

***DiosPro Language
Tutorial
DiosPro Series
Volume 4***

Dios Language Tutorial

Version 2.4

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

This manual will provide a basic understanding of the Dios Language to those beginners interested in programming the Dios family of microcontrollers.

It is important that you read this document in the order that it is presented the first time through. The information is presented in a way to work you up to more complex explanations and examples.

Table of Contents

1: Getting Started

Introduction5
Function Definition6
Print6
Comments6
Calling a Function7
Program Flow8
Labels8
Spaghetti Code9
Gosub9
Return9
IO Ports10
Output Mode10
ioport11
Input Mode11
Integer Variables12
Dim statement12
Variable Scope12
Global Variables13
Floating Point Variables14
Printing Floating Point Numbers14
Floating Point Restrictions15
Integer Math16
Floating Point Math18
Floating Point Rounding18
Signed Numbers18
for / next19
if / then / else21
Using multiple and/or statements22
while / wend23

2: The Next Step

Introduction25
Passing Values to Functions26
Returning a value from a function.27
A function may also return a value.27
Parameter Conversion28
Arrays29
Array Restrictions30
Byte Arrays31
Accessing Variable Bits31
Accessing Registers32
Strings32
String Direct Access33
String Insertion34
Partial String Assignment35
String Usage36
Tables37
Placing Other Data in Tables38

3: IRQ Basics

Introduction39
Types of IRQ's40
How Do Software IRQ's Work?40
Dios Software IRQ Block Diagram41
INT0 IRQ Example42
TMR0 IRQ Example43
Software IRQ things to keep in mind44
Issues with IRQ's in general44

Introduction

This chapter will provide the basic knowledge needed to start programming the Dios. It is assumed you know how to start the Dios PCsoftware and hookup the Dios.

We will cover creating and calling functions, variable declaration and program loops.

The examples provided are small enough that it recommended that you hand code each one to help you grasp the concepts discussed.

1: Getting Started

Function Definition

The Dios language is made up of a series of instructions stored in program areas. These program areas are self-contained and execute their instructions from beginning to end. These program areas are called functions, each identified by a unique name.

Example 1.1 shows the simplest function in the world. We use the **func** command to start a function and the **endfunc** command to end it. In this case we have created a function called main.

A list of arguments can be inserted following the name of the function. This list must be enclosed in parentheses. In the example we don't have any arguments.

Example 1.1

```
func main()
```

```
endfunc
```



The function name is always followed by parentheses.

Later we will pass arguments to the function inside these parentheses.



You can also use the commands **function**, **endfunction**, **sub** and **endsub**. They are all the same, just a different way of starting and ending a function.

Print

In order to get our function to do a little more let's look at the **print** command.

The **print** command will allow us to send information to the debug terminal. In order to send text to the debug terminal we must enclose the text in quotes.

Example 1.2

```
func main()
```

```
print "Hello World"
```

```
endfunc
```



This example prints "Hello World" in the debug terminal window.



When the Dios software detects a **print** command, the debug terminal will automatically pop up.

Comments

There are many times you will want to place comments and remarks in your code to make the code easier to understand. Comments are started by a single quote. Everything after the single quote is ignored by the Dios Compiler.

Example 1.3

```
func main()
```

```
'This is a comment
```

```
endfunc
```



You can also place comments at the end of an instruction.



The **print** command is the most powerful command on the Dios. It can print any variable, table or literal. You can change the format of numeric output on the fly. The **print** command has many other options. For more information on the **print** command check out the Dios Ultra Manual Volume 1.

Calling a Function

In Example 1.2 we created a function that had a single function with a single **print** command. Let's expand on that a bit.

In Example 1.4 we added a second function. However when the program is run it only prints "Hello World". It never gets to main2. Why?

Example 1.4

```
'Root Function
func main()
  print "Hello World"
endfunc
```



Hey! This example does the same as the last one. Whats up?

```
'2nd Function
func main2()
  print "Hello Again"
endfunc
```



When the end of the root function is reached the Dios will no longer execute any of your commands. However certain actions such as advanced interrupts are still executed

When the Dios is started the first defined function is the function that is executed. This first function becomes the root function. When the end of the root function is reached the Dios stops execution.

In order to run another function we must tell the program to run that function specifically. We do this by calling the function. We call a function by using its name.

In Example 1.5, when the **main2()** function is encountered in the root function the Dios jumps to that function and starts executing. When the end of main2() is reached the Dios exits main2() and continues where it left off in the root program.

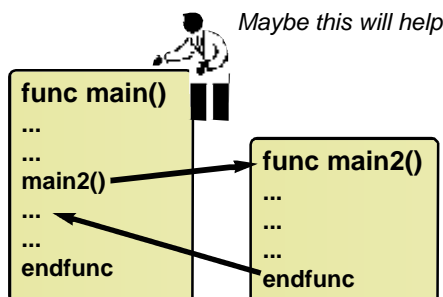
Example 1.5

```
'Root Function
func main()
  print "Hello World"
  main2()
  print "Im Back"
endfunc
```



You jump to another program area by using its name. When it is finished you continue where you left off.

```
'2nd Function
func main2()
  print "Next Function"
endfunc
```



1: Getting Started

Program Flow

Normally the program flows from top to bottom executing commands as they are encountered. We call this program flow. One way to change the program flow is by calling a function as we saw in Example 1.5. Another way to change the program flow is using the **goto** command. In order to use the **goto** command you must be able to tell the program where to go. We do this with a **label**.

Labels

A **label** is a place holder for a location in your program code. **Labels** are only valid within a function. To declare a **label** just use a name followed by a colon. The name is alphanumeric and must begin with a letter. The names are case sensitive, so the **label** "loop:" is different than the **label** "Loop:".

Example 1.6 shows simple loop to demonstrate both the **goto** and **label** commands.

Example 1.6

```
func main()
loop:
  print "Hello World"
  goto loop
endfunc
```



This is the program that never ends. The endfunc command is never reached.

Remember labels are just place holders. They represent a location in your code.

The program will print the words **Hello World** over and over again to the debug window. It will do this forever.

You cannot goto a label in another function. In other words, you can't jump from one function to another.

Example 1.7 shows an illegal **goto** and will cause an error when compiled.

Example 1.7

```
func main()
  goto loop
endfunc

Func testfunc()
loop:
  print "Where am I"
endfunc
```



All names created must adhere to certain rules. This applies to function and label names

- They must begin with a letter.
- They may not contain spaces.
- They may contain only letters and numbers.
- They are case sensitive.
- They may not be one of the Dios reserved words.



