Top Level Topics

Creating a Robot

Robots are created using a simple scripting language. All you need is an ASCII text editor and an imagination. Although helpful, no previous programming knowledge is required. In fact, Robot Battle is a great way to start learning how to program. Most modern computer systems are based on event driven techniques very similar to those you will learn playing Robot Battle.

To create your own robot, start by looking at the Sample Robot. Do not worry about the specific commands, they are described in Scripting Language help. Just get the feel for the layout of a robot's instructions.

Once you have looked over the Sample Robot, move on to the Scripting Language help. It provides detailed explanations of all robot operations. Again, do not get too hung up on the specifics. The best way to learn is by examining prefabricated robots and actually playing the game. Robot Battle comes with many thoroughly commented sample robots. Use a text editor to view their instructions, then start the game and watch them slug it out in the arena!

Scripting Language Reference

Robots are created using a simple scripting language. Robot scripts tell robots what they should do in various situations. Robot scripts contain several major components. The list below briefly describes each of these components. Click on an entry to get more detailed information about the component.

Robot Functions	Commands that tell a robot what is should do
System Variables	Variables that describe a robot's state during game play
User Defined Variables	Variables that are defined by the creator of a robot
Special Sections	Sections that handle events without being registered
Math Operators	Mathematical operators
Logic Operators	Logical operators
Language Semantic	Technical description of scripting language components

Sample Robot

Notice that capitalization is not important. A variable named "VARIABLE" will be that same as "variable" or "Variable". This is also true for robot functions and section names.

```
Init                                # Init section
{
      Name( "Sample" )              # Sets robot's name

      RegCore( Core )               # Registers core event handler
      RegCldMissile( MissileHit,1 ) # Registers missile hit handler
      RegDtcRobot( FoundRobot, 2 )  # Registers robot detection handler

      fire_engy = 1                 # User defined variable: fire_engy
      LockGun( ON )                 # Locks gun and radar together
}

Core                                # Core section
{
      Scan( )                       # Looks for other objects
      RadarRight( 5 )               # Turns radar (and gun) right
}

MissileHit                          # MissileHit section
{
      Ahead( 20 )                   # Moves robot ahead
}

FoundRobot                          # FoundRobot section
{
      fire( fire_engy )             # Fires energy missile
      Scan()                        # Looks for other objects
}
```

Scripting Language User Variables

User defined variables are the primary means of storing information about a robot. As the designer of a robot, you decide how many user variables you need and what their names should be. All variables are global. Global means that a variable can be accessed and changed from anywhere in a script file.

User variables must start with a letter (A-Z) or an underscore (_). Variables may contain numbers (1-9), but they may not start with a number. Any name may be used for variables except section names. In other words, **round** is a valid variable name, but **init** is not.

User variables are automatically defined by placing them on the left side of the assignment command (=). This may be done anywhere in a robot's script. At the start of a game, user defined variables are initialized to zero. The only exception to this rule are variables stored with a previous call to the Store function. The following line defines a user variable called `fire_engy`:

```
fire_engy = 1
```

Assuming Store has not been called for `fire_engy` in a previous game, the variable `fire_engy` is created and assigned a value of **zero** at the start of a game. Not until the line above is actually executed will the variable contain a value of **one**. If this line is never reached during a game, `fire_engy` will always contain a value of **zero**.

Variables that were stored in a previous game with the Store function do not default to values of zero. When these variables are created at the start of a game, they are assigned the values they contained when Store was last called for each variable. This allows variables to be passed from one game to the next in a match.

Variables may contain values in the range of ± 3.4e ± 38 with a precision of 6 digits. As with most computer languages, floating point (real number) math is not perfectly accurate. Testing for equality after performing calculations may produce unexpected results. For example, **acos(cos(20))** may yield 19.9999 instead of 20. Use the Round function if this problem arises.

Example:

```
test1 = variable
variable = -10.5
test2 = variable
print( test1 )
print( test2 )
```

Assuming Store has not been called, what are the values of `test1` and `test2`? When the game starts, `test1`, `variable`, and `test2` are all created and assigned values of zero. Line 1 therefore assigns `test1` to a value of zero (which it already was). Line 2 changes `variable` from a value of 0 to a value of -10.5. Line 3 then changes `test2` from a value of 0 to a value of -10.5.

Thus, the Print functions display:

0
-10.5

Scripting Language Semantics

The following is a technical description of the Robot Battle scripting language. If this stuff does not make sense, just ignore it. The Robot Battle scripting language is fairly intuitive. This definition has been provided primarily for the computer science types among us.

Robot Battle scripts are composed of many lines. Each line contains one and only one statement. In Robot Battle, a function or command is a statement. Robot Battle statements do not return values. A statement may contain zero, one, or many expressions. An expression always evaluates to a value. An expression alone does not constitute a statement. This implies that an isolated expression is an error since every evaluated line of a robot script must contain a statement. Expressions are composed of one or many values and operators.

As with other computer languages, operators are used to provide notational convenience. This is particularly true in Robot Battle since statements do not evaluate to values and may not be nested. Without operators, compound expressions would not be possible. For example, the + operator is a notational convenience for building the expression `'a + b'` in this statement:

```
Ahead( b + c )
```

If operators were not provided, this line would have to be broken into multiple statements:

```
Add( b, c )
Ahead( result )
```

The only statement in robot battle that is not obvious is the assignment (≡) statement. The assignment statement may be thought of as a function that takes an expression and an lvalue. In Robot Battle, lvalues are simply user defined variables.

This statement:

```
a = b + c
```

May be thought of like this:

```
Assign( a, b + c )
```

Where `'a'` is an lvalue and `'b + c'` is a compound expression.

Scripting Language Functions

These functions make up the robot scripting language. They are used to tell a robot what it should do. Parameters are the values that are passed into a function. Different functions take different numbers of parameters. No robot functions return values. The functions below are arranged alphabetically . Remember, capitalization is used for clarity only, Robot Battle does not recognized capitalization.

