

This tutorial describes how to write and compile source code into a macro using the Tiger Development System (TDS) program.

The tutorial is organized in the following chapters and appendix:

## Chapters



### **Getting Started**

Chapter 1 introduces you to the TDS program. It lists the minimum system requirements to successfully run the compiler, describes how to install the software from disk or download, and takes you through the basic steps to launch the compiler after it has been loaded. It also provides definitions of all tools and buttons to get you started.

---



### **Basic Lessons**

Chapter 2 consists of 12 lessons that describe the basic building blocks to write and compile simple macros. The lessons begin with simple macros and introduce new source code development methods in each lesson.

---



### **Intermediate Lessons**

Chapter 3 consists of 17 lessons designed to take you from the basic building blocks into sophisticated macros.

---



### **Advanced Lessons**

Chapter 4 consists of 16 lessons designed to enhance your sophisticated macros with added functionality and sophistication.

---

## Appendices



### **Appendix A**

Appendix A is a glossary of terms used throughout the tutorial and also lists items such as keywords and pre-defined macro labels.

---

## Conventions Used in this Manual

## General Notices & Tips

This document contains four graphic symbols to aid you:



### NOTE Symbol

The **note** symbol is generic to all Tiger Series user manual supplements and indicates important or helpful information, or a reference to further information on the topic being discussed. Where the note is a reference to further information, it stands alone in bold text on a shaded background. Where it is important or helpful information it follows the general flow of the text and is indicated by the symbol only.



### PROGRAMMING TIP Symbol

The **programming tip** symbol is generic to all Tiger Series documents and indicates helpful tips when programming the instrument, or in this case, compiling macros.



### QUESTION Symbol

The **question** symbol forms part of the tutorial structure and poses a question the reader may ask themselves about the topic followed by an answer.



### CAUTION Using the WARNING Symbol

The **warning** symbol is used in this tutorial to draw the reader's attention to the fact that failure to carry out or avoid a specific action in the process of writing a macro can result in loss of data.

## Definitions

### TDS

The term TDS has been used throughout this document and is simply the acronym for the Tiger Development System.

### Meter

The term meter, as used throughout this document, is a generic term for all Tiger Series programmable meter controllers.

# Getting Started

---



# 1

Introduction . . . . .	.2
System Requirements . . . . .	.2
Installing the TDS Software . . . . .	.3
Getting Started . . . . .	.4

## Introduction

### Purpose of this Manual

This tutorial provides 45 lessons describing the steps required to write and compile the source code for a macro using the TDS program. The TDS uses the BASIC programming language and can be downloaded free from our website at [www.texmate.com](http://www.texmate.com). The lessons begin with simple macro instructions and progress to more complicated macro instructions.

### Supplementary Information

To understand the functionality and versatility of our Tiger Series meters, we suggest that the following documents are read in conjunction with this tutorial:

- Relevant Tiger Series meter user manual.
- Programming Code Sheet (NZ101).
- Serial Communications Module Supplement (NZ202).
- Meter Configuration Utility Program Supplement (NZ206).
- Registers Supplement (CA101 or CA102).

Texmate have a range of application specific supplements available covering meter functions such as linearization, totalizing, and setpoint and relay control. Please contact Texmate for an up-to-date list of all available supplements.

## System Requirements

The minimum system requirements to successfully install and run the TDS program are:

- A personal computer with a 486DX 66 MHz processor (Pentium CPU recommended).
- 16 MB of RAM (24 MB recommended).
- VGA or higher resolution monitor (16-bit or 24-bit SVGA recommended).
- One 3.5-inch high-density floppy disk drive.
- A CD-ROM drive (only if software supplied on CD-ROM).
- HTML browser (only if software downloaded from Texmate website).
- Windows 95 or later operating system.

## Installing the TDS Software

The TDS program runs on an IBM or compatible personal computer. The software can be purchased on a CD or is available for downloading at [www.texmate.com](http://www.texmate.com).

### Installation from a CD

#### To install your TDS program from a CD:

- 1) Place the CD into your CD ROM drive.
- 2) Click the **Start** button to display the Start menu.
- 3) Click **Run** on the Start menu.
- 4) At the **Open** prompt, type D:\Setup.exe (or the letter assigned to your CD drive).



Or, click the Browse button and find the Setup.exe file for the TDS program.



- 5) Click the OK button to begin installing the software.
- 6) Follow the onscreen prompts to load the software.

### Installation from a Downloaded File

#### To install your TDS program from a downloaded file:

- 1) Download the TDS program from the Texmate website.
- 2) Double-click on the **Setup.exe** file.
- 3) Follow the onscreen prompts to load the software.

# Getting Started

## Launching the TDS Program

To launch the BASIC Compiler program:

- 1) On your Desktop, double-click the **TDS Program** icon,
- OR



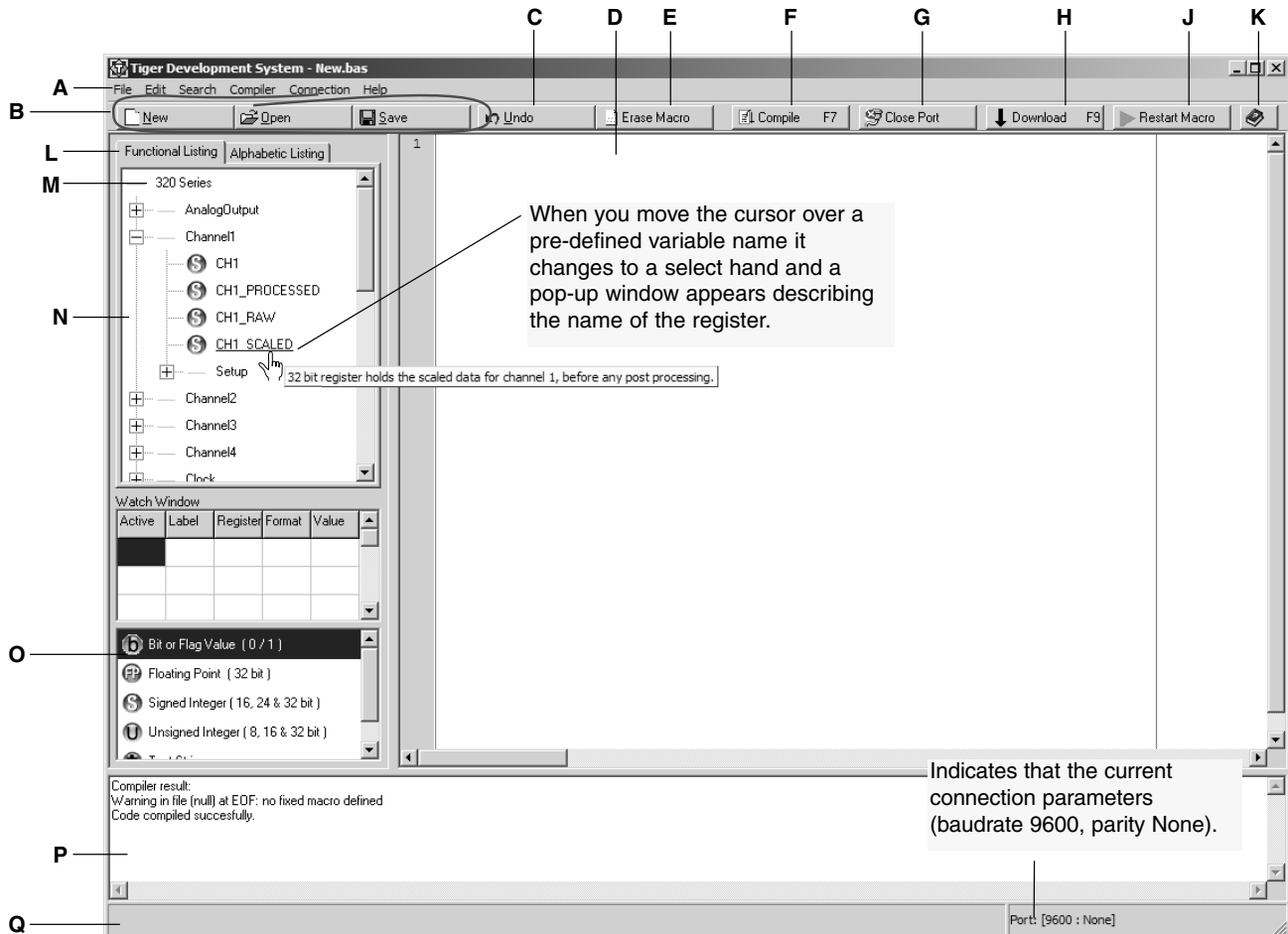
Click the **Start** button and select the TDS program from the Start menu.



A Texmate splash-screen appears and, after a few seconds, the TDS program opens.

## The TDS Program Working Screen

When you launch the TDS program, you'll see a working screen similar to the one shown below (See Working Screen Legend on the opposite page):



## Working Screen Legend

- |   |   |
|---|---|
| <b>A</b> – Main Menu bar.                           | <b>I</b> – Macro On/Off button.                                   |
| <b>B</b> – Standard New, Open, Save option buttons. | <b>J</b> – Start/Stop Macro button.                               |
| <b>C</b> – Undo typing button.                      | <b>K</b> – Online Help.   |
| <b>D</b> – Source Code editor.                      | <b>L</b> – Switcher between Functional and Alphabetical List View |
| <b>E</b> – Erase macro button                       | <b>M</b> – Meter type indication.                                 |
| <b>F</b> – Compile macro button.                    | <b>N</b> – Predefined Variables Tree View window.                 |
| <b>G</b> – Open Port button.                        | <b>O</b> – Tree View Legend                                       |
| <b>H</b> – Download macro button.                   | <b>P</b> – Message window   |
|   | <b>Q</b> – Status bar.  |

## The TDS Toolbox

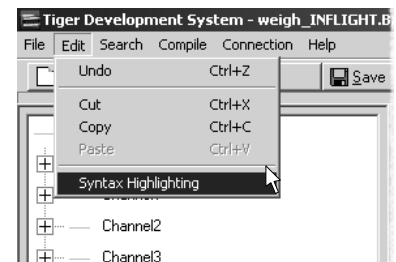
The working screen contains the following tools to write, compile, and edit a macro for any application.

### Main Menu bar

File Edit Search Compile Connection Help

This is the horizontal strip displayed at the top of the working screen and contains the following menu commands:

- **File.** This is a standard Windows-based File menu that contains commands such as New, Open, Save, Save As, and Print.
- **Edit.** This Windows-based Edit menu has standard commands such as Undo, Cut, Copy, and Paste as well as a Syntax Highlighting command.



Syntax highlighting automatically formats the various parts of a macro, such as comments and reserved words. Click on the Syntax Highlighting command to open the Syntax Highlighting dialog box. This dialog box allows you to edit the font, format, and color of the text displayed in the source code editor.

# 1-6

## Getting Started



### PROGRAMMING TIP

The default settings shown here are used throughout the lessons in this tutorial.

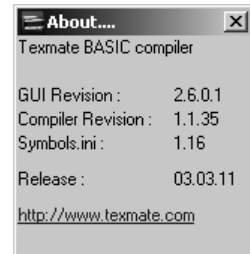
Changes the background color of the source code editor

Changes the type and size of the font used in the source code editor

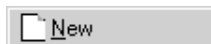
Changes the font color and format of the various text types



- **Search.** This menu provides access to the Find, Replace, and Search Again commands.
- **Compile.** This button compiles the code currently displayed in the source code editor.
- **Connection.** This menu provides access to the Connect, Download, and Macro On/Off commands.
- **Help.** This menu lists the revision status of the TDS program in an About dialog box.



**New button**



Click the **New** button to reset the Source Code editor to a blank screen for typing new code.

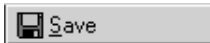


**Open button**

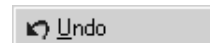
Click the **Open** button to display the **Open** dialog box. Select and open the required macro (macros have a .bas file extension).

**File types**

To edit or compile an existing macro, select **Source** as the file type from the **Files of type** list. This shows source code files with a .bas extension. The compile process generates another file type: the **Macro Code** with a .cmp extension. This file can be opened and downloaded to the meter with the TDS or the Configuration Utility if the source code is not available. This can be useful for OEM users who want to release an update to their customers, but do not want to make the source code public.

**Save button**

Click the **Save** button to save the displayed source code of a new macro, or the edited source code of an existing macro.

**Undo button**

Click the **Undo** button to undo typing in single steps back to the last save.

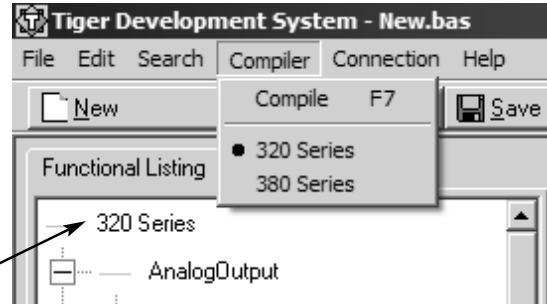
**Erase Macro button**

The erase Macro button is used to erase a macro from the meter. When connected to the meter, pressing the **Erase Macro** button activates an Erase Macro Warning dialog. To continue, press the **Erase Macro** button in the dialog. To abort the procedure, press the **Cancel** button.



### Compile button

Click the **Compile** button to compile the displayed source code into a macro ready to send to the meter. By default the TDS compiles for Tiger 320 meters. You can change the meter type in the Compiler menu.



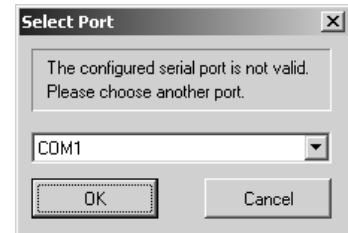
The currently selected meter type is indicated at the top of the register list window.

### Open Port button

Click the **Open Port** button to connect to the meter via the serial port. A **Select Port** dialog box opens to allow you to select another COM port if COM Port 1 is busy or not valid.

When connected, the button becomes the **Close Port** button. Click the **Close Port** button to close the serial port connection.

Being connected to the meter allows you to enable or disable any current macros in the meter using the **Macro On/Off** button, and also erase macros with the **Erase** button.

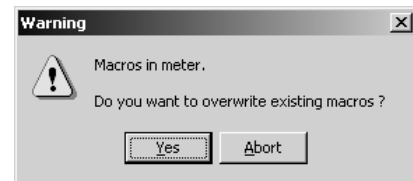


### Download button

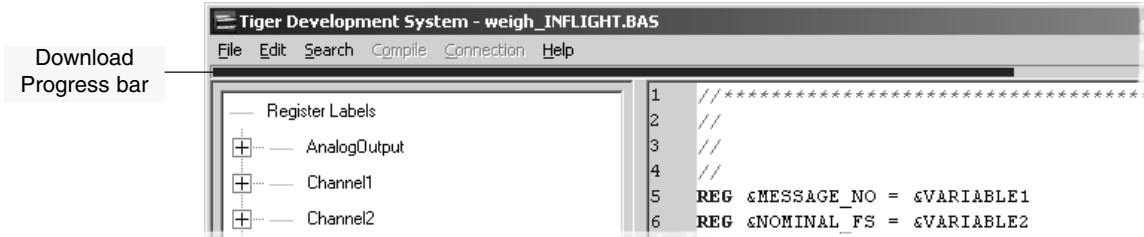
Click the **Download** button to download compiled macros and data to the meter.



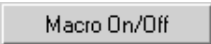
If you attempt to download a new macro to the meter and there are existing macros in the meter, the following **Warning** dialog box appears:



If you click **Yes** to overwrite, the existing macro or macros are erased from the meter's memory. When downloading a macro to the meter, the Download Progress bar indicates the progress of the downloading process.



**Macro On/Off button**



When connecting to the serial port for the first time after power up, the **Macro On/Off** button is inactive. Click the **Open port** button to connect to the meter through the serial port. The **Macro On/Off** button then becomes either **Start Macro** or **Stop Macro** reflecting the current state of the meter.

If the button changes to **Start Macro**, a green arrow appears on the button. This indicates that the macro in the meter is currently turned OFF and can be turned ON by clicking the **Start Macro** button.



If the button changes to **Stop Macro**, a red square appears on the button. This indicates that the macro in the meter is currently turned ON, and can be turned OFF by clicking the **Stop Macro** button.

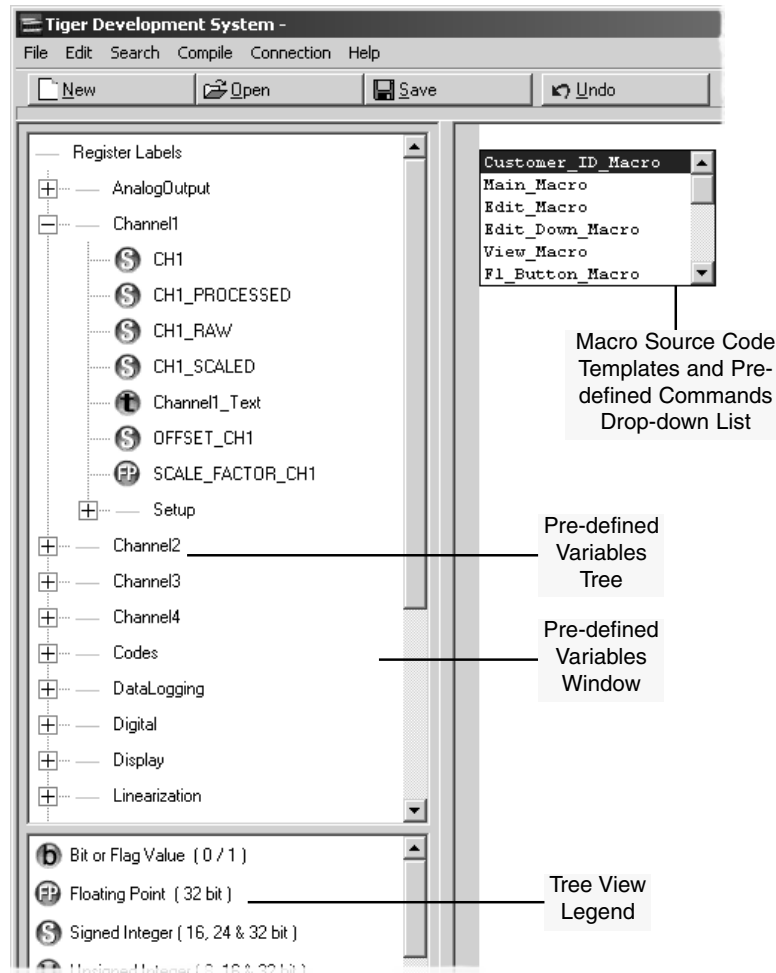


When the serial port is closed by clicking the **Close Port** button, the **Macro** button (whether Stop or Start) becomes the active **Macro On/Off** button again. Clicking this or the **Open Port** button opens the serial port again.



### Source Code editor

The **Source Code editor** is where you type basic source code to be compiled into a macro. Use the right mouse button (right-click) to display a list of source code templates in the **Source Code editor**. These allow you to quickly insert the basic structure of various commands into your source code



### Pre-defined Variables window

The **Pre-defined Variables** window shows a list of all the pre-defined registers available in the meter in a tree view format.

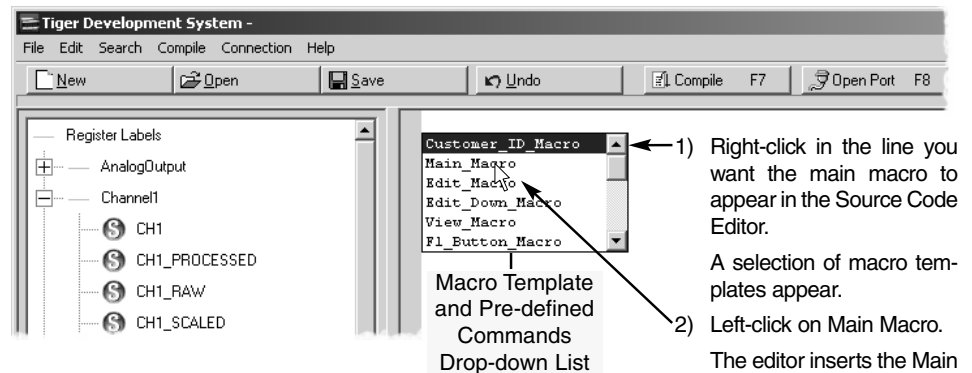
### Tree View legend

Below the Pre-defined Variables window is a legend describing all the symbols used in the tree view.

## Getting Around in the TDS Program

### Adding macros

We will assume that you want to add the **main macro**. Once you have launched the TDS program, proceed as follows:



1) Right-click in the line you want the main macro to appear in the Source Code Editor.

A selection of macro templates appear.

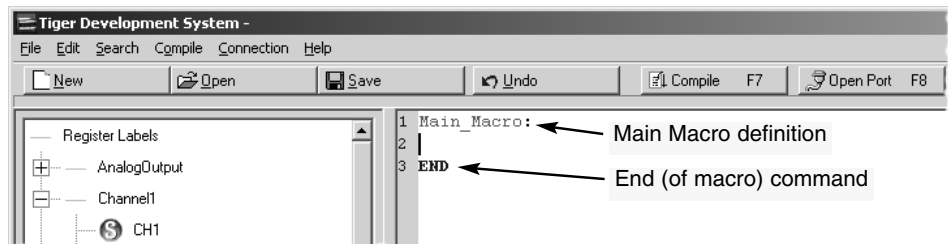
2) Left-click on Main Macro.

The editor inserts the Main Macro definition as well as an end (of macro) command.



### PROGRAMMING TIP

Using the source code templates and pre-defined commands in the drop-down list, ensures that the template or pre-defined command definition is entered correctly and will not generate an error.



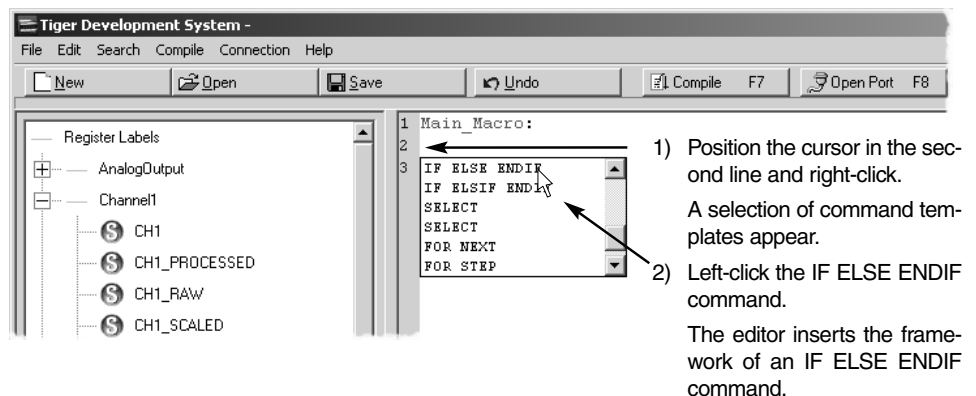
1 Main\_Macro: ← Main Macro definition

2 |

3 END ← End (of macro) command

### Editing macros

We will now assume that you want to add an **IF ELSE ENDIF** command to the **main macro**.



1) Position the cursor in the second line and right-click.

