A programmer's guide to the Dallas DS2904 MicroLan Coupler

The Dallas MicroLan is a system allowing elegant and efficient interconnection of a family of chips with many functions. For a more general introduction to the MicroLan, see **`http://www.arunet.co.uk/tkboyd/e1didx.htm`** If you have no luck with that URL, use Google to search for "Sheepdog Software" (a trademark I own), and/or "TK Boyd".

First I will describe the basic function of the DS2409. Fear not! The engineers put lots of other goodies into the chip, too, and we'll come to those in due course. Some aspects of the chip's operation are not explained in detail in this document, but I believe that almost everything there is to know is at least mentioned.

You can, of course, read the Dallas data sheet directly... this derives from it. If you are an experienced engineer, you will probably find the data sheet more satisfactory. If you are not an experienced engineer, you may find that this document is helpful.

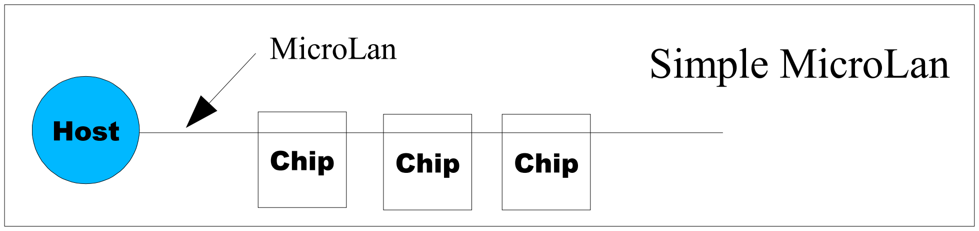
In conjunction with this, you will also find at my website the Delphi source code for the program outlined in this document. It is called DS033. Note that before it will run properly for you, you will have to change the DS2409 chip ID specified in DS033. You may very well also have to change the specification of adapter type and location. See the part of the DS033 source code which puts values into....
saDMLRomID[0], iDMLPortNum, and iDMLAdapterType.

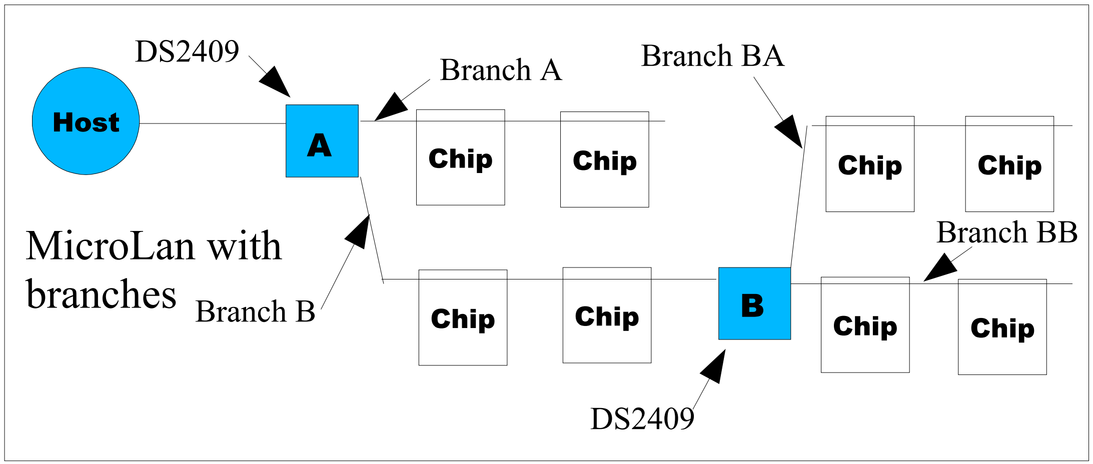
If you find unfamiliar terms in the text, look in the main appendix at the end of the document- you may find definitions there.

One last thing before we get started: This is only the second document I've published via pdf. Do you like it better than the html tutorial pages I have published in the past? Pros? Cons? How to email me is explained at
`http://sheepdogsoftware.co.uk/ctact.htm`

Basic function of the DS2409:

Sometimes, it is advantageous to split a MicroLan into branches. The DS2409 is a switch for accomplishing that. Below is a simple MicroLan, and one with branches....





In the illustration “Chip” represents any 1-Wire chip, sitting on the MicroLan.

The Simple MicroLan just has three chips, all “visible” to the host all the time.

The MicroLan With Branches uses two DS2409s to break up the MicroLan into two primary branches, one of which is further sub-divided. The host can send messages to DS2409 “A” telling it to connect Branch A, or Branch B, or neither to the host. If Branch B is connected to the host, the host “sees” the second DS2409, DS2409 “B”, and can send messages to that to connect Branch BA, or Branch BB, or neither to the host.

You cannot have both of the downstream branches connected at once.