=	Assigns a value to a user defined variable
Abs	Calculates an absolute value
Ahead	Moves the robot ahead
AscanEvents	Turns on or off auto scanning events
Back	Moves the robot back
Blocking	Turns command blocking on or off
BodyLeft	Turns the robot to the left
BodyRight	Turns the robot to the right
CldCookieEvents	Turns on or off cookie collision events
CldMineEvents	Turns on or off mine collision events
CldMissileEvents	Turns on or off missile collision events
CldRobotEvents	Turns on or off robot collision events
Continue	Continues previously aborted movement
CoreEvents	Turns on or off core events
CustomEvents	Turns on or off custom events
DtcCookieEvents	Turns on or off cookie detection events
DtcMineEvents	Turns on or off mine detection events
DtcRobotEvents	Turns on or off robot detection events
Else	Evaluated when previous the If() or ElseIf() is false
ElseIf	Evaluated when previous the If() is false
Endif	Marks the end of a logical the If() block
Fire	Fires an energy missile
GetHitsOther	Determines the number of times the robot has hit another robot
GetHitsSelf	Determines the number of times the robot has been hit by energy missiles
GetHitStr	Determines the average damage done by the robot's missiles
GetOthers	Counts the number of other robots left in a game
GetRandom	Generates a random number
GetShots	Determines the number of energy missiles fired by the robot
GetTurns	Determines the number of turns the robot has had
Gosub	Causes execution to continue in another section
GunLeft	Turns the robots gun to the left
GunRight	Turns the robots gun to the right
If	Starts a logical If block
LockAll	Turns rotation locking on for all robot components
LockGun	Turns rotation locking on for the robot's gun and radar
Name	Sets the robot's name
Max	Determines the smaller of two values
Min	Determines the larger of two values
Print	Adds a string to the output display window
Print	Adds a variable to the output display window
RadarLeft	Turns the robot's radar to the left
RadarRight	Turns the robot's radar to the right
RegAscan	Registers an event handler for auto scanning

RegCldCookie	Registers an event handler for collision with energy cookies
RegCldMine	Registers an event handler for collision with energy mines
RegCldMissile	Registers an event handler for collision with energy missiles
RegCldRobot	Registers an event handler for collision with other robots
RegCore	Registers an event handler for the robot's core behavior
RegCustom	Registers an event handler for custom defined events
RegDtcCookie	Registers an event handler for detection of energy cookies
RegDtcMine	Registers an event handler for detection of energy mines
RegDtcRobot	Registers an event handler for detection of other robots
Return	Causes current section to end at the current line
Round	Rounds the specified value
Scan	Sends out a radar ping to search for other objects
SetAccel	Sent the robots lateral acceleration
Stall	Causes robot to freeze
Stop	Causes robot to abort further movement
Store	Stores values for retrieval in later games
SyncAll	Aligns the robot's body and gun to its radar
SyncGun	Aligns the robot's gun to its radar
Truncate	Truncates the specified value
WaitFor	Creates a user defined block

Any Valid Expression

This parameter may be any valid expression. Expressions are composed of variables, numeric values, math operators, and logical operators

lvalue = rvalue

lvalue - User declared variable
rvalue - Numerical value to be copied EXP

The assignment command does not follow the standard Robot Battle command syntax. The non-standard format is more natural and matches the syntax of other computer languages.

The assignment command copies a value into a user defined variable. This is the only way to change the value of a user variable. Variables are automatically defined by placing them on the left side of the assignment command. All variables have an initial value of *zero* until explicitly assigned a different value.

RegCore(*section*)

section - Name of a section in the robot script

Registers an event handler for the robot's core behavior. Core events occur when no other events are happening. In other words, this section is called repeatedly until the robot dies. The core section may be re-registered at any time during a game to change the robot's core behavior. All other registered events have higher priorities that the core event.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the CoreEvents command to deactivate the event handler.

See Also:
Events Description

RegAscan(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for auto scanning. Auto scanning events occurs only when a robot is moving. Auto scanning provides robots an opportunity to continue searching for other objects while moving. When an auto scan event handler is registered, it will be called repeatedly while the robot is moving and no higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called. This applies to lateral movement only, not rotation. Both the Ahead and Back functions have no meaning in a section handling auto scan events.

Auto scan events are triggered by the moving variable. This variable is always *true* while a robot is moving laterally and *false* while it is stationary or only rotating.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the AscanEvents command to deactivate the event handler.

See Also:
Events Description

RegCldRobot(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for collisions with other robots. The section specified above will be called whenever the robot runs into another robot and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called. Hitting another robot will result in an energy loss of 1 point to each robot.

Robot collision events are triggered by the cldrobot variable. When a robot collision event handler returns, the cldrobot variable is automatically set to false causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the CldRobotEvents command to deactivate the event handler.

See Also:
Events Description

RegCldMissile(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)_
EXP

Registers an event handler for collisions with energy missiles fired by other robots. The section specified above will be called whenever the robot is hit by an energy missile and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called. The amount of damage done by an energy missile depends upon both the amount of energy put into it and the distance it has traveled.

Missile collision events are triggered by the cldmissile variable. When a missile collision event handler returns, the cldmissile variable is automatically set to false causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the CldMissileEvents command to deactivate the event handler.

See Also:
Events Description

RegCldCookie(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for collisions with energy cookies. The section specified above will be called whenever the robot runs into an energy cookie and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called. Hitting an energy cookie will result in an energy *gain* of 20 point.

Cookie collision events are triggered by the cldcookie variable. When a cookie collision event handler returns, the cldcookie variable is automatically set to false causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the CldCookieEvents command to deactivate the event handler.

See Also:
Events Description

RegCldMine(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for collisions with energy mines. The section specified above will be called whenever the robot runs into an energy mine and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called. Hitting an energy mine will result in an energy *loss* of 20 point.

Mine collision events are triggered by the cldmine variable. When a mine collision event handler returns, the cldmine variable is automatically set to false causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the CldMineEvents command to deactivate the event handler.

See Also:
Events Description

RegDtcRobot(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for detection of another robot. The section specified above will be called whenever another robot is detected by a call to Scan and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called.

Robot detection events are triggered by the dtcrobot variable. When a robot detection event handler returns, the dtcrobot variable is automatically decremented by one potentially causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the DtcRobotEvents command to deactivate the event handler.

See Also:
Events Description

RegDtcCookie(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for detection of energy cookies. The section specified above will be called whenever an energy cookie is detected by a call to Scan and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called.

Cookie detection events are triggered by the dtccookie variable. When a cookie detection event handler returns, the dtccookie variable is automatically decremented by one potentially causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the DtcCookieEvents command to deactivate the event handler.

See Also:
Events Description

RegDtcMine(*section, priority*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority)
EXP

Registers an event handler for detection of energy mines. The section specified above will be called whenever an energy mine is detected by a call to Scan and no other higher priority events are occurring. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called.

