from the menu. Selecting the desired source file and then program from the menu will place its name into the command line. **Notice the difference between the two methods (in particular, running *MYPROG* vs. running *PROG1*. We will touch upon their differences in Part II.)**

Part II: Arguments and Local Variables

1. Passing arguments

A procedural function (program) can be made to behave very much like a mathematical function in that arguments may be passed at runtime. Functional notation in mathematics takes the form \(f(x) \) where \(f\) is the name of the function and \(x \) is its argument (input). A procedural function is similar in that it also uses the *name(argument)* syntax. We will modify *PROG1()* so that it can take input at runtime. Suppose we wish to enable the user to type *PROG1("Name")* and have *PROG1* display "Hello Name!" instead of "Hello world!". The modified source code would look like:
Code:

```
EXPORT PROG1(name)
BEGIN
  MSGBOX("Hello " + name +"!");
END;
```

Now, if we run this program from the command line, we can call it with, say, *PROG1("Bob")* and the program will produce a message box that says: Hello Bob! The input "Bob" is temporarily stored in a local variable called *name*. The variable *name* is merely a dummy variable and its contents are discarded once *PROG1* ends. The HP Prime has the ability to concatenate two strings, and this operation is *overloaded* into the addition operation.

Currently, programs can have up to 16 arguments. Each argument variable name is separated by a comma -- assuming default settings. The argument variables belong to the class of local variables. The template for such a function is as follows:
Code:

```
EXPORT NameOfFunction(var1, var2, ..., var16)
BEGIN
  // source code here
END;
```

No checks are made on the type of variable for each input. In fact, we could have type *PROG1({1, 2})* to pass a list of two numbers, and the program would produce a message box showing: Hello {1,2}! This can be both useful and annoying. We can make use of overloaded operations such as addition to combine strings with non-string objects (among many other combinations). Thus *PROG1* is versatile in that it can actually handle many types of input with a single argument variable. However there is no built-in argument checking as a result, should we want to restrict the input type to just a string of characters.

Using a list is also a means of bypassing the 16 argument limit. Since lists may contain just about any type of object, we may even place all our arguments in a list and use only one argument variable name. The contents of the list are extracted using the syntax: *listname(position)*. That is, if *list* is the variable name of a list, then *list(2)* is a "name" for the second object within *list*.

Remark: When a program which takes arguments is executed from the *Run* menu option in the *Program Catalog*, then the argument variables are initialized to real numbers, and have a default value of 0. While this feature is nice in that it allows users to enter their arguments in a nice input form, the input form will only accept real-valued inputs and produce input errors otherwise. So while *PROG1* was designed to expect a string of characters as its input, executing it from the *Run* menu option prevents users from actually using a string as an argument!

2. Local variables

As mentioned above, input variables are a form of local variables. However, should our program need more variables that are distinct from those passed to the program, then they must be declared using the *LOCAL* declaration. While these declarations can be made anywhere within a program, most programmers tend to place all local variable declarations toward the top of the program block. Below are acceptable ways of declaring local variables for use within a program block.
Code:

```
LOCAL var1, var2; // creates two local variables named var1 and var2
LOCAL var1=1, var2="Charlie"; // creates two local variables and sets their initial values
LOCAL var1:=1, var2:="Charlie"; // same as above
```

In the current firmware, up to 8 local variables may be declared with a single *LOCAL* statement in which the variables (and optionally, their initial values) are separated by a comma.

We now modify our code so that *PROG1* will actually display some additional information stored in local variables.
Code:

```
EXPORT PROG1(name)
BEGIN
  LOCAL var1=1; // declare var1 as local and initialize to 1

  MSGBOX("Hello" + name + "!");

  // use either := or ▶ to assign values to a variable
  var1:=var1 + 1; // var1 now has value 2
  var1 * 2 ▶ var1; // var1 now has value 4

  // here is an example of declaring a local variable "as needed"
  // rather than placing it up at the top like var1
  LOCAL var2:="Reminder #" + var1 + ": Alice called today.";

  MSGBOX(var2);
END;
```

3. Scope of local variables

When local variables are created by declaring them either as arguments of a program, or within a program block, then their scope is limited to that program only. In the example immediately above, the local variables *var1* and *var2* are only available to commands that are both within the definition of the program *MYPROG1* and which come AFTER the declaration of each variable (e.g. the *MSGBOX()* command).

