iDev Programming Guide

Written by Austin Barlis

(INCOMPLETE) Revision Date: 21/09/2012 Revised by: Austin Barlis

TABLE OF CONTENTS

Introduction 6
The iDev Language 6
Getting Started 6
1. Starting an iDev Project 7
1.1. What you need? 7
1.2. Main File 7
1.3. The iDev Format and Command Structure 7
1.4. Including Sub files 8
1.4.1. Include Example 8
1.4.2. Library Example (Images) 10
1.4.3. Library Example (Fonts and Sound files) 12
1.5. Setup (System) 14
1.6. RESET 15
2. Creating Pages 16
2.1. Setting up Page (Page Style) 17
2.2. Positioning Components 19
2.3. Defining Components 20
2.3.1. Text Style 20
2.3.2. Text Component 22
2.3.3. Text Manipulation 24
2.3.4. Image Style 28
2.3.5. Image Component 30
2.3.6. Image Manipulation 33
2.3.7. Draw Style 35
2.3.8. Draw Component 37
2.3.9. Draw Manipulation 39
2.3.10. Key Style 41
2.3.11. Key Component 42
2.3.12. Key Manipulation 46
2.4. Updating Components 49
2.5. Page Components Manipulation 52
2.5.1. SHOW 52
2.5.2. HIDE 53
2.5.3. DEL 53

2.5.4. Update Style – LOAD 54
2.6. Functions 57
2.7. Loop 58
2.8. Navigation between pages (linking) 59
3. Manipulating Data 63
3.1. Data Storage 63
3.1.1. Variable Data Style 63
3.1.2. Declaring Variables 65
3.1.3. Text Variable Update – LOAD 66
3.1.4. Integer/Float Variable – LOAD 70
3.1.5. Pointer 73
3.1.6. Array 78
3.2. Formatting Data 98
3.3. Moving and Updating Data – LOAD 103
3.4. Comparing Data or Creating Conditions – IF 105
3.5. Case – Switch/Select 113
3.6. Calculation 120
3.6.1. Arithmetic 120
3.6.2. Text Strings 126
3.6.3. Data Buffers 136
3.6.4. Other Calculation methods (Incomplete) 139
3.7. Counters 146
3.7.1. I/O Counters 146
3.7.2. Runtime Counter 147
3.8. Timers 149
3.9. Delay – WAIT 150
4. Interfaces and Communication 151
4.1. RS232 Interface 152
4.2. RS422/RS485 Interface 159
4.3. CMOS Asynchronous Interface (AS1, AS2, DBG) 167
4.4. SPI (Master and Slave) Interface 177
4.5. I2C/TWI (Master and Slave) Interface 186
4.6. Digital Input/Output (I/O) Interface & External Keyboard 194
4.7. Interrupts 203
4.8. Handling Data in Interfaces – LOAD 207
4.9. Controlling I/O Interface and External Keyboard (Incomplete) 208

5. Controlling PWM, ADC and Piezo Buzzer 213
5.1. PWM 213
5.2. ADC 217
5.3. Piezo 220
6. Real Time Support (RTC and RTA) 222
6.1. Real Time Clock (RTC) 222
6.2. Real Time Clock Alarm (RTA) 227
7. File handling for SD/micro SD Card and NAND – FILE (Incomplete) 229
7.1. APPEND 230
7.2. CLOSE 231
7.3. COPY 231
7.4. DATE 232
7.5. DELETE 232
7.6. EXISTS 233
7.7. GETPOS 233
7.8. MKFN 234
7.9. OPEN 234
7.10. READ 235
7.11. READALL 236
7.12. RENAME 237
7.13. SAVE 237
7.14. SETPOS 238
7.15. SIZE 239
7.16. WRITE 239
7.17. WRITEALL 240
7.18. File Object Variable 241
7.19. File Result 241
7.20. File and Directory NAmes 242
7.21. Potential Future Commands (not yet supported) 243
7.22. File Examples 243
8. File Transfer and Memory 245
8.1. Transfer via micro SD Card 245
8.2. Transfer via SD Card or micro SD Card Adaptor 246
8.3. Transfer to NAND – FPROG & LOAD 248
8.4. Transfer via USB 253
8.5. EEPROM 257

9. Example Codes (Incomplete) 258
10. Glossary 259
11. Appendix 266
12. Accessories (Incomplete) 266
12.1. CANBUS Adaptor 266
12.2. Capacitive Touch 266
12.3. Rotary Encoder 266
12.4. Battery Connector 266
12.5. External Soundcard 266
13. Command Format Archive (Incomplete) 267
14. Image Files Used in Guide 276

INTRODUCTION

THE IDEV LANGUAGE

iDev is a language developed by Noritake-Itron UK Engineers to manipulate and control the Itron Smart TFT displays. The itron SMART TFT displays simplifies the application of TFT technology into different products. The itron SMART TFT displays have a built-in microcontroller and interfaces which provides extensive functionality. The iDev language has some aspects present on multiple pre-existing languages namely: C, HTML and Basic. This won't mean anything for inexperienced developers but this gives experienced developers an idea on the layout and structure of the iDev language. The iDev language achieves simplification of programming by creating the least amount of commands possible but still utilise the full capability of the module.

GETTING STARTED

This guide is created to educate beginners in iDev development. If you do not have any experience in programming before, there is no need to worry because this guide is aimed for complete beginners in programming. As a developer myself, I have read different beginner's guides and tutorials before and I have always thought that there are always some parts missing that I don't fully comprehend after finishing the whole guide. This guide has a glossary page that explains advanced terms used in this guide. If you are unsure on what a specific term means then go to the glossary page of this guide for a full explanation of the term concerned. I will reassure you that after completion of this guide you will know everything that is needed to know about iDev programming.

1. STARTING AN IDEV PROJECT

1.1. WHAT YOU NEED?

An itron SMART TFT module is not necessarily needed but having one would greatly enhance your learning experience because connections from the module via the interface to an external module such as a sensor would be possible. If you have an itron SMART TFT module, then downloading the iDevTFT development software (link here) is not needed to observe the code you created because it can be uploaded on the device via a micro SD card. Any text editor can be used to create the program for the itron SMART TFT module. However, downloading the iDevTFT software is greatly recommended because it has features that would greatly benefit not only beginners in the iDev language but also others who are experienced enough.

1.2. MAIN FILE

Projects are uploaded on the TFT module by the use of iDev nomenclature/classification. When the TFT module is powered, it looks for a specific file name and type on which the program/code is written on to. The filename structure is based on the device's product code followed by a suffix of *.mnu* which sets the file as the main menu file. The product code of a typical 4.3" TFT module is TU480x272… Names of main menu files are based on the TFT module's product code; hence the menu file for a 4.3" TFT module is called *TU480a.mnu* eg. A 7" TFT module's part number is normally TU800x480… so the menu file would be *TU800a.mnu* and so on. Menu files can be created on any text editor program so a Microsoft equivalent would be Notepad. From here on, the "*TU480a.mnu*" file will be addressed as the main menu file.

1.3. THE IDEV FORMAT AND COMMAND STRUCTURE

In the course of this guide, various iDev commands will be used to perform certain tasks that a developer would want to do. Also different developers have different writing styles in software development. There are a few things that an iDev developer should remember when creating iDev projects:

a) **library file names, page names, page component names, style names, function names and variable names are case sensitive, however iDev commands, setup parameters and style parameters are case insensitive** – it does not matter whether the commands and parameters are all in upper case or lower case or a mixture of both
b) **names for library files, pages, page components, styles, functions and variables must start with _ or letter** – there is however, naming conventions that most developers use to help for identification and better referencing in any languages which can be applied in iDev
c) **iDev uses semicolons as a termination character for each command and style/setup parameters** – the use of semicolons and brackets in any programming languages is always abundant and important because sometimes one missing semicolon or bracket can be the cause of why an iDev project is not working properly

d) **adding comments is important** – comments are added in iDev by adding "//" before the comment in the menu file, all example codes in this guide will be commented for better assistance.
e) **text data, calculation methods and file locations should be enclosed in "** – e.g *VAR(mytext, "Hello",TXT);*

The use of semicolons and brackets can be compared to using commas, full stops and exclamation mark in the English language, the use of punctuation marks in English language completes the grammar of a sentence much like in iDev, the use of semicolons and brackets completes the format of the code.

1.4. INCLUDING SUB FILES

Menu files are normally created to organise code created by the developer on iDev. Instead of putting the entire developer's code in one big main menu file, it can be divided into different sub files. Typically menu files for pages, functions, variables and styles are created. Developer specified images and fonts are added as well. As the TFT module looks for the main file when turned on, at least one "include" instance should be used in the main menu file. When adding images, fonts or sound files to the project however, the *LIB* command is used. The current system does not yet recognise directory structures in the SDHC card so it is important to remember that all active files used for the iDev project have to be placed in the root folder of the SDHC card.

1.4.1. INCLUDE EXAMPLE

In this example, menu files stated will be added to the iDev Project.
Main menu file for 4.3" module: *TU480a.mnu*
Other menu files to be added: *Pages.mnu*
Funcs.mnu
Vars.mnu
Styles.mnu

INC command format:
For single files
INC("Source/Filename") ;
For multiple files
INC("Source/Filename1", "Source/Filename2", "Source/Filename3"…);

If the developer wants to add the other menu files to the main menu file then the following lines of code have to be added:

```
//FILENAME: TU480a.mnu

INC("SDHC/Pages.mnu");      //add pages menu file to the main menu file
INC("SDHC/Functions.mnu"); //add functions menu file to the main menu file
INC("SDHC/Variables.mnu"); //add variables menu file to the main menu file
INC("SDHC/Styles.mnu");     //add styles menu file to the main menu file

//OR

INC("SDHC/Pages.mnu","SDHC/Funcs.mnu","SDHC/Vars.mnu", "SDHC/Styles.mnu");
//add all the other menu files in one line
```

Fig.1.1 *Adding the other menu files to the main menu file*

In Fig 1.1, the source parameter is set to “SDHC” because the menu files that are going to be included are stored on the SD card. In some cases when the menu files being added are stored in the NAND flash then the source parameter is changed to “NAND”.

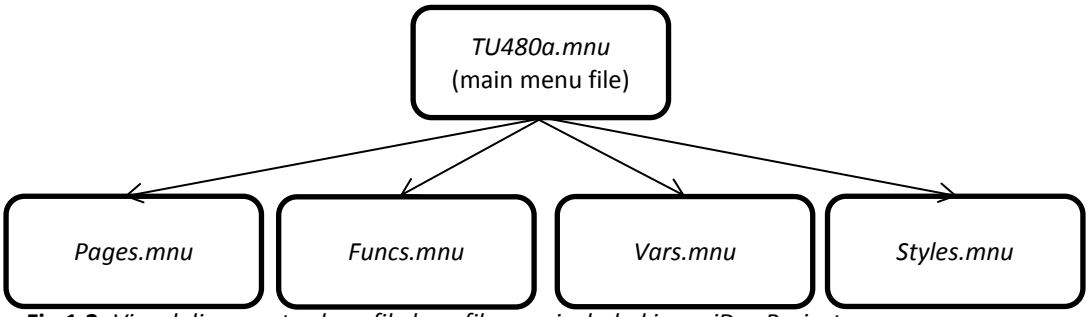


Fig.1.2. *Visual diagram to show file how files are included in an iDev Project*

Another way of including files is by nesting them. As stated before, at least one include instance in the main menu file has to be done. In this example, the developer wants to add *Pages.mnu* and *Vars.mnu* in the main menu file. Then include the *Funcs.mnu* file to *Pages.mnu* and the *Styles.mnu to Vars.mnu*. The following lines of code have to be added to the appropriate menu files.

```
//FILENAME: TU480a.mnu

INC("SDHC/Pages.mnu"); //add pages menu file to the main menu file
INC("SDHC/Vars.mnu");  //add variables menu file to the main menu file

//OR

INC("SDHC/Pages.mnu","SDHC/Vars.mnu");
//add both pages and variables menu file in one line
```

Fig.1.3. *Pages.mnu and Variables.mnu files are added to the main menu file*

```
//FILENAME: Pages.mnu

INC("SDHC/Funcs.mnu"); //add the Functions.mnu file to Pages.mnu file
```

Fig.1.4. *Funcs.mnu file is added to the Pages.mnu file*

```
//FILENAME: Vars.mnu

INC("SDHC/Styles.mnu"); //add the Styles.mnu file to Variables.mnu file
```

Fig.1.5. *Styles.mnu file is added to the Vars.mnu file*

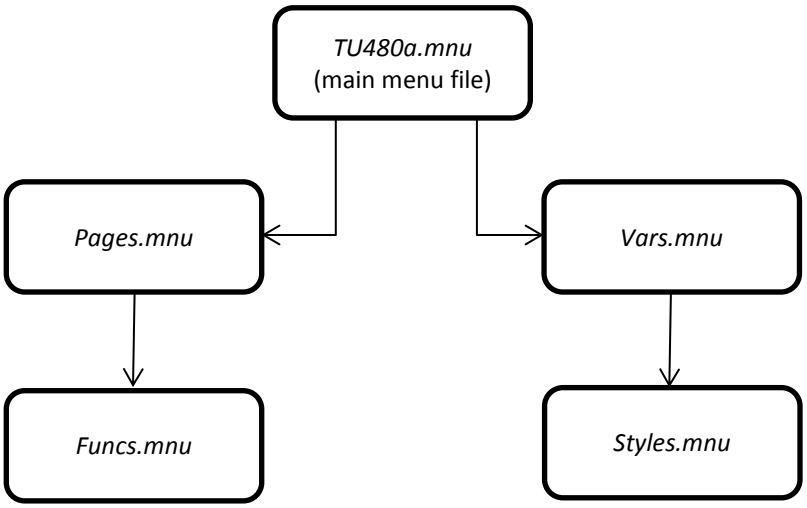


Fig.1.6. *Visual diagram to show file how menu files are nested to each other in an iDev Project*

In this second example the menu files are nested on 2 levels i.e. *Vars.mnu* is included in *TU480a.mnu* (1st level) and *Styles.mnu* is included in *Vars.mnu* (2nd level). The iDev language allows up to 7 levels of include instances.

1.4.2. LIBRARY EXAMPLE (IMAGES)

Developer images can be stored to the project library by using the LIB command. The iDev language supports BMP and JPG image file formats.

LIB command format for images:
LIB(Library image name, "Source/Filename");

In this example, 3 images will be added to the project library:

```
//FILENAME: TU480a.mnu

LIB(myimage1,"SDHC/image1.bmp"); //add image1.bmp to project library
LIB(myimage2,"SDHC/image2.bmp"); //add image2.bmp to project library
LIB(myimage3,"SDHC/image3.bmp"); //add image3.bmp to project library
```

Fig.1.7. *Adding image files to project library using LIB command*

In Fig 1.7 the BMP and JPG image file format does not support transparency; if the developer needs transparency in the image being used, and then additional parameters are added to the LIB command format as shown below:

LIB command format for transparency:
LIB(Library image name, "Source/Filename?back=Colour in HEX");

The example below would use the same images in Fig 1.7 and set specified colours to be transparent.

```
//FILENAME: TU480a.mnu

LIB(mylibimage1,"SDHC/image1.bmp?back=\\FFFFFF");
//add image1.bmp to library and set the HEX colour FFFFFF to be transparent
LIB(mylibimage2,"SDHC/image2.bmp?back=\\FF0000");
//add image2.bmp to library and set the HEX colour FF0000 to be transparent
LIB(mylibimage3,"SDHC/image3.bmp?back=\\FFFF00");
//add image3.bmp to library and set the HEX colour FFFF00 to be transparent
```

Fig.1.8. *Adding image files to project library using LIB command and setting the image's transparency*

In Fig 1.8, transparencies are set for the three images with different colours. For myimage1 the transparency parameter is set to *FFFFFF* in HEX code which refers to the white pixel colour. So when the image called image1.bmp is used in the iDev project concerned all the white pixels are set to be transparent. For myimage2 and myimage3, similar effect occurs but *FF0000* (red) is set to be transparent in image.2bmp and *FFFF00* (yellow) is set to be transparent in image3.bmp. In the examples given above, BMP file formats are used rather than JPG. Although transparency can be applied to JPG images as well, it is highly recommended to use BMP file formats when applicable. The JPEG file format uses lossy compression to significantly reduce the image's file size; this means that there is a reduction in the original image's quality. Using JPEG images may not provide accurate transparency capability and is only suitable for backgrounds. The advantages of using JPEG files are: it decreases loading time and smaller file size.
If the developer requires rotation for the image stored in the library, the rotate parameter is added:

LIB command format for rotation:
LIB(Library image name, "Source/Filename?rotate=0°, 90°, 180° or 270°");

The example below would use the same images in Fig 1.7

```
//FILENAME: TU480a.mnu

LIB(mylibimage1,"SDHC/image1.bmp?rotate=90");
//add image1.bmp to the library and set the image to rotate by 90 degrees
LIB(mylibimage2,"SDHC/image2.bmp?rotate=270");
//add image2.bmp to the library and set the image to rotate by 270 degrees
```

Fig. 1.9 *Adding image files to project library using LIB command and setting the image's rotation*

So from Fig 1.9, the library image myimage1 is set to have a rotation of 90°. In the iDev project, when myimage1 is used the image is rotated by 90°. This is the same for the library image myimage2 but instead a rotation of 270° is used. Another transformation that can be done to library images is scaling. The image stored in the library can be scaled higher or lower. The scale parameter is added:

LIB command format for scaling:
LIB(Library image name, "Source/Filename?scale=value");

Scale Value	Scale Definition
integer value	set the scaling of the library image
fit	set a non-proportional fit into the height and width specified, otherwise the size of screen is assumed
min	set a proportional fit into the width and height specified by developer
max	set a proportional fit into high value of width and height specified by developer (not implemented)
FitX	set a fit to width specified by developer (not implemented)
FitY	set a fit to height specified by developer (not implemented)

Fig. 1.10 *Table to explain what each scale value parameter means*

The example below uses the same images in Fig 1.7

```
//FILENAME: TU480a.mnu

LIB(mylibimage1,"SDHC/image1.bmp?scale=200");
//
LIB(mylibimage2,"SDHC/image2.bmp?scale=min&width=220&height=90");
//
```

Fig. 1.11 *Adding image files to project library using LIB command and setting the image's scaling*

Fig 1.11 shows how the scaling of the library image *myimage1* is increased to 200%. The library image *myimage2* is set to have a proportional fit into a 220 by 90 dimension. There is no absolute limit set when scaling images. When multiple transformations are required, then a different format has to be used:

LIB command format for multiple transformations:
LIB(Library image name, "Source/Filename?transformation1&transformation2..");

The example below uses the same image in Fig 1.7

```
//FILENAME: TU480a.mnu

LIB(mylibimage1,"SDHC/image1.bmp?scale=200&back=\\FF0000");
//
LIB(mylibimage2,"SDHC/image2.bmp?scale=min&rotate=180&back=\\FFFFFF");
//
```

Fig. 1.12 *Adding image files to project library using LIB command and applying multiple transformations to each image*

All the transformations can be applied at once using the LIB command. Multiple transformations are performed with the use of "&" as evident in Fig 1.12.

1.4.3. LIBRARY EXAMPLE (FONTS AND SOUND FILES)

Adding developer fonts and sound files is done in a similar way as to add images. The iDev language supports FNT font file format and WAV sound file format. Although there is no size limit when adding sound files to the project library, it is

highly recommended to use smaller sound files. The use of larger sound files significantly increases the loading time of the TFT module when it is powered on i.e. boot-up time is increased. If a large sound file is required however, quicker start-up of the device can be achieved by only loading the sound file on demand and setting a flag to indicate if they have been loaded.

LIB command format:
LIB(Library font/sound name, "Source/Filename");

The example below is showing how to add 2 font files and a sound file.

```
//FILENAME: TU480a.mnu

LIB(mylibfontasc24,"SDHC/asc24.fnt");    //
LIB(mylibfontasc24b,"SDHC/asc24b.fnt"); //
LIB(mylibsound,"SDHC/sound1.wav");       //
```

Fig 1.13 *Adding 2 font files and a sound file to the project library*

Adding multiple font files may sometimes require mapping of hex codes to specific characters to display appropriate characters. This is done by adding another parameter to the LIB command format when adding a font file. The example below adds 2 font files with different mapped starting HEX values.

LIB command format for mapping fonts:
LIB(Library font name, "Source/Filename?start=HEX value to be mapped");

```
//FILENAME: TU480a.mnu

LIB(mylibfontasc24,"SDHC/asc24.fnt?start=\\0080");
//
LIB(mylibfontasc24b,"SDHC/asc24b.fnt?start=\\105F");
//
```

Fig 1.14 *Adding 2 font files with different mapped starting values*

By mapping the correct values for multiple font files, the developer can then use multiple fonts in the same text STYLE which will be explained further in the text part of this guide.

1.5. SETUP (SYSTEM)

This feature of the iDev language is mainly involved in debugging iDev projects. Certain system parameters can be changed by the developer in main menu file. The system setup command in iDev consists of two parts namely "*Setup Header*" and "*Setup Body*".

SETUP command format:
Setup Header
SETUP(SYSTEM)
Setup Body
{
parameter1 = parameter value1;
parameter2 = parameter value2;
parameter3 = parameter value3;
…
}

System Setup		
Parameter	**Expected Values**	**Definition**
startup	all	display messages and bar at start up **(default)**
	bar	display loading bar at start up
	none	don't display anything at startup
bled	0 (OFF) to 100 (ON)	set backlight level of TFT module, 0 being OFF and 100 to be highest backlight brightness level **(default = 100)**
wdog	0 to 16000 milliseconds	set the watchdog time out period in milliseconds **(default = 16000)**, watchdog timeout is used to prevent the processor from hanging or latching up
rotate	0	set orientation of the screen with respect to PCB, 0 – don't rotate the screen **(default)**
	180	rotate the screen 180 degrees with respect to PCB
test	hideTouchAreas	hide touch areas **(default)**
	showTouchAreas	show touch areas during product development, useful for placement of key/touch actions
angles	degrees	use degrees in calculations **(default)**
	radians	use radians in calculations
encode	s	menu text strings contain single byte ASCII **(default)**
	w	menu text strings contain 2 bytes for UNIcode
	m	menu text strings contain multiple byes for UTF8
calibrate	y	start the internal touch screen calibration when turned on
	n	don't start touch screen calibration when turned on **(default)**

clkfreq	80000000 Hz to 92000000 Hz (in 2 MHz steps)	set the external bus clock **(default = 92 MHz)**
ignore	allErrors	ignore all error and continue execution, only recommended when in debugging and testing stages because it can cause undesired results **(default = ignore is disabled)**
	invalidJpg	ignore error for unsupported JPG formats (e.g. progressive) and the image is skipped
	imageTooBig	ignore errors when there is not enough memory to load image and the image is skipped

Fig. 1.15 *Table that describes the parameters that can be changed, expected values and definition of parameters for System Setup*

Based on Fig. 1.15 and the setup command format, an example system set up can be created:

```
//FILENAME: TU480a.mnu

SETUP(SYSTEM)            //
{
bled = 100;              //
wdog = 1000;             //
rotate = 0;              //
calibrate = n;           //
test = showTouchAreas;   //
angles = degrees;        //
startup = bar;           //
encode = s;              //
clkfreq = 92000000;      //
ignore = imageTooBig;    //
}
```

Fig 1.16 *System setup example*

Parameters are changed in the *Setup Body* and the *Setup Header* is used for setup recognition. Not all the parameters have to be specified in the *Setup Body*, if a parameter is not there, then the parameter value of the omitted parameter is set to its default value. Later on, the setup command will be used in changing the setup of interfaces; in the example above, only the system setup is altered.

1.6. RESET

In iDev, the *RESET* command can be used to clear the contents of Runtime counter, EEPROM or perform a system reset. The *RESET(SYSTEM)* command only works reliably (with no lock-up)in the most recent TFT module versions. For the 3.3" from version 2 onwards, 4.3" is version 6 onwards, 5.7" is version 4 onwards and 7.0" is version 5 onwards. The *RESET* command can be used in any parts of the iDev project e.g. inside a function or an action of a key component.

RESET command format:
RESET(Name of iDev property)

Name of iDev property	Definition
SYSTEM	reset the system which performs a re-boot at power ON
RUNTIME	clear the runtime counter
EEPROM	clear the EEPROM contents and reload redefined variables
NAND	clear the NAND flash memory
NANDMNU	clear the MNU files stored in NAND flash
NANDLIB	clear the BMP, FNT, WAV files stored in NAND flash
LIBRARY	clear the contents of the library
DELETED	clear the page components and iDev components added to the deleted list (not implemented)

Fig 1.17 *Table to describe which iDev properties the RESET command*

2. Creating Pages

In iDev, a page refers to a container for a group of components that is visible on the TFT module. There are 4 main types of components that can be added to a page namely: text component, image component, drawing component and key component. An iDev page is analogous to a blank piece of canvas, and the page's components are the drawings or sketches that an artist is adding to the canvas. As with the canvas, the artist can decide what colour, shape or form i.e. style of drawings to add, in iDev the developer can decide the component's style as well. Creating a page in iDev comprises of two parts, called "*Page Header*" and "*Page Body*".

PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
Page Components…
}

The *Page Header* gives specific pages that a developer will use in iDev for page identification. This is essential because almost all iDev projects will have more than one page, so the developer should know when and where to show a specific page. On the other hand, *Page Body* is where the developer states what components to be added in the page concerned. A *Page Header* consists of two parameters namely *Page name* and *Page style*. As the name suggests, *Page name* refers to the name that the developer has assigned for that specific page. *Page style* specifies what page style the specific page is going to use.

2.1. SETTING UP PAGE (PAGE STYLE)

It was demonstrated in the previous part of the guide on how to create a page, in this part of the guide setting up a style for page will be shown by introducing a new iDev language command. Setting the style of the page enables the developer to maintain a common theme throughout the project. Similar to creating a page and system setup, setting up the page style consists of *Style Header* and *Style Body*.

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}

Styles in iDev can be inherited from previously created syles as well. This enables the developer to create a style based on a different style if both styles are similar and only one or two style parameters are required to be different.

STYLE command format inherit:
Style Header
STYLE(New Style name, Style name inherit)
Style Body
{
new style parameter 1 = new style value 1;
new style parameter 2 = new style value 2;
new style parameter 3 = new style value 3;
…
}

Page Style		
Parameter	**Expected Values**	**Definition**
update	all	define page refresh method as all **(default)**, all the page components will be refreshed
	changed	define page refresh method as changed, only the specified page components will be refreshed
sizeX	(1 to 3) × LCD width	set the width of the page, can upscale the page up to three times larger than the LCD width **(default value depends on LCD's size e.g. 4.3" module default = 480 5.7" module default = 640)**
sizeY	(1 to 3) × LCD height	set the height of the page, can upscale the page up to three times larger than the LCD height **(default value depends on LCD's size e.g. 4.3" module default = 272 5.7" module default = 480)**
posX	(−4 to 4) × LCD width	set the X position/coordinate of the page relative to the screen **(default = 0)**

posY	(−4 to 4) × LCD height	set the Y position/coordinate of the page relative to the screen **(default = 0)**
opacity	0 to 100	set the opacity of the page **(default = 100)**, 100 is opaque (not transparent) and 0 is transparent
back	colour name or in HEX \\000000 to \\FFFFFF	set the background colour of the page **(default = black)**
image	library image name	set the background image of the page using the stored image in the project library **(default = none)**

Fig. 2.1 *Table that describes the parameters that can be changed, expected values and definition of parameters for Page Style*

Using the style command format and the table in Fig 2.1 an example page style can be defined:

```
//FILENAME: TU480a.mnu

STYLE(mypagestyle,Page) //
{
update = all;  //
sizeX = 480;   //
sizeY = 272;   //
posX = 0;      //
posY = 0;      //
back = white;  //
}
STYLE(mynewpagestyle,mypagestyle) //
{
back = red;    //
opacity = 75; //
}
```

Fig. 2.2 *Example to show how to define page style properties*

The *Style Header* provides the developer to specify and identify appropriate styles in the iDev project. This makes it easier for the developer to apply a specific style to another page that will be created. The *Style Body* consists of different parameters that change the features of the page. Style parameters help the developer design his/her project to how they want it to be. The style parameters that are not defined in the *Style Body* are set to its default values. Now that page style is introduced, in the next example, an image stored in the library will be used as the background of the page that is going to be created.

LIB command format:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}

PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
Page Components…
}

```
//FILENAME: TU480a.mnu

LIB(mylibimage1,"SDHC/image1.bmp"); //

STYLE(mypagestyle,Page) //
{
update = all;          //
image = mylibimage1;   //
}

PAGE(mypage,mypagestyle)   //
{
//Page Components…
}
SHOW(mypage);   //
```

Fig 2.3 *Example to show how to create a page, apply a background to the page by using the back parameter in the page style*

The example in Fig 2.3 shows a how a typical page is prepared in iDev. The system setup parameters were not defined in this example which means that all the values taken for the system parameters are default. Now that the page is created and its style is defined, page components need to be added to the page. The *SHOW* command is used to display pages and page components in iDev, this command will be explained thoroughly in Chapter 2.5.1.

2.2. Positioning Components

Adding page components require the developer to define the position of a page component being added and/or created. The use of positioning allows the developer to create a layout that they specifically wanted. The position command changes the cursor; the cursor is a non-visible indicator that identifies the point in the TFT module that will be affected by input from the developer. The *Page/Page Component* can be left blank if the developer only wants to change the cursor's position. The cursor can be repositioned in relation to its previous position by using + or – offset values.

POSN command format:
To change cursor position
POSN(x coordinate, y coordinate);
To reposition cursor position based on previous cursor position
POSN(+ /- x coordinate,+ /- y coordinate);
For single Page/Page Component
POSN(x coordinate, y coordinate, Page/Page Component);
For multiple Page/Page Components
POSN(x coordinate, y coordinate, Page1/Page Component1, Page2/Page Component2…);

In the code examples, the most common *POSN* command format used is to change cursor position. The other *POSN* command formats are only useful when moving pages or page components as shown in the example code below:

```
//FILENAME: TU480a.mnu

POSN(100,25,mypage);  //
POSN(100,80,mypagecomponent1,mypagecomponent2,mypagecomponent3);
//
```

Fig. 2.4 *Example code to show how POSN command format is applied*

2.3. DEFINING COMPONENTS

Each page component should have a style defined when being added to the page. As explained before, using styles enables the developer to keep a consistent theme throughout the developer's iDev project. The style command format is consistent in iDev; the only difference is the style type, parameter and values:

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}

2.3.1. TEXT STYLE

The text style describes the features that the developer require for the text component to be added to a page. The text style command is used in a similar way on how the page style command is used.

Text Style		
Parameter	**Expected Values**	**Definition**
font	library font name	state the font to be used by either a preloaded .FNT file or built in fonts: Ascii8, Ascii16, Ascii32 **(case-sensitive, default = Ascii16)**
size	positive integer values	set the font size of the text e.g. a 12x12 font is enlarged to 24x24 if font size is increased from 1 to 2 **(default = 1, max = 32)**
col	reserved words from colour chart or colour in HEX code	set the colour of the text **(default = white)**
back	reserved words from colour chart or colour in HEX code	set the colour of the background of the text component **(default = black)**
opacity	0 to 100	set the opacity of the text **(default = 100)**, 100 is opaque (not transparent) and 0 is transparent
maxLen	1 to 512	allocate the RAM for the maximum length of text component **(default = 32)**

maxRows	1 to 32	allocate the RAM for the maximum rows of text component **(default = 1)**, if more than 1, new line \\0D\\0A is used (usage explained in Chapter 2.3.3)
rotate	0	rotate the text component by 0° relative to the screen – don't rotate text **(default)**
	90	rotate the text component by 90° relative to the screen
	180	rotate the text component by 180° relative to the screen
	270	rotate the text component by 270° relative to the screen
curRel	CC	specify the placement/justification of text relative to the cursor to Centre Centre **(default)**
	TC	specify the placement/justification of text relative to the cursor to Top Centre
	BC	specify the placement/justification of text relative to the cursor to Bottom Centre
	RC	specify the placement/justification of text relative to the cursor to Right Centre
	LC	specify the placement/justification of text relative to the cursor to Left Centre
	TL	specify the placement/justification of text relative to the cursor to Top Left
	BL	specify the placement/justification of text relative to the cursor to Bottom Left
	TR	specify the placement/justification of text relative to the cursor to Top Right
	BR	specify the placement/justification of text relative to the cursor to Bottom Right
xTrim	Y or N	set the spacing in between each characters in text components, if yes spacing is spread evenly and no spacing is set to none – text is compressed **(default = Y)**

Fig. 2.5 *Table that describes the parameters that can be changed, expected values and definition of parameters for Text Style*

Using the table above and the style command format, an example to define a text component can be done:

```
//FILENAME: TU480a.mnu

STYLE(myfontAscii32,Text) //
{
font = Ascii32;  //
size = 2;        //
col = white;     //
opacity = 75;    //
maxRow = 1;      //
maxLen = 11;     //
}
```

Fig. 2.6 *Example to show how to define text style properties*

2.3.2. TEXT COMPONENT

The text component format is really similar when creating/adding the other components to the page. The text data source can either be raw text enclosed in quotation marks or text data variable.

Fig 2.7 *Screen shot of what you will see on the TFT module when example code in Fig 2.8 is uploaded: Text Component, "Hello World" should appear in the middle*

TEXT command format:
TEXT(Text component name, "Text component", Text Style);
TEXT command format with text data source from a variable:
TEXT(Text component name, Text variable, Text Style);

Now that all the knowledge necessary to add a text component to a page is introduced, an example can be created. Building upon the example in Fig 2.4, a text component can be added. To provide better guidance, newly added and modified parts of the code are highlighted:

LIB command format for multiple transformations:
LIB(Library image name, "Source/Filename?transformation1&transformation2..");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}

PAGE command format
Page Header
PAGE(Page name, Page style)
Page Body
{
POSN(x coordinate, y coordinate);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}

```
//FILENAME: TU480.mnu

LIB(mylibimage1,"SDHC/image1.bmp");  //

STYLE(mypagestyle,Page)     //
{
update = all;               //
image = mylibimage1;        //
}
STYLE(myfontAscii32,Text)   //
{
font = Ascii32;  //
size = 2;        //
col = white;     //
opacity = 75;    //
maxRow = 1;      //
maxLen = 11;     //
}

PAGE(mypage,mypagestyle)   //
{
POSN(240,135);                              //
TEXT(mytext,"Hello World",myfontAscii32);   //
}
SHOW(mypage);  //
```

Fig 2.8 *Example code on how to add text component to a page*

In iDev, memory allocation is an important aspect that a developer has to keep track of. Poor memory allocation can lead to errors, long loading times or TFT module crashing. On the other hand, good memory allocation provides an efficient way that allows the TFT module to run smoothly. The pages and page components in iDev uses memory, good memory allocation can be achieved by allocating the maximum predicted size that a page component will be using. In the example code above, memory allocation is achieved by setting the style parameter *maxRow* and *maxLen* to the maximum predicted row and length by the developer. In the example code above, *maxRow* is set to 1 and *maxLen* is set to 11 because only 1 row of text is needed and required and the text component has 11 characters.

2.3.3. TEXT MANIPULATION

There are other ways in iDev to manipulate text components other than using different style values. This part will explain what types of text component manipulation is possible and how to achieve it. When the developer requires the text component to be displayed in more than one row, then the HEX code *\\0D\\0A* has to be used. The code in Fig 2.7 will be used and modified to display text components in different rows.

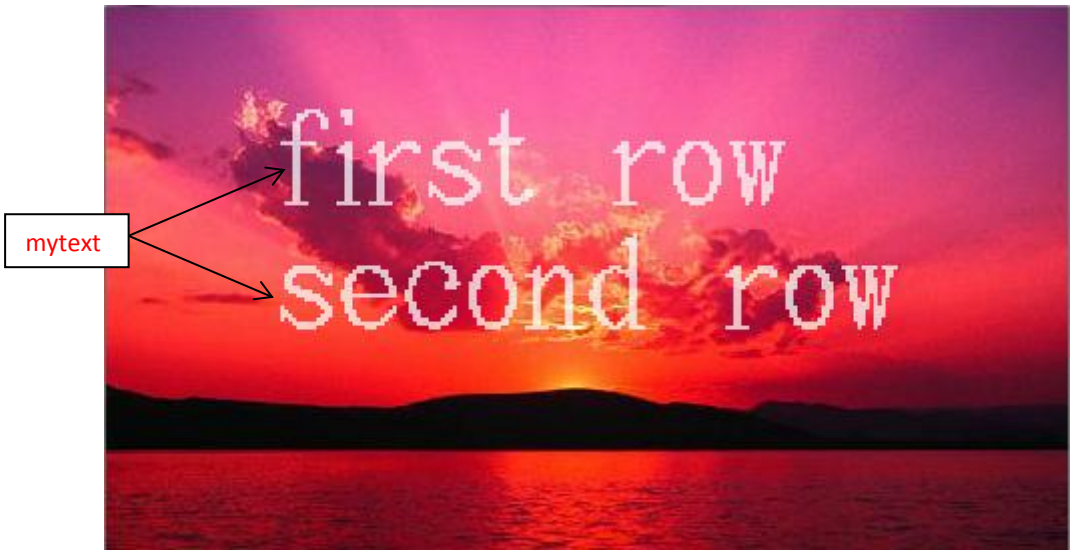


Fig 2.9 *Screen shot of what you will see on the TFT module when example code in Fig 2.10 is uploaded*

LIB command format:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
POSN command format:
POSN(x coordinate, y coordinate);
TEXT command format using text component and text cursor manipulation:
TEXT(Text component name, "\\HEX CodeText component", Text Style);
}

```
//FILENAME: TU480a.mnu

LIB(mylibimage1,"SDHC/image1.bmp"); //

STYLE(mypagestyle,Page)   //
{
update = all;             //
image = mylibimage1;      //
}
STYLE(myfontAscii32,Text) //
{
font = Ascii32; //
size = 2;       //
col = white;    //
opacity = 75;   //
maxRow = 2;     //
maxLen = 24;    //
}

PAGE(mypage,mypagestyle)  //
{
POSN(240,135);  //
TEXT(mytext,"first row\\0D\\0Asecond row",myfontAscii32);
//
}
SHOW(mypage); //
```

Fig 2.9 *Example code to show how to manipulate text row components using HEX code \\0D\\0A*

Like in any text editors such as Microsoft Word, when writing and editing text, the "Enter" button moves the cursor indicator to the next row. In iDev, the HEX code *\\0D\\0A* is an equivalent to the "Enter" button in text editors. This enables the developer to create multiple rows of text defined in one text component. The style parameter *maxRow* is increased to 2 because 2 rows of text component are required and *maxLen* is increased to 24 because approximately 24 characters would be the maximum requirement. This is a perfect example on how to achieve good memory allocation. The developer can specify certain types of text so that appropriate text components can be displayed on the TFT module. In iDev a text component can contain single byte HEX code from \\00 to \\FF.

TEXT command format using text component and text cursor manipulation:
TEXT(Text component name, "\\HEX CodeText component", Text Style);

It is important to remember that more than one text cursor and text component manipulating HEX codes can be applied at a time.

HEX code	Definition
\\01	Using this HEX code defines the text data as a password. If a text component is defined as a password then usually, the developer would want to display "*" on the screen for each character.
\\02	This HEX code inserts a hidden text cursor in the text component with insert off and overwrite on. This is useful in iDev projects that use a virtual on screen keyboard. When overwrite is on, then the text component is replaced by the new text input that is present on and after its current location from the on screen keyboard.

\\03	This HEX code inserts a hidden text cursor in the text component with insert on. When insert is on, then the text component is not replaced by the new text input from the on screen virtual keyboard, instead a character is inserted at the cursor's current position and in the process moving all characters past it one position further.
\\04	This HEX code inserts a visible horizontal line/underline cursor under the next character with insert off and overwrite on. This is useful in iDev projects that use a virtual on screen keyboard.
\\05	This HEX code inserts a visible horizontal line/underline cursor under the next character with insert on. This is useful in iDev projects that use a virtual on screen keyboard.
\\06	This HEX code inserts a block cursor in the next character with insert off and overwrite on. This is useful in iDev projects that use a virtual on screen keyboard.
\\07	This HEX code inserts a block cursor in the next character with insert off. This is useful in iDev projects that use a virtual on screen keyboard.

Fig. 2.11 *Table to show appropriate HEX codes to apply different text component and text cursor manipulation*

The code example in Fig 2.9 will be used and modified to provide a usage example of the HEX codes for text component and cursor manipulation. A screen shot of what is to be expected when the modified code is uploaded into the TFT module is found below.