Mine detection events are triggered by the dtcmine variable. When a mine detection event handler returns, the dtcmine variable is automatically decremented by one potentially causing the event to end.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the DtcMineEvents command to deactivate the event handler.

See Also:
Events Description

RegCustom(*section, priority, expression*)

section - Name of a section in the robot script
priority - Importance of event relative to others (lower numbers have higher priority) EXP
expression - Expression that evaluates to True (non-zero) or False (zero) EXP

Registers an event handler for a custom defined event. The custom event occurs whenever the provided expression evaluates to *true* and no other higher priority events are occurring. The expression may be composed of any legal variables, math operators, or logical statements. Any expression that is legal inside an If statement may also be used as a custom event. The priority value should be a whole number, decimals will be dropped. If two events registered with the same priority occur at the same time, it is unspecified which event handler will be called.

Each section may only have one custom event attached to it. There may be any combination of standard events, but only one custom event per section. When two custom events need to use the same section, the events may be combined into one with an OR statement. Alternatively, two small helper sections could be created that both use Gosub calls to share the same logic. When multiple custom events are registered to one section, only the *last* one will apply.

Unlike "standard" events, custom events are not ended automatically. For example, when a section registered to handle collision events returns, the collision variable is reset to false ending the event. When a custom event handler returns, is has no effect on the state of the custom event. If events are not ended somehow, the handler section will execute continuously.

Note: When an event handler is registered or re-registered, it becomes immediately active. Use the CustomEvents command to deactivate the event handler.

See Also:
Events Description

CoreEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of core events (core events occur when no other events are occurring). This function does not effect which section handles core events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegCore.

See Also:
Events Description

AscanEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of auto scan events. This function does not effect which section handles auto scan events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegAscan.

See Also:
Events Description

CldRobotEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of robot collision events. This function does not effect which section handles robot collision events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegCldRobot.

See Also:
Events Description

CldMissileEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of missile collision events. This function does not effect which section handles missile collision events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegCldMissile.

See Also:
Events Description

CldCookieEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of energy cookie collision events. This function does not effect which section handles cookie collision events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegCldCookie.

See Also:
Events Description

CldMineEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of energy mine collision events. This function does not effect which section handles mine collision events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegCldMine.

See Also:
Events Description

DtcRobotEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of robot detection events. This function does not effect which section handles robot detection events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegDtcRobot.

See Also:
Events Description

DtcCookieEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of energy cookie detection events. This function does not effect which section handles cookie detection events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegDtcCookie.

See Also:
Events Description

DtcMineEvents(*bool*)

bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of energy mine detection events. This function does not effect which section handles mine detection events, only whether the events are handled or ignored. The event handler section may be changed by another call to RegDtcMine.

See Also:
Events Description

CustomEvents(*section*, *bool*)

section - Name a section in the robot script
bool - True (non-zero) or False (zero) value EXP

Used to either turn on or off handling of a specific custom event. Since there may be many registered custom events, the specific event must be identified by its handler section. This function does not effect which section handles the custom event, only whether the event is handled or ignored. Custom events may not really be re-registered. To move a custom event to a different handler, turn off the custom event using this function (or register a new custom event to the section) then call RegCustom to register a different handler.

See Also:
Events Description

SetAccel(*accel*)

accel - Acceleration value EXP

Sets the robot's lateral acceleration to a value between 1 and 5. While moving, robots are constantly accelerating. Therefore, this value approximately represents a robot's speed. This function changes the accel variable described below. If this function is never called, acceleration defaults to 3.

Ahead(*dist*)

dist - Distance to move EXP

Moves the robot ahead the specified amount. If the amount is negative, the robot will move backward. Running into another robot will cause damage to both robots in the collision. Each robot will lose one energy point per collision. Hitting a wall will stop a robot, but causes no damage.

Note: The playing arena is a square measuring 400 unit in both directions while robots measure 33 units in both directions. **Ahead** requires multiple turns to complete, therefore causing command blocking.

See Also:
Back

Back(*dist*)

dist - Distance to move EXP

Moves the robot back the specified amount. If the amount is negative, the robot will move forward. Running into another robot will cause damage to both robots in the collision. Each robot will lose one energy point per collision. Hitting a wall will stop a robot, but causes no damage.

Note: The playing arena is a square measuring 400 unit in both directions while robots measure 33 units in both directions. **Back** requires multiple turns to complete, therefore causing command blocking.

See Also:
Ahead

Stop( )

Causes the robot to abort further movement. This includes both lateral and rotational movement. This function is useful during an event handling routine. When a new event occurs, all movement will continue unless **Stop** or a new movement function is called.

This function stores both the incomplete lateral movement and rotations from the aborted movement in a continue buffer. This continue buffer is used by the Continue function.

Note: If **Stop** is called when no motion is occurring, the continue buffer is left unchanged. Each time **Stop** aborts movement, however, the previous continue buffer is overwritten.

Continue( )

Continues all movement previously aborted by a call to Stop. This includes both lateral movement and rotations. Calling **Continue** also resets the continue buffer.

This function only continues aborted movement, it does not restore location. For example, if a robot rotates or moves laterally between calls to Stop and **Continue**, movement will be continued from the new location and orientation.

Note: Just like other commands that cause movement, **Continue** requires multiple turns to complete, causing command blocking.

BodyLeft(*degrees*)

degrees - Degrees to rotate body EXP

Turns the robot's body counter-clockwise by the amount specified. Negative values will cause clockwise rotation. The maximum rotation rate of a robot's body is 5 degrees per turn.

Note: Rotation speeds of a robot's body, gun, and radar differ. A robot's body rotates the slowest, its gun rotates twice as fast as its body, and its radar rotates three times as fast as its body. **BodyLeft** requires multiple turns to complete, therefore causing command blocking.

See Also:
BodyRight

BodyRight(*degrees*)

degrees - Degrees to rotate body EXP

Turns the robot's body clockwise by the amount specified. Negative values will cause counter-clockwise rotation. The maximum rotation rate of a robot's body is 5 degrees per turn.

Note: Rotation speeds of a robot's body, gun, and radar differ. A robot's body rotates the slowest, its gun rotates twice as fast as its body, and its radar rotates three times as fast as its body. **BodyRight** requires multiple turns to complete, therefore causing command blocking.

See Also:
BodyLeft

GunLeft(*degrees*)

degrees - Degrees to rotate gun EXP

Turns the robot's gun counter-clockwise by the amount specified. Negative values will cause clockwise rotation. The maximum rotation rate of a robot's gun is 10 degrees per turn.