It is possible to create a local variable whose scope is much wider -- covering the entire source file. In a previous edition, these local variables were mislabeled as "non-exported global variables." They have been confirmed to be local variables that are declared using a shorthand declaration style. But to get to the point, a local variable that is declared OUTSIDE of any program block is considered to be "global" to all programs within the source file as any programs defined after such a variable declaration may use that variable. However, it is local not only because it is normally declared with *LOCAL*, but that its scope is limited to within the source file. The following examples show how this type of local variable can be declared and used:

Code:

```
LOCAL var1, var2, var3; // declares var1, var2, and var3 as local to this source file; can be used by all programs defined below this line of code
var4, var5, var6; // essentially the same as above; i.e., LOCAL is optional when declaring local variables for use within the entire source file

EXPORT MAINPRG()
BEGIN
  var1:=var2+var3;
  var4:=1;
END;

SUBROUTINE()
BEGIN
  LOCAL var7; // this local variable can only be used within SUBROUTINE()
  var7:=var4+var5-var1;
END;
```

The local variables *var1* through *var6* may be used by any program defined later in the source file. Unlike local variables declared inside a program block, such as *var7*, local variables declared outside of any program block **retain their values even after program execution reaches completion and up until the source is recompiled**. This is the only effective difference. They are still not accessible to programs defined in a separate source file, or the user.

Part III: Global Variables and Calling Programs within a Program

For this part, we will start with a completely new source file and create something useful, as opposed the the "Hello world" type of program in the previous parts. Begin by creating a new source file named *PROJECT*. Erase the default template that is created, and create the following program:
Code:

```
EXPORT ANG(a)
BEGIN
  a:=a/PI*180;
  RETURN(a);
END;
```

1. Returning a value

The program above simply takes a numerical value, presumably representing an angle in radians, and returns the angle as if measured in degrees. The code could be further shortened to simply have *RETURN(a/PI*180)* in between the BEGIN/END pair. The main concept here is the *RETURN* command. This command is used to end program execution as well as return a value back to the user (i.e. placed on the history in the Home screen). Even programs which do not have an explicit *RETURN* statement will still return a value -- namely the value of the very last command executed prior to the program reaching its end. Thus, we can accomplish the same goal of returning the value of a the angle in degrees to the Home screen by omitting the *RETURN* command!

In the "Hello world" example, we simply displayed a message on the screen. After the program ended, a 1 appeared on the history in the Home view. This is what was returned by the *MSGBOX* command. However, if we had run the program from the *Run* menu option, then after pressing *OK* another dialogue box shows up with the message *PROG1 1*. Again, the value after *PROG1* what was returned by the *MSGBOX* command.

The *RETURN* command serves one additional purpose: returning program execution back to any calling programs. This will be explained later.

2. Calling a program within a program

Now suppose that we want to create another program that needs to do angle conversion (from radians to degrees). In the example that follows, we will create a program that takes two angles (assumed to be in radians) of a triangle, and return the third angle in degrees. Within the same source file *PROJECT* we can add a second program so that the source file looks like the following:
Code:

```
// program declarations
THIRDANG();
ANG();


// THIRDANG(a, b)
// Takes two angles a and b (in radians) of a triangle and
// returns the third angle (in degrees)
EXPORT THIRDANG(a,b)
BEGIN
  LOCAL angle1, angle2;

  angle1:=ANG(a);
  angle2:=ANG(b);
  RETURN(180-angle1-angle2);
END;


// ANG(a)
// Takes an angle a (in radians) and returns the same angle in degrees
EXPORT ANG(a)
BEGIN
  a:=a/PI*180;
  RETURN(a);
END;
```

Below is an explanation of the source code above:

1. Within the single source file name *PROJECT* we have created two programs: *THIRDANG* and *ANG*. These become new "commands" which we can use in the Home view command line.
2. At the top of the file are the program declarations. These are only needed if we plan to use the programs listed there as subroutines. In this example, *ANG()* is called (as a subroutine) from within *THIRDANG()* and therefore must be declared "in front of" (i.e. above, in terms of source code location) the source code for the *THIRDANG()* program. We went ahead and added *THIRDANG()* up at the top as well. However, as it is not called by any other

program, its inclusion is unnecessary. (We did this because we may actually need *THIRDANG()* as a subroutine later on, and because it is a good way to track all the programs created within this source file.) In general, subroutines must be declared before they are called within a program. Optionally, we could have placed the source code of *ANG()* above the source code of *THIRDANG()* to avoid have to declare any subroutines whatsoever (as defining a program is equivalent to declaring it for use). However, this is not good a programming technique because there will be cases when two programs call pass program execution back and forth, and therefore a declaration statement must be made for at least one of them. I personally suggest placing all such declarations at the top of the source file so that one never has to deal with the ordering of the actual source code for each program.
3. Subroutine declarations require the parenthesis after the name of the program. If the parentheses are left out, then the names will be interpreted as global variables! (More on those in a bit.) However, the arguments of those programs are excluded. That is, we declared *ANG()* for use as a subroutine, but leave out its arguments until we actually define the program via source code.
4. The value returned by *ANG()* can be saved in the program calling it. For example, *angle1:=ANG(angle1);*.
5. For most types of variables, self-referencing is not an issue. That is, in a statement such as *angle1:=angle1-1;* the right hand side is evaluated first before overwriting the value stored in *angle1* with the new value.
6. The return statement in *THIRDANG()* written as is allows us to avoid having to use another local variable.
7. When selecting *Run* from the *Program Catalog*, users are now presented with a choice of whether to run *THIRDANG* or *ANG*.

To use the *THIRDANG* program, simply type: *THIRDANG(PI/3, PI/2);* and the result is 30.

3. Global variables

If you have not already read up on the various types of variables, it is recommended that you at least skim through this article about variable types and their priorities. In this section, we will only cover how to create global variables and their scope.

In this current project, we have two programs which each use local variables for temporary data storage. We can replace the local variables with global variables so that the two programs can make use of the same variable without have to pass any sort of arguments from one to the other. For example, consider the following modification to the code above:
Code:

```
// program declarations
THIRDANG();
ANG();

// global variables
EXPORT angle1, angle2;

// THIRDANG(a, b)
// Takes two angles a and b (in radians) of a triangle and
// returns the third angle (in degrees)
EXPORT THIRDANG(a,b)
BEGIN
  angle1:=a;
  angle2:=b;
  ANG();
  RETURN(180-angle1-angle2);
END;

// ANG()
// Converts angles stored in global variables angle1 and angle2 to degrees
EXPORT ANG()
BEGIN
  angle1:=angle1/PI*180;
  angle2:=angle2/PI*180;
END;
```

Some brief comments about this modified code:

1. Global variables CANNOT be defined/declared inside a program block!
2. Local variables are no longer used except for those passed as arguments to *THIRDANG*.
3. Global variables *angle1* and *angle2* are declared in the same fashion as subroutines (previously explained).
4. *ANG()* no longer requires any arguments when called.
5. The global variables now appear in the *Memory Browser* and behave no differently from built-in global variables A through Z. That is, a user may type: *angle1:=2;* at the command line to change the value of *angle1*. Notice that there is no prompt to "create" *angle1* as a new variable.
6. The global variables *angle1*, *angle2*, and the programs *THIRDANGLE()* and *ANG()* may be used as subroutines in programs defined in a different source file.

4. Exporting vs. not exporting variables and programs

The *EXPORT* command that appears in front of global variables and program definitions in the code above make the respective variables and programs visible to the entire system: the user, other programs, apps, etc. Sometimes it is necessary to "hide" such variables and programs from the user (or the *Memory Browser* to prevent clutter) so that their values are not altered, or to prevent other programs from other source files from tampering with the values or calling the programs.

