

DiosPro Command Syntax

DiosPro Series
Volume 2

Kronos Robotics 
and Electronics

www.KronosRobotics.com

DiosPro Command Syntax

Version 4.0

Updates

Software and updated manuals can be obtained from the Kronos Robotics web site located at www.kronosrobotics.com.

Forums

A web-based discussions board is located at one of the Kronos Robotics sister sites. These forums cover everything from the Dios product line to motor controller basics and robotics. They are located at: www.mgsweb.com/forum.

Warranty

Kronos Robotics warrants its products against defects in material and workmanship for a period of 90 days. If you discover a defect Kronos Robotics will, at its option, repair, replace, or refund the item's purchase price. Simply contact Kronos Robotics for an RMA. The customer is responsible for all return shipping expenses.

Disclaimer of Liability

Kronos Robotics cannot be held responsible for any incidental, or consequential damages resulting from the use of any Kronos Robotics products.

15-Day Money-Back Guarantee

It is very important to us here at Kronos Robotics that our customers are completely satisfied. If you are not happy with any product you purchase from Kronos Robotics it may be returned for a full refund or exchange within 15 days of the invoice date.

The returned items must be in unused condition and have original packaging. There will be a restocking fee of 30% only on items that do not meet this condition. Also note that this return policy does not apply to items that you have destroyed or damaged. For example if you hook up a microcontroller backwards you may not return it.

Shipping and Handling fees are non-refundable. The customer is responsible for all return shipping expenses.

Shipping Responsibility

Kronos Robotics ships all packages Priority Air Mail via the United States Postal Service with delivery confirmation. We have found this combination gives the fastest and most reliable delivery at a very reasonable cost.

Once a package has been delivered we are no longer responsible for the package. More specifically, if some one steals the package from your doorstep we will not be held responsible and no refund will be provided.

If your package has not been delivered and sufficient time has passed please email us and we will verify delivery. If delivery has been made you will be given the confirmation number and delivery details. Thereafter you must contact your local Post Office for further information.

TradeMarks

PIC is a registered trademark of Microchip Technology, Inc. Windows is a trademark of Microsoft Corporation. 1-wire is a trademark of Dallas Semiconductor.

Contacts

email: sales@kronosrobotics.com
phone: 703 779-9752
fax: 703 779-9753
web: www.kronosrobotics.com



Welcome

Welcome to the world of the Dios. Dios chips, modules and boards are tiny computers (microcontrollers) complete with memory and IO ports. They have built-in hardware features such as timers, PWM generators and an interrupt driven hardware UART. Some modules and boards have a built in regulator and PC interface.

There are several Dios Carriers available. The Dios Workboard Deluxe even has a built in LCD interface as well as a built-in breadboard. The Dios Workboard Basic has the same connections but has a large prototyping area instead of the breadboard.

The DiosPro provides 22-33 IO lines depending on the chip, module or board used. They have a built-in command set capable of running several thousand instructions per second. Several interrupts and in-line assembly are supported.

Language Overview

Introduction	5
Functions	6
Variables	8
Constants	9
Variable Pointers	10
Parentheses	10
MATH	11
Defining Floating Point Variables	12
Using round and trunc Commands	12
Conversion	13
Bit Fiddling	14
Arrays	15
String Variables	16

Command Syntax

Command Index	19
---------------------	----

Introduction

The Dios language was designed for both the beginner and the advanced engineer. You can start off simple and then move into the advanced features at your own pace. The language is very similar to other forms of basic. It has **goto**, **gosub**, **If then else**, and **for next** commands. It also has some very special commands like **pulsein**, **shiftin**, **shiftout** and **hserin**. If all that were not enough, the Dios language also supports functions and libraries, local and global variables, arrays, floating point, and string variables.

A compiler takes your high level instructions and converts them into low level instructions that a microcontroller can understand. The problem with compilers is that they can create inefficient code and some code duplication can occur using much more memory than necessary.

An interpreter uses an internal process (on microcontroller) to interpret tokens generated by a PC program to operate your microcontroller. Interpreters can be very efficient in memory usage but are slower than compilers.

The Dios Compiler Software is the best of both worlds. It's a compiler and efficient tokenizer. Compiled code and tokens will both be uploaded to the Dios Engine on the Dios Ultra Chip. This means that code that normally uses 8k bytes of program space will only use 2k.

Important

There is a current 64K text size limit in each Edit form. Once your file starts to approach this size simply place some of your functions in an include file.

Functions

Functions

The core of the Dios language is its ability to use functions. A function is a self-contained area in the program. It has its own variable space and when no longer needed it will free up this space.

Functions can have any number of parameters passed to them. Both floating point and integer values may be passed.

To pass a floating point value you must declare the argument as a float parameter.

In **example 1.1** the data1 parameter will expect a floating point value to be passed. If the value passed is not a floating point value it will be converted. The next 2 parameters, data2 and data3, will expect integer values passed to them. Again, if the value passed is not an integer values it will be converted. Notice that data3 does not have a declaration type. When the declaration type is omitted an integer value is assumed.

All functions take space from the call stack. This means that each time a function is called a certain amount of stack space is claimed.

Function overhead 7 bytes
Each local integer 2 bytes
Each local float 4 bytes

Each time a gosub command is called 2 bytes are used from the stack. Once the return command is executed the 2 bytes are reclaimed.

Once a called function exits the stack space it has claimed is released.

In addition to local variables all functions have access to global variables. This allows you to easily manage several hundred variables in very little space.

Functions are totally self-contained with their own goto's and gosub's and local constants.

In **example 1.2** the function get831 will return the result from an ADC831 Analog to digital chip. It is only about 100 bytes long.

Setting a Functions Return Type

By default, when a function returns a value it returns a 16-bit integer. To return a floating point value you need to define the function as a float value as shown in **example 1.3**.

Note: If a function is set to return a float and it is used in an integer expression the floating point value will be converted to integer. This will allow you to write generic libraries.

```
'example1. 1
func collect(data1 as float, data2 as integer, data3)
endfunc
```

```
'example 1.2
func main()
dim a
loop:
  a=get831(3,4,5)
  print a
  goto loop
endfunc

func get831(Dat,CLK,CS)
dim retdata

'Put the connected ports into correct mode
output CS,CLK
input Dat

'Select the chip and sets things in motion
low CS,CLK

pulseout CLK,1 'This starts the stream
shiftin Dat,CLK,200,retdata '8+64+128

high CS
exit retdata
endfunc
```

```
'example1. 3
func getpie() as float
exit(3.17)
endfunc
```

Optional parameters

You can set up functions so that they have optional parameters.

This is done by reading the software register OPP8 as shown in **example 1.4**, which contains the number of parameters that were actually passed. Note that the OPP8 register must be accessed as the first execution in the function call as other commands and function calls will change its value. If you need to access it later in your function, make a copy of it.

Something to keep in mind when using the OPP8 register: floating point variables take up 2 slots.

```
'example 1.4
func doit(port,timeout)
'OPP8 is a Register that contains count of variables passed.

  if OPP8 < 2 then
    timeout = 5000
  endif

endfunc
```

Passing String Data

When passing string, table or text data to a string the actual data is not passed. A reference to the data is passed. In order to access this data you must collect the data using the `getstringbyte` command.

In **example 1.5** the function call `disptext("hello")` does not pass the "hello" but instead passes an internal pointer to the data "hello" where it is stored in the program space. By using the `getstringbyte` command and the address variable we can cycle through the string data and do things with it.

```
'example 1.5
func main()
  disptext("hello")

endfunc

func disptext(addr)
  dim char
  again:
    getstringbyte addr,char,done
    print char
    goto again
  done:
endfunc
```

Other Function Notes

A function may be defined with any of the following keywords.

func
function
sub

Functions may also be ended with any of the following keywords.

endfunc
endsub
endfunction

The behaviors of these commands are the same. They were added to make porting VB functions a bit easier

Important

The use of parentheses are optional when using functions. It is however recommended that you use parentheses when using a function in a variable assignment.

Variables

Variables

There are three types of variables in the Dios: 16-bit integer variables, 32-bit floating point variables, and string variables. Both integer and floating point variables may be assigned as local or global variables. String variables can only be assigned as global.

Local Variables

When a variable is local, the space it takes will be freed up once the function in which it was declared has exited.

In **example 2.1** the three variable slots taken up by the x,y,z variables in function displaystuff will be freed up for use by other functions once the displaystuff function has exited.

Why do this? This features makes for a very efficient use of memory. It makes it easier on the programmer to keep track of things and to modularize code.

To declare a local variable use the dim statement.

To create an integer variable use:

dim xyz as integer

or

dim xyz

To create a floating point variable use:

dim xyz as float

```
'example 2.1
func main()
  dim myvarb1
  displaystuff()
endfunc

func displaystuff()
  dim x,y,z
  print x," ",y," ",z
endfunc
```

Global Variables

When a variable has been defined as a global variable it can be accessed by all functions. Global variables never go away.

To declare a global variable use the statement global. Global variables may be declared anywhere, even outside functions.

To create a global integer variable use:

global xyz as integer

or

global xyz

To create a global floating point variable use:

global xyz as float

Byte Variables

You can also access/modify a normal integer variable through its .byte(x) modifier.

For example

myvarb.byte(0) = myvarb.byte(0) + 1

Variable Scope

It is possible to have a global variable called **zzz** and a local variable called **zzz**. In this case the local variable will take precedence while in the function that created the local variable. The same is true for constants.

Variable/Constant/Function/Label names

Name length

There is no limit to the size of the names.

Valid Characters

The first digit in a name must be a letter or underscore. After the first digit is defined you may use numbers in the name.

How many

The amount of available RAM determines the actual number of variables that can be active. That is to say you can have several local variables defined but if the function is not currently being called the variable does not take any space.

You may define up to 500 global constants and 500 local constants. These have no effect on Dios memory.

Constants

Constants are numbers that are referenced by a useable name. For example, **const CLK 5** would allow us to use the word CLK to represent the number 5. That way you can have several references to the CLK port. To change the port from 5 to 6 means only having to change 1 line of code as shown in **example 2.2**.

Once a constant has been defined the value cannot be changed. In other words you can't have the statement `CLK = 25`.

Constant statements are local to the functions that they were declared, just like local variables.

```
'example 2.2
func main()

    const CLK 5

    output CLK
    high CLK
    pulseout CLK,100

endfunc
```

Global Constants

You can also define global constants, i.e constants that are available to all functions. To define a global constant use the `gconst` statement just like you would use the `const` as shown above.

Global constants can be defined anywhere in the program file.

Note: Constants may hold both floating point and integer values. They cannot hold string values.

Variable Pointers

Variable Pointers

A variable pointer is a integer variable that contains the address (location) of a variable. You can have pointers to integer, floating point and string variables. The string pointers are covered under the strings section.

Pointers require somewhat abstract thinking and are considered an advanced concept.

To create a pointer first create a integer that will become a pointer.

dim mypointer as integer

Next assign the pointer an address of a variable by using the @ operator.

```
mypointer = @somevariable
```

Why would you want to use pointers?

Normally values are passed to functions by value. This means a copy of the variable is made of the original. This is fine in most cases but what if you want to change the value of a variable by a function. To do this we want to pass the variable by reference not value.

mytestfunc(@myvariable) or you can pass a pointer **mytestfunc(mypointer)**

This passes a reference to the variable. Now inside the function you can actually manipulate the value of a passed variable.

```
func mytestfunc(addr)
  addresswritei addr,100
endfunc
```

addresswrite commands

To write a value to a variable pointed to by a pointer you need to use the addresswriti and addresswritef commands. These commands will correctly populate the internal bytes of the variable with the correct values.

```
'pointer example
func main()

  dim a,b,x
  clear

  mytestfunc(@x)

  print x

endfunc

func mytestfunc(addr)
  addresswritei addr,21
endfunc
```

Parentheses

In order to use parentheses in your expressions you need to be using 2.4 or later of the compiler and version 43 or later firmware. **DiosPro** chips prior to this firmware release must be updated at KronosRobotics. Standard Dios chips may use the onboard upgrader.

To use parentheses with function parameters you must be using parentheses with your functions

MATH

The Dios language supports two types of math: 16-bit integer math and 32-bit floating point math. In many cases the two are interchangeable but there are some subtle differences.

Integer Math

Integer math on the Dios is very simple. In most cases 16-bit math is used except in cases where multiplication results can be up to 32 bits. **All integer math is unsigned.**

The following operations are supported:

- *Multiplication
- /Division
- +Addition
- Subtraction
- &And
- |Or
- ^Xor
- //Return the Remainder after a Division
- **Return the high word after a multiplication

You can use math anywhere an expression is used or in variable assignments.

Example:

```
dim res as integer
res = 10 + 5
```

Floating Point Math

Floating point math is very similar to integer math. However larger numbers and signs are supported along with fractions. The use of Dios floating point math is almost as fast as integer math. It requires no extra memory overhead as the routines are all built into the Dios Engine.

The following operations are supported

- *Multiplication
- /Division
- +Addition
- Subtraction

You can use math anywhere an expression is expected.

Example:

```
dim res as float
res = 10 + 5
```

Floating Point Exceptions and rules

Any expression that expects an integer will have each component converted to integer before insertion into the expression.

Examples:

- Integer variables in assignments
- Most built-in commands
- Function arguments defined as integer

Any expression that expects a floating point value will have each component converted to 32 bit floating point before insertion into the expression

Examples:

Floating point variables

Function arguments defined as float

Output commands like print, debug, serout and hserout will automatically detect the use of floating point and display the appropriate value with 7 digits of accuracy..

All commands that require variables to hold target data are restricted to integer variables.

The following commands can be used with floating point variables.

while/wend

if/then/else

int

address

clear

Also note that floating point variables can be used in commands that require integer expressions. The floating point variable will be converted internally to integer before its used in the expression. The variable itself will remain intact.

Defining Floating Point Variables

You may define multiple integer variables using the short cut method as show here

```
dim x,y,z
```

As a default variable defined this way are integers. You can not use the shortcut method to define a floating point variable. You must use the as operator to tell the compiler explicitly that the variable is floating point.

```
dim myvarb as float
```

Using round and trunc Commands

The IEEE floating point routines can operate in trunk or round mode. For example $100 / 20$ does not come out to exactly 5 using the IEEE floating point math. Its more like 4.99999999 when the routines are in trunc mode and 5 when in round mode.

When the Dios is powered up it will default in trunc mode.

By using the round command before the actual math you can force the math calculations to round for you. By placing it before the display routine you force the display routines to round it for you.

Why do we default to trunc mode? Because it's faster and keeps a more accurate version of the number internally. (in most cases)

Conversion

The compiler supports three types of number or constant conversion.

Binary Conversion

You can assign a binary number to a variable or as part of an expression by preceding the number with a %.

Example:
res = %100

res will contain 4. Note that the MSB is on the left side.

Hex Conversion

You can assign a hexadecimal number to a variable or as part of an expression by preceding the number with a \$.

Example:
res = \$41

res will contain 65.

ASCII conversion

You can convert an ASCII character to a byte by enclosing the character in single quotes.

Example:
res = 'A'

res will contain 65

Bit Fiddling

Integer variables are made up from two memory locations (bytes). Each byte consists of 8 bits. You can access both the individual bytes and bits of integer variables by using extensions.

Byte Extension

.byte(x)

To access each byte of a variable use the byte extension, where x is the byte number. Use 0 for the low order byte and 1 for the high order byte.

Example 4.1 will print out 232 and 3.

You can also set a variable's individual bytes as in the following example.

Example 4.2 will print 1000.

```
'example 4.1
func main()
  dim a
  a = 1000

  print a.byte(0)
  print a.byte(1)

endfunc
```

```
'example 4.2
func main()
  dim a
  a.byte(0) = 232
  a.byte(1) = 3

  print a

endfunc
```

Bit Extension

.bit(x)

To access each bit in a variable use the bit extension, where x is the bit number 0-15.

Example 4.3 will print out 1

To set bits is just as easy.

Example 4.4 will print 3.

Note that when setting the bit, if the expression on the right side of the = is 0 then the bit will be set to 0. Anything else will set the bit to 1.

```
'example 4.3
func main()
  dim a
  a = 3

  print a.bit(0)

endfunc
```

```
'example 4.4
func main()
  dim a
  a.bit(0)=1
  a.bit(1)=1

  print a

endfunc
```

Register Access

Accessing the hardware and software registers is identical to accessing variables except that only a single byte is used and there is no byte extension.

Example 4.5 will toggle IO port 0 by directly accessing the hardware register.

```
'example4.5
func main()
  output 0

  loop:
    PORTB.bit(7) = 0
    PORTB.bit(7) = 1
    goto loop

endfunc
```

Arrays

You can create both floating point and integer arrays. An array is a block of the same kind of variable that can be accessed with an index. This effectively allows you to cycle through a list of variables.