Note: Rotation speeds of a robot's body, gun, and radar differ. A robot's body rotates the slowest, its gun rotates twice as fast as its body, and its radar rotates three times as fast as its body. **GunLeft** requires multiple turns to complete, therefore causing command blocking.

See Also:
GunRight

GunRight(*degrees*)

degrees - Degrees to rotate gun EXP

Turns the robot's gun clockwise by the amount specified. Negative values will cause counter-clockwise rotation. The maximum rotation rate of a robot's gun is 10 degrees per turn.

Note: Rotation speeds of a robot's body, gun, and radar differ. A robot's body rotates the slowest, its gun rotates twice as fast as its body, and its radar rotates three times as fast as its body. **GunRight** requires multiple turns to complete, therefore causing command blocking.

See Also:
GunLeft

RadarLeft(*degrees*)

degrees - Degrees to rotate radar EXP

Turns the robot's radar counter-clockwise by the amount specified. Negative values will cause clockwise rotation. The maximum rotation rate of a robot's radar is 15 degrees per turn.

Note: Rotation speeds of a robot's body, gun, and radar differ. A robot's body rotates the slowest, its gun rotates twice as fast as its body, and its radar rotates three times as fast as its body. **RadarLeft** requires multiple turns to complete, therefore causing command blocking.

See Also:
RadarRight

RadarRight(*degrees*)

degrees - Degrees to rotate radar EXP

Turns the robot's radar clockwise by the amount specified. Negative values will cause counter-clockwise rotation. The maximum rotation rate of a robot's radar is 15 degrees per turn.

Note: Rotation speeds of a robot's body, gun, and radar differ. A robot's body rotates the slowest, its gun rotates twice as fast as its body, and its radar rotates three times as fast as its body. **RadarRight** requires multiple turns to complete, therefore causing command blocking.

See Also:
RadarLeft

LockAll(*bool*)

bool - True (non-zero) or False (zero) value EXP

Turns on or off rotational locking of all robot components (body, radar, and gun). Turning locking on causes all components to rotate together at *body* rotation speeds. For example, with locking on, calling the RadarLeft function will cause the entire robot to turn left by the specified amount. Remember, both the gun and radar are forced to rotate at slower body rotation speeds.

See Also:
LockGun

LockGun(*bool*)

bool - True (non-zero) or False (zero) value EXP

Turns on or off rotational locking of a robot's gun and radar. Turning locking on causes the gun and radar to rotate together at *gun* rotation speeds. For example, with locking on, calling the RadarLeft function will cause both the gun and radar turn left by the specified amount. Remember, the radar is forced to rotate at slower gun rotation speeds.

See Also:
LockAll

SyncAll()

Synchronizes both the robot's body and gun to the current radar angle. This function will temporarily override any rotation locks established by previous calls to LockAll and LockGun.

Note: **SyncAll** requires multiple turns to complete, therefore causing command blocking.

See Also:
SyncGun

SyncGun()

Synchronizes the robot's gun to the current radar angle. This function will temporarily override any rotation locks established by previous calls to LockAll and LockGun.

Note: **SyncGun** requires multiple turns to complete, therefore causing command blocking.

See Also:
SyncAll

Scan()

Sends out a radar ping in the direction of the radar. The ping travels in a straight line away from the robot. The distance of the first obstacle encountered is placed in the scandist variable described below. Distance is measured from the *robot's boundary* to the boundary of the other object or wall. If the first obstacle is another robot, mine, or cookie the dtcrobot, dtcmine, or, dtccookie variable will be incremented respectively. This may cause event handlers to be called. Every time **Scan** is called, both the dtcenergy and dtcbearing variables are changed as well.

Fire(*energy*)

energy - Amount of energy to use EXP

Fires an energy missile in the direction of the robot's gun. The amount of damage done by an energy missile is directly proportional to the amount of energy used to fire it and the distance the missile travels. Energy used to fire a missile is removed from the robot's overall energy store. Valid firing values are from 1 to 7. Zero is ignored, negative numbers cause an error, and values greater that 7 are simply reduced to 7. Remember, energy missiles lose energy as they travel. Hitting targets at a great distance has a smaller effect than hitting close targets.

After firing an energy missile, a robot's gun requires time to cool down. **Fire** may be called continuously, but nothing will happen until the gun cools down. Although most robots just ignore this and call **Fire** as often as required, the gunheat variable can be used to determine the current heat of the gun.

Note: An energy missile's total energy is the amount of energy put into a missile multiplied by 4. Although a missile loses energy as it travels, its strength will never go below 4. The damage done to another robot will never go below 5 since 1 point is also lost due to the collision.

If(*expression*)

expression - Expression that evaluates to True (non-zero) or False (zero) EXP

Used to start a logical if block based upon the value of an expression. **If** blocks may be nested, but there should only be one **If** statement opening each block. The expression may contain any legal variable, numeric value, logical operator, or math operator.

See Also:
Elseif, Else, Endif, Logic Operators

Elseif(*expression*)

expression - Expression that evaluates to True (non-zero) or False (zero) EXP

Evaluated if the opening If or previous **Elseif** statement in a logical If block evaluates to *false*. Behaves exactly like an If statement, but may not be the first statement in a logical if block. There may be multiple **Elseif** statements in a single block. The expression may contain any legal variable, numeric value, logical operator, or math operator.

See Also:
If, Else, Endif, Logic Operators

Else

Evaluated when the all previous If and Elseif statements in a logical If block have evaluated to false. If blocks may be nested, but there may only be one **Else** statement in each block.

See Also:
If, Elseif, Endif

Endif

Marks the end of a logical If block. If blocks may be nested, but there may only be one **Endif** statement ending each block.

See Also:
If, Elseif, Else

Gosub(*section*)

section - Name of a section in the robot script

Causes execution to continue at the first line of the specified section. When the called section finishes its last line or hits a Return statement, execution continues at the line after the **Gosub** call.

Note: Sections that are executed with a **Gosub** command inherit the priority of their callers. This implies that sections executed with the **Gosub** command have no unexpected effect on events; they behave exactly as their callers behave.

Return

Causes the current section to end at the current line, returning to the caller. If there was no explicit caller, then next event will be processed.