Fig. 2.12 *Screen shot of what you will see on the TFT module when example code in Fig 2.13 is uploaded*

LIB command format:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
POSN command format:
POSN(x coordinate, y coordinate);
TEXT command format using text component and text cursor manipulation:
TEXT(Text component name, "\\HEX CodeText component", Text Style)
}

```
//FILENAME: TU480a.mnu

LIB(mylibimage1, "SDHC/image1.bmp"); //

STYLE(mypagestyle,Page)     //
{
update = all;               //
image = mylibimage1;        //
}
STYLE(myfontAscii32,Text)   //
{
font = Ascii32;             //
size = 1;                   //
col = white;                //
opacity = 75;               //
maxRows = 2;                //
maxLen = 24;                //
}

PAGE(mypage,mypagestyle)    //
{
POSN(240,100);              //
TEXT(mytext1, "this is a \\01password\\01 text", myfontAscii32);
//
POSN(240,135);              //
TEXT(mytext2, "H\\04ello World", myfontAscii32);
//
}
SHOW(mypage);
```

Fig. 2.13 *Example code to demonstrate how text component and text cursor manipulation are applied by using HEX codes*

In the example code above, the style of the text component is reduced to 1 and the HEX codes \\01 and \\04 are used. The HEX codes are placed before the applicable

character in all cases. In the text component *mytext1* the HEX code \\01 is used twice, this is because the HEX code \\01 is used for passwords and acts as a toggle. If the HEX code \\01 is only used (if the text component is *"this is a \\01password text"*), all the characters after it will be recognised as a password, hence displayed as *"**********"*. The text component *mytext2* contains the HEX code \\04 which inserts an underline cursor under the next character applied. The screen shot below shows the changes that the screen will display. The cursor manipulating HEX codes \\02 to \\07 are fully utilised in an iDev keyboard projects (link here for 'Keyboard Project' and link here for 'Multi Language Keyboard Project'). In some iDev projects, the text string in the text component that has been created before can be changed and to make this change visible then a different command format followed by a page refresh usually is needed.

TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;

2.3.4. IMAGE STYLE

The features of an image used in iDev are controlled by its style. In iDev, all styles have the same command format, so Image styles yield to this command format as well.

Image Style		
Parameter	**Expected Values**	**Definition**
scale	positive integer value	set the scaling of the library image **(default = 100)**
maxX/sizeX	1 to 3 × width of TFT module	allocate the RAM for the maximum width of the image **(default = width of TFT module: 4.3" default = 480)**
maxY/sizeY	1 to 3 × height of TFT module	allocate the RAM for the maximum height of the image **(default = height of TFT module: 4.3" default = 272)**
rotate	0	rotate the image component by 0° relative to the screen – don't rotate image **(default)**
	90	rotate the image component by 90° relative to the screen
	180	rotate the image component by 180° relative to the screen
	270	rotate the image component by 270° relative to the screen
action	i or instant	set the way in which an image appears on the screen instantly to the specified screen position/coordinates, the action parameter is useful for slideshows or animations **(default)**
	u or up	set the way in which an image appears on the screen moving up towards the specified screen position/coordinates
	d or down	set the way in which an image appears on the screen moving down
	l or left	set the way in which an image appears on the screen moving left towards the specified screen position/coordinates

	r or right	set the way in which an image appears on the screen moving right towards the specified screen position/coordinates
	ur or ru or upright	set the way in which an image appears on the screen moving up and right towards the specified screen position/coordinates
	dr or rd or downright	set the way in which an image appears on the screen moving down and right towards the specified screen position/coordinates
	ul or lu or upleft	set the way in which an image appears on the screen up and left towards the specified screen position/coordinates
	dl or ld or downleft	set the way in which an image appears on the screen down and left towards the specified screen position/coordinates
	a or all	sets to sequence through all image actions except instant
step	1 to LCD TFT height/width	sets the steps that the image action will take to appear on the screen, e.g. if the step is set to 5 and action to up, then the image will move up towards the specified destination, 5 pixels at a time and the time it takes for the image action to finish depends upon the image size **(default = 20)**
opacity	0 to 100	set the opacity of the image **(default = 100)**, 100 is opaque (not transparent)
curRel	CC	specify the placement/justification of text relative to the cursor to Centre Centre **(default)**
	TC	specify the placement/justification of text relative to the cursor to Top Centre
	BC	specify the placement/justification of text relative to the cursor to Bottom Centre
	RC	specify the placement/justification of text relative to the cursor to Right Centre
	LC	specify the placement/justification of text relative to the cursor to Left Centre
	TL	specify the placement/justification of text relative to the cursor to Top Left
	BL	specify the placement/justification of text relative to the cursor to Bottom Left
	TR	specify the placement/justification of text relative to the cursor to Top Right
	BR	specify the placement/justification of text relative to the cursor to Bottom Right

Fig 2.14 *Table to describe Image Style parameters and its definition*

Similar to text styles, there are different combinations available when creating image styles to suit a specific iDev project. If Style command format is required for better referencing, go to Chapter 2.3 of this guide. Using the style command format and the information in Fig 2.14, an Image Style example can be done:

```
//FILENAME: TU480a.mnu

STYLE(myimagestyle,Image)   //
{
scale = 150;   //
maxX = 600;    //
maxY = 300;    //
opacity = 85;  //
step = 10;     //
}
```

Fig 2.15 *Example code to show how to define image style properties*

2.3.5. IMAGE COMPONENT

In the iDev language, BMP and JPG image file types are supported. However, if transparency transformation is required then BMP files are highly recommended. Image components can be added to a page by the developer from 2 different sources. The source image can come from the iDev library or straight from the SDHC card.

IMG command format for image already stored in iDev Library:
IMG(Image component name, Library Image name, Image Style);
IMG command format for image stored in SDHC card:
IMG(Image component name, "Source/Filename", Image Style);

The *IMG* command format is used to add 2 images to the example code in Fig 2.17. A screenshot of what to expect when the code is uploaded is found below.

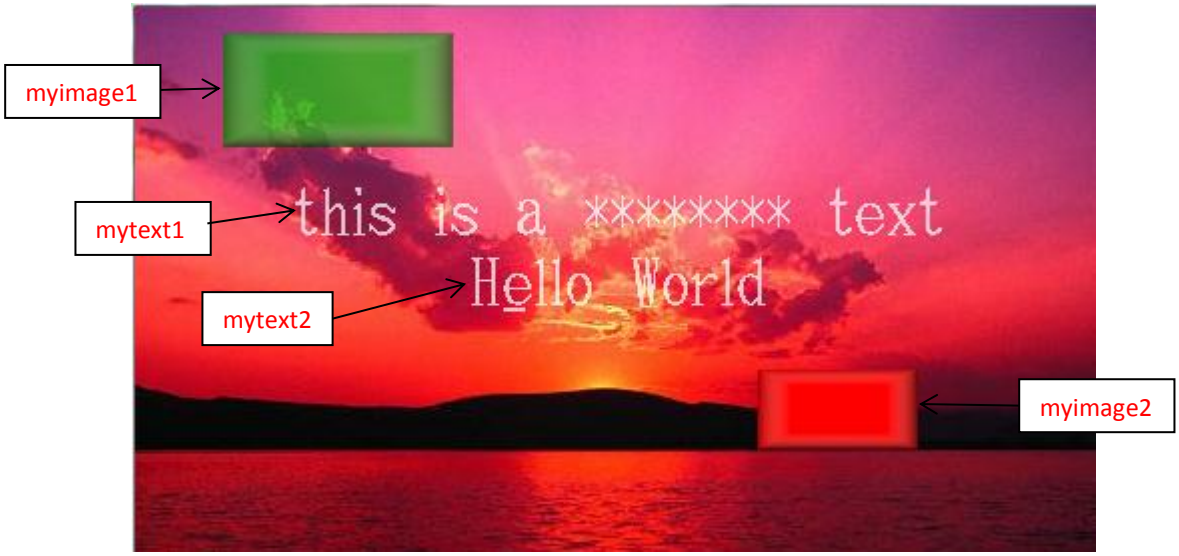


Fig 2.16 *Screen shot of what you will see on the TFT module when example code in Fig 2.17 is uploaded*

An example based on Fig 2.13 is going to be used and modified to add two image components to the page *mypage*:

LIB command format:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
IMG command format:
IMG(Image component name, Library Image name, Image Style);
IMG command format for image stored in SDHC card:
IMG(Image component name, "Source/Filename", Image Style);
}

```
//FILENAME: TU480a.mmu

LIB(mylibimage1,"SDHC/image1.bmp"); //
LIB(mylibimage2,"SDHC/image2.bmp"); //

STYLE(mypagestyle,Page) //
{
update = all;           //
image = mylibimage1;    //
}
STYLE(myfontAscii32,Text)  //
{
font = Ascii32;            //
size = 1;                  //
col = white;               //
opacity = 75;              //
maxRows = 2;               //
maxLen = 24;               //
}
STYLE(myimagestyle1,Image) //
{
maxX = 100;                //
maxY = 50;                 //
scale = 115;               //
opacity = 70;              //
}
STYLE(myimagestyle2,Image) //
{
maxX = 100;                //
maxY = 50;                 //
scale = 80;                //
opacity = 100;             //
}                          //

PAGE(mypage,mypagestyle)   //
{
POSN(240,100);  //
TEXT(mytext1,"this is a \\01password\\01 text",myfontAscii32);
//
POSN(240,135);  //
TEXT(mytext2,"H\\04ello World",myfontAscii32);
//
POSN(100,40);   //
IMG(myimage1,mylibimage2,myimagestyle1);
//
POSN(350,200);  //
IMG(myimage2,"SDHC/image3.bmp",myimagestyle2);
//
}
SHOW(mypage);   //
```

Fig 2.16 *Example code do demonstrate how to add image components to a page from two different sources*

The example in Fig 2.16 uses 2 different image styles with different image style parameters. This shows how multiple styles can be applied in different image components. In this example memory allocation is achieved by setting the *maxX* and *maxY* parameter to the size of the image used.

2.3.6. IMAGE MANIPULATION

In Chapter 1.4.2, library image transformations were explained and in this part of the guide those transformations will be used. For easier visualisation, 6 images will be added to the page in the example code and each image will have different transformation applied. Since 6 image components are now going to be used, typical iDev naming convention will be introduced to minimise confusion.

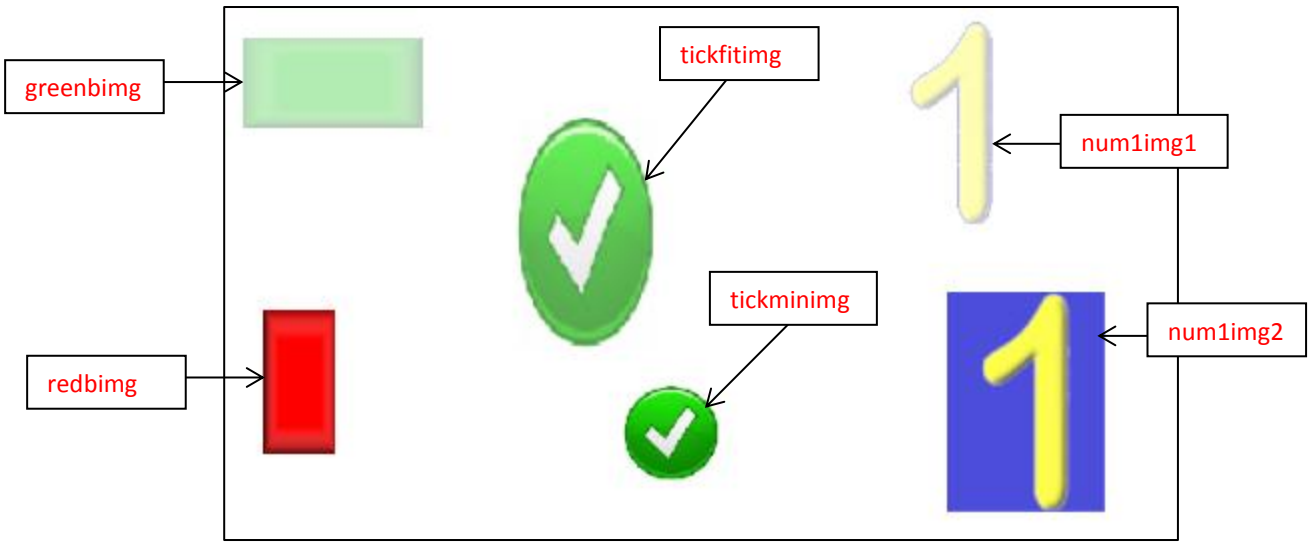


Fig 2.18 *Screen shot to show what would be displayed on the TFT module when the code in Fig 2.19 is uploaded*

LIB command format for multiple transformations:
LIB(Library image name, "Source/Filename?transformation1&transformation2..");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
POSN command format:
POSN(x coordinate, y coordinate);
IMG command format:
IMG(Image component name, Library Image name, Image Style);
}

```
//FILENAME: TU480a.mnu

LIB(greenblib,"SDHC/greenbox.bmp?scale=150");
//
LIB(redblib,"SDHC/redbox.bmp?rotate=90&scale=90");
//
LIB(tickfitlib,"SDHC/number1.bmp?scale=fit&width=60&height=100");
//
LIB(tickminlib,"SDHC/number1.bmp?scale=min&width=60&height=100");
//
LIB(num1lib1,"SDHC/tick.bmp?back=\\0000CD&scale=70");
//
LIB(num1lib2,"SDHC/tick.bmp?scale=40");
//
STYLE(mypagestyle,Page)   //
{
update = all;             //
back = white;             //
}
STYLE(myimagestyle1,Image) //
{
maxX = 250;                //
maxY = 180;                //
scale = 115;               //
opacity = 70;              //
curRel = CC;               //
}
STYLE(myimagestyle2,Image) //
{
maxX = 100;                //
maxY = 100;                //
scale = 80;                //
opacity = 100;             //
curRel = LC;               //
}
STYLE(myimagestyle3,Image) //
{
maxX = 100;                //
maxY = 100;                //
scale = 60;                //
opacity = 30;              //
curRel = RC;               //
}

PAGE(mypage,mypagestyle)   //
{
POSN(100,40);              //
IMG(greenbimg,greenblib,myimagestyle1);    //
POSN(20,+150);             //
IMG(redbimg,redblib,myimagestyle1);        //
POSN(180,-75);             //
IMG(tickfitimg,tickfilib,myimagestyle1);   //
POSN(+20,+100);            //
IMG(tickminimg,tickminlib,myimagestyle1);  //
POSN(400,60);              //
IMG(num1img1,num1lib1,myimagestyle1);      //
POSN(+0,200);              //
IMG(num1img2,num1lib2,myimagestyle1);      //
}
SHOW(mypage);              //
```

Fig. 2.19 *Example code to demonstrate image manipulation*

The example above exhibits image manipulation by using the *LIB, POSN, STYLE,* and *IMG* commands. As explained previously in Chapter 1.4.2, some image transformations can already be applied using the *LIB* command but some

transformations such as scaling and rotation can be applied using the *STYLE* command for an image as well. If both commands are used to perform the same transformation such as scaling then, the scaling transformation in the *LIB* command is applied first then the scaling is applied to the image component by the *STYLE* command. For example, the library image *num1img2* from the example code in Fig 2.17 has been set to decrease the scale of the image *tick.bmp* to 40%; the image component *myimage6* has used *num1img2* as an image source and *myimagestyle1* as an image style. The image style *myimagestyle1* is set to increase the scale of the image component to 115%; as a result the scale image *tick.bmp* is reduced to 40% and then increased to 115%. The same occurs when a rotation transformation is applied to an image component. Another image manipulation that most developers would end up using is the image transparency transformation, unlike the scale and rotate transformation; transparency can only be applied using the *LIB* command. The difference between images with transparency transformation applied is evident in the screen shot below; for better demonstration the same image is used for both image components. The image component *num1img1* has the transparency transformation applied but image component *num1img2* doesn't. The library image *tickfitimg* and *tickminimg* have the other two types of scale transformation that can be applied to an image and the difference between the two is evident in the screen shot below. As stated in Chapter 2.2, *POSN* commands can be used to reposition the cursor; the usage is demonstrated in the example code's *Page Body* in Fig 2.18.

2.3.7. DRAW STYLE

The features of shapes and graphs drawn in iDev are controlled by its style.

Draw Style		
Parameter	**Expected Values**	**Definition**
type	c or circle	circle shape style – draws a circle of the specified diameter, lower value in x and y parameter taken as diameter **(default)**
	b or box	box shape style – draws a box of the specified height (x parameter) and width (y parameter)
	l or line	line shape style – draws a line of the length and angle based on x and y parameter
	x or xBar	bar x graph style – draws horizontal line from 0 to x and clears from x+1 to xmax
	y or yBar	bar y graph style – draws a vertical line from 0 to y and clears from y+1 to ymax
	p or pixel	scatter graph style – creates a pixel at the point x,y
	t or trace	trace/line graph style – joins the dots between current point and previous point
maxX	1 to 3 × width of TFT module	allocate the RAM for the maximum width of the draw component **(default = width of TFT module: 4.3" default = 480)**
maxY	1 to 3 × height of TFT module	allocate the RAM for the maximum height of the draw component **(default = height of TFT module: 4.3" default = 272)**
col	HEX code or colour name, Alpha in HEX	specify the line border colour of the shape, if transparency is required use alpha in HEX e.g. col = \\80FFFFF00; displays 50% transparent yellow **(default = white)**

back	HEX code or colour name, Alpha in HEX	specify the fill colour of the shape, if transparency is require use alpha in HEX, same with col parameter **(default = black)**
opacity	0 to 100	set the opacity of the draw component **(default = 100)**, 100 is opaque (not transparent)
width	positive integer value	specify the graph/shape component width **(default = 1)**
rotate	0	rotate the draw component by 0° relative to the screen – don't rotate shape/graph **(default)**
	90	rotate the draw component by 90° relative to the screen
	180	rotate the draw component by 180° relative to the screen
	270	rotate the image component by 270° relative to the screen
curRel	CC	specify the placement/justification of shape/graph relative to the cursor to Centre Centre **(default)**
	TC	specify the placement/justification of shape/graph relative to the cursor to Top Centre
	BC	specify the placement/justification of shape/graph relative to the cursor to Bottom Centre
	RC	specify the placement/justification of shape/graph relative to the cursor to Right Centre
	LC	specify the placement/justification of shape/graph relative to the cursor to Left Centre
	TL	specify the placement/justification of shape/graph relative to the cursor to Top Left
	BL	specify the placement/justification of shape/graph relative to the cursor to Bottom Left
	TR	specify the placement/justification of shape/graph relative to the cursor to Top Right
	BR	specify the placement/justification of shape/graph relative to the cursor to Bottom Right
xOrigin	positive integer value	for graph styles, specify x origin with respect to declared graph **(default = 0)**
yOrigin	positive integer value	for graph styles, specify y origin with respect to declared graph **(default = 0)**
xScale	positive integer value	for graph styles, scale the x coordinate value automatically to fit the declared graph
yScale	positive integer value	for graph styles, scale the y coordinate value automatically to fit the declared graph
xScroll	positive integer value	for graph styles, set the scroll direction and increment/speed of graph to display along the x axis e.g. xScroll = 5, left to right by 5 pixels or xScroll = -20, right to left by 20 pixels **(default = 0 – none)**
yScroll	positive integer value	for graph styles, set the scroll direction and increment/speed of graph to display along the y axis, same format as xScroll **(default = 0 – none) non operational**

Fig 2.20 *Table to describe Text Style parameters and its definition*

In this stage of the guide, the developer should now be familiar with the *STYLE* command format and its usage. For better referencing and guidance, the *STYLE* command format can be found in Chapter 2.3.

```
//FILENAME: TU480a.mmu

STYLE(mydrawstyle,Draw) //
{
type = b;          //
width = 5;         //
back = grey;       //
col = red;         //
curRel = LR;       //
}
```

Fig 2.21 *Example code to show how to define draw style properties*

2.3.8. DRAW COMPONENT

Draw components can either be a shape or a graph. The different types of shapes and graphs that can be used are found in Fig 2.20. These draw components can be just an outline or filled. Alpha blending of the draw component colour is supported in iDev but quicker display updates is achieved if the draw component doesn't have alpha blending.

DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);

In the example code below, the page *mypage* based on the previous example codes will be modified to show a box shape component and circle shape component.

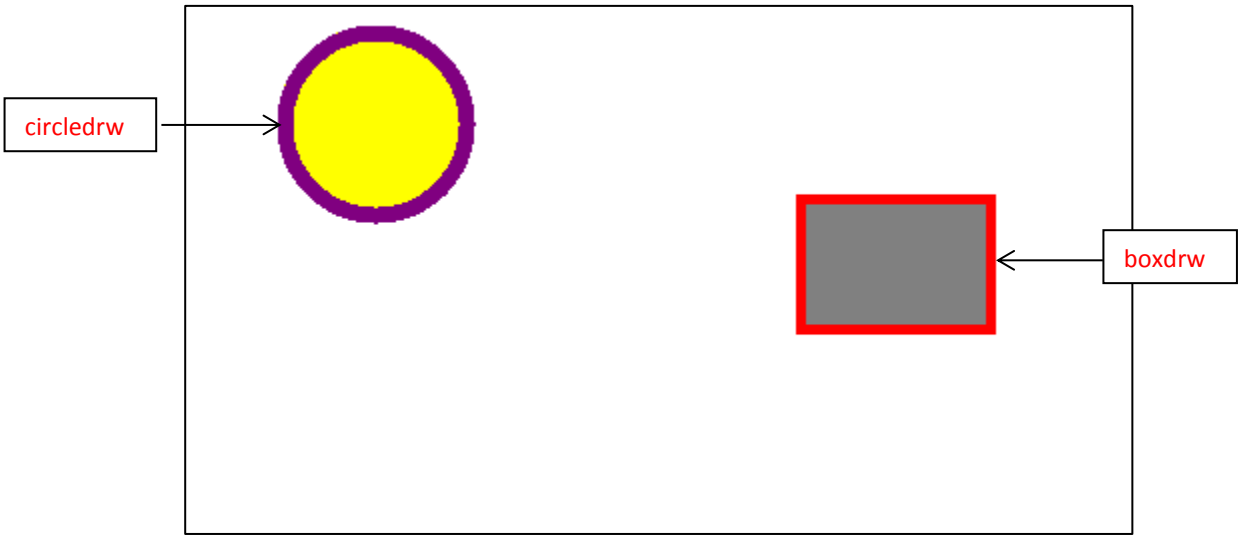


Fig 2.22 *Screen shot to show what would be displayed on the TFT module when the code in Fig 2.23 is uploaded*

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
}

```
//FILENAME: TU480a.mnu

STYLE(mypagestyle,Page)     //
{
update = all;  //
back = white;  //
}
STYLE(myboxstyle,Draw)      //
{
type = b;      //
maxX = 100;    //
maxY = 70;     //
width = 5;     //
back = grey;   //
col = red;     //
curRel = LC;   //
}
STYLE(mycirclestyle,Draw) //
{
type = c;        //
maxX = 100;      //
maxY = 100;      //
width = 8;       //
back = yellow;   //
col = purple;    //
curRel = RC;     //
}

PAGE(mypage,mypagestyle) //
{
POSN(310,130);                        //
DRAW(boxdrw,100,70,myboxstyle);       //
POSN(150,60);                         //
DRAW(circledrw,100,150,mycirclestyle); //
}
SHOW(mypage);     //
```

Fig 2.23 *Example code to demonstrate how the draw command is used*

As mentioned previously in this guide, memory allocation is important and the code in Fig 2.23 demonstrates how it is implemented in draw components. For the draw style *myboxstyle*, the style parameter *maxX* is set to be the same as the box component's length and same goes with *maxY* and width. The diameter of a circle component in iDev can either be the x or y parameter value in the *DRAW* command format. The diameter taken is the lower parameter value; in the example above the circle component *circledrw* has a diameter of 100, not 150.

2.3.9. DRAW MANIPULATION

If the developer requires alpha blending in draw components, then it should be specified in the draw style of the draw component concerned. The example code in Fig 2.23 will be modified to include alpha blending to the draw components and also the background will be changed to emphasise the alpha blending.

Fig 2.24 *Screen shot to show what would be displayed on the TFT module when the code in Fig 2.25 is uploaded*

LIB command format for images:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}

PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
POSN command format:
POSN(x coordinate, y coordinate);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
}

```
//FILENAME: TU480a.mnu

LIB(sunsetlib,"SDHC/sunset.bmp"); //

STYLE(mypagestyle,Page) //
{
update = all;       //
image = sunsetlib;  //
}
STYLE(myboxstyle,Draw) //
{
type = b;           //
maxX = 100;         //
maxY = 70;          //
width = 5;          //
back = \\BECCCCCC;  //
col = red;          //
curRel = LC;        //
}
STYLE(mycirclestyle,Draw)  //
{
type = c;           //
maxX = 100;         //
maxY = 100;         //
width = 8;          //
back = \\70FFFF00;  //
col = purple;       //
curRel = RC;        //
}

PAGE(mypage,mypagestyle)        //
{
POSN(310,130);                  //
DRAW(boxdrw,100,70,myboxstyle); //
POSN(150,60);                   //
DRAW(circledrw,100,150,mycirclestyle);  //
}
SHOW(mypage);                   //
```

Fig 2.24 *Example code demonstrating how alpha blending is applied to draw components*

In order to apply alpha bending the colour of the draw component have to be converted into HEX code. The colour yellow when converted to HEX code is *FFFF00* and grey is *CCCCCC*. The level of alpha blending or transparency to be applied to the draw component should also be in HEX code. There are 256 levels of transparency/alpha blending that can be applied to a draw component. In the example above, the alpha value used in HEX is *BE* for *circledrw* and *70* for *boxdrw*. In effect the draw component *circledrw* now has an alpha blending level of 190 (BE

in HEX is 190 in decimal) and *boxdrw* has an alpha blending level of 112 (70 in HEX is 112 in decimal). Graphs can also be created using the *DRAW* command in iDev but it is not as simple as creating shapes. Graph components are fully utilised if either x and y or both coordinate values are continuously varying e.g. it can be used in iDev projects displaying sensor outputs. Due to the complexity required applying graphs, an example project using the *DRAW* command and other iDev commands to create graphs is introduced in Chapter 9. There is an example project called '25 Samples / Second ADC1 Graph Project' (link here) from the website that uses the *DRAW* command to create graphs displaying values based on ADC inputs.

2.3.10. KEY STYLE

The features of key/touch actions in iDev are controlled by its style.

Key Style		
Parameter	**Expected Values**	**Definition**
type	touch	specify the type of input for the TFT display to be touch **(default)**
	keyio	specify the type of input for the TFT display to be an external keyio (see Chapter 4.9)
debounce	value in milliseconds	set the time delay for the key press to be detected and stabilise **(default = 50)**
delay	value in milliseconds	set the time delay before autorepeat occurs – when autorepeat is enabled, key press action is repeatedly detected **(default = 500, if 0 autorepeat is disabled)**
repeat	value in milliseconds	set the repeat period if the key action is held down **(default = 500)**
action	d or down	set the action detected when a key is pressed **(default)**
	u or up	set the action detected when a key is released
	c or change	set the action detected when either a key is pressed or released (not supported in external keyio)
curRel	CC	specify the placement/justification of key component relative to the cursor to Centre Centre **(default)**
	TC	specify the placement/justification of key component relative to the cursor to Top Centre
	BC	specify the placement/justification of key component relative to the cursor to Bottom Centre
	RC	specify the placement/justification of key component relative to the cursor to Right Centre
	LC	specify the placement/justification of key component relative to the cursor to Left Centre
	TL	specify the placement/justification of key component relative to the cursor to Top Left
	BL	specify the placement/justification of key component relative to the cursor to Bottom Left
	TR	specify the placement/justification of key component relative to the cursor to Top Right
	BR	specify the placement/justification of key component relative to the cursor to Bottom Right

Fig. 2.26 *Table to describe Key Style parameters and its definition*

For the *KEY* command there is built in styles that may be suitable for general use and the table below describes the styles' parameter values.

Parameter	Built in KEY styles		
	TOUCH	TOUCHR	TOUCHC
type	touch	touch	touch
debounce	50	50	50
repeat1	0	1000	1000
repeat2	0	200	200
action	D	D	C

Fig. 2.27 *Table describing built-in KEY styles in iDev*

2.3.11. KEY COMPONENT

In iDev, key components refer to touch/press input by the user. The developer can manipulate the key components' size/detecting area or define a key on the external keyboard. Using the *KEY* command, the developer can then specify the function to start, e.g. a toggle button would have a function that changes the colour of the button when it is pressed or inline commands such as *RUN*, *SHOW*, *HIDE* and *LOAD* commands. Multiple inline commands can be used, a comma can is used to separate each command. The *X* and *Y* parameter refers to the size of the key area in the screen when the style type of the key is touch. If the style type of the key is keyio i.e. external keys is used, then *X* and *Y* parameter refers to the location of the connection between the key input/output ports and the external key.

KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);

In the example code below, the page *mypage* based on the previous example codes will be modified to create a key component that can be used as a toggle button. They *KEY* command requires actions when a key is pressed by the user and sometimes the *FUNC* command is used. The *FUNC* command is explained further in Chapter 2.6 of this guide, for now inline commands *SHOW* and *HIDE* will be used in the example code below:

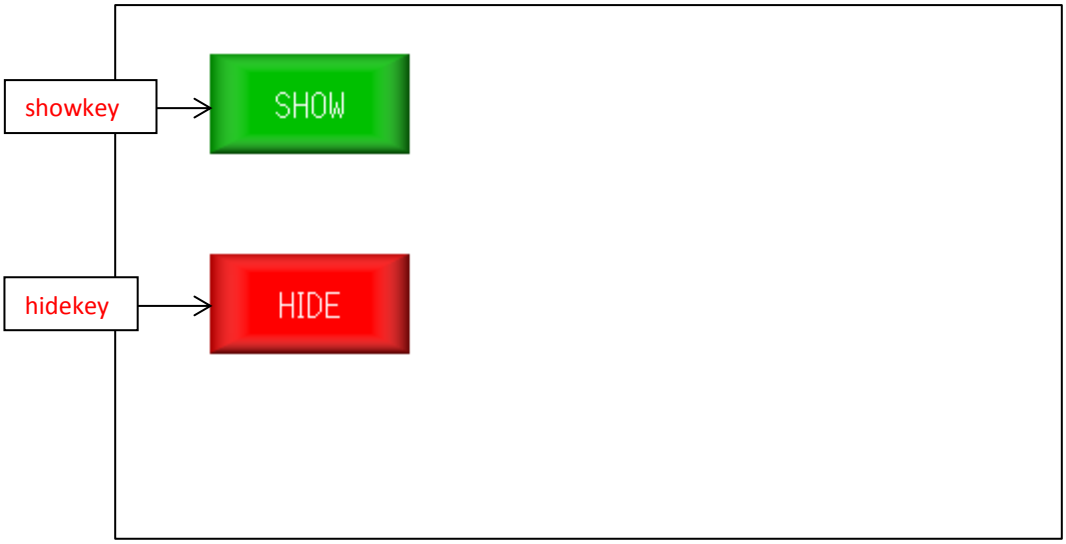


Fig. 2.28 *Screen shot to display when the key component hidekey is pressed*

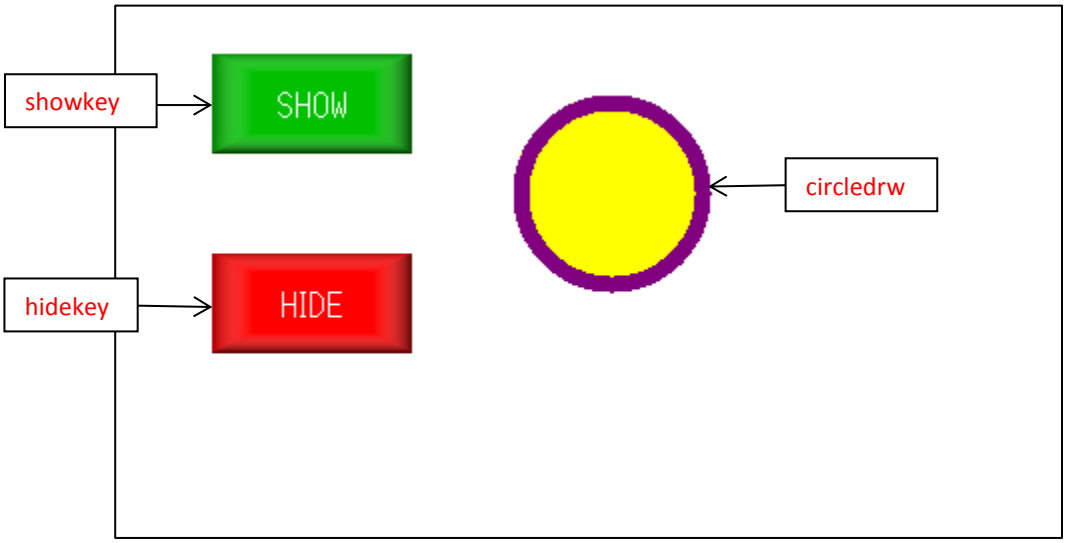


Fig. 2.29 *Screen shot to display when the key component showkey is pressed*

LIB command format for images:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}

PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
}

```
//FILENAME: TU480a.mnu

LIB(greenblib,"SDHC/greenbox.bmp"); //
LIB(redblib,"SDHC/redbox.bmp");     //

STYLE(mypagestyle,Page)    //
{
update = changed;  //
back = white;      //
}
STYLE(mycirclestyle,Draw) //
{
type = c;          //
maxX = 100;        //
maxY = 100;        //
width = 8;         //
back = yellow;     //
col = purple;      //
curRel = CC;       //
}
STYLE(myfontAscii16,Text) //
{
font = Ascii16;    //
size = 1;          //
col = white;       //
maxRows = 1;       //
maxLen = 24;       //
}
STYLE(mykeystyle,Key)      //
{
debounce = 30;     //
action = d;        //
}

PAGE(mypage,mypagestyle) //
{
POSN(250,95);                          //
DRAW(circledrw,120,100,mycirclestyle); //
HIDE(circledrw);                       //
POSN(100,50);                          //
KEY(showkey,[SHOW(circledrw);;],100,50,mykeystyle);  //
IMG(greenbut,greenbimg,myimagestyle);  //
TEXT(showtxt,"SHOW",myfontAscii16);    //
POSN(+0,+100);                         //
KEY(hidekey,[HIDE(circledrw);;],100,50,mykeystyle);  //
IMG(redbut,redbimg,myimagestyle);      //
TEXT(hidetxt,"HIDE",myfontAscii16);    //
}
SHOW(mypage);                          //
```

Fig. 2.30 *Example code demonstrating how to use the KEY command with inline commands*

The key components *showkey* and *hidekey* are used to show and hide the draw component *circldrw* respectively. The *SHOW* command is used to display page or page components and *HIDE* command is used to choose which page or page components not to display. They key components both use the *SHOW* and *HIDE* commands which is explained thoroughly in Chapter 2.5.1 and 2.5.2 of this guide. The example code also uses the *IMG* and *TEXT* commands to create a "button". The usage of double semicolons (;;) is explained in Chapter 2.4 of this guide, but it is basically used for page components that have currently change its state so a page component refresh is required. Since a key component isn't a visible component, the screenshots below would display when the button is pressed and not pressed and the position of the key component would be exactly the same as the image component and the text component.

Another example that will be shown is the usage of external keys for the key components. As this involves a physical keyboard and the keys input and output interface, a screenshot would not demonstrate anything properly. The use of interfaces is explained in Chapter 4 of this guide.

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2], X, Y, Key style);
}

```
//FILENAME: TU480a.mnu

SETUP(KEYIO)          //
{
active = \\0000001F; //
keyb = \\0000000F;   //
}

STYLE(mypagestyle,Page)      //
{
update = changed;  //
back = white;      //
}
STYLE(myfontAscii16,Text)    //
{
font = Ascii16;    //
size = 1;          //
col = white;       //
maxRows = 1;       //
maxLen = 24;       //
}
STYLE(mykeystyle,Key)        //
{
type = keyio;      //
debounce = 10;     //
action = d;        //
}

PAGE(mypage,mypagestyle)  //
{
POSN(250,95);                          //
DRAW(circledrw,120,100,mycirclestyle); //
HIDE(circledrw);                       //
POSN(100,50);                          //
KEY(showkey,[SHOW(circledrw);;],K00,K01,mykeystyle);  //
TEXT(showtxt,"SHOW",myfontAscii16);    //
POSN(+0,+100);                         //
KEY(hidekey,[HIDE(circledrw);;],K02,K03,mykeystyle);  //
TEXT(hidetxt,"HIDE",myfontAscii16);    //
}
SHOW(mypage);                          //
```

Fig. 2.31 *Example code to demonstrate how key component with style keyio is used*

As mentioned in Chapter 2.3.11, the *X* and *Y* parameter in keyio refers to the connection between the external keyboard and the key input and output ports of the TFT module. The key component *showkey* is set to perform the inline command *SHOW(circledrw)* when external key *K00* and *K01* are connected and the key component *hidekey* when external key *K02* and *K03*. There are 30 external keys that can be connected to the TFT module and external keys that are being used can be specified in the *KEY* command format by using HEX codes.

2.3.12. KEY MANIPULATION

In iDev, the use of key components enables the developer to get inputs from the user and also provide a user interface. In all user interfaces that use a touchscreen, toggle buttons are always present. The example below would be using the *VAR, FUNC* and *IF* command, all of which is explained in Chapter 3.1.2, Chapter 2.6 and Chapter 3.4 of this guide respectively.