Integer Arrays

Creating an integer array works just like creating normal integers. You simply need to add a bit more information so we can tell the compiler how many we want to define.

dim myvarb(10) as integer

Will create a block of 10 16-bit integer variables. You can access each individual element by using the index. Note that the first array element is 0 so in this case we will use 0-9 to represent the elements.

```
myvarb(0) = 5000 'The first element in the array  
myvarb(9) = 400 'The last element in the array
```

Floating Point Arrays

Float arrays work just like integer arrays. The only difference is that we will use the float declaration instead of integer.

dim myvarb(10) as float

This will create a block of 10 floating point variables. Again you can access each element by using an index.

```
myvarb(0) = 2.76 '1st element  
myvarb(2) = 300.54 '3rd element
```

Byte Arrays

While there are only 16-bit integers you can access the individual bytes in a single byte array using the byte index.

dim myvarb(10) as integer

Creates a block of 10 16-bit integers or 20 bytes. You can access each byte by using the .byte extension.

```
myvarb.byte(0) = 17 'First byte element  
myvarb.byte(6) = 32 '7th element
```

Array exceptions and rules

- You can use arrays in any expression. Automatic conversion will take place between each type.
- You can not pass the whole array to a function. Only a single element can be passed. You can use global variable array to manipulate array data in different functions
- You can not return whole arrays with the exit command. You can only return a single element.
- When a command requires a variable for data return you may not use an array.
- You can not access individual bits and bytes of array elements. For example myvarb(1).bit(7) is not currently allowed.
- There is no bounds checking on arrays so if you index outside the allocated block you will access data in other variables.

String Variables

String variables allow you to manipulate text data. You can define string variables only by using the global command.

```
global myvarb(20) as string
```

This defines a 20-byte string variable. You can store up to 19 characters in this variable. The last position is reserved for use as a string terminator. There are only 256 bytes of string memory available so use it wisely. The use of the table command can aid you in cutting down string space usage. Tables allow you to define string data in the program space. However, they cannot be written to.

Once a variable has been defined it can be assigned values.

Direct Assignment

```
myvarb = "jump"  
myvarb will contain jump
```

```
myvarb2 = myvarb+" down"  
myvarb2 will contain jump down
```

```
myvarb = * + " up"  
myvarb will contain jump up
```

String Insertion

```
myvarb="ABCDEFGH"  
myvarb(3)="mike"  
myvarb will contain ABCmikeH
```

```
myvarb(2)=90  
myvarb will contain ABZmikeH
```

```
myvarb(5)="1234567"  
myvarb will contain ABZmi1234567
```

Partial string Assignment

```
myvarb = "ABCDEFGG"  
myvarb2 = myvarb(1,5) 'Starting character 1 for 5 characters  
myvarb2 will contain BCDEF
```

```
myvarb2 = myvarb(1,200) 'Starting character 1 for 200 characters or end of string  
myvarb2 will contain BCDEFG
```

String Usage

There are some restriction in string variables usage.

You cannot assign a string variable to its-self. For instance, `myvarb = myvarb + "s"` is not allowed. If you need to add something to the end of an existing string use the `*` operator. This tells the string pointer to move to the end of the string.

```
myvarb = * + "s"
```

You cannot pass a string to a function or return a string from a function. You can, however, pass a reference to the string so that the function can compare or manipulate it.

Math operators do not work the same when using strings. For instance.

```
myvarb=65  
myvarb will contain A
```

```
myvarb=90+89+88  
myvarb will contain ZYX
```

Assigning an integer variable to a string works the same way. The lower 8 bits are converted to ASCII and added to the string.

There is no bounds checking on string assignments. So if you assign something outside of a string-defined size it will be inserted into the next string.

Once a string is defined its contents are unknown until you assign something to it.

String Terminator

An end of a string indicated by a 0 value character. Hence a string "jump" will actually contain 106,117,109,112,0.

Most of the manipulation of the string terminator is done automatically. You can also manipulate the terminator.

```
global myvarb(20) as string  
myvarb=0  
or  
myvarb(0)=0  
will create an empty string
```

```
myvarb="ABCDEFGH"  
myvarb(3)=0  
myvarb will contain ABC
```

Notes on string insertion

As long as the insertion is located inside the bounds of the original string no terminator will be inserted.

```
globalvarb1(10) as string  
varb1="ABCDEFGH"  
varb1(3)=90  
varb1 will contain ABCZEFHG
```

If the insertion point starts inside the bounds and extends outside the bounds a new terminator will be added to adjust the length.

```
globalvarb1(20) as string  
varb1="ABCDEFGH"  
varb1(5)="1234567"  
varb1 will contain ABCDE1234567
```

If the insertion point is outside the original string no terminator will be inserted.

```
global varb1(10) as string
global varb2(10) as string
```

```
varb1="ABCDEFGH"
varb2="123456789"
```

```
varb1(13) = "mike"
```

```
varb1 will contain ABCDEFGH
varb2 will contain 123mike89
```

String Address Operator

You can use the string address operator # to assign a string based on an address. Let's take the following example.

```
global varb1(20) as string
global varb2(20) as string
```

note that the address of varb2 is 20 (end of varb1).

```
varb2="Mike"
varb1=#20
```

The address operator takes the integer expression value and treats it like a string address. This can be useful in quick string or table access with functions where the address is passed.

```
dim x as integer
dim z as integer
global myvarb(20) as string
```

```
myvarb="Mike"
```

```
z=myvarb
print forcstring z 'Tells the print routines that z is a string address and to print the complete string
print #z 'Tells the print routine to print the contents of a string pointer (single character)
```

```
x=#z 'Assign a single character pointed to by string pointer z to variable x
print nodec x 'Print then variable. Force print command to print raw value with nodec modifier.
```

Command Index

absoff	.21	lcdinit	.56
abson	.21	lcdwrite	.56
address	.22	lookdown	.58
addressreadi / addressreadf	.23	lookdownmode	.59
addresswritei / addresswritef	.23	lookup	.58
arrayget	.24	low	.59
arrayset	.24	memXread / memXwrite	.60
bintodec	.25	nop	.61
branch	.25	odhigh	.62
break	.26	odlow	.62
clear	.26	onirq	.61
customportget	.27	onportgosub	.62
customportset	.28	onportgoto	.62
data	.29	output	.63
debug	.31	pause	.63
dec	.30	pauseus	.63
decmask	.32	portbitget	.65
dim	.32	portbitset	.65
eeread	.33	portget	.64
eewrite	.33	portset	.64
end	.34	pot	.67
endasm	.35	print	.66
endfunc	.34	pullupoff	.69
endirq	.34	pullupon	.69
endirqasm	.35	pulsein	.68
exit	.36	pulseout	.69
exitirq	.36	rccount	.70
fabs	.37	readfromscratchX	.70
finv	.37	return	.71
for next	.38	round	.71
func	.38	savetoscratchX	.72
fuzzy	.39	serin	.73
getbyte	.39	serout	.74
getidpacket	.42	sersetup	.75
getpacket	.40	setdecmask	.76
getstringbyte	.44	setfloatmask	.77
getvpacket	.40	shiftin	.78
global	.44	sleep	.79
gosub	.45	sonar	.79
goto	.45	sqrt16	.79
high	.46	startasm	.80
hserin	.46	startirqasm	.99
hserout	.47	strout	.100
hsersetup	.48	swapbyte	.100
if then else endif	.50	swapnib	.101
inc	.49	table	.101
include	.51	timer commands	.102
input	.49	toggle	.104
int	.49	trunc	.104
ioport	.52	waitport	.105
irq...	.53	watchdogoff	.108
irqfunc	.54	watchdogon	.107
KRASsembler Commands	.84	while wend	.106
lcdchar	.55		
lcdcontrol	.55		
lcdgoto	.55		

Command Syntax Overview

Each command name will be followed by a syntax example and a syntax parameter type. If the command syntax example is omitted then the command has no parameters. Some commands can take repeating arguments. These will be indicated by the

For example:

inc variable
inc ivarb

inc variable

This syntax example is used as a command reference and each parameter will be explained in the command description.

inc ivarb

This syntax example shows the parameter type. It indicates the type for each parameter. Each type will be explained in detail below.

- **exp**: Any variable type, literal mathematical expression. Bit and Byte extensions are supported. Registers and array elements are also supported.
- **ivarb**: Integer variable accepted only. No arrays, bit or byte extensions.
- **fvarb**: Floating point variable accepted only. No arrays, bit or byte extensions.
- **varb** Both floating point and integer variables are accepted. No arrays, bit or byte extensions.
- **label**: Valid location label.
- **string**: Quoted string as in "hello world". String variables and tables are also supported.
- **operator**: Valid operator as in > < >= <= <> != ==
- **functionname**: Valid function name.
- **irqname**: Valid IRQname
INT0, INT1, INT2, TMR0, TMR1, TMR2, TMR3, CCP1, CCP2, AD, RB, TX, LVD, SSP, PSP

_ Continuation Operator

This character can be placed at the end of a line. This tells the compiler that the current line continues to the next line. This is useful when doing tables as well as lookup and lookdown commands. The continuation operator can not be contained in quotes.

Function Declarations

There are a few other function declaration commands that can be used to declare functions.

func
function
sub

Also

endfunc
endfunction
endsub

absoff

D DP

absoff
absoff

Description

When a floating point number or variable is assigned to a integer it will convert negative numbers to zero. When the abson command is issued it changes the default to convert to the absolute value.

IE -12.2 will return 12

The absoff command returns the system to the default mode so

-12.2 will return 0.

```
'abson example  
func main()
```

```
  dim x as float  
  dim y as integer  
  x = -12.16  
  abson  
  print x
```

```
  y = x  
  print y
```

```
endfunc
```

abson

D DP

absoff
absoff

Description

When a floating point number or variable is assigned to a integer it will convert negative numbers to zero. When the abson command is issued it changes the default to convert to the absolute value.

IE -12.2 will return 12

address

D DP

address varbin,varbout
address ivarb,ivarb

Description

Returns the address location of the global or local variable. Can be used with both float and integer variables. This command is used to retrieve the address so it can be used with one of the read or write commands

- **varbin** - The variable you want the address of. This can be a float or integer variable.
- **varbout** - The variable to place the address into. This must be a integer variable.

Related Commands

mem1read
mem1write
mem2read
mem2write

```
'address example
func main()

dim v1loc,v2loc
dim varb1 as integer
dim varb2 as integer

address varb1,v1loc
address varb2,v2loc

print "Varb1 located at ",v1loc
print "varb2 located at ",v2loc

endfunc
```

D DP

addresswritei / addresswritef

addressreadi / addressreadf

```
addresswritex address,value
addresswritei exp,exp
addresswritef exp,exp
```

```
addressreadx address,value
addressreadi exp,ivarb
addressreadf exp,fvarb
```

Description

The address read and write commands are used in conjunction with variable pointers. If you pass the address of a integer or float variable to a function it allows you to manipulate that variable. This ability can allow you to change the value of multiple variables with one function call.

- **address** - You must pass the address of the variable you wish to write to or read from. You can use the @ operator with any variable to obtain its address.
- **value** - The value you wish to place into the variable pointed to by address. Used with addresswritei and addresswritef commands.
- **ivarb** - The variable to place the integer value read from the variable pointed to by address. Used with addressreadi.
- **fvarb** - The variable to place the float value read from the variable pointed to by address. Used with addressreadf.

```
'addresswritei example
func main()
  dim b

  b=100

  print "Before Call b=",b
  changeit(@b)
  print "After Call b=",b

endfunc

func changeit(baddress)
  dim x

  addressreadi baddress,x
  print "Use addressreadi to access variable value. b original
value=",x

  addresswritei baddress,2113
endfunc
```

```
'addresswritef example
func main()
  dim b as float

  b=100.7

  print "Before Call b=",b
  changeit(@b)
  print "After Call b=",b

endfunc

func changeit(baddress)
  dim x as float

  addressreadf baddress,x
  print "Use addressreadi to access variable value. b original
value=",x

  addresswritef baddress,2113.2
endfunc
```

arrayget

D DP

```
arrayget startvarb,index,resultvarb  
arrayget ivarb,exp,ivarb
```

Description

All variables are stored in contiguous memory in the order in which they are declared with the dim or global statements. By passing the startvariable and using index you can retrieve values from variables in memory. This will allow you to cycle through variables like arrays.

- **startvarb** The reference point into memory.
- **index** Expression that will be added to the startvarb reference.
- **resultvarb** is the integer variable to store the retrieved value.

Related Commands

arrayset

```
'arrayget example  
func main()  
  dim d0,d1,d2,d3  
  dim x,vout  
  
  d0=20  
  d1=35  
  d2=40  
  d3=50  
  
  for x = 0 to 3  
    arrayget d0,x,vout  
    print "At count ", x, " array value=", vout  
  next  
  
endfunc
```

arrayset

D DP

```
arrayset startvarb,index,value  
arrayset ivarb,exp,exp
```

Description

All variables are stored in contiguous memory in the order in which they are declared with the dim or global statements. By passing the start variable and using index you can assign values to variables in memory. This will allow you to cycle through variables like arrays.

- **startvarb** The reference point into memory.
- **index** Expression that will be added to the startvarb reference.
- **value** The value to assign to the indexed variable.

Related Commands

arrayget

```
'arrayset example  
func main()  
  dim d0,d1,d2,d3  
  dim x  
  
  for x = 0 to 3  
    arrayset d0,x,100  
  next  
  
  print "arrayset result d0=", d0, " d1=", d1, " d2=", d2, " d3=", d3  
  
endfunc
```

bintodec

D DP

```
bintodec value,v10k,v1000,v100,v10,v1
bintodec exp,ivarb,ivarb,ivarb,ivarb,ivarb
```

Description

Converts a variable or value into 5 individual variables. This is primarily for display systems like LCD's

- **Value** is a value or variable that represents the value to convert.
- **V10k** is where the 10 thousands indicator will be stored
- **V1000** is where the thousands indicator will be stored
- **V100** is where the hundreds indicator will be stored
- **V10** is where the tens indicator will be stored
- **V1** is where the ones indicator will be stored

Note: In order to display the returned values you will need to add the ASCII value 48 to each one as shown in the example.

Related Commands

none

```
'bintodec example
func main()
  dim vnum,v10000,v1000,v100,v10,v1

  vnum = 1500

  bintodec vnum,v10000,v1000,v100,v10,v1

  print "raw values:"
  print "v10000=",v10000
  print " v1000=",v1000
  print " v100=",v100
  print " v10=",v10
  print " v1=",v1

endfunc
```

branch

D DP

```
branch offset,location,location,location....
branch exp,label,label,....
```

Description

Based on a given offset branch to a specified location.

- **offset** specifies an expression that specifies which location to jump to.
- **location** specifies the labels to jump to.

Related Commands

goto
gosub

```
'branch example
func main()
  dim x

  for x = 0 to 4
    branch x,Loc0,Loc1,Loc2
    print "Fail"
  cont:
  next

  alldone:
  goto alldone

  Loc0:
  print "Branch to Loc0"
  goto cont

  Loc1:
  print "Branch to Loc1"
  goto cont

  Loc2:
  print "Branch to Loc2"
  goto cont

endfunc
```

break

D DP

break expression, expression....
break exp, exp

Description

This command is identical to the print command except execution in the Dios is halted until the continue button is clicked on the debug window.

Note that all IRQ's are stopped when the break command is encountered. By setting DIOSFLAGS.bit(7)=1 IRQ's will continue to run when break is encountered.

All commands that send output to the debug port have a 256us delay added to accommodate slower machines and some USB com ports. This can be changed by changing the value in the DEBUGPADL and DEBUGPADH registers. This value is in microseconds.

Related Commands

print

```
'break example
func main()
  dim x

  for x = 1 to 10
    break "x=",x
  next

endfunc
```

clear

D DP

clear variable,variable,variable....
clear varb,varb,...

Description

Clears the indicated variables. (sets them to 0). If no variables are indicated it clears all global and local variables regardless of scope.

```
'clear example
func main()

  dim myvarb
  myvarb = 1000
  print myvarb
  clear
  print myvarb

endfunc
```



customportget

customportget resultvariable
customportget ivarb

Description

Sets the bits in the result variable based on the states of input ports. The input ports are selected by setting various registers. Effectively you are building a custom 8-bit input port.

- **resultvariable** is where the data will be stored.

To setup the ports use the following registers:

CUSTOMBIT0
 CUSTOMBIT1
 CUSTOMBIT2
 CUSTOMBIT3
 CUSTOMBIT4
 CUSTOMBIT5
 CUSTOMBIT6
 CUSTOMBIT7

Here is an example on how to set bit 0 of your custom port to IO port 8.

CUSTOMBIT0=8

You also need to set the CUSTOMMASK register. This tells the command which bits you want to read or write. If it is set to 0 all bits are read or written to. If it is set to 255 no bits are read or written.

The CUSTOMMASK register is used to mask out bits that you don't want to read or write. This is useful when communicating with devices that only require a few bits such as a 4-bit LCD interface.