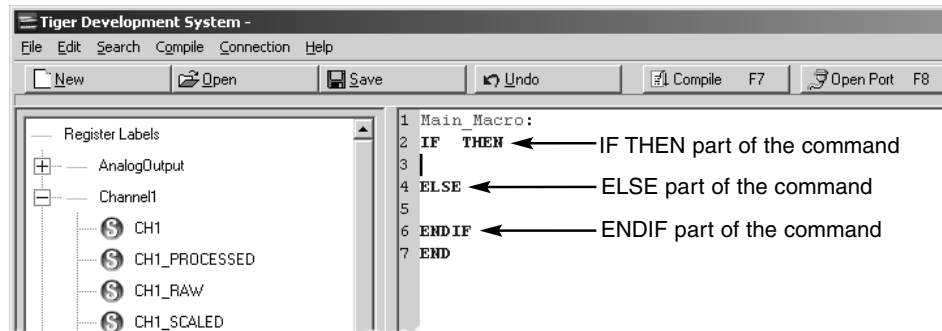
A selection of command templates appear.

2) Left-click the IF ELSE ENDIF command.

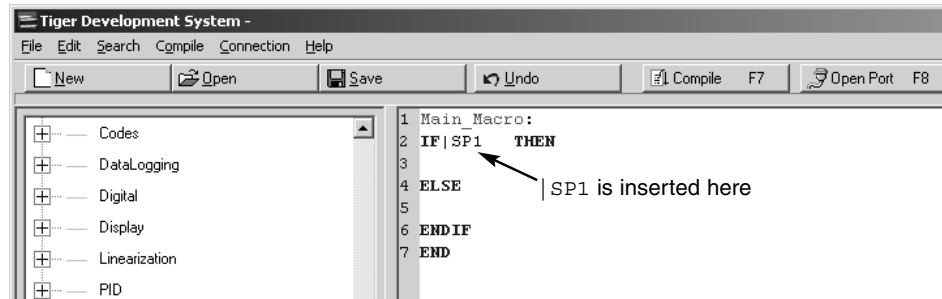
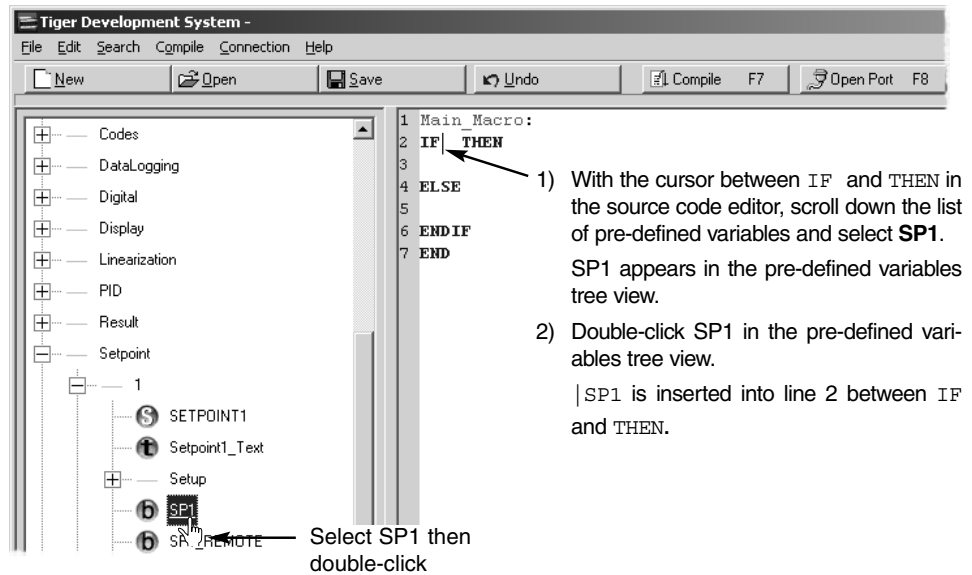
The editor inserts the framework of an IF ELSE ENDIF command.

# 1-12

## Getting Started



Add SP1 as a pre-defined variable to the framework of the IF ELSE ENDIF command.

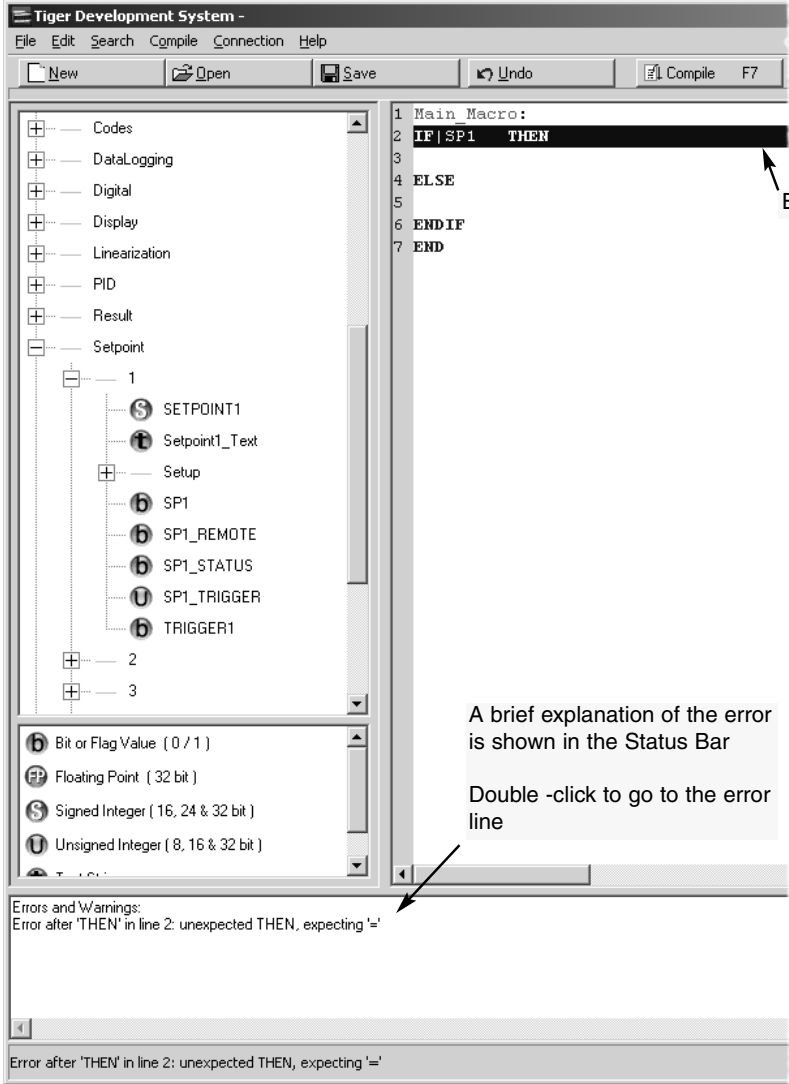


### Correcting Code errors

We will assume that you made an error in your source code. When you click the **Compile** button, a critical error message displays on the screen.

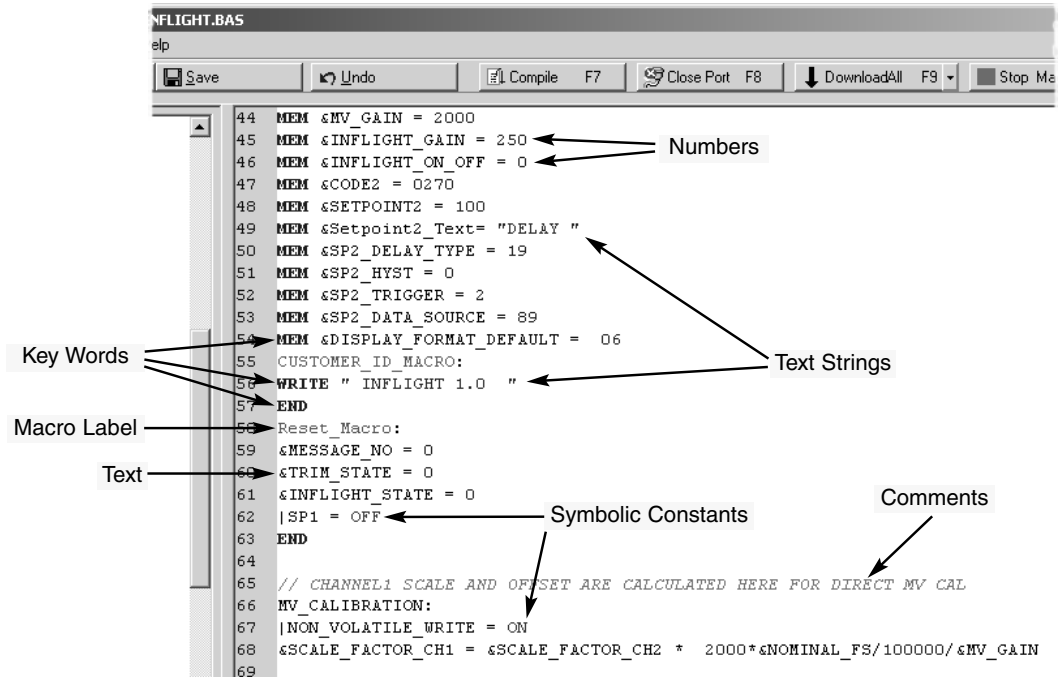


This informs you that there is a compiler error in the macro and a brief explanation of the error is shown in the **Message** window. After pressing the **OK** button, the line where an error is detected is highlighted. Correct the error and try compiling again. If there was more than one error, another critical error message is displayed and the next error in line is highlighted. Keep correcting until all errors have been corrected. All errors must be corrected before the macro will compile.



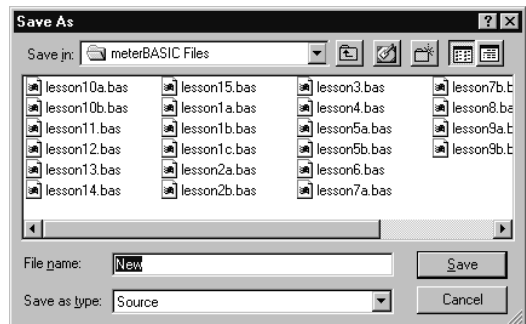
### Syntax Variations

The text used to compile the macro code in the source code editor is automatically formatted by the program when it recognizes certain names or symbols. For example, the default settings for all text is 10 point black Courier New. Anything typed into the source code editor window begins as this. As soon as the program recognizes a key word, macro label, or any of the other syntax variations, it changes the format and color to suit the syntax settings. Pre-defined macro names are case insensitive. Key words can be either upper or lower case, but not both. Register names and any user defined variables, bits, registers, and labels are case sensitive. All ASCII constants are upper case.



### Saving Macros

As you are going to be creating a number of macros while going through the lessons in Chapter 2, it's a good idea to decide on file name conventions and where to save them beforehand. All saved macros have a .bas file extension.





# Basic Lessons

# 2

---

Lesson 1 – The Main Macro . . . . .	.2
Lesson 2 – The IF THEN Command . . . . .	.6
Lesson 3 – The IF THEN ELSE Command . . . . .	.8
Lesson 4 – The IF THEN ELSIF Command . . . . .	.9
Lesson 5 – Registers and Bit Flags . . . . .	.11
Lesson 6 – Nested IF THEN Commands . . . . .	.14
Lesson 7 – Multiple Conditions . . . . .	.16
Lesson 8 – User Defined Variables . . . . .	.18
Lesson 9 – The Reset Macro . . . . .	.20
Lesson 10 – User Defined Bit Variables . . . . .	.21
Lesson 11 – Constant Values . . . . .	.23
Lesson 12 – Basic Maths Operators and Number Formats . . . . .	.25

## Lesson 1 – The Main Macro

### Connect to the Meter

Before attempting Lesson 1, we suggest that you connect the meter to the PC and establish communications.



For full details to connect a meter to a PC and configure the meter in the ASCII mode, see **Serial Communications Module Supplement (NZ202)**.

### Set the Meter to the ASCII Mode

Make sure the meter is operating in the ASCII mode by setting Code 3 to [XX0]. This setting must be correct before attempting to connect.

### The Main Macro

The main macro is a small program that is run repeatedly by the meter 10 or 100 times a second depending on how the meter has been configured.

Now let's write a very simple program in the main macro to scroll the text "Hello World" across the display of the meter.

Type the following in the source code editor.

```
MAIN_MACRO:
WRITE "Hello World"
END
```

**Step 1** In the first line, type the macro name: `MAIN_MACRO:`

As you complete the macro name and type the colon, the text changes color to teal\*.

*\*All colors quoted are default.*

**Step 2** Press Return. In line 2, type: `WRITE "Hello World"`

As you start to type, the word **WRITE** changes color to blue\*. When you type the double-quotes the color of the text string (the words enclosed in the double-quotes) changes to maroon\*.

**Step 3** Press Return. In line 3, type: `END`

As you start to type, the word **END** changes color to blue\*. This is the last line of the program and defines the end of the main macro.

That's it!

**Pre-defined Macro Names**

The first line of this macro is the macro name. It tells the TDS program that you want to place the following source code in the `MAIN_MACRO` area. The `:` (colon) is required to tell the compiler that this is a label as opposed to a register or a variable, which I'll explain about later on. There are a number of pre-defined macro names that exist in the meter and they can all be edited for a particular application. When any of these names are typed in, the name turns teal. Each macro name can only be declared once in each program. *See Appendix 1 for a full list of pre-defined macro names.*

**The Write Command**

The `WRITE` command tells the meter to scroll the text enclosed in quotation marks across the display. The command `WRITE` is a keyword and cannot be used to describe a variable. *See Appendix 1 for a full list of keywords.*

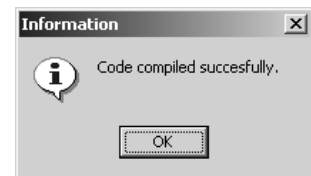
**End Instruction**

The last line defines the end of the main macro. The word `END` signals the macro engine to stop executing macro code instructions and pass control back to the operating system of the meter. Each macro must have at least one `END` instruction to stop executing the macro, though some macros may have more than one `END` instruction.

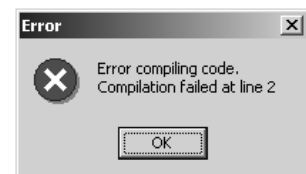


**When editing macros, if using the drop-down list the macro template labels are shown in capitols in the Source Code Editor and the pre-defined commands in capitols bold.**

Now let's try compiling and downloading this macro to your meter. Click on the **Compile** button. When the compiler has successfully compiled the program, an Information dialog box appears and informs you that the macro was compiled successfully. Click the OK button.



If there is an error in the macro, an Error dialog box appears informing you which line is in error. An error message in the message window provides a brief description of the error. Click the OK button to accept this. The line in error is highlighted. The error must be corrected before moving on.



If there are no errors, click the **Download** button.

**Modifying the Text String**

Downloading may take a while on larger programs and you may see the display flicker. Don't worry, this is quite normal.

When it has finished you should see the text "Hello World" scroll across the display. After the text scrolls across the display, the display flashes back to the meter reading for a second and scrolls again, repeating this sequence over and over.

**Congratulations! You have just written your first macro.**

I don't know about you, but I think the text is quite hard to read because it starts scrolling before you get a chance to read it!

To solve this problem, let's modify our program to the following:

```
MAIN_MACRO:
WRITE "   ___Hello World           " //Scroll Hello World
                                     //across the display
END
```

Now let's compile it again and download it.

That's better! By placing three spaces and three underscores ' \_\_\_' before the text and six spaces ' ' after the text, it gives your eyes a chance to adjust to the movement before you see the text.



**A text string can be up to 100 characters long, and may contain letters, numbers, and standard symbols (except the '~' tilde symbol, which is a special symbol used by the macro in text strings). On some types of displays, some letters and symbols cannot be displayed and in some cases two display digits are required to display a single letter. For example, the letters M and W.**

### Adding Comments to a Program

Notice the two forward strokes // (solidus symbols) on line 2 after the text string followed by some more text. When the compiler sees the two forward strokes it ignores everything on that line past that point. This is used to insert comments into your macro source code to explain what the macro is doing.



It is always a good idea to include comments in your code. It helps you and others to understand your code at a later date.



A // comment can follow a command or be on a separate line. If you have a comment that you want to be particularly noticeable, this can be shown as follows:

```
//  
// Scroll Hello World across the display  
//
```

Instead of a comment, a solid line of stars (or another character) can also be used to indicate a division in more complex code, provided it is preceded by //.

```
//*****
```

At the beginning of a line you can also use the REM command to start a comment:

```
REM Hello_World.bas  
REM  
REM last revision 2004 August 23
```

## Lesson 2 – The IF THEN Command

### Adding a Conditional Test

Now let's try something more useful. Suppose we only wanted to scroll a message across the display when a particular event or condition is true. Let's assume that our meter has been configured to display the fluid level in a tank in liters. We could write a macro that scrolls a warning message when the tank reaches 1000 liters. It could look like this:

```

Line 2  MAIN_MACRO:
        IF &DISPLAY = 1000 THEN      // if meter display =
                                        // 1000 then
        WRITE "    ___Warning - tank is Full    "
                                        // scroll text across display
        ENDIF
        END
  
```

In line 2, I have used an **IF THEN** command that works as follows. If the condition after the **IF** instruction is true, **THEN** all of the instructions up to the **ENDIF** instruction are executed. If the condition after the **IF** instruction is false, then the next instruction after the **ENDIF** is executed.

The ampersand symbol **&** in line 2 tells the compiler that the word **DISPLAY** is a predefined register that, in this case, contains the numerical value that is currently being displayed on the meter.



**If the compiler cannot find this register in its list of predefined registers (or if the name is misspelled), it produces an error message in the Status Bar at the bottom of the Source Code Editor window during compiling.**

When the macro runs this code, it looks at the current value on the display, and if it equals 1000, it scrolls the message " \_\_\_Warning - tank is Full \_\_\_".



This macro would work well if the tank stopped filling at exactly 1000 liters, but what if it stopped at 1001 liters or just kept on filling? Our warning might never be seen! We could easily change this macro to solve this problem.

```
Line 2      MAIN_MACRO:
            IF &DISPLAY >= 1000 THEN           //if meter display
                                                    //is >= 1000 then
            WRITE "    ___Warning - tank is Full    "
                                                    // scroll text across display
            ENDIF
            END
```

## Operators

In line 2, the equals sign operator = has been changed to a greater than or equals sign operator >=. Now the message would keep on scrolling if the tank level is greater than or equal to 1000 liters. Here is a list of different operators that can be used with the **IF THEN** instruction.

'='	If equal to.
'<'	If less than.
'>'	If greater than.
'>='	If greater than or equal to.
'<='	If less than or equal to.
'<>' or '!='	If not equal to.



In the macro above you may have noticed that I have indented the line containing the **WRITE** command by 2 spaces from the **IF THEN** command in the line above. This is not mandatory, but it helps to make your code easier to read and makes it easier to see which **ENDIF** belongs to which **IF THEN**, etc. This will become more obvious later on when we look at more complicated macros. You can use spaces or tabs to indent text.

## Lesson 3 – The IF THEN ELSE Command

Let's expand on our previous macro to include two messages instead of one. We could do this by using an **IF THEN ELSE** command, as follows:

```
MAIN_MACRO:
  IF &DISPLAY >= 1000 THEN
    WRITE "    ___Warning - tank is Full      "
  ELSE
    WRITE "    ___Tank is still filling      "
  ENDIF
END
```

### Adding a Second Condition

The first three lines of this macro are identical to our previous macro, but in line 4, I have added an **ELSE** instruction. Now, if the condition in line 2 is true, all the code up to the **ELSE** instruction is executed, and then the next instruction after the **ENDIF** is executed. In our simple macro, we only have one command (i.e. the **WRITE** command) between the **IF THEN** test and the **ELSE**, but we could have many more commands, including more **IF THEN ELSE** commands. Later, in Lesson 6, we discuss nested **IF THEN** commands in more detail.

If the condition in line 2 is false, then all the instructions between the **ELSE** and the **ENDIF** instructions are executed. Once again, we could have more than one command between the **ELSE** and the **ENDIF** instruction.



## Lesson 4 – The IF THEN ELSIF Command

We could add more conditions to our macro by using the **IF THEN ELSIF** command as follows:

```
Line 2      MAIN_MACRO:
Line 3      IF &DISPLAY > 1050 THEN
Line 4      WRITE "    ___Warning - tank is about to over " + \
            "flow! "
            ELSIF &DISPLAY >= 1000 THEN
            WRITE "    ___Tank is Full          "
            ELSE
            WRITE "    ___Tank is still filling      "
            ENDIF
            END
```

### Multiple Test Conditions

This macro functions in a similar way to our previous macro except that now we have two test conditions. If the condition in line 2 is true, then all the code up to the **ELSIF** instruction is executed, and then the next instruction after the **ENDIF** is executed. If the condition in line 2 is false and the condition in line 4 is true all the code up to the **ELSE** instruction is executed, and then the next instruction after the **ENDIF** is executed.

If the conditions in line 2 and line 4 are both false, then all the commands between the **ELSE** and the **ENDIF** instructions are executed.

In this example I have only used one **ELSIF** instruction, but I could have any number of **ELSIFs**, one after the other.



The only conditions are that all **ELSIF** instructions must come after an **IF THEN** and all of the **ELSIFs** must be placed before an **ELSE** command.

The **ELSE** command is optional, so you could have only an **IF THEN** instruction followed by one (or more) **ELSIF** instructions, and then the **ENDIF** instruction. If you do use the **ELSE** instruction, then it must be used last, just before the **ENDIF**.

At the end of line 3 you'll notice the symbols `+ \` after the text string, followed by some more text on line 4 . The `+` symbol can be used with the `WRITE` command to tell the compiler that there is more text to be added to this string (we'll look at this in more detail when we get to Lesson 30). The `\` symbol can be used with long command lines to make the code easier to read. It simply tells the compiler that there is more information for this command on the next line. When the compiler sees the `\` symbol, it stops compiling and continues on the next line. The `\` symbol can be used as many times as you require in a single command.

The use of the `\` symbol is optional. The compiler still works with long command lines but you will have to use the scroll bar at the bottom of the source code editor to view all of the text.

## Lesson 5 – Registers and Bit Flags

### Pre-defined Variables

In our previous macros we have used a pre-defined register called &DISPLAY, which contains a numerical value corresponding to the reading on the front of the meter. The meter contains many different pre-defined registers, several thousand in fact! But don't worry, for most macros you will only need a small number of these.



In the pre-defined variables window, you will see a list of the available pre-defined registers laid out in a tree view style menu. Each pre-defined register holds data relating to a different function in the meter. The size of the pre-defined registers vary between 8-bits, 16-bits, and 32-bits, and the format of the data might be in fixed point or floating point. For most macros you don't have to worry about this because the macro engine in the meter knows exactly how to handle each pre-defined register.



You will have to know a little about which registers to use in a macro. To help you with this, you can read a brief explanation about each pre-defined register by holding your mouse pointer over the register name in the tree view register list in the pre-defined variables window.



**For more information on program navigation, see Getting Around in the TDS Program in Chapter 1.**

### Inserting Pre-defined Registers



As stated earlier, every time you use a pre-defined register you must put an ampersand & in front of the register name to tell the compiler that this is a predefined register.

A quick way to insert a pre-defined register label into your program is to set the cursor to the position in the source code editor window where you want to insert the register name. Then find the register name in the list and double-click on it. It is automatically inserted into your source code with an & in front of it!

**Pre-defined Bit Flags**

As you look down the list of pre-defined registers you will also see some pre-defined bit flags (see the tree view legend). These are similar to pre-defined registers. The only difference is that these are only one bit wide, meaning that their data range is only from 0 to 1, or 'OFF' and 'ON'. They are usually associated with digital input pins, front panel buttons, relays, etc. Let's re-write the macro we used in Lesson 4 using setpoints instead of checking the level ourselves. I'll assume that we have programmed setpoint 1 to 1050 for the overflow level, and setpoint 2 to 1000 for the full level. This is how it looks:

```

Line 2      MAIN_MACRO:
Line 3      IF |SP1 = ON THEN      // if setpoint 1 has activated
Line 5      WRITE "    ___Warning - tank is about to over" +\
            "flow! "
            ELSIF |SP2 = ON THEN  // if setpoint 2 has activated
            WRITE "    ___Tank is Full      "
            ELSE
            WRITE "    ___Tank is still filling      "
            ENDIF
            END

```

You will see that only lines 2 and 5 have changed. In line 2, I am testing the bit flag |SP1 to see if it is ON or OFF. The '|' symbol tells the compiler that |SP1 is a bit flag that has only two possible states, ON or OFF. |SP1 is a flag that shows the status of setpoint 1. If it is ON it means that setpoint 1 has been activated.

