Expanding
Possibility

Function Manual for Hi6 Controller

Robot Language (HRScript)

HD HYUNDAI ROBOTICS

Contents

1. Overview

1.1 Introduction of HRScript

Hyundai Robotics' Hi6 Controller allows the user to program the robot's tasks in a robot language called HRScript. Created programs can be saved as several files with the extension .job.

HRScript is a scripting language that will be interpreted and executed line by line by the interpreter without a compilation procedure. It is similar to the Python or JavaScript languages but has a simpler syntax.

2. Basic Syntax

Described in this section are the basic terms of HRScript. The basic concept of the job program could be understood by following the method for defining a variable, constructing a simple expression using operators, and assigning the resulting value to a variable.

2.1 Statements

The statement refers to each command string that becomes the execution unit of the job program. HRScript allows only one statement per line. Take note of how the four examples of statements are written below, particularly their appearances.

```
    move P,po3,spd=80%,accu=1,tool=3 until do33

10  z_pos = (base_height+offset)*1.05

    # robot has to wait sensor2 input

    *err_handle
```

For statements other than a step statement (move statements, etc.) that moves the robot, you can optionally add a line number (1 to 9999) at the beginning of the line. The number 10 in the second line is an example of a line number.

It does not matter if there are any number of spaces or tabs before and after the statement.

Proper indentation in statements is recommended for readability. Both spaces and tabs are allowed for indentation and do not affect the operation during execution.

2.2 Identifiers

Names must be given to commands, variables, functions, and labels that are described. These names are collectively referred to as “identifiers.” When deciding an identifier, it must comply with the following rules for the HRScript’s identifiers.

- It must consist only of uppercase and lowercase letters, numbers, and underscores.
- The first character must only be either a lowercase or uppercase letter or an underscore, not a number.
- It should not contain a space or tab.
- Identifiers already defined in the system, such as “if” and “for,” cannot be used.
- There is no limit to the length.

The following shows correct and incorrect examples of identifiers:

```
myvar (O)

myvar2 (O)

_myvar (O)

MyVar (0)

310a (X) – Started with a number

move (X) – An identifier already defined in the system

v300$ (X) – Used a symbol other than an underscore ($)

my var (X) – Included a space
```

2.3 Types of Statements

The four types of statements of HRScript are as follows:

- Procedures
- Assignment statements
- Comment statements
- Labels

2.3.1 Procedures

A procedure consists of a command and a 0–N number of parameters.

```
move P,po3,spd=80%,accu=1,tool=3 until do33
```

The three types of procedure parameters are as follows:

Type	Syntax	Example
Position parameter	〈value〉	P, po3
Keyword parameter	〈keyword〉 = 〈value〉	spd=80%, accu=1, tool=3

Preposition parameter	〈preposition〉 〈value〉	until do33

The position parameter's role is determined by its position, so it should not be moved and must always be at the front of the procedure.

Keyword parameters should be placed after position parameters. However, the order between keyword parameters does not affect the operation.

The preposition parameters are placed last.

2.3.2 Assignment Statements

An assignment statement consists of the left side, the assignment operator (=), and the right side. The left side (lvalue) must be a variable that can store a value. No constants or expressions are allowed.

On the other hand, constants, variables, and expressions are allowed on the right side (rvalue).

```
height=(500+margin)/2
```

2.3.3 Comment Statements

A comment statement is used to describe the contents of the job program in a way that they can be understood easily. Even if the comment statement is executed, no operation is performed. As shown below, a description is attached after the hash sign (#). It can be used as a single statement or attached after another statement.

```
# robot has to wait sensor2 input

var work_w,work_h  # width and height of a workpiece
```

2.3.4 Labels

A label is used to mark the target point to move to according to the goto statement. It consists of an asterisk (*) and an identifier.

2.4 First Program – Hello, World!

Let us create a simple job program that prints a string on the teach pendant screen. After creating a new job, record the print statement as shown below, and attach the string parameter “Hello, World!”

```
print “Hello, World!”
```

The print statement is used to print the value at the bottom of the teach pendant’s job panel. Now, when you run the program, you can see the text, “Hello, World!” printed at the bottom of the job panel.

2.5 Data Type

2.5.1 String Data Type

The first program in the previous paragraph used the data “Hello, World!” as the print statement’s parameter, a string data type. The value of the string data type begins and ends with double quotes. There is no limit for the length of the string.

```
print "Welcome to the Robot World."
```

A sequence beginning with a backslash (\) represents double quotes or special characters in a string. This sequence is called the “escape character.”

The supported escape characters are shown in the table below.

\"	Double quotes
\	Backslash
\t	Ttab)
\n	New line character

```
print "Message: \nPlease, press \"OK\" button."
```

```
Result of print

Message:

Please, press "OK" button.
```

2.5.2 Number Data Type

The number data type stores an integer or real number. Let us print using the print statement. If you list multiple values separated by commas (,) in the print statement, as shown in the example below, each value will be displayed separated by a space.

```
280

3.141592

-99
```

```
print 280, -99
```

Inside the system, integers and real numbers are processed separately. Each data size is as follows:

Data type	Data size (byte)
Integer	4
Real number	8

2.5.3 Boolean Data Type

There are only two values, true and false, as the result of the following logic and comparison operations.

```
var x=true

print false and x

print 10 〉 5

print 10 〈= 5


result of print

false

true

false
```

2.5.4 Array Type and Object Type

In addition, there are array types and object types. These will be discussed in further detail in Sections 4.1 and 4.2.

2.6 Variables

A variable can store values and has an identifier name. Variables are divided into global and local variables, and the difference between them will be described later. Examples of local variables are first described here.

Variables can be created with the var command, as shown in the following. This is called defining a variable. Multiple identifiers can be created at once by enumerating multiple identifiers after the var command.

```
var myvar

var width, height, depth
```

Storing a value in a variable is called “assignment.” The assignment may be performed while defining or after defining a variable. If the assignment is not performed while defining, the variable will have a number value of 0 by default.

```
var myvar=0

var message, width=200

message="Invalid input value"
```

In HRScript, (=) does not mean equal. It is used as an assignment operator and means that the value on the operator’s right side is assigned to the variable on the left side. The value stored in the variable may be printed through the print statement.

```
var myvar=0

var message, width=200

message="Invalid input value"

print width, message
```

A different value may be assigned to a variable to which a value has already been assigned. It is called a variable because its value can change.

```
var width=200

width=300
```

2.7 Binary and Hexadecimal

All the number type values previously described as examples are interpreted as decimal numbers. It can represent binary or hexadecimal values just by adding 0b or 0x prefixes, respectively, as shown in the following.

```
var binary = 0b10010011

var hexadecimal = 0xFF4A38C0
```

2.8 Operators and Expressions

In the following example, the variable margin is added to the number value 500, and the resulting value is divided by 2. Thus, the calculated value is assigned to a variable called “height.”

```
var height, margin=10

height=(500+margin)/2

print height
```

Through the print statement, it is possible to check that 255, which is the result of the expression, is assigned to “height.”

In this way, an expression can be created by concatenating operands, which are values or variables, using various operators, and the result can be assigned to a variable or be used as a parameter of a statement.