Dallas calls the two branches “Main” and “Auxiliary”. If you connect, say, Main to the host, Auxiliary is automatically disconnected, and vice versa. (In this document I'm going to use slightly unorthodox capitalization rules. If, for instance, you see “Main”, it will mean I'm referring to the chip output called “Main”.)

So far, it sounds like a simple switch, doesn't it? Well... it is.... but with an extra wrinkle: any deselected branch will be connected to a weak source of 5 volts. Without this, the chips on a branch would reset each time the branch was deselected. Undesirable, in general. (There is a way to force a reset if one is wanted. Explained later.) The level of the pin can be pulled down without damaging the chip or making large power demands.

This is fundamental to the basic premises of the MicroLan's design. You may want to use either or both as inputs, and you must not be alarmed by that mention of the pull up to 5v.

Extra features:

What self-respecting engineer is going to leave a chip at such an unexciting (easy to master!) level as that?

The nice people at Dallas thought of a few frills for us...

Control Output:

First, there was an extra pin on the package, pin 5. Not to be wasted! They named it “Control”, and it is an output from the chip. It can drive an LED (or other device... just be sure to use a suitable buffer if you want to drive something that requires more power. Don't try to run a motor with it, for instance, for several reasons. If I've read the datasheet correctly... and I am an amateur... that pin either floats (is connected to nothing) or is connected to ground.

The state of the pin is determined quite simply. (That's the good news.) Neither I, nor, in my opinion Dallas, have found a simple way to describe the simple system! (That's the bad news.)

More good news: You can just turn it on or off “by hand”. (I.e. with direct commands from the host.) Dallas calls this the “manual” mode. I will be including a way to do that in the software that goes with this document. (The software is written in Delphi, source code is available, and it is called DS033.)

Alternatively, you can set the chip so that the Control output's state depends in some way (simple, I think, but I haven't tested this conclusion) on the state of either pin 3 or pin 4... the connections to the Main and Auxiliary branches, respectively. You tell the chip which one to monitor. This mode of Control output control is called “auto-control”. The chip defaults (i.e. comes up from a power-on) to auto-control with the state on the Main branch determining the state of the Control output.

Main and Auxiliary as inputs:

N.B. You can't use Main or Auxiliary as inputs at the same time as you are using the DS2409 to connect a subordinate MicroLan branch to the main MicroLan via the switch. (You can use one for connecting a subordinate branch, and one as an input... but each must remain dedicated to one role.) But! If you have the “switch” open, i.e. the branch disconnected, then you can use pin 3 or pin 4 (Main and Auxiliary, respectively) as inputs, and the host can “see” the state of those pins. You can even set the chip up to watch for a negative edge (see appendix) on the input, and record the occurrence of a negative edge. If set up that way, the host can “see” a brief on/off/on (or off/on/off) event which took place at a time when it wasn't looking at the chip.

Software- part one:

With any collection of 1-Wire chips on a MicroLan, there are two basic parts to doing something with any one of the chips:

a) First you use the general functions of the 1-Wire system to put all of the chips on the MicroLan ***except*** the one you want to talk to to sleep. How do you "target" just one chip? They each have a unique "registration number" or id. How do you put the others to sleep? This is something I'm not going to address in detail here. See the source code for comments, or other discussions I've written about using 1-Wire devices. (http://www.arunet.co.uk/tkboyd/e1didx.htm is worth a try. At the time I'm writing this, I'm aware that my 1-Wire material is poorly organized.... sorry! I *am* trying to improve the organization!)

The functions which are common to all of the 1-Wire chips are referred to as the "Rom Functions" (or commands) in Dallas documentation. Typically, you execute a "Match" command to put all the other chips "to sleep".

b) Once you've prepared the way, as just explained in "a", you select commands from the set of commands that apply to the specific chip you have left "awake". The Dallas documentation refers to them as "Control Functions" (or commands).

In the case of the DS2409, there are six:

Status Read/Write
All Lines Off
Discharge
Direct-On Main
Smart-On Main
Smart-On Auxiliary

See the appendix entry about the Control Functions for a brief summary of what each does. A tour of some of the main points of the main functions begins here. You only need Status Read/Write and Direct-On Main for most basic DS2409 work.

Status Read/Write (Command code 5Ah (i.e. 5A hex, base 16))
This command is not hard to use, but does take a little explaining. I would have started with the other commands, as they are all easier to understand.... but this one is the critical command for the chip!

Immediately after sending the Status Read/Write command, your program should send a "status control byte". More on this in a moment. Next, the DS2409 will send back over the MicroLan a "status information byte". More on this, too, in a moment. Lastly, the DS2409 will send back over the MicroLan a "confirmation byte". That, if all is well, will just be a repeat of the status information byte. It is sent as a check that all is working properly.