Round(*value*, *decimals*)

value - Numerical value that should be rounded EXP
range - Number of decimal places to which *value* should be rounded EXP

The first argument is rounded to the number of decimal places specified by the second parameter. The resulting number is placed in the result variable. The *decimals* argument must be an integral number in the range of 0 to 38 inclusive.

Truncate(*value*)

value - Numerical value that should be truncated EXP

The decimal portion of the specified value is removed. The resulting whole number is placed in the result variable.

Abs(*value*)

value - Numerical value whose sign will be dropped EXP

The sign of the specified value is dropped and copied to the result variable. The absolute value of any number has the same magnitude as the original and a positive sign.

Max(*value1, value2*)

value1 - Numerical value that will be tested for maximum EXP
value2 - Numerical value that will be tested for maximum EXP

The two values are compared to determine which is the largest. The number which has the greatest value is copied to the result variable.

Note: Negative numbers close to zero are larger than negative numbers far from zero.

See Also:
Min

Min(*value1, value2*)

value1 - Numerical value that will be tested for minimum EXP
value2 - Numerical value that will be tested for minimum EXP

The two values are compared to determine which is the smallest. The number which has the least value is copied to the result variable.

Note: Negative numbers far from zero are smaller than negative numbers close to zero.

See Also:
Max

GetRandom(*range*)

range - Limiting range for random number generation EXP

Fills the result variable with a pseudo-random number. The generated number will be between 0 and the specified range. Valid ranges are from -32767 to 32767 inclusive. Zero of course, is not a valid range. For example, a random rotational value might be generated by using a range of 359. The resulting random number would be between 0 and 359 inclusive.

GetHitStr()

Fills the result variable with the average damage done by this robot to all other robots in the current game. This is only damage done by missile hits, not collisions. Missed shots do not affect this number. This information might be used to adjust firing tactics.

GetHitsOther()

Fills the result variable with the number of times the robot has hit other robots with an energy missile. This number is often combined with the results of GetShots to modify firing tactics.

GetShots()

Fills the result variable with the number of energy missiles the robot has fired. This number does not reflect whether or not these shots hit something. This number is often combined with the results of GetHitsOther to modify firing tactics.

GetOthers()

Fills the result variable with the number of other robots left in the current game not including the robot calling this function. This number is often used to gauge a robot's performance.

GetTurns()

Fills the result variable with the number of turns the robot has had in the current game.

GetHitsSelf()

Fills the result variable with the number of time the robot has been hit by other robot's energy missiles.

Store(*variable*)

variable - Variable name

This function allows a robot that is fighting in a multiple game match to pass values from one game to the next. This function stores the specified variable in permanent storage for the current match. When the next game starts, all stored variables will be automatically restored. Stored variables will have the same values they contained the last time **Store** was called in a previous game. This may be useful for robots that learn during a match, changing behavior dynamically. This function can not be used to store variables across multiple matches.

Name(*string*)

string - Text surrounded by quotation marks

Sets the robot's name. The string will be used to reference the robot during game play. If this function is not called anywhere in a robot's script, a name will be automatically assigned.

Print(*string*)

string - Text surrounded by quotation marks

Adds the specified string to the output display in a robot's information window. Also, a time stamp is prepended to the output display. At any given point in a game, this time stamp will have the same value for all robots. The output display is limited to 200 entries. When **Print** is called more than 200 times, the oldest entries will be removed first. This function is useful primarily when debugging a robot. During game play, click on a robot's name button to display its information window.

See Also:
Print for expressions

Print(*variable*)

variable - Variable name or numeric value EXP

Adds the specified value to the output display in a robot's information window. Numerical values have 7 digits of precision, but 3 decimal places are always displayed for clarity. Also, a time stamp is prepended to the output display. At any given point in a game, this time stamp will have the same value for all robots. The output display is limited to 200 entries. When **Print** is called more than 200 times, the oldest entries will be removed first. This function is useful primarily when debugging a robot. During game play, click on a robot's name button to display its information window.

See Also:
Print for strings

Stall(*time*)

time - Amount of time to stall EXP

Causes the robot to freeze for the specified amount of time. This command is very useful for debugging purposes.

Note: The robot will not even respond to events. This function completely disables a robot.

Blocking(*bool*)

bool - True (non-zero) or False (zero) value EXP

This is an advanced feature. Use of the **Blocking** command is not required to play robot battle.

This function allows command blocking to be turned on or off. When blocking is turned off, it remains off for the entire robot script until explicitly turned back on.

The default behavior is for blocking to be **on**. When blocking is on, calls to commands that require multiple turns block. This means that within a section, execution will pause on the multi-turn command. Code following the multi-turn command will not be executed until the multi-turn command completes. In other words, all function calls are *synchronous*. When blocking is turned **off**, multi-turn commands do not block. Code following the multi-turn command executes immediately. In other words, all function calls are *asynchronous*.

Blocking should be turned off with great care. A robot's body, gun, and radar can perform only one multi-turn command (i.e. movement) at a time. Only the last command on each body part takes effect. For example, when blocking is off, if a call to BodyLeft is followed immediately by a call to Ahead, the original BodyLeft will be ignored while the robot moves ahead. When blocking is turned off, all previously blocked commands remain blocked. Likewise, when blocking is turned on, all previously unblocked commands remain unblocked. Only commands that are called after a change in blocking are effected by the change.

Turning blocking off is used primarily with the Continue command. When an event handler is called, for example, movement may be stopped and continued without blocking on the Continue command. This allows the event handler to be ended while restricting blocking to the section and line that initiated to original movement.

Note: This command is not related to and has no effect on events or event registration.

See Also:
WaitFor

WaitFor(*expression*)

expression - Expression that evaluates to True (non-zero) or False (zero) EXP

This is an advanced feature. Use of the **WaitFor** command is not required to play robot battle.

This command provides a means of creating a user defined command block. This means that within a section, execution will pause on the **WaitFor** command until *expression* becomes true. Code following the **WaitFor** command will not be executed until *expression* becomes true. Generally, blocks are created using expressions that change over time. Blocks that are based on constant value expressions either block permanently or never block.

This command is generally used as a *synchronization* method. This is particularly useful when normal command blocking has been turned off with the Blocking command.

The **WaitFor** command has no effect on events. All events will be handled normally. If a higher priority event occurs while blocking, for example, its event handler will be called. When the higher priority event handler ends, control will again return to the **WaitFor**.

See Also:
Blocking

Command Blocking

Command blocking is an advanced feature. Unless the Blocking or WaitFor functions are being used, this information should not be needed.