What operation will be performed first if the addition sign and multiplication sign are used without grouping, as shown below? Multiplication and division will be performed first before addition and subtraction because there is an operation order in which operators are applied, which is called “operator precedence.” Because the operator precedence of multiplication is higher than that of addition, multiplication will be performed first even though the multiplication sign is located at a later place.

```
print 10+10*2
```

Strings will be concatenated when the (+) operator is used for them.

```
var name="axis1", type="rotational"

print name + ":" + type
```

The operators supported by HRScript are as follows. The higher it is, the higher the operator precedence. (In other words, operators with higher operator precedence will be executed first.)

Operator	Meaning	Example
( )	Grouping	(10+10)*2 ; 40
[ ]	Accessing array elements	arr[3]
**	Exponentiation	10**3 ; 1000
+x, -x	Sign	-300
*, /, mod	Multiplication, division, remainder	300/3 ; 100, 8 mod 3 ; 2
+, -	Addition, subtraction	300-100 ; 200
~	Bitwise NOT	~0b11010010 ; 0b11111111111111111111111100101101
& ^ | << >>	Bitwise AND Bitwise XOR Bitwise OR Shift left Shift right (sign maintained)	0b11010010 & 0b11110000 ; 0xd0 0b11010010 ^ 0b11110000 ; 0x22 0b11010010 | 0b11110000 ; 0xf2 0b11010010 << 2 ; 0b1101001000 0b11010010 >> 2 ; 0b00110100
<, <=, >, >=, !=, ==	Comparison operation (!=) means different, (==) means equal.	30 <= 29 ; false response != "ok"
not x	Logical operation NOT	not error_state
and or	Logical operation AND Logical operation OR	height>100 and invert==false timeout or work_count>3

When operands are number or Boolean values, the result of comparison and logical operations is a Boolean data type, and the result type of other operations is a number data type.

In cases where an operand of Boolean type is used without a comparison operator, it means whether its value is equal to true. For example, the two lines below have the same meaning.

```
var result= timeout

var result= (timeout==true)
```

Operators that can be used for strings are addition (+), comparison (!=, ==), and assignment (=). String addition makes it possible to concatenate operand strings, as previously shown.
A comparison operation determines whether a string is different or equal.

```
var response="ok"

print response=="ok"

print response=="ng"
```

Sometimes, the data type of an operand may change automatically during the process of operation.
When a number is used as an operand of a logical operator, the result will be regarded as false if it is 0 and true if it is not..

```
var count_a=1, count_b=0, height=100

print count_a and height〉99

print count_b and height〉99
```

“bitwise NOT” and “shift left/right” are calculated on a 32-bit length basis.

2.9 Functions

What is the process of converting the angle 60 to a radian value or finding the length of the string that the variable mystr contains?

HRScript provides various functions that receive inputs through parameters, perform some processing, and return the result values.

Functions can be used as part of an expression, as shown below..

```
    var dg=60, rd

    rd=deg2rad(dg)



    var limit=40, message="Input your code number"

    var validity= len(message) 〈 limit
```

The list of functions provided in HRScript is as follows. (The tables are sorted in the ascending order of names.)

2.9.1 Math Functions

Function	Description	Example of usage	Result
abs(**a**)	Returns the absolute value of **a**	abs(-300)	300
acos(**a**)	Returns the arc cosine value of **a** in radian format	acos(0.5)	1.0472
asin(**a**)	Returns the arc sine value of **a** in radian format	asin(0.5)	0.5236
atan(**a**)	Returns the arctangent value of **a** in radian format	atan(0.5)	0.4636
atan2(**a**, **b**)	Returns the arctangent value of a triangle with **a** for the y length and **b** for the x length in radian format	atan2(2,1)	1.1071
cos(**r**)	Returns the cosine value of **r** in radian format	cos(3.1415)	-1
deg2rad(**d**)	Returns the radian value of **d** in degree format	deg2rad(-90)	-1.570796

dist(**x**, **y**)	Returns the Euclidean distance from the origin to the (**x**, **y**) coordinate	dist(3.5,10)	10.59481
max(**a**, **b**)	Returns the greater value between **a** and **b**	max(-1.23, -3)	-1.23
min(**a**, **b**)	Returns the lesser value between **a** and **b**	max(-1.23, -3)	-3

near(**a**, **b** [,**e**])	Returns 1 if the difference between the real number values **a** and **b** is less than or equal to **e** and returns 0 if the difference is larger than **e**	near(0.005, 0.0058)
		near(0.005, 0.006)
		near(0.005, 0.006, 0.1)
			1
			0
			1

rad2deg(**r**)	Returns the degree value of **r** in radian format	rad2deg(1.570796)	90
sin(**r**)	Returns the sine value of **r** in radian format	sin(1.5*3.1415)	-1

sqr(**a**)	Returns the square root of **a**	sqr(16)
		sqr(0)
			4
			0

tan(**r**)	Returns the tangent value of **r** in radian format	tan(3.141592/4)	0.9999

2.9.2 String Functions

Examples with var str="hello, world" executed

Function	Description	Example of usage	Result
bin(**a**)	Returns the string of number **a** in binary representation	bin(0b0010)	"10"
chr(**a**)	Returns the character with **a** of the ASCII code in string type	chr(65)	"A"
double(**s**)	Returns the real number type value of the real number string **s** (Interpret only up to the position where interpretation is possible, and discard the rest.)	double("29.38E-2")	0.2938

hex(**a**)	Returns the string of number **a** in hexadecimal representation	hex(0x7A2F)	"7A2F"

int(**s**)	Returns the integer type value of the integer string **s** (Interpret only up to the position where interpretation is possible, and discard the rest.)	int("13.25")
		int("29.38E-2")
			13
			29

left(**s**, **n**)	Returns a string of the first **n** characters of the string **s**	left(str, 3)	"hel"

len(**s**)	Returns the length of the string if **s** is a string and returns the number of elements in the array if **s** is an array	len("HELLO")
		len([20, 30, 80])
			5
			3

mid(**s**, **i**, **n**)	Returns a string of **n** characters starting from the **i**-th character of the string s (The position of the first character is 0.)	mid(str, 3, 5)	"lo, w"
mirror(**s**)	Returns the string inverted from the string **s**	mirror("HELLO")	"OLLEH"
right(**s**, **n**)	Returns a string of the last **n** characters of the string **s**	right(str, 3)	"rld"
str(**a**)	Returns a string of number **a** in decimal representation	str(13.25)	13.250000

strops(**s**, **p**)	Returns the first position in the string **s** that matches the string **p** (The first character position will be 0 or -1 if there is none.)	strpos(str, "llo")
		strpos(str, "hi")
			2
			-1

2.9.3 Date and Time Functions

Function	Description	Example of usage	Result
date( )	Returns the current date in string type (YYYY-MM-DD format)	date( )	"2019-04-17"
time( )	Returns the current time in string type (HH:MM:SS format)	time( )	"08:48:14"
timer( )	Returns the time elapsed in seconds (sec) from when the power was turned on	timer( )	2796.37

2.9.4 Constructor Functions

These functions receive an input of a parameter and then create and return an object.