Before we can talk about the status control and information bytes, I have to be sure that you have some specific skills. Suppose I said to you that we need a byte made up of zeros in every bit position except bits 2 and 0. Would you understand this makes binary 00000101, which (would you know?) is hex 05, which can be written $05 or 05h? If not, you may want to study "Bits and Bytes", Appendix 4, before starting on the following.

Status Control Byte:

The DS2409 data sheet tells us that the status control byte is made up as follows....

Bit:	7	6	5	4	3	2	1	0
Purpose:	Data	Cntr Sel	Mode	R/not W	R/not W	X	X	X

Clear as mud?

Lets start with the good news: Bits 2, 1 and 0 can have anything in them without affecting how the chip behaves. They're often called "Don't care" bits.

More good news: Bits 4 and 3 must have a zero in them when you send a Status Control Byte to the DS2409, or it will ignore the byte, assuming there has been some error, or that the data got corrupted in transit.

So what about bytes 7, 6 and 5? They control how the chip will behave, and, in some cases, supply data to the chip.

Dallas made understanding what is going on more difficult when they named pin 5 of the chip the "control output". The pin is an output, presumably meant for controlling some further circuit. So "control" is an adjective in that context. Don't confuse that use of the word "control" with it's use in the context of 1-Wire "control functions", ***nor*** with the use of "control" in the phrase "auto-control mode" which is in the data sheet's explanation of the table I presented little way above here!

Before you get even more confused, as I did, let me say this:
The Status Read/Write command cannot be used to change the state of either of the "switches" (neither "main", nor "auxiliary") in the DS2409. They are changed by other commands.

If you just want to make the Control Output "on" or "off", by hand, that's quite easy:
To turn it "on", send binary 10100000 ($A0) as the Status Read/Write command's Status Control Byte.
To turn it "off", send binary 00100000 ($A0) as the Status Read/Write Status Command's Control Byte.
(Having bit 5 a one says you want manual control of the output, and what's in bit 7 determines the state of the "Control Output".

So what's the alternative?

Instead of setting or clearing the "Control Output" by hand, you can set the chip up to switch the "Control Output" when one or the other of the "switches" (either "main", or "auxiliary") is on. The default behavior of the device is that the "Control Output" will be on if the "main" "switch" is on. You can re-establish that behavior by sending binary 00000000 ($00) using Status Read/Write. If you send binary 01000000 ($40) using Status Read/Write, then the "Control Output" will be on if the "auxiliary" "switch" is on.

Now that wasn't so bad, was it????

Moving on.....

After you have sent a Status Control Byte to the DS2409, using the Status Read/Write command, the DS2409 will send you some information. Indeed, you may sometimes *re*-send a Status Control Byte which you've sent previously, which should not "need" sending again, just to get the information the DS2409 sends after receiving a Status Control Byte. That information is sent in a single byte (i.e. 8 bits) called the Status Info Byte. It is made up as follows....

Bit:	7	6	5	4	3	2	1	0
Purpose:	Mode	Cntr Stat	Evnt Main	Evnt Aux	Aux Level	Aux Stat	Main Level	Main Stat

We can do this!!! Hang in there. Nearly done.

Bit 7 merely tells you what state the DS2409 is in with respect to how the "Control Output" is being set. Whatever was in bit 5 of the Status Control Byte should be in bit 7 of the Status Info Byte. You could check that it is, just as an additional check that everything is working properly. I can see no other use for the information. You can't get a Status Info Byte without having just sent a Status Control Byte, so you should already know what mode the chip is in.

Bit 6 is, again, determined by what you just "said" to the DS2409. Consult the data sheet if you feel the need of a headache.

The other bits are more interesting and useful!

Bits 2 and 0 tell you the status of the auxiliary and main "switches", respectively. Remember: The Status Read/Write command is not how you manipulate those switches... but it does allow you to check their states. The DS2409 defaults to both being open. This is good, as, if you choose to, you can use pins 4 and 3 of the chip as inputs... but if you do, you should not close the "switch" to connect 4 or 3 to 2, as long as 4 or 3 is being used as an input. If bit 2 is a zero, then the "Auxiliary" pin (pin 4) is connected to the upstream MicroLan via the "switch" to pin 2. (And if bit 0 is a zero, "Main" is connected.)

Bits 3 and 1, when the "switches" are not closed, tell you the states of pins 4 and 3 ("auxiliary" and "main", respectively). I.e., this is how you read them when using them as inputs. (Bit 3 will be low when pin 4 is low, high when high. Bit 1 works the same way for pin 3).

Nothing in what I've said should lead you to think that you can't route the MicroLan through the DS2409 via pins 2 and 3 (MicroLan In and "Main" out) ***and*** use pin 4 ("Auxiliary") as an input.