The Dios allows 500 labels per program.



Important

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

Spaghetti Code

Lets look at a common programming error. Example 1.8 shows what we call spaghetti code. It jumps all over the place and is hard to follow. Unfortunately the use of extensive goto commands can lead to some pretty bad spaghetti code.

Example 1.8

```
func main()

    goto step1

Step2:
    print "In step 2"
    goto done

Step1:
    print "In step 1"
    goto step2

Done:
endfunc
```

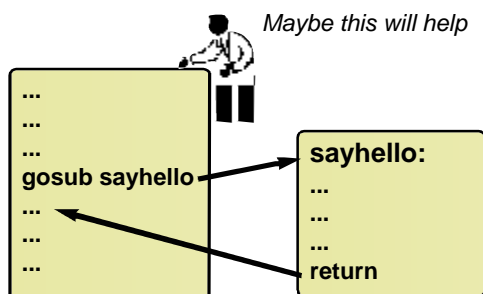


Gosub

One way to eliminate spaghetti code is to create small subroutines within your functions. You can do this with the **gosub** command. The gosub command works identical to the **goto** command with one exception. When a **gosub** command is executed it saves its location so that the program may return where it left off.

Return

The return command is used to tell the program to return back to where the **gosub** command was issued. It's kind of like calling a function but with the **gosub** and **return** you stay inside the current function.



Example 1.9

```
func main()
loop:
    gosub sayhello
    goto loop

sayhello:
    print "Hello"
    return

endfunc
```

When the gosub is reached we jump to the sayhello label.

When we reach the return we go back to the point just after the gosub command.

1: Getting Started

IO Ports

IO ports are one of the most important aspects of the Dios or any other microcontroller for that matter. The IO ports are the main reasons we want to use a microcontroller in the first place. The IO ports are how we control the outside world.

The Dios IO ports can be set up for input or output. We sometimes refer to this as the direction. To change the direction of the IO ports use the **input** and **output** commands.

Output Mode

In Example 1.10 we set IO port to output mode. Once a port is in output mode you can change its state by using the **high**, **low** and **toggle** commands.

Example 1.10

```
func main()
    output 5
endfunc
```

When a port is in the high state it will measure pretty close to 5v (or Vdd). When in the low state it will measure pretty close to 0v (or Vss).

Take a look at Example 1.11. In this example we change port 5 from high to low very fast. It will generate a 43.5Khz signal on port 5.

Example 1.11

```
func main()
    output 5

loop:
    high 5
    low 5
    goto loop
endfunc
```



You can only change a port's state with the high and low commands when it is on output mode.



At powerup, the Dios IO ports default to input mode.



Dios IO ports really have three states.

- High
- Low
- High impedance

When an IOport is in input mode it asserts no level on the port. This feature will allow us to do some advanced operations such as I2c and 1Wire communications without external hardware

ioport

There are other ways to access the IO ports. One of the most common ways is the **ioport** command. In Example 1.12 we have replaced the **high** and **low** commands with **ioport(x)=**. When the ioport is set to 1 (or anything other than 0) the port goes high; when it is set to 0 the port goes low.



Using the **ioport** command requires a bit more processing than using the **high** and **low** commands. This makes them a little slower.



A great many of the built-in commands access and change the IO ports. You can also access the ports via inline assembly or directly with hardware registers. These methods will be covered in other manuals.

Example 1.12

```
func main()
```

```
output 5
```

We set the port to output mode with the **output** command.

```
loop:
```

```
ioport(5) = 1
```

```
ioport(5) = 0
```

```
goto loop
```

```
endfunc
```

Input Mode

Now let's take a look at accessing input ports. In Example 1.13 we set the port to input and again access it via the **ioport** command. Here the **print** command will print 0 or 1 depending on the state of the ioport.

Example 1.13

```
func main()
```

```
input 5
```

We set the port to input mode with the **input** command.

```
loop:
```

```
print ioport(5)
```

```
goto loop
```

```
endfunc
```

The Dios IO ports will register a low if the voltage level is below 1.2 volts. The IO port will register high if over 1.35 volts. Between these voltage values is a grey area that could make the port swing either way.

1: Getting Started

Integer Variables

If all you could do was jump from place to place in the code you would not get much done. We want to be able to manipulate some data. Data space is different than program space and is normally accessed by using variables. The Dios has several types of variables but we will concentrate on the simplest, the integer variable.

Dim statement

To declare (create) an integer variable, use the **dim** statement as shown in Example 1.14. By default any variable created this way is an integer.

Example 1.14

```
func main()  
  
  dim age  
  
endfunc
```



You can specifically tell the compiler to define an integer by using the statement:
dim age as integer

To put a variable to work we need to store information in that variable and retrieve information from it. One way to store information in a variable is to use the assignment operator. The value on the right side is stored in the variable on the left side. In Example 1.15 the variable age will contain the value of 25.

Example 1.15

```
func main()  
  
  dim age  
  age = 25  
  print age  
  
endfunc
```



The = operator can be used to assign a value to a variable.



The print statement is very versatile. It can print all types of variables directly.

Variable Scope

Variables that are defined in a function are only valid until the function exits. This is called scope. Once the function that created the variable exits the data space is freed up. This makes for some very efficient use of variable space.

In Example 1.16 each age variable is valid only within the function it was created (dim command). Had we not used the dim command in testfunc we would have received an error. This is because the age variable in the main function does not exist in the testfunc function.



When a variable is defined in a function it is important to know that any value assigned will not remain intact when the function exits.

The only function where the variables will remain intact is the root function, since it will never exit unless the program ends

Example 1.16

```
func main()
  dim age
  age = 25
  testfunc()
  print "main age ",age
endfunc
```

```
func testfunc()
  dim age
  age = 32
  print "testfunc age ",age
endfunc
```



These two age variables are different and have nothing to do with one another

Global Variables

The variables that are defined inside functions are considered local. You can also define variables that can be shared with all functions. These variables are called global variables. All functions can access these variables. You use the **global** statement to declare (create) global variables.

Local variables can only be created inside functions and global variables can be created anywhere in the program.



Dios integer variables can store 0-65535. They cannot contain signed numbers or decimal points.

Example 1.17

```
func main()
  global age
  age = 25
  print age
  testfunc()
  print age
endfunc
```

```
func testfunc()
  age = 44
endfunc
```



Once a variable is created with the global statement we can access it from all functions.

Global variables never go away.