Related Commands

portset
 portget
 customportset

```
'customportget example
func main()
  dim dat
  input 4,5,6,7,8,9,10,11

  CUSTOMBIT0=4
  CUSTOMBIT1=5
  CUSTOMBIT2=6
  CUSTOMBIT3=7
  CUSTOMBIT4=8
  CUSTOMBIT5=9
  CUSTOMBIT6=10
  CUSTOMBIT7=11
  CUSTOMMASK=0 'Read all ports

  customportget dat
  print "Ports 4-11 =",dat

endfunc
```

customportset



customportset expression

customportset exp

Description

Sets the IO ports based on the value in the expression. The output ports are selected by setting various registers. Effectively you are building a custom 8-bit output port.

- **expression** is used to set the 8 custom ports. The lower 8-bits are used to set the particular ports. For example, if bit 5 is 0 then the IO port pointed to by the register CUSTOMBIT5 will be set to low. If not it will be set high.

To setup the ports use the following registers:

CUSTOMBIT0
CUSTOMBIT1
CUSTOMBIT2
CUSTOMBIT3
CUSTOMBIT4
CUSTOMBIT5
CUSTOMBIT6
CUSTOMBIT7

Here is an example on how to set bit 0 of your custom port to IO port 8.

```
CUSTOMBIT0=8
```

You also need to set the CUSTOMMASK register. This tells the command which bits you want to read or write. If it is set to 0 all bits are read or written to. If it is set to 255 no bits are read or written.

The CUSTOMMASK register is used to mask out bits that you don't want to read or write. This is useful when communicating with devices that only require a few bits such as a 4-bit LCD interface.

Related Commands

portset
portget
customportget

```
'customportset example
func main()
  output 4,5,6,7,8,9,10,11

  CUSTOMBIT0=4
  CUSTOMBIT1=5
  CUSTOMBIT2=6
  CUSTOMBIT3=7
  CUSTOMBIT4=8
  CUSTOMBIT5=9
  CUSTOMBIT6=10
  CUSTOMBIT7=11
  CUSTOMMASK=0 'write to all ports

  'Turns on all 8 ports
  customportset 255

endfunc
```

data

D DP

data value,value,value.....
data literal,literal,literal,...

Description

The data statement lets you set up predefined data that will be stored in the internal eeprom. You can specify numeric data or strings. Strings of data can be created by enclosing the data in quotes.

If you use a value larger than 255 it will be divided into two bytes with the high order byte first. By using the word,byte and auto key words you can force the data command to write words or bytes to the eeprom regardless of the size of the number.

You can have as many data statements as you like, but there are only 1024 bytes of eeprom data available on the DiosPro, and 256 on the Dios.

Related Commands

eeread
 eewrite

```
'data example
func main()
  dim size

  data byte ' Forces byte mode
  data 21,22,23
  data 17,21,15

  eeread 0,size
  print size

  eeread 1,size
  print size

  eeread 2,size
  print size

endfunc
```

dec



dec varb
dec ivarb

Description

Does the same thing as `varb = varb - 1`. It decrements the variable by 1, but is much faster and uses less memory.

- **varb** The variable you wish to decrement.

Related Commands

inc



debug

debug value1,value2
debug exp,exp

Description

Sends data to the debug port.

- **value1-value_n** is the data to be sent to the hardware serial port.
Separate values with a comma

Notes

You can send a string of characters to the debug port by using quotes. You can precede the value with the **dec** modifier. This will convert the raw value to a decimal representation so it can be displayed.

The format operators may also be used to change the numeric format. See the print command.

For example, debug 65 will write the character A to the terminal or serial LCD. But the command debug dec 65 will output the actual characters 6 and 5 to the terminal or serial port. Note that the actual format of the dec command can be changed by the decmask command.

All commands that send output to the debug port have a 256us delay added to accommodate slower machines and some USB com ports. This can be changed by changing the value in the DEBUGPADL and DEBUGPADH registers. This value is in microseconds.

You can use setdecmask and setfloatmask to change the formatting of numeric printout.

See the print command for a complete listing of modifiers.

Related Commands

print
serout
hserout
strout
setdecmask
setfloatmask

```
'debug example
func main()

  debug "Hello world",13,10

  debug {05} dec 123,13,10

endfunc
```

decmask

D DP

decmask value
decmask exp

Description

This command affects how the dec modifier is applied to the print, debug, serout, and hserout commands.

• Bits 0-4

Digit masks. These bits determine which digits get displayed. If you only want the last two digits displayed you would use 00011. For all digits use the default of 11111.

• Bit 6

Display spaces. When leading digits are enabled we can display them as 0's or spaces. If this bit is 1, they will be spaces.

• Bit 7

Display leading characters. When a number that is less than 10000 is displayed we must decide what to do with the leading digits. We can display them or discard them. If this bit is 0 we will discard them (default). However if this bit is 1 we will display leading digits.

Related Commands

print
debug
hserout
serout

```
'decmask example
func main()

  debug "Look a the default ",dec 200,13,10
  decmask 128+31
  debug "Now you see leading zeros ",dec 200

endfunc
```

dim

D DP

dim variable,variable,variable..... (only valid for integers)
dim variable as float
dim variable as integer

Description

The Dim statement is used to define all local variables. A local variable is a variable that is only valid and visible within its function. For instance a local variable defined in func main will not be visible to func disp unless it is passed as a value. You can define multiple variables on the same line by separating them with a comma.

To create a integer variable use:

```
dim xyz as integer
or
dim xyz
```

To create a floating point variable use:

```
dim xyz as float
```

Related Commands

global
clear

```
'Example
func main()

  dim tint as integer
  dim tfloat as float

endfunc
```

eeread

D DP

```
eeread address,resultvarb
eeread exp,ivarb
```

Description

We can read from the internal eeprom memory with the eeread command.

- **address** - The address of the eeprom to read from. 0-1023
- **resultvarb** - Where we will store the value we retrieve.

Related Commands

```
eewrite
data
```

```
'eeread and write example
func main()
  dim addr,dat

  'First write some data to eeprom
  for addr = 0 to 255
    eewrite addr,addr
  next

  for addr = 0 to 255
    eeread addr,dat
    print "Read: ",addr," Value=",dat
  next

endfunc
```

eewrite

D DP

```
eewrite address,value
eewrite exp,exp
```

Description

We can write to the internal eeprom memory with the eewrite command.

- **address** - The address of the eeprom to write to. 0-1023
- **value** - The value to write to the given address

Related Commands

```
eeread
data
```

endfunc / endirq / end

endfunc

D DP

Description

The endfunc statement is how we tell the program where a function ends. Please note that if the calling function is expecting a result from a function, that endfunc will always return 0.

Related Commands

func
exit
exitirq

```
'endfunc example
func main()

endfunc
```

endirq

D DP

Description

Use the endirq command to end the Dios irqfunc.

Related Commands

irqfunc
exitirq

```
'endirq example
func main()
  output 2
  global counter
  clear

  irqglobalstart
  irqperfstart
  onirqTMR0 irqtest

  irqTMR0start
  timer0mode16bit
  timer0sourceclock
  timer0prescale 4
  timer0on
```

```
loop:
  toggle 2
  goto loop
```

```
endfunc
```

```
irqfunc irqtest()
  counter = counter + 1
  print counter
  exitirqTMR0

endirq
```

end

D DP

Description

Stops program execution. Hardware IRQ's will still function

```
'end example
func main()
```

```
  dim x
```

```
  for x = 1 to 1000
```

```
    print "x=",x
```

```
    if x = 15 then end
```

```
  next
```

```
endfunc
```

endirqasm

D DP

Description

Use the endirq command to end an assembly IRQ routine.

Related Commands

startirqasm

```
'startirqasm example
'Note that INT0 is on IO port 7
func main()
  global myvarb
  clear
```

```
'Start INT0 IRQ
irqglobalstart 'enable global IRQ's
irqINT0start() 'enable INT0 IRQ
```

```
loop:
  print myvarb
  pause 100
  goto loop
endfunc
```

```
'Must be placed outside functions
```

```
'-----
'IRQ handler for INT0 irq
'-----
```

```
startirqasm INT0
```

```
  inc16 myvarb,myvarb+1
```

```
endirqasm
```

```
'-----
```

endasm

D DP

Description

The endasm command takes the Dios out of assembly language mode. This command is used with the startasm command.

Related Commands

startasm

```
'endasm example
'This example will toggle the IO Port 0 at about 2.5 million
'times a second.
func main()
```

```
'-----
```

```
startasm
  output 0 'Set IOPORT 0 as output
```

```
loop
  high 0 'Turn Port on
  low 0 'Turn Port off
  goto loop
```

```
endasm
```

```
'-----
```

```
endfunc
```

exit

D DP

exit value
exit exp

Description

The exit command gives another way of exiting a function. You must specify a value when using the exit function.

- **value** is the value you want to return to the calling function.

Related Commands

func
endfunc
exitirq

```
'exit example
func main()
  dim tnum

  tnum = getnumber()

  print "The getnumber function returned ", tnum
endfunc

func getnumber()
  exit 2588
endfunc
```

exitirq

D DP

exitirqINAME
exitirqirqname

Description

When the IRQ handler function exits it will not be called again unless the onirq function is setup again. By using the exitirq command the function exits and automatically does an onirq command for you.

- **INAME** is the name of the IRQ that will trigger this function.

Valid IRQ names:

INT0
INT1
INT2
TMR0
TMR1
TMR2
TMR3
CCP1
CCP2
AD
RB
LVD
SSP
PSP
TX

Related Commands

func
endfunc
exit
exitirq
onirq

```
'exitirq example
func main()
  output 2
  global counter
  clear

  irqglobalstart
  irqperfstart
  onirqTMR0 irqtest

  irqTMR0start
  timer0mode16bit
  timer0sourceclock
  timer0prescale 4
  timer0on

  loop:
  toggle 2
  goto loop

endfunc

irqfunc irqtest()
  counter = counter + 1
  print counter
  exitirqTMR0
endirq
```

fabs

D **DP**

fabs fvarb
fabs fvarb

Description

Will remove the negative sign from a floating point variable.

Related Commands

int
 finv

```
'fabs example
func main()

    dim x as float

    x = -12.16
    print x
    fabs x
    print x

endfunc
```

finv

D **DP**

finv fvarb
finv fvarb

Description

Will invert the sign of a floating point variable

Related Commands

int
 fabs

```
'finv example
func main()

    dim x as float

    x = -12.16
    print x
    finv x
    print x

endfunc
```

for next

D DP

```
for varb = value1 to value2 step stepamount  
next
```

```
for ivarb = exp to exp step exp  
next
```

Description

The for command allows us to cycle through a variable with a start and stop point. You can count forward or backward by using a step -1 option. Unlike other compilers and chips the for next command in the Dios works much like high end compilers. If the counter variable is outside the bounds of value2 it will not execute. You can manipulate the counter variable. You can not change the direction once started.

- **varb** is the variable that we will use as our counter. This must be a variable. It cannot be a register. It must be an integer variable.
- **value1** is the starting expression. This is where the count will start.
- **value2** is the ending expression. This is where the count will begin.
- **stepamount** is an expression representing the amount to step. Any amount may be expressed from 0 to 65535. If stepamount is preceded by a - then the command will step backwards.

Related Commands

none

```
'for next example  
func main()  
  
dim count  
  
for count = 1 to 5  
  print "count=", count  
next  
  
for count = 5 to 1 step -1  
  print "count=",count  
next  
  
endfunc
```

func

```
func name(variable declarations) [as float][as integer]
```

Description

The func command is used to identify functions. You can also define local variables to be passed in the function.

- **name** is the name the function will be given.
- **variable** can be declared in the variable declaration section of the func command.

Note that even though a variable is defined it is optional that the calling program will actually pass a value. If the calling program does not pass a value the variable will contain random data until you write to it.

To define a function that returns float values use the "as float" option at the end of the function.

Related Commands

exit
exitirq
endfunc

```
'func example  
func main()  
  
dim retnum  
  
retnum = sqr(5)  
print "Result of call: retnum=",retnum  
  
endfunc  
  
func sqr(dat)  
  exit dat*dat  
endfunc
```

fuzzy

D DP

fuzzy varb,searchvalue,item1A,item1B,item2A,item2B,item3A,item3B...
 fuzzy ivarb,exp,exp,exp,exp,exp,exp...

Description

The fuzzy caommand tests each value pair in the list and will return the index of the found pair in the passed variable. The value must fall on or between the pair

The pair command has two modes of operation which is set using the lookdownmode command.

mode 0 - The index returned is 0-n and the variable will remain unchanged if the searchvalue is not found. This is the default mode.

mode 1 - The index is 1-n and will return 0 if not found.

The mode is changed with the lookdownmode command.

- **varb** is where the result of the search will be placed.
- **searchvalue** is the value that must match one of the items in our list.
- **item1-n** is the values that we will try to match against testvalue. There is no limit to the number of expressions you can have in the lookdown command. Note that these muse be supplied in pairs.

Related Commands

lookup
 lookdown
 lookdownmode

```
'fuzzy example
func main()

  dim x as integer
  dim y as integer

  lookdownmode 1
  y = 650

  fuzzy x,650, 400,500, 501,502, 601,700

  print y," Falls Between set ",x

endfunc
```

getbyte

D DP

getbyte address,resultvarb
 getbyte exp,ivarb

Description

Used to pull data directly from program memory.

- **address** - The 16-bit address of the byte of data we want to read.
- **resultvarb** - Integer variable that we use to store the byte retrieved.

```
'getbyte example
func main()

  disptext("hello")

endfunc

func disptext(addr)
  dim x,y

  print "addr=",addr
  x = 0

loop:
  getbyte addr + x,y
  if y = 0 then exit 0
  debug y
  x = x + 1
  goto loop

endfunc
```

getpacket

D DP

getpacket faillabel,indexvarb,valuevarb1,valuevarb2...
getpacket label,ivarb,ivarb,ivarb...

Description

This command will collect data from remote device. Once all data elements are filled it will fall through. This command works with the hardware UART so only port 8 is used. If all your packet elements have not been received the command will jump to the fail label that you provide.

- **faillabel** - The location to jump to if all the data elements have not been populated.
- **indexvarb** - Variable that will contain the current index (byte number) of the populated data.
- **valuevarb1-valuevarbn** - The variables to be populated.

```
'getpacket example
func main()

hsrsetup baud,HBAUD9600,start,clear

dim cmdidx,dat1,dat2,dat3
clear

loop:
getpacket loop,cmdidx,dat1,dat2,dat3
print dat1," ",dat2," ",dat3
goto loop

endfunc
```

getvpacket

D DP

getvpacket faillabel,timervarb,indexvarb
getvpacket label,ivarb,ivarb

Description

This command will collect data from remote device similar to the getpacket command. The very first byte recieved is the packet length byte. This byte determines the length of the packet.

If all the bytes as indicated from the byte field have not been received the command will jump to the fail label that you provide. The command will also jump to this label if no data is recieved.

The timervarb variable is used to keep track of the number of times the command is called since the last byte of data has been recieved.

You only pass the index variable. All the other bytes you expect to receive must also be defined directly following the definition of the index variable.

This command works with the hardware UART so only port 8 is used.

```
DiosPro
'getvpacket example
func main()

hsrsetup baud,HBAUD115200,start,clear

dim timervarb,cmdidx,bytecount,dat1,dat2,dat3
clear

loop:
getvpacket loop,timervarb,cmdidx
print dat1," ",dat2," ",dat3

goto loop

endfunc
```

- **faillabel** - The location to jump to if all the data elements have not been populated.
- **timervarb** - Variable that will keep track of the number of times the getvpacket command is called. This variable should be set to 0 the first time. If no data is received in 5125 calls to the the getvpacket command this counter will reset the command index byte counters thus resetting the command.
- **indexvarb** - Variable that will contain the current index (byte number) of the populated data. All the remaining variables used to store the packet must be defined using the dim command following the indexvarb variable.

Example

In the example shown you would send the the following bytes:

```
4 65 66 67
```

The first byte indicates the total number of bytes in the packet. In this case 4 bytes total (counting the packet length byte). The next three bytes are the data bytes.

getidpacket



getidpacket faillabel,timervarb,indexvarb,targetid
getidpacket label,ivarb,ivarb,exp

Description

This command will collect data from remote device similar to the getvpacket command. The very first byte recieved is the packet length byte. This byte determines the length of the packet. The second byte is the target id. It must match the targetid value passed to the command (Last Argument).

If all the bytes as indicated from the byte field have not been received the command will jump to the fail label that you provide. The command will also jump to this label if no data is recieved.

The timervarb variable is used to keep track of the number of times the command is called since the last byte of data has been recieved.

You only pass the index variable. All the other bytes you expect to recieve must also be defined directly following the definition of the index variable.

This command works with the hardware UART so only port 8 is used.