One last, optional, delight: Bits 5 and 4, when the "switches" are not closed, tell you the *history* of the states of pins 4 and 3 ("auxiliary" and "main..", respectively).

If there has been a negative edge (see main appendix) on the input since the most recent of:

a) switch was opened,
b) the chip was powered up, or
c) an All Lines Off command was issued....

....then the bit will hold a one. A zero means there's been no negative edge in the period monitored. Note what is implied in the above: The event flags can be cleared by sending the All Lines Off command to the chip.

That's *almost* it for the Status Read/Write command, thank heavens. After the DS2409 sends the Status Info Byte, which we have just discussed exhaustively, it also sends a confirmation byte, which should be a repeat of the Status Information Byte.

All Lines Off (Command code 66h)
One role of this command is to disconnect both Main and Auxiliary from the upstream part of the MicroLan. So far so simple!

All Lines Off will also restore the usual weakly asserted 5v to the disconnected Main and Auxiliary branches. The Dallas documentation calls this “ending a discharge cycle”. You may want to forget the word “cycle”. The Discharge command, as explained more fully in a moment puts the subordinate branches into a discharge state. They stay that way until any further command to the chip. There are advantages to using either All Lines Off or Status Read/ Write as the “further command” which takes the chip out of the discharge state. When the chip is not in a discharge state, the sub-branches are (weakly) held high when not connected to the upstream part of the MicroLan. Again, fairly simple, once you understand the discharge state.

All Lines Off also clears the two event flags. The event flags have already been explained in the section on the Status Read/ Write command.

The Dallas data sheet points out some considerations that you shouldn't overlook when including All Lines Off commands in your programs.

Discharge (Command code 99h)

The coupler is rather clever; it is more than a mere switch. When a downstream branch (either Main or Auxiliary) is not connected to the upstream part of the MicroLan, it is (weakly) connected to a source of 5 volts to keep the chips in the sub-branch in whatever state they have been put into. Without this cleverness, terrible nuisances would arise as the chips in the subordinate branches would go though reset sequences whenever the branch was reconnected to the upstream part of the MicroLan. However, there will be times when you *want* to force those chips to reset, and that's what the Discharge command is for.

When a Discharge command is issued, first both the Main and the Auxiliary branches are disconnected from the upstream part of the MicroLan. Then the sub-branches are both pulled low. Things will remain thus until a new command is issued to the DS2409. A discharge time of at least 100ms is suggested, and it is suggested that the first command to follow a Discharge command is either All Lines Off or Status Read/ Write. There are further details on these points in the manufacturer's data sheet.

Direct-On Main (Command code A5h)
After the Direct-On Main command is issued, pin 2 will be connected to pin 3, i.e. the "Main" sub-branch will be connected to the upstream part of the MicroLan. If the Auxiliary branch was connected previously, it will have been disconnected.

If the output called "Control", which leaves the chip via pin 5, has been configured (see the Status Read/Write command) for "auto-control" operation, and certain other conditions are met, a call of Direct-On Main may cause the output's state to change. (If you study the material explaining the Status Read/Write, you should be able to say what those conditions would be.)

There is no "Direct-On Auxiliary" command. (See the "Smart-On Auxiliary" command.)

Smart-On Main (Command code CCh)
(See note in main appendix entry for control functions... in a nutshell: You'll have to figure this one out yourself.)

Smart-On Auxiliary (Command code 33h)
(See note in main appendix entry for control functions... in a nutshell: You'll have to figure this one out yourself.)

Software- part two:

Now that we have covered the basic elements we will be working with, I want to turn to how you would construct a program to interact with a DS2409.

Because the program will be easy to test, the first thing we are going to do with a DS2409 will be to use manual control to turn the Control output on and off.

Because it is the function I happen to need, we will then work out how to use a Direct-On Main command to close the "switch" that connects the Main sub-branch to the upstream part of the MicroLan. Once we have done these two things, I think you'll be in a position to do anything else you need.

We're going to create procedures to take care of our needs. Once this has been done, we can forget the internals of the procedures and just remember how to call them, and how to interpret the values returned.

See Appendix 1 if you are not familiar with "data types"
See Appendix 2 if you are not familiar with Delphi procedure definitions and calls

The procedure will be called DMLRomMatch. (I use the DML prefix liberally in such things so that they can be pasted into larger programs with a reduced chance of name clashes. "DML" from *D*allas *M*icro*L*an). The procedure declaration will look like this:

```
procedure DMLRomMatch(bIndex:byte;
         var sTKBErr:string;
         var iDalErr:integer);
```

The procedure has three parameters:

bIndex will point to an array which will hold the ID of the chip we want to select.
sTKBErr will return information about errors encountered, or hold nothing.
iDalErr will, if sTKBErr has something in it, return the error code returned by the Dallas routine which failed to execute. (If sTKBErr holds a null string, the contents of iDalErr will be meaningless.)