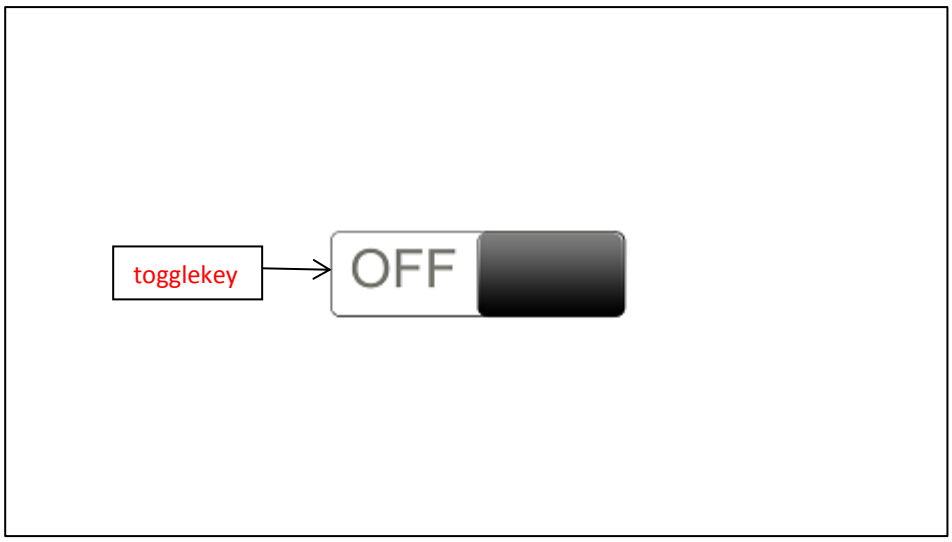


Fig. 2.32 *Screen shot to display when the key component togglekey is not pressed*

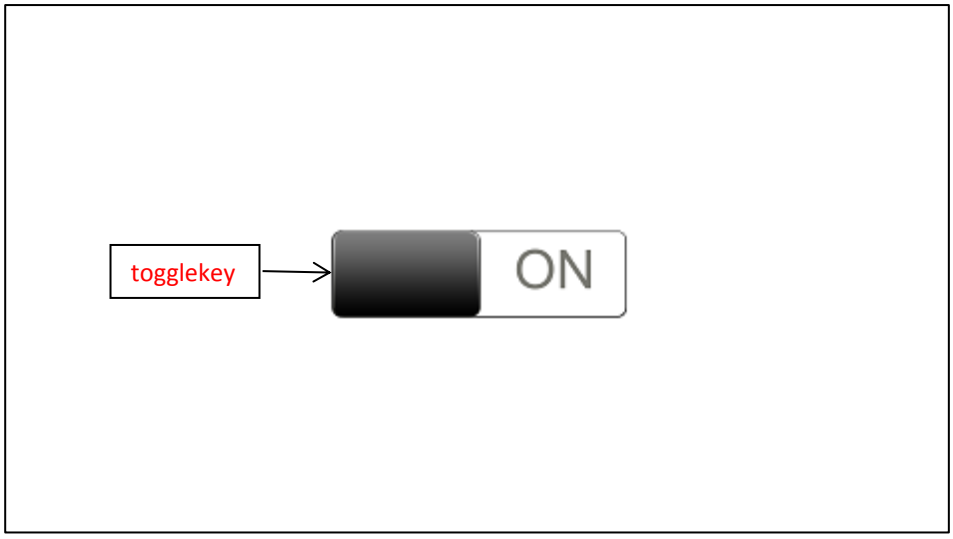


Fig 2.33 *Screen shot to display when the key component togglekey is pressed*

LIB command format for multiple transformations:
LIB(Library image name, "Source/Filename?transformation1&transformation2..");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body

{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format
KEY(Key component name, Function name, X, Y, Key style);
}

```
//FILENAME: TU480a.mnu

LIB(togofflib,"SDHC/togoff.bmp"); //
LIB(togonlib,"SDHC/togon.bmp");   //

VAR(togvar,0,U8);           //

STYLE(mypagestyle,Page)     //
{
update = changed;  //
back = white;      //
}
STYLE(myimagestyle,Image)   //
{
maxX = 150;        //
maxY = 50;         //
}
STYLE(mykeystyle,Key)       //
{
debounce = 30;     //
action = d;        //
}

PAGE(mypage,mypagestyle)    //
{
POSN(240,135);                                  //
KEY(togglekey,istogfunc,150,50,mykeystyle);     //
IMG(togoffbut,togofflib,myimagestyle);          //
IMG(togonbut,togonlib,myimagestyle);            //
HIDE(togonbut);                                 //
}

FUNC(istogfunc)    //
{
IF(togvar = 0?[RUN(togfunc);]:[RUN(nottogfunc);]);
//
}
FUNC(togfunc)      //
{
HIDE(togoffbut);   //
SHOW(togonbut);    //
LOAD(togvar,1);;   //
}
FUNC(nottogfunc)   //
{
HIDE(togonbut);    //
SHOW(togoffbut);   //
LOAD(togvar,0);;   //
}
SHOW(mypage);      //
```

Fig 2.34 *Example code demonstrating how to create a toggle button*

The toggle button works by having two images placed in the same position and on top of each other. Then when the “button” is pressed the appropriate image is shown depending on the toggle button’s state. The image being shown or hidden is

achieved by the use of the *SHOW* and *HIDE* command. The example code in Fig 2.32 uses *VAR* command to create a variable. This variable is used as an indicator for the state of the toggle button. The *VAR* command is described thoroughly in Chapter 3.1 of this guide. The *FUNC* command is also used in this example to specify the actions to take when the toggle button is pressed. If further understanding of the *FUNC* command is needed refer to Chapter 2.6 of this guide. Lastly the *IF* command is used to create a condition to specify which function to run depending on the state of the toggle button. The *IF* command is explained in Chapter 3.4 of this guide. The state of the toggle button determines which function to perform; if the button is pressed show the appropriate image and vice-versa. There are other touchscreen key elements that can be created in iDev such as scroll bars or cursor/mouse indicator. There are example codes in Chapter 9 demonstrating how to create.

2.4. UPDATING COMPONENTS

All the page components are now introduced, but in some example codes it is apparent that some lines of code uses double semicolons (;;) instead of a single one (;). In iDev, page components and other commands require refreshing or updating if their current state or value has changed to perform the operation completely. In Chapter 2.3.11, the example code in Fig 2.28 uses the page refresh command. The page refresh command in that example is used to update the page when to show or hide the draw component *circledrw*. The page refresh command (;;) updates the state of the page being displayed on the TFT module, so that the page components that have a different state from before is updated e.g. when to show or hide components. Basically, the page refresh command is used if the developer wants to change the page component's state visually. The use of page refresh is applied and fully utilised by the page style parameter *update*. If *update* = *all*, then all the page components and other iDev components (refers to other components that is created by the developer such as variables, functions or buffers etc…) are updated i.e. everything in the page is redrawn. On the other hand, if *update* = *changed* then only the page components and other iDev components that have been changed are updated. An iDev developer should learn to use the page refresh command efficiently to improve his/her iDev project's loading time and the page components and other iDev components to be updated accordingly. The efficient use of the page refresh command is demonstrated below. There will be two example codes below which will be compared. The part of the code with the page refresh command is highlighted.

```
//FILENAME: TU480a.mnu

LIB(togofflib,"SDHC/togoff.bmp"); //
LIB(togonlib,"SDHC/togon.bmp");   //

VAR(togvar,0,U8);   //

STYLE(mypagestyle,Page)   //
{
update = changed;  //
back = white;      //
}
STYLE(myimagestyle,Image) //
{
maxX = 150;        //
maxY = 50;         //
}
STYLE(mykeystyle,Key)     //
{
debounce = 30;     //
action = d;        //
}

PAGE(mypage,mypagestyle)  //
{
POSN(240,135);                                  //
KEY(togglekey,istogfunc,150,50,mykeystyle);     //
IMG(togoffbut,togofflib,myimagestyle);          //
IMG(togonbut,togonlib,myimagestyle);            //
HIDE(togonbut);
}

FUNC(istogfunc)   //
{
IF(togvar = 0?[RUN(togfunc);]:[RUN(nottogfunc);]);
//
}
FUNC(togfunc)     //
{
HIDE(togoffbut); //
SHOW(togonbut);  //
LOAD(togvar,1);; //
}
FUNC(nottogfunc) //
{
HIDE(togonbut);  //
SHOW(togonbut);  //
LOAD(togvar,0);; //
}
SHOW(mypage);     //
```

Fig 2.35 *This example code demonstrates the proper of use of the page refresh command*

As you can see, the page refresh command is only used in the last line for the functions *togfunc* and *nottogfunc*; the page is only refreshed once when the appropriate functions are called.

```
// FILENAME: TU480a.mnu

LIB(togofflib,"SDHC/togoff.bmp"); //
LIB(togonlib,"SDHC/togon.bmp");   //

VAR(togvar,0,U8);   //

STYLE(mypagestyle,Page)   //
{
update = changed;  //
back = white;      //
}
STYLE(myimagestyle,Image) //
{
maxX = 150;        //
maxY = 50;         //
}
STYLE(mykeystyle,Key)     //
{
debounce = 30;     //
action = d;        //
}

PAGE(mypage,mypagestyle)  //
{
POSN(240,135);                                  //
KEY(togglekey,istogfunc,150,50,mykeystyle);     //
IMG(togoffbut,togofflib,myimagestyle);          //
IMG(togonbut,togonlib,myimagestyle);            //
HIDE(togonbut);
}

FUNC(istogfunc)   //
{
IF(togvar = 0?[RUN(togfunc);]:[RUN(nottogfunc);]);
//
}
FUNC(togfunc)     //
{
HIDE(togoffbut);; //
SHOW(togonbut);;  //
LOAD(togvar,1);;  //
}
FUNC(nottogfunc)  //
{
HIDE(togonbut);;  //
SHOW(togonbut);;  //
LOAD(togvar,0);;  //
}
SHOW(mypage);     //
```

Fig. 2.36 *This example code shows the improper use of the page refresh command*

The code in Fig 2.36 demonstrates improper use of the page refresh command. Although each line of code in the functions *togfunc* and *nottogfunc* are changing the state of the page components and iDev components, it is not needed to place a page refresh command in each one to update the page appropriately. Placing a page refresh command redraws all the page components and iDev components. So in the example above, the page is refreshed three times effectively every time the functions are called; this is unnecessary. The loading times would not be noticeable because the example code above does not have a lot of page components but an iDev project that has loads of page components and have improper use of page refresh can significantly increase loading times of pages. The correct usage of the

page refresh command may not be obvious at this point because it is based entirely on what the developers want to achieve. To improve on using the page refresh command, it may be worth practicing with some example code that uses the *SHOW* and *HIDE* commands and experimenting.

2.5. PAGE COMPONENTS MANIPULATION

Page components can be manipulated by the use of iDev commands: *SHOW*, *HIDE* and *DEL*. The definitions of each of these commands are basically in their name, i.e. *SHOW* command shows or displays components, *HIDE* command hides components etc… These iDev commands aid the developer in manipulating iDev components to be specifically tailored to suit their project's needs.

2.5.1. SHOW

This command reveals page or page components (definition of page components can be found in the Glossary-Chapter 10) in iDev. This command places the selected page on the top (visible) layer of the screen. In almost all of the example codes above, the *SHOW* command is used to display the page called *mypage*, without this command the TFT module the page created would not appear. In the future, a developer is always going to use the *SHOW* command to display his/her iDev project's main page. Note that the maximum allowable parameters in iDev are 16, so the maximum number of pages or page components that can be included inside a *SHOW* command is 16. The *SHOW* command can also be used to enable or disable interrupts in iDev. Interrupts are introduced properly in Chapter 4.7.

SHOW command format:
SHOW(Page name or page component name);
SHOW command format for multiple page components:
SHOW(Page name1/component name1,Page name2/component name2…);
HIDE command format to disable interrupts:
HIDE(Interrupt name1, Interrupt name2…);

```
//FILENAME: TU480a.mnu

SHOW(mypage);                          //
SHOW(mypagecomponent);                 //
SHOW(mypage1,mypage2,mypage3);         //
SHOW(mypagecomponent1,mypageocomponent2,mypagecomponent3);
//
SHOW(myinterrupt1,myinterrupt2); //
```

Fig. 2.37 *Example code demonstrating the usage of the SHOW command*

The *SHOW* command can be used to display multiple pages if the size of the page is smaller than the screen i.e. a popup page. If the page is the same size as the screen then only one page can be displayed at a time. There are no limitations on how many pages and page components can be displayed. There are reserved names in iDev that is used for page navigation:
SHOW(PREV_PAGE) – shows the page that was launched before the current displayed page
SHOW(THIS_PAGE) – refresh or update the current displayed page

The use of *SHOW* command is regularly used with the refresh command because it updates a page component's state but a refresh command is not always required. Key components are invisible components so the use of *SHOW* command would not display anything on the screen but instead it enables the key component selected.

2.5.2. HIDE

HIDE command is used to choose which pages or page components not to display. This command is basically the opposite of the *SHOW* command. When a page or page component selected to be hidden is still showing on the screen but then the page is refreshed then the selected page or page component will disappear from the module's view/screen. Similar to the *SHOW* command, the maximum number of pages or page components that can be hidden in a single *HIDE* command is 16. Lastly the *HIDE* command can also be used to enable and disable interrupts. Interrupts are introduced properly in Chapter 4.7.

HIDE command format:
HIDE(Page name or page component name);
HIDE command format for multiple page components:
HIDE(Page name1/component name1,Page name2/component name2…);
HIDE command format to disable interrupts:
HIDE(Interrupt name1, Interrupt name2…);

```
//FILENAME: TU480a.mnu

HIDE(mypage);                          //
HIDE(mypagecomponent);                 //
HIDE(mypage1,mypage2,mypage3);         //
HIDE(mypagecomponent1,mypageocomponent2,mypagecomponent3);
//
HIDE(myinterrupt1,myinterrupt2);  //
```

Fig. 2.38 *Example code demonstrating the usage of the HIDE command*

Similar to the *SHOW* command, when using *HIDE* usually a page refresh command is used to display the changes of the state of the selected page or page components to hide. Since key components are not visible on the display then hiding a key component simply disables it.

2.5.3. DEL

This command is used to iDev components created by the developer that are stored in the memory (SDRAM). If a page or page component that is currently displayed on the screen is selected to be deleted then they will remain visible until the page is refreshed. The *DEL* command is mainly used for large iDev projects where memory allocation is of importance. The use of *DEL* command is an equivalent of deleting items in Microsoft Windows OS. Also in Windows the deleted items go into the recycling bin and is only truly deleted once the recycling bin has been emptied, in iDev the iDev components is only truly deleted to free up memory when the command RESET(DELETED); is used (non-operational). A maximum of 16 iDev components can be deleted in a single *DEL* command.

DEL command format:
DEL(iDev component name);
DEL command format for multiple iDev components:
DEL(iDev component name1,iDev component name2…);

```
//FILENAME: TU480a.mnu

DEL(myiDevcomponent);
DEL(myiDevcomponent1,myiDevcomponent2,myiDevcomponent3);
```

Fig. 2.39 *Example code to show how to use the DEL command*

2.5.4. UPDATE STYLE – LOAD

As stated in the other chapters, the *STYLE* command controls the features of the page and page components. There could be an instance whereby the developer would like to change a certain aspect of the page component when a button is touched. An example would be the colour of a draw component shape can be changed using the *LOAD* command with the dot operator.

LOAD command format to update styles:
LOAD(Style name.Parameter,New Parameter Value);

The *LOAD* command can be used to perform different tasks but for updating styles then the command format stated above can be used. The example code below uses this command format to update and change the draw component shape's colour.

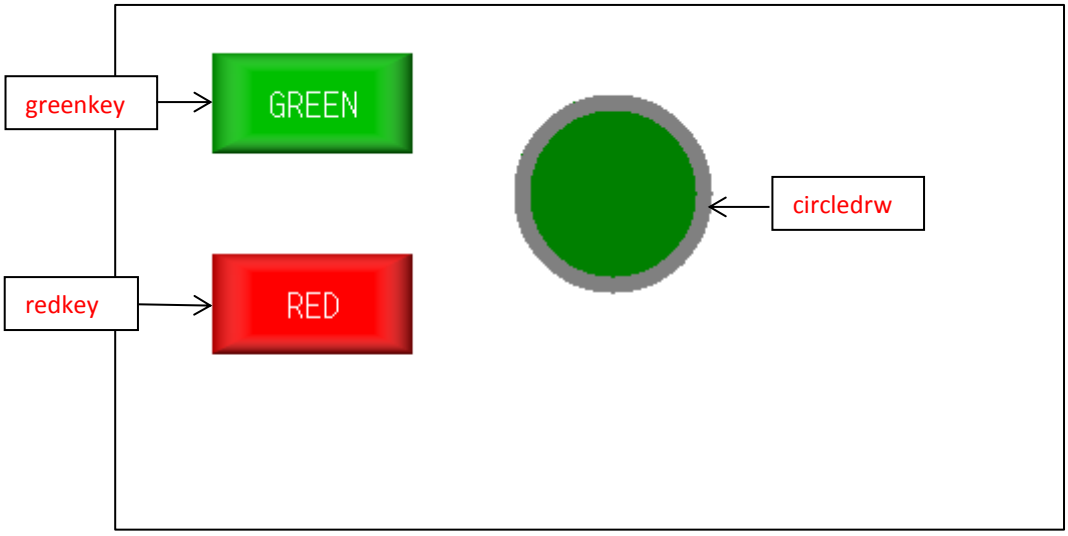


Fig. 2.40 *Screen shot to show what will be displayed when the GREEN button is pressed*

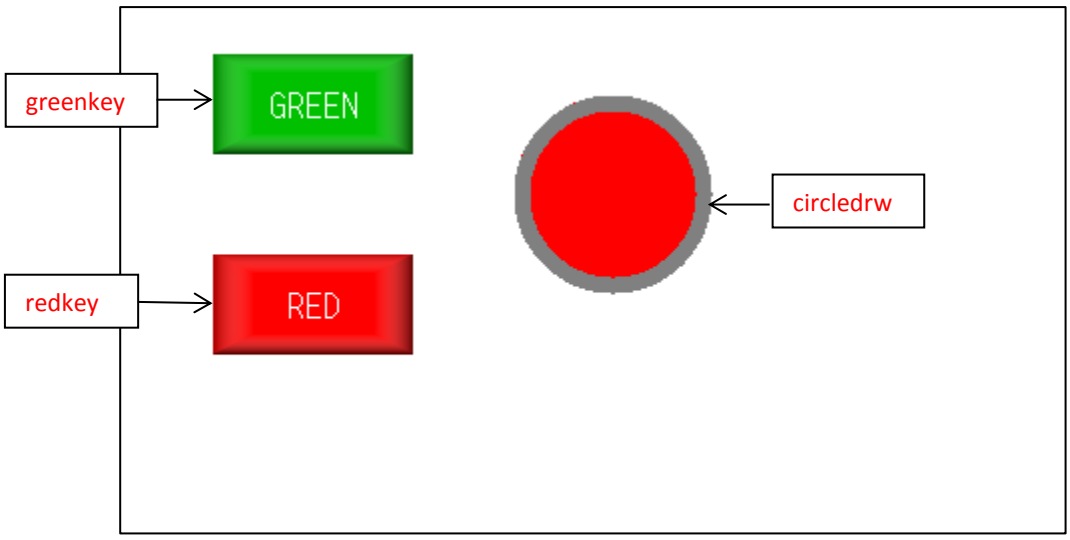


Fig. 2.41 *Screen shot to show what will be displayed when the RED button is pressed*

LIB command format for multiple transformations:
LIB(Library image name, "Source/Filename?transformation1&transformation2..");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style);
IMG command format for image already stored in iDev Library:
IMG(Image component name, Library Image name, Image Style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
LOAD command format to update styles
LOAD(Style name.Parameter,New Parameter Value);
}

SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

LIB(greenblib,"SDHC/greenbox.bmp"); //
LIB(redblib,"SDHC/redbox.bmp");     //

STYLE(mypagestyle,Page)    //
{
update = changed;  //
back = white;      //
}
STYLE(myfontAscii16,Text)  //
{
font = Ascii16;    //
size = 1;          //
col = white;       //
maxRows = 1;       //
maxLen = 24;       //
}
STYLE(myimagestyle,Image) //
{
maxX = 150;        //
maxY = 50;         //
}
STYLE(mycirclestyle,Draw) //
{
type = c;          //
maxX = 100;        //
maxY = 100;        //
width = 8;         //
back = green;      //
col = grey;        //
curRel = CC;       //
}
STYLE(mykeystyle,Key)    //
{
debounce = 30;     //
action = d;        //
curRel = CC;       //
}

PAGE(mypage,mypagestyle) //
{
POSN(250,95);                          //
DRAW(circledrw,120,100,mycirclestyle); //
POSN(100,50);                          //
KEY(greenkey,[LOAD(mycirclestyle.back,green);;],100,50,mykeystyle);
//
IMG(greenbut,greenblib,myimagestyle);  //
TEXT(greentxt,"GREEN", myfontAscii16); //
POSN(+0,+100);                         //
KEY(redkey,[LOAD(mycirclestyle.back,red);;],100,50,mykeystyle);
//
IMG(redbut,redblib,myimagestyle);      //
TEXT(redtxt,"RED", myfontAscii16);     //
}
SHOW(mypage);                          //
```

Fig. 2.42 *Example code demonstrating the use of the load command dot operator to update page component's style parameter*

The example code in Fig 2.42 creates two buttons which have separate actions: the green button changes the colour of the draw component to green when pressed and the red button changes the colour to red. The changes are achieved using the *LOAD* command dot operator format. When styles are updated, visible changes usually occur so this means that a page refresh (;;) is needed to display the correct changes on the screen.

2.6. FUNCTIONS

Function is an important aspect in all programing languages. The use of functions enables the developer to group certain lines of code into a unit, which can then be commanded from other parts of the program. The use of functions makes programming easier and more efficient because it enabled the developer to perform a number of tasks just by calling a function. Functions that have been predefined before can be demanded or called in any parts of the iDev project by using the *RUN* command. Functions in iDev have two parts namely: *Function Header* and *Function Body.* The *Function Header* contains the function name that a developer assigns; this is helpful for identification of appropriate functions in other parts of the code. The *Function Body* contains codes that perform certain tasks for the function's purpose. In iDev, functions can be nested into each other with a maximum of 12 times i.e. functions can be called within a function within a function and so on… for 12 times. The iDev commands *IF*, *RUN*, *INT* and *KEY* requires a function as a parameter in their command formats and sometimes inline functions can be used. In cases where function is only going to be use once and not elsewhere, inline functions may be appropriate.

RUN command format:
RUN(Function Name);
RUN command format with **Inline Function:**
RUN([Function contents]);
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
Inline Function command format :
In the function parameter of the iDev command
[Function contents]

```
//FILENAME: TU480a.mnu

RUN(togfunc); //RUN command usage
FUNC(togfunc)     //Function togfunc without inline
{
HIDE(togoffbut);  //
SHOW(togonbut);   //
LOAD(togvar,1);;  //
}
RUN([HIDE(togoffbut); SHOW(togonbut); LOAD(togvar,1);;]);
//Function togfunc with inline using RUN command
```

Fig. 2.43 *Example code to show the difference of functions with or without inline and usage of RUN command*

The example code in Fig 2.42 clearly demonstrates the difference between functions with or without inline commands and how the *RUN* command is used with inline functions. In iDev, the maximum amount of parameters per command allowed is 16 i.e. the maximum amount of commands inside a bracket that can either be separated by a comma (,) or a semi colon (;) is 16. This means that the maximum amount of commands in an inline function is 16. Both ways of using functions will achieve the purpose of the function but efficient use of functions is based on its purpose. If the developer can anticipate that the **function will be used more than once** in the iDev project then **functions without inline commands** would be more suitable. On the contrary, if the developer knows that the **function will definitely be only used once** then **the function with inline commands** should be used. The *RUN* command can also be used to send iDev commands through an interface. The required iDev commands that are sent through an interface can be stored in a text variable first then the commands are processed and sent by using the *RUN* command.

```
//FILENAME: TU480a.mnu

PAGE(mypage,mypagestyle)
//RUN command sent through an interface in a page

{
POSN(100,50);                                   //
KEY(greenkey,[LOAD(cmd,"LOAD(RS2,\\22Hello\\22);");],100,50,mykeystyle);
//
IMG(greenbut,greenblib,myimagestyle);           //
POSN(+0,+100);                                  //
KEY(redkey,[RUN(cmd);],100,50,mykeystyle);      //
IMG(redbut,redblib,myimagestyle);               //
}
```

Fig. 2.44 *Example code demonstrating how iDev commands are sent through an interface by using the RUN command*

2.7. LOOP

Loops are used to repeat specified actions a number of times in a page. The *LOOP* command in iDev can only be used inside a *PAGE* or a *FUNCTION*. A loop can be nested up to 12 times i.e. a loop can be called within a loop within a loop within a loop so on… for 12 times. Similar to other iDev command formats, the *LOOP* command has a *Loop Header* that contains the name and the duration. The duration parameter dictates the number of times the commands or actions in the loop is repeated. The duration values can be from 1-65000 or if the loop is required to run constantly then the text *FOREVER* is used as a value of the duration parameter. The *Loop Body* holds the actions and commands that require being repeated. As with other programming languages there are ways to create conditional loops in iDev by using the *EXIT* command. The *EXIT* command can be used as a terminator of a loop itself or to terminate a specified loop.

LOOP command format:
Loop Header
LOOP(Loop name, Loop duration)
Loop Body
{
Loop contents…
}

EXIT command format:
EXIT();
EXIT command format for a specific loop
EXIT(Loop name);

```
//FILENAME: TU480a.mnu

FUNC(myfunc)                    //LOOP command in a function
{
 LOOP(myloop,FOREVER)           //
 {
  LOAD(mydrawstyle.col,black);  //
 }
}

PAGE(mypage,mypagestyle)        //LOOP command in a page

{
POSN(240,120);                      //
TEXT(mytext,"Hello",myAscii32font);  //

 LOOP(myloop,3)                     //
 {
  LOAD(myAscii32font.size,4);       //
  LOAD(myAscii32font.col,black);    //
 }
}
```

Fig. 2.45 *Loop command usage example in a page or function*

Like in any other languages like C, an equivalent of 'while' loops in iDev is achieved by the use of the *IF* command with a condition and an action that ends the loop as shown below. The *IF* command is explained thoroughly in Chapter 3.4 of this guide.

```
//FILENAME: TU480a.mnu

FUNC(loopfunc)                  //
{
 LOOP(myloop,FOREVER)           //
 {
  CALC(myvar1,myvar1,3,"+");    //
  IF(myvar1=90?[exit(myloop);]); //
 }
}
```

Fig. 2.46 *Example code to demonstrate how to create loops with conditions*

The example code above creates a counter that goes on until the variable *myvar1* reaches *90* then the loop *myloop* is terminated. The conditional loop only works when a loop is in a function not when it is in a page.

2.8. NAVIGATION BETWEEN PAGES (LINKING)

Most iDev projects that a developer would create use more than 1 page. It is important to link these pages together to enable navigation for the user. The example code in Fig 2.50 links all the three pages together using *SHOW* and *HIDE* command.

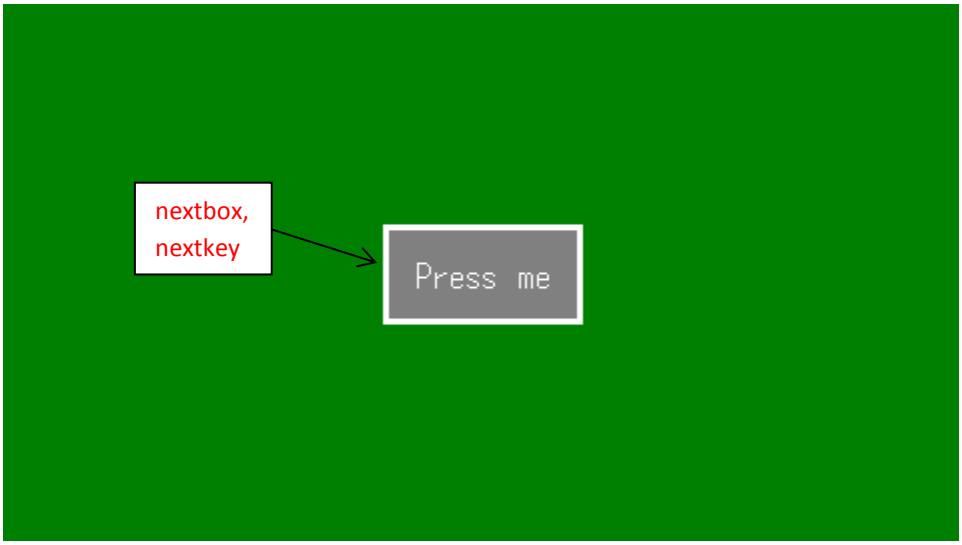


Fig. 2.47 *Screen shot to show the page homepg*

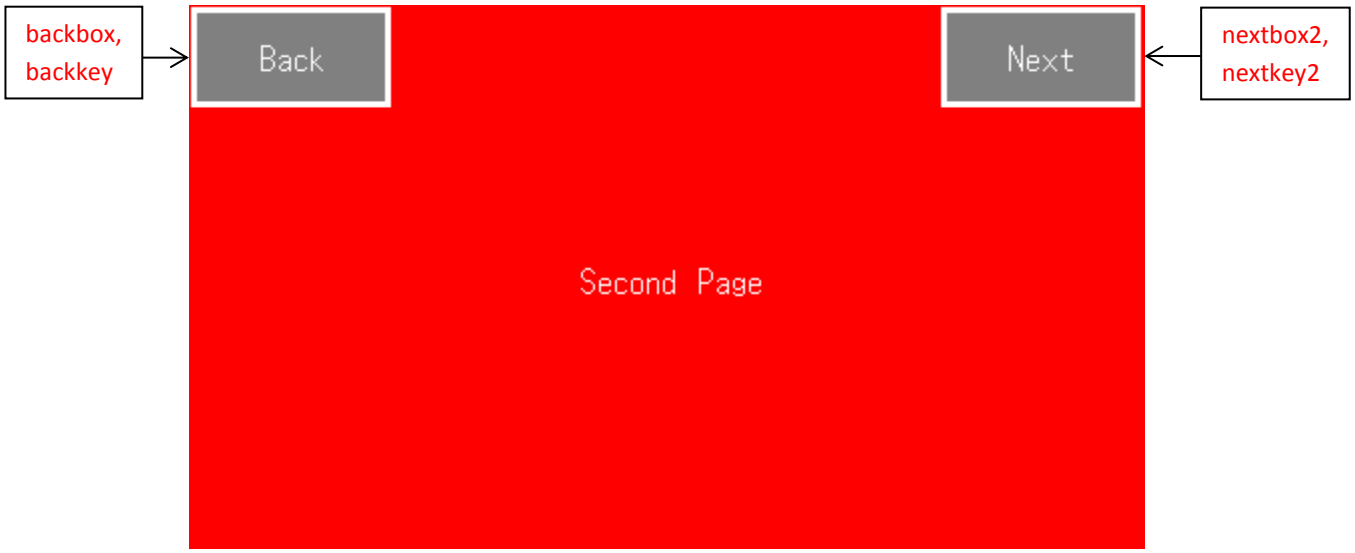


Fig. 2.48 *Screen shot to show the page redpg*

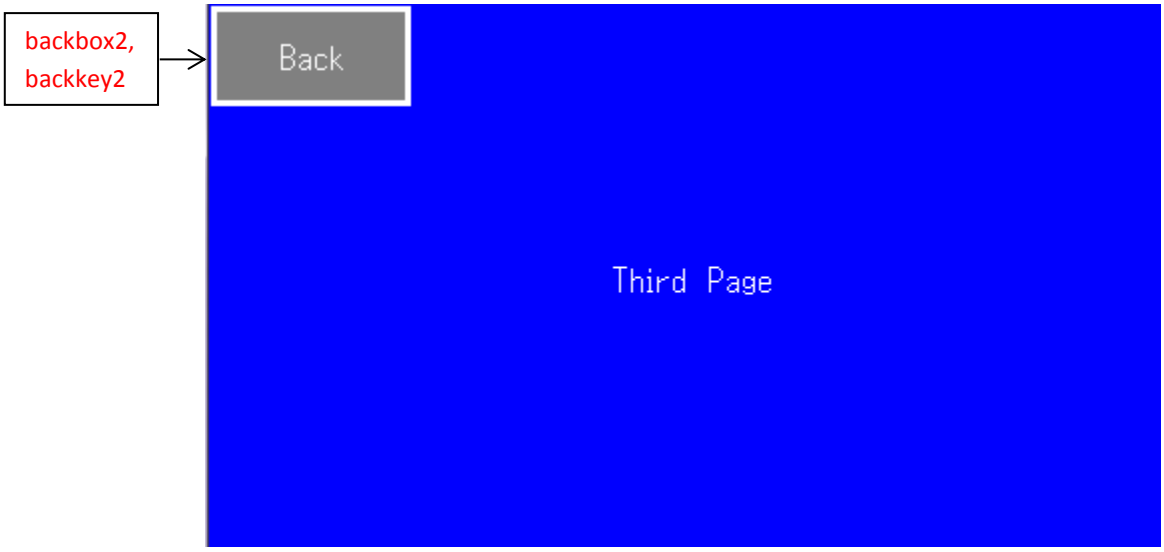


Fig. 2.49 *Screen shot to show the page bluepg*

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
STYLE command format inherit:
Style Header
STYLE(New Style name, Style name inherit)
Style Body
{
new style parameter 1 = new style value 1;
new style parameter 2 = new style value 2;
new style parameter 3 = new style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
SHOW command format:
SHOW(Page name or page component name);
}

```
//FILENAME: TU480a.mnu

STYLE(homepgst,Page)       //
{
update = changed;          //
back = green;              //
}
STYLE(redpgst,homepgst)    //
{
back = red;                //
}
STYLE(bluepgst,homepgst)   //
{
back = blue;               //
}
STYLE(Ascii16txst,Text)    //
{
font = Ascii16;            //
col = white;               //
maxRows = 1;               //
maxLen = 24;               //
}
STYLE(boxdrwst,Draw)       //
{
type = b;                  //
maxX = 100;                //
maxY = 50;                 //
width = 3;                 //
back = grey;               //
col = white;               //
}

PAGE(homepg,homepgst)                          //
{
POSN(240,136);                                 //
KEY(nextkey,[SHOW(redpg);],100,50,TOUCH);      //
DRAW(nextbox,100,50,boxdrwst);                 //
TEXT(nexttxt, "Press me",Ascii16txst);         //
}
PAGE(redpg,redpgst)                            //
{
POSN(240,136);                                 //
TEXT(redtxt, "Second Page",Ascii16txst);       //
POSN(50,25);                                   //
KEY(backkey,[SHOW(homepg);],100,50,TOUCH);     //
DRAW(backbox,100,50,boxdrwst);                 //
TEXT(backtxt, "Back",Ascii16txst);             //
POSN(430,25);                                  //
KEY(nextkey2,[SHOW(bluepg);],100,50,TOUCH);    //
DRAW(nextbox2,100,50,boxdrwst);                //
TEXT(nexttxt2, "Next",Ascii16txst);            //
}
PAGE(bluepg,bluepgst)                          //
{
POSN(240,136);                                 //
TEXT(bluetxt, "Third Page",Ascii16txst);       //
POSN(50,25);                                   //
KEY(backkey2,[SHOW(PREV_PAGE);],100,50,TOUCH); //
DRAW(backbox2,100,50,boxdrwst);                //
TEXT(backtxt2, "Back",Ascii16txst);            //
}
SHOW(homepg);                                  //
```

Fig. 2.50 *Example code to showing how three pages are linked together to provide user navigation*

The maximum amount of pages that can be created in iDev depends upon the TFT amount of page components in a page, module size and ram. In this guide, the TFT module size that have been used as an example is a 4.3" TFT module hence the filename *TU480a.mnu.* The bigger the module's screen size is (5.7" and 7.0"), the more ram is needed to display a page. This means that bigger modules have less maximum pages allowed. The main limiting factor is the amount of page components and iDev components in a page e.g. the more images, images or even animation is in a page, the more ram is needed to display the page. This emphasises the importance of memory allocation when creating iDev projects.

3. MANIPULATING DATA

In iDev, data handling is accomplished by the use of iDev commands. Data manipulation is crucial in almost all iDev projects as this widens the capabilities of the developer's iDev project. There is also a capability of storing the previous value of data in EEPROM and this value can then be accessed in other parts of the iDev project.

3.1. DATA STORAGE

Like in all other languages data is stored by the use of variables. In iDev, variables have styles that can be altered depending on its purpose. Also different data types can be stored and accessed as variables in iDev.

3.1.1. VARIABLE DATA STYLE

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}

Variable Data Styles		
Parameter	**Expected Values**	**Definition**
type	U8	data type for unsigned 8 bit integer = 0 to 255 **(default)**
	U16	data type for unsigned 16 bit integer = 0 to 65,535
	U32	data type for unsigned 32 bit integer= 0 to 4,294,967,295
	S8	data type for signed 8 bit integer = -128 to 127
	S16	data type for signed 16 bit integer = -32,768 to 32,767
	S32	data type for signed 32 bit integer = -2,147,483,648 to 2,147,483,647
	TEXT	data type to store text strings
	FLOAT	data type for float with higher resolution – up to 17 decimal places, used for numbers that require high degrees of accuracy/decimal places

	POINTER	pointer to use for page components
	FILE	used to create file object variables (see Chapter 7)
length	1 to 8192	for text data, specify the maximum length that can be stored **(default = 32)**
decimal	0 to 17	for float, specify the number of decimal places **(default = 2)**
format	d	RTC format for day: day of month with leading zeros = 01-31 **(default = disabled)**
	j	RTC format for day: day of month without leading zeros = 1-31
	S	RTC format for day: ordinal suffix for day of month = st, nd, rd, th (usage explained in Chapter 6)
	F	RTC format for month: full textual representation of month = January-December
	m	RTC format for month: numeric representation of month with leading zeros = 01-12
	M	RTC format for month: short textual representation of month with three letters = Jan-Dec
	n	RTC format for month: numeric representation of month without leading zeros = 1-12
	Y	RTC format for year: full numeric representation of year with 4 digits = 1900-2099
	y	RTC format for year: two digit representation of year = 00-99
	a	RTC format for time: lowercase ante meridiem and post meridiem = am, pm
	A	RTC format for time: uppercase ante meridiem and post meridiem = AM, PM
	g	RTC format for time: 12-hour format of hour without leading zeros = 1-12
	G	RTC format for time: 24-hour format of hour without leading zeros = 0-23
	h	RTC format for time: 12-hour format of hour with leading zeros = 01-12
	H	RTC format for time: 24-hour format of hour with leading zeros = 00-23
	i	RTC format for time: format of minutes with leading zeros = 00-59
	s	RTC format for time: format of zeros = 00-59
location	SDRAM or RAM	specify to store data in SDRAM **(default)**
	EEPROM	specify to store data in EEPROM (see Chapter 8.5)

Fig. 3.1 *Table to describe different variable data style parameters and expected values*

The variable data style is applied in iDev using the same style command format. Sometimes in iDev, variables that are used don't require the other data style parameters to be altered. There are built-in data styles in iDev that are ready to use without the need to create a style if the built in parameters are appropriate for the variable's purpose.

Built in Style	type	location	decimal	length
U8	U8	SDRAM	N/A	
U8E	U8	EEPROM		
U16	U16	SDRAM		

U16E	U16	EEPROM		
U32	U32	SDRAM		
U32E	U32	EEPROM		
S8	S8	SDRAM		
S8E	S8	EEPROM		
S16	S16	SDRAM		
S16E	S16	EEPROM		
S32	S32	SDRAM		
S32E	S32	EEPROM		
PTR	POINTER	SDRAM		
PTRE	POINTER	EEPROM		
TXT	TEXT	SDRAM	N/A	32
TXTE	TEXT	EEPROM		32
FLT1	FLOAT	SDRAM	1	N/A
FLT1E	FLOAT	EEPROM	1	
FLT2	FLOAT	SDRAM	2	
FLT2E	FLOAT	EEPROM	2	
FLT3	FLOAT	SDRAM	3	
FLT3E	FLOAT	EEPROM	3	
FLT4	FLOAT	SDRAM	4	
FLT4E	FLOAT	EEPROM	4	

Fig. 3.2 *Table to describe the built-in variable data styles in iDev*

3.1.2. DECLARING VARIABLES

In iDev, variables consist of the variable name, starting/default value and style. The variable names used in iDev must start with a letter or "_". The use of a leading underscore is NOT recommended and usual naming convention uses lower case letters when naming variables. Like in any other programming languages variables are declared in the beginning of the main menu file. Values stored in a variable can be stored in the EEPROM which stores the last stored value even though the module is turned off (non-volatile memory). The stored value is accessed as a starting value when the module is powered on again.

VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);

```
//FILENAME: TU480a.mnu

STYLE(flt15st,Data)          //
{
type = FLOAT;                //
decimal = 15;                //
}

