PDL Reference Manual for D-series Drives

PDL

Process Description Language

Version 1.1
September 2014
HIWIN Mikrosystem Corp.

Table of Contents

1. Preface

1.1. About the manual

This document is applicable for mega-fabs and HIWIN D-series servo drives. It describes how to use **PDL (Process Description Language)**. PDL is a package for motion control programs developed for mega-fabs or HIWIN drives. Users are able to write a PDL program with *.pdl file (text file) on PCs, and to execute the PDL program in a drive's flash by MDP (DSP firmware).

Properties of PDL are described as following:

- Multitasking can handle up to maximum 4 tasks (0~3) at the same time. (The task 0 is reserved for system use.)
- Users are allowed to define variables (name, type, size) by themselves in programs.
- Support array and pointer.
- Support "**loop**, **while**, **if**, **else**, **till**, and **goto**" to control program flow.
- Support to create mathematical algorithms.
- Memory size for code is up to 32 Kbytes.
- Memory size for user variables is up to 400 long words.
- The name of a user variable is up to 17 characters.
- A **label** name is up to 24 characters.
- A **proc** name is up to 24 characters.

1.2. Multitasking

PDL supports to execute up to 4 tasks at the same time. In the memory of DSP, each task has its own counter, stack pointer, flag, and stack space. Assume that all tasks have the same priority, and there are *n* tasks going to be executed. If all tasks are not at **sleep** and **stop** states, each one will be executed once in *n* phases (66.7μsec per phase). The following is going to introduce the method of changing the priority of task. Users are able to decide the order of task 0 to task 3 after resetting drive by using the command of **#task/n** (*n* is the desired order number). However, it is not necessary for each task to be executed after resetting drive.

Global variables are public for all tasks. The same command is also able to be used in all tasks. Therefore, to avoid global variables, global functions, and other resource be occupied by two or more tasks, user are able to use **lock** and **unlock** commands to control data access by different tasks. Furthermore, the multitasking synchronization is able to be achieved by **lock** *a*nd **unlock**.

(1) Task is able to be executed by the following methods:

- Put **#task/n** in any section of the program. When the program goes through that section, the task with number *n* will be executed. Users are able to use **ret** to terminate the task with number *n*.
- Users are able to use the **run** command to create a new task through a PC (referring to ***RunFuncPdlN*** and ***SetArrayRunFuncN*** in the mpi.dll file). When using the **run** command, users should include the label address of task. The symbol of "_" has to be added in front of label. If users want to terminate the label, they should use the **halt** command. One label is allowed to be executed once on a PC.

(2) Task is able to be terminated by the following three methods:

- Execute **halt/ret/rertprintl** commands.
- Users are able to use the **kill** command to close other task.
- Users are able to terminate a task through PC (referring ***killTask*** in mpi.dll file).

1.3. Variable Type

Any variable has the following attributes:

- Defined by: system or user variables.
- Type: **long** (32 bit), **float** (32 bit), **short** (16 bit), **state** (1 bit), **float pointer** (32 bit), or **long pointer** (32 bit).
- Size: All variables can be considered as arrays with the minimum size of 1.
- Scope: (**long** and **float**) user variables can be temporary (procedure local variables) if they are declared in the header and within a procedure.

(1) System variable

System variables are predefined variables (name, length, location). Each one performs a certain function. For example, the system slave variables ***X_vel_max***, ***X_acc***, ***X_dcc***, ***X_new_sm_fac*** define the velocity trapeze profile, and setting the variable ***X_trg*** in the slave to a new value causes the X axis to move at this profile velocity.

(2) User variable

User variables are variables defined in the *.PDL file with **#long**, **#short**, and **#float** directives commands. They can be defined as **long** or **float** pointers (long pointer contains an address of another **long**/**short** variable, and **float** pointer contains an address of another **float** variable). If the user variable is declared in the header or within a procedure, it is **temporary** (local procedure variable). Variables may have 16 (**short**) or 32 (**long/float/pointers**) bit lengths.

(3) State

States being always system variables are special 1-bit variables used in conditional statements. States can be modified by using **seton/setoff/toggle/state assignment** commands. States may be represented inputs/outputs digital signal or any internal states (axis run/not run for example). All states are the bits of the status array variable.

A **short** or **long** variable may be interpreted as a signed or unsigned number depending on its function. The **size** attribute is relevant to array. Access to an item in array is done by using the square breakers “[ ]”.

Example:

```
vel_max[2]=20000;
abspos[n]=50000;          // n is a short local variable.
```

A variable can be indexed by a **constant** or a **short**/**long** variable. Any variable can be indexed no matter its size. When indexed with a constant, the compiler doesn’t check size overflow.

For example:

```
#short Eli, Rafi, Benny
My_array[100], N;              // Declare short user variables/arrays
```

```
Eli[0]=15;                  // Equivalent to Eli=15
Eli[1]=100;                 // Equivalent to Rafi=100
Eli[2]=500;                 // Equivalent to Benny=500
N=55;
My_array[N]=Benny ;         // Equivalent to My_array[55]=500
```

(4) Pointer

Pointers are 32-bit variables, which are used to store address to other variable.

Example:

```
#long v, *pv;   // pv is defined as pointer to long or short variable.
pv=&v;          // pv is assigned with the address of v
*pv=55;         // i.e, v=55
```

The indirection '*' is allowed only for variables that are declared as pointers. It means that the variable value is an address of another stored value that should be taken. The operator '&' when precedes variable name means that the address of the variable should be taken (and not its value). Pointers can be declared as **long** or **float** according to the variable type that they are going to point to.