A similar test is done in line 5 but this time on |SP2, which is the status flag for setpoint 2.

Functionally our new macro is the same as the one we wrote in Lesson 4, however, it has a few advantages in practice. If the tank levels need to be changed in the Lesson 4 macro, then the macro needs to be re-written. With our new macro, the setpoints can be easily changed from the front panel of the meter and our macro still works correctly. Also, now we have 2 relays switching at these levels that could be used to turn off a valve or start up a pump, etc.

In most cases, bit flags are only tested for two possible states, ON or OFF. However, some bit flags have a third special state called NORMAL. This is used with bit flags that are normally controlled by the meter, but can also be used in a remote mode.

For example, in the previous macro, we are only testing the bit flag |SP1. The flag itself is controlled by the meter being set or cleared, depending on the input signal, the value of the setpoint, etc. So we would only need to test this flag for an ON or an OFF state. This is the normal mode of operation.

However, in some macros it might be necessary for the macro to take control of a particular setpoint, either continuously, or for a short period. Here's a simple macro that shows how this is done:

```
MAIN_MACRO:
  IF &DISPLAY > 1050 THEN
    WRITE "    ___Turning pump on    "
Line 4    |SP1 = ON           //macro takes control
Line 5                                     //of SP1 and turns it on
    ELSE
Line 7    |SP1 = NORMAL    //returns control of SP1 to
                                     //setpoint logic in meter
    ENDIF
  END
```

In line 4, the instruction |SP1 = ON tells the compiler to put setpoint 1 into the remote mode and turn it on. In this mode, the setpoint logic in the meter no longer has any control over the setpoint and it remains on until the macro changes.

In line 7, the instruction |SP1 = NORMAL tells the compiler to put setpoint 1 back into the normal mode of operation, where the setpoint is controlled by setpoint logic in the meter.

This special case applies to |SP1 to |SP6 and |LED1 to |LED6.

## Lesson 6 – Nested IF THEN Commands

Well that's all there is to the **IF THEN** type command! Of course, in practice you will probably find that there is more code between each of the tests. Often you will also need to test more than one condition before doing a certain task.

### Nested IF THEN Commands

The way that the TDS does this is by allowing multiple **IF THEN** commands. These are referred to as nested **IF THEN** commands. Let's have a look at how this works:

```

MAIN_MACRO:
  IF |SP1 = ON THEN // if setpoint 1 is activated and
    IF |SP3 = ON THEN //if setpoint 3 is
                        //activated and
      IF |SP4 = OFF THEN //if setpoint 4 is
                          //de-activated
        WRITE " ___Warning - tank is about to " + \
          " over flow! "
      ELSE
        WRITE " ___Sorry, but you've got a big " + \
          "mess to clean up! "
      ENDIF
    ENDIF
  ELSIF |SP2 = ON THEN // if setpoint 2 has activated
    WRITE " ___Tank is Full "
  ELSE
    WRITE " ___Tank is still filling "
  ENDIF
END

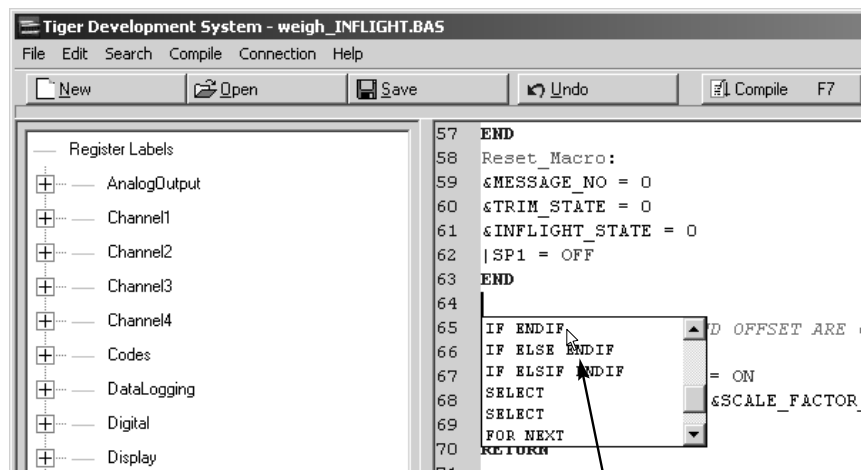
```

In the macro above, the text "Warning - tank is about to over flow!" is only displayed if setpoint 1 is **ON** and setpoint 3 is **ON** and setpoint 4 is **OFF**. The text "Sorry, but you've got a big mess to clean up!" is only displayed if setpoint 1 is **ON** and setpoint 3 is **ON** and setpoint 4 is **ON**.

The logic for nested **IF THEN** commands is exactly the same as the previous lessons. You can use **ELSIF** and **ELSE** with nested **IF THEN** commands provided that you keep to the same rules mentioned in Lessons 2 to 4. The main thing to remember is that for each **IF THEN** command, you must have an **ENDIF** instruction. If a condition is true, then the code under that condition is executed up until the next **ELSIF**, **ELSE**, or **ENDIF** instruction. If you try to compile a macro that has an unequal number of **IF THEN** and **ENDIF** instructions, the compiler displays an error.



The TDS program has some handy features to ensure that you always have the right number of **ENDIF**s. When you want to insert a new **IF THEN** command, position the cursor where you want the command to start and then right-click. This brings up a selection of templates. Left-click on the command you want and the editor inserts it into your code along with the **ENDIF** command! Now all you need to do is fill in the blanks. If you use this feature each time you start a new command, you won't have the problem of inadvertently forgetting an **ENDIF** command.



- 1) Place the cursor where you want the new **IF THEN** command to start.
- 2) Click the right-hand mouse button. A selection of commands appear.
- 3) Left click the selection you want. The editor inserts it into your code along with the **ENDIF** command.

## Lesson 7 – Multiple Conditions

In Lesson 6, I explained how nested **IF THEN** commands can be used to test for several conditions. Unfortunately, this tends to become hard to follow. Have a look at the following main macro to see what I mean:

```

MAIN_MACRO:
  IF |SP1 = on THEN           // if setpoint 1 is
                              // activated and
    IF |SP3 = ON THEN        // if setpoint 3 is
                              // activated and
      IF |SP4 = OFF THEN     // if setpoint 4 is
                              // deactivated
        WRITE " over flow! "
      ENDIF
    // or if capture pin is connected to common
    IF |CAPTURE_PIN = ON THEN
      WRITE " over flow! "
    ENDIF
  ENDIF
ENDIF
END

```

In this macro, it doesn't matter if setpoint 4 is deactivated or if the capture pin is activated, in either case the overflow message is written but this is not obvious.

With newer compiler versions you can test for multiple conditions. Single conditions can be logically combined using **AND**, **OR**, and parentheses. So we can now rewrite the above macro to look like this:

```

MAIN_MACRO:
  IF |SP1 = ON AND |SP3 = ON AND \
    (|SP4 = OFF OR |CAPTURE_PIN = ON) THEN

    WRITE " over flow! "
  ENDIF
END

```



Well, I don't know about you, but this seems much simpler to me! Now we have used only one IF THEN command, which says:

IF ISP1 and ISP3 are both ON, and IF ISP4 = OFF or the |CAPTURE\_PIN is ON, then display the message "over flow!".

**CAUTION:**

AND and OR coincide with the logical bit operators. Therefore, logical bit operations have to be put inside parentheses in conditions:

```
MAIN_MACRO:
  IF (&DISPLAY AND 0x00000001) = 0 THEN
    // display value is even
  ENDF
END
```

We will look at this in more detail in Lesson 32.

## Lesson 8 – User Defined Variables

### User Defined Variables

Often when writing a macro you will find that you need somewhere to temporarily store a data value so that you can use it later in your macro. In the TDS these are called user defined variables. When you define one of these variables, the TDS reserves some memory in RAM to store your data. You can define a variable using any name you wish provided it is not a keyword (*see Appendix A for a list of keywords*). Your name can include letters, numbers, and underscores `_`, provided the first character in the name is always a letter.

User defined variables are prefixed with a pound sign `#` or percent sign `%` depending on what type of variable you need. The `#` indicates that your variable is a fixed-point number, meaning that it can contain any integer between `-2147483648` and `2147483647`. The `%` indicates that your variable is a single precision floating point number between `+/- 1.175494e-38` and `+/-3.402823e+38`.

Let's write a simple macro that uses both predefined registers and user defined variables:

```
Line 2      MAIN_MACRO:
            #MY_RESULT = &CH1+&CH2           //define a variable
                                                //called MY_RESULT
Line 4      IF #MY_RESULT < 10000 THEN // test MY_RESULT
            WRITE "    ___Tank is still filling    "
            ELSE
            WRITE "    ___Tank is Full            "
            ENDIF
            END
```

In line 2, I have generated a variable called `MY_RESULT` and because I only want an integer result, I have placed a `#` symbol in front of it. I have then made the variable `#MY_RESULT` equal to the sum of the two predefined registers `&CH1` and `&CH2` (which happen to be channel 1 and channel 2 data registers respectively).

Then, in line 4 I am testing the variable `#MY_RESULT` with an `IF THEN` instruction in the same way that we did earlier on with predefined registers. In fact, once you have defined a variable, you can use it in the same way as a predefined register, provided you always add the appropriate prefix (`#` or `%`).



You can include up to 10 different integer variables and up to 8 different floating point variables (20 integer and 16 floating point variables in the Tiger 380) in your macro. If you try to use more than this, you will generate a compiler error.

An important point to note is that you must always define a variable as equal to some value before you use it in an expression or as part of a conditional instruction.

Try compiling the following macro:

Line 2

```
MAIN_MACRO:
&CH1 = #MY_RESULT+&CH2
IF #MY_RESULT < 10000 THEN
    WRITE "    ___Tank is still filling    "
ELSE
    WRITE "    ___Tank is Full        "
ENDIF
#MY_RESULT = &CH3+&CH2
END
```

You should find that the compiler comes up with an error in line 2. That's because the TDS is a single pass compiler and when it gets to line 2 it hasn't allocated the name `#MY_RESULT` to a memory location yet, so it doesn't recognize it! The next lesson looks at the `RESET_MACRO` which should solve this problem.

## Lesson 9 – The Reset Macro

All of the macros we have looked at so far have been written for the `MAIN_MACRO`. As we mentioned at the beginning of this chapter, the `MAIN_MACRO` is run (or called) repeatedly by the meter, 10 or 100 times a second, depending on how the meter has been set up. Now we'll look at a different macro area called the `RESET_MACRO`. As you might have guessed from the name, this macro is only run once, when the meter is first turned on. Its purpose is to initialize any registers or variables at power-up so that the meter and macros always start operating from a known state. In many macros you won't need to worry about this, but in others you will. Let's take the second macro from Lesson 8 and include a `RESET_MACRO` to define the variable `#MY_RESULT` as being equal to `ZERO` initially.

```
RESET_MACRO:
#MY_RESULT = 0           //define a variable
                        //called MY_RESULT

END

MAIN_MACRO:
&CH1 = #MY_RESULT+&CH2
IF #MY_RESULT < 10000 THEN
    WRITE "    ___Tank is still filling      "
ELSE
    WRITE "    ___Tank is Full              "
ENDIF
#MY_RESULT = &CH3+&CH2
END
```



Now this will compile okay. Remember, because you need to define your variables before you use them in an equation or test, the `RESET_MACRO` should be placed before any other macros in your file.

The only difference between the `RESET_MACRO` and the `MAIN_MACRO` is when it is run (or called) by the meter. Apart from that, they are both capable of executing the same commands. You could write any macro you want and place it in the `RESET_MACRO`. When you switch on the meter it would run your macro once!

## Lesson 10 – User Defined Bit Variables

In Lesson 8, I introduced user defined variables for integer and floating point values. But in many applications you only want to store whether a condition is met or not – or in terms of programming languages whether it is true or false. Of course you could use an integer variable and set it to 1 or 0 respectively. This would work out fine for macros where you don't need many variables. But you would also use up a 32-bit register to store a single bit. The simple solution for this problem is user defined bit variables.

User defined bit variables are prefixed with an '|' symbol (just like bit registers) with the same restrictions to its name as user defined variables: any mixture of letters, numbers, and underscores \_, provided the first character in the name is always a letter. The allowed values are `on` or `true` and `off` or `false`.

```
RESET_MACRO:
// initialize bit variables
|OPERATION_COMPLETE = false
|LED1_AT_RESET = off

Line 6   IF |LED1 = on THEN
Line 7   |LED1_AT_RESET = on
Line 8   ENDIF
        |LED1 = off
        END

MAIN_MACRO:
IF &CH1 < 0 AND (|HOLD_PIN = off OR |LOCK_PIN = off) \
THEN
        |OPERATION_COMPLETE = true
ENDIF

IF |OPERATION_COMPLETE = true THEN
Line 20  WRITE "      Operation complete      "
Line 21  IF |LED1_AT_RESET = on THEN
Line 22  |LED1 = on
        ENDIF
ENDIF
END
```

In lines 6-8 of the previous macro, the value of a bit register is stored in a bit variable and its value is restored in lines 20-22. As this is rather common, the TDS allows you to equate one bit variable to another one. Look at the next macro to see what I mean:

```

RESET_MACRO:
// initialize bit variables
|OPERATION_COMPLETE = false
Line 4  |LED1_AT_RESET = |LED1

|LED1 = off
END

MAIN_MACRO:
IF &CH1 < 0 AND (|HOLD_PIN = off OR |LOCK_PIN = off) \
THEN
                                |OPERATION_COMPLETE = true
ENDIF

IF |OPERATION_COMPLETE = true THEN
    WRITE "      Operation complete      "
Line 17  |LED1 = |LED1_AT_RESET
ENDIF
END

```

In lines 4 and 17 I have replaced other **IF THEN** commands with a line | which simply makes one bit variable equal to another one.



**CAUTION:**

You can include up to 32 different bit variables in your macro that will actually use up a single integer variable. If you try to use more than this, you will generate a compiler error.

## Lesson 11 – Constant Values

Most macro's include numbers in the source code. These numbers are referred to as constants because once the source code is compiled, their value does not change. Constants may form part of an `IF THEN` command or they may be part of a mathematical equation. It is quite common to use the same constant value several times in different parts of the macro. Lets look at the following macro:

```
MAIN_MACRO:
  IF &TIMER1 > 8 * 60 * 60 * 10 THEN //8 hour timeout
    WRITE "          Maximum runtime exceeded          "
  ELSIF &TIMER2 > 24 * 60 * 60 * 10 THEN //24 hour
    //timeout
    WRITE "          Shut down meter          "
  ENDIF
END
```

In this macro, the timeouts for `&TIMER1` and `&TIMER2` are measured in hours, but because these timers count in 0.1 second steps, an 8 hour time period is shown as 8 (hrs) x 60 (mins) x 60 (secs) x 10 (0.1 secs). Now let's assume you want to test the macro. Of course you don't want to wait for these long timeouts. So you could change both timeout values for testing and then change them back to the correct values after you have finished your testing. That is fine if you only have two values to change. However, imagine that you use the same timers many times over in your macro. Changing them all is very laborious and opens up the possibility of missing one and creating a bug in the operation of your macro which will not be detected during testing!

User defined constants allow you to associate a constant number value (integer or floating point) with a name which can be used in its place. The name has no prefix and can be any combination of letters, numbers, and underscores `_`, provided the first character in the name is always a letter. The definition can be inside or outside a macro.

So let's see what we have to change for testing:

```
Line 2 // CONST Time_unit = 60 * 60 * 10 // 1 hour
CONST Time_unit = 60 * 10 // 1 minute - only for
// testing
Line 4 CONST MaxRunTime = 8 * Time_unit
Line 5 CONST MaxOperationTime = 24 * Time_unit

MAIN_MACRO:
IF &TIMER1 > MaxRunTime THEN //8 hour timeout
WRITE " Maximum runtime exceeded "
ELSIF &TIMER2 > MaxOperationTime THEN //24 hour
//timeout
WRITE " Shut down meter "
ENDIF
END
```

User defined constants can be used just like the number value they are associated with, even in calculations. In the macro above you will see in line 2 the constant `Time_unit` is defined as  $60 \times 10$  (or 1 minute) for testing purposes only. Then in lines 4 and 5 the constants `MaxRunTime` and `MaxOperationTime` both use the constant `Time_unit` as part of their calculations.

When you have finished your testing, just remove the `‘//’` marks at the beginning of line 1 and delete (or comment out) line 2. Re-compile the macro and now you can be sure that all references in your macro to these times will be correct.



If you define a constant as the result of a calculation the value will be calculated only once at compile time.

Constants can also be used to make your code more readable. In Lesson 21, I will show you how to use constants to define different operating states of a macro instead of just using numbers.



## Lesson 12 – Basic Maths Operators and Number Formats

Most of the macros we have written up until now have been designed to display text messages on the display when a certain condition is satisfied. Another powerful feature of the macro engine is the ability to apply a mathematical equation to your input data and display a more meaningful result. You can use any of the following basic maths operators in your macro (You can also use higher maths operators, but we'll discuss these later in Lesson 31):

### Operators

- + Addition
- Subtraction
- \* Multiplication
- / Division
- ( ) Parentheses
- ^ Power of

All of these operators require two operands. For example, I could write a macro like this:

```
MAIN_MACRO:
#COMPENSATION = (&CH2 + &CH3) * 0.1234

IF #COMPENSATION < 10000 THEN
    &RESULT = &CH1 * (#COMPENSATION - 10000)/10000
ELSE
    &RESULT = &CH1 ^ #COMPENSATION
ENDIF
END
```

In line 2, I have defined the variable `COMPENSATION` to be equal to the sum of `&CH2` plus `&CH3`, times 0.1234. The parenthesis tells the compiler to add `&CH2` and `&CH3` first, and then multiply the sum by 0.1234. Up to 10 levels of parenthesis can be used, depending on the complexity of the equations within the parenthesis.

The line under **ELSE** makes `&RESULT` equal to `&CH1` to the power of the variable `#COMPENSATION`.

Notice also how I have multiplied the sum of `&CH2 + &CH3` by a floating point number (0.1234) in line 2, but I have defined `#COMPENSATION` as an integer with the '#' symbol. This is quite acceptable, as the macro engine automatically converts the result from a floating point format to an integer format (fixed point) before saving it in the variable `#COMPENSATION`. Of course, in doing this, any fractional part of the result is lost, but for our macro, this is not important. If I wanted to retain the fractional part as well, I could have written:

Line 2

```

MAIN_MACRO:
%COMPENSATION = (&CH2 + &CH3) * 0.1234

IF %COMPENSATION < 10000 THEN
    &RESULT = &CH1 * (%COMPENSATION - 10000)/10000
ELSE
    &RESULT = &CH1 ^ %COMPENSATION
ENDIF
END

```

### Numeric Constants

In the TDS, numeric constants can be entered in several different ways. A number by itself, or with a decimal point, will be interpreted as a decimal number. A number with a '0x' in front of it will be interpreted as a base 16 hexadecimal number. Numbers can also be entered with an exponent. Here are some examples.

56	Decimal integer
0153	Octal (base 8)
0x3C	Hexadecimal (base 16)
0xa6	Hexadecimal (base 16)
1.234	Decimal floating point
0.03693E12	Decimal with exponent
1.572e-8	Decimal with exponent

Decimal numbers that are integers can be in the range of -2147483648 to 2147483647. Decimal numbers that contain a fractional part can be in the range of +/-1.175494E-38 to +/-3.402823E+38.

Hexadecimal numbers can be in the range of 0x0 to 0xFFFFFFFF.

# Intermediate Lessons

# 3

---

Lesson 13 - The EDIT Mode . . . . .	.2
Lesson 14 - Non-Volatile User Memory . . . . .	.7
Lesson 15 - Redefining Register Names . . . . .	.10
Lesson 16 - The Customer ID Macro . . . . .	.12
Lesson 17 - Function Button Macros . . . . .	.14
Lesson 18 - Text Strings . . . . .	.17
Lesson 19 - Text String Arrays . . . . .	.21
Lesson 20 - Edit String Arrays . . . . .	.25
Lesson 21 - The SELECT CASE Statement . . . . .	.27
Lesson 22 - The GOSUB Command . . . . .	.31
Lesson 23 - The REPEAT UNTIL Loop . . . . .	.34
Lesson 24 - The FOR NEXT Loop . . . . .	.36
Lesson 25 - Expressions in IF ELSIF Commands . . . . .	.38
Lesson 26 - Number Arrays and Register Arrays . . . . .	.39
Lesson 27 - Redefining Registers as Bit Arrays . . . . .	.44
Lesson 28 - Data Source Registers . . . . .	.46
Lesson 29 - Presetting Registers . . . . .	.49

## Lesson 13 – The EDIT Mode

So far, all of the macros we have looked at have used registers or flags that are updated automatically by the meter without any user intervention. Now let's look at an application that requires the user to adjust a parameter via the front panel buttons of the meter.

Let's suppose that our meter is measuring a variety of gases. For each type of gas, the user must enter a compensation factor from the front panel buttons. For this type of application, we need to be able to do the following:

- 1) Press a button to enter an editing mode.
- 2) Display a parameter to be edited.
- 3) Display some text to describe what the parameter is.
- 4) Use the UP and DOWN buttons to adjust the parameter.
- 5) Limit the adjustment range of the parameter being edited.
- 6) Press a button to store the edited value and return to the normal display when we have finished editing.

We could probably write a macro to do all of this in the `MAIN_MACRO`, but it would become fairly lengthy and quite complex. To make things easier, the TDS includes a special mode called the **edit mode** that allows many of the above functions to be handled automatically by the meter. Let's write a macro and see how the edit mode works:

**Macro starts on next page .....**

```
RESET_MACRO:
#SENSOR_TYPE = 1          //at power up always start
                          //with sensor 1

END

MAIN_MACRO:
IF |PROG_BUTTON = ON THEN //if the Prog button has
                          //been pressed

    IF &STATE = 0 THEN
        &STATE=1          //used to keep track of
                          //operational state

        EDIT #SENSOR_TYPE

        &EDIT_MAX = 4      //maximum value = 4
        &EDIT_MIN = 1      //minimum value = 1
        &EDIT_DEF = 1      //default value = 1
                          //(Up & Down pressed together)

        WRITE "Sensor"    // this flashes on the display
    ENDIF
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT #SENSOR_TYPE
    // &STATE = 0
ENDIF
END
```

In the first 3 lines I have used the `RESET_MACRO` to define a variable called `#SENSOR_TYPE`, which we will use to store the currently selected sensor type. When the meter is switched on, we'll set the sensor type to a default value of 1.

The `MAIN_MACRO` continuously checks the bit flag `|PROG_BUTTON` to see when the PROGRAM button has been pressed. If the PROGRAM button is pressed, the macro then checks a predefined register called `&STATE`. This is used to keep track of which operational state our macro is currently in.

Although our simple macro only has two states (0 = normal display, 1 = edit sensor), it is not uncommon for macros to have many more.

If the program button is pressed while the operational state is zero (i.e. normal display mode), then `&STATE` is changed to 1, to signify that the meter is now in the edit sensor mode.