Procedure DMLRomMatch will not, in itself, achieve anything useful. It will merely prepare the way for one of the following procedures, each of which can only be used if DMLRomMatch has just been called, and called successfully.

To achieve our goal of a program to manually turn the Control output high and low, we'll need:

```
procedure DMLStatusRW(bIndex,bStatusByte:byte;
     var bByteReturned:byte;
     var sTKBErr:string;
     var iDalErr:integer);
```

The procedure has four parameters:

bIndex will point to an array which will hold the ID of the chip we want to select.
bStatusByte will be the byte we want to send to the DS2409.
bByteReturned will be the byte sent from the DS2409 to the host over the MicroLan.

sTKBErr will return information about errors encountered, or hold nothing.
iDalErr will, if sTKBErr has something in it, return the error code returned by the Dallas routine which failed to execute. (If sTKBErr holds a null string, the contents of iDalErr will be meaningless.)

For our program to connect the DS2409's "Main" to the upstream part of the MicroLan, we'll have:

```
procedure DMLDirectOnMain(bIndex:byte;
     var sTKBErr:string;
     var iDalErr:integer);
```

I'll trust you to figure out the parameters! (Read nearby paragraphs above, if puzzled.)

For the details of how those procedures can be implemented, see the source code.

Software- part three:

Heavy going so far? Well now you get your reward. With the procedures explained in "Software- part 2", a program controlling a DS2409 is quite simple to write.

As we said earlier, our first goal will be to turn the Control output on and off "manually".

We'll put a button on our form called "Force Ctrl On". The code behind the button merely calls DMLStatusRW to send the right Status Control Byte to the DS2409. (That label might be called misleading. We are actually, I think, disconnecting the Control output from everything. However, as the alternate state is more truly low, thus reasonably called "off", it seems logical to call the other state "on".)

We'll make a similar second button called "Force Ctrl On". The code behind the button merely calls DMLStatusRW to send the right Status Control Byte to the DS2409.

The next button I will add to the form will be called "Connect Main". The code for this button will simply be a call of DMLDirectOnMain.

At this point, the things I need to be able to do (for the moment!) with a DS2409 have been provided for, and I'm on to the next project. To round out the functions DS033 should have, you should add indicators to the form which can report on the bits of the Status Info Byte returned from a call of DMLStatusRW, and provide buttons to put the device through paces not yet provided for in DS033!

Don't forget that the iButton Viewer (supplied, free, by Dallas with the TMEX drivers) will allow you to exercise most of the functions of the chip. It can't toggle the Control output, though, and, unlike this essay, the iButton Viewer does not come with Delphi source code which I hope helps you get to where you want to be faster than if you had to start from scratch! You can run the program (DS033.exe) that comes with this essay at the same time as you are running the iButtonViewer. Each program will "complain" once in a while, but they won't crash one another, nor damage your hardware. (The "complain" when both try to use the TMEX interface at the same instant.) If you use DS033.exe to close the "Main" "switch", when you have the iButtonViewer DS2409 window open, you should see the virtual switch depicted in that window close.

Appendixes

The main appendix follows the numbered appendices. It defines terms.

Appendix 1: Data Types

(Version of this appendix: 28 July 04)

There are two parts to this. If you already know about data types in general, you may still need to skip down to the paragraphs at the end about Dallas's use of data type terms.

In most languages, data is typed. The two primary types are numeric and string. Even early BASICs didn't allow you to store, say, "Hello World" in a variable that was meant to be holding a number. In Delphi, as in Pascal from which Delphi derives, there is a rich collection of data types available to the programmer. The rest of this appendix speaks from a Delphi perspective.

If you want to use a variable, you have to declare it first. Part of declaring a variable is to say what type of data it will hold. Because the name of a variable can be (almost) anything, without using a voluntary convention, there is no way to know what type of data will be in a given variable without referring back to the declaration. If I'm going to use a byte called Tmp for temporary storage of byte-type data, I call the variable bTmp. If I were going to store string type data, I would have called the variable sTmp. You can see my other conventions by inspecting the source code.

The way Dallas uses some type terms:

There are differences between Delphi and Dallas in their uses of some terms. I'm not convinced that I'm 100% clear on those difference, and, just to keep things interesting, I believe there are some differences between version 1 of Delphi and in later versions.

As an example of how it is easy to become confused, consider the TMEX function TMTouchByte. This can be used to write 8 bits to a 1-Wire chip.... OR to read a longer number... 16 bits, I think it is... from the MicroLan.

If (after the proper preparation) you were to do, say....

```
iD0:=TMTouchByte(liDMLSHandle,$FF);
```

then iDO would be filled with a number read from the MicroLan. If you change the $FF to any other value, and $FF is the largest allowed, then you would write that value to the MicroLan. To have something called "TouchByte" in some cases return something that is more than a byte can confuse!