Example:

```
#float f, *pf;      // pf is defined as pointer to float variable.
pf= &f;
*pf=5.4;            // i.e., f=5.4
```

No short type pointers. Pointers for **long** may also use for **short** as well. Pointers may be indexed in several ways.

Example:

```
#long v[6],n, *pv;
pv=&v;
*pv=0;        // v[0]=0
*pv[1]=10;    // v[1]=10
pv=pv+2;
*pv=20;       // v[2]=20
pv=&v[3];
*pv=30;       // v[3]=30
n=4;
pv=&v[n];
*pv=40;       // v[4]=40
n=1;
*pv[n]=50; // v[5]=50. Note that, operator * is performed before operator [].
```

```
        *pv[1]=50;    // v[5]=50
```

Pointers must be initialized to any variable address before using them with the indirection operator ‘*’. Otherwise, results are unpredictable (including software fail). Pointers may be defined for integer (**long**/**short**) or **float**. Users should use the correct pointer type when they access the variable otherwise an incorrect value will be read/written.

Pointers may be used to pass output parameters (“by reference”) to procedures.

1.4. Procedure

Procedures are similar to functions in C/C++. Not like C, they do not return the value directly. Return value can be achieved by pointer parameters.

Syntax of procedure:

```
proc <procedure name>(<variable type> variable name, <variable type> variable name,…) do
    …
    Procedure body
end;     // End procedure
```

Example:

```
// The following procedure computes the sum of 3 values (a, b, c parameters) and return
// result to sum parameter.
    proc add3( long a, long b, long c, long * sum) do
        #long tmp;    // Define temporary user variable.
        tmp=a+b;
        *sum=tmp+c;
    end;                    // Return from procedure
    …….
    #long v1,v2, s;       // Define global user variables.
    add3(4,5,6,&s);       // Result s = 15.
    v1=10;
    v2=100'
    add3(v1,v2,1,&s);     // Result s = 111.
```

Notes:

- The procedure deceleration starts with the key word **proc.**
- The procedure body starts with **do** and closes with **end.**
- All parameters variables and internal declared variables are temporary, i.e. they are not recognized outside the procedure. Temporary variables are allocated in the stack. Their scope lifetime is only when the procedure is executed. Non-temporary variables are global variables, which are declared outside any procedure, or system variables. Temporary variables are called also local variables in standards languages like C/C++. The procedure can access also all global variables.
- Labels that are within the procedure are recognized only in the procedure, and global labels or labels in other procedures are not recognized by the procedure.
- Because each task has its own stack area more than one task, it may run the same procedure without interfering each other (unless they access global variable(s)).

- Temporary variables can be **long** or **float**, and can be pointers, not allowed **short** type.
- Calling to procedure syntax is similar as in C/C++, i.e. the specified procedure name is followed by the arguments list enclosed in (). Each argument can be a constant or a variable. If the argument is constant, the compiler automatically converts it to the correct type if needed (**float**/**long**)
- When calling to procedure, the compiler checks the matching type of parameters in the parameter list again. The parameter list is declared in the procedure. It also checks the correct number of parameters (unlike C++, no default values is allowed, and no overload procedures).
- Temporary variables can be given the same name as global user variables (but not system variables) or the same name as temporary variables in other procedure. The same is applied to labels.
- When the procedure is declared below the line where using it, users may put a prototype of it to avoid compile error.

  Example:

```
proc mul(float a, long b, float * res);    // Procedure prototype.
    #float f1, m;                           // Global user variables.
    f1=44.2;
    mul(f1, 20, &m);        // The 'mul' is now recognized.

proc mul( float a, long b, float * res) do    // Procedure implementation.
     *res=a*b;
end;                                          // Return from procedure.
…….
```

2. Commands

This chapter describes commands in PDL.

2.1. Assignment commands

An assignment can take one of the following formats:

(1) var = var/constant;

(2) var = {const_0, const_1, const_2, ... , const_n};

(3) var = “string…”;

(4) var = -var;

(5) var = <state_name>; var = <label_name>;

(6) var = var op var/constant;

(7) var = func_name(arg_1, arg_2,..., arg_n);

(8) state = conditional expresses;

(9) var op= var/constant;

Unless other specified variables in the functions’ descriptions, any variable can be **short**, **long** or 32-bit **float**. Users may mix a **short**, **long**, and **float** variables in the same statement.

“arg_1, arg_2,... , arg_n” in case (7) stand for variables, where the last argument in any function is allowed to be also a constant.

When assign **short** variable to a long data, the MSB word will be lost. When assign **long** variable a to **short** data, the data is sign extended (excepts for the unsigned function). Assignment form **float** to **long**/**short** will be incorrect if the value of the **float** variable is outside of the range of the **long**/**short** variable.

A constant can be interpreted as a character code.

Notes:

- In case (1), no more than one side of the equation is allowed to be a slave variable.
- In cases (4), (5), (6), (7), only the left side of the equation is allowed to be a slave variable.
- <op> can be one of the following operators:
  - + (add)
  - - (subtract)
  - * (signed multiplied)
  - / (divide)
  - & (logic and)
  - | (logic or)
  - @ (logic xor)
  - % (mode, take reminder of divide)
  - ~ (invert)

Operation assignment are:

"+=", "-=", "*=", "/=", "&=", "|=", "@=", "%=".