1: Getting Started

Floating Point Variables

You learned about integer variables. Now its time to get your hands a little dirty and learn about the floating point variables. In many cases you will use them just as you would the integer variables.

The Dios uses 32-bit IEEE 754 floating point. What does floating point mean? In vary basic terms it means we have a number and the decimal point can be calculated to reside at various places in that whole number. If this sounds a bit confusing don't worry. Unless you want to manipulate the individual bits of the 4 bytes that make up the number, you won't have to concern yourself with the behind the scene details.

You will have to keep a few things in mind. Like Visual Basic or other high level languages 32-bit floating point is considered single precision. You will get 6 to 7 significant digits of accuracy.

The more digits on the left side of the decimal place the less digits on the right side of the decimal point. Also when using decimal points you will notice rounding errors as the digits get close to the outer range. Note that these rounding errors are display only and don't affect the real value of the data being displayed.

How do we define a floating point number?

You define a floating point variable with the **dim** statement just like you did with integer variables. There is no shortcut to define floating point variables. You must always use the "as float" option.

Example 1.18

```
func main()
```

```
    dim mynumber as float
    global myglobnumber as float
```

```
endfunc
```



This is how a floating point variable is defined.

*The first **dim** statement is a local variable. The second **global** statement is a global variable.*



While a floating point variable can hold values up to 16,677,215 the display routines can only display values in the range of -8388607 to 8388607 with the print command.



The actual display range of the print command for floating point depends on the number of digits to the right of the decimal point.

3 decimal places. Use **{0.3}**
-8388.607 to 8388.607

2 decimal places. Use **{0.2}**
-83886.07 to 83886.07

1 decimal place. Use **{0.1}**
-838860.7 to 838860.7

whole numbers **default** Use **{0.0}**
-8388607 to 8388607

When the print command encounters a floating point number that is out of range for the current display format it will display "OV". Also bit 0 of the FPFLAGS register will be set to 1.

Printing Floating Point Numbers

The **print** command will automaticly detect the use of floating point and display appropiatly.

Example 1.19 will start counting 70000 and increment by 10 until the print limit is reached. At that point "OV" will be displayed until the number is back in range.



You can also use the **setfloatmask** command to change the floating point output format.

Check out the Volume 1 of the Dios Ultra Manual for more information on the **setfloatmask** command.



The format operators `{}` will allow you to change the format for all subsequent output.

The syntax is as follows:
`{X.Y}`

Some Examples:

`{4.2}` Will display `##. to 99.99` with a space allocated for the sign.

`{-04.2}` will display `00.00 to 99.99` with no space for the sign.

`{.3}` will display `. to 8388.607`

For more detailed information see the appendix A in volume 1 of the Dios Ultra Series

Example 1.19

```
func main()
  dim number as float
  number = 70000

loop:
  print number
  number = number + 1
  goto loop

endfunc
```



This example will display:

```
70000
70001
70002
.....
```

until the number is out of range for the display format.

In Example 1.19 the print command displayed a number until it was out of range for the particular display format. To change the display format you can use the format operators.

Example 1.20

```
func main()
  dim number as float
  number = 0

loop:
  print {0.1} number
  number = number + .1
  goto loop

endfunc
```



This example will display:

```
.0
.1
.2
.3
.....
```

until the number is out of range for the display format.



Don't place a comma between the format data and the values.

Example 1.20 will count from 0 to 8,388,60.7 in .1 increments.

Floating Point Restrictions

When a command requires a variable parameter you cannot substitute integer and floating point variables. The command or function syntax will tell you which is required.

Floating point variables cannot be used with the **for/next** commands. If you need to loop, use the **while/wend** commands that we will get into later.

1: Getting Started

Integer Math

Math on the Dios works as you may expect with just a few exceptions. As shown in Example 1.21, we simply place the result variable on the left side of the equal sign and the math expression on the right side. In this case $2 * 7$ will be evaluated to 14 and placed in the variable called value.

Example 1.21

```
func main()

  dim value as integer
  value = 2 * 7
  print value

endfunc
```



Parentheses are permitted in version 2.3.3 or of the compiler. You must also be running version 41 or later chip firmware.

The Dios calculates expression in the order that they are encountered. This is done for speed and simplicity and actually works quite well.

In Example 1.22 the math calculation will work like this: 2 and 1 will be added. 7 will then be multiplied by the result.

Example 1.22

```
func main()

  dim value
  value = 2 + 1 * 7
  print value

endfunc
```



2 is placed in result.

1 is added to result.

Result is multiplied by 7.

The result 21 is placed in the variable value.

You can also use other variables in your math expressions. In Example 1.23 we first assign values to both y and z variables.. Next we take the two variables and use them in an expression assigning them to the x variable.

Example 1.23

```
func main()

  dim x,y,z
  y = 10
  z = 5
  x = y / z
  print x

endfunc
```



When using the shortcut to declare integers you can place multiple variables on the same line.

You can only do this with integers



y (10) is placed in result.

Result is divided by z (5).

Result is placed in variable.



Notice the dim statement in Example 1.21. When the shortcut version of the dim statement is used you may place multiple variables on the same line. Simply separate them by a comma.

A few notes on integer math. In integer math the values will contain whole numbers only. If a result contains a decimal number it is simply removed. The result in Example 1.24 will be 2 not 2.5. There is no rounding in integer math.

Example 1.24

```
func main()

  dim x
  x = 10 / 4
  print x

endfunc
```

Any command that accepts expressions can also be passed a math expression.

The print command is a good example. As shown in Example 1.25, when a math expression is passed to a command it will be evaluated first then passed.

Example 1.25

```
func main()

  print 4 * 5

endfunc
```



The expression is evaluated then passed to the command.

Many commands and functions will call for expressions as input. This can be a simple number or it can be a complex math expression.

1: Getting Started

Floating Point Math

Floating point math works exactly like integer math with a few exceptions.

Floating Point Rounding

When calculating floating point expressions the Dios has the ability to round or truncate the results.

The **round** and **trunc** commands are used to control the rounding of the floating point results.

Take example 1.26. When 10 is divided by 6 the result is 1.666666, which goes on for ever. Notice how it is handled when rounding is turned on and off.

Example 1.26

```
func main()
  round
  print {.3} 10.0/6
  trunc
  print {.3} 10.0/6
endfunc
```



*This one prints 1.667
The numbers to the right of the displayed digits are rounded and the results placed in the last digit displayed.*



*This one prints 1.666
The numbers to the right of the displayed digits are simply chopped off*



In Example 1.26 we used 10.0/6 in the expressions. This is done to tell the print command that this is a floating point expression.