VAR(myvar1,1,U8E);           //
VAR(myvar2,1,S8);            //
VAR(myvar3,0,flt15st);       //
VAR(mytextvar,"first",TXT);  //
```

Fig. 3.3 *Example code exhibiting the use of the VAR command*

Sometimes the developer might change the variable style from an EEPROM stored

variable to an SDCARD stored variable (i.e. change from U8 to U8E); this can cause error/s in the iDev project. The error is caused because the variable name is stored and allocated in the EEPROM of the module and when powered on, the same variable is retrieved from the same location. Since the variable is not located in the EEPROM anymore, an error occurs. There are two ways in fixing this issue. First, the developer can use the *RESET(EEPROM)* command to clear the EEPROM. It is important to place the *RESET(EEPROM)* command before the variables are stated in the main menu file for this method to work properly. This reset is a one-time process so after the module is powered on once the *RESET(EEPROM)* command can be removed. Another way is not as efficient as the previous method; the developer can change the name of the variable with the data style location being altered. This means that in the whole iDev project, the developer has to rename when the variable is used. This method is inefficient because the variable is still stored in the EEPROM but it isn't accessed; this causes some EEPROM storage space being used even though it is not needed.

3.1.3. TEXT VARIABLE UPDATE – LOAD

The variable set as a text can also be used as a text component of which the value can be updated depending on the contents of the text variable. This is achieved by using the *LOAD* command. Multiple text variables can also be loaded in one text variable using the *LOAD* command. The text data source can either be raw text enclosed in quotation marks or text data variable.

LOAD command format to change stored text data:
LOAD(Destination variable name, Text data source);
LOAD command format to change stored text data from multiple sources:
LOAD(Destination variable name, Text data source1, Text data source2…);

An example code is created to manipulate the text component in the page by using the *LOAD* command to change the contents of the text component. Three text variables were created namely, *myvar*, *myvar1* and *myvar2* each containing different text components.

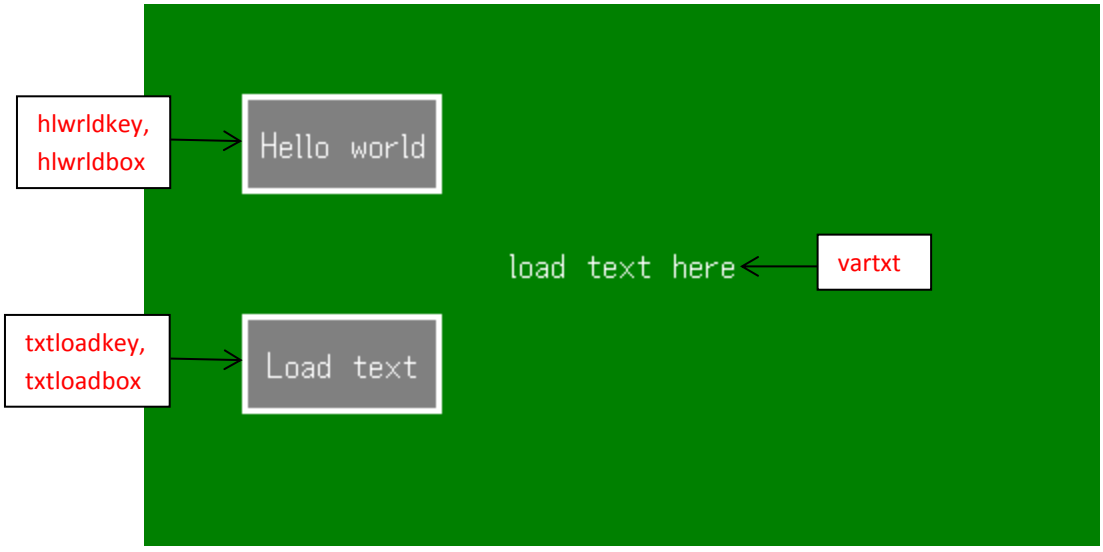


Fig. 3.4 *Screen shot showing what will be displayed before the buttons are pressed*

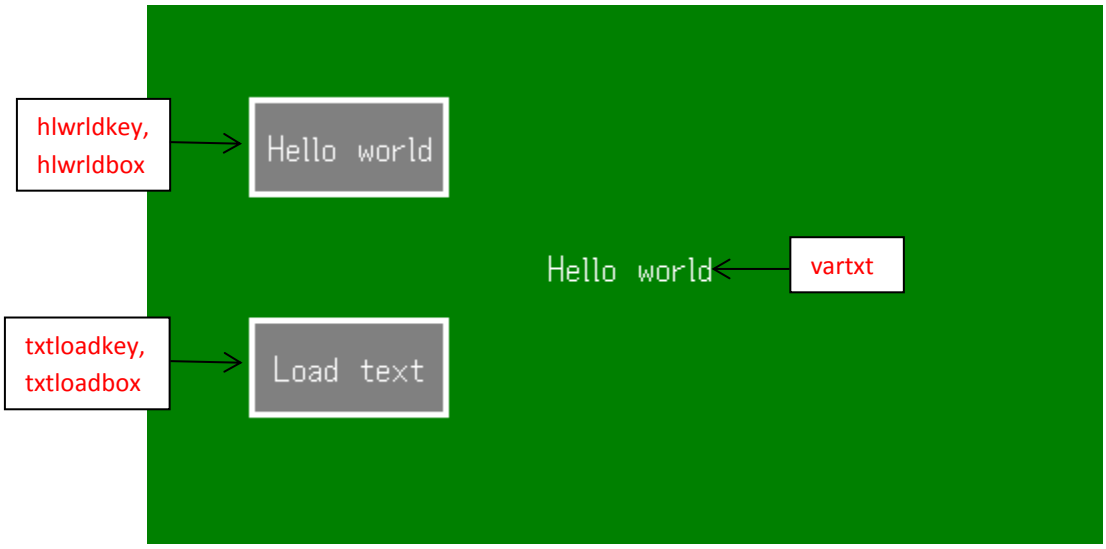


Fig. 3.5 *Screen shot to demonstrate what will be shown when the Hello world button is pressed*

Fig 3.6 *Screen shot to demonstrate what will be shown when the Load text button is pressed*

VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
LOAD command format to change stored text data:
LOAD(Destination variable name, Text data source);
LOAD command format to change stored text data from multiple sources:
LOAD(Destination variable name, Text data source1, Text data source2…);
SHOW(Page name or page component name);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;

```
//FILENAME: TU480a.mnu

VAR(mytxtvar,"load text here",TXT);     //
VAR(mytxtvar1,"mytxtvar1",TXT);         //
VAR(mytxtvar2," with mytxtvar2",TXT);   //

STYLE(homepgst,Page)      //
{
back = green;             //
}
STYLE(Ascii16txst,Text)   //
{
font = Ascii16;           //
col = white;              //
maxRows = 1;              //
maxLen = 32;              //
}
STYLE(boxdrwst,Draw)      //
{
type = b;                 //
maxX = 100;               //
maxY = 50;                //
width = 3;                //
back = grey;              //
col = white;              //
}

PAGE(homepg,homepgst)                          //
{
POSN(100,70);                                  //
KEY(hlwrldkey,hlwrldfunc,100,50,TOUCH);        //
DRAW(hlwrldbox,100,50,boxdrwst);               //
TEXT(hlwrldtxt,"Hello world",Ascii16txst);     //
POSN(+0,170);                                  //
KEY(txtloadkey,txtloadfunc,100,50,TOUCH);      //
DRAW(txtloadbox,100,50,boxdrwst);              //
TEXT(txtloadtxt,"Load text",Ascii16txst);      //
POSN(240,130);
TEXT(vartxt,mytxtvar,Ascii16txst);             //
}

FUNC(hlwrldfunc)                          //
{
LOAD(mytxtvar,"Hello world");             //
TEXT(vartxt,mytxtvar);;                   //
}
FUNC(txtloadfunc)                         //
{
LOAD(mytxtvar,mytxtvar1,mytxtvar2);       //
TEXT(vartxt,mytxtvar);;                   //
}
SHOW(homepg);                             //
```

Fig. 3.7 *Example code to demonstrate how text data can be manipulated by using the LOAD command*

From the example code in Fig 3.7, the functions *hlwrldfunc* and *txtloadfunc* has the line *TEXT(vartxt,mytxtvar);;*. This line is important to update the text component *vartxt* appropriately and display the correct updated text data when those functions are demanded. If that line of code isn't present then the text component *vartxt* would not be updated on the display although the contents have been changed.
It is important to remember that the text data style has a length parameter which determines the maximum amount of text data that can be stored in a text variable. In the example above the default length parameter of 32 is enough so it isn't

changed, so the length parameter can be changed as appropriate. Another factor to remember is the maximum length in the text component style works similarly to the length parameter in text data styles. Again the value of the *maxLen* parameter is changed in the example code above to accommodate all the text data in the text component *vartxt*.

3.1.4. INTEGER/FLOAT VARIABLE – LOAD

The *LOAD* command can be used to change the value stored in an integer or float data variable. The number that can be stored in an integer or float variable depends on the type i.e. if a U8 integer variable type is used then the variable can hold any integer value from 0-255. For better guidance regarding the different types of integer and float variables that can be used in iDev, refer to Chapter 3.1.1 of this guide.

LOAD command format to change stored integer/ float data:
LOAD(Destination variable name, Int/float data source);
LOAD command format to change stored integer/float data from multiple sources:
LOAD(Destination variable name, Int/float data source1, Int/float data source2…);

A common use of the integer and float variables is setting flags. In programming, flags refer to fixed values that help the developer indicate the state, mode or behaviour if his/her code i.e. its purpose is to indicate when a point in the processing has been reached. An example below uses a 'flag' to determine whether the key component has been pressed or not, and processes are carried out depending on the state of the flag.

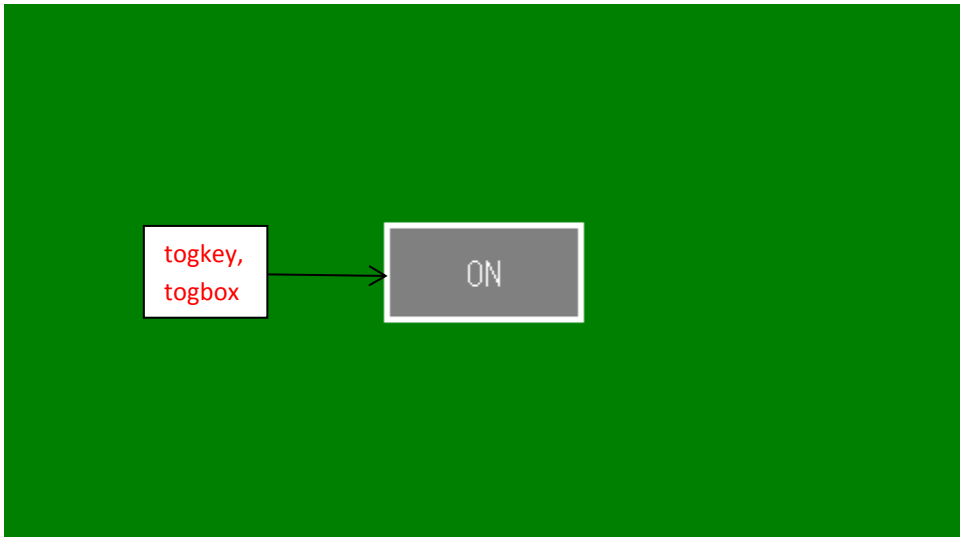


Fig. 3.8 *Screen shot to demonstrating when toggle button state is pressed (ON)*

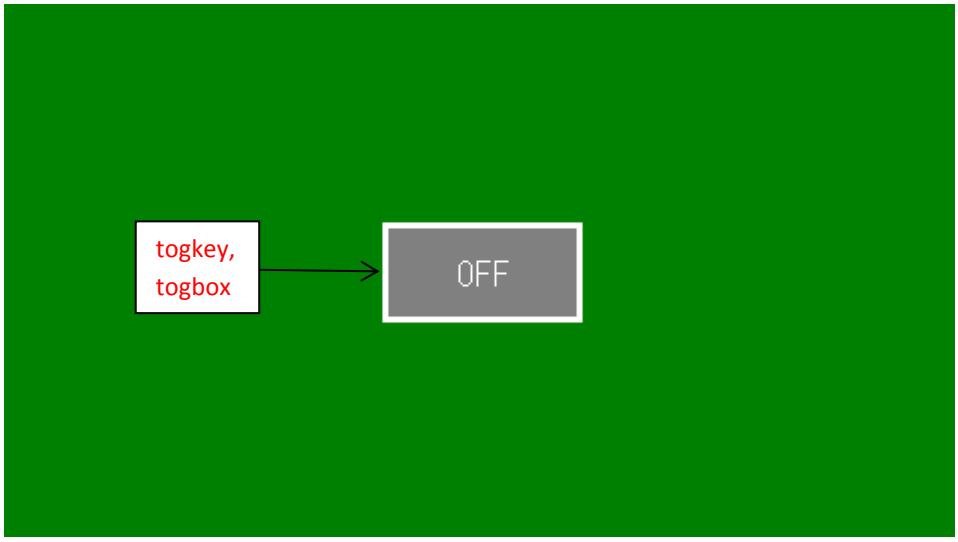


Fig. 3.9 *Screen shot to demonstrating when toggle button state is pressed (OFF)*

VAR command format:
VAR(Variable name, Starting value, Variable Style);
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body

{

Function contents…

}

LOAD command format to change stored integer/ float data:

LOAD(Destination variable name, Int/float data source);

SHOW command format:

SHOW(Page name or page component name);

HIDE command format:

HIDE(Page name or page component name);

```
//FILENAME: TU480a.mnu

VAR(togvar,0,U8);          //

STYLE(homepgst,Page)       //
{
back = green;              //
}
STYLE(Ascii16txst,Text)    //
{
font = Ascii16;            //
col = white;               //
maxRows = 1;               //
maxLen = 32;               //
}
STYLE(boxdrwst,Draw)       //
{
type = b;                  //
maxX = 100;                //
maxY = 50;                 //
width = 3;                 //
back = grey;               //
col = white;               //
}

PAGE(homepg,homepgst)                  //
{
POSN(240,135);                         //
KEY(togkey,istogfunc,100,50,TOUCH);    //
DRAW(togbox,100,50,boxdrwst);          //
TEXT(ontxt,"ON",Ascii16txst);          //
TEXT(offtxt, "OFF",Ascii16txst);       //
HIDE(offtxt);                          //
}
FUNC(istogfunc)      //
{
IF(togvar = 0?[RUN(togfunc);]:[RUN(nottogfunc);]);
//
}
FUNC(togfunc)        //
{
HIDE(ontxt);         //
SHOW(offtxt);        //
LOAD(togvar,1);;     //
}
FUNC(nottogfunc)     //
{
HIDE(offtxt);        //
SHOW(ontxt);         //
LOAD(togvar,0);;     //
}
SHOW(homepg);        //
```

Fig. 3.10 *Example code demonstrating how integer data variables are manipulated*

From the example code in Fig 3.8, the state of the key component is determined by an *IF* statement. The *IF* command is explained thoroughly in Chapter 3.4 of this guide so for now if it should be ignored to avoid confusion. The U8 integer variable *togvar* is used as a 'flag', when its value is 0 then the function *togfunc* is undertaken which effectively shows the text component *offtxt* and hides the text component *ontxt* but on the other hand when the value is 1, then the function *nottogfunc* is undertaken which does the opposite. The *LOAD* command is used inside the *togfunc* and *nottogfunc* functions, to enable the toggle button to work in switching states indefinitely. If these lines of code are removed, then the toggle button will only work once. The starting value of a variable determines the starting state of the 'flag' so it is important to take that into account when using 'flags' in an iDev project. For better understanding of the *LOAD* and *VAR* commands then it might be worth trying to change the starting state of the toggle button in the example code above from 'OFF' to 'ON'.

In iDev, text data can be assigned to an integer or float variable by the use of *LOAD* command. This can be useful in storing calculations from a text component.

```
//FILENAME: TU480a.mnu

LOAD(myintvar,mytextvar);   //transfer text variable to int variable
LOAD(myintvar,"145","23");  //transfer text data to int variable
```

Fig. 3.11 *Load command example demonstrating text data contents moved to an integer or float variable*

In the example above, the variable *mytextvar* is analysed until a non-valid numeric value is found i.e. if *mytextvar* contains "1689 pounds", then the integer 1689 is stored in the integer variable *myintvar* provided that the variable type can accommodate this value (has to be U16 or U32). The second example simply combines the text data into one integer value and stores in the variable *myintvar*. In effect, the contents of *myintvar* are then changed to *14523*.

3.1.5. POINTER

Pointer is a certain variable type that is used to locate another variable. The use of pointers allows the developer to perform data related operations quicker. In iDev, pointers can be used to point to other pointers, iDev components and page components but in most cases pointers are used to direct to another variable.

VAR command format for **pointers**:
VAR(Pointer variable name>"Shared destination value", Pointer type);

The pointer variable command format consists of three parts. The pointer variable name refers to the unique name that the pointer is given when declared. The shared destination value refers to the common part of the final destination value. Lastly, the pointer type is used to determine whether to store the pointer in *SDRAM* or *EEPROM*. The *LOAD* command with pointers has a different format from the other *LOAD* command formats introduced so far.

LOAD command format for using pointers:
LOAD(Pointer variable name>"Shared destination value", Destination Identifier);

The *LOAD* command is used to assign or change the current pointer value by the parameter destination identifier.

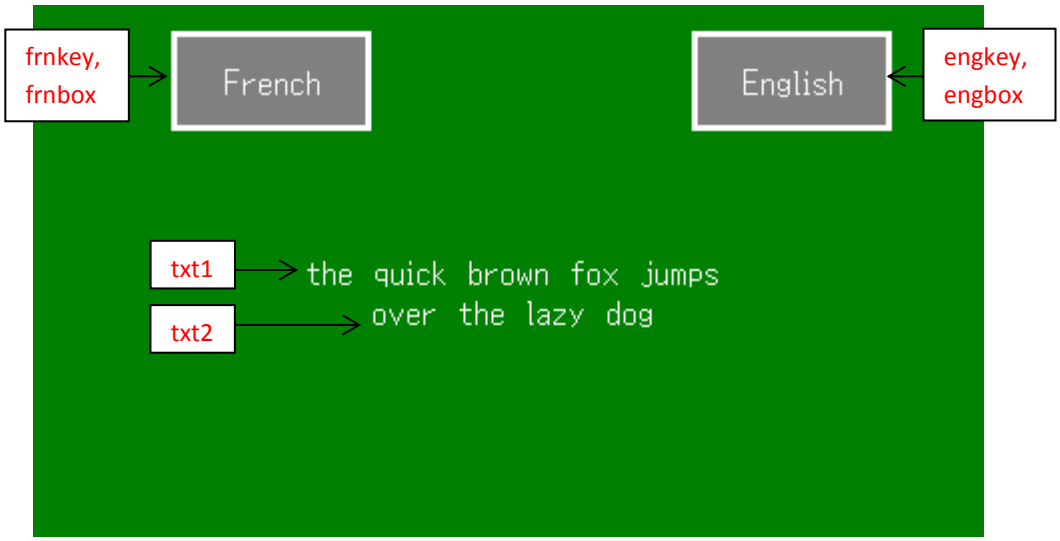


Fig. 3.12 *Screen shot to show what would be expected from the screen when the English button is pressed*

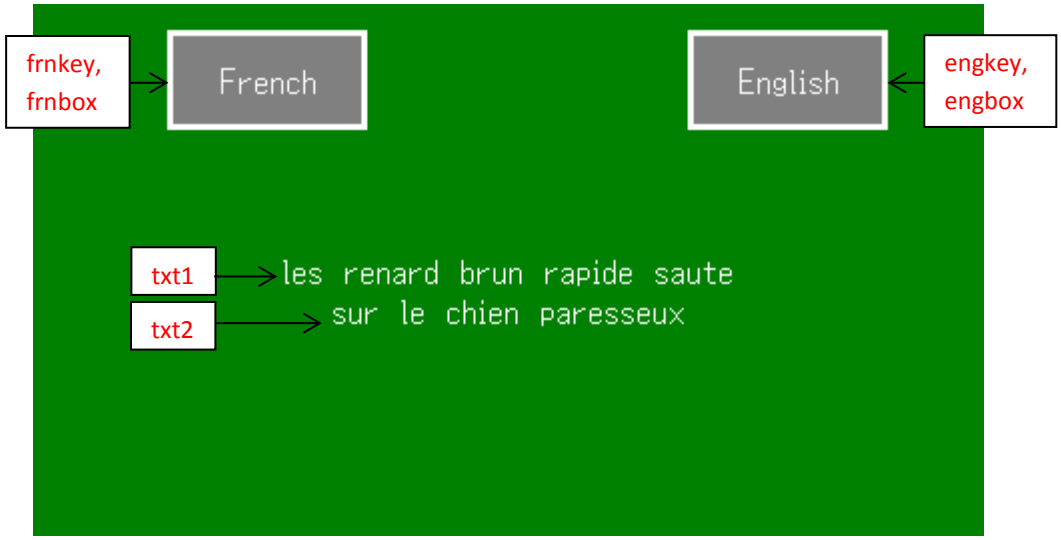


Fig. 3.13 *Screen shot showing what to expect from the screen when the French button is pressed*

VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
VAR command format for pointers:
VAR(Pointer variable name>"Shared destination value", Pointer type);
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
LOAD command format for using pointers:
LOAD(Pointer variable name>"Shared destination value", Destination Identifier);
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

VAR(lang,0,U8);                                  //
VAR(ln10,"the quick brown fox jumps",TXT);       //
VAR(ln11,"les renard brun rapide saute",TXT);
//
VAR(ln20,"over the lazy dog",TXT);               //
VAR(ln21,"sur le chien paresseux",TXT);          //
VAR(lnptr1>"",PTR);                              //
VAR(lnptr2>"",PTR);                              //

STYLE(homepgst,Page)          //
{
back = green;                 //
}
STYLE(Ascii16txst,Text)       //
{
font = Ascii16;               //
col = white;                  //
maxRows = 1;                  //
maxLen = 32;                  //
}
STYLE(boxdrwst,Draw)          //
{
type = b;                     //
maxX = 100;                   //
maxY = 50;                    //
width = 3;                    //
back = grey;                  //
col = white;                  //
}
FUNC(langfunc)                //
{
LOAD(lnptr1>"ln1",lang);      //
LOAD(lnptr2>"ln2",lang);      //
TEXT(txt1,lnptr1);            //
TEXT(txt2,lnptr2);            //
}

PAGE(homepg,homepgst)                    //
{
POSN(240,135);                           //
TEXT(txt1,lnptr1,Ascii16txst);           //
POSN(+0,+20);                            //
TEXT(txt2,lnptr2,Ascii16txst);           //
POSN(120,40);                            //
KEY(frnkey,[LOAD(lang,1);RUN(langfunc);;],100,50,TOUCH);
//
DRAW(frnbox,100,50,boxdrwst);            //
TEXT(frntxt,"French",Ascii16txst);       //
POSN(380,+0);                            //
KEY(engkey,[LOAD(lang,0);RUN(langfunc);;],100,50,TOUCH);
//
DRAW(engbox,100,50,boxdrwst);            //
TEXT(engtxt,"English",Ascii16txst);      //
}
RUN(langfunc);                           //
SHOW(homepg);                            //
```

Fig. 3.14 *Example code to show how pointer variables are used and manipulated in the iDev language*

The example code in Fig 3.14 uses pointers to display appropriate text depending on which button is pressed. The two buttons created switches the language of the text component from English to French and vice-versa. The text data are stored in

text variables. The pointers are used to display the appropriate text on the screen. The function *langfunc* contains *LOAD* commands that manipulate the pointers to direct the correct text variable for the text component. The diagram below best explains the method of pointers in the iDev language. It is important to keep on referring to the example code in Fig 3.12 when looking at the diagram in Fig. 3.13 for a better understanding.

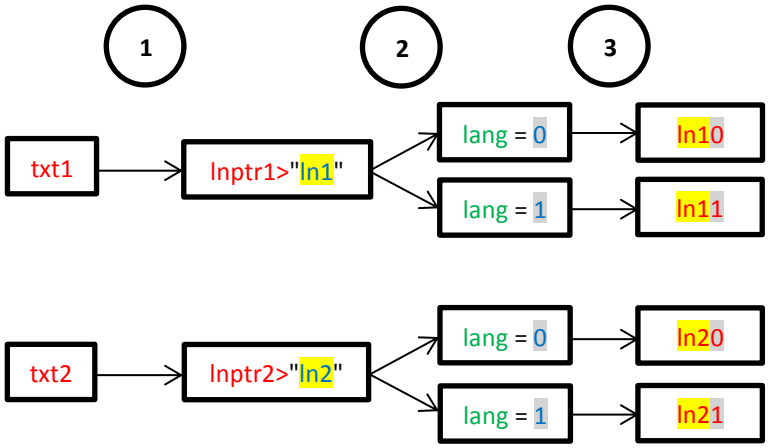


Fig. 3.15 *Diagram to explain pointers in iDev*

The text components *txt1* and *txt2* in the example code in Fig 3.14 uses a pointer variable as a source i.e. text component *txt1* takes its text data from pointer variable *lnptr1,* so if *lnptr1* points to a text variable containing "hello" then the text component *txt1* would display "hello" on the screen and the same with *txt2* and *lnptr2*; this is evident in **1** from the diagram. At **2** from the diagram, the pointer variable *lnptr1* and *lnptr2* followed by the common destination value (*ln1* and *ln2)* are used to choose which text data to display. Shared destination value refers to the common portion of the final destination value; it is highlighted yellow in the diagram above to demonstrate why the word 'shared' is used. If the developer wants to display the text data contained in the text variable *ln11* (in the example above, *ln10*, *ln11*, *ln20*, *ln21* are text variables), then the value of the variable *lang* has to be 1 as seen at **3.** The same method is initiated when accessing the other text variables. The diagram above explains what the function *langfunc* does before it updates the text components on the screen. Also it is important to note that the pointer variable itself does not determine the final destination value but the destination identifier as well. The destination identifier from the example above is the variable *lang* and this variable value is changed when the key component *frnkey* or *engkey* is pressed. This can be seen as one of the inline commands (*LOAD(lang,1)* for *frnkey* and *LOAD(lang,0)* for *engkey*) that are set when the key component is created in the example above. The destination identifier variable *lang* is an unsigned 8-bit integer variable so the final destination value for the pointer can be *ln10* to *ln1255*, but it is also possible to use a destination identifier variable that is a text variable. If the variable *lang* is changed to a text variable and the inline commands in the key components are changed to *LOAD(lang,a)* or *LOAD(lang,b)* then the possible final destination values would be *ln1a*, *ln1b*, *ln2a*, *ln2b* as opposed to *ln10*, *ln11*, *ln20*, *ln21.* Although it is possible to use text destination identifier variable it is recommended to use integer variables as destination identifiers as this allows calculations to be carried out.

In iDev, pointers to pointers are supported as well. The similar command format is used to apply this.

```
//FILENAME: TU480a.mnu

VAR(myintvar,1234,U16);       //
VAR(myptr1>"myintvar",PTR);   //
VAR(myptr2>"myptr1",PTR);     //
VAR(myptr3>"myptr2",PTR);     //

TEXT(mytext,myptr1);          //
LOAD(myptr3,4321);            //
```

Fig. 3.16 *Example code showing how pointer to pointer is applied in iDev*

Looking at the example above, the text component *mytext* is set to have the text data contained in pointer *myptr1* but *myptr1* is set to be pointing at the variable *myintvar*; in effect the text component *mytext* displays 1234 on the page. The *LOAD* command in the example above is used to change the contents of *myptr3* to 4321, since the *myptr3* is pointed at *myptr2; myptr2* pointing at *myptr1* and lastly *myptr1* to *myintvar* then the effective final destination of the pointer *myptr3* is *myintvar*. The contents of the variable *myintvar* are changed from 1234 to 4321. There is no limit on how many times pointer to pointer instances can occur in iDev, the pointer variables are checked until a non-pointer value is found.

3.1.6. ARRAY

If the developer requires multiple values to be grouped into one unit then arrays become useful e.g. amount of rainfall each day in a one-week period. Instead of creating a variable to store each value of rainfall for different days, all the values can be stored in one array. Consequently, four arrays can represent the amount of rainfall per day in approximately a month (approximately four weeks in a month so four arrays are used and so 28 days for 28 values). In iDev, arrays up to 4 dimensions are supported. The maximum number of elements or size that an array can accommodate per dimension is 32,767. Currently the types of data that can be stored in an array are integers but future support for text data will be released. For better comprehension, each type of array is explained in separate parts.

One Dimensional Array explained
VAR command format for **one-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D);

In iDev, all array elements are defined by using zero-based indexing. Elements are the contents of an array where values are stored. The array element values are changed by the use of *LOAD* command. The *LOAD* command is explained further in Chapter 3.3 of this guide. The values can either be changed at the same time or only one element alone. The element value to be defined in an array can either be an integer value or an integer variable. Up to 15 elements can be changed at a time in a one-dimensional array because of the 16-parameter limit in iDev.

LOAD command format to change single element in one-dimensional array:
LOAD(Array name.1D, New element value/variable);
LOAD command format to transfer single element to variable:
LOAD(Variable name, Array name.1D);
LOAD command format to change all elements with a single value in one-dimensional array:
LOAD(Array name, Single value);
LOAD command format to change multiple elements in one-dimensional array:
LOAD(Array name,1st element value,2nd element value, 3rd element value…);
LOAD command format to pass array elements to serial interface/text variable or another array:
LOAD(Destination of array elements, Array source name);
LOAD command format to pass all array elements to text component:
TEXT(Text component, Array source name);
LOAD command format when array elements come from serial interface (serial buffer):
LOAD(Array name, Serial interface source);

```
//FILENAME: TU480a.mnu

//One-dimensional array is declared
VAR(myintvar,4,U8);             //
VAR(mytxtvar,"154",TXT);        //

VAR(myarray,0,U8,4);            //
VAR(myarray2,0,U8,4);           //

//Inside a Page or Function
TEXT(mytext,"hello",mytxtst);   //

LOAD(myarray.0,1);          //change 1st element of array value to 1
LOAD(myarray.2,36);         //change 2nd element of array value to 36
LOAD(myarray.3,myintvar);   //change 3rd element value of array to myintvar

LOAD(myintvar,myarray.1);  //Transfer single element to variable
LOAD(myarray,52);          //Change all elements with a single value
LOAD(myarray,52,48,27,53); //Change multiple element value
LOAD(RS2,myarray);         //Transfer array elements to serial interface
LOAD(mytext,myarray);      //Transfer array elements to text component
LOAD(mytxtvar,myarray);    //Transfer array elements to text variable
LOAD(myarray2,myarray);    //Transfer array elements to another array
LOAD(myarray,RS2);
//Transfer contents from serial interface to array (serial buffer)
```

Fig. 3.17 *Example to show how elements in a one-dimensional array are manipulated in different conditions*

The array *myarray* and *myarray2* is created as a 4 element array with unsigned 8-bit integer data and initial values of 0. It is evident from the example code above how simple it is to change specific elements in an array. When multiple arrays are changed and some of the elements in the array weren't specified then the element's current value wouldn't be replaced.

The line of code: *LOAD(myarray,52,48,27,53)* and the table below describes how array elements are assigned when multiple elements are changed.

1st array element	2nd array element	3rd array element	4th array element
myarray.0	myarray.1	myarray.2	myarray.3
52	48	27	53

Fig. 3.18 *Table describing how elements are assigned when multiple elements are changed using the LOAD command*

If the developer requires using arrays as a serial buffer, there are loads of factors that decide how the data received is divided when transferred to the elements. Factors such as size of each packet of data received, type of encoding or size of array are the typical factors than can affect data allocation to the array. The interfaces in iDev are introduced properly in Chapter 4.

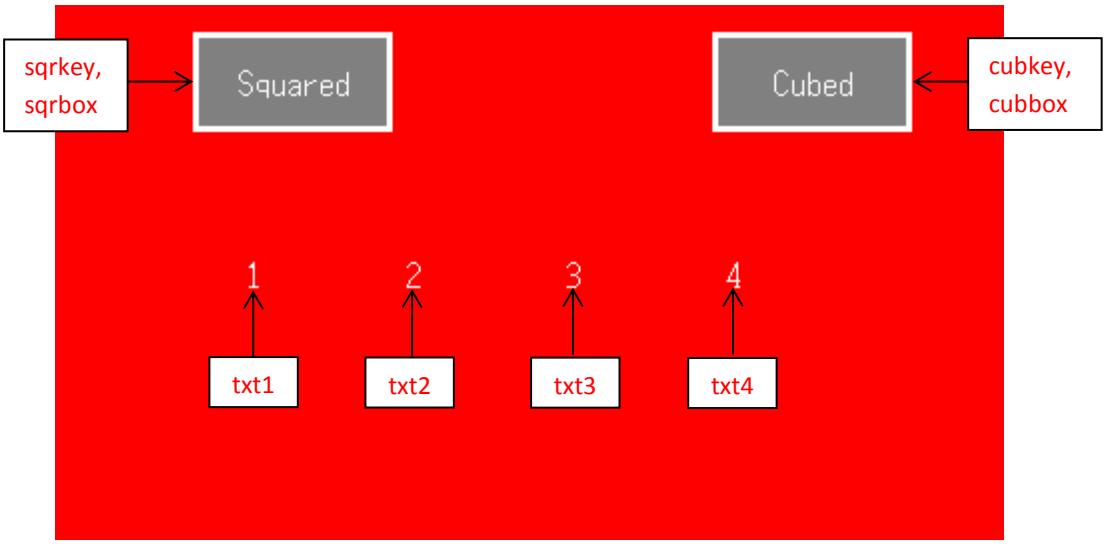


Fig. 3.19 *Screen shot to show is displayed before the buttons are pressed*

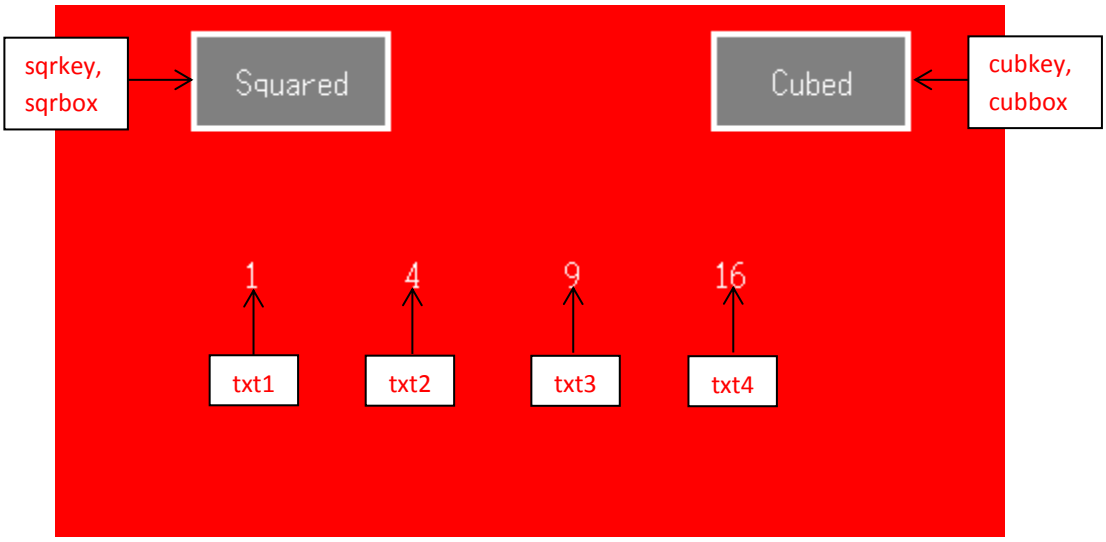


Fig. 3.20 *Screen shot demonstrating the changes in the screen when the squared button is pressed*

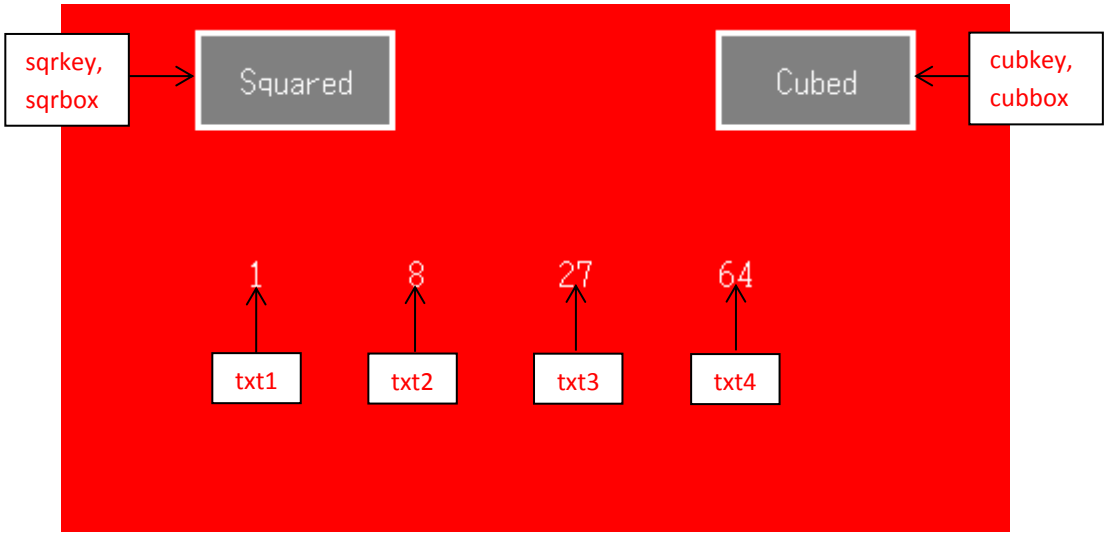


Fig. 3.21 *Screen shot demonstrating the changes in the screen when the cubed button is pressed*

VAR command format for **one-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D);
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
LOAD command format to change multiple elements in one-dimensional array:
LOAD(Array name,1st element value,2nd element value, 3rd element value…);
LOAD command format to pass all array elements to text component:
TEXT(Text component, Array source name);
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

VAR(myarray,0,U8,4);       //

STYLE(homepgst,Page)       //
{
back = green;              //
}
STYLE(Ascii16txst,Text)    //
{
font = Ascii16;            //
col = white;               //
maxRows = 1;               //
maxLen = 32;               //
}
STYLE(boxdrwst,Draw)       //
{
type = b;                  //
maxX = 100;                //
maxY = 50;                 //
width = 3;                 //
back = grey;               //
col = white;               //
}

PAGE(homepg,homepgst)          //
{
POSN(100,136);                 //
TEXT(txt1,"1",Ascii16txst);    //
POSN(+80,+0);                  //
TEXT(txt2,"2",Ascii16txst);    //
POSN(+80,+0);                  //
TEXT(txt3,"3",Ascii16txst);    //
POSN(+80,+0);                  //
TEXT(txt4,"4",Ascii16txst);    //

POSN(120,40);                            //
KEY(sqrkey,sqrfunc,100,50,TOUCH);        //
DRAW(sqrbox,100,50,boxdrwst);            //
TEXT(sqrtxt,"Squared",Ascii16txst);      //
POSN(380,+0);                            //
KEY(cubkey,cubfunc,100,50,TOUCH);        //
DRAW(cubbox,100,50,boxdrwst);            //
TEXT(cubtxt,"Cubed",Ascii16txst);        //
}

FUNC(sqrfunc)              //
{
LOAD(myarray,1,4,9,16);    //
TEXT(txt1,myarray.0);      //
TEXT(txt2,myarray.1);      //
TEXT(txt3,myarray.2);      //
TEXT(txt4,myarray.3);;     //
}
FUNC(cubfunc)              //
{
LOAD(myarray,1,8,27,64);   //
TEXT(txt1,myarray.0);      //
TEXT(txt2,myarray.1);      //
TEXT(txt3,myarray.2);      //
TEXT(txt4,myarray.3);;     //
}
SHOW(homepg);              //
```

Fig. 3.22 *Example code to show how one-dimensional arrays are used in iDev*

The code in Fig 3.22 is created to produce two buttons that changes the text components displayed to a squared or cubed value. An array named *myarray* with 4 elements, unsigned 8 bit integer data with initial values of 0 is created. The 'buttons' are created in the same way as how it is created in previous example codes but just the text component is changed to an appropriate one to signify the button's purpose. When either one of the buttons is pressed then the functions to change the transfer the elements from the array to the text components are carried out. Looking at the function *sqrfunc*, there is a *LOAD* command which changes the elements contained in the one-dimensional array to 1, 4, 9 and 16. The rest of the commands in the function are added to change the text in the text components to the contents of the array. The same method is applied to the other function.

Two-dimensional Array explained
VAR command format for **two-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D,Size2D);

Two-dimensional arrays in iDev follow the similar arrangement with the one dimensional arrays. In iDev it would be easier to think of multidimensional arrays as group of sets; the first dimension is the highest level set and the second dimension is the second level set i.e. the elements in the second dimension is a subset of the elements in the first dimension. A two dimensional array can be pictured as a table with rows as the 1st dimension and column as the 2nd dimension. Since another dimension is introduced, the command format has one added parameter to define the second dimension of an array. The *LOAD* command is used to manipulate elements in a two-dimensional array just like in the one-dimensional array. Up to 15 elements can be changed at a time in a two-dimensional array because of the 16-parameter limit in iDev.

LOAD command format to change single element in two-dimensional array:
LOAD(Array name.1D.2D, New element value/variable);
LOAD command format to transfer single element to variable:
LOAD(Variable name, Array name.1D.2D);
LOAD command format to change multiple elements in 1st dimension of a two-dimensional array:
LOAD(Array name.1D,1st element value,2nd element value,3rd element value…);
LOAD command format to pass array elements in the specified 1st dimension to serial interface/ text variable or another array:
LOAD(Destination of array elements, Array source name.1D);
LOAD command format to pass all array elements in the specified 1st dimension to text component:
TEXT(Text component, Array source name.1D);
LOAD command format when all array elements come from serial interface (serial buffer):
LOAD(Array name, Serial interface source);
LOAD command format when 1st dimension array elements come from serial interface (serial buffer):
LOAD(Array name.1D,Serial interface source);

```
//FILENAME: TU480a.mnu

//Two-dimensional array is declared
VAR(myintvar,0,U8);              //
VAR(mytxtvar,"24",TXT);          //

VAR(myarray,0,U8,3,5);           //
VAR(myarray2,0,U8,3,5);          //

//Inside a Page or Function
TEXT(mytext,"hello",mytxtst);    //

LOAD(myarray.0.2,25);            //
LOAD(myarray.2.1,86);            //
LOAD(myarray.1.2,myintvar);      //