That is instead of writing for example "var1=var1+var2", users may write "var1+=var2". Operation assignments are shorter in code, executes faster, and are more convenient.

Example :

```
#short x,y,ar[100],YS,DN,EL;          // Declare short user variable/array
#long   p1,p2,pr[60];                 // Declare long user variable/array
#float f1,f2;

f1=0,5;
f2=f1*4000;
p1=600000;
p2=p1;
pr[5]=p1;
pr=400;                          // Equivalent to pr[0]=400;
pr=0x02FC;                       // Constant written in Hexformat.
x=7; y=5;
pr[x]=pr[y]+400000;
x=x & y;
ar[x]=ar[2] * ar[y];
ar[x]=ar[x] * ar[y];
wait 20;            // wait 2.5 frames till p1 to be updated with ref_pos of axis 2
p2=p1+100000;       // now can use p1 variable
ar={1,3,5,6,900,-8000};     // Init first 6 variables in array
ar[2]={1,3,5,6,900,-8000};   // Init 6 first items in array start from 2
x=3;
ar[3]={1,3,5,6,900,-8000};   // Init 6 first items in array start from 3
YS={100,200,300};              // i.e., YS=100; DN=200; EL=300;
x+=y;                         // x=x+y, operation assignment
x*=0.5;                       // x=x*0.5, operation assignment
```

Notes:

The initialization of array is able to be started at any element, and assigns a series of values to following elements one by one. Only up to 7 values can be assigned to array at a time in the initialization of array.

Example:

```
#long array[100];
array[ 0 ] = {10, 11, 12, 13, 14, 15, 16};   // Initialize ‘array’ starting from the
                                    // 0th element to 6th element of ‘array’. array[0]=10,
                                   // array[1]=11,......, array[6]=16.
array[ 7 ] = { 17, 18, 19, 20, 21, 22, 23 };
```

2.1.1. Index auto increase

The index to array can be incremented automatically.

Example:

```
#short n,p;
#long myar1[10], myar2[10];
…….
n=0; p=0;
loop (10) do
myar1[n+]=myar2[p+];      .
end;
```

Notes:

The index to array is incremented after getting the value. The destination variable index is incremented after the source variable(s) index incremented. So if users want to use the one index (say *n*) for both destination and source, they should replace the assignment statement in the example by

myar1[n+]=myar2[n]; // The increment is on the destination variable index.

2.1.2. Auto increment/decrement

To increment/decrement **long/short** variable, users can use “++” or “–“ operator.

Example:

```
#long *po, myvar[10];          // Define pointer and array
myvar ++;                      // The same as: myvar = myvar +1;
myvar --;                      // The same as: myvar = myvar -1;
myvar [4]++;
myvar [n]++;
po=&myvar;
*po++;         // i.e., myvar++;
*po[1]++;      // i.e., myvar[1]++;
*po[n]++;      // i.e., myvar[n]++;
```

2.1.3. State assignment

The assignment of the state/condition expression to variable (**float**/**long**/**short**) is possible by using the keyword '**sttovar'**.

Example:

```
#long var1[5], var2;
#float f1;
var1= sttovar s2; // Simple state assignment
var1[var2]= sttovar ~s2;
var1[3]= sttovar (s2 & ~s3) | (~s2 &s3);
f1 = sttovar s1 | ( s3 & v2>1000 ) ;
var1 = s1 &   var2->3   & var2->10   // This variable is set to 1 if the logic
                                    // expression is true, and to 0 if it is false.
```

2.1.4. sizeof

The **sizeof** command is used to return the size of array.

Example:

```
#long myarray[275], onevar;
var = sizeof(myarray);   // Assign the size of 'myarray' to 'var' ('var'=275)
var = sizeof(onevar);    // Assign the size of 'onevar' to 'var' ('var'=1)
```

Notes:

The statement of **sizeof** in PDL is similar to that in C/C++. In PDL, it will return the size of array; however, in C/C++, it will return the size of object (unit is byte).

2.2. Program flow commands

2.2.1. halt

Format:

halt;

When a task arrives to **halt** command, it stops execution. At this time, the task is at the **idle** state and is available for run commands internally or externally.

2.2.2. wait, sleep

Format:

wait <local variable/constant >

sleep <local variable/constant >

The **wait**/**sleep** command holds the execution of task for a period specified by the argument. The time delay is calculated by:

sleep: value * 0.001sec = 20 phases

wait: value * 0.00005 sec = 1phase

The important difference between **wait** and **sleep** commands is that the **sleep** command puts the task in the state without consuming slots time (phases). So, other tasks may run more quickly.

Examples:

```
#short dd,x,ar[20];
sleep   1000;      // Wait for 1 sec
dd=2000;
sleep dd;          // Wait for 2 sec
ar[4]=5000;
ar[5]=7000;
x=5;
wait ar[4];        // Wait for 0.25 sec (5000*0.05 msec)
sleep ar[x];       // Wait for 7 sec
```

Notes:

The **sleep**/**wait** command doesn't work well with pointers. For example if you want to do

```
#long *ps,t
ps=&anyvar;
```

```
    Sleep *ps;
```

Please use this format instead.

```
    t=*ps;
    sleep t;
```

Don't use the **sleep** command between **lock** and **unlock** commands, and use the **wait** command instead.