Function	Description	Example of usage	Result

Array(n)
Array(a, b, c)
	Creates and returns an array of “n” elements
	The initial value of the element is 0.
	A multidimensional array is created if two or more elements are designated.
	See section 4.1.
		Array(900)
		Array(3,4)
			Array [900]
			Array [3] [4]

Pose(element)	Creates and returns a pose object Refer to Section 5.1.		Pose object
Shift(element)	Creates and returns a shift object Refer to Section 5.2.		Shift object

2.9.5 Other Functions

Function	Description	Example of usage	Result
cpo(crd, mode)	Returns the current pose of the robot to the “crd” coordinate system For values that can be used as “crd” elements, see the table under Section 5.1.	cpo("joint", "cmd")	Pose* that stores the command value of the robot to the axis coordinate

	If the mode is "cmd," it is the command value, and if the mode is "cur," it is the current value. The "crd" and "mode" parameters may be omitted, and their default values are "base" and "cur," respectively.		system

mkucs(n,po)
mkucs(n,po1,po2,po3)
	Creates and registers the nth user coordinate system object
	Refer to Section 5.5.
			0: OK
			〈0: Error code

result()	For some procedures, it may be necessary to check the results. If the result() function is called right after the procedure is executed, the execution result can be returned.	result()	

* Pose is a data type that represents the posture of the robot or the position of the tool tip. Details will be described later in Section 5.1.

3. Control Statements and Subprograms

The statements in the job program are executed line by line in top-to-bottom order. However, depending on certain conditions, the statements can be skipped without being executed, or certain statements can be executed repeatedly. Let us look at the control statements that can control the flow of the program in this manner.

3.1 Address

Moving to another position in the program without executing the next line in order is called a "branch."

The address is the destination of the branch.

There are two ways to define an address: line number and label. In the following example, "10" in the second statement is the line number, and the last statement "*err_handle" is the label.

```
         move P,po3,spd=80%,accu=1,tool=3 until do33

    10   z_pos = (base_height+offset)*1.05

         # robot has to wait sensor2 input

        *err_handle
```

3.2 Stop Statement and Wait Statement

This statement can stop the execution of a program or make it wait for a certain period of time or until the conditions are satisfied.

3.2.1 Stop Statement

Description	This will stop the program. When the program is restarted, execution will continue from the next line.
Syntax	stop

Example of usage	if di9
	stop
	endif

3.2.2 End Statement

Description	This will stop the program. Execution will restart from the beginning of the main program when in continuous playback mode or in restart mode.
Syntax	end
Example of usage	move p,spd=70%,accu=1,tool=0

3.2.3 Delay Statement

Description	Makes it possible to progress to the next command statement after waiting for a designated time.		
Syntax	delay 〈time〉		
Parameter	Time	Time to wait	Arithmetic expression 0.1~60.0 sec
Example of usage	delay 3.5		

3.2.4 Wait Statement

Description	Makes it possible to move to the next command statement after waiting until a designated condition becomes true.		
Syntax	wait <condition>[,<timeout>,<timeout address>]		
Parameter	Condition	Conditions in which waiting is required	Conditional expression
	Timeout	Maximum time limit during which waiting will occur when the condition is false (timeout)	Arithmetic expression 0.1~60.0 sec
	timeout address	Address to which branching will be made when the timeout is exceeded.	address
Example of usage	wait sensor_ok wait (sensor_ok and pos_ok),10,*timeout		

3.3 Goto Statement

Makes it possible to go to a different address, without conditions..

3.3.1 Goto Statement

Description	Makes it possible to go to a designated address.	
Syntax	goto <address>	
Parameter	address	Address to go to An arithmetic expression is possible in the case of a line number.
Example of usage	goto 99 goto addr goto *err_hdl	

3.4 Conditional Statements

These statements allow a certain operation to be or not to be executed depending on certain conditions.

3.4.1 Single-Line if Statement

The form of a single-line if statement is as follows: If 〈Boolean expression〉 is true, branching to 〈address〉 will occur. If false, moving to the next statement will occur.

```
if 〈Boolean expression〉 then 〈address〉
```

Below is an example of the single-line if statement. If the condition that pressure is greater than the limit is true, branching to the label address “*err will occur,” making it possible to print a warning that the pressure is too high. If the condition is false, the next statement will be executed one after the other without branching, so “In normal operation ” will be printed, ending the program.

```
var pressure=95, limit=90

if pressure 〉 limit then *err

print " in normal operation."

end

*err

print " warning: pressure is too high."
```

3.4.2 Complex if-endif Statement

If the single-line if statement is true, only the operation of branching to a specific address will occur. If executing other operations or multiple statements is necessary, the complex if statement should be used.

The form is as follows: If 〈Boolean expression〉 is true, the multiple number of 〈statement〉 between if and endif will be executed in order. If 〈Boolean expression〉 is false, skipping to the position after endif will occur without the 〈statements〉 being executed.

```
if 〈 Boolean expression 〉

        〈 statement 〉

        …

endif
```

In the following example, if the pressure is greater than the limit, the following assignment and print statements will be executed. Otherwise, branching to the end will occur without the statements being executed.

```
var pressure=95, limit=90, exceed
if pressure 〉 limit
    exceed = pressure - limit
    print " warning: pressure is too high."
endif
end
```

In the example program, the statements between if and endif are indented by two spaces. These statements are indented to make it easier to recognize that they are codes for the blocks nested between if and endif.

3.4.3 Complex if-else-endif Statement

If the expression is false and if there are statements to be executed, the following form is used:

If the expression is true, statement A will be executed. If false, statement B will be executed.

```
if 〈 Boolean expression 〉
    〈statement A〉
    …
else
    〈statement B〉
    …
endif
```

An example of its usage is as follows

```
var pressure=95, limit=90, exceed
if pressure 〉 limit
    exceed = pressure - limit
    print " warning: pressure is too high."
```

```
else
    print " in normal operation."
endif
end
```

3.4.4 Complex if-elseif-else-endif Statement

In the case of multiple conditions, the elseif statement can be used in the following form.

```
if 〈Boolean expression〉
    〈statement A〉
    …
elseif 〈 Boolean expression〉
    〈statement B〉
    …
elseif 〈Boolean expression〉
    〈statement C〉
    …
else
    〈statement N〉
    …
endif
```

An example of its usage is as follows.

```
var pressure=95, limit_h=90, limit_m=80
if pressure 〉 limit_h
    print " warning : pressure is too high."
elseif pressure 〉 limit_m
```

```
          print " notification: pressure is high."
     else
          print " in normal operation."
     endif
     end
```

3.4.5 switch~case~break~end_switch statement

A switch statement evaluates a numeric expression and compares it with the resulting value of the numeric expression designated by a case statement. It is executed from the case statement of equal value until a break statement is encountered.

In the following example, if the resulting value of Expression X is equal to the resulting value of Expression B1 or B2, (1) through (3) will be executed, and it will move to the point of the end_switch statement (note that there is no break below the command statement B). Meanwhile, if the resulting value of Expression X is equal to that of Expression C, (2) through (3) will be executed.

If the resulting value of Expression X is not equal to that of any case statement, it will be moved to the default, and (4) through (5) will be executed. Then, the default section may be omitted.

```
     switch 〈expression X〉
     case 〈expression A〉
          〈statement A〉
          …
          break
     case 〈expression B1〉
     case 〈expression B2〉
          〈statement B〉     ⋯ (1)
     case 〈expression C〉
          〈statement C〉     ⋯ (2)
          …
          break              ⋯ (3)
     default
          〈statement N〉     ⋯ (4)
          …
          break               … (5)
     end_switch
```

Any expressions such as Boolean, numeric, string constant, parameter, and numeric, are permissible.