There is more on this subject in the source code. Bottom line: Be careful!

Appendix 2: Delphi Procedure Calls (Version of this appendix: 28 July 04)

Most programming languages allow you to define procedures. If you define a procedure, you essentially add a new word to the language for use anywhere within the program for which it has been defined. (Unless you place it in a library, and link new programs to the library, the word is not more generally available.)

We'll start with a very simple procedure that creates a word that puts “Hello World” on the screen as a message. We'll call the procedure “Hi”.

Confession: I'm not sure of the *best* place for the following... there are several places it can go... but the place I'm telling you to put it will work!

When your Delphi program was started, near the beginning, the following existed:

```
private
  { Private declarations }
public
  { Public declarations }
end
```

Gradually, things accumulate as you develop the program. By now, that section of your program could look like....

```
private
   { Private declarations }
public
   { Public declarations }
bTmp, bError: byte;
sName, sAge: string;
procedure SomethingOrOther;
function WhatsIt:byte;
procedure Hi;
end
```

(When you look at it in the Delphi editor, some words in the text above will probably be in bold, or have other attribute changes.)

Focus on the block starting “public” and ending “end”. In the second half of that, procedures and functions are declared, the order doesn't matter. Before the first procedure or function are a bunch of variable declarations.

For our “Hi” procedure, add

```
procedure Hi;
```

just before the “end”. This says that there is *going to be* a procedure called “Hi”.

Now, somewhere in the implementation section of the program, assuming you've called the form this is for DS033f1, add the following. (ShowMessage is built in to Delphi.)

```
procedure TDS033f1.Hi;
begin
   showmessage('Hello World');
end;
```

That's all there is to it.... for a simple procedure. (You can change the name of the form. Note the “T” (for Type) in front of the “DS033f1” after the “procedure”.)

The first enhancement of procedures you should know about is passing parameters to the procedure. As an example, we'll consider a procedure that can display error messages. At various places in the source code, you will put things like `RepErr('Overflow')`, or `RepErr('Number too big')`. When the program runs, messages will appear like “There was and error: Overflow” or “There was and error: Number too big”. To achieve this, in the second part of the “public” declarations, you'd put....

```
procedure RepErr(sMsg:string);
begin
   showmessage('There was and error: +sMsg);
end;
```

What's going on? Our “new word”, RepErr, must only be used with a string datum in parentheses following the word itself. In my example, I've shown the string datum being supplied in the least complicated way. The following would be pretty pointless, but would work, and hints at things that would be useful...

```
sTmp:='Value passed to proc in variable';
RepErr(sTmp);
```

You can have as many parameters as you like, and they can be of mixed types. Suppose (Why? Who knows!!) you wanted a procedure that printed some message followed by a number, and the number should be the largest of 5 passed to the procedure with the string to be printed. The procedure declaration would be...

```
procedure RepErr(sMsg:string,
            b1,b2,b3,b4,b5:byte);
```

(This could have been written on a single line, but doing it as above makes it more readable.)

With passing parameters, you can supply the procedure with data for use within the procedure. The second (and last!) enhancement for procedures allows them to pass information back for use by the part of the program that called the procedure.

Suppose, for instance, that you need to find out what 17.5% of a number is, *and* have that number available for other things after the procedure has been executed. (17.5% is the rate of “sales tax” (VAT) in use in the UK as this document is being written.) You might *use* the procedure as follows. (IntToStr is built into Delphi. It merely makes a string out of an integer datum.)

```
iCostOfSocks:='20';
CalcVAT(iCostOfSocks,iTaxOfSocks);
           (*Note that before
           the procedure is called, there
           is no (known) value in iTaxOfSocks*)
iTmp:=iCostOfSocks+iTaxOfSocks;
showmessage('The total cost of the purchase '+
      'will be: '+IntToStr(iTmp));
```

The way the CalcVAT procedure would be made is similar to how we made the previous two procedures:

Add to the “public” part of the code:

```
procedure CalcVAT(iCost:integer;
             var iTax:integer);
```

Add somewhere after “Implementation:

```
procedure TDS033f1.CalcVAT(iCost:integer;
             var iTax:integer);
begin
iTax:=iCost*0.175;
end;
```

The “secret” is in that little word “var”. That is what makes the procedure pass the value back. (You may know about functions, which are another way to pass values back from a subroutine, but unless you also know about records, you won't be able to pass more than one value back with a function, will you? With a procedure, you can have as many “var” qualified parameters as you like.)

Appendix 3: Delphi Function Calls (Version of this appendix: 28 July 04)

(Happily: Not needed for this document!)

Appendix 4: Bits and Bytes

(Version of this appendix: 28 July 04)

What is the difference between having twelve roses, a dozen roses or (6+6) roses? Not much, I hear you say... unless English is not your first language. We don't even normally "see" the difference between, say, 12 and twelve.