2.2.3. goto

Format:

```
    goto <label>;
```

The **goto** command transfers the execution to the location marked by the label, which is any string (25 character maximum length) and marks the location in the program. The label should be ended with **':'**.

Example:

```
   ….
   goto loca1;
   ….
   ….
   loca1:
```

2.2.4. Indirect goto

Format:

```
    goto <label>[ <var/constant>];
```

Example:

```
    #short indx
    ……
    goto label_tab[2];      // Relevant to jump to func2
    ……
    indx=3;
    goto label_tab[indx];     // Relevant to jump to func3

    label_tab:
        goto func0;
        goto func1;
        goto func2;
        goto func3;
    goto func4;
```

2.2.5. call, ret

Format:

call <label>;

The **call** command transfers the execution to the location marked by this label, and pushes the current location to stack. The **ret** command is a return from subroutine (i.e. pops back the location from stack and returns to this location).

Example:

```
call subr1;
   …
   …
   subr1:
   …
   …
   ret;
```

The maximum number of nested **call** commands (together with **loop** commands) depends on the size of stack. Each **call** command uses one stack location. However, the **call** of procedure takes more stack locations (depending on the number of parameters and temporary variables that it uses).

Notes:

The task that runs externally (from host) also can be ended by the **ret** command. However, the task that runs after the reset (by using the directive **#task/n**) should not be ended by the **ret** command (only by **halt**). If users want to pass parameters to subroutine, they should use **procedure** instead of using the **call**/**ret** mechanism.

2.2.6. exitproc

Format:

exitproc;

This command is used to exit from procedure. Normally, the program exits from procedure when it arrives to the **end** command, that closes the **do** command after the procedure header. Use this command to exit in the middle of procedure.

Example:

```
proc func1(long flag) do
……
If (flag=-1) exitproc;        // Exit the procedure of func1 in the middle
……
end;          // Exit from the procedure in the end
```

2.2.7. loop

Format:

```
loop( <local variable/constant> ) do
      .
      .
end;
```

The **loop** command executes commands in the block (from **do** to **end**) *n* times, where *n* is specified by a local **short** variable or a constant. If *n* = 0 and *n* is interpreted as an unsigned number, the block will be executed 65536 times. If the argument is **long**, only the LSB word is taken.

Example:

```
nk=20;
loop(nk) do          // Execute the block 20 times.
….
pos1=pos2;
  loop(100) do       // Execute the block 100 times.
  ….
  ….
  end;
end;
```

The maximum number of nested **loop** commands (together with **call** commands) is defined by the size of stack for each task.

2.2.8. if, else

The **if** and **else** commands take one of the following formats:

a. if (conditional expression) command;

b. if (conditional expression) do

```
      ….
```

```
    end;
c. if (conditions expression) do
        ….
    else do
        ….
    end;
```

Where the conditional expression is the logic combination of several conditions or only one condition, i.e. the condition has the following format:

(condition1 [&/| contidion2] [&/| contidion3]....)

The conditional expression may be nested by using '( )', and can be inverted by using '~'. Only the length of lines will limit the number of conditions.

In the case a, it does not allow to use the **wait, state assignment,** another **if, while, till,** or **loop** command. The condition takes one of the following formats:

```
(1) <state>
(2) ~<state>          // That is not state.
(3) < <var> >         // The var is a variable that contains the state number.
(4) < ~<var> >
(5) ~(conditions expression)
```

In cases (3) and (4), variables assume to store state address (bit number). The relation can be one of the following operators:

```
(1) =       // Equal to
(2) >       // Greater than
(3) <       // Less than
(4) >=      // Greater than or equal to
(5) <=      // Less than or equal to
(6) <>      // Not equal to
(7) TOUT    // Time out detection. Find the full description of this operator below.
```

The maximum level of nested **if** or **else** commands (together with **loop** and **while** commands) is limited by the compiler to 200. The '&' is for 'logic and' of conditions, the '|' is for 'logic or' of conditions, and the '~' is for 'logic invert' of condition.

Example:

```
pos2=-100000;
if (pos1>100000 | pos1<pos2) do
    if (pos1>1000000) do
```

```
            abspos=pos2;
         else do
            abspos=-pos2;
         end;
         if (LEFT_LIM)    call sub1;
    end;
    #short st;
    st= <LEFT_LIM>;
    if ( <st>) pos2=pos2+100;    // If the left limit is triggered, do pos2=pos2+100.
    if (~<st> ) pos2=pos2-100; // If the left limit is not triggered, do pos2=pos2-100.
    if (pos>1000) call func1;      // If the condition is hold, call func1.
    if (pos>1000) goto label1;   // If the condition is hold, goto label1.
    if (pos>1000) func1(pos,300);    // If the condition is hold, call func1(pos,300).
    if (pos>1000) sleep 100 ;      // If the condition is hold, wait for 0.1 sec.
    if( (st1 &~st2) | ( ~st1 & st2) ) call sub1;      // If st1 xor st2, call sub1.
    if( ~((st1 &~st2) | ( ~st1 & st2)) ) call sub1; // If st1 not xor st2, call sub1.
```

Notes:

- The **else** command replaces the **end** command to close the **do** block of **if** command. It has the same nesting level as the **if** command.
- In the case a (**if (conditional expression) command;**), it is not to use the **else** command.
- In the case a, if the length of condition and command is less than the length of command buffer, the **if** command may be executed within 50 usec.