By the way, you can use any number you like for the edit sensor state, but you should always use zero for the normal operating state of the meter.

### Enter the Edit Mode

On the next line, you will see the command `EDIT #SENSOR_TYPE`. The word `EDIT` is the command that puts the meter into the special edit mode. The variable or pre-defined register name that follows the edit command tells the macro what to edit. In this case we have told the macro to edit the variable `#SENSOR_TYPE`, so it will automatically load the current value of the user defined variable `#SENSOR_TYPE` into its edit buffer. The edit command can be used with any user defined register or pre-defined register from the tree view list in the pre-defined variables window, or constant value. For example, I could have written:

```
EDIT &CH1
```

The next 3 lines below the `EDIT` command are used to specify the editing range that you want. The pre-defined register `&EDIT_MAX` is loaded with the maximum value that you want to allow for the editable range. The pre-defined register `&EDIT_MIN` is loaded with the minimum value that you want to allow for the editable range. The pre-defined register `&EDIT_DEF` is loaded with a default value. The edit value will be set to the default value when both the UP and DOWN buttons are pressed together in the edit mode. This provides a quick method of getting to a normal value when in the edit mode.

The final command in the `MAIN_MACRO` is `WRITE "Sensor"`. This causes the text 'Sensor' to flash (or scroll) alternately with the edit value.

That's all there is to getting into the edit mode! The meter will be flashing between 'Sensor' and '1' and it will keep flashing until you change the value by pressing the UP or DOWN button. Remember that while this is happening, the meter is operating in a special edit mode, not it's normal mode of operation.

You will notice that our macro doesn't finish with the `MAIN_MACRO`, but includes another type of macro area called the `EDIT_MACRO`. The `EDIT_MACRO` is only used while the meter is operating in edit mode, and is called once, each time the `PROGRAM` button is pressed.

In this case, the edit macro checks to see what operating state we are in. If `&STATE` is equal to 1 (i.e. edit sensor type), then the command `EXIT_EDIT #SENSOR_TYPE` is executed. The `EXIT_EDIT` command is used to end the edit mode and return to the normal operating mode of the meter. At the same time, it will store the newly edited data value into the register or variable name specified with the command. So in our macro, it will store the edited value back into `#SENSOR_TYPE`. The `EXIT_EDIT` command will also set `&STATE` back to zero (normal operating mode), so we don't need a separate line that reads `&STATE = 0`.

That's it! Once in edit mode, the meter handles most of your editing requirements automatically. I think you should download this one and try it out!



Well, what happened? You should find that it worked quite well, up until the point at which you pressed the `PROGRAM` button to save your edited value and return to the normal display mode. You probably had great difficulty exiting the edit mode, and could only do it if you pressed the `Program` button really fast!

This is because the `MAIN_MACRO` is operating so fast, that by the time you have lifted your finger off the `PROGRAM` button to exit the edit mode, the `MAIN_MACRO` sees the `PROGRAM` button still pressed and goes straight back into it.

There is a very simple fix for this. The meter contains two predefined registers called `&TIMER1` and `&TIMER2` that you can use in your macro. Both of these registers count up 1 count every 0.1 seconds. You can read or write to these timers whatever you like. Let's re-write our macro using a timer to solve our problem.

**Macro starts on next page .....**

```

RESET_MACRO:
#SENSOR_TYPE = 1          //at power up always start
                          //with sensor 1

END

MAIN_MACRO:
IF |PROG_BUTTON = ON THEN //if the Prog button has
                          //been pressed

    IF &STATE = 0 THEN
        IF &TIMER1 > 10 THEN //if timer1 > 1 second
                              //then

            &STATE=1          //used to keep track of
                              //operational state

            EDIT #SENSOR_TYPE
            &EDIT_MAX = 4    // maximum value = 4
            &EDIT_MIN = 1    // minimum value = 1
            &EDIT_DEF = 1    // default value = 1
                              //(Up & Down pressed together)

            WRITE "Sensor"  // this flashes on the display
        ENDIF
    ENDIF
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT #SENSOR_TYPE
//&STATE = 0
    &TIMER1 = 0            //set timer1 to zero to stop
                          //going back into edit mode

ENDIF
END

```

That's better! All I have done is to set &TIMER1 to zero when I exit the edit mode, and check that &TIMER1 is greater than 1 second (10 x 0.1) before entering the edit mode. This means that the macro won't re-enter the edit mode until 1 second after it exits the edit mode.

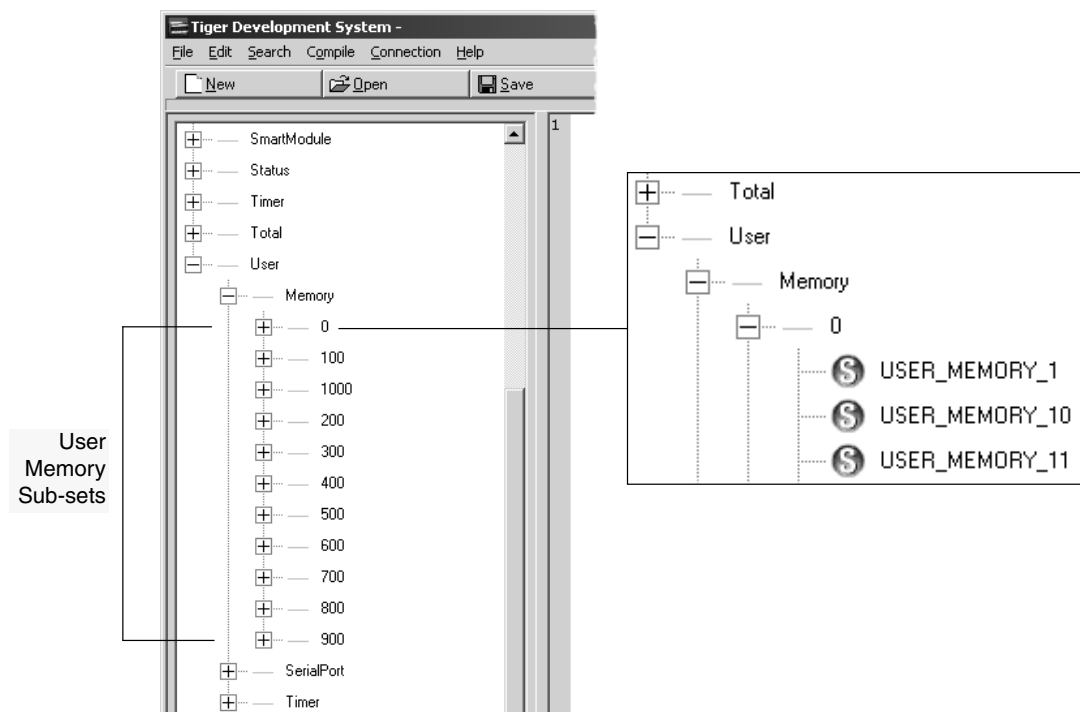


## Lesson 14 – Non-Volatile User Memory

In our previous macro, we defined a variable called `#SENSOR_TYPE` that we edited from the front panel. All user defined variables are stored in RAM, which means that their contents are lost when the meter is switched off. In our previous macro we used the `RESET_MACRO` to set `#SENSOR_TYPE` to 1 each time the meter is turned on. This might be OK for some applications, but for many this would be a nuisance.

To solve this problem, the meter includes a bank of pre-defined registers that retain their contents, even when the meter is switched off (i.e. non-volatile registers). If you look down the tree view list of pre-defined variables, you will see **User** at the bottom of the tree. Click on the + sign beside User. This opens a sub-set list with **Memory** at the top of the list. Click on the + sign beside Memory to get a list of registers. There are 1024 of these registers available for data storage, setup information, or tables. Each register is a 16-bit signed register. This means it can hold a value from -32768 to 32767.

Let's re-write the previous macro to use a non-volatile user memory to store the sensor type:



Macro starts on next page .....

```

MAIN_MACRO:
IF |PROG_BUTTON = ON THEN //if the Prog button
                               //has been pressed

    IF &STATE = 0 THEN
        IF &TIMER1 > 10 THEN //if timer1 > 1 second
                               //then
            &STATE=1           //used to keep track of
                               //operational state

            EDIT &USER_MEMORY_1 //user memory 1 =
                               //sensor type

            &EDIT_MAX = 4      //maximum value = 4
            &EDIT_MIN = 1      //minimum value = 1
            &EDIT_DEF = 1      //default value = 1
                               //(Up & Down pressed together)

            WRITE "Sensor" // this flashes on the display
        ENDIF
    ENDIF
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &USER_MEMORY_1 //exit edit mode, save
                               //sensor type in user memory 1
//&STATE = 0
    &TIMER1 = 0                //set timer1 to zero to stop
                               //going back into edit mode

ENDIF
END

```

Now we don't need to use the `RESET_MACRO` to initialize the value of sensor type because the last value is retained in `&USER_MEMORY_1`.



One important point to note about the user memories on the Tiger 320 is that there is a limited number of times that you can write to these registers. Because they are stored in Electrically Erasable memory (E<sup>2</sup>), you must not exceed 100,000 write cycles. That all sounds quite technical, but what it really means is that you must not update a user memory every time the `MAIN_MACRO` is called. If you did, you would find that after a short period of time, the user memory might not reliably store your data.

You can read a user memory as many times as you like. So that means you could use a user memory as part of a calculation or as part of a lookup table, as many times as you like. In the macro above, `&USER_MEMORY_1` is updated in the line `EXIT_EDIT &USER_MEMORY_1`. This is only used when the sensor type is changed, so the user of this macro could change the sensor type 100,000 times before they would have a problem!



In the Tiger 380 the user memories are stored in RAM and backed up in FRAM. Therefore the number of writes is not limited in the 380 series.

Please refer to Lesson 43 for further changes regarding the user memories in the 380.

## Lesson 15 – Redefining Register Names

Sometimes it may be more convenient to give pre-defined registers different names that relate directly to an application. For example, channel 1 of our meter might be measuring pressure, channel 2 might be measuring flow rate, and channel 3 temperature. Or maybe you're using a large number of user memories and it's hard to remember what the function of each one is.

The TDS allows you to re-assign new names to pre-defined registers and bit flags. Let's take the previous macro from Lesson 14 and rewrite it with more meaningful register names. Here's how it works:

Line 2

```

REG &SENSOR_TYPE = &USER_MEMORY_1
BIT |SELECT = |PROG_BUTTON

MAIN_MACRO:
IF |SELECT = ON THEN           //if the Select button
                                //has been pressed

    IF &STATE = 0 THEN
        IF &TIMER1 > 10 THEN    //if timer1 > 1 second
                                //then
            &STATE=1              //used to keep track of
                                //operational state
            EDIT &SENSOR_TYPE    //user memory 1 = sensor
                                //type
            &EDIT_MAX = 4         //maximum value = 4
            &EDIT_MIN = 1        //minimum value = 1
            &EDIT_DEF = 1        //default value = 1
                                //(when Up & Down are pressed together)
            WRITE "Sensor"      //this flashes on the display
        ENDIF
    ENDIF
ENDIF
END

```

Macro continued on next page .....

..... From previous page

```
EDIT_MACRO:
//This macro is run each time the select button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &SENSOR_TYPE //exit edit mode, save
                                //sensor type in user memory 1
//&STATE = 0
    &TIMER1 = 0 //set timer1 to zero to stop
                //going back into edit mode
ENDIF
END
```

In the first line, you will see the word **REG**, followed by `&SENSOR_TYPE = &USER_MEMORY_1`. This tells the compiler to substitute the register `&USER_MEMORY_1`, whenever it sees the register called `&SENSOR_TYPE`. Similarly, on line 2 the word **BIT** followed by `|SELECT = |PROG_BUTTON`, tells the compiler to substitute the bit flag `|PROG_BUTTON`, whenever it sees the flag `|SELECT`.



You must position the re-definitions before the point in your program where you use the new register names, otherwise the compiler won't be able to recognise them.

## Lesson 16 – The Customer ID Macro

Once you get into the swing of writing macros, you might have hundreds of Texmate meters around (we're hoping!), running specialised macros for specific tasks. Occasionally you may want to change or upgrade a macro to fix a problem or simply improve a process.



The question you will probably be asking yourself at that point is, 'Which macro version is this meter running?'

The TDS has a simple solution to this called the `CUSTOMER_ID_MACRO`. This is just another macro section, similar to the `MAIN_MACRO`, the `RESET_MACRO` and the `EDIT_MACRO`. Again, the only difference is when this macro is run (or called) by the meter.

The Tiger Series meter has a built-in model and software version number display function that is accessed by pressing the PROGRAM, UP and DOWN buttons at the same time. This causes the display to flash the model number and software version number of the meter for a few seconds. Then, it will try to execute the `CUSTOMER_ID_MACRO`.

Normally, with no macro loaded, the meter displays the model number and software version number for a few seconds, and then returns to the normal operating mode. If the meter has a macro loaded that includes a `CUSTOMER_ID_MACRO`, it executes this macro before returning to the normal operating display.

Let's have a look at the first macro we wrote and include a customer ID macro:

```
CUSTOMER_ID_MACRO:
WRITE "   ___This is version 1.0 of my first " + \
      "macro      "
END

MAIN_MACRO:
WRITE "   ___Hello World      " //Scroll Hello
      //World across the display
END
```

If you run this macro in your meter, it should just scroll the message " \_\_\_\_Hello World \_\_\_\_ " until you press the PROGRAM, UP, and DOWN buttons at the same time. It should then flash between the model and software version number for a few seconds. After that, it should start scrolling " \_\_\_\_This is version 1.0 of my first macro \_\_\_\_ " and then return to the normal display.

You could do anything you want in the CUSTOMER\_ID\_MACRO, but it's primary purpose is to identify the currently loaded macro, so normally it only contains a message that is scrolled across the display.



We suggest that you make it a practice to start each macro you write by first writing a CUSTOMER\_ID\_MACRO, at the top of your program. To simplify things, I haven't included a CUSTOMER\_ID\_MACRO in all of these tutorials, but I recommend that you do when you start writing your own macros!

## Lesson 17 – Function Button Macros

Some meter applications may require several different parameters to be changed by the operator from the front panel buttons of the meter. This could be done by pressing the PROGRAM button each time and sequencing through the parameters. However, this can be confusing for the operator.

To solve this problem, Tiger Series meters are available in a variety of display options, some of which include extra function buttons. Up to three function buttons are currently available on some models of meter. These are labeled F1, F2, and F3. Apart from macro use, they have no other dedicated function in the meter.

Each function button has its own macro section, which is called each time the button is pressed. Let's take the macro we wrote in Lesson 15 and re-write it so that we use the F1 button to edit the sensor type:

**Macro starts on next page .....**



```
CUSTOMER_ID_MACRO:
WRITE "    ___Macro to select sensor type V1.0    "
END

REG &SENSOR_TYPE = &USER_MEMORY_1

F1_BUTTON_MACRO:
//*****
//The F1 button is now the select button to enter
//the edit mode
IF &STATE = 0 THEN
    &STATE=1 //used to keep track of
                //operational state
    EDIT &SENSOR_TYPE //user memory 1 = sensor type
    &EDIT_MAX = 4 //maximum value = 4
    &EDIT_MIN = 1 //minimum value = 1
    &EDIT_DEF = 1 //default value = 1
                //(when Up & Down are pressed together)
    WRITE "Sensor" //this flashes on the display
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &SENSOR_TYPE // exit edit mode, save
                            //sensor type in user memory 1
//&STATE = 0
ENDIF
END
```

## 3-16

### Intermediate Lessons

This looks very similar to the macro in Lesson 15, except that now we don't need a `MAIN_MACRO` and we don't need to test if the `PROGRAM` button is pressed. I have used the `F1_BUTTON_MACRO`, which is called each time the F1 button is pressed. Now we don't need to use a timer, as we did in Lesson 15, because we are using different buttons to enter and exit the edit mode.



Notice that this macro doesn't have a `MAIN_MACRO`. In most practical macros, you would probably still have a `MAIN_MACRO` to do some calculations, but this is not mandatory.

Note that the F1 button is only used to enter the editing mode. The `PROGRAM` button is still used to save the edited value and return to the normal operating state. This could be changed if required, but because this is all done automatically in the `EDIT_MACRO`, it is the simplest way of doing this.

Although I haven't used them in this macro, there is also an `F2_BUTTON_MACRO` and an `F3_BUTTON_MACRO`.

## Lesson 18 – Text Strings

In Lesson 1 we looked at the **WRITE** command, which allows a message (or text string) of up to 100 characters long to be scrolled across the display. When a **WRITE** command is executed, the message, enclosed in the quotation marks, is copied into a special section of memory, called the string buffer. Each time the **WRITE** command is executed, the new message is written into the string buffer, starting from the beginning of the buffer. Any previous messages are overwritten and lost and the new message starts scrolling from the beginning of the message.



On multiple display meters of the Tiger 320 series text messages can only appear on the default display (top display on the DI-503, bottom display on the DI-602 and DI-802). On dual display meters of the Tiger 380 series (DI-602 and DI-802) messages can be written to both displays. This will be discussed in Lesson 41.

Sometimes it is necessary for different parts of a message to change, depending on certain conditions. For short messages we could just write each message over again, with changes to part of the message, as we did in Lesson 4. For long messages, this is not very efficient and would use up large amounts of memory for each message.

The TDS includes an **APPEND** command that allows you to add more text onto the end of an existing message string. Here is an example of how it is used. I have taken the macro from Lesson 17 and added a **MAIN\_MACRO** to display the currently selected sensor type:

We have looked at most of this macro before in Lesson 17, so let's just concentrate on the **MAIN\_MACRO** section. The first **WRITE** command of the **MAIN\_MACRO** writes the text "`___Currently selected sensor type is - Sensor`" into the beginning of the string buffer. This text remains the same for all of the messages we

**Macro starts on next page .....**

```
CUSTOMER_ID_MACRO:
WRITE "    ___Macro to select sensor type V1.0      "
END

REG &SENSOR_TYPE = &USER_MEMORY_1

MAIN_MACRO:
//*****
//This is the main macro
IF &STATE = 0 THEN
    WRITE "    ___Currently selected sensor type " + \
        "is - Sensor "
    IF &SENSOR_TYPE = 1 THEN
        APPEND "1 (Butane)"
    ELSIF &SENSOR_TYPE = 2 THEN
        APPEND "2 (Oxygen)"
    ELSIF &SENSOR_TYPE = 3 THEN
        APPEND "3 (Hydrogen)"
    ELSE
        APPEND "4 (Nitrogen)"
    ENDIF

    APPEND ". Press F1 to select new sensor " + \
        "type.      "
ENDIF
END

F1_BUTTON_MACRO:
//*****
//The F1 button is now the select button to enter
//the edit mode
```

Macro continued on next page .....

..... From previous page

```
IF &STATE = 0 THEN
    &STATE=1          //used to keep track of
                    //operational state
    EDIT &SENSOR_TYPE //user memory 1 = sensor type
    &EDIT_MAX = 4     //maximum value = 4
    &EDIT_MIN = 1     //minimum value = 1
    &EDIT_DEF = 1     //default value = 1
                    //(when Up & Down are pressed together)
    WRITE ""
    WRITE "Sensor"   //this flashes on the display
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &SENSOR_TYPE //exit edit mode, save
                        //sensor type in user memory 1
//&STATE = 0
ENDIF
END
```

want to display, so it can all be put in the **WRITE** command string. However, different text is required next, depending on the value of **&SENSOR\_TYPE**. For each different value of **&SENSOR\_TYPE**, a different text string is added onto the end of the initial string with the **APPEND** command. Finally, after the sensor type has been added to the string, the **APPEND** command is used again to add more common text onto the end of the message.

So, if **&SENSOR\_TYPE** equals 1, then the following message would be displayed.

```
"    ___Currently selected sensor type is - Sensor 1
(Butane). Press F1 to select new sensor type.    "
```

You can append a message as many times as you like, provided that the total length of the entire message does not exceed 100 characters.

You may be asking yourself "If the new message starts from the beginning each time the **WRITE** command is executed, and the `MAIN_MACRO` is run 10 times a second, then how can this macro possibly have enough time to display the whole message?" Good question!



To avoid overwriting a previous message, the macro engine first checks that the existing message has finished scrolling before it writes a new one. If a message is still in progress when a **WRITE** command is executed, the macro simply stops execution at that point, exits the macro, and hands control back to the operating system of the meter. Any commands past that point are ignored.

This can be a problem in some cases, and careful thought should be given to where you place **WRITE** commands. You need to make sure that, if a **WRITE** command cannot be executed (because a previous message is still scrolling), the same section of code will be entered next time the macro is called.



Sometimes you might want to interrupt a message that is still scrolling, and display a new message. Maybe the new message is an urgent warning or maybe it is a response to a button being pressed. This is exactly what I have done in the `F1_BUTTON_MACRO`. To ensure that "Sensor" is displayed instantaneously when the F1 button is pressed, I have included an extra **WRITE** "" instruction before the **WRITE** "Sensor" instruction.

Placing the instruction **WRITE** "", with an empty string (i.e. "") before the message you want to display, stops any current messages and clears the string buffer. The next **WRITE** is executed straight away because the string buffer is now empty.

## Lesson 19 – Text String Arrays

In the previous lesson, we used the **APPEND** command to display different text for different sensor types. This works fine if we only have 4 sensor types, but what if we had 100 different sensor types. It would be a very long **IF THEN** command!

The TDS allows you to use something called a **text string array**, which is really just a group of individual messages, one after the other. Let's look at a simple macro that just displays the currently selected sensor type:

```
DIM A[ ] = [ " ", "1 (Butane)", "2 (Oxygen)", \
              "3 (Hydrogen)", "4 (Nitrogen)" ]

REG &SENSOR_TYPE = &USER_MEMORY_1

MAIN_MACRO:
IF &STATE = 0 THEN
  WRITE "    ___Currently selected sensor type " + \
        "is - Sensor "
  APPEND A[&SENSOR_TYPE]

  APPEND ". Press F1 to select new sensor " + \
        "type.          "
ENDIF
END
```

**Macro continued on next page .....**

..... From previous page

```

F1_BUTTON_MACRO:
IF &STATE = 0 THEN
    &STATE=1          //used to keep track of
                      //operational state

    EDIT &SENSOR_TYPE //user memory 1 = sensor type
    &EDIT_MAX = 4      //maximum value = 4
    &EDIT_MIN = 1      //minimum value = 1
    &EDIT_DEF = 1      //default value = 1
                      //(when Up & Down are pressed together)

    WRITE ""
    WRITE "Sensor" //this flashes on the display
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &SENSOR_TYPE //exit edit mode, save
                          //sensor type in user memory 1

    //&STATE = 0
ENDIF
END

```

### Dimension the Array

Functionally, this macro would do the same thing as the `MAIN_MACRO` in Lesson 18, but it looks quite different. In the first line you will notice the command `DIM A[ ] =`, followed by square brackets and then some text strings, separated by commas. The `DIM A[ ] =` command is used to dimension the array. Or, in other words, to tell the compiler how many individual strings there are in the array, and what is in each string.

The `A[ ]` tells the compiler that the following group of strings belongs to the text string array called `A`. It's possible to have several different text string arrays in one program. You might, for example, have another array defined as `DIM B[ ] =` and yet another as `DIM C[ ] =`.



In the main macro, you will notice the line **APPEND** A[&SENSOR\_TYPE]. As we discussed in Lesson 18, the **APPEND** command just adds a text string on to the end of an existing string. The difference here is that the actual string is not specified directly after the command. Instead, A[&SENSOR\_TYPE] acts as a pointer to the text string that we want. The A tells the compiler that the string we want is part of the text string array A.