Command blocking occurs when a robot function requires multiple turns to execute. Only commands that cause movement require multiple turn to execute. These include Ahead, Back, BodyLeft, BodyRight, GunLeft, GunRight, RadarLeft, RadarRight, SyncAll, SyncGun, and Continue.

When a command blocks, execution will pause on that command. Code following the multi-turn command will not be executed until the multi-turn command completes. In other words, the command is *synchronous*. Since each robot component can only perform one multi-turn command at a time, blocking greatly simplifies the control of a robot.

When blocking is turned *off*, for example, a `GunLeft(20)` call followed by another `GunLeft(20)` will only move the gun left 20 degrees. Since the first call does not block, the second call immediately supersedes the first call.

Scripting Language Special Section

These sections are considered special because both of them handle events without being registered. The game will automatically call these sections when their pre-defined events occur.

Init	Handles game startup events
Dead	Handles robot death events

Init

This section handles game startup events. It is automatically called at the start of every game. It is always the first section to be executed, and will only be called automatically once. Most robots use this section to register other event handlers. Although **Init** is only called once automatically, it may be called manually at any time by either registering events to it, or by using the Gosub command.

Note: Robots *are* required to have an **Init** section.

See Also:
Dead Section

Dead

This section handles robot death events. It is automatically called when a robot is killed. Robots are killed either when their energy reaches zero or when the game they are playing in ends. Even if a robot wins a game, its dead section will be called. Since the robot is dead, only a subset of the robot functions listed above have meaning. Most robots use the dead section to perform some type of calculation then call the Store function to save information for future games. When called manually, a dead section behaves like any other section.

Note: Robots *are not* required to have a **Dead** section.

See Also:
Init Section

Scripting Language System Variables

These variables describe a robot's state during game play. They may be used in any expression in a robot's script. The only restriction is that these variables are read only. Their values are for informational purposes only and are maintained by the game itself. They may not be changed directly by assignment. Remember, capitalization is not important. A variable named "VARIABLE" will be that same as "variable" or "Variable".

accel	The robot's current acceleration
bodyaim	Current angle of robot's body
bodyrmn	Angular rotation remaining in the robot's body
cldbearing	Bearing to the last object the robot collided with
cldcookie	Cookie collision indicator
cldenergy	Energy of the last object the robot collided with
cldmine	Mine collision indicator
cldmissile	Missile collision indicator
cldrobot	Robot collision indicator
death	Indicates that another robot has died
distrmn	Distance remaining in the robot's lateral movement
dtcbearing	Bearing to the last object the robot detected
dtccookie	Cookie detection indicator
dtcenergy	Energy of the last object the robot detected
dtcmine	Mine detection indicator
dtcrobot	Robot detection indicator
energy	The robot's remaining energy level
false	Constant zero value
gamenbr	Current game number
games	Number of games in the current match
gunaim	Angle of the robot's gun
gunheat	Heat of the robot's gun
gunrmn	Angular rotation remaining in the robot's gun
moving	Lateral movement indicator
off	Constant zero value
on	Constant non-zero value
radaraim	Angle of robot's radar
radarrmn	Angular rotation remaining in the robot's radar
result	Generic computation results buffer
rotating	Rotation indicator
scandist	Distance to the nearest detected object
true	Constant non-zero value

scandist

Each time the Scan function is called, this variable is filled with the distance to the nearest object. This may be the distance to a wall, another robot, a cookie, or a mine. Energy missiles are ignored. Distance is measured from the *robot's boundary* to the boundary of the other object or wall. Also, if another robot, cookie, or mine is detected, the appropriate detection variable will be incremented and the section registered to handle the event will be called.

cldrobot

Set to true when the robot collides with another robot. When the collision occurs, the section registered by RegCldRobot will also be called. Collision indicators are mutually exclusive. When **cldrobot** is *true* all other collision variables will be *false*. This variable is reset to *false* automatically when the robot collision event handle returns. If no section has been registered to handle robot collision events, this value will remain *true* until a collision with a different object occurs.

cldmissile

Set to true when the robot collides with an energy missile. When the collision occurs, the section registered by RegCldMissile will also be called. Collision indicators are mutually exclusive. When **cldmissile** is *true* all other collision variables will be *false*. This variable is reset to *false* automatically when the missile collision event handle returns. If no section has been registered to handle missile collision events, this value will remain *true* until a collision with a different object occurs.

cldcookie

Set to true when the robot collides with an energy cookie. When the collision occurs, the section registered by RegCldCookie will also be called. Collision indicators are mutually exclusive. When cldcookie is *true* all other collision variables will be *false*. This variable is reset to *false* automatically when the cookie collision event handle returns. If no section has been registered to handle cookie collision events, this value will remain *true* until a collision with a different object occurs.

cldmine

Set to true when the robot collides with an energy mine. When the collision occurs, the section registered by RegCldMine will also be called. Collision indicators are mutually exclusive. When **cldmine** is *true* all other collision variables will be *false*. This variable is reset to *false* automatically when the mine collision event handle returns. If no section has been registered to handle mine collision events, this value will remain *true* until a collision with a different object occurs.

cldenergy

When a robot collides with any other object, this variable is filled with the energy of that object. Robots may collide with energy missiles, other robots, cookies, and mines. All objects, including mines, return positive energy values. There is no such thing as negative energy. The value of **cldenergy** will not change until another collision occurs. This variable is often used to judge an enemy robot's relative strength.

cldbearing

When a robot collides with any other object, this variable is filled with the bearing to that object. Robots may collide with energy missiles, other robots, cookies, and mines. This variable is a bearing from the robot's current heading to that object, not an absolute heading. Values are in degrees ranging from -180 to 179. A **cldbearing** of zero is always directly ahead of the robot.

For example, if a robot were heading 135 degrees and an energy missile hit the robot's body at an absolute angle of 90 degrees (3 o-clock), the **cldbearing** variable would be set to -45. In other words, the robot was hit 45 degrees left of its current heading.