2.2.9. while

Format:

```
    while (conditional expression) do
        ....
        ....
    end;
```

The conditional expression has the standard format (referring to **if** and **else** commands). The block from **do** to **end** is executed when conditions are satisfied. The maximum level of nested **while** commands (together with **loop**, **if**, and **else** commands) is limited by the compiler to 200.

2.2.10. till

Format:

till (conditional expression);

By using this command, the task holds the execution until conditions are met.

Example:

```
till (X_ref_pos>20000 & ~X_run);   // Wait till X_ref_pos is greater than 20000
              // counts and the motor X stops runing, then the next command is excuted.
```

2.2.11. TOUT condition

Format:

If (…….. v1 TOUT v2/constant)

till (…….. v1 TOUT v2/constant)

while (…….. v1 TOUT v2/constant)

Here, **v1** and **v2** should be announced to integer variables. Besides, the condition of **TOUT** has the same syntax as other conditional operators (**=**, **>**, **<**, **<=**, **>=**, **<>**). This condition is TRUE if the following condition is satisfied:

fclk-v1 >=v2;

where **fclk** is a 32 bit integer variable of system and counts with 20000Hz. tBy using '~', the condition can be inverted (as in other conditional statement). For example:

```
till( ~(v1 TOUT v2) );    // Wait till fclk - v1 < v2
```

The following example can be used to understand the motivation of using this operator.

Example:

```
#long timeout,tend;
timeout=60000;
X_jvl=2000;                  // Let motor move by using Jog
tend=fclk+timeout;           // Calculate the end time of excuting Jog
till ( X_homed | fclk>=tend );   // Wait till arriving the home or flck passes 3 seconds.
X_stop_m=1;                  // Stop motion.
if (fclk>=tend ) do
   printl/101("ERROR time out when search home limit");
else do
   printl/101("OK, found home limit");
end;
```

This code works well till the **fclk** counter wraps around from the biggest 32 bit integer value (2^31-1) to the smallest negative 32 bit integer value (-2^31). The **fclk** wrapping around happens first time after about 30 hours and every next 60 hours. When **fclk** wraps around, there is a risk that the **till** command mayl not be executed properly. One way to overcome this problem is to replace the **till** command by using the **if** command and a loop, as the following example.

Example:

```
#long timeout,t0, tmp;
timeout=60000;
X_jvl=2000;        // Let motor move by using Jog
t0=fclk;           // Recorde the start time
waitloop
tmp=fclk-t0;     // Calculate how much time passes from t0
if ( ~X_home & tmp < timeout ) goto waitloop; // If the motor does not arrive the
        // home and it is not time out, go to loop.
X_stop_m=1;     // Stop motion.
if (tmp>=timeout ) do
    printl/101("ERROR time out when search home limit");
else do
    printl/101("OK, found home limit");
end;
```

This code will work well even when **fclk** wraps around, but it needs 2 commands to execute in continuous. Therefore, the home state is tested only every 2 commands (and not each command by using **till**). So, it is recommended to use the **TOUT** operator as follows.

Example:

```
#long timeout, t0;
timeout=60000;
x_jvl=2000;         // Let motor move by using Jog
t0=fclk;            // Recorde the start time
till (X_home | t0 TOUT timeout ); // Wait till arriving the home or flck passes 3
                                  // seconds.
x_stop_m=1;         // Stop motion.
if (t0 TOUT timeout ) do
    printl/101("ERROR time out when search home limit");
else do
    printl/101("OK, found home limit");
end;
```

2.3. Build in functions

Functions are built-in routines, which get arguments, do some processes on them, and store the result in the variable. They may be considered as another type of assignment statements, but not like procedures that are user-defined and do not return the value. The general format of function is:

```
<var>=func_name(arg_1, arg_2, ..., arg_n);
```

where **arg_1**,..., **arg_n** must be local variables, and the last argument can be a constant.

2.3.1. max, min

The max and min functions have the following format:

```
<var>=max(<var>,<var/constant>);
<var>=min(<var>,<var/constant>);
```

The max and min functions may replace the **if-else** statement. For example, the statement

```
if(var1>var2) do var3=var1;   else do var3=var2; end;
```

can be replaced by

```
var3=max(var1,var2);
```

And the statement

```
if(var1>var2) do var3=var2;   else do var3=var1; end;
```

can be replaced by

```
var3=min(var1,var2);
```

2.3.2. abs

Format:

```
<var>=abs(<var/constant>);
```

This function takes the absolute value of the argument. For example, the statement:

```
if (var2>0) do var1=var2; else do var1=-var2; end;
```

can be replaced by:

```
var1=abs(var2);
```

2.3.3. sign

Format:

```
<var1>=sign(<var2/constant>);
```

If the argument is positive, the result is 1. If the argument is negative, the result is –1. If the argument is equal to 0, the result is 0.

2.3.4. sin, cos

Format:

<var1>=sin(<var2/constant>);

<var1>=cos(<var2/constant>);

The unit of argument is radian.

2.3.5. sqrt

Format:

<var1>=sqrt(<var2/constant>);

Perform the square root of **var2**, and assign the result to **var1**. If the argument is negative, the result will be also negative.

2.3.6. divi

Format:

<var1>=divi(<var2>, <var3/constant>);

The integer part (not rounded) of **var2** dividing by **var3**/constant is assigned to **var1**.