Had we used 10/4 the print command would have assumed this was a integer expression because it has no floating point numbers or variables.



Trunc is the default when the Dios is started.

Signed Numbers

With floating point numbers you gain the ability to do signed math. This is handled internally and is calculated in the expression.

Example 1.27

```
func main()
  dim myvarb as float
  myvarb = 10 - 12
  print myvarb
endfunc
```



Displays -2

for / next

While you can use the **goto** command to create loops, the **for/next** command allows you to create some very efficient and specific loops. In order to use a **for** loop you must use an integer variable to act as an index while the **for/next** command is executed. Let's take a closer look at the syntax used with the **for** command.



In the Dios language the **for/next** command is always checked before the actual loop is executed. This means if the index is out of range it will not execute even on the first pass.

```
for AAA = BBB to CCC
```

AAA is the variable to be used as the counting index
BBB is the starting expression
CCC is the ending expression

Example 1.28 will count from 1 to 10. As long as the counting values are within the range of 1 to 10 the commands between the **for** and **next** commands will be executed.

Example 1.28

```
func main()  
  dim x  
  
  for x = 1 to 10  
    print x  
  next  
  
endfunc
```



This program will loop 10 times. Each time the x variable will be incremented by 1.

There is an optional argument that can be supplied with the **for/next** command. This is the **step** argument. This allows you to specify how much you want to increment the index during each iteration. In Example 1.29 the **for/next** loop will count from 2 to 10 by 2's.

Example 1.29

```
func main()  
  dim x  
  
  for x = 2 to 10 step 2  
    print x  
  next  
  
endfunc
```



*This program will count by 2's because of the **step** command.*

1: Getting Started

You can modify the index variable freely as the for command iterates. Once the end count is reached either by counting or by user intervention, the loop will exit once the next command is reached.

In Example 1.30 we show how to manipulate the index.

Example 1.30

```
func main()
```

```
dim x
for x = 1 to 5
  print x
  x = 10
  print x
next
```



The **for/next** loop will execute 1 loop. Notice that the index variable will contain the correct index until you change it.



Only an integer variable may be used as the index variable in a **for/next** loop.

To count backwards you must use the **step** argument with a negative number as shown in example 1.31.

Example 1.31

```
func main()
```

```
dim x
for x = 10 to 1 step -1
  print x
next
```



Any time you want to count backwards you must use the **step** argument. Supply it with a negative interval.

```
endfunc
```

You can also use a **goto** command to exit a **for/next** loop. A **gosub** command may be used to temporarily exit a **for/next** loop. Once the **return** is encountered it will reenter the loop where it left off. This is the same for functions. You may temporarily exit the loop to call a function but it will return to the loop once the function exits.

if / then / else

The **if/then** command is where real programming can be done.

By combining them with the other looping commands and math expression some pretty advanced constructs can be created.

Let's look at the simplest if then command. I call this the single line if/then. The syntax goes something like this:

If expression1 condition expression2 then do something

The expressions can be just about any variable/math combination. The condition can be any of the following

- = Equal
- < Less than
- > Greater than
- <> Not Equal to
- >= Greater than Equal to
- <= Less than Equal to

In Example 1.32 the state of port 0 is compared with the value of 1. Since we are using an equal sign as the condition operator, the condition will evaluate as true only when the state of port 0 is high.

Example 1.32

```
func main()
```

```
loop:
```

```
  if ioport(0) = 1 then print "Port high"
```

```
  goto loop
```

```
endfunc
```



*If condition is true then everything following the **then** command is executed.*

To get a little fancier in example 1.33 we made the if command a multiline statement. This will allow us to actually process multiple commands if the condition is met.

Example 1.33

```
func main()
```

```
loop:
```

```
  if ioport(0) = 1 then
```

```
    print "Port high"
```

```
  endif
```

```
endfunc
```



*If condition is true then all commands between the **then** and **endif** are executed.*



if/then/else expressions support both floating point and integer variables.

1: Getting Started

To take it one step further we will add the **else** option.

In Example 1.34, by adding the **else** command we give the if statement a place to go if the test condition is false.

Example 1.34

```
func main()
```

```
loop:
```

```
if ioport(0) = 1 then
```

```
  print "Port high"
```

```
else
```

```
  print "Port low"
```

```
endif
```

```
goto loop
```

```
endfunc
```

If the port is high we will print
"Port high"



If the port is low we will print
"Port low"

You can also test two conditions by using the and/or statements. In Example 1.35 both ports 0 and 1 must be high to print "Both high"

Using multiple and/or statements

While you can use multiple and/or statements each will be calculated as they are encountered.

Example 1.35

```
func main()
```

```
loop:
```

```
if ioport(0) = 1 and ioport(1) = 1 then
```

```
  print "Both high"
```

```
else
```

```
  print "Both not high"
```

```
endif
```

```
goto loop
```

```
endfunc
```

In this example we added the **and** statement. With the **and** statement both **if** expressions must evaluate true to pass.



You can have as many **and/or** statements as you wish. They are evaluated as they are encountered. Think of all the expressions as pairs.

The first two will be compared with the **and/or** operator. At that point it will pass or fail. If it passes the next **and/or** statement is evaluated.

while / wend

We looked at the **for/next** loop a little while ago. Now let's look at one more loop. The **while/wend** loop has more flexibility than the for/next loop because you have total control of all aspects of the looping within your program code. You can also loop until a non counting condition is met, although syntax differs greatly.

```
while XXXX comparison YYYY  
.....  
wend
```

xxxx is any expression
yyyy is any expression
comparison is one of a set of comparators like = or <

As long as the expression is true the commands between the **while** and **wend** will be executed. If not, the program will jump to the command following the **wend**.

Example 1.36 shows a very basic **while** loop. It will count from 1 to 10. Notice that the big difference between the **for/next** loop and the **while/wend** loop is that we must do the variable incrementing within the code.



You can use both integer and floating point expressions and variables with the **while/wend** commands.

Example 1.36

```
func main()
```

```
  dim x
```

```
  x = 1
```

```
  while x <= 10
```

```
    print x
```

```
    x = x + 1
```

```
  wend
```

```
endfunc
```

As long as this expression is true.....



We will execute this code.

The while command is not restricted to integer variables you can use floating point variables as well. In Example 1.37 we create a very large counting loop.

You could also increment the index variable by a fraction value as well.