Remember, **cldbearing** says nothing about the *direction* an object was traveling when it collided with the robot, only *where* it hit the robot. This should be evident since the other object may not have even been moving. The value of **cldbearing** will not change until another collision occurs.

dtcrobot

This variable is incremented by one when another robot is detected by a call to Scan. It is set to zero when a call to Scan does not detect another robot. When robot detection occurs, the section registered by RegDtcRobot will be called. This variable is decremented by one automatically when the robot detection event handle returns. For this reason, many robots call Scan at the end of their detection event handlers. If no section has been registered to handle robot detection events, this value will remain non-zero until a call to Scan detects no other robots.

dtccookie

This variable is incremented by one when an energy cookie is detected by a call to Scan. It is set to zero when a call to Scan does not detect a cookie. When an energy cookie is detected, the section registered by RegDtcCookie will also be called. This variable is decremented by one automatically when the cookie detection event handle returns. If no section has been registered to handle cookie detection events, this value will remain non-zero until a call to Scan detects no energy cookies.

dtcmine

This variable is incremented by one when an energy mine is detected by a call to Scan. It is set to zero when a call to Scan does not detect a mine. When an energy mine is detected, the section registered by RegDtcMine will also be called. This variable is decremented by one automatically when the mine detection event handle returns. If no section has been registered to handle mine detection events, this value will remain non-zero until a call to Scan detects no mines.

dtcenergy

When a robot detects any other object, this variable is filled with the energy of that object. Robots may detect other robots, cookies, and mines. All objects, including mines, return positive energy values. There is no such thing as negative energy. If no objects are detected by Scan, **dtcenergy** is set to zero. This variable is often used to judge an enemy robot's relative strength.

Every time Scan is called, **dtcenergy** will change. It will either be set to the detected object's energy or zero if no object was detected. This is true even when the detected object does not have a detection event handler registered.

dtcbearing

When a robot detects any other object, this variable is filled with the bearing to that object. Robots may detect other robots, cookies, and mines. This variable is a bearing from the robot's current heading to that object, not an absolute heading. Values are in degrees ranging from -180 to 179. A **dtcbearing** of zero is always directly ahead of the robot.

This variable is provided primarily for consistence with collision variables. Since objects may only be detected by a radar ping, **dtcbearing** always matches the bearing of the robot's radar at the time Scan was called. See **cldbearing** for more details about bearing.

Every time Scan is called, **dtcbearing** will change. It will always reflect the bearing of the robot's radar, even if no objects were detected.

death

When another robot in the current game dies, the **death** variable is set to *true*. This variable is an exception to the read only rule. Since the game never resets **death** to *false*, this must be done by the robot. This variable can be used for custom events, just remember to change it to *false* at some point to end the event. It is easiest to think of the **death** variable as an automatically provided user variable.

energy

The robot's remaining energy level. This number always starts at 100 and is changed by various events during game play. When this value reaches zero, the robot is out of the game. This value will always match that shown on the game's playing field. Please see Damage Summary for more detail.

accel

Current setting of the robot's acceleration. While moving laterally, robots are constantly accelerating. Therefore, this value approximately represents a robot's movement speed. This value is changed by calling the SetAccel function and defaults to 3.

moving

True while the robot is moving laterally and *false* while the robot is stationary or rotating only.

See Also:
distrmn

rotating

True while any part of the robot is rotating and *false* while the robot is stationary or moving laterally only.

See Also:
bodyrmn, gunrmn, radarrmn

gunheat

Current heat of the robot's gun. Every time a robot calls Fire its gun heats up. As time passes, the gun cools down. A robot may only fire another energy missile when **gunheat** reaches zero. Most robots simply ignore this variable and call Fire as often as possible.

distrmn

When a robot is moving laterally, this variable contains the distance remaining until the movement is complete. This information is useful when a robot needs to store or test the amount of lateral movement remaining. If the robot is not moving, this variable will be zero. Do not confuse this variable with the internal "continue buffer" described in the Stop and Continue functions, they are similar but independent.

See Also:
Ahead, Back

bodyrmn

When a robot's body is rotating, this variable contains the amount of rotation remaining until the rotation is complete. This information is useful when a robot needs to store or test the amount of body rotation currently remaining. If the robot's body is not rotating, this variable will be zero. Do not confuse this variable with the internal "continue buffer" described in the Stop and Continue functions, they are similar but independent.

See Also:
BodyLeft, BodyRight

gunrmn

When a robot's gun is rotating, this variable contains the amount of rotation remaining until the rotation is complete. This information is useful when a robot needs to store or test the amount of gun rotation currently remaining. If the robot's gun is not rotating, this variable will be zero. Do not confuse this variable with the internal "continue buffer" described in the Stop and Continue functions, they are similar but independent.

See Also:
GunLeft, GunRight,

radarrmn

When a robot's radar is rotating, this variable contains the amount of rotation remaining until the rotation is complete. This information is useful when a robot needs to store or test the amount of radar rotation currently remaining. If the robot's radar is not rotating, this variable will be zero. Do not confuse this variable with the internal "continue buffer" described in the Stop and Continue functions, they are similar but independent.

See Also:
RadarLeft, RadarRight

bodyaim

Current angle of the robot's body. Values are in degrees ranging from 0 - 359. A **bodyaim** of zero is towards the top of the arena, or map north. This value is changed by the various rotation functions.

See Also:
BodyLeft, BodyRight

radaraim

Current angle of the robot's radar. Values are in degrees ranging from 0 - 359. A **radaraim** of zero is towards the top of the arena, or map north. This value is changed by the various rotation functions.

See Also:
RadarLeft, RadarRight

gunaim

Current angle of the robot's gun. Values are in degrees ranging from 0 - 359. A **gunaim** of zero is towards the top of the arena, or map north. This value is changed by the various rotation functions.

See Also:
GunLeft, GunRight,

result

This is a generic results buffer. Since robot functions do not return values, any function that generates a number fills this variable with its results. This value may therefore change often. It should only be used immediately after calling a function that fills it. If the value is needed at a later time, it should be assigned to a user defined variable. All functions that use this variable mention it in their description.

gamenbr

Current game number. Robot Battle matches have from 1 to 65,500 games. This variable is set to **1** for the first game of a match and incremented by one for each successive game. The **gamenbr** variable will be **2** for the second game of a match, **3** for the third, and so on.

See Also:
games

games

Number of games in the current match. This variable does not change from game to game, only from match to match. It always contains the total number of planned games in a match. Robot Battle matches have from 1 to 65,500 games.

See Also:
gamenbr

on

Evaluates to a non-zero value.

true

Evaluates to a non-zero value.

off

Evaluates to a zero value.

false

Evaluates to a zero value.

Description of Robot Events

The most important concept to robot design is event driven behavior. Event driven means that robots behave by responding to things (events) that happen to them. Designing an event driven robot involves deciding which events a robot should respond to and how it will respond. Responding to an event is commonly referred to as *handling* an event.