- **faillabel** - The location to jump to if all the data elements have not been populated.
- **timervarb** - Variable that will keep track of the number of times the getvpacket command is called. This variable should be set to 0 the first time. If no data is received in 5125 calls to the the getvpacket command this counter will reset the command index byte counters thus resetting the command.
- **indexvarb** - Variable that will contain the current index (byte number) of the populated data. All the remaining variables used to store the packet must be defined using the dim command following the indexvarb variable.
- **targetid** - This is the target ID. IF this ID does not match the 2nd byte or the 2nd byte is not 255 this command will not pass through and will jump to the fail label.

```
DiosPro
'getidpacket example
func main()

hsersetup baud,HBAUD115200,start,clear

dim timervarb,cmdidx,bytecount,device,dat1,dat2,dat3
clear

loop:
'print ". ";
getidpacket loop,timervarb,cmdidx,10
print dat1," ",dat2," ",dat3
goto loop

endfunc
```

Example

In the example shown you would send the the following bytes:

```
5 10 65 66 67
```

The first byte indicates the total number of bytes in the packet. In this case 5 bytes total (counting the packet length byte and target id byte).

The second byte is the target ID byte.

The next three bytes are the data bytes.

getstringbyte

getstringbyte addressvarb,resultvarb,donelabel
getstringbyte ivarb,ivarb,label

Description

When a string, table or text is passed to a function the getbyte command is used to access the text data. The command getstringbyte works much like getbyte except some processing is automatically done for you. For example when a value of 0 is found the program jumps to the label given. This keeps you from having to test for 0. Also the address variable is automatically incremented.

- **addressvarb** is an integer variable indicating the address where the data is located. This variable will be automatically incremented after each call.
- **resultvarb** is an integer variable that we use to store the byte retrieved.
- **donelabel** if an end of string terminator is detected, the program will jump to this label.

```
'getstringbyte example
func main()
  global myvarb(30) as string
  table line1: "One more time"
  myvarb="And once more"

  printstring("Now is the time") 'Send a literal
  printstring(line1) 'Send a table
  printstring(myvarb) 'Send a variable
endfunc

func printstring(addr)
  dim v

loop:
  getstringbyte addr,v,done
  debug v
goto loop

done:
  print

endfunc
```

global

D DP

global variable,variable,variable....
global varb....
global varb as integer
global varb as float

Description

This command works the same way as the dim statement except any function can access global variables.

Related Commands

dim
clear

```
' global example
func main()

  global x as integer
  global y as float

endfunc
```

gosub

D DP

```
gosub label
gosub label
```

Description

The gosub command allows us to create mini subroutines within a function. A gosub is always paired with a return statement. You can only gosub to a label within the same function.

- **label** is the name of the label(location) to jump to.

Related Commands

```
goto
return
branch
```

```
'gosub example
func main()

    gosub cat
    gosub dog
    gosub rat
    gosub bird

    print "All done"

alldone:
    goto alldone

cat:
    print "Cat"
    return
dog:
    print "Dog"
    return
rat:
    print "Rat"
    return
bird:
    print "Bird"
    return

endfunc
```

goto

D DP

```
gotob label
goto label
```

Description

The goto command allows to change program flow and unconditionally jump to any location within the current function.

- **label** is the name of the label(location) to jump to.

Related Commands

```
gosub
return
branch
```

```
'goto example
func main()

    goto skipstuff

    print "The program will never make it here."

skipstuff:
    print "we skipped some code"

endfunc
```

high

D DP

high port,port,port...
high exp,exp,exp...

Description

The high command is used to set the state of a port to high.

- **port** is the port number of the IO port to change.

Note that you can update as many ports as you like using the “;” to separate the port numbers. The high command will only work if the port is set up for output.

Related Commands

low
input
output
toggle
ioport

```
'high example
func main()
  output 0,1
loop:
  high 0,1
  low 0,1

goto loop

endfunc
```

hserin

D DP

hserin label,resultvar1,resultvar2,....
hserin label,ivarb,ivarb,....

Description

Retrieve a character from the input buffer from the hardware serial port.

- **label** is the location to jump to if no character is available in the buffer. There is a default timeout of a few milliseconds to look for a character in the buffer. However this can be extended or eliminated with the hsersetup command.
- **resultvar1-n** is where the character will be placed if one is available. Any number of result variables can be looked for.

Note that the hserin command is buffered and interrupt driven. And you don't need to be polling when data is received by the chip. There are 256 bytes allocated to the receive buffer and its management is performed automatically by the Dios.

The baud rate for the hardware serial port is set to 96k as a default. See the hsersetup command for enabling and setting up the hardware serial port.

Related Commands

hserout
hsersetup

```
'hserin example
func main()
  dim val
  hsersetup baud,HBAUD9600,start,txon

nodata:
  hserin nodata,val
  hserout "Recieved ",dec val,13,10
  goto nodata

endfunc
```

hserout

D DP

```
hserout value1,value2 ....
hserout exp,exp ....
```

Description

Sends data to the hardware serial port.

- **value1-value_n** is the data to be sent to the hardware serial port. Separate values with a ,

You can send a string of characters to the serial port by using quotes. You can precede the value with the **dec** modifier. This will convert the raw value to an ASCII representation so it can be displayed.

The format operators may also be used to change the numeric format. See the print command.

For example, `hserout 65` will write the character A to the terminal or serial LCD. But the command `hserout dec 65` will output the actual characters 6 and 5 to the terminal or serial port. Note that the actual format of the dec command can be changed by the decmask command.

The baud rate for the hardware serial port is set to 96k as a default. See the hsersetup command for enabling and setting up the hardware serial port.

You can use setdecmask and setfloatmask to change the formatting of numeric printout.

See the print command for a complete listing of modifiers

Related Commands

```
print
serout
debug
strout
decmask
hserin
hsersetup
setfloatmask
setdecmask
```

```
'hserout example
func main()

hsersetup baud,HBAUD9600,start,txon
hserout "Default option of 2: ",dec 1245,13,10

hserout "Default option of 2: ",{05} dec 1245,13,10

endfunc
```

hsersetup

hsersetup item,value
hsersetup item,exp

Description

Sets up various hardware serial options.

- **baud,value**

sets the baud rate for the hardware serial port. Always set the baud rate when you initialize the UART the first time.

Valid baud rates:

HBAUDAUTO*, HBAUD300* ,HBAUD1200* ,HBAUD2400,
HBAUD4800, HBAUD9600, HBAUD19200, HBAUD38400,
HBAUD57600, HBAUD115200, HBAUD230400,
HBAUD250000, HBAUD625000, HBAUD1250000

* DiosPro Only

- **start** Turns on the hardware serial port.
- **stop** Turns off the hardware serial port.
- **txon** Enables the transmit side of the hardware serial port. The txon and txoff options are provided so that a serial bus can be created. Under a bus structure the transmit side would be turned off unless information has been requested.
- **txoff** Places the hardware serial port in a high impedance state.
- **inwait,value** The number of milliseconds to wait for a byte to appear in buffer before timing out. Up to 65535 milliseconds may be defined.
- **txpad,value** The number of microseconds to wait between character transmissions. Up to 65535 microseconds may be defined.
- **clear** Clears the Receive buffer.
- **waitforbaud, milliseconds** When autoaud has been selected you can use this option to hold until the baudrate has been detected. Use 0 for unlimited timeout. Only Valid on DiosPro.

Note When the uart is activated both IO ports 8 and 9 are tied up even if the transmit port is turned off.

Note To use the AutoBaud setting with the DiosPro you must turn on the port first. The remote system must send a character U to set the baud rate.

Related Commands

hserout
hserin

```
'hsersetup example
func main()
  hsersetup baud,HBAUD115200,start,txon
  'or
  hsersetup start,txon

  hserout "Hello World"

endfunc
```

```
'hsersetup autobaud example
DiosPro
func main()
  hsersetup start,txon,baud,HBAUDAUTO,waitforbaud,0

  hserout "Hello World"

endfunc
```

inc

D DP

```
inc varb
inc ivarb
```

Description

Does the same thing as `varb = varb + 1`. It increments the variable by 1, but is much faster and uses less memory.

- **varb** The variable you wish to increment.

Related Commands

inc

```
'inc example
func main()

dim idx as integer
idx = 0

loop:
inc idx
print idx
goto loop

endfunc
```

int

D DP

```
int fvarb
int ivarb
```

Description

Will convert a floating point value to integer value. The Data is still a floating point number, only the data to the right of the decimal point is removed.

Related Commands

round
trunc

```
'int example
func main()
dim xyz as float

xyz = 100.6
trunc
print {6.2} xyz

int xyz
print {6.2} xyz

endfunc
```

input

D DP

```
input port,port...
input exp,exp...
```

Description

The input command sets indicated ports as input. When an IO ports is setup as input it is in a high impedance state. When connected to other devices you can read the actual state the port has been forced to.

- **port** is the port number of the IO lead to be set as input.

Related Commands

high
low
output
toggle
ioport

```
'input example
func main()

'set ports to input
input 0,1,2,3,4,5,6,7,8

endfunc
```

if then else endif

D DP

```
if expression1 operators expression2 then
```

```
.....
```

```
else
```

```
.....
```

```
endif
```

or

```
if expression1 operators expression2 then .....
```

Description

The if/then statements evaluate an expression(s) and executes all statements between the then and endif statement if found to be true. If you include the else statement the program will jump to the code just following the else statement if the expression is found to be false.

expression1 and **expression2** are the expressions to compare via the operators. They may contain any number of variables, constants, numbers, ioports or registers. The expressions may contain both integer and floating point numbers and variables.

Operators are used to indicate the comparison of expression 1 to expression 2.

Valid operators are:

= equal

==equal

<>not equal

><not equal

!=not equal

> Greater than

< Less than

>=Greater than equal to

<=Less than equal to

Note that if multiple comparisons are used they must be separated with and/or operators. For speed and code efficiency the and/or operators are handled in the order that they are encountered. If you want to combine and operators with or operators the and operators should be coded first. Parentheses are not allowed so multiple conditions are tested in the order encountered.

You can also have single line if then commands. Just place the commands you want to issue immediately following the then statement. Don't place an endif command. You cannot have else statements when doing one line if commands.

```
'if then else example
func main()

dim x
x = 21

if x > 10 and x < 50 then
  print "x is between 10 and 50"
endif

if x < 100 then
  print "x is less than 100"
else
  print "x is not less than 100"
endif

endfunc
```

include

D DP

include file

Description

It is not necessary to have your entire program in one file. You can also create library declaration files and include them in your program.

Insert the include statement where you want the file or library included. Usually we place libraries at the end of the program. Include statements may be nested; however watch out for recursive loops as this can crash the editor.

The full path to the library is determined using a few basic rules:

include filename

This will look for the filename at the same location as the program that is located.

include \filename

This will look for the filename in the main install directory of the Dios editor.

include \libfilename

This will look for the filename in the lib directory off of the main install directory of the Dios editor.

Notes:

You are not just limited to including functions. You can use include files to include variable definitions.

You can use multiple include statements pointing to the same file, but only the first will be loaded.

```
'include example
func main

  sayhello()

endfunc

include \lib\greetings.lib
```

'The greetings.lib located in the dios\lib dir contains:

```
func sayhello()
  print "hello"
endfunc

func saygoodbye()
  print "goodbye"
endfunc
```

ioport IOPORT

```
ioport(port)  
ioport(exp)
```

Description

This is a special register that is used to access the IO ports directly. It does not have an extension but each port is access by supplying the port number in the following format.

```
IOPORT(x)=y
```

where x is the port number and y is any value. If the value is greater than 0 then the port will be set otherwise it is reset.

You can also read the port by using the following format.

```
a=IOPORT(x)
```

Where x is the port number.

Note this is the only register that can be accessed as both lower and upper case.

- **port** is the number of the IO port to set as high or low.

Related Commands

```
portbitset  
portbitget  
high  
low
```

```
'ioport example  
func main()  
output 5  
  
loop:  
IOPORT(5)=0  
IOPORT(5)=1  
goto loop  
  
endfunc
```

irq...

irqINAMEclear
irqINAMEstart
irqINAMEstop
irqINAMEreset
irqINAMEoneshoton
irqINAMEoneshotoff
irqglobalstart
irqglobalstop
irqperipheralstart or *irqperfstart*
irqperipheralstop or *irqperfstop*

Description

Using these commands you can enable and disable Dios Interrupts.

- **INAME** - This represents the actual IRQ name. These commands are used in conjunction with the onirq and startirqasm commands. The use of these commands will cause the compiler to build a custom interrupt handler based on the commands in your software.

There are 16 names supported.

- INT0 - State change on IOport 7
- INT1 - State change on IO port 6
- INT2 - State change on IOport 5
- TMR0 - Overflow on timer 0's counter
- TMR1 - Overflow on timer 1's counter
- TMR2 - Overflow on timer 2's counter
- TMR3 - Overflow on timer 3's counter
- CCP1 - CCP1 counter capture has occurred
- CCP2 - CCP2 counter capture has occurred
- AD - The analog to digital conversion has completed
- RB - State change on any of the IO ports 0-7
- LVD - Dios voltage has dropped below preset voltage.
- SSP - Master Synchronous event has occurred. (SPI/I2c)
- PSP - Master Slave Port event has occurred
- RC - Receive event on built-in UART has occurred.
- TX - Transmit event on built-in UART has occurred.

Commands

- **clear** - Clears the internal hardware IRQ flag. This command may be needed to clear the flag before the IRQ is started.
- **start**- Starts the IRQ. It is also used to restart the IRQ if the oneshot option is enabled.
- **stop** - Stops the IRQ
- **reset** - Clears the internal software IRQ flag.
- **oneshoton** - If the IRQ is set to oneshot it will fire then turn its self off.

```
'Simple Timer0 IRQ Example
func main()
  global counter as float

  counter = 0

  irqglobalstart
  irqperfstart
  irqTMR2start

  timer2prescale 7
  timer2on

  onirqTMR2 myfunc0

loop:
  goto loop

endfunc

irqfunc myfunc0()
  counter = counter + 1
  print "counter=",counter
  exitirqTMR2
endirq
```

- **oneshotoff** - This command puts the IRQ back into continuous mode.
- **reset** - Clears the internal software IRQ flag.
- **irqglobalstart** - Turns on the use of IRQ's
- **irqglobalstop** - Turns off the use of IRQ's
- **irqperipheralstart** - Some IRQ's require this option in order to operate. You can also use the command **irqperfstart**.

IRQ's Requiring peripherals to be started:

TMR0
TMR1
TMR2
TMR3
CCP1
CCP2
SSP

- **irqperipheralstop** - Turns off peripheral interrupts. You can also use the command **irqperfstop**.

Related Commands

onirq...
exitirq...
startirqasm...

irqfunc

D DP

irqfunc name()
irqfunc functionname

Description

The irqfunc command is how we identify IRQ functions.

- **name** is the name the function will be given.

Some rules for IRQ functions

- You cannot pass variables to IRQ functions.
- You must end an IRQ function with an `endirq` command
- You cannot call IRQ functions directly. If you need to test, set them up as normal functions then rename them once debugged.

Related Commands

endirq
exitirq

```
irqfunc example
func main()
  output 2
  global counter
  clear

  irqglobalstart
  irqperfstart
  onirqTMR0 irqtest

  irqTMR0start
  timer0mode16bit
  timer0sourceclock
  timer0prescale 4
  timer0on

loop:
  toggle 2
  goto loop

endfunc

irqfunc irqtest()
  counter = counter + 1
  print counter
  exitirqTMR0
  endirq
```

Icdchar

D DP

Icdchar char
Icdchar exp

Description

Sends a raw character to the LCD. Note that you must initialize the LCD before using this command.

- **char** - The character to send to the LCD.

```
'Icdchar/Icdcontrol command example  
func main()
```

```
'Dios Carrier #4  
Icdinit 23,25,24,29,28,27,26
```

```
Icdcontrol 1  
Icdwrite "Hello"  
Icdcontrol 128+64  
Icdchar 'A'  
Icdchar 'B'
```

```
endfunc
```

Icdcontrol

D DP

Icdchar control
Icdchar exp

Description

Sends a control code to the LCD. Note that you must initialize the LCD before using this command.

- **control** - The control code to send to the LCD.

Valid codes:

- 1: Clear Display
- 2: Cursor Home
- 12: Cursor Off
- 14: Cursor On No Blink
- 15: Cursor On With Blink
- 16: Shift Cursor Left
- 20: Shift Cursor Right
- 24: Shift Display Left
- 28: Shift Display Right
- 64+x: Sets the character generator address. x=0-63
- 128+x: Sets the display address. x=0-127

Icdgoto

D DP

Icdgoto line,pos
Icdgoto exp,exp

Description

Sets the LCD for display at a particular position. Essentially this command just calculates the memory map location for a 2 line LCD display. While it can be used on 4 line LCD's you will have to add an offset value for lines 3 and 4.

- **line** - The line to display on. 0-1
- **pos** - The character position on that line.

```
'Icdgoto command example  
func main()
```

```
'Dios Carrier #4  
Icdinit 23,25,24,29,28,27,26
```

```
Icdwrite control 1, "Hello World"  
Icdgoto 2,1  
Icdwrite "Goodbye"
```

```
endfunc
```

Icdinit

D DP

Icdinit RSport,Eport,RWport,Bit0port,Bit1port,Bit2port,Bit3port
Icdinit exp,exp,exp,exp,exp,exp,exp

Description

This command is used to setup the internal lcd commands. It also initializes the LCD. Once this command has been issued it clears the LCD and all other LCD commands will automatically be sent to that LCD.