When you have been doing certain work with computers for a while, you will not "see" the difference between 1100_2, 12_{10}, $0C_{16}$, $C and 0Ch.... they are all just ways of writing what we generally know as "twelve". Furthermore, you'll become a fluent translator between the different ways of showing the numbers.

To get it out of the way, look at the last three forms: $0C_{16}$, $C and 0Ch. There is a C at the heart of each. Twice there's a leading zero. In daily life, we don't usually include leading zeros.... but they make no difference, so don't let them worry you. Even 00012 is still twelve! The ***other*** differences between the three are merely different ways to say that the number is "written in hex", which is another way of saying "written in hexadecimal", or "written using base 16". Base 16 numbers are made up of the digits we know and love, ***plus*** the letters A, B, C, D and E. The letters may be written in upper case (ABCDE) or lower case (abcde)... the case does not affect what the number is. Note that the "h" used to stand for "hexadecimal" is ***not*** one of the characters used in hex numbers.

"Everyday" numbers, i.e. decimal numbers, are written in various lengths, e.g. 2 French hens, $64000 questions, 101 Dalmatians. As I said a moment ago, we wouldn't normally include leading zeros. In much of the work you'll do with binary and hex numbers, you *will* include leading zeros in order to make binary numbers 8 digits long and hex numbers two or four characters long.

A simple word to get understood: "Bit". It derives from "*bi*nary digi*t". A "bit" is always just a one or a zero. Those are the only digits allowed in binary (base 2) numbers. If a number consists just of 1s and 0s, it may be a binary number. Electronic data sheets sometimes rely on your experience to "know" when something is a binary number. If you see a subscript 2 after the number, e.g. 1100_2, then it is definitely a binary number.

Study at the following....

Decimal	0	1	2	3	4	5	6	7	8
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000
Hex	00	01	02	03	04	05	06	07	08

Decimal	9	10	11	12	13	14	15	16	17	18	19
Binary	1001	1010	1011	1100	1101	1110	1111	10000	11001	11010	11011
Hex	09	0A	0B	0C	0D	0E	0F	10	11	12	13

You won't often care to convert between decimal and the other two systems. I've included the decimal equivalents just to help you stay sane.

You *will* want to convert between hex and binary. For numbers from 0-15, you might just as well use the table above. (If you started in the computing stone age, and had to put 23

12 bit numbers into a computer (PDP-8) *by hand, with switches*, then you will know the binary for 0-7 off by heart. It is a pity that machine's documentation used octal, or I'd know 8-15, too... but that's another story.)

When binary numbers get to be more than 8 bits long, they are converted to hex four bits at a time, starting at the right hand end. (Hex to binary is done the same way, just in reverse.) Thus....

101111_2 is 10 1111_2, which is \$2F
1110000_2 is 111 0000_2, which is \$50
1001000_2 is 100 1000_2, which is \$48
0101001_2 is 010 1001_2, which is \$29
0011010_2 is 001 1010_2, which is \$1A

So why do we **CARE????**

Data sheets sometimes have cryptic remarks like "The 8 bits of the parallel port output register map to the state of the 8 output pins." In this (fictitious) chip, we are being told that if we want the first four pins of the parallel port to be made high, and the last four to be made low, we need to put the following binary number in the parallel port output register: 11110000. The "name" for the right hand bit, by the way, is "bit 0", and the left hand bit is "bit 7". They are also known as the "most significant" and "least significant" bits, respectively, or MSB and LSB. Which is fine... but you'll also find MSB and LSB can mean "most significant ***byte***" and "least significant ***byte*** ". (This will arise when you are working with binary numbers of more than 8 bits. They convert to hex numbers of 2 or more hex characters... 0,1,2,3...9,A,B,C,D,E,F.)

You could have a data sheet for a traffic light control system which had a diagram like this:

Bit:	7	6	5	4	3	2	1	0
Controls:	NR	NY	NG	WR	WY	WG	DC	DC

The abbreviations would be explained as follows:
NR, NY, NG: The red, yellow and green lights, respectively, facing north (and south).
WR, WY, WG: The red, yellow and green lights, respectively, facing west (and east).
DC: Don't Care (i.e. these bits can be 1 or 0).

That data sheet would be telling you that if you sent binary 10000100 to the traffic light electronics, you'd get red lights north and south, and green lights east and west. Another example: binary 01010000 would cause yellow lights north and south, and red lights east and west.

The binary numbers are tedious. Most programmers rapidly learn, and then prefer, to express things like binary 10000100 as \$84, and binary 01010000 as \$50. Try it. You'll like it... eventually!