The following example illustrates the difference of **divi** from standard divide operation.

Example:

```
#long r, p;
#float f;
p=11;
f=p/4;          // f=2.75
r=p/4;          // r=3 (After rounding)
f=divi(p,4);    // f=2
r=divi(p,4);    // r=2
```

2.3.7. shift

Format:

<var1>=shift(<var2>, <var3/constant>);

This function takes **var2** as an integer and shifts it by *n* bits, where the value *n* is specified by the last argument (**var3**/constant). If *n*>0, do the left shift; while if *n*<0, do the right shift.

Example:

```
#long r1,r2,rs1, rs2;
r1=0x000000F0;     // r1=1111 0000
rs1=1; rs2=-2
r2=shift(r1, rs1);   // r2 will be set to 1 1110 0000
r2=shift(r1, rs2);   // r2 will be set to 11 1100
r2=shift(r1, -8);    // r2 will be set to 0
r2=shift(r1, -4);    // r2 will be set to 1111
r2=shift(r1, 0);     // r2 will be set to 1111 0000 (r1=r2)
r2=shift(r1, 4);     // r2 will be set to 1111 0000 0000
r2=shift(r1, 8);     // r2 will be set to 1111 0000 0000 0000
r2=shift(r1,12);     // r2 will be set to 0x000F 0000 (Hex)
r2=shift(r1, 24);    // r2 will be set to 0xF000 0000 (Hex)
r2=shift(r1, 27);    // r2 will be set to 0x8000 0000 (Hex)
```

2.3.8. bitset, bitclr, bittog

Format:

<var1>=bitset(<var2>, <var3/constant>);

<var1>=bitclr(<var2>, <var3/constant>);

<var1>=bittog(<var2>, <var3/constant>);

The **bitset** function sets a specific bit in the variable (the bit number specified by the last argument **var3**/constant). The **bitclr** function clears a specific bit in the variable. Moreover, the **bittog** function toggles a specific bit in the variable.

Example:

```
#Long v1,v2
v1= bitset (v1, 0);     // v1=0x0000 0001
v1= bitset (v1, 1);     // v1=0x0000 0003
v1= bitclr (v1, 0);     // v1=0x0000 0002
v2=31;
v1= bitset (v1, v2);    // v1=0x8000 0002
v1= bittog (v1, 4);     // v1=0x8000 0012
v1= bittog (v1, 4);     // v1=0x8000 0002
v1= bittog (v1, v2);    // v1=0x0000 0002
```

2.3.9. memcpy

Format:

<var1>=memcpy(<var2>, <var3/constant>);

where **var1:** destination variable/array;

var2: source variable/array;

var3: *n* =the number of variables to be copied (must be **long**/**short** type).

This function copies one array to other array. It is much faster to copy arrays with the **memcpy** function than performing a loop of assignments. This function does not do type conversion between **float** and **long**/**short**. That is both **var1** and **var2** should have the same type.

Example: (copy array **l1** into **l2**)

```
#long l1[3], l2[3];
l1={0x01020304,0x11223344,0xaabbccdd};   // Initialize array l1
l2=memcpy(l1,3);                         // Copy array l1 to l2
halt;
```

The execution time for this command depends on the number of variables to be copied according to:

np=1+var3/20; (**np**=the number of phases)

Notes:

The value of **var3** is not limited, but users should be aware of the array sizes of source and destination to avoid exceeding the memory size. If **var3**<=0, this function is no effect.

2.3.10. memset

Format:

<var1>=memset(<var2>, <var3/constant>);

where **var**1**:** destination variable/array;

var2: source variable value;

var3: *n* = the number of variables to be set (must be long/short type).

This function sets the content of **var1** to that of **var2**. It is much faster to set array with the **memset** function than performing a loop of assignments. This function does not do type conversion between **float** to **long/short**. That is both **var1** and **var2** should have the same type.

Example: (clear array l1)

```
#long l1[3],t;
t=0;
l1=memset(t ,3);    // Clear array l1
```

The execution time for this command depends on the number of variables to be set according to:

np=1+var3/40; (**np** = the number of phases)

Notes:

The value of var3 is not limited, but users should be aware of the array sizes of source to avoid exceeding memory. If **var3**<=0, this function is no effect.

2.3.11. memmin, memmax

Format:

<var1>=memmin(<var2>, <var3/constant>);

<var1>=memmax(<var2>, <var3/constant>);

where **var1:** destination variable;

var2: source array;

var3: *n* = the number of variables in the array (must be **long**/**short** type).

These functions find the minimum/maximum of array. It is much faster to get the result by using these functions than by performing a loop. These functions do type conversion between **float** to **long**/**short**. That is **var1** and **var2** may not have the same type

Example:

```
#long l1[3],mina,maxa;
l1={4,11,8};
mina=memmin(l1 ,3);       // mina=4
maxa=memmax(l1 ,3);       // maca=11
halt;
```

The execution time for this command depends on the size of array (**var3**):

np=1+var3/40; (**np** = the number of phases)

Notes:

The value of **var3** is not limited, but users should be aware of the array sizes of source to avoid exceeding memory. If **var3**<=0, this function is no effect.

2.3.12. memsum

Format:

<var1>=memsum(<var2>, <var3/constant>);

where **var1:** destination variable;

var2: source array;

var3: *n* = the number of variables in the array (must be long/short type).