The part inside the square brackets points to the individual string number in the array. If, for example, I wrote **APPEND** A[4] it would use the last string in array A, which is "4 (Nitrogen)". In the previous macro I have put register &SENSOR\_TYPE inside the square brackets instead of a number, but the effect is the same. The macro engine now gets the number stored in &SENSOR\_TYPE and uses this as the pointer to the string in array A.



Hold on a minute! There are actually five strings in array A. The first one is a string with only a space in it (i.e. " "). That's because in the TDS strings in an array are always numbered from zero. So if I write **APPEND** A[4], it actually points to the fifth string in the array, because **APPEND** A[0] points to the first string. I could have written the macro so that &SENSOR\_TYPE ranges from 0 to 3 instead of 1 to 4, but it's just as easy to insert a small string that doesn't do anything.

Text string arrays can be used with the **WRITE** command and the **APPEND** command. All of the following examples are valid uses of text string arrays.

```
WRITE a[1]           //get string #2 from string array a
WRITE ABC[&CH3]     //get string pointed to by CH3 from
                    //string array ABC

WRITE MSG[#MY_VARIABLE] // get string pointed to by
                    //MY_VARIABLE from string
                    //array MSG

APPEND a[1]         //concatenate string #2 from string
                    //array a
APPEND ABC[&CH3]   //concatenate string pointed to by
                    //CH3 from string array ABC
APPEND MSG[#MY_VARIABLE] //concatenate string pointed
                    //to by MY_VARIABLE
                    //from string array MSG
```



Remember that each string in the array can be up to 100 characters long, provided that, in the case of the **APPEND** command, the total message length does not exceed 100 characters.

An array can have any number of strings in it, but it will be limited by the amount of macro memory available.

If you are using a register or a variable, to point to a string in an array, you must make sure that you limit the range of the variable to match the size of the array! Neither the compiler nor the macro engine is capable of doing this for you! If a register or variable points to a string outside of the array, it will start loading the string buffer with whatever it finds at that destination. The result could be something along the lines of:

```
"!(h-#, ; ; .dyg$32&89h% () &3@+k(8b^%43fI9*~"
```

or something equally as meaningless!

## Lesson 20 – Edit String Arrays

Now that you know how to use string arrays with the **WRITE** and the **APPEND** command, we'll look at a slightly different use of string arrays in the edit mode.

Back in Lesson 13, we looked at the edit mode and wrote a macro to allow you to select a sensor type, from 1 to 4, via the front panel buttons. In the edit mode, the display flashed between **Sensor** and a number from **1** to **4**. This works well provided you know what sensor #1 is. It would be more helpful if the numbers could be replaced by text, so that you can see exactly what you are selecting.

TDS to the rescue again! The command **EDIT\_TEXT** is used in the TDS to do exactly this. Take a look at the following macro. This macro is used to select thermocouple types for temperature measurement:

```
DIM A[] = ["J type","K type","R type","S type",\  
           "T type","B type","N type"]  
  
RESET_MACRO:  
#SENSOR_TYPE=0  
END  
  
F1_BUTTON_MACRO:  
//Called by the operating system when the F1 button  
//is pressed  
IF &STATE = 0 THEN  
    EDIT #SENSOR_TYPE  
    &EDIT_MAX=6  
    &EDIT_MIN=0  
    &EDIT_DEF=0  
    WRITE ""  
    WRITE "Sensor"  
    EDIT_TEXT A[] //this line selects DIM A[] as the  
                  //string array  
    &STATE=1  
ENDIF  
END
```

Macro continued on next page .....

..... From previous page

```
EDIT_MACRO:
//Called by the operating system when Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT #SENSOR_TYPE
//&STATE = 0
ENDIF
END
```

In the first line we have defined a string array called `A` that has seven strings in it. Near the end of the `F1_BUTTON_MACRO` you will find a line that reads `EDIT_TEXT A[ ]`. This looks very similar to the string array commands used in the previous lesson, except that there is no number or register specified in between the square brackets! That's because the `EDIT_TEXT` command always uses the edit value to point to the string.

If the edit value equals zero, the display flashes between **Sensor** and **J type**. As the UP or DOWN buttons are pressed to change the sensor type, the text changes from **J type** to **K type** and so on.

The `EDIT_TEXT` command can only be used in the edit mode. Each time the edit mode is entered, by using the command `EDIT`, the edit value is displayed as numerals by default. If the `EDIT_TEXT` command is then issued, strings are displayed instead of numerals. So you must make sure that you are already in the edit mode before you use the `EDIT_TEXT` command (see Lesson 13).

If, for some reason, you want to stop displaying text and display the edit value in numerals again, you can do so by using the command `EDIT_NUMERIC` to return to the default condition.

## Lesson 21 – The SELECT CASE Statement

All of the macros we have looked at so far have used the **IF THEN** or **ELSIF** command to conditionally test registers and variables. Another useful command is the **SELECT CASE** statement. It is particularly useful, and more efficient, for programs that test the same register or variable for many different values. Let's have a look at how it is used.

```
REG &SENSOR_TYPE = &USER_MEMORY_1
REG &SENSOR_GAIN = &USER_MEMORY_2
REG &SENSOR_MODE = &USER_MEMORY_3
REG &SENSOR_CAL = &USER_MEMORY_4

F1_BUTTON_MACRO:
IF &STATE = 0 THEN
    &STATE=1                //used to keep track of
                            //operational state

    EDIT &SENSOR_TYPE      //user memory 1 = sensor type
    &EDIT_MAX = 4           //maximum value = 4
    &EDIT_MIN = 1          //minimum value = 1
    &EDIT_DEF = 1          //default value = 1 (Up & Down
                            //pressed together)

    WRITE "Sensor"        //this flashes on the display
ENDIF
END

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
SELECT &STATE            //test &STATE
    CASE 1:                //if STATE = 1 then
        EXIT_EDIT &SENSOR_TYPE //exit edit mode, save
                                //sensor type in user memory 1
        EDIT &SENSOR_GAIN //user memory 2 = sensor gain
```

Macro continued on next page .....

..... From previous page

```

&EDIT_MAX = 100 //maximum value = 100
&EDIT_MIN = 0 //minimum value = 0
&EDIT_DEF = 0 //default value = 0
// (Up & Down pressed together)

&STATE = 2
WRITE " Gain" //this flashes on the display

CASE 2: //if STATE = 2 then
EXIT_EDIT &SENSOR_GAIN //exit edit mode, save
//sensor gain in user memory 2
EDIT &SENSOR_MODE //user memory 3 = sensor mode
&EDIT_MAX = 10 //maximum value = 10
&EDIT_MIN = 0 //minimum value = 0
&EDIT_DEF = 0 //default value = 0
// (Up & Down pressed together)

&STATE = 3
WRITE " Mode" //this flashes on the display

CASE 3: //if STATE = 3 then
EXIT_EDIT &SENSOR_MODE //exit edit mode, save
//sensor mode in user memory 3
EDIT &SENSOR_CAL //user memory 4 = sensor
//calibration
&EDIT_MAX = 255 //maximum value = 255
&EDIT_MIN = 0 //minimum value = 0
&EDIT_DEF = 0 //default value = 0
// (Up & Down pressed together)

&STATE = 4
WRITE " Cal" //this flashes on the display

CASE 4: //if STATE = 4 then
EXIT_EDIT &SENSOR_CAL //exit edit mode, save
//sensor cal in user memory 4
// &STATE = 0
DEFAULT:
&STATE = 0
ENDSEL
END

```

This may look a bit daunting but it's actually quite simple. Let's just look at the `EDIT_MACRO`, as this is where `SELECT` and `CASE` statements are used.

The very first line of the `EDIT_MACRO` has the command `SELECT &STATE`. This means that we are going to test the pre-defined register `&STATE` for each of the different cases shown between the `SELECT` and `ENDSEL` instructions. The `ENDSEL` instruction specifies where the `SELECT` command finishes in much the same way as the `ENDIF` instruction sets the end of an `IF THEN ELSIF` group of tests.

The next line reads `CASE 1:`, which means that the macro tests `&STATE` to see if it equals 1. If `&STATE` equals 1, all of the commands under `CASE 1:` are executed, until the next `CASE`, `DEFAULT`, or `ENDSEL` instruction is encountered. It then starts executing the next line of code after the `ENDSEL` instruction.

If `&STATE` is not equal to 1, it drops down to the next `CASE` instruction, which is `CASE 2:`. This means the macro tests `&STATE` to see if it equals 2 and so on.

Each `CASE` statement is tested until one of them is found to be true. If none of the `CASE` statements are true, the `DEFAULT` case can be used to execute some task for all other values of `&STATE` that have not been tested in the case statements above.



The `DEFAULT` case is optional, but if it is used, it must be used last, after all the other `CASE` statements. If all of the cases tested are false, and the `DEFAULT` case is not used, then the next instruction after the `ENDSEL` command is executed.

In some programs you might want to execute the same commands for several different cases. You can do this with one `CASE` statement as follows:

**Macro starts on next page .....**

```

SELECT &STATE          //test &STATE
  CASE 1, 5, 6, 10:    //if STATE = 1,5,6 or 10 then
    WRITE " Gain"     //this flashes Gain on the
                      //display

  CASE 2:              //if STATE = 2 then
    WRITE " Mode"     //this flashes Mode on the
                      //display

  CASE 3:              //if STATE = 3 then
    WRITE "  Cal"     //this flashes Cal on the display

  DEFAULT:
    &STATE = 0

ENDSEL

```



In Lesson 11, we looked at the use of constant names instead of numbers. It is often helpful to replace case numbers in select case statements with more meaningful names. For example, we could write the following:

```

CONST OPERATE = 0
CONST EDIT_GAIN = 1
CONST EDIT_MODE = 2
CONST EDIT_CAL = 3

SELECT &STATE
  CASE EDIT_GAIN
    WRITE " Gain"

  CASE EDIT_MODE
    WRITE " Mode"

  CASE EDIT_CAL
    WRITE "  Cal"

  DEFAULT:
    &STATE = OPERATE

ENDSEL

```



## Lesson 22 - The GOSUB Command

Often you will find that the same block of instructions need to be executed several times over, from different parts of your program. You could copy these blocks into your code in the appropriate places, but this is not very efficient. It also means if you need to change something in the block, you will need to change it everywhere you have copied it as well.

The TDS has a command called `GOSUB` that allows you to write the block once, and then call it from anywhere in the program as many times as you want. This block of code is normally referred to as a sub-routine, hence the name `GOSUB` (i.e. GO - to - SUBroutine). The following example shows how it works:

```
RESET_MACRO:
#TEMP = 0
END

MAIN_MACRO:
//*****
//
//                                MAIN_MACRO
#TEMP = &CH1
GOSUB CHECK_RANGE
&CH1 = #TEMP

#TEMP = &CH2
GOSUB CHECK_RANGE
&CH2 = #TEMP

#TEMP = &CH3
GOSUB CHECK_RANGE
&CH3 = #TEMP
```

Line 9

Macro continued on next page .....

..... From previous page

```

#TEMP = &CH4
GOSUB CHECK_RANGE
&CH4 = #TEMP

END

//*****
//          Subroutine to check range
CHECK_RANGE:
Line 30  IF #TEMP > 10000 THEN
Line 31      #TEMP = 10000
Line 32  ELSIF #TEMP < 0 THEN
Line 33      #TEMP = 0
Line 34  ENDIF
RETURN

```

If you look at the bottom of the macro, you will find the subroutine and a line that reads `CHECK_RANGE:`. The `:` after the label `CHECK_RANGE` tells the compiler that this is an address label. The compiler now knows that any reference to `CHECK_RANGE` means that it has to go to this point in your program.

The next five lines (lines 30 to 34) form a simple test to check the range of a variable called `#TEMP` and limit its range. The last line of the subroutine contains the command `RETURN`. This signals the end of the subroutine and tells the macro to jump back to the point at which the subroutine was called and start executing the next instruction from there.

In the second line of the `MAIN_MACRO` (line 9) you will see the command `GOSUB CHECK_RANGE`. This causes the macro to jump, from where it is, to the point in the program where the label `CHECK_RANGE` is placed, and then start executing the next instruction from there. Before the macro jumps to `CHECK_RANGE`, it first saves the place where it is so that it knows where to come back to when it executes the `RETURN` instruction.

### Nested Subroutines

A subroutine can be called as many times as you like in your program. You can also have as many subroutines as you like in one program. You can even call one subroutine from inside another subroutine. In the TDS this is called a nested subroutine. The only restriction is that you cannot use nested subroutines of more than 4 levels deep, but in practice this is seldom a problem.



You should never place a subroutine inside another macro section. Subroutines can be placed at the beginning or end of your program, or in between other macro sections. The important thing to remember is that every subroutine must have a **RETURN** command at the end.

## Lesson 23 – The REPEAT UNTIL Loop

Sometimes it is necessary to repeat a block of instructions a number of times, but the number of repeats is not constant. To do this, the TDS includes the **REPEAT UNTIL** loop, which repeats a block of instructions until a certain condition is true.

To show this in an example, let's assume we need to calculate the result of channel 1 to the power of channel 2. (In Lesson 12 you will find that there is a special instruction that does  $X^Y$ , but here I will do it the long way).

Line 2

Line 3

```
MAIN_MACRO :
#TEMP = &CH2
&RESULT = &CH1
REPEAT
  IF &RESULT > 10000 THEN
    WRITE "    ___Over Range    "
    #TEMP = 1
  ELSE
    &RESULT = &RESULT * &CH1
    #TEMP = #TEMP - 1
  ENDIF
UNTIL #TEMP <= 1
END
```

In line 2, a variable called #TEMP is loaded with the value of &CH2. In line 3, the register &RESULT is loaded with the value of &CH1. The next line contains the instruction **REPEAT**. This just tells the compiler where to start repeating instructions from. The next few lines test the value of &RESULT and display an overrange warning if it is too large to be multiplied by &CH1.

If the value of &RESULT is inside the acceptable range, &RESULT is multiplied by &CH1, and the variable #TEMP is decremented by 1 count. A few lines further down is the instruction **UNTIL** #TEMP <= 1. The **UNTIL** instruction tells the compiler that it needs to test the following condition and jump back to the **REPEAT** instruction if it is false.

In this case, the macro checks the value of #TEMP and jumps back to the **REPEAT** instruction if it is greater than 1. If it is less than or equal to 1, it exits the **REPEAT UNTIL** loop and starts executing the next line after the **UNTIL** instruction.



Notice that each time the value of &RESULT is multiplied by &CH1, the value of #TEMP is decremented, so the value of #TEMP always starts to decrease towards 1. If the value of &RESULT is too large, then #TEMP is also set to 1. This means our program always exits the loop, regardless of the starting values of &CH1 and &CH2.

This is a very important point to note about **REPEAT UNTIL** loops. You must be careful not to allow the program to get into an endless loop, or a loop that is too long. If you do, the overall performance of the meter becomes slow and erratic. It may even appear to freeze or lock up if your macro enters an endless loop.

Another point to note with **REPEAT UNTIL** loops is that the loop is always executed at least once, because the **UNTIL** condition is only executed at the end of the loop. In the next lesson we will look at the **FOR NEXT** loop which checks the conditions at the beginning of the loop.

## Lesson 24 – The FOR NEXT Loop

Another variation on the **REPEAT UNTIL** loop is the **FOR NEXT** loop. Once again it is used to repeat a block of instructions a number of times, but there are some subtle differences. Let's have a look at an example:

```
Line 3  MAIN_MACRO:
        &RESULT = &CH1
        FOR #TEMP = &CH2 TO 1 STEP -1
            IF &RESULT > 10000 THEN
                WRITE "    ___Over Range    "
                #TEMP = 1
            ELSE
                &RESULT = &RESULT * &CH1
            ENDIF
        NEXT #TEMP
        END
```

The first difference you will notice, is that we don't need to use a separate line to load the variable **#TEMP** with **&CH2**, as we did in Lesson 20. This is done for us automatically in the instruction on line 3 **FOR #TEMP = &CH2 TO 1 STEP -1**. The **FOR #TEMP = &CH2** tells the compiler to initially load **#TEMP** with a starting value of **&CH2**.

The next part of the instruction is **TO 1 STEP -1**. This tells the compiler that each time the macro goes through the loop, it needs to subtract 1 (**STEP -1**) from **#TEMP** until it equals 1 (**TO 1**). Because I want **#TEMP** to decrease in value, I wrote **STEP -1**. I would write **STEP 22** if I wanted the value to increase by 22 each time it went through the loop. The **STEP** instruction is optional and you can omit it if you like. If you leave it off, the compiler will default to **STEP 1**.

Near the end of the macro you will see the instruction **NEXT #TEMP**. The commands between the **FOR** instruction and the **NEXT** instruction are executed as normal. Then, when the instruction **NEXT #TEMP** is executed, the value of **#TEMP** is decremented and, if it is not equal to 1, the macro jumps back up to the next line after the **FOR** instruction.

In some cases you might have a very simple **FOR NEXT** loop, like the following macro, where you just want to go through the loop ten times.

```
MAIN_MACRO:
FOR #TEMP = 0 TO 10
    &CH1 = &CH1 * &CH2
NEXT #TEMP
END
```

Other applications might require a much more complex **FOR NEXT** loop. The TDS allows you to use expressions in a **FOR** instruction, provided they are on the right-hand side of the = operator. Here is an example:

```
MAIN_MACRO:
FOR #TEMP = &CH1-&CH2 TO &CH1+&CH2 STEP &CH3*&CH4/&CH3
    &CH1 = &CH1 * &CH2
NEXT #TEMP
END
```



This looks a bit complicated, but the operation is exactly the same. Initially, #TEMP is loaded with  $(\&CH1 - \&CH2)$ . Each time through the loop, #TEMP is changed by a value of  $(\&CH3 \times \&CH4 / \&CH3)$ , until it equals  $(\&CH1 + \&CH2)$ .

If you're using a **FOR NEXT** loop like this, you really need to check that it won't end up in an endless loop! You need to take into account all possible values of input data.

## Lesson 25 – Expressions in IF ELSIF Commands

Normally **IF** and **ELSIF** tests compare a register or a variable with a constant. However, you can use simple expressions in an **IF** or **ELSIF** command, as shown in the following macro:

```
Line 2      MAIN_MACRO:
            IF &DISPLAY >= (&CH2 + 1050) THEN //if meter display
                                                    //is >= Channel2+1050 then
                WRITE "    ___Warning - tank is about to over flow!  "
            ELSIF &DISPLAY >= (&CH2 + &CH3) THEN
                WRITE "    ___Tank is Full          "
            ELSE
                WRITE "    ___Tank is still filling      "
            ENDIF
            END
```

In line 2, the value of **&DISPLAY** is compared to the value of **(&CH2 + 1050)**. This means that the numeric value of **&CH2+1050** is computed first, and then it is compared to the value of **&DISPLAY**. The content of **&CH2** is not changed in any way.

In the **ELSIF** instruction, the value of **&DISPLAY** is compared to the value of **(&CH2 + &CH3)**. This means that the numeric value of **&CH2+&CH3** is computed first, and then it is compared to the value of **&DISPLAY**. Again, the contents of **&CH2** and **&CH3** are not changed.



## Lesson 26 – Number Arrays and Register Arrays

In Lessons 19 and 20 we looked at string arrays where the array was made up of many individual strings, one after the other. For some macros, it is necessary to use an array of numbers instead of text strings. This is particularly useful when using data tables in your macro.

The following macro shows how to use number arrays in the TDS. It is similar to the macro we used in Lesson 17 to select the sensor type from 1 to 4. But, now there are 10 different sensors, and each sensor type has a preset scaling factor. We will use a number array to store the different scaling factors for sensors 0 to 9. Let's see how it looks:

```
DIM A[] = [1200,461,7891,2141,11500,17300,2000,55,800,12345]

REG &SENSOR_TYPE = &USER_MEMORY_1

F1_BUTTON_MACRO:
//*****
//The F1 button is now the select button to enter
//the edit mode
IF &STATE = 0 THEN
    &STATE=1                //used to keep track of
                            //operational state
    EDIT &SENSOR_TYPE      //user memory 1 = sensor type
    &EDIT_MAX = 9           //maximum value = 9
    &EDIT_MIN = 0          //minimum value = 0
    &EDIT_DEF = 0          //default value = 0
                            //(Up & Down pressed together)
    WRITE "Sensor" //this flashes on the display
ENDIF
END
```

Macro continued on next page .....

..... From previous page

```
EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &SENSOR_TYPE //exit edit mode, save
                                //sensor type in user memory 1
//&STATE = 0
    #SCALE_FACTOR = A[&SENSOR_TYPE]
ENDIF
END
```

Line 28

In the first line I have defined the number array using the `DIM A[] =` command, in a similar way as we did with text strings in Lessons 19 and 20. The only difference here is that there are numbers, separated by commas, inside the array. Because we want to store a number only in the array, we do **not** put quotation marks around. If we did, the compiler would treat it as a text string array instead of a number array.

You cannot mix numbers and text strings inside the same array. You must create separate arrays for numbers and text strings. The numbers in a number array can only be unsigned 16-bit fixed point numbers. This means they must be a number between 0 and +65,535 and they cannot contain fractions.

The last line of the `EDIT_MACRO` (line 28) reads `#SCALE_FACTOR = A[&SENSOR_TYPE]`. This operates in exactly the same manner as a text string array with the `WRITE` or `APPEND` command. The user defined register `&SENSOR_TYPE` points a number in array `A`. This number is then stored in the variable `#SCALE_FACTOR`.

Number arrays are useful when using large data tables in your macro. Maybe you need to have a lookup table to scale an input to a pre-defined response curve that never changes. Number arrays are fine for this because you can enter the data points into your macro and download the table with the macro, all in one step.



But what happens if you need to change some points later on? You will have to edit your macro, re-compile it and reprogram the meter again with the modified macro. That might be okay once or twice, but what if you need to change these values often? That's why the TDS includes register arrays!

The following macro shows how to use register arrays in the TDS. It is similar to the previous macro, but now there are 100 different sensors, and for each sensor type selected, we need to enter a scaling factor into the meter as well. In this macro, the 100 scale factors are stored in user memories 1 to 100 instead of in a number array in the macro. This means that the values of these scaling factors can be changed without needing to reprogram the macro. Here's how it looks.

```
REG &SCALE_TABLE = &USER_MEMORY_1
REG &SENSOR_TYPE = &USER_MEMORY_101

F1_BUTTON_MACRO:
//*****
//The F1 button is now the select button to enter
//the edit mode
IF &STATE = 0 THEN
    &STATE=1                //used to keep track of
                            //operational state
    EDIT &SENSOR_TYPE      //user memory 101=sensor type
    &EDIT_MAX = 99         //maximum value = 99
    &EDIT_MIN = 0          //minimum value = 0
    &EDIT_DEF = 0          //default value = 0
                            //(Up & Down pressed together)
    WRITE "Sensor" //this flashes on the display
ENDIF
END
```

Macro continued on next page .....

..... From previous page

```

EDIT_MACRO:
//This macro is run each time the Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT &SENSOR_TYPE //exit edit mode, save
                                //sensor type in user memory 1
Line 26    EDIT &SCALE_TABLE[&SENSOR_TYPE]
    &STATE = 2
    &EDIT_MAX = 10000 //maximum value = 10000
    &EDIT_MIN = 0 //minimum value = 0
    &EDIT_DEF = 1000 //default value = 1000
                                //(Up & Down pressed together)
    WRITE " SCALE" //this flashes on the display

Line 35    ELSIF &STATE = 2 THEN
    EXIT_EDIT &SCALE_TABLE[&SENSOR_TYPE]
    //&STATE = 0
    ENDIF
END

```

The `EDIT_MACRO` now has two states. State 1 is used to select the sensor type from 0 to 99. State 2 is used to enter a scaling factor for the currently selected sensor. In line 26 you will see the command `EDIT &SCALE_TABLE[&SENSOR_TYPE]`. We have discussed the `EDIT` command in Lesson 13, but this is a little different. Instead of specifying a pre-defined register or variable for the `EDIT` command, I have used the register array `&SCALE_TABLE[&SENSOR_TYPE]`. The operation of this is similar to the number array above. The register `&SENSOR_TYPE` is used to point to the particular register in the array starting at `&SCALE_TABLE`. So if `&SENSOR_TYPE` equals 0, then the edit buffer is loaded with the first register of the array, which is `&SCALE_TABLE[0]`, (or `&USER_MEMORY_1`).

