NEXCOBOT
Open Robot & Machines

NexMotion NRPL
Programming manual

Version: 1.2
Date: 2019-08-29

NEXCOBOT Inc.
www.nexcobot.com

Copyright Statement and Disclaimer

The contents contained in this document are the proprietary property of NexCOBOT Co., Ltd. (NexCOBOT hereafter) and is subject to the protection of intellectual property law (including, but not limited to the Copyright Act). The use of any material in relation to this document without the prior authorization of NexCOBOT is considered infringement. Without the written approval of NexCOBOT in advance, this document or any part of it shall not be photocopied, sold, distributed, modified, published, stored or otherwise used.

To keep this document and its contents correct and complete, NexCOBOT reserves the right to change or revise the document at any time without further notification.

Operating machine or equipment has a certain level of danger. It is the user's responsibility to pay special attention and have safety protection in place before operating any machine or equipment. NexCOBOT shall not be held for any and all direct or indirect damage or loss to the equipment mentioned in this document due to the use for a purpose other than the intended.

Copyright © 2019 NEXCOBOT Inc. All Rights Reserved.

Revision History

Rev.	Description
1.0	First released.
1.1	Add Pointer & Array chapter
1.2	Add Structure chapter

Copyright © 2019 NEXCOBOT Inc. All Rights Reserved.

Content

Copyright © 2019 NEXCOBOT Inc. All Rights Reserved.

Copyright © 2019 NEXCOBOT Inc. All Rights Reserved.

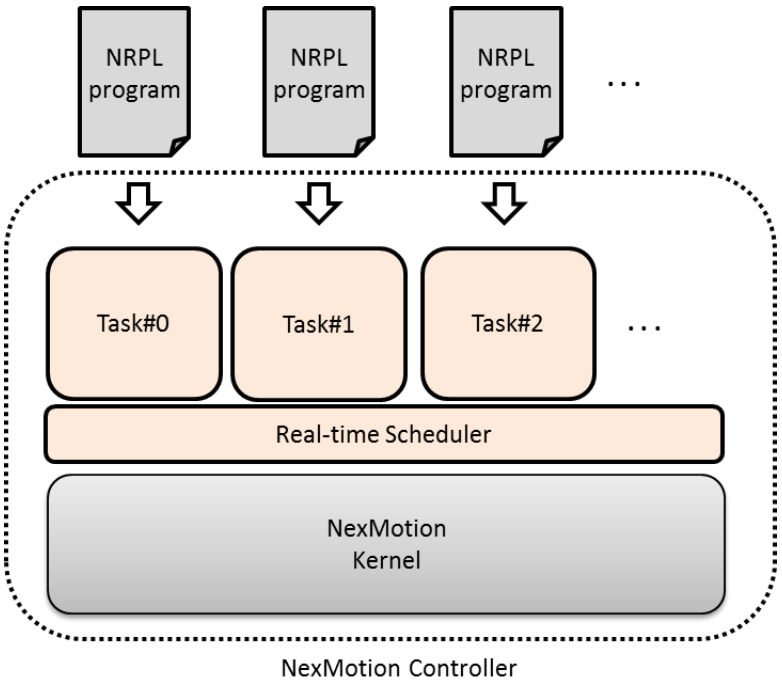
1. Introduction to NRPL

NRPL (NexMotion Real-time Programming Language) is a high-level programming language developed for industrial automation and robot motion control. Its features are as follows:

- Grammar is C language liked, easy to get started
- Support multi-tasking
- Task execution is real-time and time-deterministic
- Build-in motion objects: “GROUP” objects and “AXIS” objects
- Motion objects have rich and complete instructions
- Overloading function call supported for build-in instructions
- Support variables, arrays, structures and pointer
- Support user define subroutines
- Support math library

1.1. System Architecture

NRPL program is an independent execution of the program can be loaded into the task of the NexMotion controller. The NexMotion controller support multi-tasking. All tasks are scheduled by “real-time scheduler”. Basically, each task has the same priority. The CPU time is allocated by the scheduler depending on the amount of logical operations in the task. And the execution time of the same NRPL program is instantaneous and fixed, so the NRPL program has time determinism.



1.2. Motion objects

The NRPL has built-in motion objects, including “GROUP” objects and “AXIS” objects. The motion objects provide a series of member functions to control the object for motion control applications. When the controller is started, it automatically creates motion objects according to the actual number of groups (robots) and the number of single axes in the system.

Objects are automatically generated by the system in the form of an array. In the NRPL program, Objects can be used without additional declaration. The syntax is as follows:

```
GROUP[ 0 ].PTP( G0P1 ); //Group#0 do PTP motion
GROUP[ 1 ].PTP( G1P1 ); //Group#1 do PTP motion
```

Detailed descriptions for built-in instructions can be found in the " *NexMotion NRPL Instruction Manual* "

1.3. Overloading

The NRPL built-in instructions have "overloading" feature that allows users to flexibly use them based on application needs.

For example, a overloading function “foo()” whose definition is as follows :

```
void foo( para1,  para2  [, para3]   [, para4 ] );
```

This example foo() has four parameters are “para1” , “para2”, “ Para3” and “para4”. Para1 and para2 are necessary parameters, just like general function calls, they must be given values or variables in order. [, para3] and [, para4] with brackets are indicated as optional parameters and can be ignored depending on the application requirements.

The following examples are legal for the foo() function:

```
foo( para1 , para2 );
foo( para1, Para2, Para3 =. 5 );
foo( para1, Para2, para4 = 10 );
foo( para1 , para2, para3=5, para4=10 );
foo( para1 , para2, para4=10 , para3=5 );
```

The following is an example of a practical instruction for a GROUP object.
The GROUP has a motion instruction PTP prototype as follows:

```
PTP ( TargetPos, [VP], [TL], [BS], [Z], [ABORT], [CONT] ], [OW] )
```

The “TargetPos” (target position) is the necessary input, and the [VP] (speed percentage), [TL] (tool number) and [BS] (base number)... parameters are optional parameters. The following examples are legal usage of PTP () instruction:

```
GROUP[0].PTP( P0 ); // PTP moves to P0, other parameters are according to system parameters

GROUP[0].PTP( P1, VP=50 ); //PTP moves to P1, speed (VP) is set to 50% of system speed

GROUP[0].PTP( P2, VP=50, Z=30 ); //PTP moves to P2, speed = 50%, zone = 30 mm
```

For details on the NRPL command parameters, refer to the " *NexMotion NRPL Instruction Manual* ".