1: Getting Started

Example 1.37

```
func main()
  dim number as float
  number = 0

  while number < 100000
    print number
    number = number + 1
  wend

endfunc
```

You can also create loops that don't use index variables. In Example 1.38 we loop as long as ioport 0 is 0 (low). We can test flags or other conditions as well.

Example 1.38

```
func main()

  while ioport(0) <> 1
    print "port 0 is 0"
  wend

endfunc
```

You can also test two conditions by using the and/or statements. In example 1.39 we count from 1 to 10. However, if ioport 0 goes high the count will be interrupted.

Example 1.39

```
func main()
  dim x

  x = 1
  while x <= 10 and ioport(0) = 0
    print x
    x = x + 1
  wend
endfunc
```



Just like the **if/then/else** commands, the **while/wend** command allows you to have as many and/or statements as you wish. They are evaluated as they are encountered. That is to say you may not use parenthesis.

Introduction

In this chapter we will get our hands a little dirtier. We will discover the real power of functions. You will learn how to define and access array data. You will learn how to manipulate string data. You will also see how to use tables to save string space.

2: The Next Step

Passing Values to Functions

In order for a function to be useful you must be able to pass parameters to a function. You could use a global variable but this defeats the whole purpose of a function.

A function's real power is in its ability to isolate a segment of code from the main program. The only real way to isolate the code is to create a set of input parameters.

The parameter list is created when a function is defined. In Example 2.1 two parameters are shown as width and length.

Example 2.1

```
func printboxtotal(width as integer,length as integer)
  print "Total Box Size = ",width + length
endfunc
```

The parameter list is placed inside the parentheses just after the function name.



You must set the type of each parameter passed. As shown in Example 2.1 both are integer. Once the parameters have been defined they are treated as normal local variables to the function that created them. You can change the values that were passed as shown in Example 2.2.

Example 2.2

```
func printboxtotal(width as integer,length as integer)
  width = width * 2
  print "Total Box Size = ",width + length
endfunc
```

Changed local variables will not effect the values or variables that were used to call the function.

To call a function and pass it values, place the values you wish to pass inside parentheses in the order that they were defined as shown in Example 2.3.

Note: The use of parentheses are optional.



A couple of things happen behind the scenes when you define a parameter list.

- It provides the interface for calling the function.
- It defines the parameters in the list as local variables.

Example 2.3

```
func main()
  printboxtotal(25,10)
endfunc
```



The value 25 is placed in the width variable. The value 10 is placed in the length variable.

```
func printboxtotal(width as integer,length as integer)
  print "Total Box Size = ",width + length
endfunc
```

Returning a value from a function.

A function may also return a value. This is done with the **exit** command. The exit command allows you to return an integer or floating point value depending on how the function was defined as shown in Example 2.4.



When a function exits without the **exit** command it always returns a zero.

Example 2.4

```
func getboxsize(width as integer,length as integer) as integer
  exit width + length
endfunc
```



The exit command is used to return a value.



Here we define the function as an integer. We could also define the function as a float. If no function type is given it defaults to integer.

By default, functions are defined as integer. In order to define a function as a floating point you must use the as float option as shown in Example 2.5.

Example 2.5

```
func formal(value1 as integer,value2 as float) as float
  exit value1 / value2
endfunc
```



Notice the float declarations

2: The Next Step

In order to get the value from a function you must use the function in a variable assignment as shown in Example 2.6.

Example 2.6

```
func main()
  dim bs as integer

  bs = boxsize(25,10)
  print "Box Size = ",bs
endfunc
```



Here is where we make the call to the boxsize function. The result will be placed in the bs variable.

Note: in this case parentheses are not optional.

```
func boxsize(width as integer,length as integer) as integer
  exit width + length
endfunc
```



When you pass values to a function without a matching parameter list, the values are placed into the first defined variables. This can cause unexpected results with floating point variables and values.

You can create some compound assignments as shown in Example 2.7.

Example 2.7

```
func main()
  dim bs as integer
  dim v1 as integer
  v1 = 10
  bs = sqr(v1) + 25

  print "Value = ",bs
endfunc

func sqr(n1) as integer
  exit n1 * n1
endfunc
```

Parameter Conversion

If a parameter has been defined as float and you pass an integer value it will automatically be converted to float before being placed in the parameter variable.

If a parameter has been defined as integer and you pass a floating point value it will get converted as well. In this case you could lose accuracy of the passed value.

Arrays

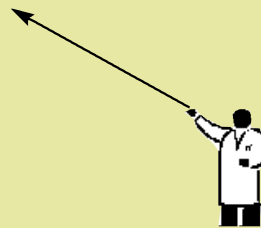
Arrays are groups of variables that are defined and indexed. We use an index to access any single variable in the group.

To define a group of variables you use the `dim` and `global` commands as you do with non array variables.

To define an array of integers you just need to indicate how many are in the group.

Example 2.8

```
dim size(10) as integer
```



This defines a group of 10 integers.

Example 2.8 shows how to define an integer array that consists of 10 integers. It works the same with floating point arrays.



Array indexes are zero based. 0 accesses the first variable. 1 accesses the next and so on.

Example 2.9

```
func main()  
  dim size(10) as integer  
  size(5)=21  
  print size(5)  
endfunc
```

Here we assign the 6th integer in the group the value of 21.



Here we read the 6th integer in the group.

In Example 2.9 we show how to access the individual elements in an array.

You can use the **for** loop to rapidly access multiple array elements as shown in Example 2.10.

2: The Next Step

Example 2.10

```
func main()
  dim size(10) as integer
  dim idx as integer

  for idx = 0 to 9
    size(idx) = 21 + idx
  next

  for idx = 0 to 9
    print "Item ",idx," size = ",size(idx)
  next

endfunc
```



In this for loop we make the assignment to each element in the array.



In this loop we access each element in the array.



There is no bounds checking on the Dios. If you go outside the defined array group you will start writing into the next variable or array. This could be useful. Later in advanced operations we will use this technique to access multiple arrays.

Array Restrictions

While array elements may be used in just about every situation as that of normal variables, there are a few exceptions.

- You may not access the individual bits in an array element.
- You may not pass a whole array to a function, only a single element.
- You may not return a whole array from a function, only a single element.
- When a function expects a variable you may not use an array element.

Many of these restrictions can be overcome and will be covered in advanced techniques.

Byte Arrays

You can access individual bytes in arrays by using the byte extension with the array name as shown in example 2.11.