By removing the *EXPORT* command, all programs become invisible to the user, and to other programs from a different source file. And what was once a global variable becomes a local variable. Consider the following slight modification (the only changes are the removal of *EXPORT* from the global variable declaration section and from *ANG()*):
Code:

```
// program declarations
THIRDANG();
ANG();

// global variables are now local
angle1, angle2;

// THIRDANG(a, b)
// Takes two angles a and b (in radians) of a triangle and
// returns the third angle (in degrees)
EXPORT THIRDANG(a,b)
BEGIN
  angle1:=a;
  angle2:=b;
  ANG();
  RETURN(180-angle1-angle2);
END;

// ANG()
// Converts angles stored in global variables angle1 and angle2 to degrees
ANG()
BEGIN
```

```
  angle1:=angle1/PI*180;
  angle2:=angle2/PI*180;
END;
```
The effects are:

1. *ANG()* is no longer accessible to the user, and does not appear as an option to *Run* in the *Program Catalog*.
2. The global variables *angle1* and *angle2* are no longer visible in the *Memory Browser*. Users attempting to access these variables via the command line will be given error messages. They have now become local variables are accessible by programs within the same source file, from the point of declaration of those variables to the end of the source file.
3. The variables *angle1* and *angle2* are no longer accessible by other programs defined in other source files.
4. The program *ANG()* is can no longer be called by programs from other source files. However, *ANG()* may still be used as a subroutine in all programs created in the same source file in which *ANG()* is defined (i.e. within our *PROJECT* source file).

Remark: The scope of exported global variables is system-wide. Just like the built-in system variables L0 through L9 may be used for list operations from anywhere in the system, exported variables may also be accessed from anywhere in the system. The scope of non-exported (local) variables is limited to the source file.

Part IV: Block statements and control structures

This is the final part to this introduction. We will explore some basic control structures below.

1. IF THEN END; and IF THEN ELSE END;

If a particular command should only be executed if a certain condition is met, then the *IF THEN END* structure is a simple way to execute a sequence of commands only if the condition(s) are met. The syntax is
Code:

```
IF expression THEN
  // here is where we place commands which are executed if 'expression' is true
END;
```
A very simple but useful example of such a control block is shown below. Suppose we wish to solve a simple problem: determine the angle of a right triangle whose two shorter legs are a and b. Further suppose that we always want the result in degrees. Mathematically, the solution is simply $\arctan(\frac{b}{a})$. The issue is that the value of the $\arctan(x)$ function depends on the angle mode. The *ATAN* function returns a number -- and the interpretation of that number is based on the user's angle settings. The system keeps track of this setting via the *HAngle* global variable. Some programmers may prefer to code in "radian mode" in the sense that all angles are assumed to be in radians even though the system itself may be set to degree mode. This example demonstrates how to create code that is independent of a user's settings, such as the angle mode without altering the user's settings.
Code:

```
ANG();

// FINDANG(a,b)
// determines the angle of a right triangle whose leg opposite of the angle
// is b and whose leg adjacent to the angle is a; result is always in degrees
EXPORT FINDANG(a,b)
  a:=ATAN(b/a); // re-use local variable a
  RETURN(ANG(a));
END;

ANG(a)
BEGIN

  // if angle mode is radian, then the value of HAngle is 0
  // otherwise HAngle is 1 for degree mode
  IF HAngle==0 THEN
    a:=a/PI*180; // the value of a is interpreted to be in radians; convert to degrees
  END;
  RETURN(a);
END;
```
In the code above, the *ANG()* program has been redefined to use an IF THEN END statement that converts an angle only if the system angle mode is set to radian. Otherwise, *ANG()* does nothing. Do note that the local variable *a* for *ANG()* is distinct from the local variable *a* used in *FINDANG()*.

Example: Angle mode independence. The really nice thing about this simple little program *ANG()* is that with a tiny modification, we can now create any program that always use angles in radians by having *ANG()* handle any conversions for us based on the user's settings. Consider *ANG()* modified to be:
Code:

```
ANG(a)
BEGIN
  // notice the slight change from HAngle==0 to simply HAngle (equivalent to HAngle==1)
  IF HAngle THEN
    a:=a/PI*180; // the value of a is interpreted to be in radians; convert to degrees
  END;
  RETURN(a);
END;
```
Now, we can use: *COS(ANG(PI/6))* in our code (perhaps because we prefer to use values associated with radian measure), and always expect the result to be 1/2 regardless of the angle mode. The alternative would be to set the user's angle mode to radians and then reset the mode after the program finishes -- though this can easily be broken.