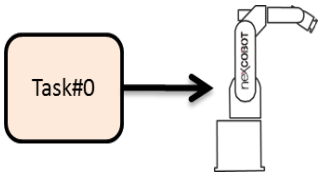
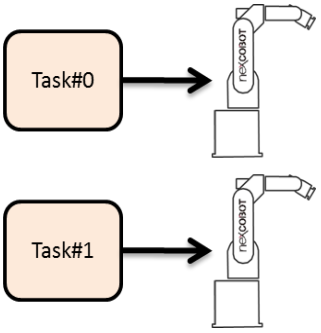
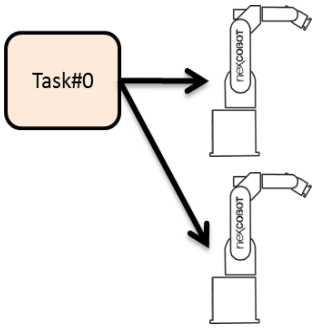
Note! User-defined function does not support overloading function call.

1.4. Tasks and motion objects

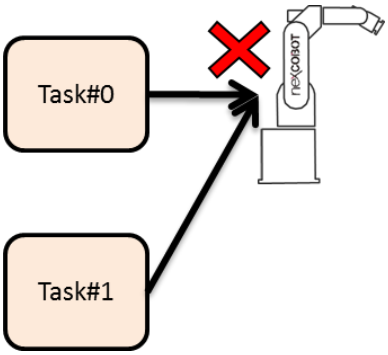
The relationship between NRPL tasks and objects has the following rules:

- One task can control one or more motion objects
- A motion object can only be controlled by one task
- A task can set a motion object as a default object

The following application scenarios are allowed:

		
1 task controls 1 object	2 tasks control 2 objects separately	1 task controls 2 objects

The following application scenario is illegal:

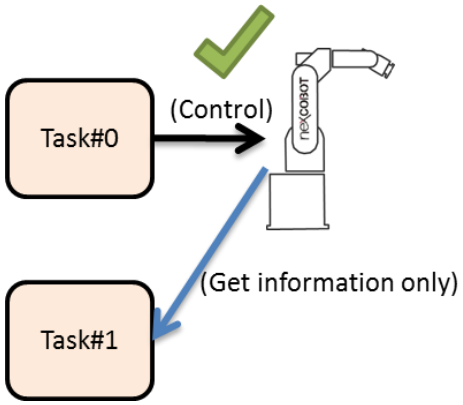


Two tasks control same GROUP object

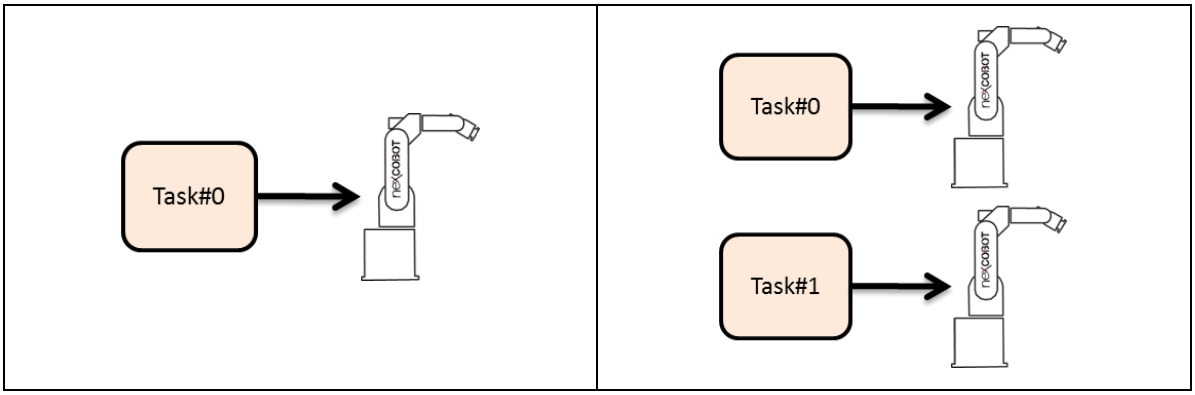
If two tasks control the same motion object at the same time, it is easy to cause a race

condition, which may cause dangerous malfunction. If this is the case, the controller will enter an error state.

If one task has control of one object, but another task tries to read the same object (for example, to get the position), as shown in the following figure, this situation is legal:



It is generally recommended that the best task assignment is: One task is responsible for controlling only one motion object, as shown below:



1.5. **Default object**

An NRPL program can have a default GROUP object. When a GROUP is set as a default object to a NRPL program, the object instruction call can be simplified, as follows:

A NRPL program:

```
void main()
{
```

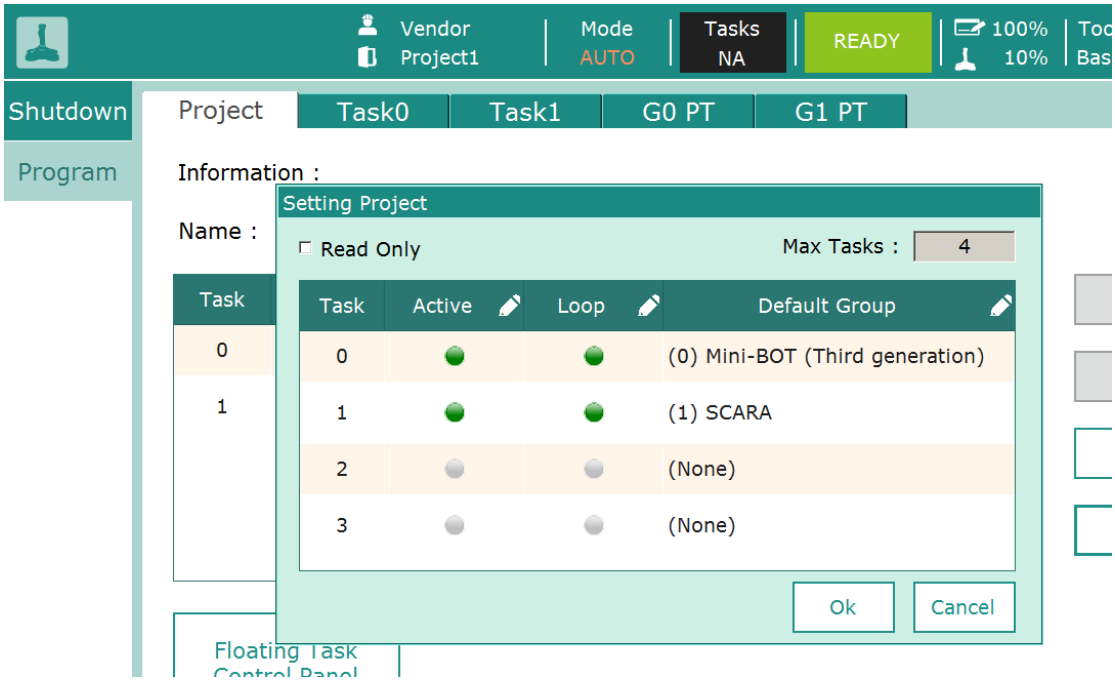
```
    GROUP[0].PTP( P0 );
    GROUP[0].PTP( P1 );
    GROUP[0].PTP( P2 );
    GROUP[0].LIN( P3, TL=1, BS=1 );
    GROUP[0].LIN( P4, TL=1, BS=1 );
}
```

If GROUP[0] is set as the default object, the above NRPL program can be simplified as follows:

```
// Default using GROUP[ 0]
void main()
{
    PTP( P0 );
    PTP( P1 );
    PTP( P2 );
    LIN( P3, TL=1, BS=1 );
    LIN( P4, TL=1, BS=1 );
}
```

When the controller compiles the NRPL program, the default object GROUP[n] is automatically added to the program, so when the task executes the NRPL instruction, for example, executing PTP(P0), the instruction can be executed with the correct object, so when only one group (Robot) is controlled by controller (Typical industrial robot system applications), the program can be greatly simplified.

The way to set the default object can be set or viewed in the NexMotion controller user interface software - NexTPUI. In general, when you use NexTPUI to create a new project, the GROUP object in the system will be automatically set to the task, so users no need to do configuration additionally.



NexTPUI default group setting screen shot

Complete details of NexTPUI operation, please refer to the " NexTPUI User Manual".

2. Program structure

NRPL program itself no matter how large or small, are a function (function) and variable (variable) thereof. The statements in the function are used to describe the application flow and calculation work. Variables are controls used in memory to store calculated values or processes. The following example is the basic framework of the NRPL program :

```
void main() //NRPL Program Entry Point
{
```

```
    // Variables deceleration in beginning...

    // main program...
}
```

2.1. Entry point

When the NRPL program runs, there will be a function entry point. The function of the entry point in the syntax is as follows:

```
void main()
{
    [ Narrative sentences ( statements )]
}
```

The function name “main” must be all lowercase, and not take any parameters, and void representatives no return value.

2.2. Statement

Statements Subject arithmetic expression (expression) of the composition, is defined as follows :

Expression (expression): Suppose two variables are added, such as A + B , then this expression becomes Expression.

Narrative sentence (Statements): will be added later expression semicolon (;) , such as A + B; , is referred to statement (statement)

2.3. Operator

There are a variety of arithmetic symbols in the NRPL syntax, such as the multiplication operator is * , the minus operator is - , and such operators can be described in the third chapter.

2.4. Comments

Can NRPL successful program note text annotation to increase the readability of the program, in the form of notes can be divided into two types :

1. Single-line annotation : use double slash / / to indicate the text after // until the end of a single line
2. Block annotation : start with /* , end with */

Here's an example :

```
Void main()
{

    // This is a single line comment

    /*
        Multi-line comments
    Block comments...
    */
}
```

3. Variables and expressions

3.1. Variable

The name must begin with the letters (A~Z , a~z) followed by the letters (A~Z) or numbers (0~9) . In addition, the underline _ (underline) is also a letter, which is used to increase readability when the variable name is very long (note : no white space !) . The uppercase and lowercase letters are completely different names in the compiler, so NEXCOBOT and nexcobot are two different names. In addition, the **keywords (keyword)** not be used as variable names, such as if, else, switch, I32_T ... etc. have been defined as a reserved word.

3.2. Data type

The basic data types are as follows :

Keyword	Size (byte)	Numerical range
F64_T	8	1.7E +/- 308 with 15 digits of precision
U32_T	4	0 ~ 4294967295
I32_T	4	-2147483648 ~ 2147483647
U16_T	2	0 ~ 65535
I16_T	2	-32768 ~ 32767
U8_T	1	0 ~ 255
I8_T	1	-128 ~ 127

3.3. Constant value

When an integer (eg 1234) is input , it is treated as an I32_T type, and when a floating point number (eg 12.34) is input , it is regarded as an F64_T type.

3.4. Variable declaration

All variables must be declared before use . The wording is first named data type, followed by one or more variable names of the type, as follows :

```
I32_T lower;
I32_T upper;
F64_T value;
```

The above example is a data type followed by a variable name, and then multiple variable names are written as follows :

```
I32_T lower, upper, step;
```

When the variable is announced, the initial value can be given by the way. After the announcement, an equal sign is added to the initial value or the expression, as follows :

```
I32_T lower = 0 ;
I32_T upper = 1 + 2;
```

When the code that declares the variable is executed, the system will configure the memory block of the type size to provide subsequent code access.

The life cycle of a variable (region, global) is used to regulate when a variable can be used.

Regional variables :

The variable is declared in the function. This variable belongs to the area variable and can only be used within the function. When the function is executed, the memory configuration of the area variable is released. In addition, **note that the area variable should be inserted before the expression, and the behavior of declaring variables should not be interspersed between the expressions** .