An example is given below:

```
      var state="timeout"
      var res=0

      switch state
      case "ok"
        res=11
        break
      case "timeout"
      case "timeover"
        res=33
        break
      case "invalid"
        res=55
        break
      case "fault"
        res=77
        break
      default
        res=99
        break

      end_switch



    99 end
```

3.5 Nested Control Statements

In the control statement block, another control statement block can be placed, as shown in the following example. In the following form, two nesting levels are shown, but multiple nesting levels can be made as much as necessary.

```
if 〈Boolean expression〉

    if 〈Boolean expression〉

            〈statement A〉

            …

    else

            〈statement B〉

            …

    endif

endif
```

An example of a nested if statement is as follows.

```
var pressure=95, limit=90, inject_on=true

if inject_on

    if pressure 〉 limit

            print " warning: pressure is high."

    else

            print " in normal operation."

    endif
endif
end
```

3.6 Repetitive Statements

Repetitive statements can be used when the same operation needs to be repeated multiple times.

3.6.1 for-next Statement

The format of the for-next statement, which repeats the same operation, is as follows.

First, the initial value will be assigned to the index variable. When the next statement is encountered while the statements under the for statement are executed, the index variable will add increment/decrement values and perform repetition from the point of the for statement. When the index variable passes the end value, the repetition will end.

If a step is not specified, 1 will be applied.

```
for 〈index variable〉=〈initial value〉 to 〈end value〉 [step 〈increment/decrement value〉]
    〈statement〉
    …
next
```

The following shows an example of a routine that accumulates 1 to 10 in the sum using the for-next statement. When the repetition is over, 11 and 55 will be printed on the screen.

```
var idx
var sum=0
for idx=1 to 10
    sum=sum+idx
next
print idx, sum
end
```

3.7 Call Statement, Jump Statement and Subprograms

If an entire large-scale robot operation is created as one job program, the program becomes large and complex, making it difficult to add functions or find and solve problems.

For the program's maintainability, it is preferable to divide the unit operations that make up the entire program into subprograms. For example, when routines, such as a routine performs communication with a sensor, a routine that calculates the target position of the tool tip with the received data, and a routine that generates an appropriate message when an error occurs, are turned into individual subprograms and allow the main program to call them, it will be easier to grasp the overall structure of the program. It will also be useful to reuse divided subprograms in other projects.

3.7.1 Format and Simple Example of Call Statement

There is no significant difference in format between the main program and the subprogram in HRScript. The first job executed by the start button or by a signal is the main program, and all other jobs called by the call statement are subprograms.

The format of the call statement is as follows.

```
call 〈job number or file name〉 [,parameter 1, parameter 2…]
```

Specify the job number of the job file name (excluding the extension) after the call statement. Then, while program A is being executed, if call B is encountered, A's execution will be stopped, and the first statement of program B, a subprogram, will continue to be executed. If the end statement is encountered while B is being executed, program A's execution will continue upon returning to the position of the next statement of program A's call statement that was previously called.

The following shows an example and the result of a subprogram called by a call statement. It seems meaningless to divide the program into two because the subprogram must handle only one print statement. However, a more practical example will be shown later.

0001.job	print "main job start" call 102_err print "main job end" end
0102_err.job	print "sub-program" end
Result	main job start sub-program main job end

3.7.2 Parameters and param Statement & return Statement

In a job program, formal parameters are used as channels through which input and output are passed. The param statement will define formal parameters at the beginning of the job program.

In the following example, job no. 105 is named as "dist2d," as it is a subjob that acquires the Euclidean distance from the origin to the coordinate value (x, y) and returns it to len.

0001.job	var x,y x=5 y=12.8 call 105_dist2d,x,y var res=result() print res end
0105_dist2d.job	# Calc. Euclide distance 2D param x,y var tmp tmp=x*x+y*y var len=sqr(tmp) # distance from origin return len
Result	13.742

In job no. 1, the dist2d subprogram is called with the call statement, and “x, y,” which are local variables, are passed. In the dist2d subprogram, “ldX,” and “ldY” defined with the param statement are called “formal parameters,” and “x, y” passed to the call statement are called “actual parameters.”

The dist2d program transports resulting values to external destinations through return statements. Returned values can be obtained by calling a result() function in the called program.

(A return statement and an end statement have the same action as they end a called program and return to the main program. However, a return statement is different from an end statement as the former can designate a resulting value as an element).

3.7.3 Format and simple example of jump statement

The format of jump statements is as follows:

```
jump 〈job number or file name〉 [,parameter 1, parameter 2, …]
```

This format is completely identical to that of call statements, and its action is also similar to that of call statements.

The only difference is that, while a call statement returns to the main program using an end program, a jump statement does not.

Therefore, if the jump statement of this example program is replaced with a call statement, the result of the replaced program will be as follows. When the end of the sub-program (0102_err) is encountered, the action cycle will end. If the next action cycle is executed, the main program (0001) will be executed from the start.

0001.job	print "main job start" jump 102_err print "main job end" end
0102_err.job	print "sub-program" end
Result	main job start sub-program

3.8 Local Variables and Global Variables

Examples have been described only using the examples of local variables defined with the var statement. Local variables are created by a var statement in one job program and are automatically destroyed when the program ends after the encounter with the end statement. Moreover, their values cannot be read or written by other programs.

In the following example, “main_v” is a local variable accessible only within 0001.job, and “sub_v” is a local variable accessible only within 0107.job. Attempting to access it from another program will cause an error.

The local variable “x” is defined in both 0001.job and 0107.job. The local variable “x” respectively defined in both programs has the same name but are different. So the value 5 for the variable “x” is set in subprogram 0107, 3 will be printed instead of 5 after the return to main program 0001.

0001.job	var main_v=10 var x=3 call 107 print main_v # ok print sub_v # error print x # 3 is printed end

0107.job	var sub_v=20
	var x
	print sub_v # ok
	print main_v # error
	x=5
	end

On the other hand, global variables defined as global can always be accessed from all job programs. If a global variable is once defined, it will not be cleared even when the program cycle is reset by an end statement or an R0 [Enter] operation of the main program.

In the following example, if a global x is executed first, a variable x will be created, and the value will be initialized to the default value of 0. Then, it will increase to 1 in the next row. If the global x is executed again in the next program cycle, it will not be defined again, and the value of 1 will be retained because the x has been defined. On the other hand, global y=10 will carry out defining and assignment so that the value of variable y will be reset to 10 when it is executed in the next program cycle.

0001.job	global x x=x+1 call 107

	print x, y # 4, 10 end

0107.job	global y=10
	print x, y # 3, 10
	x=x+1
	end

Therefore, if a global variable is to be utilized as a counter for the number of program cycles, no value should be assigned along with a definition.

Wrong
Teaching
	global count=0
	count=count+1
	…
	end

Correct
Teaching
	global count
	count=count+1
	…
	end

When there are local variables and global variables with an identical name, the local variable will be accessed preferentially. For example, while 0005.job is executed, as shown below, the global variable x and the local variable x will exist concurrently. At this time, if you read the x value, the local variable will also be read. After 0005.job returns to 0001.job, if you read the x value, the global variable will be read because only the global variable is present.

0001.job	global x=100 call 5 print x # 100 end
0005.job	var x="hello" print x # hello

	end

4. Arrays and Objects

4.1 Arrays

4.1.1 Arrays

An array is a variable type that collects and stores several values under a single name and allows access through an index number.

Arrays are defined as var or global, like any other variable. Array definitions and access formats are as follows.