LOAD(myintvar,myarray.1.3);      //Transfer single element to variable
LOAD(myarray.2,52,28,17,73,97);
//Change multiple element values in specified 1st dimension of the array
LOAD(myarray.0,25,79,2,87,62);
//Change multiple element values in specified 1st dimension of the array
LOAD(RS2,myarray.1);
// Transfer array elements in specified 1st dimension of the array to serial
interface
LOAD(mytext,myarray.2);
//Transfer array elements in specified 1st dimension of the array to text
component
LOAD(mytxtvar,myarray.0);
//Transfer array elements in specified 1st dimension of the array to text
variable
LOAD(myarray2,myarray.1);
//Transfer array elements in specified 1st dimension of the array to another
array
LOAD(myarray,RS2);
// Transfer contents from serial interface to array (serial buffer)
LOAD(myarray.1D,RS2);
// Transfer contents from serial interface to the specified 1st dimension in
the array (serial buffer)
```

Fig. 3.23 *Example to show how elements in a two-dimensional array are manipulated in different conditions*

The similarities between manipulation of elements in a one-dimensional and two-dimensional array are evident from the code in Fig 3.23. The main difference is the addition of another parameter for the second dimension of an array but the rest of the layout of the command format is kept the same. Looking at where the array *myarray* in Fig 3.23 is declared it can be assumed that the array size is 3 by 5. Looking at the example showing how to change multiple element values in a specified 1D of the array above, when both *LOAD* commands are processed then the resultant changes to the array are best described in the table below.

1D and 2D dimensions	2D (0)	2D (1)	2D (2)	2D (3)	2D (4)
1D (0)	25	79	2	87	62
1D (1)	0	0	0	0	0
1D (2)	52	28	17	73	97

Fig. 3.24 *Table to help in visualising what a two-dimensional array is in iDev*

In two-dimensional arrays, it is possible to say that 1D is the rows in a table and 2D is the column. In iDev, multiple elements can only be changed at once (one command) by specifying the first dimension (row) of an array or all the elements in

the array. If the developer requires the elements in a specific second dimension (column) changing, then it has to be done one by one. Referring to the table, the example has changed all the elements in 1D(0) (1st row) by specifying the 1st dimension address in the *LOAD* command. The same is applied when all the elements in 1D(2)(3rd row) is changed. The elements in 1D(1)(2nd row) have the value 0 because the initial value of the array is set to 0. The current value of the elements stored in an array is kept until a new value is loaded into it. When a whole array is loaded, the elements are sent by hierarchal order:
myarray.0.0,myarray.0.1,myarray.0.2,…,myarray.1.0,myarray.1.1,myarray.1.2,…,myarray.2.0,myarray.2.1,myarray.2.2,…,myarray.2.4

Three-dimensional array explained
VAR command format for **three-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D,Size2D,Size3D);

The arrays with three dimensions have the similar command format to that of the two-dimensional arrays; the only difference is an extra parameter. The three-dimensional array is more difficult to visualise because of the extra dimension. A three-dimensional array can be seen as group of sets with three levels; the 1st dimension is the highest level set and the 3rd dimension is the lowest level set. In other words, the elements in the 3rd dimension are a subset of the elements in the 2nd dimension and the elements in the 2nd dimension are a subset of the elements in the 1st dimension. Due to the addition of another dimension, more situations would be possible when manipulating a three dimensional array. Up to 15 elements can be changed at a time in a three-dimensional array because of the 16-parameter limit in iDev. All the added possible situations that use the *LOAD* command are included underneath.

LOAD command format to change single element in three-dimensional array:
LOAD(Array name.1D.2D.3D, New element value/variable);
LOAD command format to transfer single element to variable:
LOAD(Variable name, Array name.1D.2D.3D);
LOAD command format to change multiple elements in specified 2nd dimension of a three-dimensional array:
LOAD(Array name.1D.2D,1st element value,2nd element value,3rd element value…);
LOAD command format to pass array elements in the specified 2nd dimension to serial interface/text variable or another array:
LOAD(Destination of array elements, Array source name.1D.2D);
LOAD command format to pass all array elements to text component:
TEXT(Text component, Array source name.1D.2D);
LOAD command format when all array elements come from serial interface (serial buffer):
LOAD(Array name, Serial interface source);
LOAD command format when 2nd dimension array elements come from serial interface (serial buffer):
LOAD(Array name.1D.2D,Serial interface source);

```
//FILENAME: TU480a.mnu

//Three-dimensional array is declared
VAR(myintvar,0,U8);
VAR(mytxtvar,"24",U8);

VAR(myarray,0,U8,2,3,3);
VAR(myarray2,0,U8,2,3,3);

//Inside a Page or Function
TEXT(mytext,"hello",mytxtst);

LOAD(myarray.0.2.0,25);        //Change single element value
LOAD(myarray.1.1.2,86);        //
LOAD(myarray.1.2.2,myintvar);  //

LOAD(myintvar,myarray.1.2.2);  //Transfer single element to variable
LOAD(myarray.1.2,52,28,17,);
//Change multiple element values in specified 2nd dimension of the array
LOAD(myarray.0.1,25,79,2,);
//Change multiple element values in specified 2nd dimension of the array
LOAD(RS2,myarray.1.2);
//Transfer array elements in specified 2nd dimension of the array to serial
interface
TEXT(mytext,myarray.0.1);
//Transfer array elements in specified 2nd dimension of the array to text
component
LOAD(mytxtvar,myarray.0.0);
//Transfer array elements in specified 2nd dimension of the array to text
variable
LOAD(myarray2,myarray.1.2);
//Transfer array elements in specified 2nd dimension of the array to another
array
LOAD(myarray,RS2);
//Transfer contents from serial interface to array (serial buffer)
LOAD(myarray.1.2,RS2);
//Transfer contents from serial interface to the specified 2nd dimension in
the array (serial buffer)
```

Fig. 3.25 *Example to show how elements in a two-dimensional array are manipulated in different conditions*

These added situations build upon the first and second dimensional array command formats, so it is important to remember that what can be applied in the first and second dimension arrays can also be applied to the three-dimensional array but not the other way around.

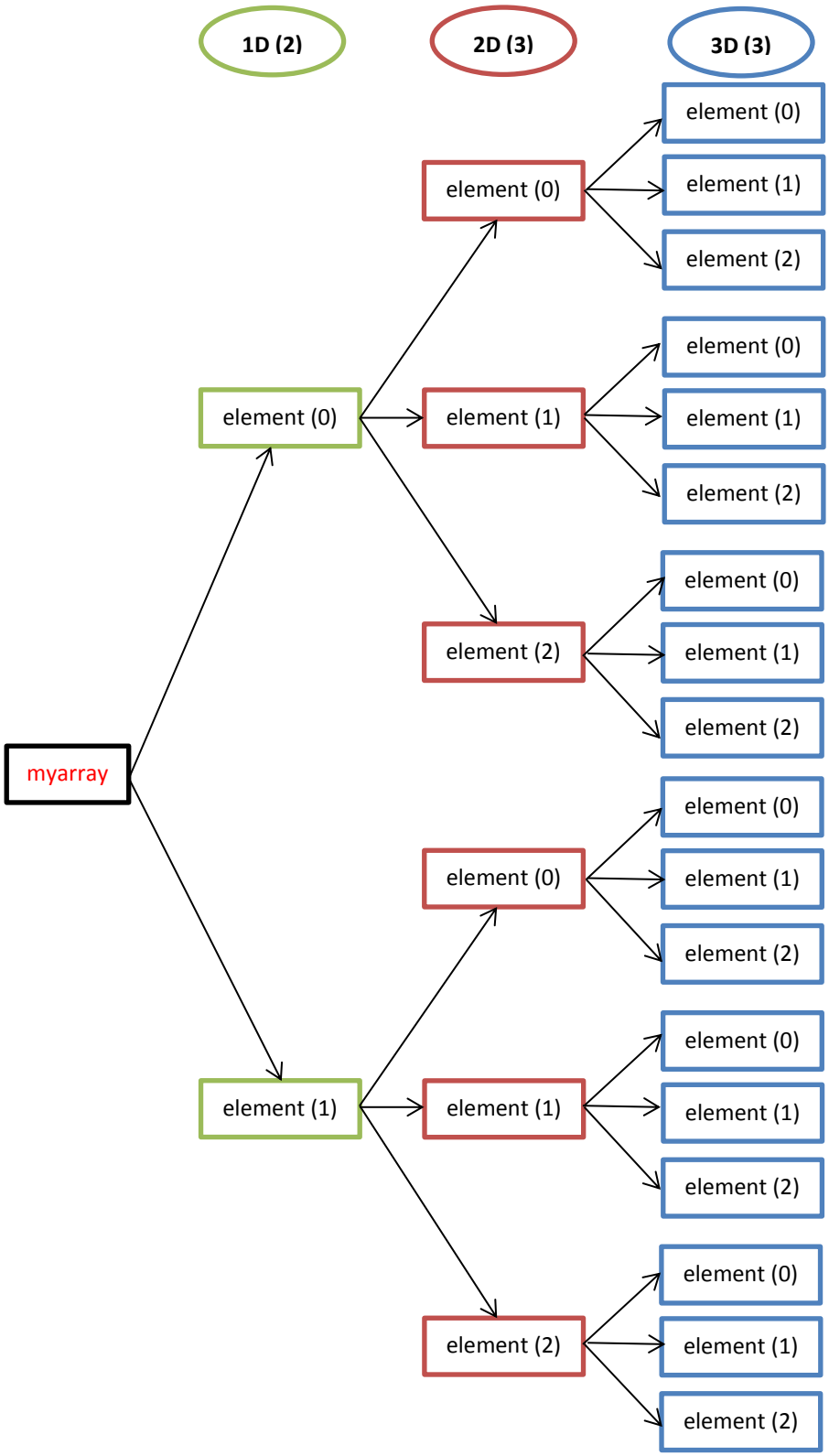


Fig. 3.26 *Diagram to show arrangement of elements in three-dimensional arrays*

The diagram above uses the array created in Fig 3.25 which is a 2 by 3 by 3 array with unsigned 8 bit integer and initial values of 0. It is easier to apply the analogy about arrays as group of sets of elements; the subset arrangement of the

elements is clearly indicated by the arrows in the diagram. The elements in a three-dimensional array are sent by hierarchal order as well:
myarrray.0.0.0,myarray.0.0.1,myarray.0.0.2,…,myarray.1.0.0,myarray.1.0.1,myarray.1.0.2,…,myarray.1.2.2
It is evident from the example usage in Fig 3.25 that multiple elements of up to two dimensions can be manipulated by stating the dimensions in the command. An example is created to show a typical application of three-dimensional arrays in iDev.

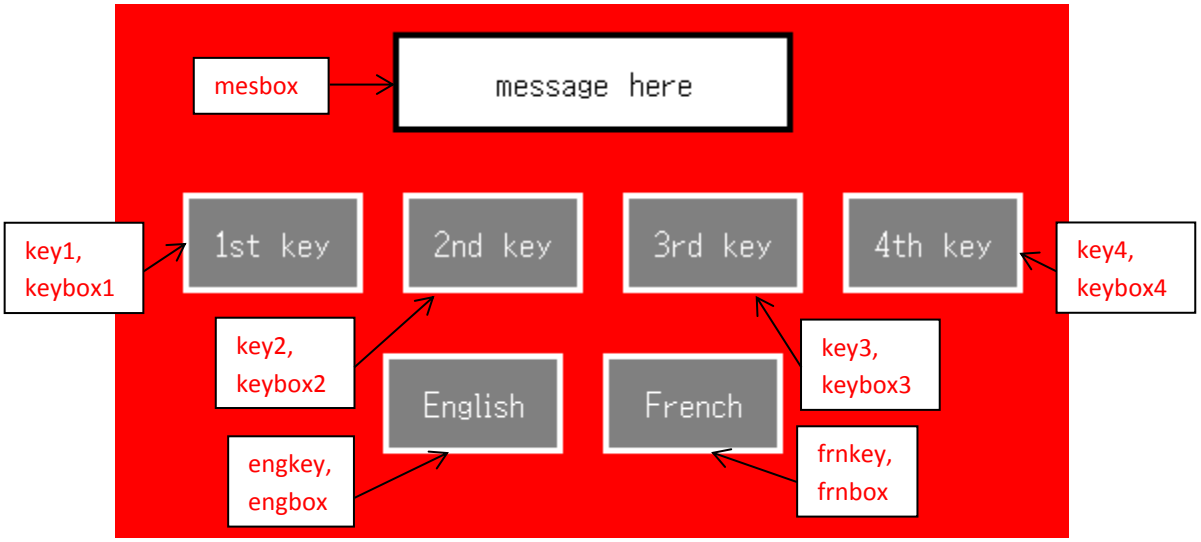


Fig. 3.27 *Screen shot showing the screen before any of the buttons are pressed*

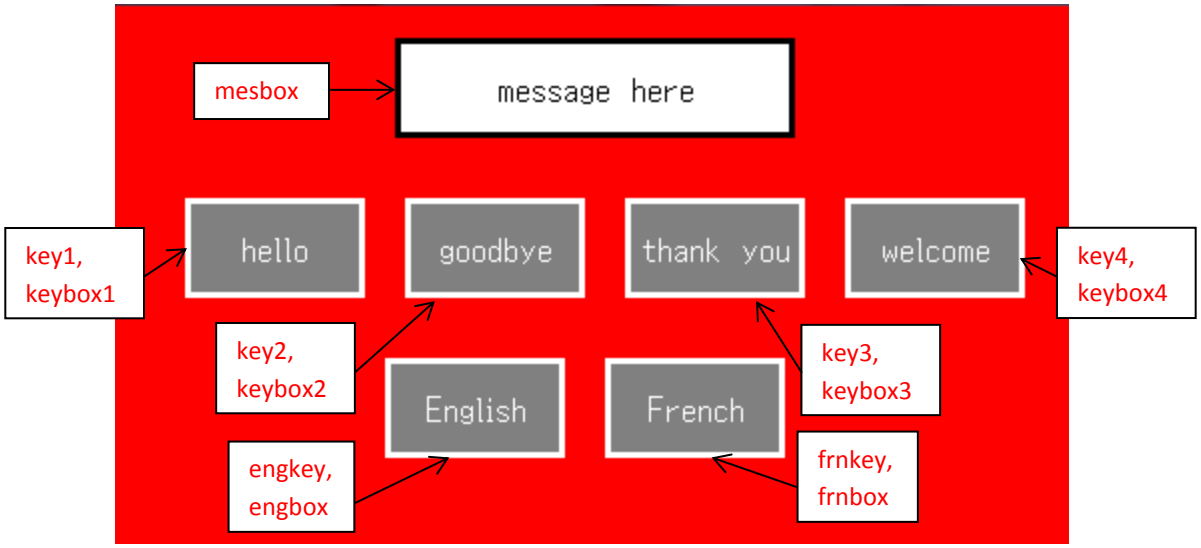


Fig. 3.28 *Screen shot showing the changes when the English button is pressed*

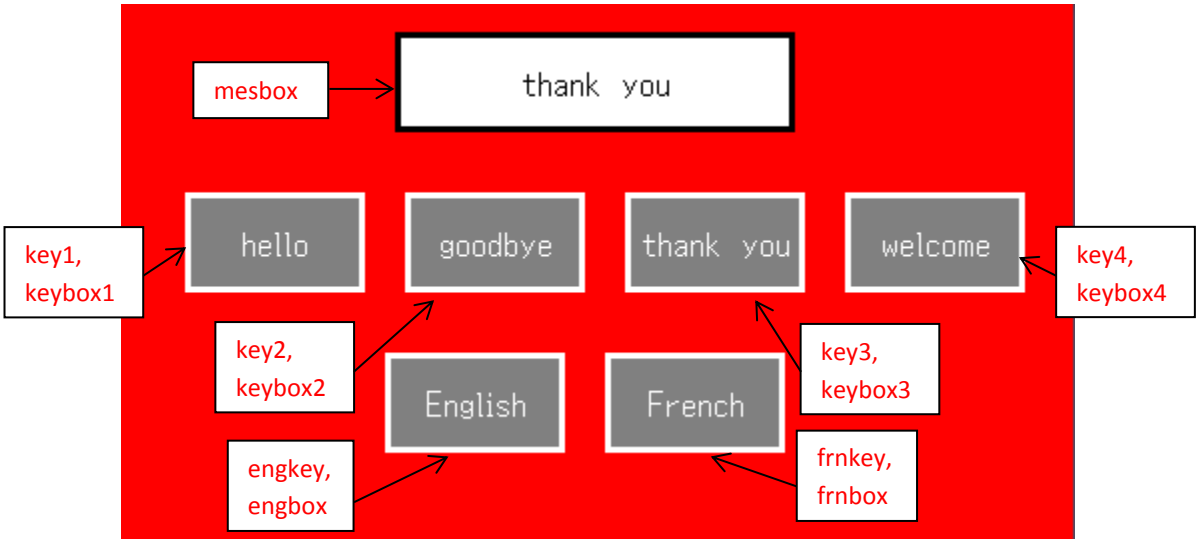


Fig. 3.29 *Screen shot showing the changes when the thank you button is pressed*

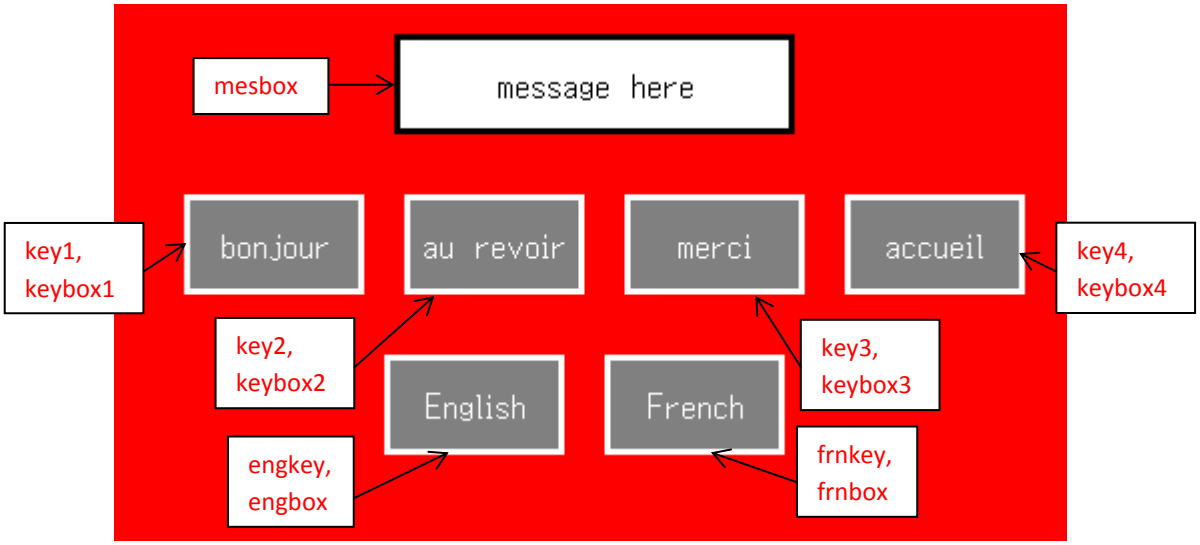


Fig. 3.30 *Screen shot showing the changes when the French button*

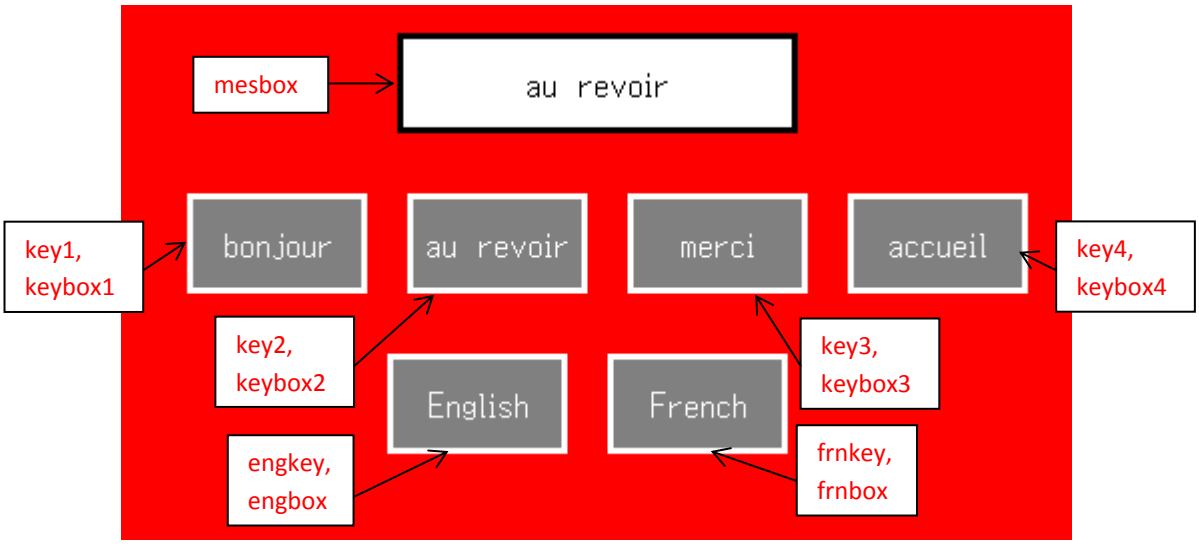


Fig. 3.31 *Screen shot to showing the changes when the au revoir button is pressed*

VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
VAR command format for **three-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D,Size2D,Size3D);
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2], X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
Inline Function command format :
In the function parameter of the iDev command
[Function contents]
LOAD command format to change multiple elements in specified 2nd dimension of a three-dimensional array:
LOAD(Array name.1D.2D,1st element value,2nd element value,3rd element value…);
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

VAR(lang,0,U8);                         //
VAR(myarray,0,U8,2,4,10);               //
VAR(heltxt,"hello\\00",TXT);            //
VAR(byetxt,"goodbye\\00",TXT);          //
VAR(tnxtxt,"thank you\\00",TXT);        //
VAR(weltxt,"welcome\\00",TXT);          //
VAR(bontxt,"bonjour\\00",TXT);          //
VAR(aurtxt,"au revoir\\00",TXT);        //
VAR(mertxt,"merci\\00",TXT);            //
VAR(acctxt,"accueil\\00",TXT);          //

STYLE(homepgst,Page)                    //
{
back = red;                             //
}
STYLE(Ascii16txst,Text)                 //
{
font = Ascii16;                         //
col = white;                            //
maxRows = 1;                            //
maxLen = 32;                            //
}
STYLE(Ascii16bltxst,Ascii16txst)        //
{
col = black;                            //
}
STYLE(boxdrwst,Draw)                    //
{
type = b;                               //
maxX = 100;                             //
maxY = 50;                              //
width = 3;                              //
back = grey;                            //
col = white;                            //
}
STYLE(mesboxdrwst,boxdrwst)             //
{
back= white;                            //
col = black;                            //
maxX = 200;                             //
}

PAGE(homepg,homepgst)                                   //
{
POSN(240,40);                                           //
DRAW(mesbox,200,50,mesboxdrwst);                        //
TEXT(mestxt,"message here",Ascii16bltxst);              //
POSN(80,120);                                           //
KEY(key1,keyfunc1,100,50,TOUCH);                        //
DRAW(keybox1,90,50,boxdrwst);                           //
TEXT(keytxt1,"1st key",Ascii16txst);                    //
POSN(+110,+0);                                          //
KEY(key2,keyfunc2,100,50,TOUCH);                        //
DRAW(keybox2,90,50,boxdrwst);                           //
TEXT(keytxt2,"2nd key",Ascii16txst);                    //
POSN(+110,+0);                                          //
KEY(key3,keyfunc3,100,50,TOUCH);                        //
DRAW(keybox3,90,50,boxdrwst);                           //
TEXT(keytxt3,"3rd key",Ascii16txst);                    //
POSN(+110,+0);                                          //
KEY(key4,keyfunc4,100,50,TOUCH);                        //
DRAW(keybox4,90,50,boxdrwst);                           //
TEXT(keytxt4,"4th key",Ascii16txst);                    //
```

```
POSN(180,+80);                          //
KEY(engkey,[LOAD(lang,0);RUN(engfunc);],100,50,TOUCH);
//
DRAW(engbox,90,50,boxdrwst);            //
TEXT(engtext,"English",Ascii16txst);    //
POSN(+110,+0);                          //
KEY(frnkey,[LOAD(lang,1);RUN(frnfunc);],100,50,TOUCH);
//
DRAW(frnbox,90,50,boxdrwst);            //
TEXT(frntext,"French",Ascii16txst);     //
}

FUNC(engfunc)                       //
{
LOAD(myarray.lang.0,%t%heltxt);     //
TEXT(keytxt1,%t%myarray.lang.0);    //
LOAD(myarray.lang.1,%t%byetxt);     //
TEXT(keytxt2,%t%myarray.lang.1);    //
LOAD(myarray.lang.2,%t%tnxtxt);     //
TEXT(keytxt3,%t%myarray.lang.2);    //
LOAD(myarray.lang.3,%t%weltxt);     //
TEXT(keytxt4,%t%myarray.lang.3);    //
}
FUNC(frnfunc)                       //
{
LOAD(myarray.lang.0,%t%bontxt);     //
TEXT(keytxt1,%t%myarray.lang.0);    //
LOAD(myarray.lang.1,%t%aurtxt);     //
TEXT(keytxt2,%t%myarray.lang.1);    //
LOAD(myarray.lang.2,%t%mertxt);     //
TEXT(keytxt3,%t%myarray.lang.2);    //
LOAD(myarray.lang.3,%t%acctxt);     //
TEXT(keytxt4,%t%myarray.lang.3);; //
}
FUNC(keyfunc1)                      //
{
TEXT(mestxt,%t%myarray.lang.0);;    //
}
FUNC(keyfunc2)                      //
{
TEXT(mestxt,%t%myarray.lang.1);;    //
}
FUNC(keyfunc3)                      //
{
TEXT(mestxt,%t%myarray.lang.2);;    //
}
FUNC(keyfunc4)                      //
{
TEXT(mestxt,%t%myarray.lang.3);;    //
}
SHOW(homepg);                       //
```

Fig. 3.32 *Example code to show how three-dimensional arrays are used in iDev*

The example above uses three-dimensional arrays to manipulate text data to be displayed on the screen. The usual application of this is in pages where translation to different languages is required. The code from Fig 3.32 uses a 2 by 4 by 10 array to store letters that is utilised to display appropriate text data. The 1st dimension is applied to control the language between English and French. Hence, the 1st dimension is set to 2. The 2nd dimension handles the words and keys (buttons) on the screen, there are four words and buttons that are created so the 2nd dimension is set to 4. Lastly, the 3rd dimension is used to store the characters of each word. The maximum amount of characters that is used is 9 but since a character

terminator is required the resulting length of the 3rd dimension is set to 10. A character terminator is important in iDev as it notifies TFT module's programmer (interpreter) that it is the end of the string in an array. These criteria are used to determine the length of each dimension in an array. Hence, the size of the array is 2 by 4 by 10. There example code creates 6 key components (buttons) and a message box. Four of the buttons are used to display different messages in English and French and the other two buttons are created to translate the language between the two. The message box's contents change depending on which key is pressed and language. When the key component *engkey* is pressed then the value of variable *lang* is changed to 0, so the set of words that are in English in the array is accessed. The function *engfunc* is executed as well which assigns the suitable text data to the key components on the screen. From the example code, it is observable that '%t%' is present when the array source and element value parameters are stated. This syntax(%t%) is used so that text data can be stored in an array. Since arrays can only store single value per slot, each letter for a certain word is stored per slot i.e. if the word 'hello' is going to be stored in the array then the first slot *myarray.0.0.0* stores the letter 'h', second slot *myarray.0.0.1* the letter 'e', third slot *myarray.0.0.2* the letter 'l' and so on... Unfortunately arrays in iDev do not support text data yet so this method to store text data is a temporary fix. The same method is used for the French language which occurs when the key component *frnkey* is pressed. The diagram below describes how elements and words are accessed in the array *myarray.*

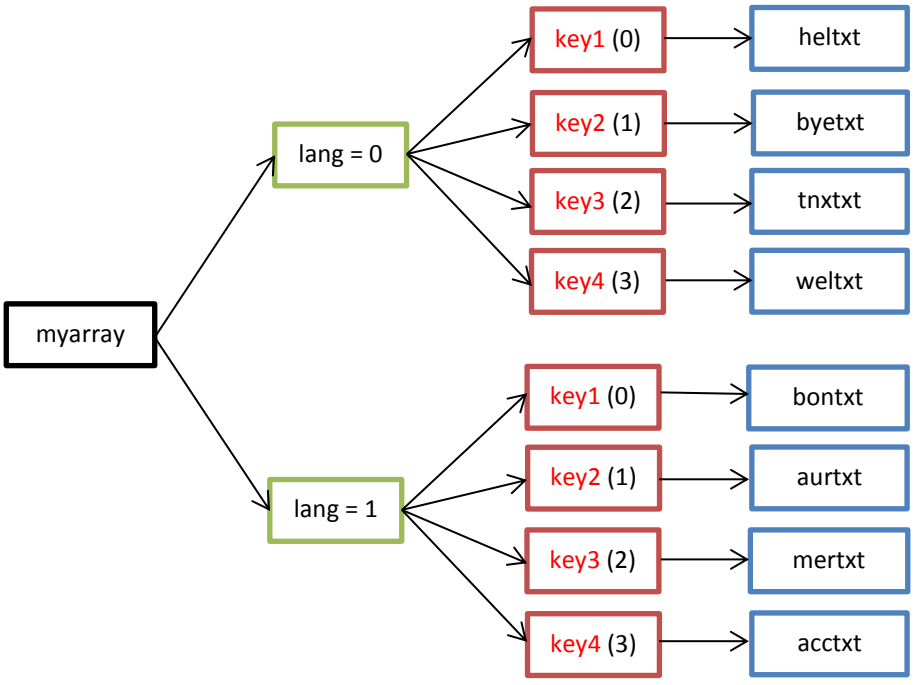


Fig. 3.33 *Diagram to demonstrate how words are chosen from a three-dimensional array in the example code in Fig 3.32*

Note that *heltxt*,*byetxt, tnxtxt,weltxt, bontxt, aurtxt, mertxt* and *acctxt* are all text variables which are assigned to the array slots to hold each letter. Looking at the diagram above, the path to access the text data *acctxt* will be used as an example. The value of the first dimension is 1, then the value for the second dimension is 3

The resultant 'address' of the path is *myarray.1.3* which is assigned to contain the appropriate text data. So when the language is set to French and the button 'accueil' is pressed then the method described is carried out to display the correct text data. The same technique is applied to access and manipulate the other elements in the array and hence display other messages/text. Screen shots are taken to demonstrate what changes would be seen on the module screen.

Four-dimensional arrays explained

VAR command format for **four-dimensional arrays**:

VAR(Array name, Array initial values, Data type,Size1D,Size2D,Size3D,Size4D);

Multi-dimensional arrays have similar command formats in iDev and a clear structure from one-dimensional, two-dimensional and three-dimensional arrays are apparent. In a four-dimensional array, the elements in the fourth dimension are subset of the elements in the third dimension; the elements in the third dimension are subset elements in the second dimension and lastly the elements in the second dimension is a subset of the elements in the first dimension. Up to 15 elements can be changed at a time` in a four-dimensional array because of the 16 parameter limit in iDev. More situations are now possible because of the added dimension to the array; these situations are similar to that of the previous examples.

LOAD command format to change single element in three-dimensional array:

LOAD(Array name.1D.2D.3D.4D, New element value/variable);

LOAD command format to transfer single element to variable:

LOAD(Variable name, Array name.1D.2D.3D.4D);

LOAD command format to change multiple elements in specified 3rd dimension of a three-dimensional array:

LOAD(Array name.1D.2D.3D,1st element value,2nd element value…);

LOAD command format to pass array elements in the specified 3rd dimension to serial interface/text variable or another array:

LOAD(Destination of array elements, Array source name.1D.2D.3D);

LOAD command format to pass array elements in the specified 3rd dimension to text component:

TEXT(Text component, Array source name.1D.2D.3D);

LOAD command format when all array elements come from serial interface (serial buffer):

LOAD(Array name, Serial interface source);

LOAD command format when 3rd dimension array elements come from serial interface (serial buffer):

LOAD(Array name.1D.2D.3D,Serial interface source);

```
//FILENAME: TU480a.mnu

//Two-dimensional array is declared
VAR(myintvar,0,U8);                 //
VAR(mytxtvar,"24",TXT);             //

VAR(myarray,0,U8,2,2,2,3);          //
VAR(myarray2,0,U8,2,2,2,3);         //

//Inside a Page or Function
TEXT(mytext,"hello",mytxtst);       //

LOAD(myarray.0.1.0.2,25);           //
LOAD(myarray.1.1.1.1,86);           //
LOAD(myarray.1.0.1.0,myintvar);     //

LOAD(myintvar,myarray.1.1.0.2);  //Transfer single element to variable
LOAD(myarray.1.1.0,52,28,17);
// Change multiple element values in specified 3rd dimension of the array
LOAD(myarray.0.0.1,25,79,2);
//Change multiple element values in specified 3rd dimension of the array
LOAD(RS2,myarray.1.0.1);
//Transfer array elements in specified 3rd dimension of the array to serial
interface
TEXT(mytext,myarray.0.1.0);
//Transfer array elements in specified 3rd dimension of the array to text
component
LOAD(mytxtvar,myarray.0.1.1);
//Transfer array elements in specified 3rd dimension of the array to text
variable
LOAD(myarray2,myarray.0.0.1);
//Transfer array elements in specified 3rd dimension of the array to another
array
LOAD(myarray,RS2);
//Transfer contents from serial interface to array (serial buffer)
LOAD(myarray.1.1.0,RS2);
//Transfer contents from serial interface to the specified 3rd dimension in
the array (serial buffer)
```

Fig. 3.34 *Example to show how elements in a four-dimensional array are manipulated in different conditions*

It is obvious that the command format and structure of the four-dimensional array is similar to the other multi-dimensional array. Elements in a four dimensional array are sent in a hierarchal order (which is the same with the others): *myarray0.0.0.0,myarray.0.0.0.1,myarray.0.0.0.2,…,myarray.0.0.1.0,myarray.0.0.1.1, myarray.0.0.1.2,…,myarray.1.1.1.2.* For a better understanding on the hierarchal order of the elements in a four-dimensional array *myarray* of size 2 by 2 by 2 by 3 is created. Two diagrams representing a four-dimensional array is created.

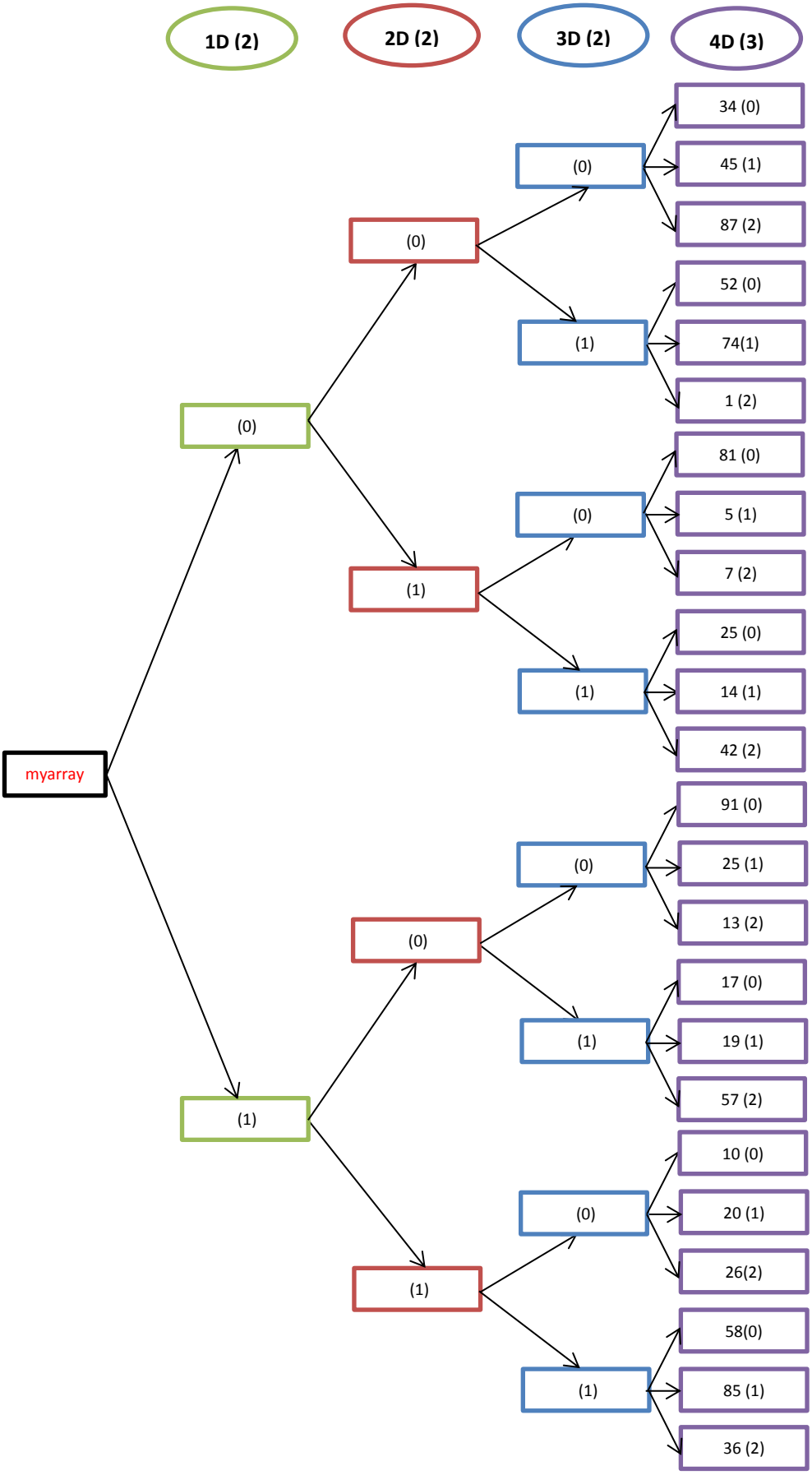
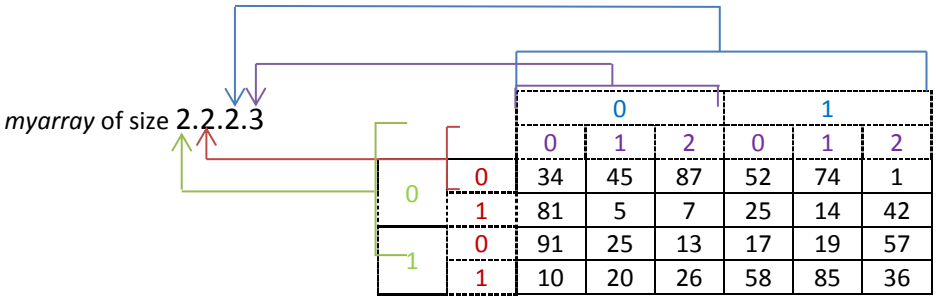


Fig. 3.35 *Diagram showing subset representation of four-dimensional arrays in iDev*



			0			1		
			0	1	2	0	1	2
0	0	0	34	45	87	52	74	1
		1	81	5	7	25	14	42
1	0	0	91	25	13	17	19	57
		1	10	20	26	58	85	36

Fig. 3.36 *Diagram showing table representation of four-dimensional arrays in iDev*

For better guidance, the diagrams and examples are colour coordinated so that a beginner developer would be able to visualise four-dimensional arrays easier. Examples are created below to demonstrate how to access single elements in the array *myarray* in diagrams in Fig 3.35 and 3.36.

myarray.0.0.0.0 = 34
myarray.1.0.1.0 = 17
myarray.0.0.1.2 = 1
myarray.0.1.0.1 = 5
myarray.0.1.0.0 = 81
myarray.1.1.0.2 = 26

Four-dimensional arrays are rarely used in iDev projects but it adds flexibility and capability in some iDev projects that require the use of four dimensions. As practice, the example code in Fig 3.27 can be modified so another dimension is added to change another visual feature on the page such as changing the background colour or even changing the whole page.

3.2. FORMATTING DATA

Data in any programming languages comes in different format such as decimal, hex or float. These data formats have to be correct so that it can be interpreted appropriately. A typical usage would be in data transmission, some external modules send data in a certain format so when the buffer is set in iDev, and it has to be the correct format for the received data to be properly interpreted. On the other hand, when sending commands to an external module that is connected through the serial interface then the data format should be appropriate.