The legal declaration is as follows:

```
void foo()
{
    I32_T value_1 = 1;     // Deceleration
```

```
    I32_T value_2 = 2;     // Deceleration
    I32_T result;          // Deceleration

    Result = value_1 + value_2 ;   //Expression
}
```

The illegal declaration is as follows:

```
void foo()
{
    I32_T result;
    Result = 0;   //Expression

    I32_T value_1 = 1 ; /  / Illegal Deceleration
    I32_T value_2 = 2 ; /  / Illegal Deceleration
    Result = value_1 + value_2;
}
```

Global variables:

In addition to declaring variables as a function of global variables, can in each one inside a function, and only when the end of the program, will release the memory, that is the end of the life cycle. Also, please note that the declaration of global variables cannot be after the function before the function is declared. The declaration of legal global variables is as follows :

```
I32_T lower; //    announce
Void foo() {...} //   function
```

Unlawful global variable declarations are as follows :

```
I32_T lower; //   announce
Void foo() {...} //   function
I32_T upper; //   Declaration is illegal because after the function
Void bar() {...} //  function
```

3.5. Arithmetic Symbols

The operator (also called arithmetic symbol) includes five binary operators: +, -, *, /, % and two unit operators (unary operators): +, - .

Binary operator:

Operator	Description	Operational metatype
+	plus	Integer , floating point
-	Less	Integer , floating point
*	Multiply	Integer , floating point
/	except	Integer , floating point
%	Remainder	Integer

Note that the integer division will discard all decimals, for example, the 5/2 answer is 2 . Since 5 and 2 are integers , the answer will also be an integer. Although the correct answer is 2.5 , the integer type will discard the decimal point, so the result is 2 . In addition, operator % is not available for floating point numbers.

Unit operator:

Operator	Description	Operational metatype
+	Positive value	Integer , floating point
-	Negative value	Integer , floating point

The unit operator - and the binary operator - are not the same, for example, 8-3 here - represents subtraction, and -3 is represented as an integer with a negative sign.

3.6. **Relational and logical operators**

Binary relational operator:

Operator	Description	Operational metatype
>	more than the	Integer , floating point
>=	greater or equal to	Integer , floating point
<	Less than	Integer , floating point
<=	Less than or equal to	Integer , floating point
==	equal	Integer , floating point
!=	not equal to	Integer , floating point

Unit relational operator

Operator	Description	Operational metatype
!	negative	Integer , floating point

When the expression 8 > 3 is established, it will return 1 , otherwise it will return 0 . The unit relational operator ! is to convert non- zero operands to 0 and 0 to 1 .

Logical relational operator

Operator	Description	Operational metatype
&&	The left and right operands are all greater than 1 , which is true.	Integer , floating point
||	One of the left and right operands is greater than 1 , it can be established	Integer , floating point

Note that the type of result obtained by the relational and logical operators is I32_T .

3.7. Type conversion

In the calculation formula, if the types of operation elements on both sides of the operation symbol are different, the calculation will be converted into the **same type** according to some rules . In general, the narrower definition domain is turned into a wider one. Many operators will make the operands convert, and use the following rules to determine the result of the operation, called the usual arithmetic conversions:

Step 1. If the operand is F64_T , the other will be forced to F64_T , otherwise enter step2
Step 2. If the operand is U32_T , the other will be forced to U32_T , otherwise enter step3
Step 3. If the operand is I32_T , the other will be forced to I32_T , otherwise enter step4
Step 4. If the operand is U16_T , the other will be forced to U16_T , otherwise enter step5
Step 5. If the operand is I16_T , the other will be forced to I16_T , otherwise enter step6
Step 6. If the operand is U8_T , the other will be forced to U8_T , otherwise

enter step7
Step 7. If the operand is I8_T , the other will be forced to I8_T .

Expression 5 / 2.0 , wherein 2.0 part of F64_T typed, and 5 to I32_T , the above rules apply, will put 5 is converted into F64_T , a result 2.5 of F64_T type. Note that if the floating point number is converted to an integer , the decimal is ignored and the values are inconsistent.

The type conversion can also be explicitly written in the expression by the cast operator (cast operator) , the format is as follows :
(type-name) expression

Suppose expression (F64_T) 5 + (F64_T) 2 , where 5 is I32_T typed, to apply casts (F64_T) , so the result is 5.0 , and the result of this calculation formula 7.0 in (F64_T) type.

3.8. Increment and decrement operators

The increment operator ++ is used to increment the operand by one and the decrement operator -- to decrement the operand by one . Operator + and - can be written before the variable (such as n ++) after or variables (such as + n) , both of which will be n value plus 1 , but + n represents the first n value by 1 in The value is taken, and n++ means that the first value is used to do the addition of 1 . For example, if the n value is 5 , then
X = n+ + ;
Will get the X value to get 5 but
X = ++ n ;
Then get the X value to get 6 ,

Both an operator can only be used for variables, so as (i + j) ++ this formulation are unacceptable.

3.9. Bit operator

Provide 6 kinds of bit operators, **please note the operands that can only be used for integers** .

Bit operator

Operator	Description	Operational metatype

&	Bitwise AND	Integer
|	Bitwise OR	Integer
^	Bit-by-doing exclusive or	Integer
<<	Shift to the left	Integer
>>	Shift to the right	Integer
~	Take 1 complement, for the unit operator	Integer

AND bit operation rule : when only two bits are 1 , the result will be 1

	0	1
0	0	0
1	0	1

Let n be U32_T type, operator & used to make bit mask, as follows :

```
n = n & 255
```

OR bit operation rule : as long as one of the bits is 1 , the result will be 1

	0	1
0	0	1
1	1	1

The n of bit8 ~ bit31 is cleared to 0 . Conversely, the operator | is used to set the bit to 1 , for example :

```
n = n | 255;
```

The n of bit0 ~ bit7 is set to 1 , the rest of the bit unchanged.

XOR bit operation rule : two bits are set to 1 at the same time, and set to 0 when they are the same

	0	1
0	0	1
1	1	0

The operators << and >> respectively shift the left operand to the left or right, and the right operand decides to shift a few bits , and the value of the right operand must

be positive. For example, n << 2 shifts n by two bits to the left and 0 to the right , which is equivalent to multiplying x by 4 .