A robot is divided into a number of sections. Each section has a name and instructions associated with that name. Sections are grouped by curly brackets {}. See the sample robot, to see actual sections. Sections are used to *handle* events. Events are associated with sections by various registration functions described in Event Registration Functions.

The sections used to handle events define how a robot will behave. Events may be *re-registered* to new handler sections at any time. This allows for extremely flexible robots. At any time during a game, a robot may re-register its event handlers, completely changing its behavior. Once registered, events may also be individually turned on or off.

Events also have a priority associated with them. Since a robot can only do one thing a time, it handles higher priority events first. When an event occurs, and no higher priority events are occurring, the section registered to *handle* that event is called.

When a handler section is called, it always executes to either the last line of code in that section or until a Return statement is hit. This does not mean execution will always go directly from the first to the last line of a handler section, however. Handlers may always be preempted by higher priority events. If an event with a higher priority than the current event happens, its event handler will be called immediately. The lower priority handler will not continue execution until the higher priority event handler completes.

The sample robot called event.prg' should be very helpful. Look at its instructions, then run it in a game. Look at its output in the Robot Information Dialog to verify that all events have occurred as expected.

See Also:
Event Registration Functions, Event Control Functions

Event Registration Functions

The following functions register and re-register event handler sections.

RegAscan	Registers an event handler for auto scanning
RegCldCookie	Registers an event handler for collision with energy cookies
RegCldMine	Registers an event handler for collision with energy mines
RegCldMissile	Registers an event handler for collision with energy missiles
RegCldRobot	Registers an event handler for collision with other robots
RegCore	Registers an event handler for the robot's core behavior
RegCustom	Registers an event handler for custom defined events
RegDtcCookie	Registers an event handler for detection of energy cookies
RegDtcMine	Registers an event handler for detection of energy mines
RegDtcRobot	Registers an event handler for detection of other robots

See Also:
Events Description, Event Control Functions

Events Control Functions

The following functions turn on or off event handling without effecting the registered handler.

AscanEvents	Turns on or off auto scanning events
CldCookieEvents	Turns on or off cookie collision events
CldMineEvents	Turns on or off mine collision events
CldMissileEvents	Turns on or off missile collision events
CldRobotEvents	Turns on or off robot collision events
CoreEvents	Turns on or off core events
CustomEvents	Turns on or off custom events
DtcCookieEvents	Turns on or off cookie detection events
DtcMineEvents	Turns on or off mine detection events
DtcRobotEvents	Turns on or off robot detection events

See Also:
Events Description, Event Registration Functions

Math Operators

Standard math operators. Operator precedence follows that of standard scientific calculations. Brackets () may be used to manually change the order of evaluation. Calculation results are also the same as those produced by a standard scientific calculator.

Description	Usage Format	Output Range
Cosine	cos(degrees)	-1 <= result <= 1
Sine	sin(degrees)	-1 <= result <= 1
Tangent	tan(degrees)	NA
ArcCosine	acos(value) -1 <= value <= 1	0 <= result <= 180
ArcSine	asin(value) -1 <= value <= 1	-90 <= result <= 90
ArcTangent	atan(value)	-90 <= result <= 90
Raise to the power	^	-3.4e ± 38 <= result <= 3.4e ± 38
Multiplication	*	-3.4e ± 38 <= result <= 3.4e ± 38
Modulus	%	integral value
Division	/	-3.4e ± 38 <= result <= 3.4e ± 38
Addition	+	-3.4e ± 38 <= result <= 3.4e ± 38
Subtraction	-	-3.4e ± 38 <= result <= 3.4e ± 38
Numeric values	± 3.4e ± 38 (6 digits precision)	NA

* Variables are automatically defined by placing them on the left side of the assignment command (=). All variables have an initial value of *zero* until explicitly assigned a different value.

Also see logical operators. For a detail description of operators in general, see language semantic.

Logical Operators

These operators are commonly used in If statements and custom events, but may be used anywhere an expression is valid. Several operators have two definitions. The second definition is provided for C' programmers who are stuck in their ways (like me).

Description	Usage Format
Equality comparison	==
Not equal to	<>, !=
Greater than or equal to	>=
Less than or equal to	<=
Greater than	>
Less than	<
Logical AND	and, &&
Logical OR	or, ||

Also see math operators. For a detail description of operators in general, see language semantic.

Tips and Techniques

- Organize robots into small sections (subroutines) the perform specific tasks. Small sections enhance robot clarity, are often reusable within a single robot, and make great cut and paste candidates for new robots.

- Check out the sample robots (fire.prg in particular) for some useful subroutines that can be copied immediately.

- Use the Print statements to help debugging. It can display both strings surrounded by quotations and expressions.

- Write special purpose debugging robots to help figure out how normal robots will behave. Debugging robots behave in a predictable manner (such as driving to the center of the arena) helping to isolate a particular feature or problem in another robot.

- When using a debugging robot, use the Stall function in the normal robot. This will give the debugging robot time to start it predetermined activities.

- Use comments liberally in robot scripts. Comments act as helpful reminders when examining robot scripts. Any text after a **#** or a **//** is considered a comment. Comments are completely ignored by Robot Battle.

- Use indentation with If statements. Indenting lines between If, Elseif, Else, and Endif statements greatly increases a robot script's readability.

- Be careful when using variables that are changed often during game play. These include result, cldbearing, cldenergy, dtcbearing, and dtcenergy. If needed at any time other than immediately after they are filled, assign their values to user defined variables.

- The arena measures 400 unit in each direction, robots measure 33 units in each direction, cookies and mines have diameters of 9 units, and energy missiles are 3 units square.

- As with most computer languages, floating point (real number) math is not perfectly accurate. Particularly after trigonometric functions, testing for equality without calling Round will not always work. For example, **acos(cos(20))** may yield 19.9999 instead of 20.

- The amount of time it takes for a game with no activity to be automatically ended may be changed. The default value is 10,000 turns. To make this value smaller or larger, open the **winrob.ini** file in an ASCII text editor. Change the value of the entry under **[WinRob]** called **auto_end_turns**. If auto_end_turns does not exist, add it under [WinRob] with the desired time-out value.

- Large robots that respond to many events can become quite complicated. Complexities often arise from the need to remember where a robot is and what it is doing before and after each event. Designing robots as **state machines** can simplify this problem. A robot's behavior can be modeled as transitions from one state to the next, allowing easy state recovery during and after an event.