This function calculates the sum of array. It is much faster to get the result by using this function than performing a loop. This function does type conversion between **float** to **long**/**short**. That is **var1** and **var2** may not have the same type.

Example: (find average)

```
#long l1[3], avr;
l1={2,11,8};
avr=memsum( l1 ,3);     // avr = 21
avr=avr/3;              // avr = 7
halt;
```

The execution time for this command depends on the size of array (**var3**):

np=1+var3/40; (**np** = the number of phases)

Notes:

The value of **var3** is not limited, but users should be aware of the array size of source to avoid exceeding memory. If **var3**<=0, this function is no effect.

2.4. Special commands

2.4.1. printl, retprintl

The format of printl/retprintl command is:

printl/mode1/mode2 (" …. String…..", var1, …, varN);

retprintl/mode1/mode2 ("…. String…..", var1, …, varN); // printl and ret

where **mode1:** hex parameter with 8 characters, which defines color, beep, print format;

mode2: hex parameter with 8 characters, which defines the event information.

The expression in the bracts '( )' has the similar format as that in the standard C language. The **retprintl** command combines 2 commands, **ret** and **printl**, which are executed at the same phase. This is useful when **ret** terminates the task which is triggered by the host, and **printl** sends the event to the host about the end of task. It ensures that the event will be received by PC when the task was already terminated. There is no delay from **printl** to **ret** in **retprintl**. However, if **printl** and **ret** are executed separately, it may take more than one phase when another tasks are running and/or other task enters to **lock** state exactly after this task executes **printl** and before executes **ret**.

Notes:

The maximum number of variables allowed is 8, while the minimum number of variables is 0. Formats of **mode1** and **mode2** are in Hex. Please do not put the '0x' prefix here.

Example:

```
printl/00000103/00000001("This is a test");
printl/00000103/00000002("position of axis 2 is: %g ",ref_pos[2]);
printl/00000103/00000003("position of axis 2 is: %g ",ref_pos[2]);
```

2.4.2. printl/retprintl + parameters

PDL version 2+ supports the improved **printl**, which sends the optional parameters in real time. This **printl** supports any variable/state which can be printed, and also variable/state which is undirected by the index of pointer or array. Also, the local variable can be printed.

Example:

```
#long *pn , n, ar[10];
pnn=&pnn;
n=2;
…..
printl/103("*pnn=%08x , n=%ld , n address is %08x , ar[2]=%ld ",*pnn, n, &n, ar[n]);
printl/103("state IN1=%d ", IN1);
proc func(long v1, float f2 , instate run) do
printl/103("local variables: v1=%ld, f1=%g , state=%ld ",v1,f1, <run>);
```

```
end;
ar="this is a test";            // Assign the string of 14 characters to array
printl/103("the string is: %s ",ar[0], ar[1], ar[2], ar[3]);   // Printl the string
```

Notes:

- To print the string stored in the array, users should provide the exact amount of the relevant variables contained in the string. Here, each **long** array contains up to 4 characters.
- From the example, users can observe that values can be printed in various formats (not only as **double float** with **%g**). Use **%g** (as well as **%G**, **%f**, **%e**, **%E**) only for **float** values. Use **%x** to print the integer in Hex format, **%d** or **%u** to print the integer in the decimal format (signed or unsigned), and **%s** to print the string. The conventional format of print is the same as the **printf** command used in C/C++ language.
- The parameter is sent in real time.
- The syntax of **printl**/**retprintl** in the improved version is the same as before. The user should be aware of **%g** only for **float** variables (all variables are no more treated as double type).

Important:

Users must put a space after each format identifier in order to format the value correctly.

Example:

The procedure gets a flag to select the appropriate **printl** message from a jump table.

```
proc printMsg(long flag, long p1, float p2) do
    if(flag<0 | flag>=5 )do
        printl/101("flag=%ld out of orange 0...4",flag);
        exitproc;
    end;
    flag=flag*5;             // The length of printl + 2 parameters + exitproc is 5
    goto prlabel[flag];      // Indirect goto
    prlabel:
        printl/103("MSG A, p1=%ld p2=%g ",p1,p2); exitproc;
        printl/103("MSG B, p1=%ld p2=%g ",p1,p2); exitproc;
        printl/103("MSG C, p1=%ld p2=%g ",p1,p2); exitproc;
        printl/103("MSG D, p1=%ld p2=%g ",p1,p2); exitproc;
        printl/103("MSG E, p1=%ld p2=%g ",p1,p2); exitproc;
 end;
```

Note that, the flag is multiplied with the code length of “**printl** +2 direct parameters + exitproc”, which is equal to 5 in this case.

Example: (call the procedure)

```
_a1:
    printMsg(0,17,4.2);
ret;
_a2:
    printMsg(1,ltest1,ff1);
ret;
_a3:
    printMsg(ltest2,ltest1,ff1);
ret;
```

2.4.3. set, clear, toggle state

Format:

```
seton <state_name | <var> >
setoff <state_name | <var> >
toggle <state_name | <var> >
```

These commands modify the statuses of state variables. If the state represents an input bit or internal state (i.e. the state that is updated every frame from the DSP software), to modify the state will be only for a short time till the next internal updating. These commands are mainly used for states that represent outputs.