Definition	var array name = [Value, Value, …]
Access	Array name [Index]

The values that make up an array are called “elements.” Distances, an array shown in the following example, has a total of five elements. The index starts from 0. Element 0 and e lement 1 of “distances” are 10 and 10.5, respectively.

The [ ] operator is used as follows to read or write the value of an array’s specific element value. The following shows an example of an object that is defined and accessed.

0001.job	var distances = [10, 10.5, 12.7, 11.92, 9.5]
	distances[1]=20.5
	print distances[0], distances[1]
	end

Result	10 20.5

The number of elements in an array can be acquired by using the len() function. Previously, the len() function was introduced as a function to acquire the length of a string. If an array is put as a parameter of len( ), it will return the number of elements in the array.

Function name	Description	Example of usage	Result

len(**a**)	Returns the length of the string if **a** is a string. Returns the number of elements in the array if **a** is an array	len("HELLO")
		len([20, 30, 80])
			5
			3

The for-next statement is mainly used to perform some processing on all elements of an array.

0001.job	var i
	var distances = [10, 10.5, 12.7, 11.92, 9.5]
	for i=0 to len(distances)-1
	  distances[i] = distances[i]+10
	  print distances[i]
	next
	end

Result	20 20.5 22.7 21.92 19.5

It does not matter if the values stored in the array are of different types..

0001.job	var i
	var arr = [10, "abc", true]
	for i=0 to 2
	  print arr[i]
	next
	end

Result	10 abc true

4.1.2 Multidimensional Arrays

An array can also be nested as an element of an array. When accessing the elements of a multidimensional array, you can use the [ ] operator consecutively. In the following example, “arr_y” is a two-dimensional array. (1)

arr_y[1] is an array of elements of index 1, namely ["abc", "jqk", "xyz"], and it is assigned to the new variable “arr_x.” (2)

So, arr_x[1] is "jqk", and arr_y[1][2] is "xyz" because it points to [2] of arr_y[1].

0001.job	var arr_y = [[10,20], ["abc","jqk", "xyz"]] # (1) var arr_x=arr_y[1] # (2) print arr_x[1] print arr_y[1][2]
Result	jqk xyz

4.1.3 Array Constructor Function

It is difficult to create an array with hundreds of elements with the notation [ ] alone. Any number of arrays may be created by calling the constructor function. Each element will be initialized to 0.

```
var name = Array(900)   # creates an array of 900 elements
```

If two or more elements are designated, a multidimensional array can be created. In the following example of a 3-dimensional array, [4] is the lowest dimension.

```
var name = Array(3,2,4)   # [3] [2] [4] numbers of 3-dimensional arrays are created

# [ [[0,0,0,0], [0,0,0,0]], [[0,0,0,0], [0,0,0,0]], [[0,0,0,0], [0,0,0,0]] ]
```

4.2 Object

As previously seen, it was found that an array could store multiple element values and are accessed by index.

Objects are like arrays in that they store multiple element values. The difference is that an object is accessed by a key, not with an index. Moreover, the key is a string, not a number.

Objects are defined as var or global, like any other variables. The definition of an object and format of its access are as follows.

Definition	var object name = { key : value, key : value, …}
Access	Object name key

The following shows an example of defining and accessing an object.

0001.job	var gap = { x:200, y:152.6 }
	gap.x = gap.x + 10
	print gap.x, gap.y

Result	210 152.6

The object's key must be in the format of an identifier, but the element's value can be of any type and can also be of different types.

An object can contain other objects or arrays as its elements. Likewise, an array can also contain other arrays or objects as its elements. In the following example, “work,” which is an object, contains “size,” which is an object, and “heights,” which is an array.

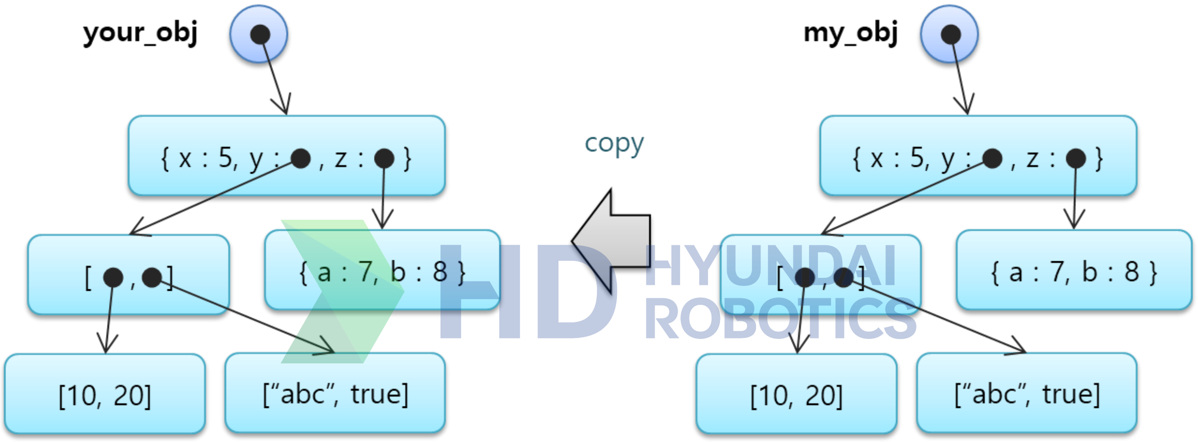
0001.job	var work = { part_no:3, name: "gear", tested : false
	, size : { x : 150, y : 80 }
	, heights : [72.89, 74.91, 81.03, 87.60, 87.11] }
	print work.tested, work.size.y, work.heights[3]

Result	false, 80, 87.600000

4.3 Copied assignment of arrays and objects

If the right side of an assignment statement has object variables, the entire values of the variables will be copied to the variables of the left side. When an array or an object includes sub-arrays and sub-objects in a complex manner as element values, such inclusion structures will be copied, which is called a deep copy.

0001.job	var my_obj = [x:5, y:0, z:0] my_obj.y=[[10, 20], ["abc", true]] my_obj.z={ a:7, b:8 } var your_obj=my_obj # deep copy print your_obj.y[0]
Result	[10, 20]



4.4 4.4. Call-by-reference and call-by-value

In the description of call statements and jump statements given in Section 3.4, the concepts of formal parameters and actual parameters were explained. When an actual parameter has been transported to a sub-program, if the sub-program ends after changing the value of the parameter, will it be reflected to the main program?

For example, let's assume that a sub-program 0005_pow3.job raises a value to the third power as follows:

0001.job	var x=2 call 0005_pow3,x print x end

0005_pow3.job	param p
	var t=p
	p=t*t*t # (1)
	end

Result	2

Although we expected that 8 is output because 2×2×2 is 8, the result is 2. It is because, when a numeric-type actual parameter is transported to a sub-program, the value is copied as a parameter. In other words, in (1), because the value raised to the third power was assigned to the copied version, it did not affect the value of the original parameter, x.

Therefore, the teaching program should be corrected so that the resulting value is transported by a return statement.

0001.job	var x=2 call 0005_pow3,x x=result() print x end

0005_pow3.job	param p
	var t=p
	p=t*t*t
	return p

Result	8

On the other hand, in the case of arrays or objects, the reference of actual parameters, not the copied versions, will be transported. A reference refers to the position of a parameter.