- **RSport** - The port connected to the RS lead (pin 4) on the LCD.
- **Eport** - The port connected to the E lead (pin 6) on the LCD.
- **RWport** - The port connected to the RW lead (pin 5) on the LCD.
- **Bit0port** - The port connected to the Data bit 0 (pin 11)on the LCD.
- **Bit1port** - The port connected to the Data bit 1 (pin 12)on the LCD.
- **Bit2port** - The port connected to the Data bit 2 (pin 13)on the LCD.
- **Bit3port** - The port connected to the Data bit 3 (pin 14)on the LCD.

```
'Icdinit command example
func main()

'Dios Carrier #4
Icdinit 23,25,24,29,28,27,26

Icdwrite control 1,"Hello World"
Icdgoto 2,1
Icdwrite "Goodbye"

endfunc
```

Icdwrite

D DP

Icdwrite value1,value2,...
Icdwrite exp,exp,.....

Description

This command works the same as the debug command except the output is sent to a lcd. You must use the Icdinit command first to setup the LCD and Dios IO ports.

LCD Control Codes

To send LCD control codes use the control modifier. For instance "Icdwrite control 1" will clear the lcd display.

- 1: Clear Display
- 2: Cursor Home
- 12:Cursor Off
- 14:Cursor On No Blink
- 15:Cursor On With Blink
- 16:Shift Cursor Left
- 20:Shift Cursor Right
- 24:Shift Display Left
- 28:Shift Display Right
- 64+x:Sets the character generator address. x=0-63
- 128+x:Sets the display address. x=0-127

Icdwrite control 128+64 sets the display to print on second line of 2line display.

See the print command for a complete listing of modifiers

```
'Icdwrite command example
func main()

'Dios Carrier #4
Icdinit 23,25,24,29,28,27,26

Icdwrite control 1,"Hello World"
Icdgoto 2,1
Icdwrite "Goodbye"

endfunc
```

Blank Page

lookup

D DP

lookup varb,index,item1,item2,item3,item4,item5...
lookup ivarb,exp,exp,exp,exp,exp,exp,exp...

Description

The lookup command takes the number in index and retrieves the nth item in the list.

- **varb** is the integer variable where the lookup value will be placed.
- **index** is the offset number to look for, 0-65535. If the index exceeds the highest given item, the variable will remain unchanged.
- **item1-n** are the expressions to evaluate for a given index. There is no limit to the number of expressions you can have in the lookup command.

Related Commands

Lookdown
fuzzy

```
'lookup example
func main()

dim index,a

print "Test 1"
for index = 0 to 10
  lookup a,index,100,21,14,15,99,33,44,55,71,99,101
  print a
next

print
print "Test 2"
for index = 0 to 7
  a = 60000 'Set to a value we can test for
  lookup a,index,10,20,30,40,50
  print a

  if a = 60000 then
    print "index ",index," out of range"
  endif
next

endfunc
```

lookdown

D DP

lookdown varb,searchvalue,item1,item2,item3,item4,item5...
lookdown ivarb,exp,exp,exp,exp,exp,exp,exp...

Description

The Lookdown tests each value in the list and will return the index of the found item in the passed variable.

The lookdown command has two modes of operation.

mode 0 - The index returned is 0-n and the variable will remain unchanged if the searchvalue is not found. This is the default mode.

mode 1 - The index is 1-n and will return 0 if not found.

The mode is changed with the lookdownmode command.

- **varb** is where the result of the search will be placed.
- **searchvalue** is the value that must match one of the items in our list.
- **item1-n** is the values that we will try to match against testvalue. There is no limit to the number of expressions you can have in the lookdown command.

Related Commands

lookup
lookdownmode
fuzzy

```
'lookdown example
func main()
  dim result as integer
  dim searchitem as integer
  dim value as integer

  lookdownmode 0

  searchitem = 10731 'This is the 6th item
  lookdown result,searchitem,21,15,33,1000,309,10731,45
  print "Result 1: ",result

endfunc
```

lookdownmode

D DP

lookdownmode mode
lookdownmode exp

Description

This command sets the way the lookdown command operates.

- **mode** sets the lookdown command mode of operation.

If mode is 0, the index returned is 0-n and the variable will remain unchanged if the searchvalue is not found. This is the default mode.

If mode is 1, the index returned is 1-n and will return 0 if the searchvalue is not found.

Related Commands

lookdown

```
'lookdown example
func main()
  dim result as integer
  dim searchitem as integer
  dim value as integer

  lookdownmode 1

  searchitem = 10731 'This is the 6th item
  lookdown result,searchitem,21,15,33,1000,309,10731,45,77,100
  print "Result 1: ",result

  searchitem = 10732 'We know this item is not in list
  lookdown result,searchitem,21,15,33,1000,309,10731,45,77,100
  print "Result 2: ",result

endfunc
```

low

D DP

low port,port,port....
low exp,exp,exp....

Description

The low command is used to set the state of a port to low.

- **port** is the number of the IO port to change.

Note that you can update as many ports as you like using the “,” to separate the port numbers. The low command will only work if the port is set up for output.

Related Commands

high
input
output
toggle

```
'low example
func main()
  output 0,1
loop:
  high 0,1
  low 0,1

goto loop

endfunc
```

memXread / memXwrite

D DP

mem1read address,variable
mem1read exp,ivarb

mem1write address,value
mem1write exp,exp

Description

Reads or writes a byte to or from a memory bank.

- **address** is the index into memory. A value of 0 to 255 is valid.
- **variable** is the name of the variable you wish to return the memory value. Used with mem1read.
- **value** is the value you wish to write to memory. Used only with mem1write.

Each memory space is described below. There are 5 and each contain 256 bytes of ram.

mem1read / mem1write

This space is used to hold the local function variables as they are used. Because the location of local variables change from function call to function call this memory bank should only be used by someone who fully understands the memory architecture of the Dios.

mem2read / mem2write

This space is used to hold global variable memory area. Variable are indexed in the order they are created.

mem3read / mem3write

This space is used to hold the main stack area. **The use of this memory area is not recommended.**

mem4read / mem4write

This space is used to hold the data as it is received from the UART. Use of this memory area is not recommended unless the UART is not in use. In that case it reverts to scratch pad memory use.

mem5read / mem5write

This space is used to hold string space. Use of this memory area is not recommended unless string variables are not in use. In that case it reverts to scratch pad memory use.

Note that the Dios Pro supports Banks 6-14 that can be used with memxread and memxwrite commands.

```
'mem1read / mem1write example
func main()
dim myvarb 'Very first memory spot
myvarb = 10
print "myvarb starts out as ",myvarb

testfunc()
print "after call myvarb= ",myvarb
endfunc

'This function will access a local variable in function main
func testfunc()
dim tempvarb,tempval

'----- Read it here -----
mem1read 0,tempval
tempvarb.byte(1)=tempval
mem1read 1,tempval
tempvarb.byte(0)=tempval
print "function testfunc accessed funcmains local variable myvarb=",tempvarb
print

'----- Change it here -----
mem1write 0,0 'Variable High byte
mem1write 1,100 'Variable low byte
endfunc
```

```
'mem2read / mem2write example
func main()
global myvarb 'Very first memory spot
dim tempvarb,tempval

myvarb = 10
print "myvarb starts out as ",myvarb

mem2read 0,tempval
tempvarb.byte(1)=tempval
mem2read 1,tempval
tempvarb.byte(0)=tempval
print "accessed global myvarb=",tempvarb

'Now change it
mem2write 0,0 'Variable High byte
mem2write 1,100 'Variable low byte

print "myvarb is now ",myvarb
endfunc
```

```
'mem4read / mem4write example
'mem5read / mem5write example
func main()
dim idx,memval

'Write a byte to scratch pad memory

print "write to scratch pad"
'Write to scratch pad area
for idx = 1 to 5
memval = 100+idx
mem5write idx,memval
print "write ",memval
next

print "read from scratch pad"
'Read from scratch pad memory
for idx = 1 to 5
mem5read idx,memval
print "read ",memval
next

endfunc
```

nop

D DP

nop

Description

The nop command does nothing but spends time. I have added the command as just another way of providing timing.

Related Commands

none

```
'nop example
func main()

loop:
  high 0
  nop
  low 0
  goto loop

endfunc
```

onirq

D DP

onirq|NAME functionname
onirq|irqname functionname

Description

There are 15 hardware interrupts that can be generated. You can poll for them by checking the bits of various flags, or you can set up a function to catch them.

- **INAME** is the name of the IRQ that will trigger the function call.

Valid IRQ names:

INT0
 INT1
 INT2
 TMR0
 TMR1
 TMR2
 TMR3
 CCP1
 CCP2
 AD
 RB
 LVD
 SSP
 PSP
 TX

- **functionname** is the name of your catch-all interrupt routine.

Related Commands

exitirq
 func

Note this command replaces the original onirq command. This one is faster and uses less memory. While the old onirq command is still supported the syntax on this one is slightly different. Notice that the IRQname is now part of the command name and there is no comma between the IRQ name and the function name,

```
'onirq example
func main()
  output 2
  global counter
  clear

  irqglobalstart
  irqperfstart
  onirqTMR0 irqtest

  irqTMR0start
  timer0mode16bit
  timer0sourceclock
  timer0prescale 4
  timer0on

loop:
  toggle 2
  goto loop

endfunc

irqfunc irqtest()
  counter = counter + 1
  print counter
  exitirqTMR0
endirq
```

odlow

D DP

```
odlow port,port,port....  
odlow exp,exp,exp....
```

Description

The low command is used to set the state of a port to open drain low. This command is used in conjunction with the odhigh command. The port must be held high with a 4.7k resistor. There is no need to set the port with the input and output commands, as it is done for you.

- **port** - The port number of the IO port to change.

```
'odhigh / odlow example  
func main()  
loop:  
  odhigh 0  
  odlow 0  
  
goto loop  
  
endfunc
```

odhigh

D DP

```
odhigh port,port,port....  
odhigh exp,exp,exp....
```

Description

The high command is used to set the state of a port to open drain high (release). This command is used in conjunction with the odlow command. The port must be held high with a 4.7k resistor. There is no need to set the port with the input and output commands, as it is done for you.

- **port** - The port number of the IO port to change.

```
'onportgoto example  
func main()  
again:  
  onportgoto 4,itslow,itshigh  
  
itslow:  
  print "Low"  
  goto again  
  
itshigh:  
  print "High"  
  goto again  
  
endfunc
```

onportgoto

D DP

```
onportgoto port,lowlabel,highlabel  
onportgoto exp,label,lebel
```

Description

Will branch (goto) to a location based on the state of an IOport.

- **port** - The port number to test
- **lowlabel** - The location to jump if the port is in the low state.
- **highlabel** - The location to jump if the port is in the high state.

```
'onportgosub example  
func main()  
again:  
  onportgosub 4,itslow,itshigh  
  print "....."  
  goto again  
  
itslow:  
  print "Low";  
  return  
  
itshigh:  
  print "High";  
  return  
  
endfunc
```

onportgosub

D DP

```
onportgosub port,lowlabel,highlabel  
onportgosub exp,label,lebel
```

Description

Will perform a call (gosub) to a location based on the state of an IO port. Once the return is encountered the program will resume at the command following the onportgosub

- **port** - The port number to test
- **lowlabel** - The location to call if the port is in the low state.
- **highlabel** - The location to call if the port is in the high state.

output

D DP

output port,port,port....
output exp,exp,exp....

Description

The output command sets the ports as output.

port is the number of the IO port to be set as output. Once set as output you can use the high, low, toggle, IO port commands to set the actual level to high or low.

Related Commands

high
 low
 input
 toggle
 ioport

```
'output example
func main()

'set the ports to output
output 0,1,2,3,4,5,6,7,8

endfunc
```

pause

D DP

pause time,time,time....
pause exp,exp,exp....

Description

The pause command will wait for the number of milliseconds indicated.

- **time** is the number of milliseconds to wait.

Note that you can cascade the time intervals to wait longer times.

Related Commands

pauseus

```
'pause example
func main()

print "Waiting for 1 second"
pause 1000

print "Done waiting"

endfunc
```

pauseus

D DP

pauseus time,time,time....
pauseus exp,exp,exp....

Description

The pauseus command will wait for approximately 1 microsecond for each unit of time. Each command has a setup and execute time. The average is 10us. These times will be added to the delay.

```
high 0
pauseus 100
low 0
```

This example will be in the high state for about 130us.

- **time** is the number of microsecond units to wait.

Related Commands

pause

```
'pauseus example
func main()

again:
high 0
pauseus 100
low 0
pauseus 100

goto again
endfunc
```

portget

D DP

portget resultvariable
portget ivarb

Description

Sets the bits in the result variable based on the states of IO ports 0 - 7.

Note that portget should not be used if you are using IO port 3 as a transmit program port. Use customportget instead.

- **resultvariable** is where the data will be stored.

Related Commands

portset
customportset
customportget

```
'portget example
func main()
  input 0,1,2,3,4,5,6,7

  dim dat

  portget dat
  print "ports 0-7=",dat

endfunc
```

portset

D DP

portset expression
portset exp

Description

Takes the value in expression and sets the IO ports 0-7 based on the bit values in expression.

- **expression** Takes the first 8 bits of this value and sets ports 0-7

Note that portset should not be used if you are using IO port 3 as a transmit program port. Use customportset instead.

Related Commands

portget
customportset
customportget

```
'portset example
func main()
  output 0,1,2,3,4,5,6,7

  "Turns on all 8 ports
  portset 255

endfunc
```

portbitget

D DP

portbitget resultvarb,bit,port
portbitget ivarb,exp,exp

Description

This routine sets the indicated bit in the result variable if the pin is high or resets the bit if the pin is low.

- **resultvarb** - The variable that will have its bits modified.
- **bit** - The bit in the result variable that will be changed. Bits 0-15 are valid.
- **port** is the IO port to test for a high or low condition. If this port is high the indicated bit in the target variable will be set.

Notice in example 1 we use the portbitget command. In example 2 we used an alternate method to do the same thing.

Example 1
 Program Size = 102
 Speed about 40% faster

Example 2
 program size = 114

Related Commands

portbitset

portbitset

D DP

portbitset value,bit,port
portbitset exp,exp,exp

Description

Sets an IO port to the high or low state depending on the state of the bit in the given expression.

- **value** - The expression that is to be tested.
- **bit** - The bit in the expression that will be tested. Bits 0-15 are valid. Note that if you use math the expression will be calculated first then the result will be tested.
- **port** - The number of the IO port to be set as high or low.

Related Commands

portbitget

```
'portbitget Example 1
func main()
dim result
const port4 4
const port5 5
const port6 6
const port7 7

input port4,port5,port6,port7 'Use Ultra Board Buttons
clear

again:
portbitget result,0,port4
portbitget result,1,port5
portbitget result,2,port6
portbitget result,3,port7

print "Result Variable = ",result

goto again

endfunc
```

```
'portbitget Example 2
'In this example we show an alternate way of setting the
'bits in a variable based on IO port status.
func main()

dim result
const port4 4
const port5 5
const port6 6
const port7 7

input port4,port5,port6,port7 'Use Ultra Board Buttons
clear

again:

result.bit(0)=iport(port4)
result.bit(1)=iport(port5)
result.bit(2)=iport(port6)
result.bit(3)=iport(port7)

print "Result Variable = ",result

goto again

endfunc
```

```
'portbitset Example
func main()
dim myvarb,mybit

const port0 0
output port0

mybit = 0

'If mybit is:
'0 then IO port will change every loop of port
'1 then IO port will change every 2 loops
'2 then IO port will change every 4 loops
again:
inc myvarb
portbitset myvarb,mybit,port0
goto again

endfunc
```

print

D DP

```
print value1,value2 ....  
print exp,exp ....
```

Description

This is the same command as the debug command except that a cr/lf is always added to the end. All numbers are converted to decimal so they can be displayed.

Note that you can keep the cr/lf from being added to the print statement by adding a ; at the end of the command.

All commands that send output to the debug port have a 256us delay added to accommodate slower machines and some USB com ports. This can be changed by changing the value in the DEBUGPADL and DEBUGPADH registers. This value is in microseconds.

```
'print example  
func main()  
dim age  
age = 25  
  
print "Hello ";  
print "World"  
print  
print "I am ",age," years old"  
  
endfunc
```

Numeric Output Format Modifiers

You can change the way that floating point and integers display with the print command. You do this with the format operators. Once the format has been changed it remains in the new format until changed again or the Dios is restarted or reset.

dec/noddec modifiers

When a value is sent to the output processor the raw value is sent. IE a value of 65 will display an A. To force the output processor to display the value instead use the **dec** modifier. The print command already does the **dec** modifier for you on all values. If you dont want the forced dec modifier use the **noddec** modifier.