VAR command format to apply different data format to value stored:
VAR(Variable name,%Data format%Starting Value, Variable Style);
TEXT command format to apply different data format to text component:
TEXT(Text Component,%Data format%Variable Source, Text Style);
LOAD command format to apply different data format to a destination (serial interface/variable):
LOAD(Destination,%Data format%Variable Source);

Data Format	Definition
s	decimal integer
h	lower case hex
H	upper case hex
h1 to h8	lower case hex with a field width where padding used are spaces
H1 to H8	upper case hex with a field width where padding used are spaces
h01 to h08	lower case hex with a field width where padding used are zeros
H01 to H08	upper case hex with a field width where padding used are zeros
f	decimal floating point
f1 to f8	decimal floating point with specified number of decimal places
r	raw data

Fig. 3.37 *Table to show and define different types of iDev data format*

This part may be too complicated for beginners in programming so this part is only for advanced developers that have used the C programming language, so for beginners just use the basic iDev data format for now. The printf formatting in C can also be applied in iDev. This is achieved by placing a '*' before the printf command in the data format parameter enclosed in '%'. In iDev, all the Printf Data Format is treated as if it is in C, so the same results would occur. The command format is then changed to:

VAR command format to apply printf data format to value stored:
VAR(Variable name,%*Printf Data Format %Starting Value, Variable Style);
TEXT command format to apply printf data format to text component:
TEXT(Text Component,%*Printf Data Format %Variable Source, Text Style);
LOAD command format to apply printf data format to a destination (serial interface/variable):
LOAD(Destination,%*Printf Data Format%Variable Source);

The Printf format in C programming language has its own parameters that are needed to specify what type of data is required for correct data interpretation. The Printf data format manipulation gives the developer more options on how they want the data to be used. Some of the iDev Data formats will have the same result with different combinations of the Printf Data format.

Printf Data format:
FlagsWidth.PrecisionLengthSpecifier

Flags	**Definition**
-	Justify to the left within the given field width (default is right-justify)
+	shows data with a plus or minus sign (default only negative numbers are placed with minus sign before it)
(space)	a blank space is inserted before the data/value
#	For o, x or X specifiers, the value is preceded with 0,0x or 0X respectively for values different than zero
	For e, E and f specifiers, the written output is forced to contain a decimal point even if no digits would follow (default, if no digits follow then no decimal point is written)
	For g or G specifiers, the same happens with e or E but trailing zeros are not removed
0	left-pads the number with zeros instead of space, where padding is specified by width

Fig. 3.38 *Definition and Expected values for the flags parameter in printf data format*

Width	Definition
(number)	sets the minimum number of characters to be shown, if data to be displayed is shorter than this number then the result is padded with blank spaces and the value is not truncated even if the result is larger

Fig. 3.39 *Definition and Expected values for the width parameter in printf data format*

.Precision	Definition
.(number)	For d, u, i, o, x, X specifiers, the value is set for the minimum number of digits to be written ,if data to be displayed is shorter than this number then the result is padded with leading zeros and the value is not truncated even if the result is larger
	For e, E or f specifiers, this value is the number of digits to be displayed after the decimal point.
	For g or G specifiers, the maximum number of significant digits to be displayed
	For s, the maximum number of characters to be displayed (default, all characters are displayed until the character terminator is encountered)
	For c, it has no effect

Fig. 3.40 *Definition and Expected values for the precision parameter in printf data format*

Length	Definition
h	For d, u, i, o, x, X specifiers, the command is interpreted as a short integer or unsigned short integer
l	For d, u, i, o, x, X specifiers, the command is interpreted as a long integer or unsigned long integer
	For c or s specifiers, the command is interpreted as a wide character or wide character string
L	For e, E, f, g or G specifiers, the command is interpreted as a long double

Fig. 3.41 *Definition and Expected values for the length parameter in printf data format*

Specifier	Definition
c	character
d or i	signed decimal integer
u	unsigned decimal integer
e	lower case scientific notation (exponential)
E	upper case scientific notation (exponential)
f	decimal floating point
g	use the shorter of exponential or decimal floating point in lower case
G	use the shorter of exponential or decimal floating point in upper case
o	unsigned octal
s	string of characters
x	lower case unsigned hexadecimal integer
X	upper case unsigned hexadecimal integer

Fig. 3.42 *Definition and Expected values for the specifier parameter in printf data format*

It is crucial to be familiar with different types of data in iDev because it can be guaranteed that not all data being sent and received in iDev projects using external modules are in the same data format. An example below demonstrates how to apply different types of data format to one variable. This one variable is constant and does not change unlike other data being received but this example would suffice.

Fig 3.43 *Screen shot to show what would be displayed when the example code in Fig 3.44 is uploaded*

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format to apply different data format to value stored:
VAR(Variable name,%Data format%Starting Value, Variable Style);
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
TEXT command format to apply different data format to text component:
TEXT(Text Component,%Data format%Variable Source, Text Style);
TEXT command format to apply printf data format to text component:
TEXT(Text Component,%*Printf Data Format %Variable Source, Text Style);
}
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

STYLE(homepgst,Page)                        //create a style for homepg
{
back = black;                               //
}
STYLE(Ascii16txtst,Text)                    //
{
font = Ascii16;                             //
col = white;                                //
maxRows = 1;                                //
maxLen = 12;                                //
curRel = LC;                                //
}
STYLE(flt7,Data)                            //create style for float variable
{
type = float;                               //
decimal = 7;                                //
}

VAR(myintvar,2031,U16);                     //create U16 variable
VAR(myfltvar,3.1415927,flt7);               //create 7 decimal point float variable
VAR(mytxtvar,%s%myintvar,TXT);              //store "2031"
VAR(myrawvar,%r%51,U8);                     //store "51"

PAGE(homepg,homepgst)                       //
{
POSN(20,10);                                //
TEXT(mytxt1,mytxtvar,Ascii16txtst);         //shows "7ef"
POSN(+0,+18);                               //
TEXT(mytxt2,%h%myintvar,Ascii16txtst);      //shows "7ef"
POSN(+0,+18);                               //
TEXT(mytxt3,%H%myintvar,Ascii16txtst);      //shows "7EF"
POSN(+0,+18);                               //
TEXT(mytxt4,%h2%myintvar,Ascii16txtst);     //shows "7ef"
POSN(+0,+18);                               //
TEXT(mytxt5,%H8%myintvar,Ascii16txtst);     //shows "     7EF"
POSN(+0,+18);                               //
TEXT(mytxt6,%h02%myintvar,Ascii16txtst);    //shows "7EF"
POSN(+0,+18);                               //
TEXT(mytxt7,%H08%myintvar,Ascii16txtst);    //shows "000007ef"
POSN(+0,+18);                               //
TEXT(mytxt8,%f%myfltvar,Ascii16txtst);      //shows "3.141593"
POSN(+0,+18);                               //
TEXT(mytxt9,%f4%myfltvar,Ascii16txtst);     //shows "3.1416"
POSN(+0,+18);                               //
TEXT(mytxt10,%f8%myfltvar,Ascii16txtst);    //shows "3.14159270"
POSN(+0,+18);                               //
TEXT(mytxt11,%*e%myfltvar,Ascii16txtst);    //shows "3.141593e+00"
POSN(+0,+18);                               //
TEXT(mytxt12,%*08X%myintvar,Ascii16txtst);  //shows "000007EF"
POSN(+0,+18);                               //
TEXT(mytxt13,%*+d%myintvar,Ascii16txtst);   //shows "+2031"
POSN(+0,+18);                               //
TEXT(mytxt14,%r%myrawvar,Ascii16txtst);     //shows "3"
}
SHOW(homepg);                               //
```

Fig. 3.44 *Example code demonstrating how different types of format can be used to manipulate how data are interpreted in iDev*

The example code in Fig 3.44 only uses the *TEXT* command to interpret data in different formats but the *LOAD* command can also be used. The same results should be expected when the *LOAD* command is used to set different data formats in iDev.

3.3. MOVING AND UPDATING DATA – LOAD

The *LOAD* command in iDev is a very versatile command. It can be used to copy contents of a page to a previously defined page. This command is also used in Chapter 2.5.4, where it is used to update parameters in a previously defined style. Another prominent use of the *LOAD* command is changing the values stored in variables as described in Chapter 3.1.3 of this guide. The *LOAD* command is also used in controlling pointers in iDev as described in Chapter 3.1.5. Array manipulation also uses the *LOAD* command as seen in Chapter 3.1.6. Multiple values can be combined or concatenated by the use of *LOAD* command as well. Variables, buffers and text can be combined and the result is copied to a variable or buffer. This allows text and variables that are joined together to be sent through an interface. Below is a summary of all the command formats that uses the *LOAD* command. The *LOAD* command formats stated in the previous parts of the guide is also found below, for the arrays, only the one-dimensional array that uses *LOAD* command is included below, if *LOAD* command format for multi-dimensional array is required then refer to Chapter 3.1.6.

LOAD command formats that have been stated in other parts of the guide:

LOAD command format to update styles:
LOAD(Style name.Parameter,New Parameter Value);
LOAD command format for using pointers:
LOAD(Pointer variable name>"Shared destination value", Destination Identifier);
LOAD command format to apply different data format to a destination (serial interface/variable):
LOAD(Destination,%Data format%Variable Source);
LOAD command format to apply printf data format to a destination (serial interface/variable):
LOAD(Destination,%*Printf Data Format%Variable Source);
LOAD command format to change single element in one-dimensional array:
LOAD(Array name.1D, New element value/variable);
LOAD command format to transfer single element to variable:
LOAD(Variable name, Array name.1D);
LOAD command format to change all elements with a single value in one-dimensional array:
LOAD(Array name, Single value);
LOAD command format to change multiple elements in one-dimensional array:
LOAD(Array name,1st element value,2nd element value, 3rd element value…);
LOAD command format to pass array elements to serial interface/text variable or another array:
LOAD(Destination of array elements, Array source name);
LOAD command format to pass all array elements to text component:
TEXT(Text component, Array source name);
LOAD command format when array elements come from serial interface (serial buffer):
LOAD(Array name, Serial interface source);

New LOAD command format for general application:

LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value/Variable);
LOAD command format to combine/concatenate values or contents of variables and pass the result to a variable:
LOAD(Text Variable, "New Text"/Text Variable1, " New Text"/ Text Variable2…);

LOAD command format to send contents of a variable or a value through an interface:
LOAD(Interface, New Value/Variable);
LOAD command format to send combined/concatenated contents of a text variable or text data through an interface:
LOAD(Interface, "New Text"/Text Variable1, "New Text"/ Text Variable2,…);
LOAD command format to use a previously defined page as a template for a new page that is created, page refresh is needed to make changes visible:
LOAD(Destination Page, Previously Defined Page);;
LOAD command format to change specific setup parameters:
LOAD(Setup Name.Parameter, New Parameter Value);
LOAD command format to transfer files from SDHC to on-board NAND flash (used with FPROG – Chapter 8.3):
LOAD(NAND, "SDHC/Filename");

The *LOAD* command formats from the previous chapters already have examples on how they are applied in the same chapter where they are stated, so the example in this chapter would only show how to use the new *LOAD* command formats in iDev.

```
//FILENAME: TU480a.mnu

//Variables to store values are declared
VAR(myintvar,2031,U16);         //create U16 variable
VAR(myintdesvar,15,U8);         //
VAR(mytxtvar,"text var",TXT);   //

//Inside a Page or Function
TEXT(mytext,"hello",mytxtst);   //

LOAD(myintdesvar,28);     //Replace the contents of variable with a new value
LOAD(mytxtvar,"6",myintvar,"here");
//Combine/concatenate variable with text data and store it in the same variable
LOAD(AS1,myintdesvar);    //Transfer the contents of a variable to a serial interface
LOAD(RS2,myintdesvar,"6",myintvar);
//Combine/concatenate variable with text data and transfer it to a serial interface
LOAD(homepg,templatepg);;
//Transfer the contents of a page to another previously defined page
LOAD(RS2.baud,myintvar); //Change the setup parameter to a value from a variable
LOAD(AS1.data,6);         //Change the setup parameter to a new value
LOAD(NAND,"SDHC/Functions.mnu");
//Transfer project files from SDHC to NAND flash - used with FPROG command
```

Fig. 3.45 *Example code demonstrating how LOAD command is applied in iDev*

When combining data and using the *LOAD* command to transfer it to another variable or serial interface, it is important to know how data is merged. From the example above, the line *LOAD(mytxtvar, "6",myintvar, "here");* would result in storing the value “62031here” to the text variable *mytxtvar.* The same principle is applied when variables are combined with text data and transferred to a serial interface e.g. from the example above the resultant data is “1562031”. The setup parameters are changed using the *LOAD* command and the dot operator. The size and watchdog parameters should not be changed as this can cause errors (not implemented)… The setup parameter change can be applied to the following: RS2, RS4, AS1, AS2, DBG, I2C, SPI, PWM, ADC, KEYIO and SYSTEM. These setups are mostly used to manipulate settings for the interfaces in iDev. Interfaces are explained in Chapter 4. Lastly, there is an option in iDev whereby files in an iDev project can be moved from SDHC or serial interface to the on-board NAND flash. If the source of the file is SDHC then the command

parameter is straightforward as seen from the example but if the source is from a serial interface then there are other parameters to be considered. Transferring files from SDHC or serial interface to NAND flash requires the use of the *FPROG* command; this is explained properly in Chapter 8.1 of this guide.

3.4. COMPARING DATA OR CREATING CONDITIONS – IF

In iDev, *IF* statements give the developer the capability to control the flow of his/her iDev project. This allows the developer to manipulate certain parts of his/her code to be executed based on the user's input. If the condition stated in the *IF* statement is true, then a function is carried out and if false then another function can be called or no action can be implemented. The functions that are called are obviously specified by the developer. In programming, the term true and false has a different meaning compared to the literal meaning of it in the English language. A statement in programming is deemed true when the result of the evaluation is a nonzero number. On the contrary, a false statement results to zero. Note that the functions in *IF* statements use the same command format as the *FUNC* command format in iDev, so inline commands can also be applied. In iDev, numeric values and text strings can be compared in if statements. When comparing data, it is important to remember that both values to be compared must be of the same type i.e. both must be text string/numeric values or variables that contain either one. So you cannot compare text string to a numeric value, which would seem useless anyway. When comparing floating point number (float variables with maximum of 17 decimal places) the lowest bit is masked before data is compared. If a floating point variable or value is compared to an integer variable or value then the integer is treated as if it's a floating point by padding zeros after the decimal point (not implemented).

IF command format to create if statements with just one action, note that a *Operand1* can be a variable and *Operand2* can be a literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function);
IF command format to create if statements with an else action, note that a *Operand1* can be a variable and *Operand2* can be a variable, literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function1:Function2);

Numerical Value/Variable and Boolean Logical Operators for IF statements			
Operator	**Definition**	**Result: TRUE then execute Function1**	**Result: FALSE then execute Function2**
= or ==	equal to	5 = 5	5 = 7
<> or !=	not equal to	5 != 2	5 != 5
=-	equal to the negative of	5 =- 5	5 =- -5
<	less than	5 < 26	5 < 1
>	greater than	5 > 3	5 > 21
<=	less than or equal to	5 <= 5	5 <= 4
>=	greater than or equal to	5 >= 2	5 >= 43
+	sum not equal to zero	5 + 4	5 + -5
-	difference not equal to zero	5 - 3	5 - +5
*	multiplication not equal to zero	5 * 29	5 * 0
/	division not equal to zero	5 / 2	0 / 5

%	modulus (remainder of division) not equal to zero	5 % 4	5 % 5
&&	Boolean Logical AND (double ampersand)	5 = 5 && 5 > 1	5 = 5 && 5 =! 5 5 = 7 && 5 <25
||	Boolean Logical OR (double pipe)	5 = 2 || 5 < 16 5 > 2 || 5 = -5	5 < 1 || 5 >= 21

Fig 3.46 *Table describing the different operators including Boolean Logical operators used in comparing numerical values/variables in iDev*

All the operators but the Boolean Logical ones from Fig 3.46 are self-explanatory because they are based on simple mathematical operations. Boolean Logical operators are used in if statements in most programming languages. Boolean Logical operators have two main types, namely Boolean Logical *AND* and Boolean Logical *OR.* It is not necessary to learn Boolean algebra to learn the purpose of these two operators basically Boolean Logical *AND* is used to determine whether both expressions in the operands are true. Looking at the Boolean Logical *AND* example in the table, the condition is deemed true because the first expression *5 = 5* AND the second expression *5 > 1* are true. That is, for the condition to be true then both expressions have to be true otherwise it's false. The Boolean Logical *OR* however is used to determine if either one of the expressions in the operands are true. From the example, the condition *5 = 2 || 5 < 16* and *5 > 2 || 5 = -5* are both deemed true because either one of the expressions in the operands are true. If both expressions in the operands are false then the whole condition is deemed false. Boolean Logical *AND* operator is useful when two expressions in the operands are required to be both true at the same time whereas the Boolean Logical *OR* operator is useful when only one expression in the operand is required to be true.

Bitwise Operators for IF statements			
Operator	**Definition**	**Operation**	**Result**
&	Bitwise AND (single ampersand)	5 (00000101) & 84 (01010100)	4 (00000100) **TRUE then execute Function1**
		5 (00000101) & 10 (00001010)	0 (00000000) **FALSE then execute Function2**
|	Bitwise OR (single pipe)	5 (00000101) | 6 (00000110)	7(00000111) **TRUE then execute Function1**
		0 (00000000) | 0 (00000000)	0(00000000) **FALSE then execute Function2**
^	Bitwise exclusive OR	5 (00000101) ^ 210 (11010010)	215(11010111) **TRUE then execute Function1**
		5 (00000101) ^ 5 (00000101)	0(00000000) **FALSE then execute Function2**

Fig. 3.47 *Table describing the Bitwise operators used in comparing numerical values/variables in iDev*

For beginner developers that are unfamiliar with bitwise operators, then this is a summary on what it is used for. The Bitwise operators are not as straight forward as the Boolean Logical operators as it involves calculation of binary values. The numbering system that is used everywhere is decimal which uses the symbols 0 through 9, binary values however only uses the symbols 0 and 1. Reading in binary is usually started from the right, and then continued to left until the last symbol 1.

Decimal Numbering					
$100(10^2)$	$10(10^1)$	$1(10^0)$	**Calculation**	**Decimal**	**Binary**
0	2	5	0(100) + 2(10) + 5(1) = 25	25	0001101
1	0	9	1(100) + 0(10) + 9(1) = 109	109	1101101
1	1	7	1(100) + 1(10) + 7(1) = 117	117	1110101
0	1	3	0(100) + 1(10) + 3(1) = 13	13	0001101

Fig. 3.48 *Table demonstrating how numbers are 'read' in decimal numbering system*

The most commonly used numbering system is decimal; it is what people use today to quantify objects. The table above describes how the decimal numbering system is calculated; each place value (100, 10, 1) can hold any of the symbols from 0 to 9. Place value in decimal is in multiples/powers of 10, each subsequent place value starting from the right is multiplied by 10. The place value is then multiplied by the corresponding symbol (0 to 9) to get the final decimal value as evident from the examples in the table above.

Binary numbering									
$64(2^6)$	$32(2^5)$	$16(2^4)$	$8(2^3)$	$4(2^2)$	$2(2^1)$	$1(2^0)$	**Calculation**	**Binary**	**Decimal**
0	0	1	1	0	0	1	0(64) + 0(32) + 1(16) + 1(8) + 0(4) + 0(2)+ 1(1) = 25	0001101	25
1	1	0	1	1	0	1	1(64) + 1(32) + 0(16) + 1(8) + 1(4) + 0(2) + 1(1) = 109	1101101	109
1	1	1	0	1	0	1	1(64) + 1(32) + 1(16) + 0(8) + 1(4) + 0(2) + 1(1) = 117	1110101	117
0	0	0	1	1	0	1	0(64) + 0(32) + 0(16) + 1(8) + 1(4) + 0(2) + 1(1) = 13	0001101	13

Fig. 3.49 *Table demonstrating how numbers are 'read' in binary numbering system*

The place value (64, 32, 16, 8, 4, 2, 1) in binary is in multiples/powers of 2 so each subsequent place value starting from the right is multiplied by 2 as opposed to 10 in decimal. All the place values are multiplied by the corresponding symbol (0 or 1) to get the final binary value. Now that binary numbers are explained then bitwise operators can be applied to them. As seen in Fig 3.47 there are three bitwise operators supported in iDev. The bitwise *AND, OR,* and exclusive *OR* operator is applied to binary numbers by 'adding' the symbols. Adding in binary is not the same as addition in mathematics. The results are based on the rules of the operators. There are 4 possible combinations when operations are performed in binary since only two symbols are used. A truth table (table containing all possible results) is produced to demonstrate the difference between the bitwise operators. Then an example taken from Fig 3.47 is broken down in a table to exhibit how the results are calculated. It is better to refer to the truth tables first before looking at the bitwise example calculations for all the operators.

All possible combinations for bitwise *AND* (&)				
Operand1 AND (&) Operand2	Binary			
Operand1	1	1	0	0
Operator	&	&	&	&
Operand2	1	0	1	0
Result	1	0	0	0

Fig. 3.50 *Truth table to show all possible results from the bitwise AND operator*

Bitwise *AND* example calculation									
Operand1 AND (&) Operand2	Binary								Decimal
Operand1	0	0	0	0	0	1	0	1	5
Operator	&	&	&	&	&	&	&	&	&
Operand2	0	1	0	1	0	1	0	0	84
Result	0	0	0	0	0	1	0	0	4

Fig. 3.51 *Table to show how resulting values are calculated using the bitwise AND operator*

Looking at the examples above, it is evident that the only way to gets resulting value of 1 using the bitwise *AND* operator is if operands are 1.

All possible combinations for bitwise *OR* (|)				
Operand1 OR (|) Operand2	Binary			
Operand1	1	1	0	0
Operator	|	|	|	|
Operand2	1	0	1	0
Result	1	1	1	0

Fig. 3.52 *Truth table to show all possible results from the bitwise OR operator*

Bitwise *OR* example calculation									
Operand1 OR (|) Operand2	Binary								Decimal
Operand1	0	0	0	0	0	1	0	1	5
Operator	|	|	|	|	|	|	|	|	|
Operand2	0	0	0	0	0	1	1	0	6
Result	0	0	0	0	0	1	1	1	7

Fig. 3.53 *Table to show how resulting values are calculated using the bitwise OR operator*

The example in Fig 3.53 and the truth table in Fig 3.52 suggest that the only way to get a resultant value of 0 is if both operands using the bitwise *OR* operator is zero.

All possible combinations for bitwise exclusive OR (^)				
Operand1 exclusive OR (^) Operand2	Binary			
Operand1	1	1	0	0
Operator	^	^	^	^
Operand2	1	0	1	0
Result	0	1	1	0

Fig. 3.54 *Truth table to show all possible results from the bitwise exclusive OR operator*

Bitwise *exclusive OR* example calculation									
Operand1 exclusive OR(^) Operand2	**Binary**								**Decimal**
Operand1	0	0	0	0	1	0	1	0	5
Operator	^	^	^	^	^	^	^	^	^
Operand2	1	1	0	1	0	0	1	0	210
Result	1	1	0	1	1	0	0	0	216

Fig. 3.55 *Table to show how resulting values are calculated using the bitwise exclusive OR operator*

Lastly, for the bitwise exclusive *OR* operator, the way to get a resultant value of 1 is if either of the operands value are 1 but both cannot be 1 and the only way to get a resultant value of 0 is if either 1 of the operands value is 0 but both cannot be 0.

Text String/Variable Operators for IF statements			
Operator	**Definition**	**Result: TRUE**	**Result: FALSE**
= or ==	total Unicode equivalent of text string equal to	"hello" = "hello"	"hello" = "hi"
<> or !=	total Unicode equivalent of text string not equal to	"hello" != "hi"	"hello" != "hello"
<	total Unicode equivalent of text string less than	"hello" < "empty"	"hello" < "world"
>	total Unicode equivalent of text string greater than	"hello" > "world"	"hello" > "thanks"
<=	total Unicode equivalent of text string less than or equal to	"hello" <= "allow"	"hello" ~= "thanks"
>=	total Unicode equivalent of text string greater than or equal to	"hello" >= "hello"	"hello" ~= "thanks"
~=	same text length	"hello" ~= "empty"	"hello" ~= "thanks"
~<	text length shorter than	"hello" ~< "longest"	"hello" ~< "hi"
~>	text length greater than	"hello" ~> "hi"	"hello" ~> "really long"
~!	not same text length	"hello" ~! "thanks"	"hello" ~! "allow"

Fig. 3.56 *Table describing the different operators in comparing text strings/variables in iDev*

Text strings can also be compared in iDev and hence included in if statements. This can be used in various iDev projects such as one that requires password input from the user. Most of the operators used to compare text strings are self-explanatory except =, !=, <, >, <= and >=. These operators actually compare the total value of the text string in Unicode. Each character in a text string is represented by a Unicode that contains a unique value in hex and decimal. So an example is the word 'hello', unicode for *h* is 68(hex) and 104(decimal), *e* is 65(hex) 101(decimal), *l* is 6C(hex) 108(decimal) and finally *o* is 6F(hex) 111(decimal). The decimal values are all added which is then compared to another text strings total value in decimal.

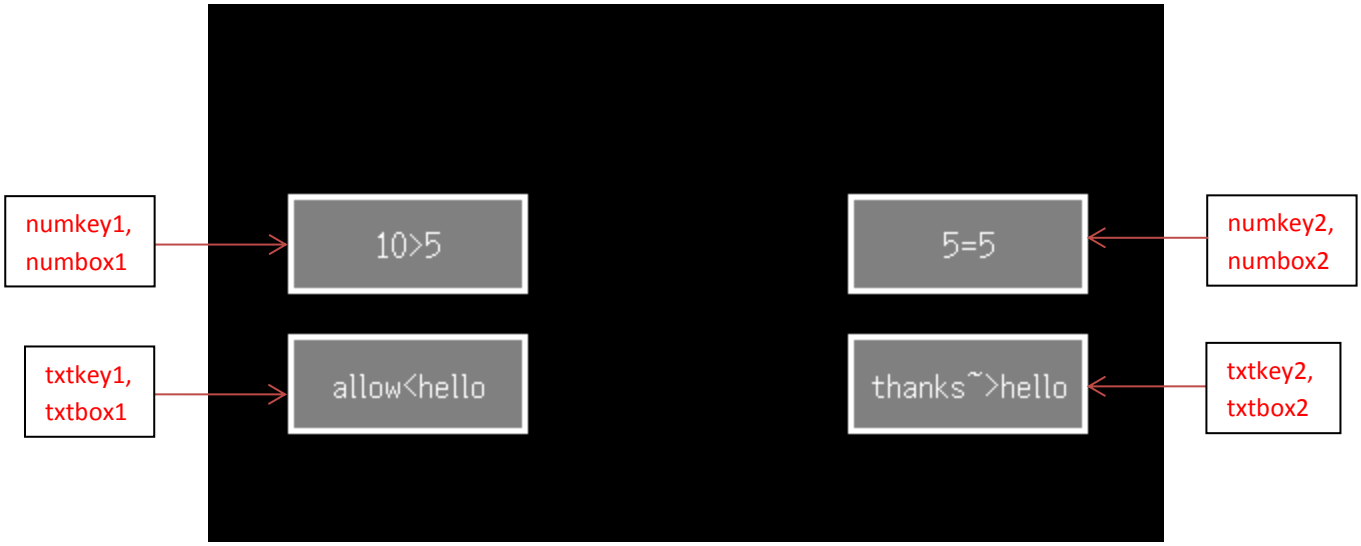


Fig. 3.57 *Screen shot displaying when numkey1 is pressed*

Fig. 3.58 *Screen shot displaying when txtkey2 is pressed*

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
VAR command format:
VAR(Variable name, Starting value, Variable Style);
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value/Variable);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
LOOP command format:
Loop Header
LOOP(Loop name, Loop duration)
Loop Body
{
IF command format to create if statements with just one action, note that a *Operand1* can be a variable and *Operand2* can be a literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function);
IF command format to create if statements with an else action, note that a *Operand1* can be a variable and *Operand2* can be a variable, literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function1:Function2);
}
}
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

STYLE(homepgst,Page)      //
{
back = green;             //
}
STYLE(Ascii16txst,Text)   //
{
font = Ascii16;           //
col = white;              //
maxRows = 1;              //
maxLen = 32;              //
}
STYLE(boxdrwst,Draw)      //
{
type = b;                 //
maxX = 120;               //
maxY = 50;                //
width = 3;                //
back = grey;              //
col = white;              //
}
VAR(myintvar,0,S8);       //
VAR(mytxtvar,"",TXT);     //

PAGE(homepg,homepgst)                        //
{
POSN(100,120);                               //
KEY(numkey1,[LOAD(myintvar,10);],120,50,TOUCH); //
DRAW(numbox1,120,50,boxdrwst);               //
TEXT(numtxt1,"10>5",Ascii16txst);            //
POSN(380,+0);                                //
KEY(numkey2,[LOAD(myintvar,5);],120,50,TOUCH);  //
DRAW(numbox2,120,50,boxdrwst);               //
TEXT(numtxt2,"5=5",Ascii16txst);             //
POSN(100,+70);                               //
KEY(txtkey1,[LOAD(mytxtvar,"allow");],120,50,TOUCH);
//
DRAW(txtbox1,120,50,boxdrwst);               //
TEXT(txt1,"allow<hello",Ascii16txst);        //
POSN(380,+0);                                //
KEY(txtkey2,[LOAD(mytxtvar,"thanks");],120,50,TOUCH);
//
DRAW(txtbox2,120,50,boxdrwst);               //
TEXT(txt2,"thanks~>hello",Ascii16txst);      //

LOOP(myloop,FOREVER)   //
{
IF(myintvar > 5?[LOAD(homepgst.back,black);LOAD(myintvar,0);;]);
//
IF(myintvar = 5?[LOAD(homepgst.back,red);LOAD(myintvar,0);;]);
//
IF(mytxtvar < "hello"?[LOAD(homepgst.back,purple);LOAD(mytxtvar,"");;]);
//
IF(mytxtvar ~> "hello"?[LOAD(homepgst.back,orange);LOAD(mytxtvar,"");;]);
//
}
}
SHOW(homepg);          //
```

Fig. 3.59 *Example code to show the application of IF statements in iDev*

The example code in Fig 3.59 uses a loop that checks the state of the variables *myintvar* and *mytxtvar*. When the condition of the *IF* statements are true then the inline command is carried out. The inline command uses the *LOAD* command dot operator to change the background of the page. Each key component loads a different value to the variable which

satisfies one of the *IF* statements in the loop and triggers the inline command concerned. The screen shots below shows what would be displayed if numkey1 and txtkey2 is pressed.

3.5. CASE – SWITCH/SELECT

In iDev, there is a method that emulates the *SELECT/SWITCH CASE* function found in other programming languages. This method enables the developer to use minimal amount of *IF* statements for an iDev project that requires long and multiple *IF* statements. The *SELECT/SWITCH CASE* method in other languages tests the contents of a variable and selectively process data according to its value. In iDev, functions located anywhere in the program can be called on-demand provided that they use a common naming method. It is possible compile a function name in a variable and then use the *RUN(variable name);* command in iDev. So together with the *CALC* command, the iDev case equivalent method ends up with less lines of code achieving the same results compared to that of the case function method in other programming languages. An example code using 5 different cases is created and applied in a basic iDev project.

Typical iDev Case Switch/Select method note that *chkstr, input* and *runfnc* are text variables, *caseval* is a S8 integer variable that were predefined:

```
LOAD(chkstr, ",", input, ",");
CALC(caseval, ", 1AG, 2GQ, 3TE, 4PL, ", chkstr, "FIND");
IF(caseval < 0? case_default: [LOAD(runfnc, "case_", input); RUN(runfnc);]);
```

In the iDev project, the variable *input* is a dynamic variable that contains changing text strings.

```
LOAD(chkstr, ",", input, ",");
```

The first line in the iDev case method encloses the contents of the variable *input* in commas i.e. if the contents of *input* is “J6E” then the resultant text string loaded to the text variable *chkstr* is “,J6E,”.

```
CALC(caseval, ", 1AG, 2GQ, 3TE, 4PL, ", chkstr, "FIND");
```

The next line then checks the contents of *chkstr* and compares it to the list of ‘acceptable’ text strings (case) that the developer has specified. In the example the ‘acceptable’ text strings are “1AG, 2GQ, 3TE, 4PL”, it is obvious that “J6E” is not in this list. This would cause the *CALC* command to return a value of -1 to the S8 (signed 8 bit integer) variable *caseval*.

```
IF(caseval < 0? case_default: [LOAD(runfnc, "case_", input); RUN(runfnc);]);
```

Lastly, the *if* statement checks if value of caseval is less than 0 and then run the appropriate functions. Since the text string “J6E” is not found in the ‘acceptable’ list then it runs the ‘default’ function named *case_default*. On the other hand if the contents of *input* is changed to “3TE” then the *CALC* command would return a value of 1 affecting the *if* statement to process the alternative *LOAD* and *RUN* commands. The *CALC* command that uses the *“FIND”* method is properly introduced in Chapter 3.6.2 of this guide. The *LOAD* command changes the input contents to the common naming method that is used; in this example with the contents of input being “3TE”, the new value of *runfnc* is now “*case_3TE”.* Effectively the *RUN* command would end up processing *RUN (case_3TE).* It is important to remember that there are functions called *case_1AG, case_2GQ, case_3TE, case_4PL* defined before the case method function is created. For a clearer understanding a fully functional example that uses the iDev case switch/select method is created.

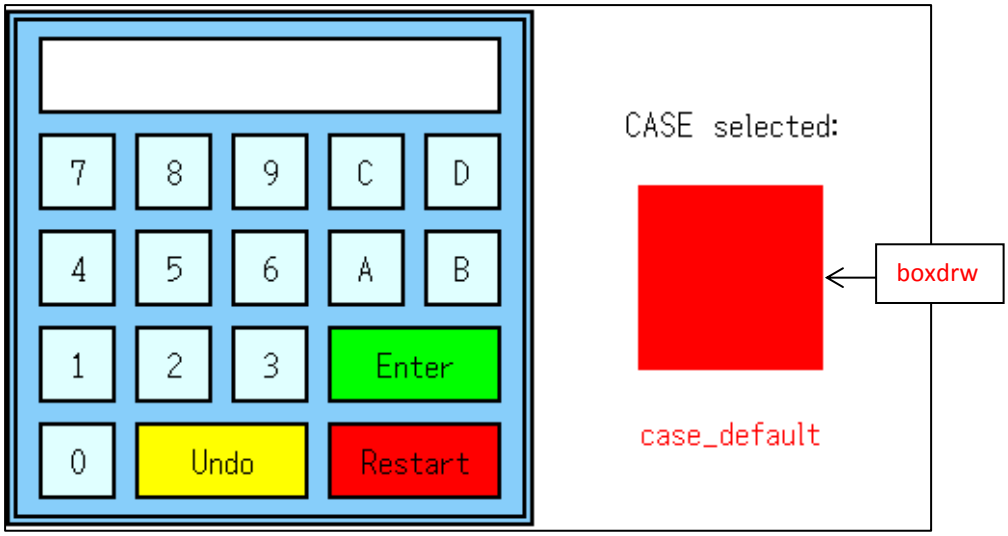


Fig. 3.60 *Screen shot showing changes when case value is case_default*

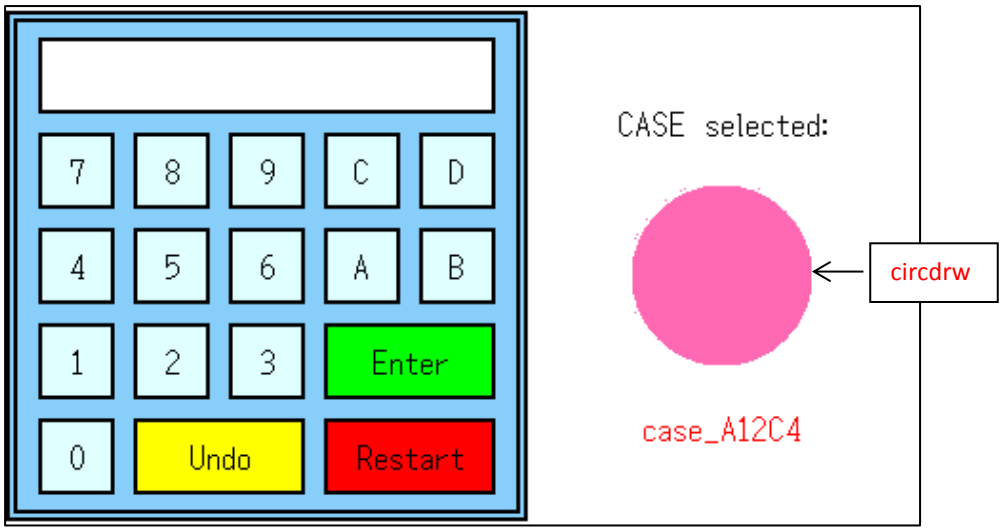


Fig. 3.61 *Screen shot showing changes when case value is case_A12C4*

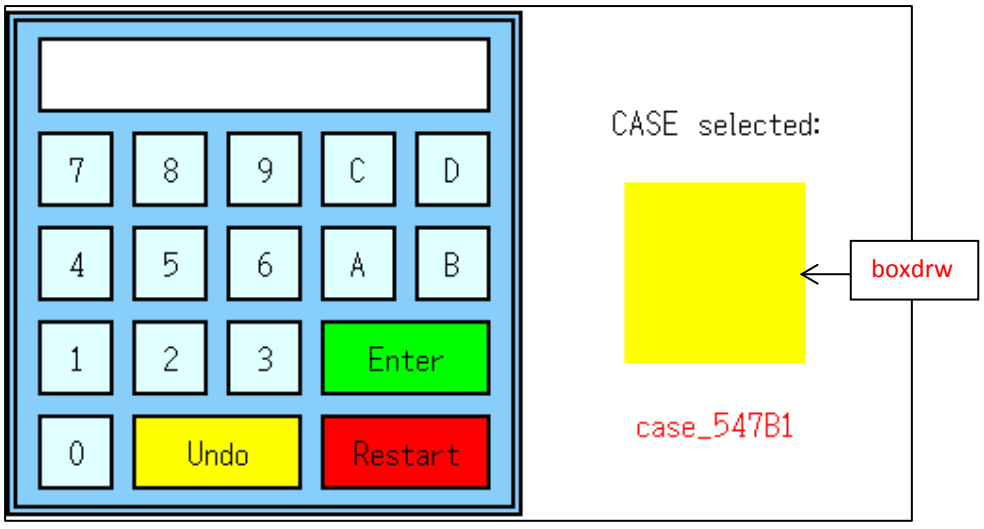


Fig. 3.62 *Screen shot to showing changes when case value is case_547B1*

LIB command format for images:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}
VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
LOOP command format:
Loop Header
LOOP(Loop name, Loop duration)
Loop Body
{
Loop contents…
}
}
SHOW command format:
SHOW(Page name or page component name);
HIDE command format:
HIDE(Page name or page component name);

FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
LOAD command format to update styles:
LOAD(Style name.Parameter, New Parameter Value);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
LOAD command format to change stored text data from multiple sources:
LOAD(Destination variable name, Text data source1, Text data source2…);
IF command format to create if statements with an else action, note that a *Operand1* can be a variable and *Operand2* can be a variable, literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function);
Typical iDev Case Switch/Select method, note that *chkstr, input* and *runfnc* are text variables, *caseval* is a S8 integer variable that were predefined:

```
LOAD(chkstr, ",", input, ",");
CALC(caseval, ", 1AG, 2GQ, 3TE, 4PL, ", chkstr, "FIND");
IF(caseval < 0? case_default: [LOAD(runfnc, "case_", input); RUN(runfnc);]);
```

```
//FILENAME: TU480a.mnu

LIB(backgrnd,"SDHC/Back.bmp");  //

STYLE(mypagest,Page)            //
{
image = backgrnd;               //
}
STYLE(Ascii16txst,Text)         //
{
font = Ascii16;                 //
col = black;                    //
maxLen = 24;                    //
curRel = RC;                    //
}
STYLE(casetxst,Ascii16txst)     //
{
col = red;                      //
curRel = CC;
}
STYLE(titletxst,casetxst)       //
{
col = black;                    //
}
STYLE(drwbxst,Draw)             //
{
type = b;                       //
back = red;                     //
width = 2;                      //
}
STYLE(drwcrst,drwbxst)          //
{
type = c;                       //
}

VAR(caseval,0,S8);              //
VAR(input,"",TXT);              //
VAR(chkstr,"",TXT);             //
VAR(runfnc,"",TXT);             //

PAGE(homepg,mypagest)                              //
{
POSN(35,185);                                      //
KEY(key1,[LOAD(input,input,1);],50,50,TOUCH);      //
POSN(85,185);                                      //
KEY(key2,[LOAD(input,input,2);],50,50,TOUCH);      //
POSN(135,185);                                     //
KEY(key3,[LOAD(input,input,3);],50,50,TOUCH);      //
POSN(35,135);                                      //
KEY(key4,[LOAD(input,input,4);],50,50,TOUCH);      //
POSN(85,135);                                      //
KEY(key5,[LOAD(input,input,5);],50,50,TOUCH);      //
POSN(135,135);                                     //
KEY(key6,[LOAD(input,input,6);],50,50,TOUCH);      //
POSN(35,85);                                       //
KEY(key7,[LOAD(input,input,7);],50,50,TOUCH);      //
POSN(85,85);                                       //
KEY(key8,[LOAD(input,input,8);],50,50,TOUCH);      //
POSN(135,85);                                      //
KEY(key9,[LOAD(input,input,9);],50,50,TOUCH);      //
POSN(35,235);                                      //
KEY(key0,[LOAD(input,input,0);],50,50,TOUCH);      //
```

```
POSN(185,85);                                    //
KEY(keyC,[LOAD(input,input,"C");],50,50,TOUCH);  //
POSN(235,85);                                    //
KEY(keyD,[LOAD(input,input,"D");],50,50,TOUCH);  //
POSN(185,135);                                   //
KEY(keyA,[LOAD(input,input,"A");],50,50,TOUCH);  //
POSN(235,135);                                   //
KEY(keyB,[LOAD(input,input,"B");],50,50,TOUCH);  //

POSN(210,185);                        //
KEY(keyOK,entfunc,100,50,TOUCH);      //
POSN(110,235);                        //
KEY(keybs,fncbsp,100,50,TOUCH);       //
POSN(210,235);                        //
KEY(keydel,delfunc,100,50,TOUCH);     //

POSN(240,35);                                 //
TEXT(passtxt,"",Ascii16txst);                 //
POSN(375,140);                                //
DRAW(boxdrw,100,100,drwbxst);                 //
DRAW(circdrw,100,100,drwcrst);                //
HIDE(circdrw);                                //
POSN(375,60);                                 //
TEXT(title,"CASE selected:",titletxst);       //
POSN(375,220);                                //
TEXT(casetxt,runfnc,casetxst);                //

LOOP(textlp,FOREVER)            //
{
TEXT(passtxt,input);;           //
}
}

FUNC(delfunc)                   //
{
LOAD(input,"");                 //
}
FUNC(fncbsp)                    //
{
IF(input ~> 0?[CALC(input,input,-1,"DEL");TEXT(passtxt,input);;]);
//
}
FUNC(case_11C34)                //
{
SHOW(boxdrw);                   //
HIDE(circdrw);                  //
LOAD(drwbxst.back,lime);;       //
}
FUNC(case_D5541)                //
{
SHOW(circdrw);                  //
HIDE(boxdrw);                   //
LOAD(drwcrst.back,blue);;       //
}
FUNC(case_547B1)                //
{
SHOW(boxdrw);                   //
HIDE(circdrw);                  //
LOAD(drwbxst.back,yellow);;     //
}
FUNC(case_A12C4)                //
{
SHOW(circdrw);                  //
HIDE(boxdrw);                   //
LOAD(drwcrst.back,hotpink);;    //
}
```

```
FUNC(case_default)                //
{
SHOW(boxdrw);                     //
HIDE(circdrw);                    //
TEXT(casetxt,"case_default");     //
LOAD(drwbxst.back,red);;          //
}
FUNC(deffunc)                     //
{
RUN(case_default);                //
TEXT(casetxt,"case_default");;    //
}
FUNC(otherfunc)                   //
{
LOAD(runfnc,"case_",input);       //
RUN(runfnc);                      //
TEXT(casetxt,runfnc);;            //
}
FUNC(entfunc)                     //
{
LOAD(chkstr , "," , input , ",");         //
CALC(caseval, ",11C34,D5541,547B1,A12C4," , chkstr, "FIND");
//
IF(caseval < 0 ? deffunc : otherfunc);    //
RUN(delfunc);                     //
}
SHOW(homepg);                     //
```

Fig. 3.63 *Example code demonstrating how the iDev case switch/select method is applied*

The example code in Fig 3.63 displays a keypad which allows the user to input data strings. When the user press the “Enter” button the function that uses the case switch/select method is called. In total there are 5 different cases that is in this example code, each one producing different coloured shapes on the right hand side of the screen to verify that the function associated with the case is correct. Also the case that has been selected is displayed as a text component on the bottom right hand side of the screen. The shape manipulation is achieved by using the *SHOW*, *HIDE* and the *LOAD* commands as evident from the example code above.