If `&SENSOR_TYPE` equals 49, then the edit buffer is loaded with the fiftieth register of the array which is `&SCALE_TABLE[49]`, (or `&USER_MEMORY_50`).

In line 35 the same register array has been used with the `EXIT_EDIT` command. This stores the edited value into the register array starting at `&SCALE_TABLE`. The register `&SENSOR_TYPE` is used to point to the particular register in the array.

Here are some other uses of register arrays.

```
&CH1 = &TABLE2_INPUT1[20]
#VALUE_A = &TABLE1_INPUT1[#INPUT_POINT] + &CH1
&SCALE_TABLE[&SENSOR_TYPE] = 1000
&SCALE_TABLE[10] = 1000
```

As you can see above, the pointer to the register in the array can be a pre-defined register, a variable, or a constant (i.e. a number).

Again, care must be taken to ensure that the pointer does not point to a register outside of the array. If it does, you will be reading or writing to registers outside the array that could produce very strange results. The compiler does not check for this so you will have to check this yourself. In some cases you may need to include extra tests to limit the value of the pointer.

## Lesson 27 – Redefining Registers as Bit Arrays

Sometimes you may only be interested in a single bit of a pre-defined register that is not pre-defined as a bit register itself. This can be achieved by redefining the register as a bit array using the **BITREG** command:

```

BITREG &CODE3 = [ |DONT_CARE, |MASTER_MODE,
|DONT_CARE2 ]

F1_BUTTON_MACRO:
|MASTER_MODE = ON
PRINT "RESULT = " + &RESULT + CHR(CR) + CHR(LF)
|MASTER_MODE = OFF
END

```

Just like the **BIT**, **REG**, and **DIM** command **BITREG** is used outside any macro. Of course the register has to be defined prior to its redefinition. You can only define as many bit variables as there are bits available to the register. In the example **&CODE3** is an 8-bit value, so up to 8 bit variables would be possible.



Please note that you don't have to redefine all available registers. The first bit variable will refer to Bit 0, the second to Bit 1 and so on. The bit variable names have the same restrictions as other user defined bit variables.

If you take a look at the programming code sheet (NZ101) you may notice that the serial mode is actually defined by the three lower bits of **&CODE3**. So if **|DONT\_CARE = ON** the macro would actually change between **Modbus** mode and **Print** mode. So, it would be wise to make sure that bit 0 and 2 are not set:

```

RESET_MACRO:
|DONT_CARE = off
|DONT_CARE2 = off
END

```

Sometimes you really don't care what the other bits are set to if you change it. In this case you can simply omit to name those bits:

```
BITREG &CODE2 = [ , , , , , , , |FAST_MODE ] // only bit 7
```

Here the first seven bits (bit 0 to bit 6) stay nameless and only the seven commas remain, only bit 7 is defined as |FAST\_MODE.



This omission of array fields is only available for BITREG definitions.

## Lesson 28 – Data Source Registers

In some applications you may need to change which input channel is displayed on the front of the meter or which data source is being used by the analogue output, setpoints, or totalisators. To achieve this you have to set the corresponding data source register to the appropriate register number. But how do you know the register number for CH1?

Of course you could look it up in the Register Supplement (NZ209) but there is a much easier way. You can use the ADDR command! Look at the macro below.

Line 6

```
MAIN_MACRO:
#My_Result = -1
IF |CAPTURE_PIN = ON THEN
    &DATA_SOURCE_DISPLAY1 = ADDR(&CH1)
ELSE
    &DATA_SOURCE_DISPLAY1 = ADDR(#My_Result)
ENDIF
END
```

The ADDR command simply looks up the register address of its argument for you. You like that?



In line 6, the ADDR command is used with a user defined variable. The actual register associated with a user defined variable is only available at compile time. Using ADDR is the only reliable way to determine the register number of a user defined register!

As the data source registers are the only registers that the ADDR command applies to, it's possible that you may overlook the addition of the ADDR command. If you do, the TDS will assume that you have, and will compile this shorter form as though the ADDR command is still there. For example, look at this next macro.



```
Line 6      MAIN_MACRO:
            #My_Result = -1
            IF |CAPTURE_PIN = ON THEN
                &DATA_SOURCE_DISPLAY1 = &CH1
            ELSE
                &DATA_SOURCE_DISPLAY1 = #My_Result // Caution
            ENDIF
            END
```

This will compile without an error – but the compiler will issue a warning. The end result will be functionally the same as the macro above. In line 6 the short form still looks up the register address of #My\_Result and not its value!

If you use the short form, there are some potential traps that you need to be aware of. For example, look at this next macro.

```
Line 6      MAIN_MACRO:
            #My_Result = -1
            IF |CAPTURE_PIN = ON THEN
                &DATA_SOURCE_DISPLAY1 = &CH1
            ELSE
                &DATA_SOURCE_DISPLAY1 = #My_Result + 1 // Caution
            ENDIF
Line 7
Line 8      &DATA_SOURCE_TOTAL1 = &DATA_SOURCE_DISPLAY1
            END
```

Here the short form doesn't apply in line 6 anymore because a mathematical expression has been added to the right hand side of the equals sign. Instead the expression will be evaluated and the result (in this case 0) will be assigned as a source register! If you want to add an offset like this to a register number you must use the ADDR command like this:

```
&DATA_SOURCE_DISPLAY1 = ADDR(#My_Result) + 1
```

Another exception is shown in line 8. The compiler recognizes that the register on the right hand side of the equals sign is also a data source register. In this case it copies the value of `&DATA_SOURCE_DISPLAY1` to `&DATA_SOURCE_TOTAL1` rather than it's address. Once again, if you wanted to copy the address you would have to use the `ADDR` command.



Because of the above potential problems, we recommend that you always use the `ADDR` command instead of relying on the short form. If you do forget, the compiler will issue a warning.

## Lesson 29 – Presetting Registers

In Lesson 14, we discussed user memories for permanent data storage when the meter is turned off. As soon as you start using the user memory you have to consider whether you need to initialize these registers. Have a look at this next macro.

```
DIM Selection[] = [ "", " Set 1", " Set 2", " Set 3" ]
REG &Selection = &USER_MEMORY_1

F1_BUTTON_MACRO:
IF &STATE = 0 THEN
    EDIT &Selection
    &EDIT_MIN = 1
    &EDIT_MAX = 3
    &EDIT_DEF = 1
    EDIT_TEXT Selection[]
    WRITE "CHOOSE"
    &STATE = 1
ENDIF
END

EDIT_MACRO:
EXIT_EDIT &Selection
&STATE = 0
END
```

If you run this macro for the first time and press F1 you might see garbage on the display or nothing at all. As soon as you press the UP or DOWN buttons you will be able to select one of the three sets. The next time you press F1 you will get your last selection on the display.

The reason for the empty or garbled display is that the user memory register is not initialized. The `EDIT` command doesn't check whether its argument lies within the boundaries of `EDIT_MIN` and `EDIT_MAX`. So if the user memory register contains a value outside of these boundaries the display will nevertheless try to find the string with this invalid offset.

If you initialize the register in the `RESET_MACRO` you will reset it every time at startup – so it actually becomes a volatile variable. Of course, you could set the appropriate register using the serial protocol, but this is pretty cumbersome. Let's see if we can do better:

```
Line 3      DIM Selection[] = [ "", " Set 1", " Set 2", " Set 3" ]
            REG &Selection = &USER_MEMORY_1
            MEM &Selection = 1
            // ...
```

In line 3 above you will see I have added the word `MEM` at the beginning of the line followed by `&Selection = 1`. The `MEM` assignment tells the TDS to preset the register `&Selection` to a value of 1.



Older versions of the TDS created a separate file with the suffix `.ee`. This is no longer the case as the TDS uses a simplified version of the file format used by the Configuration Utility.

Not all registers in the meter have a non-volatile memory backup. The `MEM` command will only work with user memories and configuration registers, which do have a non-volatile memory backup. Currently `MEM` cannot be used to initialize single bit registers. Instead, you have to set the full register the bit register belongs to. For more information on configuration and bit registers see Register Supplement NZ209.

The `MEM` command also allows you to treat any register as an array offset:

```
MEM &SETPOINT1 [] = [ 100, 200, 300 ]
```

This initializes `SETPOINT1` with 100, `SETPOINT2` with 200, and `SETPOINT3` with 300. The only restriction to these arrays is that they can only handle one register type, integer, floating point or string.



The compiler will issue a warning if a value doesn't fit into a register or is initialized more than once.

One obvious use of array initialization is the linearization table registers. The lines below show how to initialize the input and output points for Linearization Table 1.

```
MEM &TABLE1_INPUT1 [] = [ 10, 20, 30, 40 ] // ...
MEM &TABLE1_OUTPUT1 [] = [ 1000, 2000, 3000, 4000 ]
                               // ...
```

Well, that works okay, but in a large table it is quite hard to match up a specific input point with its respective output point.



Wouldn't it be nice if you could actually enter input and output values in pairs instead of two separate arrays? The TDS provides a special format just for the initialization of linearization tables as shown below.

```
MEM TABLE1 [] = [ 10, 1000, 20, 2000, \
                   30, 3000, 40, 4000 ] // ...
```

You could make this even more readable by changing it to this...

```
MEM TABLE1 [] = [ 10, 1000, \
                   20, 2000, \
                   30, 3000, \
                   40, 4000 ] // ...
```

Remember, this special format only works for linearization tables and you must use the labels `TABLE1`, `TABLE2`, `TABLE3`, and `TABLE4`.

This Page Intentionally Left Blank

# Advanced Lessons

# 4

---

Lesson 30 – Special String Characters . . . . .	2
Lesson 31 – Higher Math Operators . . . . .	3
Lesson 32 – Logical Operators. . . . .	5
Lesson 33 – Linearization . . . . .	8
Lesson 34 – Printing to the Serial Port in Macro Mode . . . . .	9
Lesson 35 – ASCII Characters . . . . .	12
Lesson 36 – Reading an ASCII String From the Serial Port . . . . .	13
Lesson 37 – Reading Non ASCII Data From the Serial Port . . . . .	17
Lesson 38 – String Registers . . . . .	18
Lesson 39 – Parsing . . . . .	21
Lesson 40 – Modbus Master Macro . . . . .	27
Lesson 41 – Output port for serial commands and WRITE . . . . .	32
Lesson 42 – The INCLUDE Command . . . . .	34
Lesson 43 – Data Logging . . . . .	35
Lesson 44 – USER_MEMORY vs. USER_MEMORY_BYTE. . . . .	37
Lesson 45 – The GOTO Command. . . . .	39

## Lesson 30 – Special String Characters

In earlier lessons we discussed the use of the **WRITE** and **APPEND** commands with text strings. These commands allow the scrolling of a message string across the display.

In some applications you may require information from a variable or a predefined register to appear somewhere in the text string. The next macro shows you how to do this:

```
MAIN_MACRO:
IF |SP1 = ON THEN
    WRITE "    ___Channel 1 = " + &CH1 + " Volts    "
ENDIF
END
```

In the third line of this macro you will see the text **WRITE** `___Channel 1 = " + &CH1 + " Volts "`. The first half of this instruction looks familiar, but then it has `+ &CH1 + " Volts "`. The `+ &CH1` tells the compiler to insert the numerical value of `&CH1` into the message string. Then the text `+ " Volts "` adds the text `" Volts "` on to the end of the string.

If the value of `&CH1` is 1.2345 at the time the **WRITE** command is executed, the message `" ___Channel 1 = 1.2345 Volts "` is scrolled across the display.

This can be used with both the **WRITE** and the **APPEND** commands.



## Lesson 31 – Higher Math Operators

In Lesson 9 we looked at some of the simple maths operators, but BASIC also includes higher maths operators. Unlike the simple operators, these higher maths operators only require one operand. The following list shows the higher maths operators available and the allowable input range:

SQR	Square root (positive number)
LN	Natural log (base e)
LOG	Common log (base 10)
SIN	Sine (-65535 to +65535)
COS	Cosine (-65535 to +65535)
TAN	Tangent (-65535 to +65535)
SINH	Hyperbolic sine (-65535 to +65535)
COSH	Hyperbolic cosine*
TANH	Hyperbolic tangent*
ARCSIN	Arc sine (-1 to +1)
ARCCOS	Arc cosine (-1 to +1)
ARCTAN	Arc tangent*

\* These functions accept a value in the range of  $\pm 1.175494E - 38$  to  $\pm 3.402823E38$ .

Here are some examples of how these operators are used:

```

MAIN_MACRO:
&CH1 = LN (SIN &CH1 * COS 6) + SQR &CH2
&RESULT = 10 * LOG &CH1
END

```



When using higher maths operators you need to ensure that all input values into an equation are within the acceptable range for the operator you are using. This may necessitate testing of some input values before processing.

The other point to note when using higher maths operators is the amount of processing time required. Most of these functions require more processing power than the simple maths operators and other functions. This is not normally a problem for small equations, but for very complicated maths it may be. If you are using complicated formulas and you notice the meter or display operating slow or erratic, the cause may be a shortage of processing time.



If you are operating the meter at the 0.01 second update rate, then you should change this to the 0.1 second update rate.



**For full details on configuring the meter display update rate, see Initial Setup Procedures in the relevant Tiger 320 Series User Manual.**

Sometimes you may be interested only in the absolute value of calculation and not whether it is positive or negative:

---

```
&RESULT = ABS (&CH1 - &CH2)
```

---

The ABS() function simply multiplies the expression in parentheses with -1 if its value is negative. Please note that the parentheses are mandatory for this function.

## Lesson 32 – Logical Operators

Although not often required in simple macros, sometimes it may be necessary to modify or test only part of a register value without affecting or seeing the contents of the rest of the register. This is particularly true for many of the registers in the meter that are used to control the meter's operation. Often one register may contain individual bit flags, or groups of bit flags, that each control different meter functions. Because of this, it would be very difficult to test the whole register for all possible functions.

BASIC includes the following simple logic operators to solve this:

AND	Logical And
OR	Logical Or
XOR	Logical Exclusive Or

In each case, these operators require two operands. Lets look at a macro that uses some of these operators.

```
DIM A[] = ["J type","K type","R type","S type",\  
           "T type","B type","N type"]  
  
F1_BUTTON_MACRO:  
//Called by the operating system when the F1 button  
//is pressed  
IF &STATE = 0 THEN  
    #TEMP = &CODE2 AND 0x38  
    IF #TEMP = 0x8 THEN                //if thermocouple  
                                        //mode then  
        #TC_TYPE = &CODE2 AND 0x7    //get current  
                                        //thermocouple type  
  
    EDIT #TC_TYPE  
    &EDIT_MAX=6  
    &EDIT_MIN=0  
    &EDIT_DEF=0  
    WRITE ""  
    WRITE "Sensor"
```

Macro continued on next page .....

..... From previous page

```

        EDIT_TEXT A[]           //this line selects DIM A[] as
                                //the string array

        &STATE=1

    ENDF
ENDIF
END

EDIT_MACRO:
//Called by the operating system when Prog button is
//pressed in edit mode
IF &STATE = 1 THEN
    EXIT_EDIT #TC_TYPE
    &CODE2 = (&CODE2 AND 0xF8) OR #TC_TYPE
ENDIF
END

```

The macro above allows the user to change the thermocouple (TC) sensor type from the F1 button. The macro first checks that the meter is running in thermocouple mode before allowing entry to select the TC sensor type. &CODE2 sets the TC mode and sensor type. Bits 0, 1, and 2 select the TC sensor type, while bits 3,4 and 5 select the operating mode for channel 1. Bits 6 and 7 of &CODE2 are used for other functions in channel 1 that we don't want to change.

In the 2nd line of the F1\_BUTTON\_MACRO you will see the instruction #TEMP = &CODE2 AND 0x38. This means that the value of &CODE2 is ANDed with 0x38 (hex) before it is stored in #TEMP. So, effectively we are only looking at the TC mode (i.e. bits 3, 4 and 5). The register &CODE2 itself is unaltered, only it's value is used and ANDed.

If &CODE2 = 77 (dec), then let's look at the result of ANDing this with 0x38 (hex). To understand this correctly, we first need to convert these numbers to a binary format, because the logical operators all operate in a bitwise fashion:



```

&CODE2 = 77 (dec) =      01001101 (bin)
0x38 (hex) =            00111000 (bin)
ANDing =                00001000 (bin)

#TEMP = 0x8 (hex)

```

The logical **AND** of two bits means that the result is a 1 only if both bits are a 1. So in the case shown above, by **AND**ing with the constant 0x38 (00111000 bin), we are effectively only looking at bits 3, 4, and 5, and are ignoring all other unwanted bits. The result that is loaded into #TEMP is 00001000 (bin) which equals 0x8 (hex).

In the third line of the `EDIT_MACRO` you will see the instruction `&CODE2 = (&CODE2 AND 0xF8) OR #TC_TYPE`. In this case, the brackets force `&CODE2` to be **AND**ed with 0xF8 first, and then **OR**ed with the variable `#TC_TYPE`. The logical **OR** of two bits means that if either bit is a 1, the result is 1. Assuming that `&CODE2 = 77` and `#TC_TYPE = 3`, this is the result.

```

&CODE2 = 77 (dec) =      01001101 (bin)
0xF8 (hex) =            11111000 (bin)
ANDing =                01001000 (bin)
#TC_TYPE = 3 (dec) =    00000011 (bin)
ORing =                 01001011 (bin)

#TEMP = 75 (dec)

```

The logical exclusive or (**XOR**) of two bits means that if either bit is a 1, the result is a 1, excluding the case when both bits are a one. This operator is useful when looking for a difference in bits between two variables or registers. Any bits that are not zero indicate a difference.

## Lesson 33 – Linearization

The Tiger 320 Series meter includes up to four tables that are used for linearization of an input signal. The meter can be configured so that each of the four input channels can have a user defined linearization curve applied to it.

The TDS also includes an instruction called **SCALE** that allows any register or variable to have the same linearization applied to it. Here's how it is used:

```
MAIN_MACRO :  
#TEMP = &CH1 + &CH2 + &CH3  
&RESULT = SCALE (#TEMP, 2)  
END
```

In line 2, #TEMP is loaded with &CH1 + &CH2 + &CH3. The next line reads &RESULT = **SCALE** (#TEMP, 2). The text **SCALE** (#TEMP, 2), tells the compiler to apply the linearization curve, specified in Table 2, to the value stored in variable #TEMP. After linearization, the new value is then stored in &RESULT. The value of #TEMP is not changed by the **SCALE** instruction.

When using the **SCALE**(*x*,*y*) instruction, *x* can be a number, a variable or a predefined register. While *y* specifies the 32-point table to be used for linearization and must be a number from 1 to 4. The curve in tables 1 to 4 must be programmed in the usual way, as described in the Tiger 320 Series user manual Linearizing Supplement (NZ207).

The **SCALE** instruction requires more processing time than many of the standard instructions. If it is executed many times over in the same macro, it may slow down the overall performance of the meter. This is of particular importance if the meter is being run in the fast update mode.

## Lesson 34 – Printing to the Serial Port in Macro Mode

Tiger 320 Series meters incorporate several modes of serial communications. Included in these are industry standards such as Modbus and Devicenet, and also a simple ASCII and printer mode developed by Texmate. However, in some applications, these modes may not be suitable.

For instance, you may need to print several different messages to a printer, but the standard print mode in the meter only provides one. Or maybe you are trying to control an instrument that uses a nonstandard protocol.

The TDS allows you to send a string of characters to the serial port with the **PRINT** instruction. To use the **PRINT** instruction, the meter must be set to macro mode in Code 3[XX2] and the baud rate, parity, time delay, and address must be set up correctly in the Calibration Mode [20X].



**For full details on configuring the meter in the MACRO Mode, see Serial Communications Module Supplement (NZ202).**

The **PRINT** instruction has many similarities to the **WRITE** instruction, with a few additions. Remember the first macro we wrote? Well let's try it again, but this time, instead of sending it to the display of the meter, we'll send it to the serial port so that it can be printed on a serial printer.

```
F1_BUTTON_MACRO :  
PRINT "Hello World"  
END
```

That's all there is to it! I used the `F1_BUTTON_MACRO` this time so that the message prints to the serial port every time the F1 button is pressed.

The **PRINT** instruction can also be used with special string characters, similar to the **WRITE** instruction. Here's an example:

```
MAIN_MACRO:
  IF &TIMER1 > 10 THEN
    &TIMER1 = 0
    PRINT "Channel 1 = " + &CH1 + " Volts"
  ENDIF
END
```

This macro prints a string to the serial port once per second. The **PRINT** instruction in line 4 tells the compiler to print the string "Channel 1 = ", then the current value of &CH1, and then the string " Volts".

If you download and run the above macro, you will find that it keeps printing in a long line across the page. This is because I haven't included any carriage returns or line feeds. The next macro should fix this problem:

```
MAIN_MACRO:
  IF &TIMER1 > 10 THEN
    &TIMER1 = 0
    PRINT "Channel 1 = "+&CH1+" Volts"+CHR(CR)+CHR(LF)
  ENDIF
END
```

As you can see, on the end of the **PRINT** instruction I have added +CHR(CR) + CHR(LF). This text tells the compiler to add the ASCII character number for a carriage return (CR) and a line feed (LF). These are special ASCII characters that tell the printer where to start printing. There are several of these special characters that can be used. These are as follows:

CR	carriage return
LF	line feed
BS	back step
NUL (or NULL)	null character
ESC	escape character
TAB	tab character
FF	form feed
VTAB	vertical tab
BELL	'beep'



For some applications, you may need to send other non-printable ASCII characters to the serial port. This is often the case if the meter is connected to a PC instead of a printer. You can do this by using the `PRINT CHR(x)`, where `x` can be any number from 0 to 255. You can use decimal, octal, or hexadecimal numbers for `x`, as shown in the following macro:

```
MAIN_MACRO:
  IF &TIMER1 > 10 THEN
    &TIMER1 = 0
    PRINT CHR(0xAA)+CHR(0xAA)+CHR(11)+"Hello " + \
      "World"+ CHR(0127)+CHR(NUL)
  ENDIF
END
```

An octal number has a leading 0, and a hexadecimal number has a leading 0x.

The rules regarding the `PRINT` instruction are similar to those for the `WRITE` instruction. The `PRINT` command can only print one string at a time. If you try printing a new string before the current string has finished, the macro terminates and hands control back to the operating system of the meter.

After a print string has been completely sent, the serial port is reset into a standby mode. In standby, it is ready to receive serial data or transmit new data with the `PRINT` instruction.

If you want to stop a current string from being printed you can use the instruction `PRINT ""` (i.e. a print instruction with an empty string). This functions similar to the `WRITE ""` instruction. It flushes the serial buffer and sets the serial port into standby mode.



An important point to note is that the meter has only one serial buffer, which is shared between transmit and receive. If you use the `PRINT ""` instruction while serial data is being received, you could clear part, or all of a received message. In Lesson 36 we will learn more about the receive side of the serial port.

The meter only operates in half duplex, which means that it can be either transmitting or receiving, but not both at the same time.