The operator ~ changes its operand element 1 to 0 , 0 to 1 , that is, takes 1 complement. For example, let n be U8_T type and the value is 0 .

n = ~n;

The result n is 255 ;

3.10. Setting operators and expressions

An expression like this :

n = n + 2;

The variable n to the left of the equal sign appears immediately on the right and can be written in a concise form :

n += 2;

Herein + = is the assignment operator (assignment operator) one.

Operator	Description	Operational metatype
=	Assign	Integer , floating point
+=	plus	Integer , floating point
-=	Less	Integer , floating point
*=	Multiply	Integer , floating point
/=	except	Integer , floating point
%=	Remainder	Integer
&=	AND	Integer
^=	XOR	Integer
|=	OR	Integer
<<=	Left displacement	Integer
>>=	Right displacement	Integer

If both expr1 and expr2 are arithmetic , then

Expr1 op= expr2

Just equal

Expr1 = (expr1) op (expr2)

Only the former expr1 is only counted once. Note that brackets clamp expr2 to indicate that it is a self-contained expression, for example :

x *= y+1

Is equal to

x = x * (y+1)

Instead of

x = x * y + 1

3.11. **Conditional expression**

Conditional expressions use a ternary operator consisting of a question mark and a colon. The format is as follows :

Expr1 ? expr2 : expr3

First calculate expr1 , if its value is 1 (for true) , then calculate expr2 , and the value of expr2 is treated as the value of the conditional expression. Otherwise (expr1 is 0) , expr3 is evaluated as the value of the conditional expression. Note that expr2 and expr3 will only calculate one of them. Examples are as follows :

```
z = (a > b ) ? a : b ; /* z=max(a, b) */
```

You can put a larger value of a or b into z . If the types of expr2 and expr3 are different, the result is determined according to the rules of general arithmetic transformation (Section 2.2.7) . If a is F64_T and b is I32_T , the result type is F64_T .

3.12. **Operational symbol precedence**

calculating signs	Calculation order
( ) , [ ], ->, .	Left to right
!, ~, ++, --, +(positive value), -(negative value), *(indirection), &(take address),	Left to right
*(multiply), /(division), %(take the remainder)	Left to right

+(plus), -(minus)	Left to right
>>, <<	Left to right
<, <=, >, >=	Left to right
==, !=	Left to right
& (bitwise AND)	Left to right
^ (bitwise XOR)	Left to right
| (bitwise OR)	Left to right
&&	Left to right
||	Left to right
? :	Right to left
=, +=, -=, /=, %=, &=, ^=, |=, <<=, >>==	Right to left

4. Control process

4.1. Statement and block

The expression such as x = 0 or x++ is followed by a semicolon (;) to become the statement . The semicolon in the grammar is the termination symbol of the Statement .

Braces {} are used to group Statement sandwiched becomes Compound Statement , or segments, called (Block) , may be considered in a single grammar Statement . For example, braces enclose all the statements of the entire function as an obvious example. After the end of the section, there is no need to add a semicolon after the brackets.

4.2. if – else

Use if-else to express decisions, the syntax is as follows

```
If ( expression )
  Statement_1
Else
  Statement_2
```

The else part is optional. Execute the expression first , if it is non- zero

(representing true) , execute statement_1 , otherwise
execute statement_2 . Although statement_1 and statement_2 are only capable of a statement , but the aforementioned compound statement can be, but as a statement to use, so the use of braces can write a group statement , as follows :

```
If (expression)
{
    Statement
    Statement
    Statement
} else
{
     Statement
     Statement
}
```

Further else part optional, so the nest -like if statement (i.e. if statement contains another if statement) if a dispense else partial, so that it will read the code confusion, the following examples :

```
If (n > 0)
   If (a > b)
     z=a;
   Else
     z=b;
```

In this example there are two if but only one else , so else of course, is closer to belong to him if , therefore we will else with the second one if justified, to represent else belongs to the second one if , even deliberately else first
The if alignment does not change its meaning. If you want to make else belong to the first if , then use braces, as follows :

```
If (n > 0)
{
      If (a > b)
      z=a;
} else
   z=b;
```

Therefore, when writing a program, you must develop the habit of using a parenthesis when you have a nested if .

If-else can also be written in a variety of decisions, the general if-else is written as follows :

```
If (expression)
  Statement_1
Else
  Statement_2
```

And a variety of decision-making (else-if) , is to replace statement_2 with if-else , as follows :

```
If( expression)
  Statement_1
Else
  If( expression)
    Statement_2
  Else
    Statement_3
```

And so on, if you need to replace the statement_3 with if-else , and the above code style will make the reading code confusing, so it can be organized as follows :

```
If (expression)
  Statement_1
Else if (expression)
  Statement_2
Else
  Statement_3
```

The various expressions are in the order of top-down. As long as one expression is true , the corresponding statement is executed , and the entire structure is terminated. If all the expressions are not true, the statement of the else part is executed .

4.3. switch

The statement of multiple decision making in another place is switch . He tests

whether the value of the expression is the same as one of many constants. According to this, the corresponding statement is executed , and the writing is as follows :

```
Switch (expression)
{
    Case const-expr : statement
    Case const-expr : statement
    Default : statement
}
```

During which you can have a lot of case , every one case must have after **a constant and is an integer** , in addition to a man named default: , but only a default . Performing switch value calculation formula of, sequentially from top to bottom and then compare case constant value after this, if equal, from the case after the statement execution continues, if all does not meet the case , the implementation of default of statement . Please note that if the default misspelled will be treated as ordinary can goto bit standard .

Execute switch when, if found for the case will execute it, and do not jump out switch block, the following lines :

```
Switch (expression)
{
Case 1 :
      Statement_1
Case 2 :
      Statement_2
Case 3 :
      Statement_3
Default :
      Statement_4
}
```

Suppose expression calculated value of 2 , according to the above code will be executed . 3 th Statement , respectively, and statement_2, statement_3, statement_4 . To end the switch block after executing statement_2 , add the break keyword to indicate the end of the switch block, as follows :

```
Switch (expression)
{
Case 1 :
      Statement_1
   Break ;

Case 2 :
      Statement_2
   Break ;

Case 3 :
      Statement_3
   Break ;

Default :
      Statement_4
}
```

4.4. for

for return to ring format syntax is as follows :

```
 For (expr1; expr2; expr3)
   Statement
```

Syntax for brackets expr1, expr2, expr3 are expression (note therebetween separated by a semicolon) . The operation process is as follows :

When expr2 is false , the for loop is ended . Further, for the exprl, expr2, expr3 , Jieke omitted, but expr2 is omitted, the test results would be considered to true , formed endless return ring, the following :

```
For ( ; ; )
{
      Statement
}
```

To jump off the loop, use break .