In the following example, where the sub-program 0006_pow3.job raises each element of an array to the third power, the values of the elements of the actual parameter array are changed.

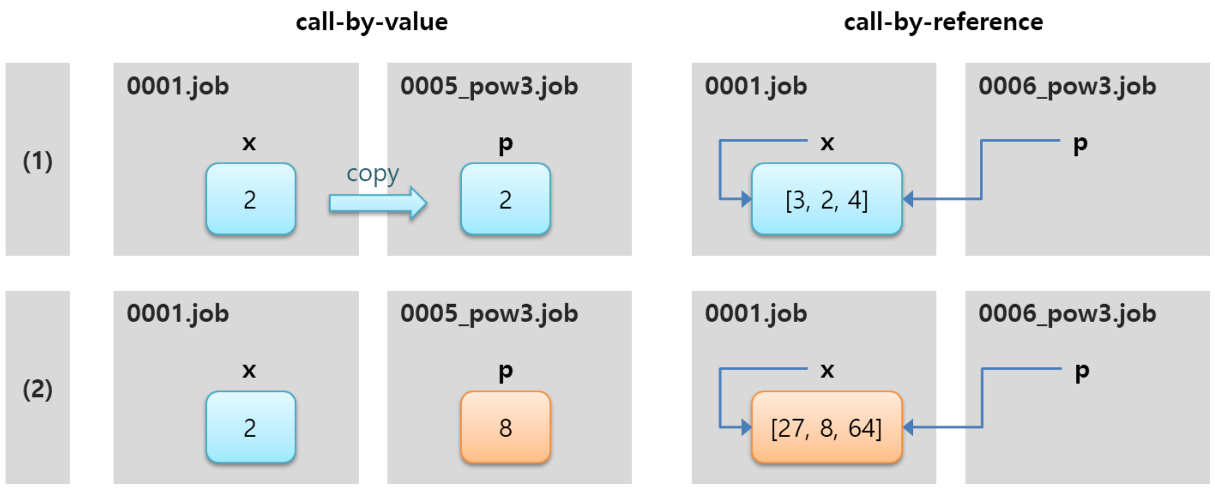
0001.job	var x=[3, 2, 4] call 0006_pow3,x print x end

0006_pow3.job	param p
	var i,t
	for i=0 to len(arr)-1
	  t=p[i]
	  p[i] = t*t*t
	next
	end

Result	[27, 8, 64]

When a sub-program is called, if the copied version of the value of an actual parameter is transported, it is referred to as call-by-value; and if a reference is transported, it is referred to as call-by-reference. Whether it will be call-by-value or call-by-reference is determined by the type of values as follows:

call-by-value	Boolean, numeric, and string types
call-by-reference	Array and object types



5. Moving a Robot with Robot Language

After understanding the pose that expresses the target position of the robot, let us learn about the commands to move the robot.

5.1 Pose

Pose is an object type embedded in the Hi6 Controller and represents each axis of the robot or the Cartesian coordinates and direction of the tool tip.

Poses are created by calling the constructor function Pose( ). All function parameters are position parameters. Meanwhile, crd and cfg are string types, and the rest are number types.

```
var pose variable name = Pose(j1, j2, j3, …)                              # axis coordinate

var pose variable name = Pose(x, y, z, rx, ry, rz, j7, j8,…, crd, cfg)    # base coordinate
```

Refer to the following examples of creating the poses for 6 axes + 1 additional axis and for Cartesian + 1 additional axis.

```
var po1 = Pose(10, 90, 0, 0, -30, 0, -1240.8)                      # axis coordinate

var po2 = Pose(1850, 0, 2010.5, 0, -90, 0, -1240.8, "base", "nf;r2")   # base coordinate
```

Alternatively, the pose constructor function may be called using a single array or string parameter. With this, files or data may be converted into poses, acquired through remote communication, and used.

```
var pose variable name = Pose(array)

var pose variable name = Pose(string)
```

Refer to the following example.

```
var arr = [10, 90, 0, 0, -30, 0, -1240.8]

var str = "[1850, 0, 2010.5, 0, -90, 0, -1240.8, \"base\", \"nf;r2\"]"

var po3 = Pose(arr)

var po4 = Pose(str)
```

Elements of the pose object can be accessed with the following keys.

Key	Type	Value range	Description	Unit, Remarks
nj	Integer type	1~16	Axis count	
j1 ~ j16	Real number type	8-byte real number range	Axis value	mm, deg
x, y, z	Real number type	8-byte real number range	Cartesian coordinate value	mm
rx, ry, rz	Real number type	8-byte real number range	Cartesian direction value	deg
crd	String type	joint	Joint coordinate system (default)	
		base	Base coordinate system	
		robot	Robot coordinate system	
		u1 ~ u10	User coordinate system	
cfg	String type	s	|S|>=180	Possible to perform combination by dividing with ";" The default is all flags turned off.
		r1	|R1|>=180	
		r2	|R2|>=180	
		b	|B|>=180	
		re	rear	
		dn	down	
		nf (old version[1]:fl)	non-flip	
		auto	auto (automatic decision)	

The pose element values can be accessed as shown in the following example.

```
po1.j2 = po1.j2 + 5

print po2.z, po2.cfg
```

[1] For V60.06-06 or older versions, fl is non-fl.

5.2 Shift

Shift is an object type embedded in the Hi6 Controller and represents the pose's change value.

Shifts are created by calling the constructor function Shift( ). All function parameters are position parameters. Meanwhile, crd and cfg are string types, and the rest are number types.

```
Var shift variable name = Shift(j1, j2, j3, ⋯)                     # axis coordinate

var shift variable name = Shift(x, y, z, rx, ry, rz, j7, j8,⋯, crd)  # base coordinate
```

Refer to the following examples of creating the shifts for 6 axes + 1 additional axis and for Cartesian + 1 additional axis.

```
var sft1 = Shift(30, 0, 0, 0, -5.8, 0, -120)                 # axis coordinate

var sft2 = Shift(0, 0, 55.2, 0, -5, 0, -120, "base")         # base coordinate
```

Alternatively, the constructor function shift may be called using a single array or string parameter. With this, files or data may be converted into shifts, acquired through remote communication, and used.

```
var shift variable name = Shift(array)

var shift variable name = Shift(string)
```

Refer to the following example.

```
var arr = [30, 0, 0, 0, -5.8, 0, -120]

var str = "[0, 0, 55.2, 0, -5, 0, -120, \"base\"]"

var sft3 = Shift(arr)

var sft4 = Shift(str)
```

Elements of the shift object can be accessed with the following keys

Key	Type	Value range	Description	Unit, Remarks
nj	Integer type	1–16	Axis count	

j1–j16	Real number type	8-byte real number range	Axis value	mm, deg
x, y, z	Real number type	8-byte real number range	Cartesian coordinate value	mm
rx, ry, rz	Real number type	8-byte real number range	Cartesian direction value	deg
crd	String type	joint	Joint coordinate system (default)	
		base	Base coordinate system	
		robot	Robot coordinate system	
		tool	Tool coordinate system	
		u1–u10	User coordinate system	

5.3 Pose Expression

The expression in which the result value becomes a pose is called a “pose expression.”