Example 2.11

```
func main()
  dim size(10) as integer
  dim idx as integer

  for idx = 0 to 19
    size.byte(idx) = 21 + idx
  next

  for idx = 0 to 19
    print "Item ",idx," size = ",size.byte(idx)
  next

endfunc
```

This example works almost identical to the last except we used the byte extension to access each byte element in the integer array.



An integer contains two bytes, so if we define an array of 10 integers we can access 20 bytes individually using the byte extension. It works just like a normal array, but only one byte is accessed at a time.

When you define a float array keep in mind that float variables contain 4 bytes each. So if we define a 10 variable float array we can access 40 bytes. This works great for storing large amounts of 8 bit data.

Accessing Variable Bits

You can access individual bits of integers by using the .bit extension as shown in Example 2.12.

Example 2.12

```
func main()
  dim number as integer

  number=0
  number.bit(2)=1
  print number

endfunc
```

Set the 3rd bit in number to 1.

'00000000 00000100'



You can access up to 16 bits in integer variables and up to 8 bits in registers.

When setting a bit, if the expression equates to 0 then the bit will be cleared. If the expression equates to any thing else it will set the bit.



Like arrays, bits and bytes are also zero based. 0 accesses the first bit or byte. 1 accesses the next and so on.

2: The Next Step

Accessing Registers

The Dios Language has several software and hardware registers that effect how the Dios operates.

These registers are all 8-bit and are accessed just like variables. You access them by name. You can also use the .bit extension to access individual bits.

There are even several user registers that can be used for your own applications as shown in Example 2.13.

Example 2.13

```
func main()
```

```
    BYTE0=21
```

```
    BYTE1=77
```

```
    print BYTE0, " ", BYTE1
```

```
endfunc
```

There are 26 BYTE registers defined for your use. They are labeled BYTE0 - BYTE25



Registers play an important role in the Dios Language and can be broken down into two groups.

Software

These registers will allow you to change certain aspects of the the language. For instance, the floatmask formats can be accessed directly with the FLOATMASK1 and FLOATMASK2 registers.

Hardware

These registers will allow you to access various hardware features such as timers and IRQ's.

The registers will be looked at in detail in later volumes.

Strings

The Dios has real string variables. The Dios has 256 bytes dedicated to string variables. This assures that if you make a mistake you won't cross boundaries into other variables or stack space.

You define a string by using the global statement. All strings are global and can be accessed by name from all functions.

Example 2.14 shows an example of how to define a string. Its much like what we do to define an array.



Strings require a termination character of 0 to let the Dios engine know where the end of the string is located.

Normally the insertion of the termination character is done for you.

Example 2.14

```
func main()
```

```
  global name(20) as string
```

```
  name = "mike"
  print "Hello ",name
```

```
endfunc
```



Define the string here. This string will hold 19 characters. The string variable requires 1 extra byte to hold a termination character.

Let's look at some of the ways we can access and manipulate strings.

String Direct Access

Direct access works much like you would expect. You assign a set of characters to a string using quoted characters or other strings.

Example 2.15 show basic direct access.



You can't use normal math operators when assigning strings. The only two valid operators are:

* move reference to end of string
+ add to string

Example 2.15

'Direct Access with strings

```
func main()
```

```
  global name(20) as string
  global greeting(20) as string
  global misc(50) as string
```

```
  name="Mike"
  greeting="Hello"
```

```
  misc= greeting + " " + name
```

```
  print misc
```

```
endfunc
```



First we define 3 strings.



Next we assign a few values to the strings.



In this assignment we build a string based on other strings.



Here we print the final string

You cannot assign a string variable to itself. For instance you cannot use the expression `myvarb = myvarb + "mike"`.

If you need to add something to the end of a string use the * operator. This tells the target string to move its reference to the end before adding the new value. Example 2.16 shows the use of the append operator.

2: The Next Step

Example 2.16

'Direct Access with strings

```
func main()
```

```
    global names(20) as string
```

```
    names="Mike"
```

```
    names = * + " and Bob"
```

```
    print names
```



This is how we add something to the end of a string.

```
    names = "Hello " + *
```

```
    print names
```



This is not allowed!! You cannot add a string to the front of a string without using other variables

```
endfunc
```



You may only use the append operator "*" to add strings to the end of the target string.

String Insertion

By treating the string as a character array we can place data into the middle of a string. It's like a substring insert.

Example 2.17 shows how its done.

Example 2.17

'String Insertion

```
func main()
```

```
    global testvarb(20) as string
```

```
    testvarb="ABCDEFGH"
```

```
    testvarb(3)= "mike"
```

```
    print testvarb
```



This will display "ABCmikeH"

```
endfunc
```

In Example 2.17 the string of characters "mike" will be inserted into the string variable testvarb starting at the 4th position (remember the index is zero based)

Example 2.18

```
'String Insertion
func main()
```

```
    global testvarb(20) as string
```

```
    testvarb="ABCDEFGH"
    testvarb(6)= "mike"
    print testvarb
```



This will display "ABCDEFmike"

```
endfunc
```

Notice in Example 2.18 the string was extended because the inserted string ran past the end of the original string. In this case the termination character is automatically moved.

If you add a string past the terminator of the target string a few things will happen. The second string will be inserted but no terminator will be written.

Example 2.19

```
'String Insertion
func main()
```

```
    global testvarb(20) as string
```

```
    testvarb="ABCDEFGH"
    testvarb(9)= "mike"
    print testvarb
```



This will display "ABCDEFGH"

```
endfunc
```

Example 2.19 shows what will happen when you write to a string after its original terminator.

When the target string is displayed it shows no changes. This happens because the engine detects the end of the first string and will not allow you to start writing past the end.

If this seems a bit confusing right now don't worry; this is an advance feature and will only be used when you want to mimic string arrays.

Partial String Assignment

It is possible to assign portions of a string to another.

2: The Next Step

Example 2.19

```
'Partial String Assignment
func main()
```

```
global testvarb(20) as string
global myvarb2(20) as string
```

```
testvarb="ABCDEFGH"
myvarb2 = testvarb(1,5)
```

```
print myvarb2
```

```
endfunc
```



The (1,5) says take sting varb testvarb and starting at the 2nd pos (Zero Based) take 5 characters.

This Example will display "BCDF"

If you are familiar with the mid string function in other languages the partial string assignment is very similar.

String Usage

You cannot pass a string to a function. You cannot return a string value. You can however pass a string pointer or return a pointer. This is an advanced topic and will be covered in one of the advanced volumes.