4.5. while

while back loop in the following format :

```
While (expression)
    Statement
```

The operation process is as follows :
Only when the expression is false when, before the end of the back loop. You can also use BREAK , to end while back loop. The infinite loop of while is written as follows :

```
While( 1)
      Statement
```

4.6. do – while

do-while back loop in the following format :

```
Do
      Statement
While (expression)
```

Do-while and the while behavior are just the opposite. While the first is to judge the expression to be the statement , and the do-while is to make the statement before the expression is evaluated . The do-while operation flow is as follows :
You can use BREAK , to end while back loop.

4.7. break and continue

Sometimes it is convenient to leave in the loop, break is to provide early, or , do-while to leave early, or escape the switch block. Note however that break leaving only the innermost back loop or switch (ie break where that layer) . For example, break in the bilayer for the

```
I32_T m, n;
I32_T result = 0;
```

```
For (m=0 ; m < 10 ; m++)
{
      For( n=0; n<10 ; n++)
      {
      If( n==5)
            {
        BREAK; //     escape the second one for back loop
      }
      Result += 1;
      }
}
```

The second one for back inside the circle, to perform n == 5 , the will come out, thus double back lap will result + = 1 performed 50 times.

And break Correspondingly the Continue , it does not leave the circuit, but this time to immediately stop the loop and jumps to test back to the local loop, in order to decide whether the next back to the circle. In for back loop is to jump expr3 place. continue only for back circle, does not apply Switch . Examples are as follows :

```
I32_T m;

For( m=0 ; m <50 ; m++ )
{
      If ( ( m % 2 ) == 0 )
      {
      Continue; / /   ignore m is a multiple of 2
   }
   Statement
}
```

In the above example, the statement is only executed when m is not a multiple of 2 .

4.8. goto and label

Goto and label can only be used within a function, the format is as follows

```
Void main()
{
Goto label;
      Statement


Label :
   Statement
}
```

Goto should be followed by the label name, followed by a colon with a label , as shown in the following example

```
   Goto ERROR;
   Statement_1
   Statement_2
ERROR:
   Statement_3
```

When the goto ERROR is executed , it will immediately jump to the ERROR and start executing statement_3 without executing statement_1, statement_2 .

In theory, goto is unnecessary, but in some cases it is suitable to use goto . The most common is in very deep nested back to forgo treatment circle, for example, two or more immediately end more layers of back lap, then break it quite difficult to use, so the break only to jump off a layer of back ring, So the wording is as follows :

```
For( expr1 ; expr2; expr3 )
{
      For( expr3; expr4; expr5 )
      {
            If (disaster)
         Goto ERROR;
   }
}
```

ERROR:
    Statement

The label can be attached to any statement before the goto can jump anywhere in the function. Programs that use goto are usually not only difficult to read but also difficult to maintain, but can be used as appropriate.

5. Function

Function (function) is an independent program units. Using functions can break down large computational work into smaller jobs, and it's easy to use other people's or previously written functions without having to rewrite them. The appropriate function hides the details of the operation. The person using the function does not have to know the details, making the whole program clearer and easier to modify.

5.1. User-defined function

The format of the function is as follows :

```
Return-type function-name ( parameter declarations )
{
  Statements
}
```

The return type can be F64_T, I32_T...., etc., and the void type means that this function has no return value. Except for void , all other types must return the corresponding type in the function. The format of the return is as follows :

```
Return expression;
```

In which expression of the type required and return type same as or deemed illegal. Parameter declarations are the parameters passed in using the function, and the format is as follows :

Type variable-name

The incoming parameters can be multiple, separated by commas ' , ' , as in the following example :

```
I32_T add (I32_T a, I32_T b)
{
      Return a + b;
}
```

In addition, parameter declarations can also be written, as follows :

```
I32_T get_ value()
{
      Return 10;
}
```

A program is a collection of variables and functions. Communication between functions can be done through parameters, global variables , and function values. The following sample code for a fee type sequence (Fibonacci Sequence) ,

```
I32_T fib( I32_T n)
{
   If( n==0 ) return 0;
   If( n==1 ) return 1;
   Return (fib(n-1) + fib(n-2));
}

Void main()
{
      I32_ T   value ;
   Value = fib(10); //   the 10th value of the fee sequence
}
```

The above fib function calls its own function, which is called recursion . In some algorithms, such as quick sort , it can make the code simple and easy to understand.

5.2. NRPL built-in functions

NRPL has built-in moving objects, object member variables and related data structures for motion control of equipment or robots. Users do not need to set or announce to call directly. For detailed description of built-in object parameters, please refer to " *NexMotion NRPL Directive*."*Manual* ".

6. Pointer and array

Index (pointer) is also a variable, kept the address of another variable (address) , use the index to make the program more streamlined and efficient, but if the improper use of indicators may also lead to program crash (Crash) .

6.1. Pointer and address

A pointer is simply a variable that stores a memory address .

Each of a memory cell (memory cell) are consecutive numbers or called addresses (address) , can be accessed individually or several consecutive information units as a group. Each a cell size of one byte (byte) can put a U8_T size variable, and the four consecutive units (4 bytes) can be
placed U32_T or I32_T . The NRPL indicator type is a size of 4 consecutive bytes , mainly used to store an address .

The following example shows how to declare indicator variables :

```
U8_T * value_ptr; / /  declare the indicator type, use type (U8_T) followed by * to indicate
U8_T value = 5; //    declare U8_T type and initialize it to 5
U8_T value_2 = 0; //  declare U8_T type and initialize to 0
```

The above variable declaration can be illustrated by the following figure, when U8_T *value_ptr is declared , where U8_T* indicates that the variable is

a U8_T indicator type, and the memory bit is at 0x1000 . Declaring U8_T value = 5 indicates that the memory address of the system assigned value is 0x2000 and the content is 5 .