## Lesson 35 – ASCII Characters

In lesson 28 I introduced the CHR() function to send arbitrary ASCII characters with the PRINT command. For the most common characters (carriage return, line feed...) ASCII code constants are available (CR, LF...). All standard alphanumeric characters can be entered directly in the string.

```
F1_BUTTON_MACRO:
PRINT "RESULT = " + &RESULT + CHR(CR) + CHR(LF)
END
```

However, the meter includes registers for an optional text character in the least significant digit of the display. These registers need to be loaded with the ASCII code of the character required. For example, if Channel 1 was measuring watts and you wanted to display a "W" in the right hand side of the display, you can do this:

```
RESET_MACRO:
&TEXT_CHARACTER_CH1 = 0x57 // "W" for Watts
END
```

That's great, but you first have to know what the ASCII code for the letter "W" is! Instead of looking up the corresponding ASCII code yourself, you can also use the ASC() function which returns the ASCII code of a given character.

```
RESET_MACRO:
&TEXT_CHARACTER_CH1 = ASC("W") // "W" for Watts

// to disable character display
// &TEXT_CHARACTER_CH1 = NULL
END
```

Keep in mind that not all ASCII characters can be displayed. If you want to return to the standard display without a character in the least significant digit, set the character register to 0 or NULL:

```
F1_BUTTON_MACRO:
// to disable character display
&TEXT_CHARACTER_CH1 = NULL
END
```

## Lesson 36 – Reading an ASCII String From the Serial Port

In Lesson 34 we looked at the `PRINT` instruction, which is used to send (or transmit) data from the meter to another device. For printing applications, or data logging to a PC, this is usually all that is needed.

However, for more sophisticated applications, it is often necessary for the meter to send and receive data, and in many cases, other devices will have their own communications protocol. The macro allows you to create your own serial protocol and access the serial port using the Macro mode. In Macro mode, only very low level serial port functions are handled by the meter, leaving protocol issues to be handled by the macro. The meter looks for a specified string, stores the string in its receive buffer, and leaves the rest up to the macro.

Messages can be either fixed or variable length, and most serial protocols use a special character in the message string to signal the start or the end of the string. The Tiger 320 Series meter has two predefined registers that allow it to extract the required string from an incoming message. These are called `&STRING_LENGTH` and `&STRING_CHARACTER`. The following macro shows the use of these two registers and the serial port in Macro mode:

```
MAIN_MACRO:
  IF &TIMER1 > 10 THEN
    &TIMER1 = 0
    PRINT ""           //clear buffer and reset serial port
    PRINT "SR2*"      //read the display register in
                      //remote meter

    &STRING_LENGTH = 0 //receive string is variable
                      //length, so string length=0

    &STRING_CHARACTER = 10 //end of string character=
                          //line feed (10)

  ENDIF
```

Macro continued on next page .....

..... From previous page

```
IF |RECEIVE_READY = ON THEN
  #TEMP_CHAR = &RECEIVE_BUFFER[0]
  IF #TEMP_CHAR <> 0 THEN
    &RESULT = &RECEIVE_RESULT
  ENDIF
ENDIF
END
```

The above macro is designed for two Tiger meters connected together through the serial ports. The meter running the macro is set to Macro mode, while the second meter is set to ASCII mode. The macro shown here is written to copy the ASCII mode, but acting as a master. Every second, the macro sends a request to the second meter and waits for a response.



**For full details on configuring the meter in the Macro Mode, see Serial Communications Module Supplement (NZ202).**

In the fourth line, the `PRINT ""` instruction clears the serial buffer and resets the serial port. The next line then prints the message `"SR2*"` to the serial port. In the Texmate ASCII mode, `SR2*` is a request to read the display register of the meter. The expected result from the second meter is an ASCII number of varying length, terminated in a carriage return and line feed.

For variable length messages, the register `&STRING_LENGTH` is set to zero, which tells the meter to expect a string of unknown length. If `&STRING_LENGTH` equals zero, the register `&STRING_CHARACTER` functions as an end of message (or terminating) character, so in this case, it is loaded with the ASCII character for a line feed (i.e. 10). The meter then keeps receiving data until it finds a line feed character, and then it sets the `|RECEIVE_READY` flag to say that it has received a new message.

The next part of the macro checks the `|RECEIVE_READY` flag. If it is true, it then loads the variable `#TEMP_CHAR` with the first byte of data from the receive buffer. The receive buffer is treated as an array, with the first byte located at `&RECEIVE_BUFFER[0]` and the last byte at `&RECEIVE_BUFFER[99]`. In this macro, the first byte of the receive buffer is checked for a value other than zero, because zero is an ASCII null, which signals an error condition in the ASCII mode.

If the first byte is not a zero, then the register `&RESULT` is loaded with a register called `&RECEIVE_RESULT`. This is a special register used by the meter in macro mode, to store numeric values from an ASCII string. When a message is received that contains a string of ASCII numerals, the meter picks out the first string of numerals and stores them in the register `&RECEIVE_RESULT`. They are stored in a fixed point format and the decimal point is ignored, so a string of 12.345 will be stored as 12345. Any following numeric values in the same string are ignored.

Some devices use a fixed length message with a start character at the beginning of the string. In this mode, `&STRING_LENGTH` is loaded with a value from 1 to 255, which defines the total length of the string you want to receive. The register `&STRING_CHARACTER` is then loaded with a number from 0 to 255, which defines the start of string character. In this mode, the meter searches through the incoming serial data, discarding data until it finds the correct start character. It then reads in the specified number of bytes of data and sets the `|RECEIVE_READY` flag.

When the meter has set the `|RECEIVE_READY` flag, in either mode, it disables the serial receive mode and waits for the macro to reset it. Another register called `&RECEIVE_COUNT` can be used to determine the total length of the receive string.

You can also compare a received message with an ASCII text string, as shown in the following macro:

```
MAIN_MACRO:
&STRING_CHARACTER = 10
&STRING_LENGTH = 0
IF |RECEIVE_READY = ON THEN
  IF SERIAL_INPUT = "HELLO" THEN
    WRITE "    ___YOU GOT IT RIGHT!    "
  ENDIF
  PRINT ""
ENDIF
END
```

The line **IF SERIAL\_INPUT = "HELLO" THEN** tells the compiler to compare the received string with the text `HELLO`, and if it is the same it executes the code under the **IF** instruction. It starts comparing each character in the string, starting from the first character in the string (i.e. `&RECEIVE_BUFFER[0]`). This instruction is case sensitive, so each character must be an exact match. If the received string is longer than the test string, the rest of the characters in the received string are ignored.

## Lesson 37 – Reading Non ASCII Data From the Serial Port

In Lesson 36 we considered message strings that were made up of ASCII characters, but sometimes non ASCII data is also used. This can still be done with the macro, but it is more involved.

You still need to set up `&STRING_LENGTH` and `&STRING_CHARACTER`. To test the received data you can still use the string compare test as follows:

```
MAIN_MACRO:
&STRING_CHARACTER = 0x55
&STRING_LENGTH = 5
IF |RECEIVE_READY = ON THEN
//check for 0x55,0xAA,0x00
    IF SERIAL_INPUT = CHR(0x55)+CHR(0xAA)+CHR(0) THEN
        &RESULT = (&RECEIVE_BUFFER[3] * 256)+ \
        &RECEIVE_BUFFER[4] // get data
    ENDIF
    PRINT " "
ENDIF
END
```

Instead of testing for a complete ASCII string, this macro tests for a start sequence of 0x55, 0xAA, 0x00, using the line `IF SERIAL_INPUT = CHR(0x55) + CHR(0xAA) + CHR(0) THEN`. This uses the `CHR(x)` instruction, which enables you to test for any number between 0 and 255 in decimal, hex, or octal.

If the macro finds the correct start sequence, it then extracts the fourth and fifth bytes from the receive buffer and treats these as data.

For more complicated protocols, you can interrogate each byte of the received message separately by accessing `&RECEIVE_BUFFER[x]`, where `x` can be a number from 0 to 99, or a variable or register. You can also use the predefined register `&RECEIVE_COUNT` to determine the length of the new message.

## Lesson 38 - String Registers

In the Tiger 320 series, only a small number of string registers are available such as the print string and the display texts (e.g. SP\_1 or CH\_3). In a macro they can only be preset with the MEM command during the macro download. All other texts have to be constant. Even the text string arrays - defined with the DIM command (in lesson 19) - are actually constant, even though they can be used with a variable index with the WRITE/APPEND command. Here's an example of a macro which prints different messages on a 320 series meter.

```
Line 1      DIM ConstMsg[] = [ "      ", "OK", "ABOVE", "BELOW" ]
            CONST mSPACE = 0
            CONST mOK = 1
            CONST mABOVE = 2
            CONST mBELOW = 3

            RESET_MACRO:
            #msg_index = mOK
            END

            MAIN_MACRO:
            if |SP1 = on then      // above high setpoint
                #msg_index = mABOVE
            elsif |SP2 = on then  // below low setpoint
                #msg_index = mBELOW
            else
                #msg_index = mOK   // between setpoints
            endif

            // show current message
Line 21     write ConstMsg[mSPACE]
Line 22     append ConstMsg[#msg_index]
Line 23     append ConstMsg[mSPACE]
            END
```

In line 1 the DIM command defines the text string array that is used in lines 21-23 to show the current message according to the status of



the setpoints.



The notation used with WRITE/APPEND in this macro is unique to text string arrays. It cannot be used with the PRINT command or with register arrays with a variable index. The above macro has two disadvantages: If you want to change the texts you actually have to change the macro. The second problem is that these texts are stored in the macro area. So if you have a very complicated macro and at the same time want to use a lot of text messages, you may actually run out of space. Therefore in the Tiger 380 series 64 USER\_TEXT registers as well as 8 text variables have been added. These can be used for text messages on the display or for serial communication. So let's have a look how the above macro would look like with text registers:

Line 3

```

REG &CUR_MSG = &TEXT_VARIABLE1
REG &MSG = &USER_TEXT1
MEM &MSG[] = [ "OK", "ABOVE", "BELOW" ]
CONST mOK = 0
CONST mABOVE = 1
CONST mBELOW = 2

```

```

RESET_MACRO:
#msg_index = mOK
END

```

```

MAIN_MACRO:
if |SP1 = on then // above high setpoint
    #msg_index = mABOVE
elsif |SP2 = on then // below low setpoint
    #msg_index = mBELOW
else
    #msg_index = mOK // between setpoints
endif

```

Line 22

```
// show current message
```

Line 23

```

&CUR_MSG = &MSG[#msg_index] // copy text to variable
write "          " + &CUR_MSG + "          "
END

```

In line 3 the registers USER\_TEXT1 to USER\_TEXT3 are preset with messages. As with the USER\_MEMORY registers it is recommended to initialize these registers before their first use. In line 22 the current message is copied to a text variable register. This is necessary as the WRITE/APPEND/PRINT command cannot cope with a register array with variable index.

Finally, in line 23 the copy of the message is sent to the display, just like any other register.

Besides copying one text register to another, you can also assign a constant string to a text register. Furthermore you can compare text registers or constant strings with each other, whether they are equal or not. As each text register can hold up to 30 characters you can save up to 2 KB of macro space when you change from DIM arrays to USER\_TEXT registers. But the more interesting advantage is that the texts can be changed without having to change the macro. So when you use these texts for your user interface, your customer could change these - for example translate it into another language - with the Configuration Utility. Also for industrial use, several meters may be used in the same plant but in different locations. Here you could use the location as identifier for service calls:

```
REG &ID = &USER_TEXT1
REG &LOCATION = &USER_TEXT2
MEM &ID = "Tank Station"
MEM &LOCATION = "Block A 21"

CUSTOMER_ID_MACRO:
write "      " + &ID + " --- " + &LOCATION \
      + "      "
END
```

## Lesson 39 - Parsing

In lessons 36 and 37 we learned how to read incoming serial data and that we can compare the SERIAL\_INPUT to a constant string. This is fine as long as you are only waiting for a single message from a single device.

But when you connect several devices together over a RS485 bus or via Ethernet you will have to deal with various messages. Even though there are a few widespread protocols like Modbus to deal with that situation, there are many different serial protocols that are only used by a single manufacturer. In this lesson we will show you how the Tiger 380 can help you to deal with custom protocols.

As an example, let us assume that our meter is in Macro mode and therefore does not automatically act upon serial commands - not even in our own Texmate ASCII protocol.

Usually it is easier just to use another serial port for the communication in ASCII mode (currently the Tiger 380 can have up to 3 serial ports) but here we want to show you how to deal with the parsing:

```
CONST DO_READ = ASC("R")
CONST DO_WRITE = ASC("W")

RESET_MACRO:
#address = 0
#command = DO_READ
#register = 0
&STRING_LENGTH1 = 0
&STRING_CHARACTER1 = ASC("*")
END
```

Macro continued on next page .....

..... From previous page

```

MAIN_MACRO:
if |RECEIVE_READY1 = true then
Line 14     &SERIAL_POINTER1 = 0
Line 15     if SERIAL_POINTER 1 = "S" then
Line 16     #address = INTEGER(SERIAL_POINTER 1, 3)
           if #address <> &SERIAL_ADDRESS1 then
           print "" : write "not me" : END
           endif
Line 20     #command = &RECEIVE_BUFFER1[&SERIAL_POINTER1]
Line 21     &SERIAL_POINTER1 = &SERIAL_POINTER1 + 1
           #register = INTEGER(SERIAL_POINTER 1, 5)
           if #command = DO_READ then
Line 25     if &SERIAL_POINTER1 <> &RECEIVE_COUNT1 then
           print "" : write "error" : END
           endif
           select #register
           case addr(&CH1):
           print &CH1 + CHR(CR) + CHR(LF)
           default:
           print CHR(NULL) + CHR(CR) + CHR(LF)
           endsel
           elseif #command = DO_WRITE then
           if SERIAL_POINTER 1 <> ", " then
           print "" : write "error" : END
           endif
           select #register
           case addr(&OFFSET_CH1):
           |NON_VOLATILE_WRITE = on
           &OFFSET_CH1 = INTEGER(SERIAL_POINTER 1)
           case addr(&SCALE_FACTOR_CH1):
           |NON_VOLATILE_WRITE = on
Line 45     &SCALE_FACTOR_CH1 = FLOAT(SERIAL_POINTER 1)
           endsel
Line 47     print ""

```

Macro continued on next page .....

..... From previous page

```
Line 49         else
                print " "
            endif
Line 52         else
                print " "
            endif
        endif
    END
```

The parsing commands are all based on the serial pointer which indicates the position in the RECEIVE\_BUFFER where parsing commands start to look for a pattern match.

In line 14 the &SERIAL\_POINTER1 register is reset to 0 so the next parsing command begins at the start of the buffer. With the string comparison in line 15 we look for the first occurrence of an S in the buffer.



In line 15, "SERIAL\_POINTER 1" is a special reference which effectively says *IF the string starting at the position pointed to by &SERIAL\_POINTER1 starts with "S" THEN*. Because it is a command it does not have the '&' prefix and should not be confused with the register &SERIAL\_POINTER1.

The &SERIAL\_POINTER1 register indicates the current position in the RECEIVE\_BUFFER1 (starting with 0) and can be read or written to like any other register.

When the parsing routines use the SERIAL\_POINTER command - without '&' prefix - the value of &SERIAL\_POINTERx will be modified to point to the position after the matching pattern.

So if the first character really is an S, the &SERIAL\_POINTER1 will be 1 in the next line. But if there are more characters in front of the S they will be simply ignored and the pointer will be set after the S. If there is no S at all in the string the condition will be false and the pointer will stay where it was before the test.



This is different from the string comparison with the SERIAL\_INPUT which always starts searching from the beginning of the string and leaves the serial pointer set to 0 when there is no match.



If you use multiple comparisons in a condition, each comparison will be evaluated from left to right (even if you use OR and the first condition is already true). This has to be considered as each comparison with either SERIAL\_INPUT or SERIAL\_POINTER might change the serial pointer.

The INTEGER command in line 16 looks for numbers and decimal points at the SERIAL\_POINTER. Multiple decimal points are possible but each will be ignored as by definition an integer does not have any decimals. Any other characters before the number will be ignored any other character after the number will terminate the pattern match and the pointer will stop at that position.

The second parameter is optional and restricts the search to a maximum of 3 characters. The serial address of the Tiger is in the range from 0 to 255 so there should be only 3 number characters for a valid address. Without this restriction the command will try to match as many consecutive numbers and decimal points as possible. So for our protocol we would not really need the restriction as the following pattern is a text character.

To match a single character it is easier to directly access the receive buffer (see Lesson 37). But as we do not use a parsing command in line 20, we have to increase the serial pointer explicitly in line 21.

The FLOAT command in line 45 works similar to the INTEGER command, but only one decimal point is allowed. The pattern also matches a two digit (signed) exponent, for example -1.234E-02 or -0.001234e1 would both be recognized as -0.01234.

Finally, we have to reset the serial communication registers with a PRINT "" (line 47, 49, and 52). This will clear the RECEIVE\_READY flag and reset the RECEIVE\_COUNT as well as a few more internal registers. However, it will not reset the &SERIAL\_POINTER register.

The above macro has been written to work with serial port 1 but could just as easily have been written to work with a different serial port by replacing the 1 in the appropriate registers and commands with the

number of the required serial port. If no number is specified in a special command like SERIAL\_POINTER or SERIAL\_INPUT then port 1 is used by default.

So much for the simple Texmate ASCII protocol - simple because it does not care whether the data is correctly received or not. You would have to confirm that with another request - or use the Modbus protocol to start with. But there are many serial protocols that include a checksum to ensure that the messages are correctly received. Here is a small subroutine to handle that:

```
Line 3      check_data:
Line 4      |data_corrupt = true
            &SERIAL_POINTER1 = &RECEIVE_COUNT1 - 2
            #temp = HEX(SERIAL_POINTER 1, 2)
            #checksum = 0
            for #index = 0 to &RECEIVE_COUNT1 - 3
              #checksum = #checksum + &RECEIVE_BUFFER[#index]
            next #index
            if #temp = (#checksum and 0x0000ffff) then
              |data_corrupt = false
            endif
            return
```

In this example we assume that the last two bytes before the terminating character are the 16bit checksum. So in line 3 the serial pointer is reset to the position of the checksum.

The HEX command in line 4 looks for hexadecimal numbers (0-9a-fA-F) - no decimal point allowed. The HEX command automatically limits the pattern length to 8 characters as we only have 32bit registers to store the data. But you can restrict it even more with the optional second parameter.

Apart from the parsing commands for the number conversions the serial pointer can also be used to copy a string to a text register:

```
&RECEIVE_BUFFER1[10] = CHR(NULL)
&SERIAL_POINTER1 = 2
&TEXT_VARIABLE1 = SERIAL_POINTER
```

The assignment in line 3 copies the text from the buffer until it encounters the end-of-string character NULL. At the moment there is no limiting parser command as for the numbers. Therefore a NULL character is written to the serial buffer in line 1.

So the text in &TEXT\_VARIABLE1 will contain the characters from index 2 to 9.

Writing to the RECEIVE\_BUFFER actually overwrites the received data. So if you want to extract a substring in this fashion you have to parse the data after the string first!





## Lesson 40 - Modbus Master Macro

In lessons 34-36 we talked about the serial Master/Slave ASCII protocol. This is often enough for communication between two Tiger meters or with another device.

However, in many industrial applications there are often more than two devices talking to each other and they may not be similar devices. To simplify communications these different devices typically speak the same language: a common protocol like Modbus.

Similar to our ASCII protocol, there is one master and up to 246 slaves, each with a unique address. The master can read or write to 16bit registers in the slaves. The messages include a checksum to confirm that the data was received correctly. The slaves always give a response, either the requested or changed value or an error message.

Even though the Tiger 320 can already be configured as a Modbus slave, only the Tiger 380 is able to work as a Modbus master. So let's have a look how this is done:

```
CONST mbREAD = 0
CONST mbWRITE = 1

RESET_MACRO:
#modbus = mbREAD
&DATA_SOURCE_DISPLAY1 = addr(&RESULT)
Line 7 &POLL_TIME=200 // run Modbus master macro every 2 secs
Line 8 &RESPONSE_TIME=10 // time out if no reply in 1 sec
END

F1_BUTTON_MACRO:
#modbus = mbWRITE
&RESULT = 0 // reset remote counter
END
```

Macro continued on next page .....

..... From previous page

```

F2_BUTTON_MACRO:
#modbus = mbWRITE
&RESULT = &CH3 // synchronize remote counter
END

Line 21  MODBUS_MASTER_MACRO:
         select #modbus
         case mbREAD:
           |LED1 = on
Line 25  MODBUS_READ (5, addr(&CH3), &RESULT)
         gosub check_message
           |LED1 = off
         case mbWRITE:
           |LED6 = on
Line 30  MODBUS_WRITE (5, &RESULT, addr(&CH3))
         gosub check_message
         #modbus = mbREAD
           |LED6 = off
         endsel
         END

```

In this example two Tiger 380 meters are connected to each other, both with &CH3 setup as a counter. The master meter polls &CH3 from the slave every two seconds. If the F1 button is pressed it will reset &CH3 on the slave to 0, if F2 is pressed it will set it to the value of its own &CH3.

### The Modbus Master Macro

All Modbus commands must be invoked within a MODBUS\_MASTER\_MACRO (line 21). If you attempt to use a Modbus read/write command in another macro the compiler will issue an error. The reason for this is that the serial communication usually takes longer than one macro cycle. So when the MODBUS\_MASTER\_MACRO issues a Modbus command it waits in the background for the response. As soon as it gets the response it continues with the next instruction. That way the meter doesn't freeze until it gets a reply and you don't have to check for a reply in the macro.

You will notice that the `RESET_MACRO` loads two new registers in lines 7 and 8. The `&POLL_TIME` register effectively sets the rate at which the `MODBUS_MASTER_MACRO` is executed (or called). A count of 1 in `&POLL_TIME` represents a time interval of 0.01 seconds. Here we have set this to a value of 200 which means that the `MODBUS_MASTER_MACRO` will be run every 2.0 seconds. The second register `&RESPONSE_TIME` sets the maximum time the meter will wait for a reply to be returned from a slave device. A count of 1 in `&RESPONSE_TIME` represents a time interval of 0.1 seconds. If the master does not receive a reply from the slave device within this time, it continues on with the next line in the `MODBUS_MASTER_MACRO`.

OK, let's have a closer look at the actual Modbus commands. In line 25 the master sends a read request to device number 5. The request is for the `&CH3` register on the slave and the data is stored in the local `&RESULT` register. In line 30 the master sends a write request to device number 5. The request is to write the value of the local `&RESULT` register to the `&CH3` register on the slave.

So for both commands the parentheses contain: device number, data source, and data destination.



In line 27 we use the `addr()` command to determine the source register number on the remote device. If you are not talking to another Tiger 380 you will have to enter the register number directly including the reference offset of 30000 or 40000. The same is true for write requests: here you have to enter the destination address with the reference offset of 40000. Only output registers (of reference type 4x) and input registers (reference type 3x) are supported with these commands. Coil/Bit addresses (reference type 0x and 1x) are currently not supported.



But wait a minute, `&CH3` and `&RESULT` are 32bit registers whereas Modbus can only send 16bit values. So what happens to values that are greater than 16bit?

Well, we simply send two consecutive registers for each 24/32bit register :-). This workaround is actually used by many other Modbus devices. To make it easier for you, the compiler determines from the type of the local register (the data destination for read, the data source for write) whether it should read/write one or two registers.