When you assign an integer to a string it will be converted to a character and added to the string.

Example 2.20

```
'Integer assignement
func main()
```

```
global testvarb(20) as string
```

```
testvarb = 65 + 66 + 67
```

```
print testvarb
```

```
endfunc
```



This Example will display "ABC"

If you use the + operator with integer values or variables each item will be added as a character. Example 2.20 shows this.

There are commands that help manipulate string data and they will be covered in one of the advances volumes.



When you use the (x,y) option with strings the string will pull y characters or until it reaches the source string's terminator.



There are many string functions for manipulating string data.

Check out the DiosString library.



The strout command is very powerful. It can print data just like the debug command and place it in a string.

Tables

With the use of tables actual string usage can be kept to a minimum. Think of tables as read only string variables. Tables are actually stored in program memory near the end of your program.

You define a table by using the table command as shown in Example 2.21.

Example 2.21

'Tables

func main()

table name1: "mike"

table name2: "joe"

table name3: "fred"

print name1

print name2

print name3

endfunc

Define 3 tabels here.

Notice how the names are followed with a colon just like a label.

This shows one way of using tables. We can print them just like strings.

You access the table by using its name. You use the name of the table just like a string variable as shown in Example 2.22.

Example 2.22

'Tables

func main()

table item1: "This is cool"

global myvarb(20) as string

myvarb = item1(8,4)

print myvarb

endfunc

As you can see the table name is used just like a string.

This Example will display "cool"

The use of tables goes a long way to saving string space. Use strings for manipulating string data. Use tables to store string data that will not change.

2: The Next Step

Placing Other Data in Tables

You are not restricted to quoted string data. You can also place individual bytes and characters in a table as shown in Example 2.23.

Example 2.23

```
'table example 2  
func main()
```

```
table item1: 65,66,67,0
```

```
'OR
```

```
table item2: 'D','E','F',0
```

```
print item1
```

```
print item2
```

```
endfunc
```



Notice when we place data into the table one byte at a time no termination character is added to the table.

If we are going to use this data with strings we will need to add the string termination character (0) ourselves.



If you use an integer that is larger than 256 it will automatically be divided up and two bytes will be written to the table. The most significant byte first.



The data written to a table is done at compile time so you cannot use variables or expressions. Your options are:

- Quoted string
- Single integer
- ASCII character (with ' operators)
- Hex Value (with \$ operator)
- Binary Value (with % operator)

Tables can be used to store various types of data. This data can be retrieved as a string or as bytes using the **getbyte** command.

These will be covered in one of the advanced topic manuals.

Introduction

In this chapter we will cover software IRQs.

3: IRQ Basics

Types of IRQs

The Dios Ultra has 15 IRQs

- 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
- TX - Transmit event on built-in UART has occurred.

The RC IRQ is handled automatically once the hsersetup command has started the UART receiver. This cannot be manipulated by the user.

The actual detail of each hardware device will be covered in the Dios hardware primer manual.

There are two types of IRQ handlers built into the Dios Language: hardware and software.

Hardware interrupts are an advanced topic and require at least some understanding of assembly language. They will be covered in one of the advanced topic manuals.

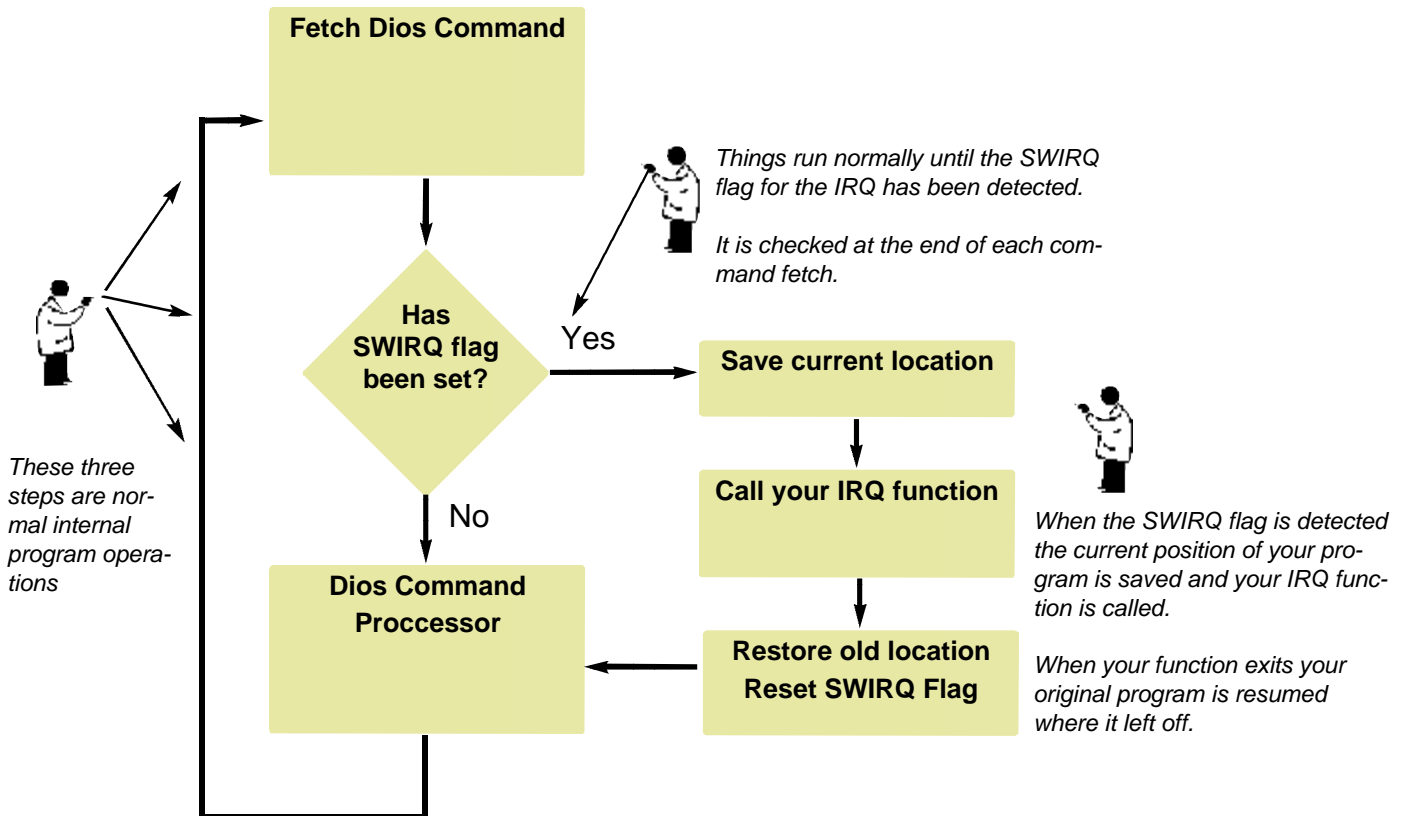
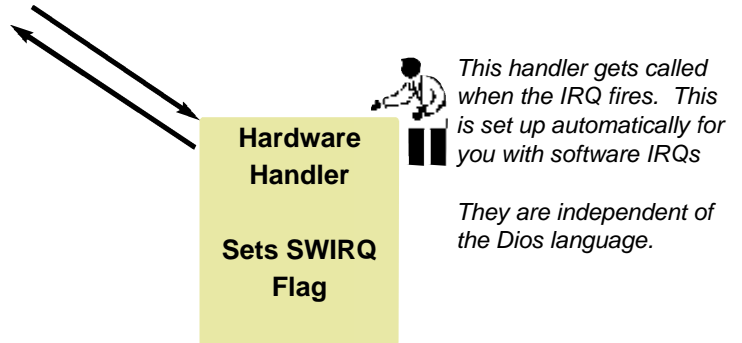
Software interrupts are simpler and have prebuilt handlers that will allow you to execute a Dios function once a hardware IRQ has been detected. These are much simpler IRQs and while they are not as fast as hardware IRQs they do allow you to do multitasking and some pretty neat tasks that would be quite difficult otherwise.