Using the unit operator & can obtain the address of the object (usually the address of the variable) , called the address operation , as follows :

```
value = & value_ptr; //  the value of the address (0x2000) saved to value_ptr
```

After the execution, as shown in the figure below, the address of the value is set to the indicator variable value_ptr , and the value_ptr is pointed to the value . This **unit operator & can only be used for objects in the memory** , ie elements of variables or arrays. Both the constant and the expression cannot get the address.

The unit operator * is an indirection or an operation symbol called dereferencing . Use indirect addressing as follows :

```
Value_2 = *value_ptr; //  to the address stored in value_ptr , and take the content value of U8_T
```

Here * value_ptr , and value_ptr content is 0x2000 , thus can be considered * 0x2000 , which operating operation process is to address 0x2000 taken U8_T patterns (due value_ptr as U8_T index) of the content value, the value 5 . Execute the above expression, as shown below :
In addition to the value, the indicator can also specify the address setting content, as follows :

```
*value_ptr = 10 ;
```

Since *value_ptr is written to the left of the equal sign, it is regarded as the content value set to address 0x2000 . After the execution, as shown below :

6.2. The arguments of pointer and functions

Argument function (argument) is a method by value (call by value) function is called with no way to directly change the function called by coming variables. Suppose you want to reverse the two variables, use the swap function to do

this, and write as follows :

```
Swap( a, b);
```

Can not achieve the purpose, the swap function at this time is implemented as follows :

```
void swap( I32_T x, I32_T y )
{
      x = x ^ y;
      y = x ^ y;
      x = x ^ y;
}
```

Since call by value is used , swap cannot change the contents of a and b . It just reverses the copied version of a and b . After leaving the function, it is equivalent to nothing.

The correct way is to reverse the content values of a and b , that is, to reverse the content value of the memory. Therefore, using the indicator method, the memory addresses of a and b are passed in , as follows.

```
Swap( &a, &b);
```

Using the unit operator & to obtain the variable address, &a is the indicator pointing to a . However, this is not enough. The parameter type of the swap function should also be declared as the corresponding indicator type, and accessed by indirect reference. The method is as follows :

```
Void swap(I32_T *px, I32_T *py)
{
      *x = *x ^ *y;
   *y = *x ^ *y;
   *x = *x ^ *y;
}
```

6.3. Array

Declaring an Array variable is a collection of the same type of variables in a contiguous memory space. Array provides a simple way to represent similar information items, which is provided in the same variable name with injection subscript (subscript) represents in which individual data items.

To declare an array, you can use the following syntax:

Variable type array name [element number];

Here's an example :

```
U8_T   a [. 5]; //  declare a is a continuous . 5 th U8_T size of the space
```

Defines a a size there . 5 th U8_T array size is a continuous memory space, respectively, a [0], a [. 1] ~ a [. 4] , as shown below :

If a_ptr is declared as an indicator of U8_T , as follows :

```
U8_T *a_ptr;
```

The memory configuration is as follows :

Set a_ptr :

```
A_ptr = &a[0]; // give the address of a[0] (0x1000) to a_ptr
```

The memory configuration is as follows :

At this time a_ptr will point to a [0] space, so a [0] can use * a_ptr to access, as

```
x = *a_ptr; // *0x1000 is equivalent to a[0];
```

If a_ptr points to an element of the array, then a_ptr + 1 points to the next element of the array. The algorithm is as follows:

A_ptr + (1 * sizeof(U8_T)) => 0x1000 + (1 * 1) => 0x1001

Because a_ptr is an indicator of U8_T , and U8_T is 1 btye . To take a third one of array elements, worded as follows :

x = * (a_ptr + 2) ; //a_ptr+2 calculates the address to be 0x1002 , which is equivalent to a[2]

The general rules for adding or subtracting indicators can be as follows :

Ptr + expression or ptr - expression

The expression expression here must be an **integer, and** the addition algorithm of the index is as follows :

Ptr + (expression * sizeof(type))

ptr + expression is ptr next integer count number addresses elements, that is to say automatically expression Size multiplied by the element (in byte units) . In addition, **the value of the array name is equal to the address of the 0th element** , if executed :

A_ptr = & a[0];

Equal

A_ptr = a;

In other words, write a and & a [0] is the same, whereas a [1] can also be written as * (a + 1) , and therefore can be a seen as pointing U8_T indicators, so as long as the index patterns are variable array may be injection subscript (subscript) [i] is accessed, the following examples :

```
A_ptr[0] = 10 ; // is  equal to a[0]=10
*(a + 1) = 20 ; // is    equal to a[1]=20;
*(a_ptr + 2) = 30; // equal to a[2]=30;
```

```
a[3] = a_ptr[0]; //   equal to a[3] = a[0]
```

If the array name a is passed to the function, the fact upload is &a[0] , which is also the address of a[0] (0x1000) . The following example will initialize the array with the function :

```
Void init_a_array ( U8_T *ptr )
{
    Ptr[ 0]=0;
   Ptr[ 1]=1;
   Ptr[ 2]=2;
   Ptr[ 3]=3;
   Ptr[ 4]=4;
}
```

6.4. Address calculation

If the index p points to an array element, the p ++ make p points to the next element, and p + = i would p pointing down count i th element. These forms are the simplest indicator calculations or address calculations.