Remark: *HAngle* (home angle) is a global variable which reflects the system angle mode. If its value is 0, then the angle mode is radian. And if 1, then the angle mode is degree. There is yet another system variable which tracks the angle mode: *AAngle* (app angle). If *AAngle* is 0, then the app angle mode defaults to *HAngle*. If *AAngle* is 1, then the app angle mode is in radian, and 2 for degree. The app angle has higher priority than the system angle since an app is always in view.

The second variation of conditional branching is
Code:

```
IF expression THEN
  // code for when 'expression' is true (non-zero)
ELSE
  // code for when 'expression' is false (zero)
END;
```
The expression in either form of an "IF THEN" branch need not be a boolean test. All that matters is whether the expression is non-zero (true) or zero (false). If the expression is itself a command or a call to another function, then the command/call should not end in a semicolon. For example,
Code:

```
// notice the lack of the semicolon ";" after MYPROG1()
```

```
IF MYPROG1() THEN
  // code for which MYPROG1 returns a non-zero value
END;
```

2. CASE IF THEN END; END;

If a program consists of several branches, of which only one will be executed based on certain conditions, then the CASE branch should be used. The syntax is
Code:

```
CASE
  IF expression1 THEN
    // code if expression1 is true
    // then resume execution at the end of the CASE structure
  END;
  IF expression2 THEN
    // code if expression2 is true
    // then resume execution at the end of the CASE structure
  END;

    // ...

  IF expressionN THEN
    // code if expressionN is true
    // then resume execution at the end of the CASE structure
  END;

  DEFAULT
    // code for when all previous cases fail
    // the "DEFAULT" keyword is optional as well as any commands which follow it
END;
```

Remember that a CASE branch behaves much like a piecewise function in that only one of the embedded IF THEN END conditional branches will ever be executed. On the other hand, having a sequence of multiple IF THEN END statements (without using CASE) would result in each individual IF THEN END branch being tested.

Example: Basic Key/Touchscreen Handler. Below is a skeleton for a basic keyboard and touchscreen handler using the *WAIT(-1)* command. By passing -1 as the argument, *WAIT* will essentially pause the program and wait for input from either the keyboard or the touchscreen. If no input is received after a minute, then -1 is returned. Otherwise, either the key number (for a keypress) or a "mouse" list (for touchscreen gestures) will be returned. Due to the different types of objects (integer vs list) returned by this command, and the fact that mis-matched object types in a branch (such as an *IF THEN END* statement) will cause run-time errors, we must carefully parse the value(s) returned by *WAIT*. (Note that the touchscreen events MUST be parsed first in this implementation and that only part of the handler is shown; the necessary subroutines would need to be added accordingly.)
Code:

```
EXPORT KMINPUT()
BEGIN
  LOCAL run:=1, key;

  // change run to 0 to exit the while loop
  key:=WAIT(-1);
  CASE
    IF TYPE(key)==6 THEN
      // we have touchscreen events
      CASE
        IF key(1)==0 THEN kmMouseDown(); END;
        IF key(1)==1 THEN kmMouseUp(); END;
        // ... more mouse events
        IF key(1)==7 THEN kmMouseLongClick(); END;

        kmMouseEvent(); // default handler
      END;
    END; // end of touch events

    // at this point, all touch events were handled due to the
    // TYPE(key)==6 check; so only key-presses are left
    // and if a touch event occurred, these tests below are
    // never reached

    IF key==0 THEN kmDoAppsKey(); END;
    // ... more key definitions here
    IF key==50 THEN kmDoPlusKey(); END;

    kmOtherEvents(); // handle all remaining undefined keys
  END;

END;
```

Remark: The way *WAIT(-1)* processes a mouse event is not the way one might expect. That is, a mouse click generates three separate events, so that three consecutive instances of *WAIT(-1)* are required to finally capture the mouse click. So when a mouse click occurs, the first instance of *WAIT(-1)* returns a mouse-down event, the second instance of *WAIT(-1)* returns a mouse-up event, and finally the third instance of *WAIT(-1)* returns a mouse-click. The same goes for long clicks and other types of events. Please keep this in mind when creating a mouse/keyboard handler.