Floating Point Format Syntax

{-D.P}

- **P** - This field designates the number of digits on the right side of the decimal point with a maximum of 3. If 0 or not supplied the number will not display a decimal point and all digits to the right will be dropped.

Note if the round flag has been set the number will be rounded up if the value to the right is .5 or higher.

- **D** - This indicates how many digits are to be displayed. If this field is 0 or not supplied the all digits will be displayed as needed and will be left justified.

A value of 1-7 will display only the number of digits indicated. The digits will be right justified. If a 0 is located in this field the unpopulated slots to the left of the number will be populated with 0's

Note that a space is provided for the sign field. unless the - is detected in the D field. If a - is detected the sign will not be

```
'print command format example  
func main()  
  
'Floating Point  
print {04.2} -1.08 'Prints -01.08  
print {-04.2} -1.08 'Prints 01.08  
  
'Integer  
print {04} 12 'Prints 0012  
  
endfunc
```

displayed.

Integer Format Syntax

{D}

- **D** - This indicates how many digits are to be displayed. If this field is blank (Only {} supplied) then all digits will be displayed as needed and will be left justified.

A value of 1-5 will display only the number of digits indicated. The digits will be right justified. If a 0 is located in this field the unpopulated slots to the left of the number will be populated with 0's

forcing modifier

This modifier will force a integer variable to become a string address and the whole string to be displayed.

Note that the format operators can be used on the serout, hserout debug, lcdwrite and strout commands. If used with these commands you will need to use the dec command to display the numeric data.

Related Commands

- serout
- hserout
- debug
- strout
- setdecmask
- setfloatmask
- lcdwrite

pot

D **DP**

pot port,resultvarb
 pot exp,ivarb

Description

Places the indicated port into output mode and high for 5ms then places the port into input mode. The port is then polled and a counter is incremented until the port goes into low state. Returns 1-65535 or 0 if port times out.

Place a capacitor between the IOport and Vss. Place a potentiometer between the IO port and Vss.

Start with a .1 capacitor and a 50K potentiometer. You can change the values to affect the range.

- **port** - The port to take reading on.
- **resultvarb** - The integer variable to place result into.

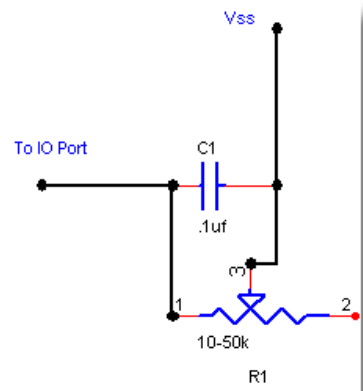
```
'Dios pot command Example
'Place a capacitor between port and Vss.
'Place a Pot between port and Vss.
func main()

    dim x

loop:

    pot 0,x
    print x
    pause 5
    goto loop

endfunc
```



pulsein

D DP

pulsein port,state,timeout,variable
pulsein exp,exp,exp,ivarb

Description

Counts the time interval between pulses.
Returns the number .1us units of a pulse. 0-65535.

- **port** - The IO port number to use.
- **state** - The state of the pulse to count. This field also indicates if the count should start after 1 full cycle.

0: measure interval of next low state.
1: Measure interval of next high state.
2: Measure interval of next low state after one complete cycle.
3: Measure interval of next high state after one complete cycle.
- **timeout** - The number of passes to make looking for the port to change state.
- **variable** - The place to store the result.

The output is in .1us units so for microseconds divide result by 10. This command uses timer 0 for reference.

The pulsein command has a special register that lets you set up a divisor. This will allow you to measure longer pulses.

PULSEINSCALE Register values

- 0 - Divide by 2 (.2us units)
- 1 - Divide by 4 (.4us units)
- 2 - Divide by 8 (.8us units)
- 3 - Divide by 16 (1.6us units)
- 4 - Divide by 32 (3.2us units)
- 6 - Divide by 64 (6.4us units)
- 7 - Divide by 128 (12.8us units)
- 8 - No Divisor (.1us units) *Default

Related Commands

pulseout

```
'pulsein example
func main()
dim pulse_pos as integer
dim pulse_neg as integer
dim pulse_total as integer

'Create some pulses with built-in PWM generator
PWMinit(1)
PWMperiod(255)
PWMcourse(2)
PWM1duty(100)

PULSEINSCALE = 8 'Default
loop:
pulsein 3,2,5000,pulse_pos

if pulse_pos = 0 then
goto loop
endif

pulsein 3,3,5000,pulse_neg

if pulse_neg = 0 then
goto loop
endif

'Some integer math
pulse_pos = pulse_pos / 10
pulse_neg = pulse_neg / 10
pulse_total = pulse_pos + pulse_neg

'Display Results
print
print "Pulsein Results"
print "Pulse High ",pulse_pos,"us"
print " Pulse Low ",pulse_neg,"us"
print " Period ",pulse_total,"us"
print " Frequency ";
print {-0.3} 1/pulse_total*1000000/1000;
print " KHz"

pause 2000

goto loop

endfunc

include \lib\DiosHWPWM.lib
```

pulseout

D DP

pulseout port,interval
pulseout exp,exp

Description

Toggles a port from current state to the other then back again.

- **port** is the number of the IO port to toggle.
- **interval** is the time in cycles to pause between the two states. The interval value is in microseconds and can be between 0-65365. Note that the pulseout command has an overhead of 2.8us. Therefore if you issue the command pulseout x,1 you will get a 3.8us pulse and pulseout x,2 will yeld 3.8us and so on.

Related Commands

pulsein

```
'pulseout example
func main()
  output 0

  loop:
    pulseout 0,1000 '1 milisecond
    goto loop

endfunc
```

pullupon

D DP

pullupon

Description

Turns on the built-in weak pullup resistors on ports 0-7. This is used when one or more ports are set to input. This will hold the ports to a high (1) state until pulled low with button or other device.

Related Commands

pullupoff

```
'pullupon example
func main()
  input 0,1,2,3,4,5,6,7
  pullupon

endfunc
```

pullupoff

D DP

pullupoff

Description

Turns off the built-in weak pullup resistors on ports 0-7. This is used when one or more ports are set to input. This will hold the ports to a high (1) state until pulled low with button or other device.

Related Commands

pullupon

```
'pullupoff example
func main()
  input 0,1,2,3,4,5,6,7
  pullupoff

endfunc
```

rccount

D DP

rccount port,state,resultvarb
pulseout exp,exp,ivarb

Description

Places the port into input mode. The port is then polled and a counter is incremented until the port goes into indicated state. Returns 1-65535 or 0 if port times out.

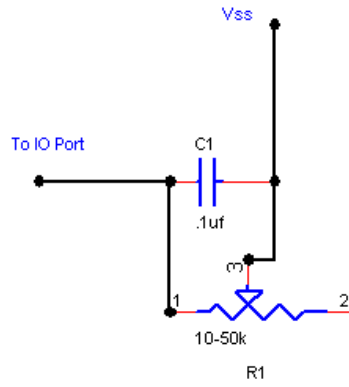
Place a capacitor between the IOport and Vss. Place a potentiometer between the IO port and Vss.

Start with a .1 capacitor and a 50K potentiometer. You can change the values to affect the range.

- **port** - The port to take reading on.
- **state** - The state to look for. 0-1
- **resultvarb** - The integer variable to place result into.

Related Commands

pot



'Dios rccount command Example
'Place a capacitor between port and Vss.
'Place a Pot between port and Vss.
func main()

```
dim x
```

```
loop:
```

```
output 0  
high 0  
pause 5  
rccount 0,0,x  
print x  
pause 5  
goto loop
```

```
endfunc
```

readfromscratchX

D DP

readfromscratchX iexp,iexp,varb
readfromscratchX scratchstart,bytes,varb

Description

This command will copy a certain number of bytes from an upper memory bank to a local or global variable. The bank is indicated by the X. Each bank has 255 bytes available. Keep in mind that each integer is 2 bytes and each floating point is 4 bytes.

Available banks:

Dios Banks 5

DiosPro Banks 5-14

- **scratchstart** - The starting point to copy to. 0 to 255
- **bytes** - The number of bytes to copy. You should copy 2 bytes for each integer and 4 for each floating point.
- **varb** - The variable to copy.

Important

You may supply enough bytes to copy more than 1 variable.

Related Commands

savetoscratchX

'savetoscratchX example

```
func main()  
dim z as integer
```

```
dim x(5) as integer  
for z = 0 to 4  
x(z) = z+100  
next
```

```
savetoscratch5 1,10,x
```

```
for z = 0 to 4  
x(z) = 0  
next
```

```
readfromscratch5 1,10,x
```

```
for z = 0 to 4  
print z, " ",x(z)  
next
```

```
endfunc
```

return

D DP

return

Description

Returns from a gosub statement.

Related Commands

gosub

round

D DP

round

Description

The floating point math routines may need to be told to round or truncate when certain calculations are made. The round command sets the internal floating point routines to round floating point calculations.

Related Commandsround
trunc
int

```
'return example
func main()

gosub cat
gosub dog
gosub rat
gosub bird

print "All done"

alldone:
  goto alldone

cat:
  print "Cat"
  return
dog:
  print "Dog"
  return
rat:
  print "Rat"
  return
bird:
  print "Bird"
  return

endfunc
```

```
'round example
func main()
  dim xyz as float

  xyz = 100.6

  print {6.2} xyz
  round
  int xyz
  print {6.2} xyz

endfunc
```

savetoscratchX

D DP

```
savetoscratchX iexp,iexp,varb  
savetoscratchX scratchstart,bytes,varb
```

Description

This command will copy a certain number of bytes from a local or global variable to a upper memory bank. The bank is indicated by the X. Each bank has 255 bytes available. Keep in mind that each integer is 2 bytes and each floating point is 4 bytes.

Available banks:

Dios Banks 5

DiosPro Banks 5-14

- **scratchstart** - The starting point to copy to. 0 to 255
- **bytes** - The number of bytes to copy. You should copy 2 bytes for each integer and 4 for each floating point.
- **varb** - The variable to copy.

Important

You may supply enough bytes to copy more than 1 variable.

Related Commands

readfromscratchX

```
'savetoscratchX example  
func main()  
  dim z as integer  
  
  dim x(5) as integer  
  for z = 0 to 4  
    x(z) = z+100  
  next  
  
  savetoscratch5 1,10,x  
  
  for z = 0 to 4  
    x(z) = 0  
  next  
  
  readfromscratch5 1,10,x  
  
  for z = 0 to 4  
    print z," ",x(z)  
  next  
  
endfunc
```

serin

D DP

serin port,label,resultvarb1,resultvarb2,resultvarb3....
 serin exp,label,ivarb,ivarb,ivarb,....

Description

Waits for serial input.

- **port** is the IO port you want to wait for serial data.
- **lable** is the location to jump to in case of a timeout.
- **resultvarb1-n** are the variables to store the data.

The baud rate for the software serial port is set to 96k as a default.
 See the **sersetup** command for setting up the software serial port.

The port used for the serin command should be setup as input.

Related Commands

serout
 sersetup

```
'serin example
'Look for input on serial port 0 and
'send it out on port 1
func main()
  dim val

  input 0 'Input port needs to be set as input
  output 1 'Output port should be set as output
  high 1 'Output port should also be set to high state

  sersetup baud,SBAUD2400

  nodata:
  serin 0,nodata,val
  serout 1,"Recieved ",dec val,13,10
  goto nodata

endfunc
```

serout

D DP

serout port,value1,value2

serout exp,exp,.....

Description

Sends serial data to any IO port via the software.

- **port** is the number of the IO port you want to send serial data to.
- **value1-value_n** is the data to be sent to the port. Separate values with a coma.

You can send a string, table or quoted text to the serial port.

You can precede the value with the modifier dec. This will convert the raw value to an ASCII representation so it can be displayed.

For example, hserout 65 will write the character A to the terminal or serial LCD. But the command hserout dec 65 will output the actual characters 6 and 5 to the terminal or serial port. Note that the actual format of the dec command can be changed by the decmask command.

The format operators may also be used to change the numeric format. See the print command.

The baud rate for the software serial ports is set to 96k as a default. To change the value, use the sersetup command.

You can use setdecmask and setfloatmask to change the formatting of numeric printout.

See the print command for a complete listing of modifiers.

Related Commands

print
hserout
debug
strout
setdecmask
setfloatmask

```
'serout example
func main()

  output 1 'Serial output port should be set as output
  high 1 'Serial output port should also be set as high

  sersetup baud,SBAUD2400

  serout 1,"Hello World",13,10

endfunc
```



sersetup

sersetup item,value,item,value,item,value
 sersetup name,const,name,const,name,const,....

Description

Sets up various software serial options. Note that all the items are optional.

valid item names:

- **baud** - Sets the baud rate for software serial ports. The baud rate defaults to 9600 when the Dios is powered up.

Valid baud rates:

- SBAUD1200
- SBAUD2400
- SBAUD4800
- SBAUD9600
- SBAUD19200
- SBAUD38400
- SBAUD57600
- SBAUD115200

- **inwait** - The number of milliseconds to wait for a byte to appear before timing out. Up to 65535 milliseconds may be defined.
- **txpad** - The number of microseconds to wait between character transmissions. Up to 65535 microseconds may be defined.

Related Commands

serout
 serin

```
'sersetup example
func main()

  output 1 'Serial output port should be set as output
  high 1 'Serial output port should also be set as high

  sersetup baud,SBAUD2400

  serout 1,"Hello World",13,10

endfunc
```

setdecmask

D DP

setdecmask item,value,item,value....

setdecmask exp,exp,exp,exp

Description

The setdecmask command allows you to change the way that integer values are output with the print,debug,setout,hsrout, and strout commands. You pass a parameter identifier followed by a value.

Note that you can supply any number of parameters to this command

Valid Identifiers

display,mask (0-31)

• Bits 0-4

Digit masks. These bits determine which digits get displayed. If you only want the last two digits displayed you would use 00011. For all digits use the default of 11111.

leaddigit,exp (1 or 0)

Display spaces. When leading digits are enabled we can display them as 0's or spaces. If this value is 1, they will be spaces.

lead,exp (1 or 0)

Display leading characters. When a number that is less than 10000 is displayed we must decide what to do with the leading digits. We can display them or discard them. If this value is 0 we will discard them (default). However if this bit is 1 we will display leading digits.

Related Commands

print
debug
hsrout
strout
serout
setfloatmask

```
'setdecmask example
func main()

    setdecmask display,7,lead,1,leaddigit,1

    print 100

endfunc
```

setfloatmask

D DP

setfloatmask item,value,item,value....
setfloatmask exp,exp,exp,exp

Description

The setdecmask command allows you to change the way that float values are output with the print,debug,setout,hsrout, and strout commands. You pass a parameter identifier followed by a value.

Note that you can supply any number of parameters to this command.

Valid Identifiers

display,exp (0-127)

Bits 0-6 are digit masks. These bits determine which digits get displayed. If you only want the last two digits displayed you would use 0000011. For all digits use the default of 1111111.

leaddigit,exp (1 or 0)

Display spaces. When leading digits are enabled we can display them as 0's or spaces. If this value is 1, they will be spaces.

lead,exp (1 or 0)

Display leading characters. When a number that is less than 10000 is displayed we must decide what to do with the leading digits. We can display them or discard them. If this value is 0 we will discard them (default). However if this bit is 1 we will display leading digits.

places,exp (0-3)

You can set the number of digits to the right of the decimal point. Up to 3 are allowed. If you need more use the printfloat function.

sign,exp (1 or 0)

If set to 0 will suppress the output of the sign. Note that if lead digits are enabled the the first position is reserved for the sign.

decpoint,exp (1 or 0)

If set to 0 will suppress the output of the decimal point. Note that id places is set to 0 no decimal point will be displayed.

Related Commands

print
 debug
 hserout
 serout
 strout
 setdecmask

```
'setfloatmask example
func main()

  const pi 3.142
  setfloatmask display,31,lead,1,sign,1,places,3,leaddigit,0

  print pi

endfunc
```

shiftin shiftout

shiftin dataport,clockport,mode,varb
shiftin exp,exp,exp,ivarb

shiftout dataport,clockport,mode,data
shiftout exp,exp,exp,exp

Description

The shiftin and shiftout commands allows us to send synchronous data out and in using 2 IO ports. One will be used for data and the other for clocking. Several single chip devices as well as sensors and other peripherals use this type of input or output. It is often referred to as SPI.

Up to 8 bits can be sent per command.

shiftin

Loads a variable with data that is shifted in from an IOport.

shiftout

Sends a sequence of bits out the IO port. A clock pulse is sent out for each bit via the clockport.

- **dataport** is the IO port to use for data.
- **clockport** is the IO port to use for the clock data.
- **mode** will determine the kinds of operation and how the actual shifting is done. The mode parameter can be broken down as:

Bits 0-3 Number of bits.