3.6. CALCULATION

The ability to make calculations in iDev is crucial to increase the versatility and capability of iDev projects. Calculations can be carried out in numerical values and text strings in iDev. In essence, it is used for numeric, mathematics, trigonometric, text and buffer manipulation, file handling and checksums for data buffers.

3.6.1. ARITHMETIC

This section assumes that the developer knows the basic operations involving mathematics, trigonometry and bitwise logical operations. If an explanation is required for bitwise logical operations, then refer to Chapter 3.4. It is important to remember that the result of the variable has to be stored in a suitable variable type i.e. the result of an arithmetic calculation should not be stored in a text variable but to an integer or a floating decimal point variable. For better explanation of each method in iDev, a table that shows how it is applied in iDev with a calculator equivalent column is produced.

CALC command format for most arithmetic methods:
CALC(Destination Variable, Operand 1, Operand2, "Method");

Numeric Handling			
Method	**Definition**	**Usage Example**	**Calculator Equivalent**
+	Operand1 plus Operand2	CALC(myintvar,2,5,"+");	2 + 5 = 7
-	Operand1 minus Operand2	CALC(myintvar,81,52,"-");	81 - 52 = 29
/	Operand1 divided by Operand2	CALC(myintvar,27,3,"/");	27 ÷ 3 = 9
*	Operand1 multiplied by Operand2	CALC(myintvar,47,17,"*");	47 × 17 = 799
%	Operand 1 modulus of Operand2	CALC(myintvar,26,4,"%");	26 Mod 4 = 2
|	Operand1 bitwise OR Operand2	CALC(myintvar,5,84,"&");	5 AND 84 = 4
&	Operand1 bitwise AND Operand2	CALC(myintvar,5,6,"|");	5 OR 6 = 7
^	Operand1 bitwise exclusive OR Operand2	CALC(myintvar,5,210,"^");	5 XOR 210 = 215

Fig. 3.64 *Table describing different methods for numeric handling in iDev*

Mathematical Functions			
Method	**Definition**	**Usage Example**	**Calculator Equivalent**
ABS	Absolute value of Operand1	CALC(myintvar, -16, "ABS");	abs(-16) = 16
EXP	Exponential function of Operand1	CALC(myfltvar,14,"EXP");	exp(14) = 1.2026e6
LOG	Natural Logarithm of Operand1	CALC(myfltvar,27,"LOG");	ln(27) = 3.29584
LOG10	Base-Ten Logarithm of Operand1	CALC(myfltvar,56,"LOG10");	log(56) = 1.74819
POW	Operand1 raised to the power of Operand2	CALC(myintvar,2,32,"POW");	2^{32} = 4294967296
SQRT	Non-negative square root of Operand1	CALC(myintvar,64,"SQRT");	√64 = 8
CBRT	Cube root of Operand1	CALC(myintvar,64,"CBRT");	$\sqrt[3]{64}$ = 4
RND	Random number(with range 0 to 0.99999) multiplied by Operand1	CALC(myfltvar,2,"RND");	rnd(2)= 1.9475

Fig. 3.65 *Table describing different methods for mathematical function in iDev*

Trigonometric Functions (degrees)			
Method	**Definition**	**Usage Example**	**Calculator Equivalent**
COS	Cosine of Operand1	CALC(myfltvar, 15, "COS");	cos (15) = 0.965926
SIN	Sine of Operand1	CALC(myfltvar, 25, "SIN");	sin (25) = 0.422618
TAN	Tangent of Operand1	CALC(myfltvar, 63, "TAN");	tan(63) = 1.96261
ACOS	Arc Cosine of Operand1	CALC(myfltvar, 0.57, "ACOS");	$\cos^{-1}$(0.57) = 55.249774
ASIN	Arc Sine of Operand1	CALC(myfltvar, 0.84, "ASIN");	$\sin^{-1}$(0.84) = 57.140119
ATAN	Arc Tangent of Operand1	CALC(myfltvar, 42, "ATAN");	$\tan^{-1}$(42) = 88.636072
ATAN2	Arc Tangent of Operand1/Operand2	CALC(myfltvar, 63, 21, "ATAN2");	$\tan^{-1}$(63÷21) = 71.565051
COSH	Hyperbolic Cosine of Operand1	CALC(myfltvar, 4, "COSH");	cosh(4) = 27.308233
SINH	Hyperbolic Sine of Operand1	CALC(myfltvar, 12, "SINH");	sinh(12) = 81377.395706
TANH	Hyperbolic Tangent of Operand1	CALC(myfltvar, 3, "TANH");	tanh(3) = 0.052408
ACOSH	Hyperbolic Arc Cosine of Operand1	CALC(myfltvar, 13, "ACOSH");	$\cosh^{-1}$(13) = 3.256614
ASINH	Hyperbolic Arc Sine of Operand1	CALC(myfltvar, 3, "ASINH");	$\sinh^{-1}$(3) = 1.818446
ATANH	Hyperbolic Arc Tangent of Operand1	CALC(myfltvar, 0.24, "ATANH");	$\tanh^{-1}$(0.24) = 0.244774

Fig. 3.66 *Table describing different methods for trigonometry in iDev*

If trigonometric calculations in radians are required by the developer then this can be changed in the system setup (see Chapter 1.5). The parameter has to be changed to radians e.g. *angle = radians.* A combination of the basic methods and operations in mathematics can be used to create a basic calculator. The basic calculator that will be created can add, subtract, divide and multiply on up to 24 digits.

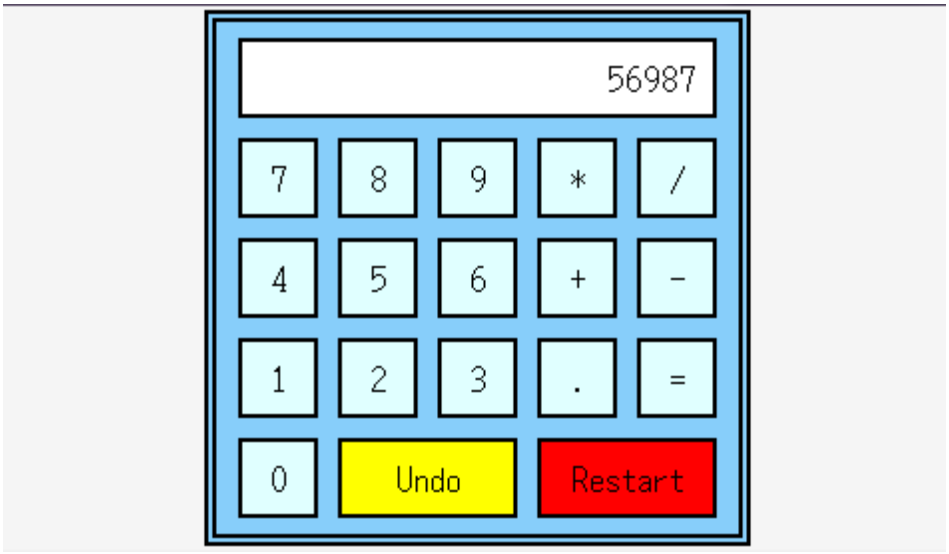


Fig. 3.67 *Screen shot to demonstrate what will be observed on the display when the code in Fig 3.68 is uploaded onto the TFT module*

LIB command format for images:
LIB(Library image name, "Source/Filename");
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
VAR command format:
VAR(Variable name, Starting value, Variable Style);
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value/Variable);

DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
LOOP command format:
Loop Header
LOOP(Loop name, Loop duration)
Loop Body
{
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
IF command format to create if statements with just one action, note that a *Operand1* can be a variable and *Operand2* can be a literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function);
CALC command format for most arithmetic methods
CALC(Destination Variable, Operand 1, Operand2, Method);
}
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

LIB(backlibimg,"sdhc/Backg.bmp");  //

STYLE(homepgst,Page)               //
{
image = backlibimg;                //
}
STYLE(Ascii16txst,Text)            //
{
font = Ascii16;                    //
col = black;                       //
maxLen = 24;                       //
curRel = RC;                       //
}

VAR(Entry,"",TXT);                 //
VAR(Operator,"",TXT);              //
VAR(Var1,0,FLT4);                  //
VAR(Var2,0,FLT4);                  //
VAR(Ans1,0,FLT4);                  //

PAGE(Homepage,homepgst)                          //
{
POSN(350,35);                                    //
TEXT(passtxt,"", Ascii16txst);                   //
POSN(140,185);                                   //
KEY(key1,[LOAD(Entry,Entry,1);],50,50,TOUCH);    //
POSN(190,185);                                   //
KEY(key2,[LOAD(Entry,Entry,2);],50,50,TOUCH);    //
POSN(240,185);                                   //
KEY(key3,[LOAD(Entry,Entry,3);],50,50,TOUCH);    //
POSN(140,135);                                   //
KEY(key4,[LOAD(Entry,Entry,4);],50,50,TOUCH);    //
POSN(190,135);                                   //
KEY(key5,[LOAD(Entry,Entry,5);],50,50,TOUCH);    //
POSN(240,135);                                   //
KEY(key6,[LOAD(Entry,Entry,6);],50,50,TOUCH);    //
POSN(140,85);                                    //
KEY(key7,[LOAD(Entry,Entry,7);],50,50,TOUCH);    //
POSN(190,85);                                    //
KEY(key8,[LOAD(Entry,Entry,8);],50,50,TOUCH);    //
POSN(240,85);                                    //
KEY(key9,[LOAD(Entry,Entry,9);],50,50,TOUCH);    //
POSN(140,235);                                   //
KEY(key0,[LOAD(Entry,Entry,0);],50,50,TOUCH);    //
POSN(290,85);                                    //
KEY(mulkey,[LOAD(Operator,"*");RUN(varlupdfunc);],50,50,TOUCH);
//
POSN(340,85);                                    //
KEY(divkey,[LOAD(Operator,"/");RUN(varlupdfunc);],50,50,TOUCH);
//
POSN(290,135);                                   //
KEY(plskey,[LOAD(Operator,"+");RUN(varlupdfunc);],50,50,TOUCH);
//
POSN(340,135);                                   //
KEY(minkey,[LOAD(Operator,"-");RUN(varlupdfunc);],50,50,TOUCH);
//
POSN(290,185);                                   //
KEY(dotkey,[LOAD(Entry,Entry,".");],50,50,TOUCH);
//
POSN(340,185);                                   //
KEY(OKkey,[RUN(equalfunc);],50,50,TOUCH);        //
POSN(215,235);                                   //
KEY(bskey,bspfunc,100,50,TOUCH);                 //
POSN(315,235);                                   //
KEY(delkey,delfunc,100,50,TOUCH);                //
```

```
LOOP(keyloop,FOREVER)    //
{
TEXT(passtxt,Entry);;    //
}
}

FUNC(delfunc)            //
{
LOAD(Ans1,0);            //
LOAD(Var1,0);            //
LOAD(Var2,0);            //
VAR(Operator,"");        //
LOAD(Entry,"");          //
}
FUNC(bspfunc)            //
{
IF(Entry ~> 0?[CALC(Entry,Entry,-1,"DEL");TEXT(passtxt,Entry);;]);
//
}
FUNC(var1updfunc)        //
{
LOAD(Var1,Entry);        //
LOAD(Entry,"");          //
TEXT(passtxt,Entry);;    //
}
FUNC(equalfunc)          //
{
LOAD(Var2,Entry);        //
RUN(calcfunc);           //
LOAD(Entry,Ans1);        //
TEXT(passtxt,Entry);;    //
}
FUNC(calcfunc)           //
{
IF(Operator = "+"?[CALC(Ans1,Var1,Var2,"+");]);
//
IF(Operator = "-"?[CALC(Ans1,Var1,Var2,"-");]);
//
IF(Operator = "/"?[CALC(Ans1,Var1,Var2,"/");]);
//
IF(Operator = "*"?[CALC(Ans1,Var1,Var2,"*");]);
//
}
SHOW(Homepage);          //
```

Fig. 3.68 *Example code to demonstrate the CALC command and create a fully functional basic calculator*

The background image of the page consists of an image of a numpad with a text screen on top to emulate what a calculator looks like. This is a trick that can be done in similar projects so instead of creating draw components and text components for each number and operators, a whole image can just be set as a background which would achieve the same purpose to indicate the position of the keys. The key components are positioned accordingly to load correct numbers to the variables when pressed. The example above uses variables to store appropriate values in the parameters of the *CALC* command. The variable *Var1* is used to store the value for Operand1, variable *Var2* is used to store the value for Operand2. There is a loop created which updates the text component in the text screen to notify the user which numbers have been pressed and stored for calculation. Then a function that set appropriate operators is called when the “=” button is pressed.

The *CALC* command format for every operation is created and is called depending on which operator is pressed by the user. The line *LOAD(Entry,Entry,1);* and all the other similar ones are used to add the value of the key pressed to the current contents of the variable *Entry.* For example if the number 56987 is 'typed' into the text screen then the user would have to press the key that calls *LOAD(Entry,Entry,5);* first. The variable *Entry's* starting value is 0 so this line would result in assigning 5 in the variable *Entry.* Then the key that calls *LOAD(Entry,Entry,6)* is called which adds 6 to the current value stored in *Entry* resulting in loading 56 in the variable Entry, the same method occurs when the buttons 9, 8 and 7 are pressed. After this, when an operator is pressed the previously entered value stored in variable *Entry* is stored in *Var1* and is set as Operand1 in the *CALC* command parameter. Then after the second value is entered and the '=' button is pressed then the second value is loaded is stored in *Var2* and set as Operand2. Finally the appropriate *CALC* command is called depending on which operator has been pressed. The 'Undo' and 'Restart' button uses the *CALC* command as well but it uses text and cursor handling methods for text string manipulation, this is explained further in Chapter 3.6.2.

3.6.2. TEXT STRINGS

In iDev, the *CALC* command can be used to manipulate text strings and cursors where editable text is to be placed on the screen similar to what can be seen in a calculator or editable text field. Different methods and combinations allow cursor movement and type, text insertion and deletion, find or delete text, cursor position and length. The destination variable has to be of suitable type for different methods used, i.e. if the result is always going to be a numerical value then the variable should be an integer variable and if text string then the variable should be a text variable.

CALC command format for most text string methods:
CALC(Destination Variable, Operand1, Operand2, Method);

Text and Cursor Handling	
Method and Definition	**Usage Example and Result**
POS • move the cursor of text in Operand1 to absolute position (0 to n) Operand2 • if absolute position is less than zero, then cursor is put before the first character and if greater than the length of the text in Operand1 then the cursor is placed after the last character • the 1st character in a string is position 0	CALC(mytxtvar, "hello", 2, "POS"); = text string "hello" with cursor at the 3rd character('l') is stored in mytxtvar

REL • move cursor of text in Operand1 from current cursor placement to specified place in Operand2 • **positive** values in Operand2 move the cursor to the **right** and **negative** values move the cursor to the **left** • if the move results in a cursor position less than zero then the cursor is put before the first character and if greater than the length of the text in Operand1 then the cursor is placed after the last character	CALC(mytxtvar, "hello", 2, "REL"); = text string "hello" with cursor at the last character is stored in mytxtvar because the previous cursor position is at the last character
INS • insert/overwrite text at Operand2 into Operand1 at the cursor • the text will either be overwritten or inserted depending on the cursor type of text in Operand1 • if no cursor is present then the text is added to the end of text in Operand1	CALC(mytxtvar, "hello", "world", "INS"); = provided that the cursor is at the 2nd character and cursor type insert, the text string "heworldllo" is stored in mytxtvar
DEL • deletes the number of characters (Operand2) from text in Operand1 at the cursor • if the number of characters in Operand2 is **positive** then the number of characters specified will be deleted **after** the cursor • if the number of characters in Operand2 is **negative** then the number of characters specified will be deleted **before** the cursor • if no cursor is present and number of characters specified in Operand2 is negative, then the characters will be deleted from the end of the text in Operand1 • if no cursor is present and number of characters specified in Operand2 is positive , then the characters will be deleted from the start of the text in Operand1	CALC(mytxtvar, "hello", -1, "DEL"); = if the cursor is at the 4th character, the text string "hlo" is stored in mytxtvar
TRIM • remove characters specified in Operand2 from start and end of text string in Operand1 • if the characters specified in Operand2 is empty ("") then spaces (20hex), tab (09hex), line feeds (0Ahex), and carriage returns (0Dhex) are removed	CALC(mytxtvar, "hello", "llo", "TRIM"); = the text "he" is stored in mytxtvar

LTRIM • removes characters specified in Operand2 from start of text string in Operand1 • if the characters specified in Operand2 is empty ("") then spaces (20hex), tab (09hex), line feeds (0Ahex), and carriage returns (0Dhex) are removed	CALC(mytxtvar, "hello", "llo", "LTRIM"); = the text "he" is stored in mytxtvar
RTRIM • remove characters specified in Operand2 from end of text string in Operand1 • if the characters specified in Operand2 is empty ("") then spaces (20hex), tab (09hex), line feeds (0Ahex), and carriage returns (0Dhex) are removed	CALC(mytxtvar, "hello", "eh", "RTRIM"); = the text "llo" is stored in mytxtvar
UPPER • convert text in Operand1 to uppercase • the resultant text is stored in destination variable	CALC(mytxtvar, "hello", 0, "UPPER"); = the text "HELLO" is stored in mytxtvar
LOWER • convert text in Operand1 to lowercase • the resultant text is stored in destination variable	CALC(mytxtvar, "HELLO", 0, "LOWER"); = the text "hello" is stored in mytxtvar
BEF • the characters specified in Operand2 are copied from before the cursor in text string Operand1 • if no cursor is present then the number of characters specified in Operand2 are copied from the end of text in Operand1 • if the number of characters specified in Operand2 is greater than the number of characters available in text in Operand1 then only the available characters are copied • if the number of characters specified in Operand2 is negative then the function performs the same method as "AFT"	CALC(mytxtvar, "hello", 2, "BEF"); = if the cursor is placed at the 3rd character then the text "he" is stored in mytxtvar

AFT • the characters specified in Operand2 are copied after the cursor in the text string Operand1 • if no cursor is present then the number of characters specified in Operand2 are copied from the start of text in Operand1 • if the number of characters specified in Operand2 is greater than the number of characters available in text in Operand1 then only the available characters are copied • if the number of characters specified in Operand2 is negative then the function performs the same method as "BEF"	CALC(mytxtvar, "hello", 3, "AFT"); = if the cursor is placed at the 2nd character then the text "llo" is stored in mytxtvar
CUR • the cursor of text in Operand1 is changed to a type specified in Operand2 • if no cursor is present, then the cursor is added to the end of the text string in Operand1 • if the type specified in Operand2 is a string then the first character is taken as the cursor type	CALC(mytxtvar, "hello", "\\03", "CUR"); = the cursor type in the text "hello" is changed to a hidden cursor with insert ON, then stored in mytxtvar
LEN • the length of the text in Operand1 and Operand2 is calculated and stored • cursor characters are not included in the length calculated • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(myintvar, "hello", 3, "LEN"); = the total length of text is 8 (5+3) and stored in myintvar
LOC • the location of the cursor of the text in Operand1 and Operand2 is stored • cursor characters are not included in the length calculated • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(myintvar, "hello", 3, "LOC"); = the current location of the cursor is 2, so the resultant location is 5 and stored in myintvar
TYPE • determine the current cursor type of text in Operand1 • if no cursor is present then a value of 0 is used • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(mytxtvar, "hello", 0, "TYPE"); = the cursor type of text "hello" is \\03 so the stored result in mytxtvar is \\03

FIND • find the first location of the match of text in Operand1 of text in Operand2 is stored in destination variable • if no matches are found then -1 is stored in the destination variable • cursor characters are not included in the calculation • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(myintvar, "hello world", "o", "FIND"); = the first location of the text is found at the 5th character of the text so the stored result in myintvar is 4
LFIND • find the last location of the match of text in Operand1 of text in Operand2 is stored in destination variable • if no matches are found then -1 is stored in the destination variable • cursor characters are not included in the calculation • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(myintvar, "hello world", "o", "LFIND"); = the last location of the text is found at the 8th character of the text so the stored result in myintvar is 7
IFIND • find the first location of the case insensitive match of text in Operand1 of text in Operand2 is stored in destination variable • if no case insensitive matches are found then -1 is stored in the destination variable • cursor characters are not included in the calculation • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(myintvar, "hello WORLD", "L", "IFIND"); = the first location of the case insensitive match is found at the 3rd character of the text so the stored result in myintvar is 2
ILFIND • find the last location of the case insensitive match of text in Operand1 of text in Operand2 is stored in destination variable • if no case insensitive matches are found then -1 is stored in the destination variable • cursor characters are not included in the calculation	CALC(mytxtvar, "hello WORLD", "ILFIND"); = the last location of the case insensitive match is found at the 10th character of the text so the stored result in mytxtvar is 9
REM • remove every occurrence of text in Operand2 from text in Operand1 • case sensitive match of text is carried out	CALC(mytxtvar, "hello worLD", "l", "REM"); = the resultant text is "heo worLD" and is stored in mytxtvar

IREM • remove every case insensitive occurrence of text in Operand2 from text in Operand1 • case insensitive match of text is carried out	CALC(mytxtvar, "hello worLD", "L", "IREM"); = the resultant text is "heo word" and is stored in mytxtvar
SPLIT • split the text in Operand1 at the character specified in Operand2 storing the text after the character in Operand1 and the text before the character into the destination variable or convert it to a number • if the specified character in Operand2 is not present then the whole text in Operand1 is copied as the result and hence stored in the destination variable • if the specified char is a string then the first character is taken as the spilt character and so the text in Operand1 is modified in this operation	CALC(mytxtvar, "hello world", "w", "SPLIT"); = the resultant text stored in Operand1 is "world" and the text "hello " is stored in mytxtvar
PIXX • determine the width in pixels of the text component Operand1 and Operand2 is stored • note that variable do not have a size and returns a 0 • text components, image components, draw components and key components and pages do have sizes • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(myintvar, mytxtvar, "0", "PIXX"); = the display width of the text in Operand1 plus the value stated in Operand2 is stored, the text mytxtvar contains a text component with the word “hello” and uses the built-in Ascii16 font style, the value stored in myintvar is 32
PIXY • determine the height in pixels of the text component Operand1 and Operand2 is stored • note that variable do not have a size and returns a 0 • text components, image components, draw components and key components and pages do have sizes • since the result is always going to be a numerical value, the destination variable has to be an integer variable type	CALC(mytxtvar, "hello", "0", " PIXY"); = the display width of the text in Operand1 plus the value stated in Operand2 is stored, the text mytxtvar contains a text component with the word “hello” and uses the built-in Ascii16 font style, the value stored in myintvar is 15

Fig. 3.69 *Table describing methods that can be used in text and cursor handling in iDev*

The table above would be a useful reference guide when the developer wants to make an iDev project that requires user manipulation of text strings such as in a keyboard. The methods in text and cursor handling enable the developer to create a

fully functional keyboard with the necessary buttons such as ‘backspace, ‘delete’, or ‘shift’. An example code is created to demonstrate the visible changes that some of the methods described in Fig 3.69 would do to an existing text string.

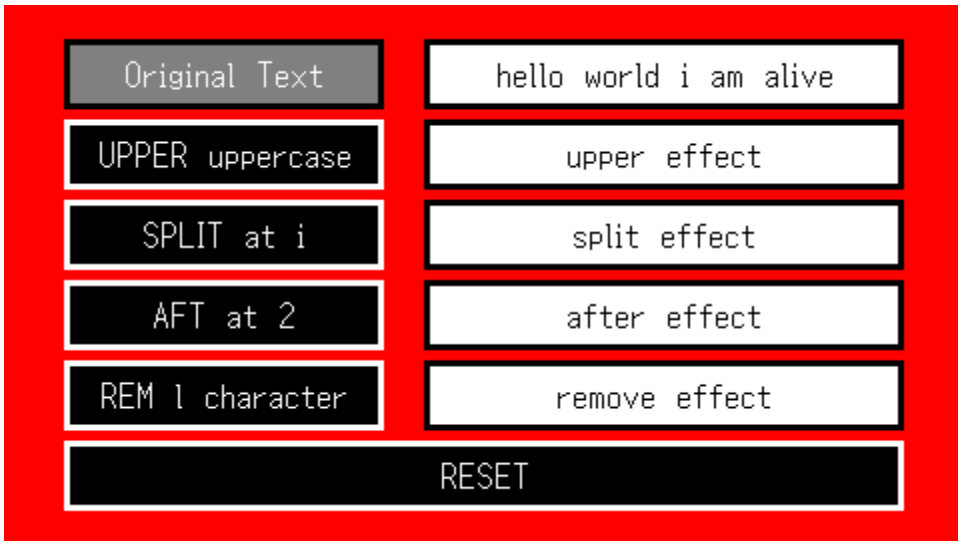


Fig. 3.70 *Screen shot showing what will be displayed before any of the buttons are pressed or when the ‘RESET’ button is pressed*

Fig. 3.71 *Screen shot demonstrating the changes that would occur when the text handling methods using the CALC command is applied*

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
VAR command format:
VAR(Variable name, Starting value, Variable Style);
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value/Variable);
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
CALC command format for most arithmetic methods:
CALC(Destination Variable, Operand 1, Operand2, Method);
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

STYLE(homepgst,Page)            //
{
back = red;                     //
}
STYLE(Ascii16txst,Text)         //
{
font = Ascii16;                 //
col = white;                    //
maxRows = 1;                    //
maxLen = 32;                    //
}
STYLE(blktxtst,Ascii16txst)     //
{
col = black;                    //
}
STYLE(boxdrwst,Draw)            //
{
type = b;                       //
maxX = 420;                     //
maxY = 50;                      //
width = 3;                      //
back = black;                   //
col = white;                    //
}
STYLE(whtboxdrwst,boxdrwst)     //
{
back = white;                   //
col = black;                    //
maxX = 240;                     //
}
STYLE(greyboxdrwst,boxdrwst)    //
{
back = grey;                    //
col = black;                    //
}

VAR(myintvar,0,S8);                                 //
VAR(mytxtvar,"",TXT);                               //
VAR(mychgtxtvar,"hello world i am alive",TXT);      //

PAGE(homepg,homepgst)                               //
{
POSN(110,35);                                       //
DRAW(orgbox,160,35,greyboxdrwst);                   //
TEXT(orgtxt,"Original Text",Ascii16txst);           //
POSN(330,+0);                                       //
DRAW(showbox,240,35,whtboxdrwst);                   //
TEXT(showtxt,mychgtxtvar,blktxtst);                 //
POSN(110,+40);                                      //
KEY(methkey1,upperfunc,160,35,TOUCH);               //
DRAW(methbox1,160,35,boxdrwst);                     //
TEXT(methtxt1,"UPPER uppercase",Ascii16txst);       //
POSN(330,+0);                                       //
DRAW(chgbox1,240,35,whtboxdrwst);                   //
TEXT(chgtxt1,"upper effect",blktxtst);              //
POSN(110,+40);                                      //
KEY(methkey2,splitfunc,160,35,TOUCH);               //
DRAW(methbox2,160,35,boxdrwst);                     //
TEXT(methtxt2,"SPLIT at i",Ascii16txst);            //
POSN(330,+0);                                       //
DRAW(chgbox2,240,35,whtboxdrwst);                   //
TEXT(chgtxt2,"split effect",blktxtst);              //
```

```
POSN(110,+40);                                  //
KEY(methkey3,aftfunc,160,35,TOUCH);             //
DRAW(methbox3,160,35,boxdrwst);                 //
TEXT(methtxt3,"AFT at 2",Ascii16txst);          //
POSN(330,+0);                                   //
DRAW(chgbox3,240,35,whtboxdrwst);               //
TEXT(chgtxt3,"after effect",blktxtst);          //
POSN(110,+40);                                  //
KEY(methkey4,remfunc,160,35,TOUCH);             //
DRAW(methbox4,160,35,boxdrwst);                 //
TEXT(methtxt4,"REM l character",Ascii16txst);   //
POSN(330,+0);                                   //
DRAW(chgbox4,240,35,whtboxdrwst);               //
TEXT(chgtxt4,"remove effect",blktxtst);         //
POSN(240,+40);                                  //
KEY(reskey,resfunc,400,35,TOUCH);               //
DRAW(resbox,420,35,boxdrwst);                   //
TEXT(restxt,"RESET",Ascii16txst);               //
}

FUNC(upperfunc)                                 //
{
CALC(mytxtvar,mychgtxtvar,0,"UPPER");           //
TEXT(chgtxt1,mytxtvar);                         //
LOAD(mychgtxtvar,"hello world i am alive");;    //
}
FUNC(splitfunc)                                 //
{
CALC(mytxtvar,mychgtxtvar,"i","SPLIT");         //
TEXT(chgtxt2,mytxtvar);                         //
LOAD(mychgtxtvar,"hello world i am alive");;    //
}
FUNC(aftfunc)                                   //
{
CALC(mytxtvar,mychgtxtvar,2,"AFT");             //
TEXT(chgtxt3,mytxtvar);                         //
LOAD(mychgtxtvar,"hello world i am alive");;    //
}
FUNC(remfunc)                                   //
{
CALC(mytxtvar,mychgtxtvar,"l","REM");           //
TEXT(chgtxt4,mytxtvar);                         //
LOAD(mychgtxtvar,"hello world i am alive");;    //
}
FUNC(resfunc)                       //
{
TEXT(chgtxt1,"upper effect");       //
TEXT(chgtxt2,"split effect");       //
TEXT(chgtxt3,"after effect");       //
TEXT(chgtxt4,"remove effect");;     //
}
SHOW(homepg);                       //
```

Fig. 3.72 *Code to demonstrate how some of the text string handling methods can be applied using the CALC command in iDev*

The example code in Fig 3.72 uses the *UPPER, SPLIT, AFT* and *REM* methods. There are 5 buttons in total that are created, the first four buttons have functions that call the *CALC* commands with their respective method and the last button is the reset button to display the starting values of the text component for each method example. Each function called when the method button is pressed has this line of code *LOAD(mychgtxtvar,"hello world i am alive");* which is important. This line of code puts the starting text string value in *mychgtxtvar* which is the text string involved in the manipulation, this allows the other methods to be independent of

each other, and otherwise the effects observed when the button is pressed would be different depending on the order that they are pressed.

3.6.3. DATA BUFFERS

In iDev, raw data that is received through a serial interface can be manipulated for its specific purpose. This gives the developer the option to set the correct buffer for their iDev projects. The *CALC* command format is added with another parameter in some methods as evident below.

CALC command format for most buffer handling methods:
CALC(Destination Variable, Operand 1, Operand2, Operand3, Method);

Method and Definition	Usage Example and Result
BCOPY • copy the length in bytes specified in Operand2 from the start or end of data string in Operand1 • if the length in bytes specified in Operand2 is negative then the data string is copied from the end	CALC(mybufvar, mytxtvar, 3, "BCOPY");
BCOPY • copy the length in bytes specified in Operand2 from the specified position in Operand3 of data string in Operand1 • if the length in bytes specified in Operand2 is negative then the data string is copied before the position specified in Operand3	CALC(mybufvar, mytxtvar, 3, 6, "BCOPY");
BCUT • remove the length in bytes specified in Operand2 from start or end of data string in Operand1 • if the length in bytes specified in Operand2 is negative then the data string is copied from the end	CALC(mybufvar, mytxtvar, 3, "BCUT");
BCUT • remove the length in bytes specified in Operand2 from the specified position in Operand3 of data string in Operand1 • if the length in bytes specified in Operand2 is negative then the data string is copied before the position specified in Operand3	CALC(mybufvar, mytxtvar, 3, 9, "BCUT");
• BINS • insert the specified data/variable in Operand2 to a specified position in Operand3 of the data in Operand1 • the resultant data string is stored in the destination variable	CALC(mybufvar, mytxtvar, myinfvar, 4, "BINS");

• BREP • replace the specified data/variable content in Operand2 by the contents of Operand1 from the position specified in Operand3 of the data in Operand1 • the string after the position set in Operand3 is replaced by the string in Operand2	CALC(mybufvar, mytxtvar, myinftvar, 7, "BREP");
• BFIND • find the first location of the data/variable specified in Operand2 from the specified data in Operand1 • the result returned is a numerical value so the destination variable has to be an integer variable	CALC(myintvar, mytxtvar, myinftvar, "BFIND");
BLFIND • find the last location of the data/variable specified in Operand2 from the specified data in Operand1 • the result returned is a numerical value so the destination variable has to be an integer variable	CALC(myintvar, mytxtvar, myinftvar, "BLFIND");
BLEN • determine the length of the data in Operand1 and number specified in Operand2 • the result returned is a numerical value so the destination variable has to be an integer variable • it is important to remember that the value specified in Operand2 is added to the result	CALC(myintvar, mybufvar, 0, "BLEN");
BTRIM • remove the number of bytes specified in Operand2 from the start and end of the data in Operand1 • the modified result is stored in the destination variable • make sure the amount of bytes specified is correct as inappropriate amount of bytes can cause errors	CALC(mybufvar, mytxtvar, 2, "BTRIM");
BLTRIM • remove the number of bytes specified in Operand2 from the start of the data in Operand1 • the modified result is stored in the destination variable • make sure the amount of bytes specified is correct as inappropriate amount of bytes can cause errors	CALC(mybufvar, mytxtvar, 3, "BLTRIM");

BRTRIM • remove the number of bytes specified in Operand2 from the end of the data in Operand1 • the modified result is stored in the destination variable • make sure the amount of bytes specified is correct as inappropriate amount of bytes can cause errors	CALC(mybufvar, mytxtvar, 1, "BRTRIM");
BREM • remove every occurrence of data specified in Operand2 from data in Operand1 • the specified string to be removed is searched from the start of the data string in Operand1	CALC(mybufvar, mytxtvar, "ell", "BREM");

Fig. 3.73 *Table to describe the buffer handling methods that can be applied in iDev*

The buffer handling methods are useful when an interface is enabled in an iDev project. The main use of these methods is in interrupts. An interrupt is setup to handle all the received data in any of the serial ports, for more information about interrupts see Chapter 4.7. An example application is when the buffer is set to receive certain keywords/commands that triggers a certain function. Using the CALC command with the appropriate methods, the keywords can be located and extracted from the buffer. The interface setup is properly introduced in Chapter 4 of this guide.

3.6.4. OTHER CALCULATION METHODS (INCOMPLETE)

NAND Directory listing method

In iDev, text variables can be populated by filenames in NAND. This option allows the developer to add 'sort' functionality in their iDev project.