Example:

```
seton vacum_sl      // Turn on the vacuum solenoid
sleep 1000
setoff vacum_sl     // Turn off the vacuum solenoid
```

2.5. Directive

Directives are commands executed in the compilation time. All directives begin with the character '**#**'.

2.5.1. task

Format:

#task/<n>;

This command initializes the task with the specified number to locate at the program, where *n* is the task number.

Example:

```
p1=p2;
halt;
#task/2;          // The task 2 will start to run from this point after the drive is reset.
call initt2;
```

The task, which is not initialized by this command, will be in the idle state after reset. The idling task can be operated by other tasks, which execute the **run** command, or externally by host. The task cannot be initialized more than once. The task that operates with task directive starts to run after the drive is reset, and may be terminated by the **halt** command.

2.5.2. long, short, float

Format:

#short [_ro] [_lp(v1,v2)] **str_1, str_2, ..., str_n;**

#long [_ro] [_lp(v1,v2)] **str_1, atr_2, ..., str_n;**

#float [_ro] [_lp(v1,v2)] **str_1, atr_2, ..., str_n;**

where **str_1, ..., str_n** can be any string (up to 15 characters, key characters are not allowed). If users want to declare array, please use brackets '[]' and put the array size within them.

Example:

```
#long u,k,yosi,my_array[200],tmp_array[800];
#float _ro v1;    // v1 is read-only.
```

The **_ro** and **_lp** are optional and is defined as read-only and/or level protected attributes. They are applied only to access variables from host.

These commands are used to declare user variables (long and short lengths). For every user variable, the compiler allocates an address in the user data memory in the drive. The variable name is recognized from the point, where it was declared. **#long**, **#float**, and **#short** can appear more than once in the program. After compiling the file successfully, float variables will be stored in **USER_n.vrs** by using 32 bit length (as long variable). The compiler will save float variables to the address of bit 29 to 1. Also, when users want to declare pointers, '*' should be put in front of variable name.

Example:

```
#long v1, *pvv;
#float ff, *pff, *pfp;
```

No **short** pointers. Please use pointers with **long** type to point variables with **short** type.

2.5.3. define

Format:

```
#define str_a str_b
#define str_a(arg1,arg2…argn)   longstring
```

This command has 2 syntax formats. In the first format, the compiler will replace the string **str_a** with the string **str_b** anywhere. This command is useful to define constants.

The second format (supported to PDL compiler with version 25+) allows the creation of function-like macros. This format accepts an optional list of parameters that must appear in parentheses. When referring to the *argument* of original definition, it can be replaced by the *token-string* argument, which has actual arguments substituted for formal parameters. The macro can use other already defined macros.

Example 1:

```
#define ADD2(r,a,b)            r = a+b;
#define ADD3(r,a,b,c)          ADD2(r,a,b)      r += c;
#define ADD4(r,a,b,c,d)        ADD3(r,a,b,c) r += d;
#long result,var1,var2,var3,var4;
…..
ADD4(result,var1,var2,var3,var4)      // result = var1+ var2+ var3+ var4
```

The macro can be written in more than one line. Users should use the '\' in the end of each line but not in the last line.

Example 2:

```
#define LONGCODE   var1=var2+var3;\
                   ff1=ff2*ff3;\
                   someproc(var1,varb)\
                   call somelabel;\
                   var++;
….
LONGCODE // execute commands grouped by the macro LONGCODE
```

Notes:

(10) If the macro is ended with ';', ';' is considered as the part of **str_b**.

Example:

```
#define size 10;
#long array[size]
```

The definition of array will be replaced by array[10;], which will fail in the compilation. In this case, users should avoid ending the macro with ';', that is

```
#define size 10
```

(11) The syntaxes of **#define**, **#undef**, **#ifdef**, **#ifndef**, **#else**, and **#endif** are similar to high/low level languages, such as C, C++, Assembler.

2.5.4. undef

Format:

#undef string

Use to remove the macro previously defined by **#define**.

Example:

```
#define KEY   STRING1
……
#undef KEY
#define KEY   STRING2
```

2.5.5. ifdef, ifndef, elifdef, elifndef, endif

Format 1:

```
#ifdef KEY
  code a
#endif
```

If **KEY** is defined, the **code a** is compiled; otherwise, it is ignored.

Format 2:

```
#ifdef KEY
  code a
#else
  code b
#endif
```

If the KEY is defined, the **code a** is compiled, else the **code b** is compiled.

Format 3:

```
#ifndef KEY
  code a
#else
  code b
#endif
```

If the KEY is not defined, the **code a** is compiled, else the **code b** is compiled. Users can use **#elifdef** or **#elifndef** to make the sequence of conditions following the start of **#ifdef/#ifndef**.

Example:

```
#ifdef KEY1
  code a
#elifdef KEY2
  code b
#endif
```

If **KEY1** is defined, only the **code a** will be compiled, else the **code b** will be compiled if **KEY2** is defined.

Every block started by **#ifdef/ifndef** should be ended with **#endif**. The nested **ifdef** is allowed. That is the **code a** and **code b** may be composed with other source directives/macros, such as **#define**, **#undef**, **#ifdef** as well as normal commands.

2.5.6. include

Format:

```
#include <filename>
```

When the compiler arrives at this command, it begins to compile the file, which is specified by **filename**. When the file compiling is finished, the compiler returns to the original file. Up to 20 nesting includes are allowed.

Example:

```
#include "..\common.h"
```

PDL Reference Manual for D-series Drives