Bit 4 Pulse Clock Mode. Overrides bit 5, 0=No 1=Yes

Bit 5 Clock transition. 0=Clk Idle low 1=Clk Idle High

Bit 6 0=Data then clock 1=Clock then data

Bit 7 Bit Order 0=LSB first 1=MSB first

Note that bit 4 set to 1 is the most popular mode and should be used 99% of the time. When using this state the clock pin should be in the idle state before using the shiftin or shiftout command.

- **varb** is the variable used to hold the input data.
- **data** is the expression that will make up the data sent out the IO dataport. Note that only the lower 8 bits will be sent.

```
'shiftin example
func main()
  dim temp
  const dataport 0
  const clockport 1

  shiftin dataport,clockport,8,temp 'Read 8 bits into temp variable

endfunc
```

```
'shiftout example
func main()

  const dataport 0
  const clockport 1

  output dataport,clockport

loop:
  shiftout dataport,clockport,8,255 'Send 8 bit out. All 1's

goto loop

endfunc
```

sleep

D DP

sleep

Description

Places the smartchip into a low power sleep mode. In this mode the chip consumes approximately 50ua of power. You can wake the chip up with a reset or with most IRQ's

The watchdog timer can also wake-up the chip.

Related Commands

watchdogon
watchdogoff

```
'sleep example
func main()
  print "Reset"

  watchdogon

loop:
  print "Going to sleep for two seconds"
  sleep
  print "We are awake"
  goto loop

endfunc
```

sonar

D DP

sonar pingport,echoport,result
sonar exp,exp,ivarb

Description

This command was designed for ping and read sonar devices. The command toggles the pingport and waits for the echo port to go high before measuring the the length of the echo pulse.

Returns distance to object in 1/64 inch incroments.

- **pingport** - This is the port that is connected to the ping or trigger pin on the sonar device. The port state of the ping port should be in the idle state for your sonar device.
- **echoport** - This is the output port of the sonar device. The command will wait till this port goes high before starting the mesurement. Note that if does not go high the command will never return.
- **result** - This is the integer variable to store the result. The result is in 1/64" of an inch measurements.

```
'sonar example
func main()
  dim SRFcount

loop:

  sonar 0,1,SRFcount
  print "Distance= ",SRFcount /64," Inches"
  pause 20
  goto loop

endfunc
```

sqrt16

D DP

sqrt16 iexp,ivarb
sqrt16 value,varb

Description

Does a 16bit integer square root.

- **value** - The value you want the square root of.
- **varb** - The integer variable to place the square root.

```
'sqrt16 example
func main()

  dim x as integer

  sqrt16 16,x
  print x

endfunc
```

startasm

Description

The DiosPro microcontrollers give you the ability to insert KRAssembly language commands.

The KRAssembler is very sophisticated and works much like the Dios Basic language. In many cases there is no difference in the commands except the fact that they run 30-60 times faster.

To start an assembly block use the startasm command. To end the block use the endasm command. All commands between these two commands will be assembled. They will be executed when the Dios engine reaches the startasm command. This means you can jump in and out of the KRAssembler at will.

Related Commands

endasm

There are differences in syntax between the KRAssembler and Dios Basic language. The following are the exceptions:

Labels

All labels must be placed on a separate line and must start on the first column of that line. All labels must end with a :

Labels are case sensitive and can be lower, upper or mixed case.

Using the jump command you can jump to labels defined outside of the assembler. In other words you may jump to a Dios language label. You can even jump to labels located in other functions but this should be discouraged as the stack may become corrupted.

Note that if you receive the following error:
Symbol not previously defined (Label_myloopy)

This is telling you that the Dios label myloopy does not exist. Click the goto error button and the cursor will be placed on the line that attempted to make the call.

Commands

Each command must be placed on a separate line. Commands may not be placed on the first column of a line. The first column is reserved for labels.

Variables

There are two types of variables for use in your assembly routines.

Global variables that are defined in the Dios language and variables that are defined in the assembler.

```
'startasm example
'This example will toggle the IO Port 0 at about 2.5 million
'times a second.
func main()
-----
startasm
output 0 'Set IOPORT 0 as output

loop
high 0 'Turn Port on
low 0 'Turn Port off
goto loop

endasm
-----

endfunc
```

```
'startasm example
func main()

startasm
output 0
loop:
high 0
low 0
goto loop

endasm

endfunc
```



Note: While the print, debug and serout commands are similar to those in the Dios language they don't support the formatting options.

```
func main()

global DiosVarb
DiosVarb = 25

startasm
dim5 AssemblyVarb
AssemblyVarb = 75

print DiosVarb, " ", AssemblyVarb

endasm

endfunc
```

To create a variable inside the assembler use the dim statement within the startasm and endasm commands.

When using the dim command inside the KRAssembler you must provide the location where you want to store the variables. These are called banks. All banks 1-5 are available on the Dios and banks 1-14 on the DiosPro.

The syntax of the dim command is as follows.

dimX V1,V2,V3,V4..... Where X is the bank you wish to use and Vn are the individual variables you wish to define in that bank. Once defined you no longer need to concern yourself with the bank. In order to share banks you may also provide an offset as in dimX+5. The +5 tells the assembler to start at memory location 5 in that bank.

The banks each contain 256 bytes of memory and are laid out as follows:

bank0 0-255 SW Registers **Not Accessible with Dim**

The BYTE0-BYTE25 and ASMDAT0-ASMDAT13 registers are located here. All the software registers have been defined and can be accessed by inline assembly. Not Accessible with dim statement.

bank1 256-511 local variables **Use dim1**

Local variables are stored in these locations. This area is the hardest to use because they are stacked based and can change from function to function. It is not recommended that you don't use this bank unless you are not using local variables.

bank2 512-767 global variables **Use dim2**

All global variables are already Accessible so its recommended that only use this bank under special circumstances or to use the dim offset so that you do not interfere with the global memory space.

bank3 768-1023 main stack **Use dim3**

Use of this bank is not recommended.

bank4 1024-1279 UART buffer **Use dim4**

If you are using the built in UART then it is not recommended that this bank be used.

bank5 1280-1535 string space **Use dim5**

String variables are stored in this location. If you are not using strings then this area is available. If you are using this area you may use the dim offset to share the space.

Banks 6-14 **Use dim6-dim14**

Only Accessible with DiosPro.

All assembly variables are global.

Registers

Most of the Dios hardware and software registers are available and accessible just like variables.

All registers are in upper case to make them consistent with the Dios language.

```
DiosPro
func main()
'-----
startasm
dim6 blocka
dim5 x,z

x = 0
while x < 100
arrayset blocka,x,50
inc x
wend

x = 0
while x < 100
arrayget blocka,x,z
print "Bank6 Loc ",x,"=",z
inc x
wend

endasm
'-----

endfunc
```

```
func main()

startasm
TRISB = 0

loop:
PORTB=0
PORTB=255
goto loop

endasm

endfunc
```

Comments

The assembler uses the same comment character as the Dios language. Any thing following the ' will be ignored.

Accessing Ports

You set ports direction and state with the input,output,high or low commands.

To test a port use the inpX statement where X is the port number 0-14.

These commands use hard coded port numbers. This makes them extremely fast and efficient but the down side is you can use variables to supply the port numbers.

If you wish to dynamically change the port numbers you may use the 6 Dios engine port commands.

```
setdiosinput
setdiosoutput
resetdiosport
setdiosport
togglediosport
getdiosinput
```

These commands use the Dios engine routines so the port numbers may be changed at will.

Expressions

In assembly there are no true expressions like there are in the Dios language.

You **can not** use the expression $A = 4 + 5$.

In assembly you must use one of the math commands as in:

```
A= 4
add A,5
```

To perform math on a variable you use the math commands.

```
shiftright
shiftright
inc
dec
add
sub
mul
div
and
or
xor
```

```
func main()
startasm
    dim5 a
loop:
    a = inp1
    print a
    goto loop
endasm
endfunc
```



You can use the inpX command to test the state of an IOport. X is the port number 0-14.



Some commands like div and div16 actually call Dios language math routines

8-bit Vs 16-bit

While the Dios language has 16-bit integer variables, in assembly we work mostly in 8-bit chunks. All the KRAssembly variables are 8-bit variables. Even when accessing the Dios variables we do it in high and low format.

To do 16-bit operations you will use variable pairs. In addition there are some 16-bit commands that you will pass two variables.

There are also special commands for accessing passed variables from Dios engine. These are ment to be used within the startasmcommand function shown later.

```
func main()
'-----
startasm
dim5 VL,VH,X
X = 0
VL=0
VH=0

while X <10
  add16 VL,VH,5
  inc X
wend

endasm
'-----

endfunc
```


and **And a value to a register or variable**

syntax `and varb,value`
`add V,vcrn`

Operands: **varb**: Variable or register to and to.
value: Value to and with Variable. 8-bit.

Description: This commands ands a value to the variable or register.

arrayget **Read from a block of variables**

syntax `arrayget Vstart,Index,ResultVarb`
`arrayget V,vcrn,V`

Operands: **Vstart**: Variable that marks the starting point into bank.
Index: Index to byte you wish to retrieve.
ResultVarb: The variable or register to place the retrieved value into.

Description: This will allow you to pull a byte from a set of variables using and index variable.

arrayset **Write to a block of variables**

syntax `arrayset Vstart,Index,Value`
`arrayset V,vcrn,vcrn`

Operands: **Vstart**: Variable that marks the starting point into bank.
Index: Index to byte you wish to write to.
Value: The byte value that you wish to write.

Description: This will allow you to write a byte to a set of variables using and index variable.

KRAssembler Commands

bitset **Set a bit**

syntax `bitset Varb,Bit`
 bitset V,n

Operands: **Varb:** Variable that holds the bit you wish to set.

Bit: Bit to set

Description: Sets a bit in a variable or register.

bitreset **Reset a bit**

syntax `bitreset Varb,Bit`
 bitreset V,n

Operands: **Varb:** Variable that holds the bit you wish to reset.

Bit: Bit to reset

Description: Resets a bit in a variable or register.

branch **Jump to a location based on a value**

syntax `branch Varb,location0,location1,....`
 branch V,label

Operands: **Varb:** Variable that holds the value that will be used as a index to the location to jump to. IE 0 will jump to location0 and 1 will jump to location1 and so on.

Description: **lable:** The labels to jump to.

checkbuff **Check the UART Buffer**

syntax *checkbuff Vstat,Vdat*
checkbuff V,V

Operands: **Vstat:** Bit 0 will be set if data is available.

Vdat: If data available will contain the next byte

Description: Checks the UART buffer. If data is in the buffer the Vstat variable's bit 0 will be set and the Vdat variable will contain the byte.

Important

In order for this command to work you must have one or more DiosPro language UART commands in your code. IE hserin, hserout, hsersetup.

If this is not possible you can place a **USE_hcheckbuff** at the start of one of your assembly routines. This tells the compiler to include the appropriate routines needed.

KRAssembler Commands

debug **Send data to debug terminal**

syntax *debug data,.....*
 debug vcrnt,.....

Operands: **data:** Data element to send to the debug port. You may also send quoted data and each character will be sent.

Description: Sends one or more data elements to the debug port.

dec **Decrement a Variable**

syntax *dec varb*
 dec V

Operands: **varb:** Variable to decrement.

Description: This commands subtracts 1 from the variable.

dec16 **16-bit Decrement**

syntax *dec16 varbL,varbH*
 dev16 V,V

Operands: **varbL:** Variable that holds low byte that you want to decrement.

varbH: Variable that holds high byte that you want to decrement.

Description: This command does a 16-bit decrement on a High/Low variable or register pair.

dim Create a KRAssembly Variable

syntax `dimX Varb,Varb,.....`
`dimn V,V....`

Operands: **X**: Bank index
varb: Variable to create.

Description: The dim statement allows you to create a KRAssembly variable. You must supply a bank number to indicate where you want the variable to reside. See text for bank definitions.

You may also supply an offset in the form of `dimX+offset`. The offset option is provided so that you can share some of the variable space with the Dios engine. For instance if you are using strings you may borrow some memory from the top of that bank.

Example `dim5+200 myvarb1,myvarb2`

This starts the variables out at the 200th position in that bank.

div Divide a Register or Variable

syntax `div varb,value`
`div V,vcrn`

Operands: **varb**: Variable or register to be divided.
value: Amount to divide. 8-bit.

Description: This command divides a Register or Variable by a value.

div16 16-Bit Divide

syntax `div16 varbL,varbH,valueL [,valueH]`
`div16 V,V,vcrn [,vcrn]`

Operands: **varbL**: Variable that holds low byte that you want to divide.
varbH: Variable that holds high byte that you want to divide.
valueL: Low byte to divide low/high variables by.
valueH: High byte to divide low/high variables by. Optional

Description: This command does a 16-bit divide of a high/low register or variable pair. Note that you can pass a single 16-bit number and it will divide it into high/low pair for you. As in : **div16 VL,VH,1000**

KRAssembler Commands

eeread	Read from EEprom
syntax	<code>eeread AddrL,AddrH,Varb</code> <code>eeread vcrn,vcrn,V</code>
Operands:	AddrL: Low byte of the eeprom address. 0-255 AddrH: High byte of eeprom address. (Only valid with DiosPro, Must pass 0 on Dios)
Description:	Varb: The variable to place the read byte into. This command reads a byte from the on-board EEprom.

eewrite	Write to EEprom
syntax	<code>eewrite AddrL,AddrH,Value</code> <code>eewrite vcrn,vcrn,V</code>
Operands:	AddrL: Low byte of the eeprom address. 0-255 AddrH: High byte of eeprom address. (Only valid with DiosPro, Must pass 0 on Dios)
Description:	Value: The value to write to the EEprom. This command writes a byte to the on-board EEprom.

exit	Exit from KRAssembly
syntax	<code>exit</code> <code>exit</code>
Description:	When your program reaches the endasm command it will exit and return to the Dios language. This command lets you exit to the endasm command from anywhere in your assembly block.

hserout	Send Serial Data out the UART
syntax	<code>hserout port,data,.....</code> <code>hserout vcrn,vcrnt,.....</code>
Operands:	data: Data element to send to the UART. You may also send quoted data and each character will be sent.
Description:	Sends one or more data elements to the UART. The hsersetup command needs to be issued in Dios.

goto **Jump to a KRAssembly Routine**

syntax goto Label
 goto Label

Description: Jumps to another assembly location.

high **Set Port High**

syntax high Port
 high n

Operands: **Port:** The port to set high. Note the port must be in output mode. You may only use a number or constant here.

Description: Sets a port high.

if / then / else / endif **Test a Condition**

syntax if condition1 operator condition2 then
 if V operator vcrn then

Operands: **Condition1:** The item to test. This can be any variable or register. Note that you may test bits of a variable or constant by specifying the bit with the syntax: register,bit

Condition2: The item to test against. This can be any variable or register number or constant. You can not use a bit here.

operator: This is the kind of comparison. You can use the following:

<, >, <=, >=, <>, !=, =, ==

Description: This command allows you to test various states and conditions then process code based on the results. You may use the else statement to process alternate code if the condition fails.

Example:

```
if PORTB,0 = 1 then
  print "its high"
else
  print "its low"
endif
```

KRAssembler Commands

inc Increment a Variable

syntax `inc varb`
`inc V`

Operands: **varb**: Variable to increment.

Description: This commands adds 1 to the variable.

inc16 16-bit Increment

syntax `inc16 varbL,varbH`
`inc16 V,V`

Operands: **varbL**: Variable that holds low byte that you want to increment.

varbH: Variable that holds high byte that you want to increment.

Description: This command does a 16-bit increment on a High/Low variable or register pair.

input Set Port to Input Mode

syntax `input Port`
`input n`

Operands: **Port**: The port to set to input. You may only use a number or constant here.

Description: Places the port into input mode.

inp Test a Port

syntax `inpX`
`inpX`

Operands: **X**: The port to set to test. It must be a number.

Description: This command is intended to be used with the print and if/then as well as the while/wend command.

load16 Loads a 16-bit value into a 16-bit register pair

syntax `load16 varbL,varbH,value`
`load16 V,V,n`

Operands: **varbL**: Variable that will hold the lower 8-bits of the value.

Description: **varbH**: Variable that will hold the upper 8-bits of the value.

This command is lets you load a 16-bit numerical value into two 8-bit registers.

low **Set Port low**

syntax low Port
low n

Operands: **Port:** The port to set low. Note the port must be in output mode. You may only use a number or constant here.

Description: Sets a port low.

mul **Multiply a Register or Variable**

syntax mul varb,value
mul V,vcrn

Operands: **varb:** Variable or register to be multiplied.
value: Amount to multiply. 8-bit.