All the following forms are recognized as poses.

```
Pose

Pose+Shift

Pose-Shift

Pose+Shift+Shift+…
```

Refer to the following example of assigning the result of a pose expression to another pose variable.

```
var po1 = Pose(10, 90, 0, 0, -30, 0)

var po2 = Pose(1850, 0, 2010.5, 0, -90, 0, "base", "nf;r2")

var po3 = cpo("robot")

var sft1 = Shift(30, 0, 0, 0, -5.8, 0)

var po4 = po1-sft1

var po5 = po2+sft1+Shift(0, 0, 55.2, 0, -5, 0, "base")
```

5.4 Move Statement

The move statement is a procedure for moving the robot. The format is as follows.

Description	The robot's tool tip moves to the pose position.		
Syntax	move 〈interpolation〉, [tg=〈pose/shift〉], spd=〈speed〉, accu=〈accuracy〉 , tool=〈tool number〉 [until 〈conditional expression〉]		
Parameter	Interpolation	P: Axis interpolation; L: Linear interpolation; C: Circular interpolation, SP: Stationary axis interpolation, SL: Stationary tool linear interpolation, SC: Stationary tool circular interpolation	
	Pose/Shift	Target posture (pose) to move to It will be omitted if there is a hidden pose. If a shift expression is specified with a + or - sign, (hidden pose + shift expression) will be applied as the target posture.	Pose expression or a signed shift expression
	Speed	Moving speed of the tool tip A unit (mm/sec, cm/min, sec, %) should be added.	Arithmetic expression
	Accuracy	Arithmetic expression The lower the value, the more accurate. If it is 0, the operation will occur discontinuously.	0~7
	Tool number	The number of the tool to be used when the robot is operating	0~31
	Conditional expression	As soon as the conditional expression is true, the robot operation will end, and the designated pose is considered to have been reached. The result of the conditional expression can be acquired with the result() function.	True if not 0 False if 0
Example of usage	move L,tg=po[0]+sft[1],spd=800mm/sec,accu=0,tool=1 move P,tg=+Shift(0,0,0,0,-10,0),spd=80%,accu=1,tool=3 until di2 (hidden pose) if result() then *sensor_on		

If the [Record] button of the teach pendant is pressed, a move statement in hidden pose type will be recorded as the current robot position. The hidden pose value can be checked or edited by placing the cursor on the move statement and pressing the [Property] button.

When the [Command] button is pressed and the [Motion] group is opened, select the move menu. As a result, a pose-type move statement is recorded.

5.5 User Coordinate System (UCS)

The user coordinate system is a coordinate system in which the user can set the position and direction.

It is created by calling the constructor function Ucs( ).

Function parameters are one pose or three poses. When calling is performed with one pose, the pose's position and direction will be set as the origin and direction of the coordinate system. When calling is performed with three poses, the coordinate system will be created so that pose1 is positioned on the coordinate system's origin, pose2 is on the x axis of the coordinate system, and pose3 is positioned on the XY plane of the coordinate system.

```
var UCS variable name = Ucs(pose1)

var UCS variable name = Ucs(pose1, pose2, pose3)
```

The mkucs function should be used to register a user coordinate system in the system. The argument is similar to the Ucs constructor, but the user coordinate system number (one or more) will be inputted as the first argument.

```
var res = mkucs(num, pose1)

var res = mkucs(num, pose1, pose2, pose3)
```

0 will be returned if successful. An error code of a negative number will be returned if failed.

6. Communicating with External Devices

6.1 FB Object: Digital I/O

Digital input/output (I/O) can be performed through 10 FB objects that can be accessed from HRScript. “FB” refers to fieldbus block, and each FB object is set to be mapped to the I/O hardware installed in the robot controller and contains input and output variables as elements.

6.1.1 Input/Output Variables

			Type	Value range
fb0 ~ fb9	Digital output	do[0~959]	bit	0, 1
		dox[0~119].b[0~7]	bit (byte unit group)	0, 1
		dob[0~119]	signed 1-byte integer	-128 ~ +127
		dow[0~118]	signed 2-byte integer	−32,768 ~ +32,767
		dol[0~116]	signed 4-byte integer	−2,147,483,648 ~ +2,147,483,647
		dof[0~116]	signed 4-byte real number	3.4E+/-38 (7 significant figures)
	Digital input	di[0~959]	bit	0, 1
		dix[0~119].b[0~7]	bit (byte unit group)	0, 1
		dib[0~119]	signed 1-byte integer	-128 ~ +127
		diw[0~118]	signed 2-byte integer	−32,768 ~ +32,767
		dil[0~116]	signed 4-byte integer	−2,147,483,648 ~ +2,147,483,647
		dif[0~116]	signed 4-byte real number	3.4E+/-38 (7 significant figures)

In do, dob, dow, dol, and dof, the suffixes b, w, l, and f mean “byte,” “word,” “long,” and “float,” respectively, and all are signed values. These are not separate memory spaces and represent the same 960-byte space just with different data types. For example, do[1~16], dob[1~2], and dow[1] are all the same output signals.

bit	do0 ~ do7	do8~do15	do16~do23	do24~do31	…
byte	dob0	dob1	dob2	dob3	…
word	dow0		dow2		…
long	dol0				…
float	dof0				…

If a value is assigned to an output variable that starts with “do,” I/O signal output will be performed. The I/O signal currently being inputted can be acquired by reading the input variable value that starts with “di.” The do variable can be read and written, but the di variable can only be read.

The FB object name can be omitted as follows.

	do notation	fb.do notation
fb0	do0 ~ do959	fb0.do0 ~ fb0.do959
fb1	do960 ~ do1919	fb1.do0 ~ fb1.do959
fb2	do1920 ~ do2879	fb2.do0 ~ fb2.do959
fb3	do2880 ~ do3839	fb3.do0 ~ fb3.do959
fb4	do3840 ~ do4799	fb4.do0 ~ fb4.do959
fb5	do4800 ~ do5759	fb5.do0 ~ fb5.do959
fb6	do5760 ~ do6719	fb6.do0 ~ fb6.do959
fb7	do6720 ~ do7679	fb7.do0 ~ fb7.do959
fb8	do7680 ~ do8639	fb8.do0 ~ fb8.do959
fb9	do8640 ~ do9599	fb9.do0 ~ fb9.do959

6.1.2 Examples

Refer to the following examples of usage.

```
do2=1                  # Turns on the bit output value of number 0 of fb0
fb2.dob3=0b00001111    # Designates the 3rd byte output value of fb2 as a binary bit string
fb[4].dob1=0x0F        # Turns on the lower 4 bits of the 1st byte output value of fb4, and turns off the upper 4 bits
var work_no=fb9.dib3    # Assigns the 3rd byte input value of fb9 to the work_no variable
if fb5.di43 then *err  # Branches to the *err label when fb5.di42 is turned on
for idx=21 to 29
  fb3.do[idx]=1        # Turns on all output signals do21 ~ do29 of fb3
next
fb2.do3=fb2.do7=fb2.do11=1   # Turns on 3rd, 7th, and 11th output signals of fb2 at once
```

6.2 ENet Module: Ethernet TCP/UDP Communication

Using the general-purpose Ethernet port of the Hi6 Controller makes it possible to transmit or receive a string with an external device through Ethernet TCP or UDP communication. It is required to create an ENet object after importing the ENet module to use this function, as shown below.

```
import enet

var udp=enet.ENet("udp")
```

In the following example, the selection of protocol is needed by passing "udp" or "tcp" as a parameter of the ENet constructor. The default is "udp," so like in the example, it can be omitted when UDP communication is performed.

```
var udp=enet.ENet()
```

Communication must be performed in the following order:

1. Create an ENet object with the constructor.
2. Set an IP address and port number with the member variable.
3. Open the communication connection with the open member procedure, and check the state with the state() member variable.