In some cases only index comparison, if p and q are the same point one array, relational operators (! ==, =, <, <=,>,> = .. , etc.) can be used, for example :

```
p < q
```

Any indicator can do == and != . But if you point to a different array, comparisons other than == and != are meaningless. Indicators can also be added or subtracted by an integer (6.3 has the addition and subtraction of the indicator) , as follows :

```
P + n
```

Representative p element pointed next count n th address elements, but also automatically is multiplied by each of the one element size. The final legal indicator operation is summarized as follows :

- Point to the mutual setting of the same type of indicator (assign)
- add or subtract an integer from the indicator

- Indicators pointing to elements of the same array can be compared with each other
- Indicators pointing to different arrays can be compared equally or unequally

6.5. Multi-dimensional array

Multi microarray may be seen as an array of arrays, e.g.
declared 3x3 in 2 -dimensional array, as follows

```
U8_T var[ 3][3];
```

Expressed in the figure, as follows :

If accessing var[1][2] , access the location of the image below

When U8_T var[3][3] is declared , a continuous 3x3 memory space is configured , as follows :

When var[1][2] is taken , its address algorithm is var+ 1*sizeof(U8_T)*3+ 2*sizeof(U8_T) is equal to 0x100+1*1*3+2*1=0x105 . Alternatively, you can use the indicator to access using one-dimensional representation.

```
U8_T *var_ptr = & var[ 0][0];
```

At this time , the content value of var_ptr is 0x100 . If var_ptr is to be used to take var[1][2] , it is represented as var_ptr[1*3+2] => var_ptr[5] and its address algorithm is var_ptr + 5 * sizeof (U8_T) = 0x100 + 5*1 = 0x105 .

6.6. Array initialization

The initialization paradigm is given when the one-dimensional array is declared, as follows :

```
U8_T var[ 10] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
```

Initialization is done by writing the values in braces and separating them with

commas. In addition to two-dimensional initialization, you can do the following :

```
U8_ T   var [2][5] =
{
      {  1 ,  2,  3,  4,  5 } , //var[0][0] ~ var[0][4]
   { 11 , 22, 33, 44, 55 } //var[1]    [0] ~ var[ 1][4]
};
```

Use the braces in brackets to divide the dimensions , separated by commas, and sort in according to the memory sort, such as the following initialization :

```
U8_ T   var [2][5] =
{
      {  1 , } , //var[0][0]
   { 11 , 22,  } //var[1][0 ] , var[1][1]
};
```

The above initialization will only be initialized var[0][0], var[1][0], var[1][1] , and the rest will be unknown.

7. Structure

Structure (Structure) is composed of one or more variables, each variable can be a different type, a name can be set through the easily processed together. Structures can be used to organize complex data and have significant benefits in large programs because it allows a group of related data to be treated as a single data unit.

7.1. **Structure definition**

Let's take an example to illustrate how to define a structure. In graphics processing we use two integer x coordinates and y coordinates to describe a point :

We can point (Point) and other elements of x and y is defined the following structure :

```
Struct point
{
    I32_T x;
    I32_T y;
} ;
```

The declaration of the structure begins with the keyword struct , followed by the structure naming, followed by a pair of braces to hold the announcement. Variables within the structure become members of the structure. Structure member names can be distinguished from common variables (non-structural members) by the same name, because they can be distinguished by context. In addition, please note that the type of the structure member variable cannot be the type of the structure itself, as follows :

```
Struct point
{
    I32_T x;
    I32_T y;
    Point p; // not legal because the type is the structure itself
};
```

After the struct type is established , the declaration of general variables can be made as follows.

```
Point pt;
```

This means that pt becomes a variable with a point structure. The structure variable can also give an initial value when it is declared . The method is to follow the equal sign and a pair of braces to clamp the corresponding value, as follows :

```
Point pt = {100, 200};
```

When you want to use members in a structure in an expression, the rules are as follows :

Structure variable name . Member Name

Origin members use to access, such as pt.x or pt.y .

The structure can be a nested structure, that is, the structure can further include a structure. Here is an example :

For example, a rectangle can be represented by two diagonal points, as follows :

Define the rectangular structure rect as follows :

```
Struct rect
{
      Point pt1;
   Point pt2;
};
```

The structure rect contains two members : pt1, pt2 are all point structures, declared as follows

```
 Rect screen;
```

Then write the following :

```
 Screen.pt1.x
```

The x coordinate of point pt1 representing screen

7.2. Structures and functions

The operations that the structure can use include :

- Copy the entire structure or set values to the entire structure
- Use & operator to get the structure address
- use structure members (values)

Copying and setting includes the transfer of arguments between functions and

the return of structures by functions.

The following is an example of a function return structure :

```
Point make_point( I32_T x, I32_T y )
{
  Point temp;
  Temp.x = x;
  Temp.y = y;
  Return temp;
}
```

The make_point() function passes in two integers and returns a point structure.
The make_point() function is used to dynamically give the initial value of any point structure. The example is as follows :

```
Rect screen;
Point middle;

Screen.pt1 = make_ point( 0, 0);
Screen.pt2 = make_ point( 100, 100);
Middle = make_point ((screen.pt1.x screen.pt2.x +) / 2,
                (screen.pt1.y + screen.pt2.y)/ 2) ;
```

You can also use the structure as an input parameter to a function. Here is an example :

```
Point add_point(point p1, point p2)
{
      P1.x += p2.x;
   P1.y += p2.y;
   Return p1;
}
```

In this case, the parameters and function return values are structures . Since the arguments of the function are passed in by " call by value" , the variables passed by the caller are not changed.

If you want a large structure passed to the function, available indicators point to this structure (pointer) is transmitted rather than copying the entire structure, to improve the efficiency of the program. The following is an example of an announcement of structural indicators :

```
Point  pt ;
Point *pp = &pt;

(*pp).x = 10;
(*pp).y = 20;
```

Declaring pp as an indicator points to a point structure pt . And (*pp).x and (*pp).y are values for their members . The parentheses in (*pp).x are necessary because the priority of the dot (.) operator is higher than the value of the asterisk (*) operator.

Structural indicator member values can also use (->) symbols, for example (*pp).x can be represented by pp -> x , note that the -> symbol is a sign followed by a minus sign.

```
Pp->x = 10;
Pp->y = 20;
```

The following example shows the use of structural indicators for the input parameters of a function : using the indicator structure to calculate the intermediate point example

```
Point middle_point( point *p1, point *p2 )
{
    Point temp;
    Temp.x= (p1->x + p2->x)/2;
    Temp.y =( p1- >y + p2->y)/2;
    Return temp;
}
```

8. Keywords

The following words are reserved as keywords identification purposes, is not available to for other purposes.

F64_T	U32_T	I32_T	U16_T	I16_T	U8_T
I8_T	Break	Case	Continue	Default	Do
Else	For	Goto	If	Return	Struct
Switch	Void	While			