However, if you know that the value is only 16bit you can explicitly tell it to use only a single register.

```
MODBUS_READ (5, 515, &RESULT, MB_SHORT)
```

Possible values for this optional fourth paramter are:

8 bit: MB\_BYTE

16 bit: MB\_SHORT

24 bit: MB\_24, MB\_24\_SWAPPED

32 bit: MB\_LONG, MB\_LONG\_SWAPPED, MB\_FLOAT,  
MB\_FLOAT\_SWAPPED

Register pairs will be used for 24bit and 32bit register types.



The Tiger 380 uses only odd numbers for 24/32bit registers so they can be read as pairs of consecutive registers in Modbus mode.

However, on the Tiger 320 these paired registers are mapped to different register numbers in Modbus mode (see Tiger 320 Series Register Supplement).

Maybe you have noticed, that after the read and write command we call the following subroutine:

```
check_message:
  if (&MODBUS_MASTER_FLAGS and 0x7f) <> 0 then
    write "  ___ERROR - " + &MODBUS_MASTER_FLAGS \
          + "      "
  endif
  return
```

The Modbus standard defines 8 different exception errors that can be sent by the slave device instead of a regular reply. These are stored in bits 0-3 of the MODBUS\_MASTER\_FLAGS register. In addition three further errors which are detected by the master meter are indicated in bits 4-6. If no errors are detected by the master meter, bit 7 is set:



#### MODBUS\_MASTER\_FLAGS

bit 0 - bit 3 = Exception errors (Modbus standard)

1 = illegal function

2 = illegal data address

3 = illegal data value

4 = slave device failure

5 = acknowledge

6 = slave device busy

7 = negative acknowledge

8 = memory parity error

bit 4 = message timeout

bit 5 = CRC receive error

bit 6 = data type error

bit 7 = reception complete or waiting for a new command

## Lesson 41 - Output port for serial commands and WRITE

For the Tiger 380 series there is a dual serial module available. Even though you may use one port only for a dedicated purpose (like talking to an HMI) you still can choose whether that is port 1 or 2. Just add the port number after the serial command:

```

PRINT "Hello World from port 1" // use default port 1
PRINT 1 "Hello World from port 1" // use port 1
PRINT 2 "Hello World from port 2" // use port 2

IF SERIAL_POINTER 2 = "1.23" THEN
    #my_var = FLOAT(SERIAL_POINTER 2) * 100
ENDIF

```

If you don't specify the port it will default to port 1.

This port infix can also be used in Modbus master mode:

```

MODBUS_MASTER_MACRO:
// read remote CH3
MODBUS_READ 2 (5, addr(&CH3), &RESULT)
// write to remote CH3
MODBUS_WRITE 2 (5, &RESULT, addr(&CH3))
END

```

For multiple display meters (DI503, DI602 and DI802) the WRITE command has also been extended to directly address the display you want to write to:

```

IF |SP1=ON THEN
    WRITE 1 " __ACTIVATED " // write to top display
ELSE
    WRITE 2 " __DEACTIVATED " // write to bottom
ENDIF

```

Again if you don't specify it, the default display will be used.

For dual line displays the default is display 2 (bottom). For the DI503 the default display is 1 (Top).



It is not possible to scroll messages simultaneously on more than one display at a time.



In edit mode you cannot write to display 1 as it is used for the edit value.

There is no infix notation for the APPEND command, but any appended message will be added to the write string in the normal manner and will be directed to the display specified in the preceding write command.

## Lesson 42 - The INCLUDE Command

As was mentioned before, you should use CONST definitions and re-define some register names (especially when it comes to USER\_MEMORY registers) to improve the readability of your code.

Whether you work on more than one project, or create variations of the same application, it is easier to use existing code or definitions in more than one macro.

For this purpose you can INCLUDE another basic file like this:

```
REM DualPump.1.0.bas
REM
REM last update 2004/07/22

CUSTOMER_ID_MACRO:
write "          Dual Pump 1.0          "
END

INCLUDE "PumpDefs.bas"

...
```

This command performs a simple textual include - it works just like a copy&paste at the position where the INCLUDE is. Therefore the included file could also contain a subroutine or a predefined macro. But you have to be careful when you make changes to the included file as you have to consider its effects for all macros that include the file.

You can have up to 10 nested includes, i.e. file1 includes file2 that includes file3 that ... that includes file10.



## Lesson 43 - Data Logging

The data logging function of the Tiger series is usually triggered by a special event like a button press or a setpoint status. This function is built into the operating system and is easy to setup with the Texmate Configuration Utility.

However, if you want to trigger a log sample only when several conditions are met, it is often easier to do this with a macro. Let's see how easy that is:

```
MEM &CODE8 = 0150 // enable data logging
CONST LOG_TIMEOUT = 10 // every 1.0 sec

MAIN_MACRO:
Line 5   if |SP1 = on and |SP2 = on then
Line 6     if &TIMER1 > LOG_TIMEOUT then
           &TIMER1 = 0 // reset timer
Line 8     FORCE_LOG // triggers a log sample
           endif
           else
           // set the timer to timeout value to trigger the
           // first sample as soon as both setpoints come on
Line 13   &TIMER1 = LOG_TIMEOUT
           endif
           END
```

When Setpoint 1 and 2 are on (line 5), a log sample is taken every second (line 6). To make sure that we get a sample right after both setpoints turn on, the timer is kept at the timeout value (line 13).

The FORCE\_LOG command in line 8 causes the meter to take a log sample. This sample uses the standard data logging setup, only in this case the trigger is the macro.



Internally, the FORCE\_LOG command just assigns a value to the &SINGLE\_LOG register, which ignores the value and triggers a log sample. However, this register has different types in the Tiger 320 (integer) and the Tiger 380 (string).

Therefore we recommend to always use the FORCE\_LOG command instead of assigning a value to the &SINGLE\_LOG register.

## Lesson 44 - USER\_MEMORY vs. USER\_MEMORY\_BYTE

Just as in the Tiger 320 series meters, there are 1024 USER\_MEMORY registers available in the Tiger 380 series - even with the same register numbers from 5121 to 6144. These are 16bit signed registers.

However, there are some applications where this arrangement is not very convenient. For example, when using recipes you may only require 8bit values, but you may need more of them. Or you may want to store several linearization tables, which use 24bit signed numbers. In that case you have to split up the 24bit number and store it in two 16bit registers, wasting 8bits per value.

Unfortunately there is no general solution for all purposes. So we decided to give the most basic access to the USER\_MEMORY in the form of 8bit unsigned registers. These are called USER\_MEMORY\_BYTE with register numbers 10241 to 12288.



But even though the register numbers are different, the physical memory is the same:

USER\_MEMORY\_BYTE\_1 and USER\_MEMORY\_BYTE\_2 are stored in the same location as USER\_MEMORY\_1, USER\_MEMORY\_BYTE\_2047 and USER\_MEMORY\_BYTE\_2048 are stored in the same location as USER\_MEMORY\_1024.

So you have to be careful if you use both USER\_MEMORY and USER\_MEMORY\_BYTE in the same macro that the actual addresses you are using do not overlap - unless you really want that.

```
MEM &USER_MEMORY_BYTE_1 = 1 //
MEM &USER_MEMORY_BYTE_2 = 0 // => USER_MEMORY_1 =
// 1 * 256 + 0 = 256
MEM &USER_MEMORY_2 = 258 // => USER_MEMORY_BYTE_3 = 1
// USER_MEMORY_BYTE_4 = 2
MEM &USER_MEMORY_3 = -1 // => USER_MEMORY_BYTE_5 = 255
// USER_MEMORY_BYTE_6 = 255
```



In the Tiger 380 series the USER\_MEMORY area is located in RAM and is backed up to FLASH at power down. This means there is no longer any restrictions on the number of writes to USER\_MEMORY with the 380. (The 320 series meters are still restricted to 100,000 writes for USER\_MEMORY).

## Lesson 45 – The GOTO Command

The last command we want to look at is the `GOTO` command. Although this type of command is generally frowned upon in most programming circles, it is sometimes useful so it is included as well.

The `GOTO` command allows a macro to jump from its current position to another position anywhere in the program. Unlike the `GOSUB` instruction, this is a one way jump. It is sometimes used to reduce the size of a program by jumping to common routines in the macro. Here's an example of how it is used:

```
MAIN_MACRO:
#TEMP = &CH2
REPEAT
  IF &CH1 > 10000 THEN
    GOTO ERROR

  ELSIF &CH2 > 10 THEN
    GOTO ERROR

  ELSE
    &CH1 = &CH1 * &CH1
    #TEMP = #TEMP - 1
  ENDIF
UNTIL #TEMP <= 1
END

ERROR:
WRITE "    ___Over Range    "
END
```

The instruction `GOTO ERROR` is used twice in this macro. In each case if a test is found to be true, the program jumps to the label defined as `ERROR`. Near the bottom of the macro you will find the instruction `ERROR:.` The `:` at the end tells the compiler that the word `ERROR` is a label, which defines an address or point in the program.

When the macro encounters a `GOTO ERROR` instruction, it knows that it has to jump to this point.

That's all there is to the `GOTO` instruction really. The reason it is frowned upon is because it destroys the structure of a program and can make it difficult to read and debug.



There are some rules to observe with labels. The first character in a label cannot be a number. It must be a letter or an underscore (i.e. "\_"). A label must be placed by itself against the left hand margin on a new line.

# Appendix

# A

---

Macro Names . . . . .	.2
TDS Commands . . . . .	.4
Key Words . . . . .	.6
Operators with IF THEN Instruction . . . . .	.7
Pre-defined Values . . . . .	.7
Higher Maths Operators . . . . .	.8
Simple Maths Operators . . . . .	.8
ASCII Characters . . . . .	.9
Simple Logic Operators . . . . .	.9
Glossary of Terms . . . . .	.10

## Macro Names

The macro name or label may only be declared once in each macro. The default color of macro labels in the **Source Code Editor** is Teal, but can be any color you wish if you change the default setting in the **Syntax Highlighting** dialog box. Remember, it is good practice to keep the color of your macro labels different from any other text.

All macro names must be followed by a colon, and each macro name requires an **END** command. The compiler provides the following pre-defined macro names.

---

### Macro Name

### Macro Function

#### CUSTOMER\_ID\_MACRO:

Macro code placed in this section is executed following the display of the meter model and software version number, which is accessed by pressing PROGRAM, UP, and DOWN buttons at the same time.

Its intended use is to scroll a text string across the display that describes the customer's details: application, macro version number, etc, but it can contain any source code. While the use of this macro is optional, it is highly recommended as it is often the only way of determining whether a meter is running a macro or not.

#### MAIN\_MACRO:

Macro code placed in this section is executed at the currently selected update rate (normally once every 0.1 seconds). It is executed after all the input channels have been processed but before the set-point logic and display are updated. This is probably the most widely used macro and would normally be used for any maths equations or other logic that needs to be applied to input data etc.



**EDIT\_MACRO:**

Macro code placed in this section is executed once, each time the **PROGRAM** button is pressed, while the meter is in Edit Mode. The meter is placed in edit mode by executing the **EDIT** command. This macro is used for editing data or setup information.

**F1\_BUTTON\_MACRO:**

Macro code placed in this section is executed once each time the **F1** button is pressed.

**F2\_BUTTON\_MACRO:**

Macro code placed in this section is executed once each time the **F2** button is pressed.

**F3\_BUTTON\_MACRO:**

Macro code placed in this section is executed once each time the **F3** button is pressed.

**RESET\_MACRO:**

Macro code placed in this section is executed once initially after power has been applied to the meter. It's intended use is to initialize any variables before any other macros are run.

**MODBUS\_MASTER\_MACRO:**

Macro code placed in this section is executed at the rate specified by the value of the **&POLL\_TIME** register (with a resolution of 0.01 secs). This macro is only available in meters of the Tiger 380 series. All Modbus commands must be invoked within this section as they usually take longer than one macro cycle. If a Modbus command is issued within this section the macro waits in the background for the response. See Lesson 40 for more information.

**VIEW\_MODE\_MACRO:**

\* *A detailed explanation of these functions is outside the scope of this tutorial. Contact Texmate if you wish to use these functions.*

\* Macro code placed in this section is executed each time the **UP** or **DOWN** button is pressed in the view mode, provided certain configuration parameters have been set.

**EDIT\_DOWN\_MACRO:**

\* Macro code placed in this section is executed each time the **PROGRAM** button is pressed, while the meter is in the setpoint editing mode, provided certain configuration parameters have been set.

**TDS Commands**

The TDS has a list of pre-defined commands that the compiler interprets as a request to perform a specific programming task. The default color for commands is blue, but again can be any color you wish. The compiler provides the following pre-defined basic commands.

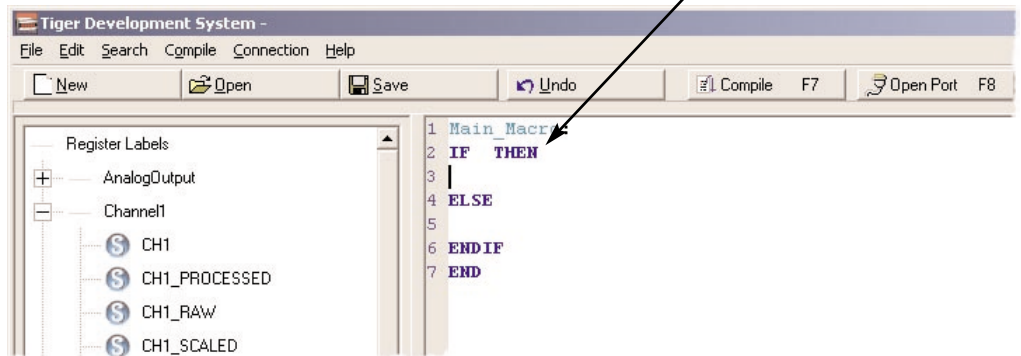
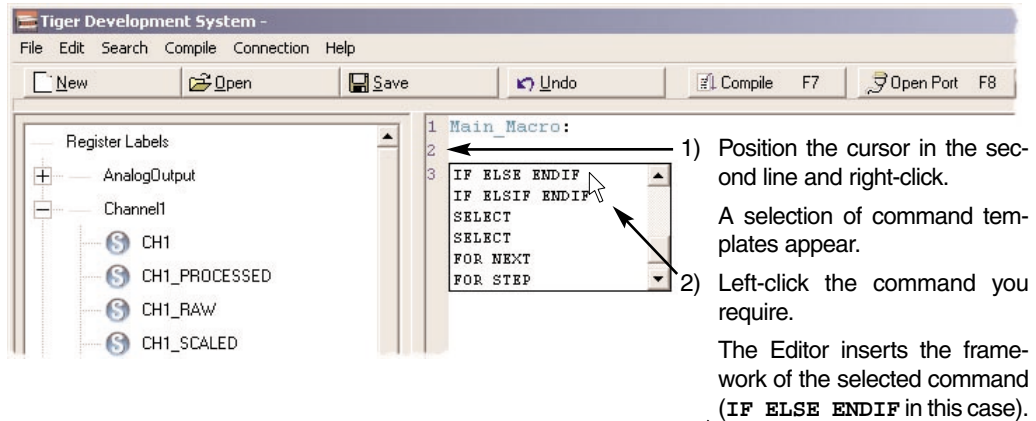


Commands are case sensitive as they can be either upper or lower case, but not mixed. Selecting a basic command from the source code editor (opened by right-clicking) automatically inserts the basic command in upper case.



**PROGRAMMING TIP**

To prevent typing errors affecting your macro, it is always good practice to use the **Templates** list to insert a command.



Command	Function
IF ENDIF	Selecting IF ENDIF inserts an <b>IF THEN</b> instruction followed by an <b>ENDIF</b> instruction. <i>For full details, see Lesson 2.</i>
IF ELSE ENDIF	Selecting IF ELSE ENDIF inserts an <b>IF THEN</b> instruction, followed by an <b>ELSE</b> instruction, followed by an <b>ENDIF</b> instruction. <i>For full details, see Lesson 3.</i>

**IF ELSIF ENDIF**

Selecting IF ELSIF END inserts an **IF THEN** instruction, followed by an **ELSIF** instruction, followed by an **ELSE** instruction, followed by an **ENDIF** instruction.

*For full details, see Lesson 4.*

**SELECT**

Selecting SELECT inserts a **SELECT** instruction, followed by two **CASE :** statements, followed by an **ENDSEL** instruction.

*For full details, see Lesson 21.*

**SELECT DEFAULT**

Selecting SELECT DEFAULT inserts a **SELECT** instruction, followed by two **CASE :** statements, followed by an **DEFAULT** instruction, followed by an **ENDSEL** instruction.

*For full details, see Lesson 21.*

**FOR NEXT**

Selecting FOR NEXT inserts a **FOR = TO** instruction, followed by an empty line to insert a block of instructions and then a **NEXT** instruction to complete the loop.

*For full details, see Lesson 24.*

**FOR STEP**

Selecting FOR STEP inserts a **FOR = TO STEP** instruction, followed by an empty line to insert a block of instructions and then a **NEXT** instruction to complete the loop.

*For full details, see Lesson 24.*

**REPEAT**

Selecting REPEAT inserts a **REPEAT** instruction, followed by an empty line to insert a block of instructions and then an **UNTIL** instruction to complete the loop.

*For full details, see Lesson 23.*

**Key Words**

Key (or reserved) words appear in bold blue and cannot be used for variable names. The compiler provides the following key words:

Command	Explained in Lesson	Command	Explained in Lesson
ABS()	31	INCLUDE	42
ADDR()	28	INTEGER()	39
AND	7, 32	LET	
APPEND	4, 18, 30	LN	31
ARCCOS	31	LOG	31
ARCSIN	31	MEM	29
ARCTAN	31	MODBUS_READ()	40
ASC()	35	MODBUS_WRITE()	40
BIT	15	NEXT	24
BITREG	27	OR	7, 32
CASE	21	PRINT	34, 41
CHR()	28	REG	15
COS	31	REM	1
COSH	31	REPEAT	23
CONST	11	RETURN	22
DEFAULT	21	SCALE	33
DIM	26	SELECT	21
EDIT	13	SERIAL_INPUT	36
EDIT_NUMERIC	20	SERIAL_POINTER	39
EDIT_TEXT	20	SET	
ELSE	3	SIN	31
ELSIF	4	SINH	31
END	1	SQR	31
ENDIF	2	STEP	24
ENDSEL	21	TAN	31
EXIT_EDIT	13	TANH	31
FLOAT()	39	THEN	2
FOR	24	TO	24
FORCE_LOG	43	UNTIL	23
GOSUB	22	WRITE	18, 30, 41
GOTO	45	XOR	32
HEX()	39		
IF	2, 25		

**Pre-defined Values**

The compiler provides the following pre-defined values. These are one bit wide pre-defined bit flags. For an explanation on the function of the pre-defined bit flags, see Lesson 5:

---

**Pre-defined Values**

NORMAL

OFF, FALSE

ON, TRUE

**Operators with IF  
THEN Instruction**

Following is a list of operators that can be used with the `IF THEN` instruction:

---

'='	If equal to
'<'	If less than
'>'	If greater than
'>='	If greater than or equal to
'<='	If less than or equal to
'<>' or '!='	If not equal to

### Simple Maths Operators

You can use any of the following simple maths operators in your macro. All of these operators require two operands:

---

'+'	Addition
'-'	Subtraction
'*'	Multiplication
'/'	Division
'()'	Parentheses
'^'	Power of



Depending on the complexity of the equations within the parenthesis, up to 10 levels of parenthesis can be used.

### Higher Maths Operators

Unlike the simple operators, these higher maths operators only require one operand. The following list shows the higher maths operators available and the allowable input range:

---

<b>ABS()</b>	Absolute (positive) value of expression in parentheses
<b>SQR</b>	Square root (positive number)
<b>LN</b>	Natural log (base e)
<b>LOG</b>	Common log (base 10)
<b>SIN</b>	Sine (-65535 to +65535)
<b>COS</b>	Cosine (-65535 to +65535)
<b>TAN</b>	Tangent (-65535 to +65535)
<b>SINH</b>	Hyperbolic sine (-65535 to +65535)

<b>COSH</b>	Hyperbolic cosine*
<b>TANH</b>	Hyperbolic tangent*
<b>ARCSIN</b>	Arc sine (-1 to +1)
<b>ARCCOS</b>	Arc cosine (-1 to +1)
<b>ARCTAN</b>	Arc tangent*

\* These functions accept a value in the range of  $\pm 1.175494E - 38$  to  $\pm 3.402823E38$ .

### Simple Logic Operators

The TDS includes the following simple logic operators. All of these operators require two operands:

---

<b>AND</b>	Logical And
<b>OR</b>	Logical Or
<b>XOR</b>	Logical Exclusive Or

### ASCII Characters

The TDS uses the following special ASCII characters:

---

<b>CR</b>	Carriage Return
<b>LF</b>	Line Feed
<b>BS</b>	Back Step
<b>NUL (or NULL)</b>	Null character
<b>ESC</b>	Escape character
<b>TAB</b>	Tab character
<b>FF</b>	Form feed
<b>VTAB</b>	Vertical tab
<b>BELL</b>	Bell character



**Register Type Flags  
for Modbus com-  
mands**

The Modbus commands INTEGER(), FLOAT(), and HEX() have an optional fourth parameter to explicitly specify the register type. The following list shows the possible values for this parameter:

---

8 bit: MB\_BYTE

16 bit: MB\_SHORT

24 bit: MB\_24, MB\_24\_SWAPPED

32 bit: MB\_LONG, MB\_LONG\_SWAPPED, MB\_FLOAT,  
MB\_FLOAT\_SWAPPED

**Modbus Error Flags**

The register &MODBUS\_MASTER\_FLAGS contains the standard Modbus exception codes as well as transmission errors detected by the master meter:

---

**MODBUS\_MASTER\_FLAGS**

bit 0 - bit 3 = Exception errors (Modbus standard)

1 = illegal function

2 = illegal data address

3 = illegal data value

4 = slave device failure

5 = acknowledge

6 = slave device busy

7 = negative acknowledge

8 = memory parity error

bit 4 = message timeout

bit 5 = CRC receive error

bit 6 = data type error

bit 7 = reception complete or waiting for a new command

**Glossary of Terms**

Following is a glossary of terms relevant to writing and compiling macros:

---

<b>Term</b>	<b>Defintion</b>
ASCII	American Standard Code for Information Interchange. A standard that identifies the letters of the alphabet, numbers, and various symbols by code numbers for exchanging data between different computer systems.
Command	A pre-defined word or group of words that the compiler interprets as a request to perform a specific programming task.
Compile	To take a sequence of commands, written in a higher level language (such as BASIC), and translate these into an intermediate language (such as macro code), that can be run directly in a specific target system (such as the meter).
Editor	A software program that allows the user to create and edit a text file.
Instruction	A pre-defined word or group of words that the compiler interprets as a request to perform a specific programming task.
Macro Engine	The fixed software inside the meter that is responsible for interpreting and executing the macro code.
Non Volatile Memory	Used for permanent data storage. Data is retained when the device is turned off.
Operand	A quantity, function, or other entity that is to have a mathematical operation performed on it.

---

Term	Defintion
Operating System	Preprogrammed code inside the meter that controls the access to all meter functions and macro engine.
Operators	A mathematical symbol, term, or other entity that performs or describes an operation. Multiplication and subtraction signs are operators.
RAM	Random Access Memory. Used for temporary data storage. All data is lost when the device is turned off.
Source Code	A text file that contains commands and comments, which can be compiled into a working program to run in a target device.
Text String	A sequence of alpha-numeric characters, enclosed in quotation (" ") marks, that forms a word or sentence to be displayed or printed.
Variable	A user defined symbol that represents an unspecified or unknown quantity.

This Page Intentionally Left Blank

This Page Intentionally Left Blank