Main Appendix- definitions of terms

1-Wire: Dallas tends to use 1-Wire and MicroLan almost interchangeably. I tend to use "1-Wire" when I need an adjective, as in "The 1-Wire chips...". See also MicroLan. (Both terms are Dallas trademarks.)

Adapter: 1-Wire chips can be connected to one another exceptionally easily. However, a string of connected 1-Wire chips are of little use without a “master”, also known as a “host”, to send them commands and respond to the signals which come back from the 1-Wire chips. The master will probably be a PC or a micro-controller. While you can avoid using an adapter, that is a tedious route, and you'll have to produce some hardware and software that will be a pain to produce. It is far better to buy an adapter. It connects the PC or the micro-controller to the connected 1-Wire chips. Adapters exist for plugging into a parallel port, a serial port or a USB port. Software to interface your programs with the adapter exist for Windows and Linux. There is a legacy product for working from DOS with TSRs, but I've never quite mastered that. I'm nearly there, and would welcome the chance to exchange a few emails with an “expert”, if any expert is feeling kind! (How to email me is explained at **`http://sheepdogsoftware.co.uk/ctact.htm`**)

Control Functions of the DS2409:

Status Read/Write: This is the most important function, described in detail in the body of this document. It is used to configure the chip, and read various information from it.

All Lines Off: This opens both of the switches, so that nothing connected to Main or to Auxiliary is connected to the upstream (q.v.) MicroLan. It also clears the event flags (q.v.), and ends any discharge cycle which may have been initiated.

Discharge: Initiates a discharge cycle. Opens both of the switches which can connect pin 3 or 4 to pin 2 (1-Wire in from upstream MicroLan), and disconnects pins 3 and 4 (Main and Auxiliary) from the sources of 5v which they would normally be provided with to prevent the downstream (q.v.) chips from resetting when reconnected to the upstream MicroLan.

Direct-On Main: After the Direct-On Main command is issued, pin 2 will be connected to pin 3, i.e. the “Main” sub-branch will be connected to the upstream part of the MicroLan. If the Auxiliary branch was connected previously, it will have been disconnected. There is no “Direct-On Auxiliary” command. (See the “Smart-On Auxiliary” command.)

Smart-On Main: This is a more sophisticated way to connect pins 2 and 3. (See “Direct-On, above.) I'm afraid I haven't investigated this command sufficiently to be able to tell you anything about it.

Smart-On Auxiliary: If you want to connect the upstream MicroLan to the Auxiliary sub-branch, you will have to master the Smart-On Auxiliary command, which I'm afraid I have not done yet.

Downstream: Any system using 1-Wire chips to solve some need will have a PC or micro-controller attached to the 1-Wire chips to send them instructions, and respond to what they sense. The PC or micro-controller is called the system host or master. Some MicroLans are broken up into branches, not all of which are “visible” at all times to the master. Branches are “hidden” by using the subject of this essay, the DS2409 MicroLan coupler chip. Parts of the MicroLan which are between the DS2409 being considered at the moment and the master are called “upstream”. Parts of the MicroLan on the other side of the DS2409 are called “downstream”.

Event Flag: An event flag is a single bit of electronic memory which is being used to record the fact that something has happened. Typically, the flag will be made zero by the chip being powered up. This is sometimes referred to as clearing the bit. When the event that flag watches out for occurs, the bit becomes a one. This is sometimes referred to as “setting” the bit. Thus, the expression “set the bit to zero” may confuse an electronics engineer even thought it would be perfectly reasonable and clear in ordinary conversation. There is always a way to read the flag, to see if the event has occurred, and there is usually a way to clear it, so a subsequent instance of the watched-for event can be detected. The marvelous thing about event flags is that they can “remember” something which happened when other parts of the electronics weren't “watching”. Typically, they detect a negative or positive edge. (See appendix entry for *negative edge.*) The specifics of what event flags are present in the DS2409 are explained in the section of the document concerning the Status Read/Write command.

Host: Alternative name for “master”. See Adapter.

Master: Alternative name for “host”. See Adapter.

MicroLan: Dallas tends to use “1-Wire” and “MicroLan” almost interchangeably. I tend to use MicroLan when I am talking about some 1-Wire chips, the host, and an adapter (q.v.), i.e. the hardware upon which a system is implemented. See also 1-Wire. (The term is a Dallas trademark.)

Negative edge: If the voltage on an input goes from positive to zero, we say a “negative edge” has occurred. If it isn't clear where this term comes from, think about what the trace would look like on an oscilloscope. (A transition from zero to positive is called a positive edge.)

TMEX: I confess that at the moment I am writing this I can't remember where the acronym comes from, but it is an adjective applied to elements of the Dallas supplied API which makes working with 1-Wire chips easier than it would be if we all had to reinvent the wheel of "talking" to the interface between the host pc and the connected MicroLan. The term is almost certainly a Dallas trademark.

Upstream: See Downstream.