How Do Software IRQs Work?

When a software IRQ has been set up, several things are put in place to handle it. A default hardware handler is loaded that sets a couple of special flags that trigger the Dios to jump to your IRQ function the first chance it gets.

Let us look at a simplified diagram that shows how software IRQs work.

Dios Software IRQ Block Diagram



3: IRQ Basics

Software IRQs may seem a bit intimidating at first. Don't worry; a understanding of what goes on behind the scenes is not necessary with software IRQs.

INT0 IRQ Example

The best way to learn about IRQs is to jump right in and start playing with them. Let us start with the INT0 IRQ. This is an IRQ that fires when IO Port 0 changes state.

Example 3.1

```
'INT0 IRQ Demo
func main()
```

```
  pullupon
```

This command is used to turn some weak pull-up resistors built into the Dios chips.



```
  'Turn on IRQs
  irqglobalstart
```

To use IRQs you must always turn them on. This is done with the irqglobalstart command.



```
  'Start the IRQ
  irqINT0start
  onirqINT0 catchINT0
```

First we start the INT0 IRQ with the irqINT0start command.



Next we need to tell the internal handler which function to use. Do this with the onirqINT0 command.

```
  'Do nothing
  loop:
  goto loop
```

Pass it the name of your IRQ function.

```
endfunc
```

```
irqfunc catchINT0()
  print "INT0 fired"
  exitirqINT0
endirq
```

The irqfunc command is used to start your IRQ function.

Notice that we exit the irqfunction with the exitirqINT0 command. This tells the handler to reset the flag and prepare for another IRQ.



Example 3.1 demonstrates how this is done.

You must always use the **irqglobalstart** command to power up all the internal IRQ hardware. If you don't do this no IRQs will fire.

There are three commands used to enable and handle software IRQs. These are the **irqxxxstart**, **onirqxxx**, and the **exitirqxxx** commands.

The **irqxxxstart** command is used to turn on the hardware associated with the hardware item you wish to handle. The **onirqxxx** command is used to set up your handler function. Think of it as a way to bridge your function with the hardware. When your IRQ function is called this bridge is broken to keep your function from firing while its running.

Use the **exitirqxxx** command to restore this bridge and reset the software IRQ flag. If you fail to use the **exitirqxxx** command, your IRQ function will not



There are two commands that control weak pull up resistors on IO ports 0-7. The IOports are pulled high and held there until you force them low.

pullupon turns on these pullups
pullupoff turns them off.



The **irqglobalstart** and **irqglobalstop** commands are a quick way to stop and start all IRQs.



If you do not use the **exitirqxxx** command, your handler function will only fire once.

You can reset your handler function by issuing the **onirqxxx** command again.

TMR0 IRQ Example

In Example 3.2 we use timer0 to create a background task that will execute once every second or so. This type of IRQ can be used to check the status of some event or device.

Example 3.2

```
func main()
  print "reset"
  output 2
  'Turn on IRQ's
  irqglobalstart

  'Turn on Peripherals
  irqperfstart

  'Start the IRQ
  irqTMR0start
```

The use of timers with IRQs require that you turn on the IRQ peripherals.

Start the timer IRQ handler. This does not start the timer.

```
onirqTMR0 backgroundtic
```

This will be our background processor.

```
'Setup the timer info
timer0sourceclock
timer0mode16bit
timer0prescale 7
timer0on
```

These commands set up the timer.

```
loop:
  goto loop
```

```
endfunc
```

```
irqfunc backgroundtic()
  if ioport(5)=1 then
    print "Port 5 is high"
  else
    print "Port 5 is low"
  endif
```

```
exitirqTMR0
endirq
```

Remember to exit the IRQ function with the proper exitirq command.

!!!IMPORTANT!!!

Software IRQs require that the Dios engine be running in order to process the onirq command.

If you don't want to do any thing as a minimum you need a loop like this one

Again you can see the principle is the same as the INT0 IRQ shown in the last example. There are an unlimited number of variations that can be used with software IRQs.

3: IRQ Basics

Software IRQ things to keep in mind

Software IRQs require the Dios engine to be running. If you use the end command the software IRQs will stop.

Software IRQs operate outside the actual hardware IRQ that fired them. This means that real fast IRQs could fire more than once for each software IRQ that fires. You are, however, guaranteed that at least one software IRQ will fire.

Software IRQs fire on Dios engine command transitions. Some commands take longer than others to complete so the exact interval of the software IRQ handler cannot be determined. For example, if you issue a pause command the software will not fire until the pause command is complete.

Issues with IRQs in general

IRQs in general play havoc with internal timing. Commands that need precise timing like serin and serout, will not operate correctly if a hardware IRQ fires while that command is operating. What can you do? For one you can disable the IRQs before the command and start them back up again after. You can change to a slower baud rate. It is less likely that an IRQ firing would affect a slow serial link at 1200 or 2400 baud.