Description: This commands multiplies a register or variable by a value.

mul16 **16-Bit Divide**

syntax mul16 varbL,varbH,valueL [,valueH]
mul16 V,V,vcrn [,vcrn]

Operands: **varbL:** Variable that hold low byte that you want to multiply.
varbH: Variable that hold high byte that you want to multiply.
valueL: Low byte to multiply low/high variables by.
valueH: High byte to multiply low/high variables by. Optional

Description: This command does a 16-bit multiply of a high/low register or variable pair. Note that you can pass as single 16-bit number and it will break it down into the high/low pair for you. As in : **mul16 VL,VH,1000**

or **Or a value to a register or variable**

syntax or varb,value
or V,vcrn

Operands: **varb:** Variable or register to or to.
value: Value to or with Variable. 8-bit.

Description: This commands or's a value to the variable or register.

KRAssembler Commands

output	Set Port to Output Mode
syntax	output Port <i>output n</i>
Operands:	Port: The port to set to output. You may only use a number or constant here.
Description:	Places the port into output mode.
pause	Pause in 1ms units
syntax	pause ValueL [,ValueH] <i>pause vcrn [,vcrn]</i>
Operands:	ValueL: The amount to pause. 0-255 ms.
Description:	ValueH: The amount to pause. Optional. By using this parameter you can pause from 0-65535 ms Cause a delay from 0-65535 ms. You may pass a single 16-bit number and it will be broken down into the high and low bytes for you.
pauseus	Pause in 1us units
syntax	pauseus ValueL [,ValueH] <i>pauseus vcrn [,vcrn]</i>
Operands:	ValueL: The amount to pause. 0-255 us.
Description:	ValueH: The amount to pause. Optional. By using this parameter you can pause from 0-65535 us Cause a delay from 0-65535 ms. You may pass a single 16-bit number and it will be broken down into the high and low bytes for you.
print	Send data to debug terminal
syntax	print data,..... <i>print vcrnt,.....</i>
Operands:	data: Data element to send to the debug port. You may also send quoted data and each character will be sent.
Description:	Sends one or more data elements to the debug port. The print command does a little formatting for you. It will automatically convert any single value to decimal. It will also add a CR/LF to the end unless you place a ; on the end of the command string.

return **Return from KRASsembler Subroutine**

syntax *return*
return

Description: Use to return from a routine that you called with a gosub command.

serout **Send Serial Data to any IOport**

syntax *serout port,data,.....*
serout vcrn,vcrnt,.....

Operands: **port:** The port to send the data to. This port must be in output mode.

Description: **data:** Data element to send to an IO port. You may also send quoted data and each character will be sent.

Sends one or more data elements to a IO port. The sersetup command needs to be issued in Dios.

shiftright **Shift Bits Right**

syntax *shiftright varb*
shiftright V

Operands: **varb:** Variable or register to shift.

Description: This commands shifts the bits in a variable or register left.

shiftright **Shift Bits Right**

syntax *shiftright varb*
shiftright V

Operands: **varb:** Variable or register to shift.

Description: This commands shifts the bits in a variable or register right.

KRAssembler Commands

sub Subtract from a register or variable

syntax `sub varb,value`
`sub V,vcrn`

Operands: **varb**: Variable or integer to subtract from.

value: Value to subtract from Variable. 8-bit.

Description: This commands subtracts a value from a variable or register.

sub16 Does a 16-bit subtract from a register or variable

syntax `sub16 varbL,varbH,valueL [,valueH]`
`sub16 V,V,vcrn [,vcrn]`

Operands: **varbL**: Variable that hold low byte that you want to subtract from.

varbH: Variable that hold high byte that you want to subtract from.

valueL: Low byte to subtract from the low/high variables.

valueH: High byte to subtract from the low/high variables. Optional

Description: This command does a 16-bit subtract from a high/low register or variable pair. Note that you can pass as single 16-bit number and it will break it down into the high/low pair for you. As in : **sub16 VL,VH,1000**

toggle Toggle a IO Port

syntax `toggle Port`
`toggle n`

Operands: **Port**: The port to toggle. Note the port must be in output mode. You may only use a number or constant here.

Description: If the port is high this command will set it low. If low it will be set high.

waitport Waitfor a condition on port

syntax `waitport Port,State,Timeout,Label`
`waitport vcrn,vcrn,vcrn,label`

Operands: **Port**: The port to test.

State: The state to waitfor.

Description: **Timeout**: Adds a bit of time out before label call.

Label: The location to jump to if condition is not met.

The command will fall through if the state is met.

while / wend

Loop on a Condition

syntax

while condition1 operator condition2
while V operator vcrn

Operands:

Condition1: The item to test. This can be any variable or register. Note that you may test bits of a variable or constant by specifying the bit with the syntax: register,bit

Condition2: The item to test against. This can be any variable or register number or constant. You can not use a bit here.

operator: This is the kind of comparison. You can use the following:

<, >, <=, >=, <>, !=, =, ==

Description:

This command allows you to test various states and conditions. As long as the condition is true the commands between the while and wend will be executed

Example:

```
dim5 x
x = 0
while x < 10
  print x
  inc x
wend
```

xor

Xor a value to a Register or Variable

syntax

xor varb,value
xor V,vcrn

Operands:

varb: Variable or register to xor to.

value: Value to xor with Variable. 8-bit.

Description:

This commands xor's a value to the variable or register.

Blank Page

startirqasm



startirqasm irqname
startirqasm irqname

Description

The startirqasm command creates an assembly handler for a given hardware IRQ. You use the KRAssembler to handle the IRQ.

Do Not use the exit command with startirqasm.

- **irqname** is the name of the IRQ that will trigger this function.

Valid IRQ names:

INT0
 INT1
 INT2
 TMR0
 TMR1
 TMR2
 TMR3
 CCP1
 CCP2
 AD
 RB
 LVD
 SSP
 PSP

See the startasm command for a list of KRAssembly commands.

Related Commands

endirqasm

There are restrictions as to what KRAssembly commands that are allowed in an IRQ routines.

Dont use any of the following commands.

pause	debug
pauseus	print
mul	waitport
mul16	eeread
div	eewrite
div16	branch
arrayget	
arrayset	
checkbuff	
serout	
hserout	

```
'startirqasm example
'Note that INT0 is on IO port 7
func main()
  global myvarb
  clear

  'Start INT0 IRQ
  irqglobalstart 'enable global IRQ's
  irqINT0start() 'enable INT0 IRQ

loop:
  print myvarb
  pause 100
  goto loop
endfunc

'Must be placed outside functions
'-----
'IRQ handler for INT0 irq
'-----
startirqasm INT0

  inc16 myvarb,myvarb+1

endirqasm
'-----
```

strout

D DP

```
strout stringaddress,value1,value2 ....  
strout exp,exp,exp ....
```

Description

Sends formatted print data to a string. This command works identical to the debug, serout, hserout commands. You pass the address of a string and the output will be placed in that string.

- **stringaddress** - This is the address of the variable to populate. Make sure you define a string large enough to hold the data you wish to generate.

Note that you must use the # or @ character as a post operator when passing a string address to the output processor.

Example:

```
#myvarbaddress or @myvarbaddress
```

- **value1-value_n** - This is the data to be sent to the hardware serial port. Separate values with a comma. You can send tables,strings, any numerical expression. Use the dec modifier to convert numeric data to ASCII characters.

The format operators may also be used to change the numeric format. See the print command.

See the print command for a complete listing of modifiers.

Related Commands

```
print  
serout  
hserout  
debug  
setdecmask  
setfloatmask
```

```
'strout example  
func main()  
  clear  
  
  global dispstr(25) as string  
  dim y as integer  
  y = 2154  
  
  strout @dispstr,"Hello world ",dec y," times."  
  
  print dispstr  
  
endfunc
```

swapbyte

D DP

```
swapbyte varb  
swapbyte ivarb
```

Description

Swaps the high and low bytes of a integer variable.

Related Commands

```
swapnib
```

```
'swapbyte example  
func main()  
  dim a  
  a=256  
  swapbyte a  
  print a  
  
endfunc
```

swapnib

D DP

swapnib varb
swapnib ivarb

Description

Swaps the high and low nibbles of the low byte in a integer variable.

Related Commands

swapbyte

```
'swapnib example
func main()
  dim a
  a=31
  swapnib a
  print a
```

```
endfunc
```

table

D DP

table label: "String data"

Description

The table command allows you to place string data into program memory. The tables are referenced with a label. You can pass these tables to the print, serout, hserout, and debug commands.

You can also pass them to functions. You can assign the table to a string as shown below:

```
global mystrvarb(10) as string
table name: "mike"
```

```
mystrvarb=name
```

A table is not a variable. You can not store new data into a table once its been defined.

By placing a + at the end of the line will tell the table command not to terminate the existing string. And will treat the next table command as though it were part of the first.

This is useful to string large strings and menus together.

e.g.

```
table menu: "01:item 1,02:item 2,"+
table "03:item 3,04:item 4"
```

- **label** This is used to identify the table.
- **string data** This is literal text stored in program memory.

```
'table example
func main()

  table name: "My name is mike"
  table age: "My age is 21"
```

```
  print name
  print age
```

```
endfunc
```

Description

This is a group of commands that will control the Dios's 4 built-in timers.

timer0 8/16-bit timer/counter

timer1 8/16-bit timer/counter

timer2 8-bit timer

timer3 8/16-bit timer/counter

- **timerXon** - Enables the given timer where X is the timer number 0-3.
- **timerXoff** - Disables the given timer where X is the timer number 0-3.
- **timerXsetall** - This command allows you to set the raw parameter values for each timer. X is the timer number 0-3.
- **timerXreadvalue ivarb** - This command allows you read the current timer count into a variable. The actual syntax is **timerXreadvalue ivarb** where X is the timer number and ivarb is the variable to place the result into.
- **timerXsetvalue exp** - This command allows to set the timers internal counter. All counters can be ether 16-bit mode or 8-bit mode except timer 2 which is always 8-bit. The actual syntax is **timerXsetvalue exp** where X is the timer number and exp is the value to write.
- **timerXsourceclock** - This command places the timer in timer mode. Its internal counter will be incremented by the system clock. It only applies to timer0, timer1, and timer3.
- **timerXsourceP15** - This command places the timer in counter mode. Its internal counter will be incremented by the rising edge of IO port 15. It only applies to timer0, and timer3.
- **timer0sourceP17** - This command places timer0 in counter mode. Its internal counter will be incremented by the state change of IO port 17.
- **timer0edgelow** - When set as a counter the command will set the counting to take place on the rising edge for timer0.
- **timer0edgehigh** - When set as a counter the command will set the counting to take place on the falling edge for timer0.
- **timerXmode8bit** - This command places the timer in 8-bit mode. Its internal counter will wrap at 256. It only applies to timer0, timer1, and timer3.
- **timerXmode16bit** - This command places the timer in 16-bit mode. Its internal counter will wrap at 65536. It only applies to timer0, timer1, and timer3.
- **timer1oscon** - Timer1 has the ability to run off of a low speed oscillator connected to IO ports 14 and 15. This command enables that oscillator.

- **timer1oscoff** - Timer1 has the ability to run off of a low speed oscillator connected to IO ports 14 and 15. This command disables that oscillator.
- **timerXsyncon** - This will cause timer1 or timer3 to synchronize the external clock input.
- **timerXsyncoff** - This will cause timer1 or timer3 disable external synchronization.
- **timer0prescale exp** - Sets timer0's prescale to one of the following values: 0=1:1, 1=1:2, 2=1:4, 3=1:8, 4=1:16, 5=1:32, 6=1:64, 7=1:128, 8=1:256
- **timer1prescale exp** - Sets timer1's prescale to one of the following values: 0=1:1, 1=1:2, 2=1:4, 3=1:8
- **timer2prescale exp** - Sets timer2's prescale to one of the following values: 0=1:1, 1=1:4, 2=1:16
- **timer3prescale exp** - Sets timer3's prescale to one of the following values: 0=1:1, 1=1:2, 2=1:4, 3=1:8

What is prescale? When a counter to timer increments and the prescale is set to 1:1 it will increment with each tick. When set to 1:2 it will take two ticks to increment the counter. This allows you to change the resolution or effect the timer speed.

- **timer2postscale exp** - Sets timer2's postscale to one of the following values: 0=1:1, 1=1:2, 2=1:3, 3=1:4, 4=1:5, 5=1:6, 6=1:7, 7=1:8, 8=1:9, 9=1:10, 10=1:11, 11=1:12, 12=1:13, 13=1:14, 14=1:15, 15=1:16
- **timerXsetall exp** - This command allows you to set the raw value of the timer register.

toggle

D DP

toggle port,port,port....
toggle exp,exp,.....

Description

Changes the state of the IO port. If the port was high then it goes low. If it was low it goes high.

- **port** is the port number to change state.

Note that you can update as many ports as you like using the “,” to separate the port numbers. The toggle command will only work if the port is set up for output.

Related Commands

high
low
input
output

```
'toggle example
func main()

loop:
  toggle 1
  goto loop

endfunc
```

trunc

D DP

trunc

Description

The floating point math routines may need to be told to round or truncate when certain calculations are made. The trunc command sets the internal floating point routines to truncate with no rounding.

Related Commands

round
trunc
int

```
'trunc example
func main()
  dim xyz as float

  trunc
  xyz = 100.6

  print {6.2} xyz

  int xyz
  print {6.2} xyz

endfunc
```

waitport

D DP

waitport port,state,timeout,timeoutlabel
waitpost exp,exp,exp,label

Description

Waits for a particular state of an IO port. If the timeout occurs the program will jump to the timeout label.

If the state is reached within the timeout period the command falls through to next command.

- **port** The IO port number to use.
- **state** The state we are looking for 0 or 1.
- **timeout** is the number of passes to make looking for the port to change state.
- **timeoutlabel** The location to jump to if a timeout occurs.

```
'waitport example
func main()

again:
waitport 4,1,5000,timeout
print "Port went low"

goto again

timeout:
print "Time out"
goto again

endfunc
```

while wend

D DP

```
while expression1 operators expression2
```

```
.....  
wend
```

```
while exp operators exp
```

```
.....  
wend
```

Description

The while command will execute a group of commands as long as the expression is true. It will do this over and over until the expression fails.

expression1 and **expression2** are the expressions to compare via the operators. They may contain any number of variables, constants, numbers, iopors or registers. The expressions may contain both integer and floating point numbers and variables.

Operators are used to indicate the comparison of expression 1 to expression 2.

Valid operators are:

```
=equal  
==equal  
<>not equal  
><not equal  
!=not equal  
>Greater than  
<Less than  
>=Greater than equal to  
<=Less than equal to
```

Note that if multiple comparisons are used they must be separated with and/or operators. For speed and code efficiency the and/or operators are handled in the order that they are encountered. If you want to combine and operators with or operators the and operators should be coded first.

```
'while wend example  
func main()  
  dim x  
  x = 0  
  
  while x < 10  
    print x  
    x = x + 1  
  wend  
endfunc
```

watchdog



watchdogon

Description

Turns on the watchdog timer. The Dios has a background watchdog timer. It does a couple of things.

1. It will reset the Dios every 2 seconds. What good is this? In this case you can use the watchdog to check reset the dios when you are doing an operation that may lock up the dios.

- Step 1 - Turn on the watchdog.
- Step 2 - Do your operation.
- Step 3 - Turn off watchdog.

If your operation lasts longer then 2 seconds the Dios will reset.

2. The most common way to use the watchdog timer is to wake the Dios up after it has been put to sleep.

- Step 1 - Turn on the watchdog.
- Step 2 - Put the Dios to sleep.
- Step 3 - The Dios will wake up after 2 seconds.

If what you are doing after the Dios wakes up takes longer than a few seconds you could turn off the watchdog. Once your wake up operations are complete go back to Step 1

Related Commands

watchdogoff

```
'watchdog example 1
func main()

  print "reset"

  watchdogon

loop:
  print "In loop"
  pause 500
  goto loop

endfunc
```

```
'watchdog example 2
func main()
  print "Reset"

loop:
  watchdogon
  print "Going to sleep for two seconds"
  sleep
  watchdogoff
  print "We are awake"
  goto loop

endfunc
```

watchdogoff

watchdogoff

Description

Turns off the watchdog timer. The Dios has a background watchdog timer. It does a couple of things.

1. It will reset the Dios every 2 seconds. What good is this? In this case you can use the watchdog to check reset the dios when you are doing an operation that may lock up the dios.

Step 1 - Turn on the watchdog.

Step 2 - Do your operation.

Step 3 - Turn off watchdog.

If your operation lasts longer then 2 seconds the Dios will reset.

2. The most common way to use the watchdog timer is to wake the Dios up after it has been put to sleep.

Step 1 - Turn on the watchdog.

Step 2 - Put the Dios to sleep.

Step 3 - The Dios will wake up after 2 seconds.

If what you are doing after the Dios wakes up takes longer than a few seconds you sould turn off the watchdog. Once your wake up operations are complete go back to Step 1

```
'watchdog example 1
func main()

  print "reset"

  watchdogon

loop:
  print "In loop"
  pause 500
  goto loop

endfunc
```

```
'watchdog example 2
func main()
  print "Reset"

loop:
  watchdogon
  print "Going to sleep for two seconds"
  sleep
  watchdogoff
  print "We are awake"
  goto loop

endfunc
```