   (In the case of TCP communication, the connect procedure should also be performed after opening).
4. Perform transmission/reception with the send and receive member procedures.
5. Close the communication connection with the close member procedure.

6.2.1 Constructor

Description	It creates an Ethernet object and returns the reference.		
Syntax	ENet(<protocol>)		
Parameter	protocol	"tcp" : TCP communication "udp" : UDP communication	If omitted, it will be recognized as "udp."
Return value	Reference of the created object		
Exampleof usage	enet0 = ENet() var tcp = ENet("tcp")		

6.2.2 Member Variables

Variable name	Data type	Description
ip_addr	String	Allows reading/writing Designates or acquires the IP address of the communication counterpart Applicable only when calling the open statement
rport	Number	Allows reading/writing Designates or acquires the port number of the communication counterpart (Remote) Applicable only when calling the open statement
lport	Number	Allows reading/writing Used in UDP communication and ignored in TCP communication Designates or acquires the port number of the controller itself (Local) The default value is 0 (if not designated), in which case the controller's port number will be automatically created. Applicable only when calling the open statement

6.2.3 Member Procedures

Description	Opens a connection for Ethernet TCP or UDP communication
Syntax	〈ENet object〉.open
Example of usage	enet_to_sensor.open

Description	Performs a connection for Ethernet TCP communication
Syntax	〈ENet object〉.connect
Example of usage	enet_to_sensor.connect

Description	Close the connection for Ethernet UDP communication
Syntax	〈ENet object〉.close
Example of usage	enet_to_sensor.close

Description	Transmits the values to the set Ethernet object		
Syntax	〈ENet object〉.send 〈value〉, 〈value〉, …		
Parameter	Value	Dadta balue to be output. Arguments separated by commas will be printed separated by spaces.	All data types
Example of usage	enet_to_sensor.send "rob:", 10, ", command:"+cmd, "＼n"		

Description	Receives the values to the set Ethernet object
Syntax	〈ENet object〉.recv 〈variable〉[, 〈waiting time〉]

Parameter	Variable	A variable to which the received string is to be passed	
	Waiting time	Time of time-out	msec
Example of usage	enet_to_sensor.recv msg, 5000		

6.2.4 Member Function

Description	Returns the state of the Ethernet object	
Syntax	<ENet object>.state()	
Return value	1	Connected (In the case of UDP, just opening it will be considered a connection. In the case of TCP, connecting after opening will be considered a connection.)
	0	Not connected
	-1	Failed to create the Ethernet socket
	-2	Failed to bind the Ethernet device
Example of usage	ret = enet_to_sensor.state()	

6.2.5 Examples of TCP and UDP Communication

```
 import enet
 global msg
global enet0=enet.ENet() # ENet("tcp") in case of TCP communication

# port no. 49152~65535 contains dynamic or private ports
enet0.ip_addr="192.168.1.172"
enet0.lport=51001 # necessary only in case of UDP communication
enet0.rport=51002

enet0.open
enet0.connect # necessary only in case of TCP communication
print enet0.state() # normal if it is 1
enet0.send "hello, "+"udp", 300, "₩n"

enet0.recv msg, 8000 # wait for 8 seconds
print msg
delay 1.5
enet0.close
print enet0.state() # normal if it is 0
delay 1.5
end
```

6.3 Http_Cli Module: HTTP Client

Using the general-purpose Ethernet port of the Hi6 Controller makes it possible to access remote web services to receive HTTP services.

To use this function, it is required to create an HttpCli object after importing the http_cli module, as shown in the following example.

```
    import http_cli

    var cli=http_cli.HttpCli()
```

After the HttpCli object is created, it must request a service by calling the get, put, post, and delete member procedures.

The HttpCli object has a property named “body.”

When a get service is requested and a response is received successfully, the remote server’s data will have the body property. The body property value can be a string, number, array, or object. When requesting the put service, it is required to assign the data to be transmitted to the body property in advance.

When requesting the post service, it is required to assign the data to be transmitted to the body property in advance, and the data sent as a response from the remote server is to be stored in the body property.

The delete service does not use the body property.

6.3.1 Constructor

Description	It creates an HttpCli object and returns the reference.
Syntax	HttpCli()
Return value	Reference of the created object
Example of usage	var cli = HttpCli()

6.3.2 Member Variables

Variable	Data type	Description
body	All types are possible	It is required to put the data, which is to be loaded on the put and post requests, in advance. Responses for the get and post requests will be stored.

6.3.3 Member Procedure

Description	Requests the HTTP get service The response data is to be received to the body property.
Syntax	〈HttpCli object〉.get 〈URL string〉
Example of usage	var domain="http://192.168.1.200:8888" cli.get domain+"/setting/max_torque"

Description	Requests the HTTP put service It is required to assign the data, which is to be transmitted, to the body property in advance.
Syntax	〈HttpCli object〉.put 〈URL string〉
Example of usage	var domain="http://192.168.1.200:8888" cli.body=500 cli.put domain+"/setting/max_torque"

Description	Requests the HTTP post service. It is required to assign the data, which is to be transmitted, to the body property in advance. The response data is to be received to the body property.
Syntax	〈HttpCli object〉.post 〈URL string〉
Example of usage	var domain="http://192.168.1.200:8888" cli.body={ name: "WORK #32", color: "green", state: "OK" } cli.post domain+"/display/update"

Description	Request the HTTP delete service. The body property is not be used.
Syntax	〈HttpCli object〉.delete 〈URL string〉
Example of usage	var domain="http://192.168.1.200:8888" cli.delete domain+"/items/3"

6.3.4 Examples of HTTP Client Communication

```
 import http_client
 var cli=http_client.HttpClient()

 var domain="http://192.168.1.200:8888"

# get
 cli.get domain+"/device/direction"
 print cli.body.ry

# put
 cli.body.ry=90
 cli.put domain+"/device/direction"

# post
 cli.body=={ name: "WORK #32", color: "green", state: "OK" }
 cli.post domain+"/display/update"

# delete
 cli.delete domain+"/items/3"

end
```

Customer support

Contact: 1670-5041 | Email: robotics@hyundai-robotics.com

Operating hours: Weekdays (Monday–Friday) 09:00–18:00 | Closed on weekends and holidays

For any inquiries about our products or services, please contact our customer support team.

GRC: 477, Bundangsuseo-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13553, Korea

Daegu: 50, Techno sunhwan-ro 3-gil, Yuga-eup, Dalseong-gun, Daegu, Republic of Korea

Ulsan: Room 201-5, Automobile & Shipping Technology Hall, 21, Maegoksaneop-ro, Buk-gu, Ulsan, Republic of Korea

Joongbu: 161, Songgok-gil, Yeomchi-eup, Asan-si, Chungcheongnam-do, Republic of Korea

Gwangju: Room 101, Building B, 170-3, Pyeongdongsandan-ro, Gwangsan-gu, Gwangju, Republic of Korea

ARS 1588-9997 | 1 Robot Sales 2 Service Sales 3 Consultation for Purchase 4 Customer Support 5 Inquiry for Investment 6 Inquiries for Recruitment and General Matters

www.hyundai-robotics.com