CALC command format for NAND directory listing:
CALC(Destination Variable, Operand1, Operand2, Operand3, "DIR");

NAND Directory operations		
Parameter	**Expected Values**	**Definition**
Operand1	"NAND"	state the source of the list of files that requires sorting
Operand2	"*.*" "*" "*.bmp" "*.jpg" "*.png" "*.wav" "*.mp3" "*.wma" "*.fnt" "*.txt" "*.mnu"	state the type of file that should be selected (filter)
Operand3	"separator" (default ",")	state the separator for each of the files

Fig. 3.74 *Table describing the parameters for CALC command format for NAND directory listing*

Example	Result	Explanation
NAND contains: song1.mp3, song2.mp3, pic1.bmp, pic2.bmp, funcs.mnu, sound.wma, img1.jpg, img2.jpg		
CALC(mytxtvar,"NAND","DIR");	mytxtvar = "song1.mp3, song2.mp3, pic1.bmp, pic2.bmp, func.mnu"	list all files in NAND
CALC(mytxtvar,"NAND","*","DIR");	mytxtvar = "song1.mp3, song2.mp3, pic1.bmp, pic2.bmp, func.mnu"	list all files in NAND
CALC(mytxtvar,"NAND","*",",","DIR");	mytxtvar = "sound.wma, img1.jpg, img2.jpg"	list all files in NAND
CALC(mytxtvar,"NAND","*.bmp",":","DIR");	mytxtvar = "pic1.bmp:pic2.bmp"	list all bmp files
CALC(mytxtvar,"NAND","*.fnt","DIR");	mytxtvar = "0"	list none because there's no font files present in NAND
CALC(mytxtvar,"NAND","*.jpg","/","DIR");	mytxtvar = "img1.jpg/img2.jpg"	list all jpg files

Fig. 3.75 *Table showing example CALC commands to manipulate the filenames in*

the NAND directory listing

Checksums

Checksum in most programming languages allows the developer to find and sometimes fix errors in message strings or set of bits that needs accuracy. Noise can sometimes affect the data packet received during data communication which can cause errors. The first type of checksum contains the exact value/size of the packet i.e. if the sum of the other bytes in the packet is 255 or less then the checksum's value is the same. The second type of checksum is the remainder of the total value/size of the packet after it has been divided by 256. It is important that a checksum of a packet does not have to be 1 byte long (0-255), it can be longer depending on what is required. In iDev there are two forms of checksums, namely *MCHK* and *TCHK*.

CALC command format for MCHK iDev Checksums where **Operand1** is the source of the buffer and **Operand2** is the type:

CALC(Destination Buffer, Operand1, Operand2, "MCHK");

This checksum copies the buffer in Operand1 to the destination buffer. If unsure what buffer means then refer to the glossary located at end of this guide. This makes a checksum of the type specified in Operand2 and append to the destination buffer.

CALC command format for TCHK iDev Checksums where **Operand1** is the source of the buffer and **Operand2** is the type:

CALC(Result, Operand1, Operand2, "TCHK");

This checksum if for testing of the type specified in the source buffer in Operand1 and returns a result of 1 if checksums are the same and 0 if not.

Checksums iDev	
Operand2 (type)	**Definition**
"SUM8ZA"	• adds all the data in Operand1 as type U8 • checksum is two's complement of the sum • stored as two ASCII hexadecimal characters • when sum is added to checksum is zero then the result is 1
"SUM8ZD"	• adds all the data in Operand1 as type U8 • checksum is two's complement of sum • stores as single U8 • when sum is added to checksum is zero then the result is 1
"SUM8A"	• adds all data in Operand1 as type U8 • checksum is the sum • stored as two ASCII hexadecimal characters • when sum is same as checksum then the result is 1
"SUM8D"	• add all data in Operand1 as type U8 • checksum is sum • stored as single U8 • when sum is same as checksum then the result is 1
"XOR8A"	• exclusive-or (XOR) of all data in Operand1 as type U8 • checksum is XOR • stored as two ASCII hexadecimal characters • when XOR of Operand1 with checksum is zero then the result is 1

"XOR8D"	• exclusive-or (XOR) of all data in Operand1 as type U8 • checksum is XOR • stored as single U8 • when XOR of Operand1 with checksum is zero then the result is 1

Fig. 3.76 *Table describing different checksums that can be used in iDev*

Advanced Checksums

For advanced developers, CRC-16 and CRC-32 are supported in iDev. For beginner developers, here is a brief definition of the CRC method. CRC stands for cyclic redundancy check which uses a similar method to the basic checksum defined previously; the only difference is that it uses division instead of addition. This is a more accurate error detection method as it can detect more errors than the basic checksum. The 16 and 32 in CRC-16 and CRC-32 refers to the number of bits of the divisor when the CRC calculation is processed, 16 means 17 bits and 32 means 33 bits.

CALC command format for CRC16 in iDev:
CALC(Destination Variable, Operand1, Operand2, Operand3, "CRC16");

CRC-16 Support						
Operand2 (type)	**Poly-nominal**	**Initial Value**	**Reflect In**	**Reflect Out**	**XOR Out Value**	**Names and aliases**
"arc"	0x8005	0x0000	Yes	Yes	0x0000	"ARC", "CRC-16", "CRC-IBM", "CRC-16/ARC", "CRC-16/LHA"
"kermit"	0x1021	0x0000	Yes	Yes	0x0000	"KERMIT" "CRC-16/CCITT", "CRC-16/CCITT-TRUE", "CRC-CCITT"
"modbus"	0x8005	0xFFFF,	Yes	Yes	0x0000	"MODBUS"
"x-25"	0x1021	0xFFFF	Yes	Yes	0xFFFF	"X-25", "CRC-16/IBM-SDLC", "CRC-16/ISO-HDLC", "CRC-B"
"xmodem"	0x1021	0x0000	No	No	0x0000	"XMODEM", "ZMODEM", "CRC-16/ACORN"
"ccitt-f"	0x1021	0xFFFF	No	No	0x0000	"CRC-16/CCITT-FALSE"
"usb"	0x8005	0xFFFF	Yes	Yes	0xFFFF,	"CRC-16/USB"
"spi"	0x1021	0x1D0F	No	No	0x0000	"CRC-16/SPI-FUJITSU", "CRC-16/AUG-CCITT"
"buypass"	0x8005	0x0000	No	No	0x0000	"CRC-16/BUYPASS", "CRC-16/VERIFONE"
"dds-110"	0x8005	0x800D	No	No	0x0000	"CRC-16/DDS-110"
"dect-r"	0x0589	0x0000	No	No	0x0001	"CRC-16/DECT-R"
"dect-x"	0x0589	0x0000	No	No	0x0000	"CRC-16/DECT-X"
"dnp"	0x3D65	0x0000	Yes	Yes	0xFFFF	"CRC-16/DNP"
"en13757"	0x3D65	0x0000	No	No	0xFFFF	"CRC-16/EN-13757"
"genibus"	0x1021	0xFFFF	No	No	0xFFFF	"CRC-16/GENIBUS", "CRC-16/EPC", "CRC-16/I-CODE", "CRC-16/DARC"
"maxim"	0x8005	0x0000	Yes	Yes	0xFFFF	"CRC-16/MAXIM"

"mcrf4xx"	0x1021	0xFFFF	Yes	Yes	0x0000	"CRC-16/MCRF4XX"
"riello",	0x1021	0xB2AA	Yes	Yes	0x0000	"CRC-16/RIELLO"
"t10-dif",	0x8BB7	0x0000	No	No	0x0000	"CRC-16/T10-DIF"
"teledsk"	0xA097	0x0000	No	No	0x0000	"CRC-16/TELEDISK"
"tms371x"	0x1021	0x89EC	Yes	Yes	0x0000	"CRC-16/TMS37157"
"a"	0x1021	0xC6C6	Yes	Yes	0x0000	"CRC-A"

Fig. 3.77 *Table describing different CRC-16 types that can be applied in iDev*

CALC command format for CRC32 in iDev:

CALC(Destination Variable, Operand1, Operand2, Operand3, "CRC32");

CRC-32 Support						
Operand2 (type)	**Poly-nominal**	**Initial Value**	**Reflect In**	**Reflect Out**	**XOR Out Value**	**Names and Aliases**
"adcpp"	0x04C11DB7	0xFFFFFFFF	Yes	Yes	0xFFFFFFFF	"CRC-32", "CRC-32/ADCCP", "PKZIP"
"bzip2"	0x04C11DB7	0xFFFFFFFF	No	No	0xFFFFFFFF	"CRC-32/BZIP2", "CRC-32/AAL5", "CRC-32/DECT-B", "B-CRC-32"
"c"	0x1EDC6F41	0xFFFFFFFF	Yes	Yes	0xFFFFFFFF	"CRC-32C", "CRC-32/ISCSI", "CRC-32/CASTAGNOLI"
"d"	0xA833982B	0xFFFFFFFF	Yes	Yes	0xFFFFFFFF	"CRC-32D"
"mpeg-2"	0x04C11DB7	0xFFFFFFFF	No	No	0x00000000	"CRC-32/MPEG-2"
"posix"	0x04C11DB7	0x00000000	No	No	0xFFFFFFFF	"CRC-32/POSIX", "CKSUM"
"q"	0x814141AB	0x00000000	No	No	0x00000000	"CRC-32Q"
"jamcrc"	0x04C11DB7	0xFFFFFFFF	Yes	Yes	0x00000000	"JAMCRC"
"xfer"	0x000000AF	0x00000000	No	No	0x00000000	"XFER"

Fig. 3.78 *Table describing different CRC-32 types that can be applied in iDev*

Text, Draw and Image Component Information

The CALC command can also be used in gathering certain attributes about a specified component. The size, status, visibility and alignment attributes can be determined using a certain *CALC* command method.

CALC command format for deducing text, draw and image component information:
CALC(Destination Variable, Operand1, "Method");

Method	Definition	Expected Results
"ESIZE"	calculates the allocated display size	value returned is in bytes
"EDEL"	determines if the component has been deleted	if component is deleted then return value is 1, otherwise 0
"EVIS"	determines if the component is visible	if component is visible then return value is 1, otherwise 0
"EALIGN"	determine the alignment of the specified component	0 = top left
		1 = top centre
		2 = top right
		3 = centre left
		4 = centre centre
		5 = centre right
		6 = bottom left
		7 = bottom centre
		8 = bottom right

Fig. 3.79 *Table describing different methods that can be used to gain text, draw and image component attributes*

User Protocol Split

In iDev a multiple split of a data buffer to a series of variables can be applied by using the *CALC* command as well. If the developer requires to separate chunks of the data buffer into different types of variable then this method would be suitable. The buffer specified in *Operand1* is split at each character specified in *Operand2* and each separated result is stored in an incrementing series of variables prefixed with the name indicated in the *Destination Pointer*. So if the *Destination Pointer* contains myvar then the first variable will be '*myvar0',* then '*myvar1'*, '*myvar2'*, '*myvar3'*… and so on. If a particular *myvarN* is not defined then the result is not stored for that split. The data is stored in the data type format specified in each *'myvarN'* allowing the buffer to be split into text, unsigned/signed integers and floats. It is important to remember that the *Destination Pointer* parameter has to be a pointer variable because it uses this as a prefix for the destination of the split variables. If unsure on how to create pointer variables then refer to Chapter 3.1.5.

VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
CALC command format for splitting data buffers:
CALC(Destination Pointer, Operand1, Operand2, "MSPLIT");

VAR declaration and CALC command	Definition
VAR(destptr>"myvar", PTR);	create a pointer variable
VAR(myvar0, 0, U8);	create the first variable to contain the first buffer split
VAR(myvar1, 0, S32);	create the second variable to contain the second buffer split
VAR(myvar2, 0.0, FLT4);	create the third variable to contain the thirds buffer split
VAR(myvar3, "", TXT);	create the fourth variable to contain the fourth buffer split
VAR(mybuf, "14:-21:3.141:Hi", TXT);	this is where the contents of the buffer is specified, a variable containing a set amount of string is created
CALC(destptr, mybuf, ":", "MSPLIT");	the *CALC* command to split the data buffers and this results to: myvar0 = 14 myvar1 = -21 myvar2 = 3.141 myvar3 = "Hi"

Fig. 3.80 *Example to demonstrate how the CALC command is used to perform multiple splits to data buffers*

The *CALC* command format using *MSPLIT* can also be used to store the split data into array elements. The result of the divided data buffer is stored in the variables stated by successive subscripts of the pointer array. The first variable will be the variable name stored in 'myptrarray.0', then 'myptrarray.1', 'myptrarray.2', 'myptrarray.3'… and so on. Similarly, if 'myptrarrayN' is not a variable or not defined then the result is not stored for that split. The data stored in the format specified in each 'myptrarray.N' variable allowing the buffer to be split into text, unsigned/signed integers and floats. In essence, everything is the same when using a normal pointer as a destination compared to that of a pointer array, the only difference is the type of destination of the data that has been split as evident in the example below.

VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
CALC command format for splitting data buffers:
CALC(Destination Pointer, Operand1, Operand2, "MSPLIT");

VAR declaration, Array loading and CALC command	Definition
VAR(myptrarr>"", PTR, 4);???	create a one-dimensional 4 element pointer array
VAR(alpha , 0, U8);	create the first variable to contain the first buffer split
LOAD(myptrarr.0>"alpha");	set the pointer so that the first element in the array is pointed to the variable 'alpha'
VAR(bravo, 0, S32);	create the second variable to contain the second buffer split

LOAD(myptrarr.1>"bravo");	set the pointer so that the second element in the array is pointed to the variable ‘bravo’
VAR(charlie, 0.0, FLT4);	create the third variable to contain the third buffer split
LOAD(myptrarr.2>"charlie");	set the pointer so that the third element in the array is pointed to the variable ‘charlie’
VAR(delta, "", TXT);	create the fourth variable to contain the fourth buffer split
LOAD(myptrarr.3>"delta");	set the pointer so that the fourth element in the array is pointed to the variable ‘delta’
VAR(mybuf, "14:-21:3.141:Hi", TXT);	this is where the contents of the buffer is specified, a variable containing a set amount of string is created
CALC(myptrarr, mybuf, ":", "MSPLIT");	the *CALC* command to split the data buffers and this results to: alpha = 14 bravo = -21 charlie = 3.141 delta = "Hi"

Fig. 3.81 *Example to demonstrate how the CALC command and pointer arrays is used to perform multiple splits to data buffers*

If a reminder on how to create pointers or arrays is needed then refer to Chapter 3.1.5 and Chapter 3.1.6 of this guide. This method using pointer arrays may be useful when the developer requires adding text data to array elements. Using pointer arrays, testing found the time to split 64 parameters using ‘*MSPLIT*’ was 8ms compared to 64 individual ‘*SPLIT*’ which took 30 ms.

3.7. COUNTERS

There are 31 built-in I/O counters and Runtime counters that use pre-defined variables which can be reset and tested for value.

3.7.1. I/O COUNTERS

The Input/Output (I/O) counter stores values in an unsigned 32 bit register (U32) named *CNTKXX*, where XX is the I/O number for the specific counter. When using I/O counters, the I/O that is used is required to be setup to be initialised but do not require an associated INT command. The increment of the counter depends on the rising or falling edge of the interrupt i.e. every time an edge is detected by the interrupt then the counter is incremented. The command *RESET(CNTKxx)* resets the I/O counter on K00. Also the command *LOAD(CNTkxx, value)* sets a starting value for the specified I/O counter. In essence, the built-in variables (*CNTKxx)* can be treated as a normal iDev variable. The maximum counter speed is 0-10 kHz+ and is dependent on the other interrupt and entity usage. The Digital I/O interface setup is explained thoroughly in Chapter 4.6 of this guide.

I/O Counter Assignment	Pin Assignment
CNTK00	Pin number 5 K00 (CN7)
CNTK01	Pin number 6 K01 (CN7)
CNTK02	Pin number 7 K02 (CN7)
CNTK03	Pin number 8 K03 (CN7)
CNTK04	Pin number 9 K04 (CN7)
CNTK05	Pin number 10 K05 (CN7)
CNTK06	Pin number 11 K06 (CN7)
CNTK07	Pin number 12 K07 (CN7)
CNTK08	Pin number 13 K08 (CN7)
CNTK09	Pin number 14 K09 (CN7)
CNTK10	Pin number 15 K10 (CN7)
CNTK11	Pin number 16 K11 (CN7)
CNTK12	Pin number 17 K12 (CN7)
CNTK13	Pin number 18 K13 (CN7)
CNTK14	Pin number 19 K14 (CN7)
CNTK15	Pin number 20 K15 (CN7)
CNTK16	Pin number 1 K16 (CN4)
CNTK17	Pin number 2 K17 (CN4)
CNTK18	Pin number 5 K18 (CN4)
CNTK19	Pin number 6 K19 (CN4)
CNTK20	Pin number 7 K20 (CN4)
CNTK21	Pin number 8 K21 (CN4)
CNTK22	Pin number 9 K22 (CN4)
CNTK23	Pin number 10 K23 (CN4)
CNTK24	Pin number 2 K24 (CN3)
CNTK25	Pin number 3 K25 (CN3)
CNTK26	Pin number 4 K26 (CN3)
CNTK27	Pin number 6 K27 (CN3)
CNTK28	Pin number 7 K28 (CN3)
CNTK29	Pin number 9 K29 (CN3)
CNTK30	Pin number 10 K30 (CN3)

Fig. 3.82 *Table to describe which I/O pin corresponds to the specific I/O counter*

There are various ways when I/O counters are applied in an iDev project, a few usage examples are found in the table below.

IF command format to create if statements with just one action, note that a *Operand1* can be a variable and *Operand2* can be a literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function);
IF command format to create if statements with an else action, note that a *Operand1* can be a variable and *Operand2* can be a variable, literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function1:Function2);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
LOAD command format to change stored integer/ float data:
LOAD(Destination variable name, Int/float data source);

Example Usage	Definition
IF(CNTK15 > 150?chgfunc);	run function *chgfunc* if the value of I/O counter at K15 is greater than 150
IF(CNTK15 = CNTK21?togonfunc:togofffunc);	run function *togonfunc* if the value of I/O counter at K15 is the same as in K21, otherwise run the function *togofffunc*
TEXT(K04txt, CNTK04, mytxtfnt);	create a text component and set the contents of the counter specified (*CNTK04)* to be the source of the text data
TEXT(K04txt, CNTK04);;	update the text component above to the most recent value of the I/O counter *CNTK04*
LOAD(CNTK07, 2451);	change the current value of the I/O counter *CNTK07*
RESET(CNTK03);	reset the I/O counter on *K03*

Fig. 3.83 *Table demonstrating how I/O counters can be used and applied on iDev*

3.7.2. Runtime Counter

The runtime counter in iDev can be set to increment in different units depending on the name of the pre-defined variable. Similar to the I/O counter, the pre-defined runtime counter variables can be treated as normal iDev variable. The command *Reset(RUNTIME)* sets all the values of all the runtime counters to zero and starts the timer again. This runtime counter is independent of the real time clock and runs continually so no setup is required. Also it is worth remembering that the runtime counter starts counting as soon as the TFT module reaches the loading screen when turned on.

Runtime Counter	Definition	Possible Values
CNTMILLI	Increments every millisecond	0-999
CNTSECS	Increments every second	0-59
CNTMINS	Increments every minute	0-59
CNTHOURS	Increments every hour	0-23
CNTDAYS	Increments every 24 hours	0 and up (maximum is 4,294,967,295)
CNTRUN	Increments every millisecond since system reset	0 and up (86,400,000 = 1 day, maximum is 4,294,967,295)

Fig. 3.84 *Table describing different runtime counters that can be used in iDev*

Runtime counters have many applications in numerous iDev projects. A table that demonstrates how runtime counters are used is found below.

IF command format to create if statements with just one action, note that a *Operand1* can be a variable and *Operand2* can be a literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function);
IF command format to create if statements with an else action, note that a *Operand1* can be a variable and *Operand2* can be a variable, literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:
IF(Operand1 Operator Operand2?Function1:Function2);
TEXT command format with text data source from a variable:
TEXT(Text component name, Text variable, Text Style);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
LOAD command format to change stored integer/ float data:
LOAD(Destination variable name, Int/float data source);

Example Usage	Definition
IF(CNTMINS > 30?halfhourfunc);	run function *halfhourfunc* when the runtime counter *CNTMINS* exceeds 30 minutes
IF(CNTMINS = 15?turnonfunc:turnofffunc);	run function *turnonfunc* when the runtime counter reaches 15 minutes, otherwise run function *turnofffunc*
TEXT(hrstxt, CNTHOURS);;	update the text component *hrstxt* to the most recent value of the runtime counter *CNTHOURS* and then refresh the screen
LOAD(CNTSECS, 10);	change the current value of the runtime counter *CNTSECS*

Fig. 3.85 *Table demonstrating how different runtime counters can be used and applied in iDev*

Unlike the I/O counter, when using the runtime counter there is no need to set it up at all, the developer only needs to use the pre-defined variables but the runtime counter can be used with interrupts i.e. wrap-around interrupt for the runtime counter. So the developer can set functions to be called depending on the runtime counter used with the interrupt as evident in the table below. Interrupts are

properly introduced in Chapter 4.7 of this guide but setting up timer interrupts uses a similar command format.

INT command format to use as wrap-around interrupt for the runtime counter:
INT(Interrupt name, Runtime counter, Function to be called);

Runtime Counter with INT	Definition
INT(myint, CNTMILLI, mymilfunc);	call the function *mymilfunc* every 1000 milliseconds
INT(myint, CNTSECS, mysecfunc);	call the function *mysecfunc* every 60 seconds
INT(myint, CNTMINS, myminfunc);	call the function *myminfunc* every 60 minutes
INT(myint, CNTHOURS, myhrsfunc);	call the function *myhrsfunc* every 24 hours
INT(myint, CNTDAYS, mydayfunc);	call the function *mydayfunc* every 4,294,967,295 days

Fig. 3.86 *Table demonstrating how wrap-around interrupts with runtime counters are used*

3.8. TIMERS

In iDev, there are 10 available timer interrupts that can be used (TIMER0-TIMER9). Timer interrupts are mainly used when specific occurrences in a program **must** happen at a given frequency e.g. an iDev project that requires the user to input calorie intake every 4 hours will have to use timer interrupts. It is easy to confuse timers with counters, it important to remember they have similarities but both are used for different purposes. Counters in iDev uses pre-defined variables such as *CNTMILLI* and *CNTSECS* but the timers in iDev don't use pre-defined variables so you **cannot** treat *TIMER0-TIMER9* as true iDev variables i.e. you cannot use commands that are normally applicable to iDev variables to *TIMER0-TIMER9*. The ten countdown timers in iDev can be setup simultaneously, each with 1 millisecond resolution (1000ms = 1s). Interrupts are properly introduced in Chapter 4.7 of this guide but setting up timer interrupts uses a similar command format.

INT command format to setup timer interrupts, where x is the number of timer interrupt being used (TIMER0-TIMER9):
INT(Timer Interrupt name, TIMERx, Function to be called);

The developer can control different timer interrupts using the *LOAD* command.

LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value);

Timer Manipulation Usage	Definition
LOAD(myvar, TIMERx);	read the remaining time value before TIMERx expires
LOAD(TIMERx, duration);	run TIMERx once based on the duration value specified
LOAD(TIMERx, duration, repeat);	run TIMERx multiple times based on repeat value and the duration value
LOAD(TIMERx, 0);	clear and reset TIMERx

Fig. 3.87 *Table showing how timer interrupts in iDev can be manipulated*

Example Usage	Definition
LOAD(TIMER2,1000);	*TIMER2* runs once and expires after 1000 ms (1 second)
LOAD(TIMER4,500,5);	*TIMER4*'s duration is repeated 5 times with each duration at 500 ms
LOAD(TIMER6,1000,0);	*TIMER9* runs forever, expiring every 1000 ms (1 second)
LOAD(TIMER3,0);	clear and reset TIMER3
LOAD(TIMER7,time);	*TIMER7* runs once and expires after the duration value specified in the variable *time*
LOAD(myvar,TIMER4);	read the remaining time left in *TIMER4* and store it as an integer in variable *myvar*

Fig. 3.88 *Example table demonstrating how manipulation on timer interrupts is applied*

3.9. DELAY – WAIT

In all programming languages, delays are present and always used for different purposes. The equivalent of delay in iDev is the use of the *WAIT* command. The wait command has a timer accuracy of 1 ms ± 200 ns. The unit of the value specified in the delay is in milliseconds (1000 ms = 1 ms) and the maximum value that can be used is 4,294,967,295 e.g. the line WAIT(5000) placed inside a function would cause the function to be delayed by 5 seconds every time the function is called/ran.

WAIT command format:
WAIT(Duration);

If the *WAIT(duration)* command is within a function called from a *KEY* command then further key presses will be **ignored.** Each touch key press function must be processed to completion before another can be processed. The interrupts and key presses still occur during the wait period and can be processed. In essence, other serial interrupts will continue to run whilst the *WAIT* command is being processed but the associated *INT* with the *WAIT* command will not get called until after the delay has finished.

4. Interfaces and Communication

The Itron SMART TFTs have the capability to connect to an external device or module through its various interfaces. An interface provides interaction between external components to the Itron SMART tft, this increases the capability that the Itron SMART tft can achieve e.g. the lighting in a room can be controlled by an Itron SMART tft through an interface. This gives developers the ability to control specific ports and manipulate the settings to fit their desired purpose. The *SETUP* command in iDev is used to change certain settings for the different interfaces. There are a total of 7 connectors found in the Itron SMART TFT for 4.3", 5.7" and 7.0". The 3.5" has only got 5 connectors. The command syntax for setting up interfaces is consistent for all the sizes of TFT modules but this guide will focus on the 4.3" TFT module. However, if there is a difference in the command format then it will be noted in this guide.

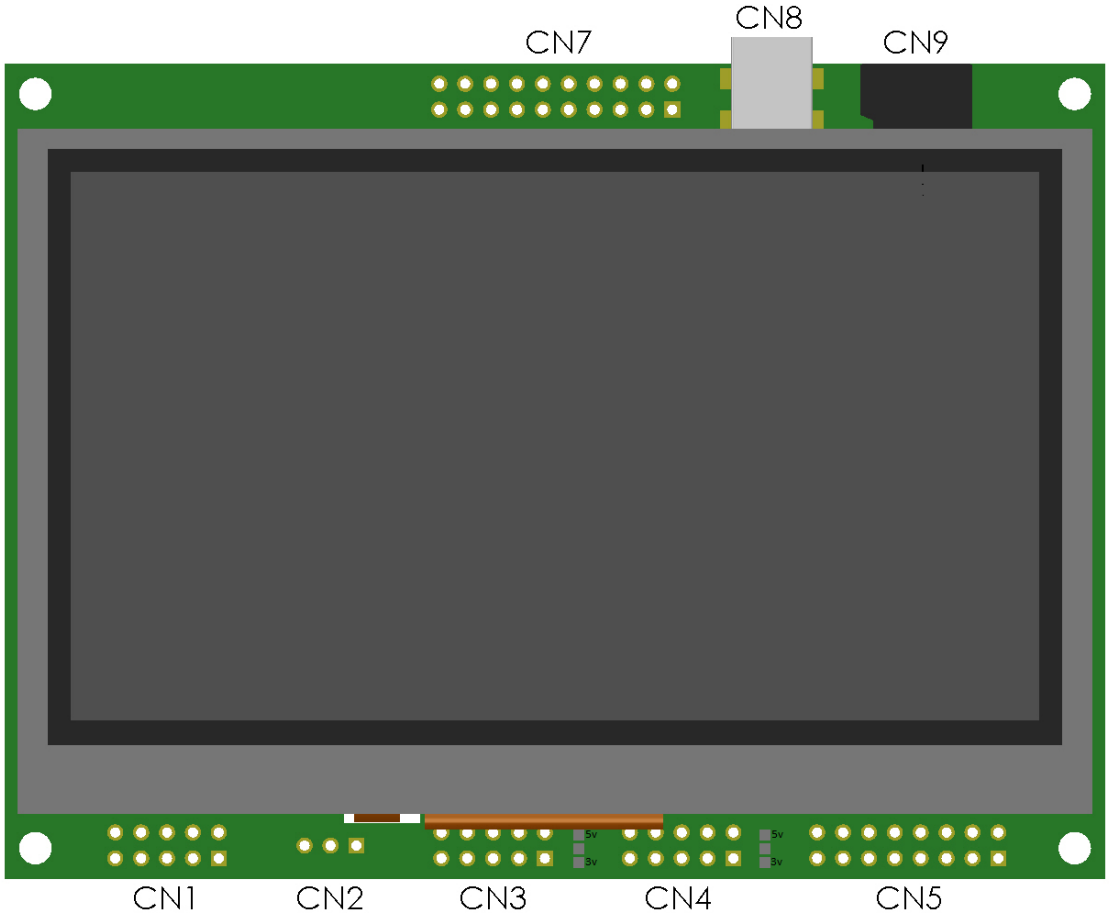


Fig. 4.1 *Picture of a 4.3" Itron SMART TFT module indicating the locations of the connectors*

4.1. RS232 INTERFACE

The RS232 interface on the itron SMART TFT modoule operates at +7V/-3V Output and +15V/-15V Input logic levels. The hardware lines RTS-CTS and DTR-DSR enable communication between host and module and are selected by jumpers on the back of the module. Only one pair can be activated at any one time (RTS-CTS or DTR-DSR). Also if the TFT module used has RS485 interface available on the module (**suffix-K611XXX**) then only RTS-CTS can be used. The location of the jumpers on the back of the module is shown below.

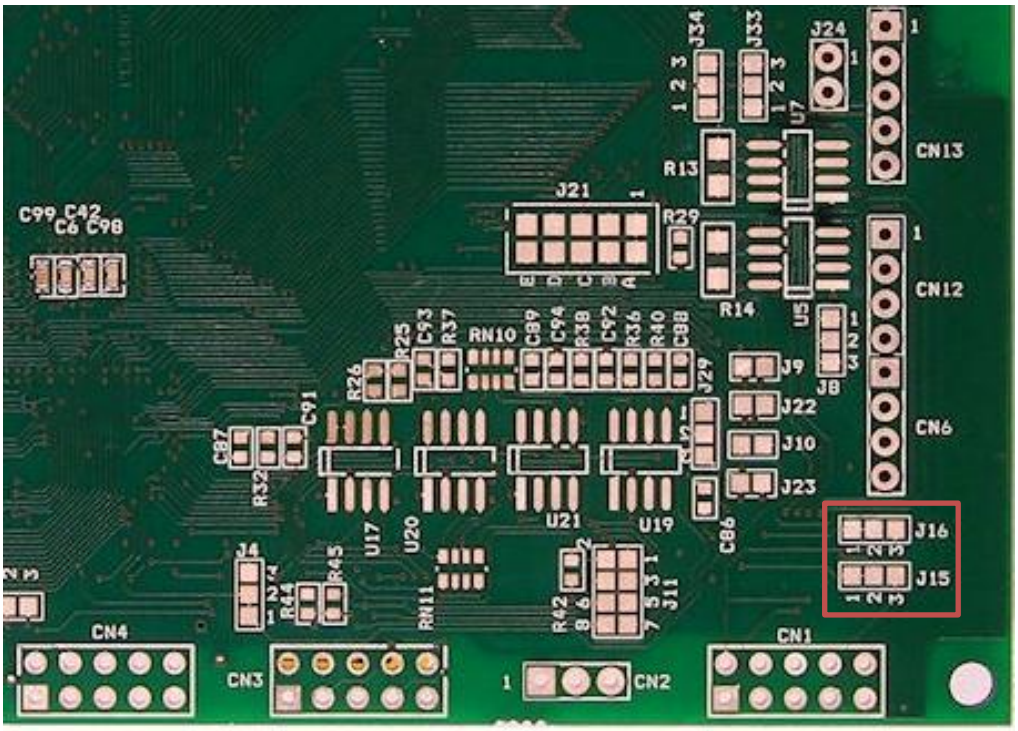


Fig. 4.2 *Picture of the back of a 4.3" TFT module indicating the location of the two jumpers for RTS-CTS or DTR-DSR hardware lines*

Note that the location of the jumpers J16 and J15 have always stayed consistent in the bottom right part on the back of the TFT module for all sizes. The location of the jumper links are highlighted in red in Fig 4.2.

Which pads to solder	J15 - RTS with RS485/DTR Jumper	J16 - CTS with RS485/DSR Jumper
Solder pad 1 and pad 2	for RTS if RS485 is fitted	for CTS if RS485 is fitted
Solder pad 2 and pad 3	for DTR if RS485 **is not** fitted	for DSR if RS485 **is not** fitted

Fig. 4.3 *Table describing how the jumpers should be soldered or linked together*

Before the RS232 interface is used, the correct pin assignments have to be connected first. There are 10 pins on for the RS232 interface of the TFT module for **all** the sizes. The TFT module is the DTE (Data Communications Equipment) device for the application and diagrams in this Chapter. The RS232 interface is located on CN1 (connector 1) of the TFT module and the pin assignments are shown on the diagram below.

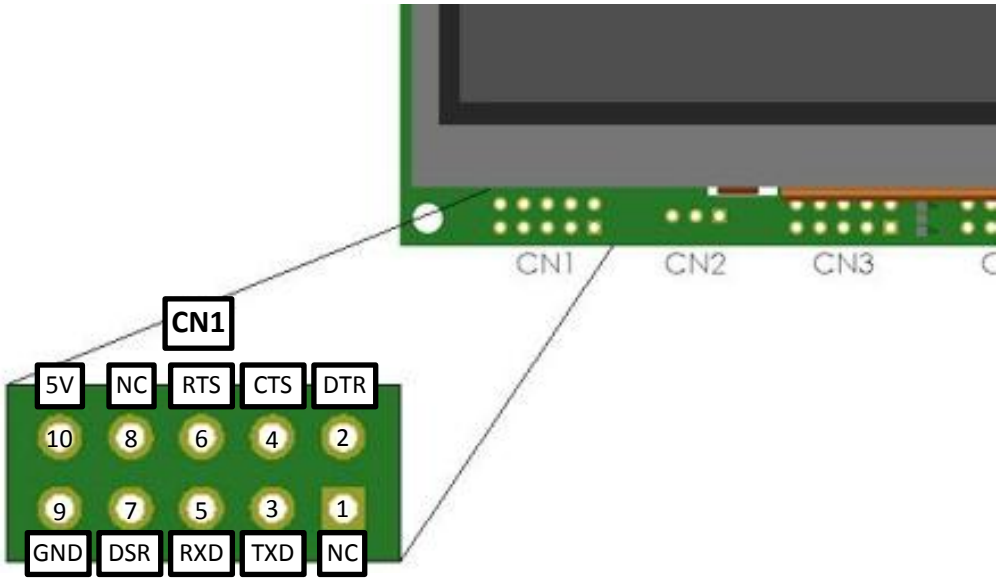


Fig. 4.4 *Diagram to show pin assignments of the RS232 interface in the TFT module*

RS232 (CN1) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	NC	Not Connected (Internally)	N/A
2	DTR	Data Terminal Ready	Output
3	TXD	Transmit Data	Output
4	CTS	Clear To Send	Input
5	RXD	Receive Data	Input
6	RTS	Request To Send	Output
7	DSR	Data Set Ready	Output
8	NC	Not Connected (Internally)	N/A
9	GND	Common Ground	Input/Output
10	5V	5V power	Input/Output

Fig. 4.5 *Table describing the pin assignments of the RS232 interface*

The Itron SMART TFT modules are powered by 5V and so any 5V pin in any connectors can act as a power source for the module and the same is applied to the GND (Common Ground) pin. However, if the TFT module is already powered by 5V then the rest of the 5V pins in the other connectors act as 5V outputs and the same with the GND (Common Ground) pins. A typical RS232 connection application between an external host module (DTE-Data Terminal Equipment) and the TFT display module (DCE-Data Communications Equipment) diagram is found below. The Itron SMART TFT is referred to as the 'Display System' in this diagram.

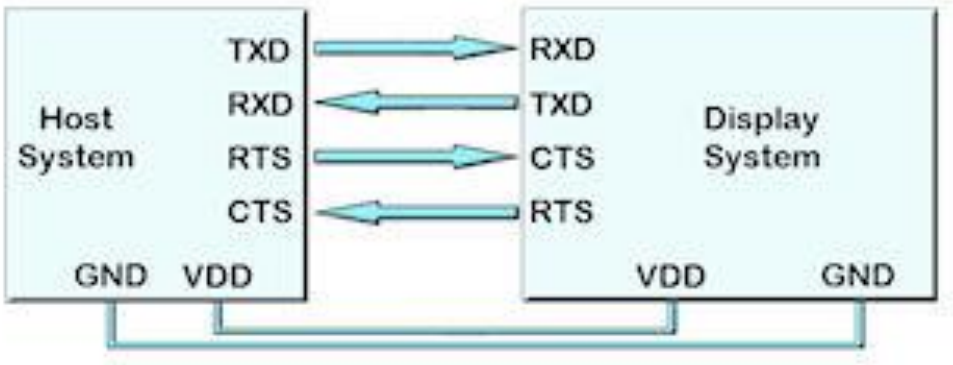


Fig. 4.6 *Diagram displaying how the RS232 interface of the TFT module is connected to a*

typical external module

It is important to remember that the diagram in Fig 4.6 is only meant to represent a general application when using the RS232 interface; some connections may be different depending on the how pins were assigned on the external module and its purpose. Once all the hardware side of the interface is finished, then the software side has to be prepared.

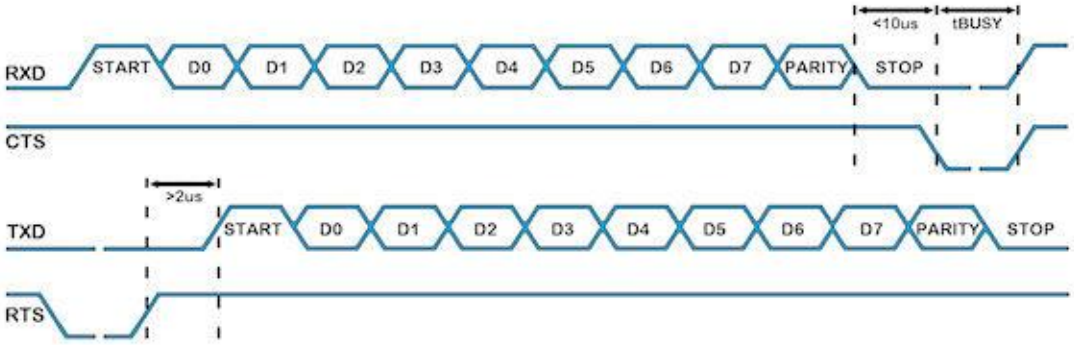


Fig. 4.7 *Diagram showing how data is sent and received through the RS232 interface in iDev*

The settings for the RS232 interface can be altered using the *SETUP* command in iDev. The *SETUP* command will be used in all the other interfaces as well. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

There is also a quick setup command format that is used for the RS232 interface in iDev. This allows the developer to change the baud rate, the parity and the communication mode of the RS232 interface using one line. Other specific parameters can also be added after the quick setup line provided that the quick setup line is the first line in the *Setup Body.*

SETUP command format for quick setup of the RS232 interface:
Setup Header
SETUP(RS232)
Setup Body
{
set = "BaudParityCommunicationMode";
}

When using the quick setup type, there are expected values for each sub-parameter. Each one is defined in the table below.

Sub-parameter	Expected Values	Definition
Baud	48	set the baud rate for the RS232 interface to 4800
	96	set the baud rate for the RS232 interface to 9600
	192	set the baud rate for the RS232 interface to 19200
	384	set the baud rate for the RS232 interface to 38400
	768	set the baud rate for the RS232 interface to 76800
	1150	set the baud rate for the RS232 interface to 115000
Parity	N (None)	remove the parity bit data sent through the RS232
	O (Odd)	set the parity bit of the data sent through RS232 to odd
	E (Even)	set the parity bit of the data sent through RS232 to even
Communication Mode	Y	enable the receive and transmit interface of the RS232 interface (i.e. rxi = Y; and txi = N;)
	N	disable the receive and transmit interface of the RS232 interface (i.e. rxi = N; and txi = N;)
	C	set the receive and transmit interface of the RS232 interface as a command processing source (i.e. rxi = C; and txi = C;)
	E	set the receive interface of the RS232 interface as a command processing source and transmit interface of the RS232 interface to echo command processing mode (i.e. rxi = C and txi = E;)
	D	set the receive interface of the RS232 interface to receive debugging data and transmit interface of the RS232 interface to echo command processing mode (i.e. rxi = D and txi = E;)

Fig. 4.8 *Table defining the sub-parameter values when using the quick setup command for the RS232 interface*

When using any interfaces such as the RS232 interface, it is important to enable the interface first by using the *SETUP* command in the TU480.mnu file. There are various setup parameters that can be altered in the RS232 interface. Similar on how the style parameters are defined, when a setup parameter is not defined in the main menu file then the default value of that particular setup parameter is assumed and used.

RS232 setup parameters		
Parameter	**Expected Values**	**Definition**
baud	110 to 6,250,000	• set the baud rate value for the RS232 interface **(default = 19200)** • any value can be set to allow trimming for deviating clocks i.e. 34850 • if unsure what baud rate is, then refer to the Glossary (Chapter 10) of this guide for further explanation
data	5	• set the number of data bits processed per data transmission through the RS232 interface to 5
	6	• set the number of data bits processed per data transmission through the RS232 interface to 6
	7	• set the number of data bits processed per data transmission through the RS232 interface to 7
	8	• set the number of data bits processed per data transmission through the RS232 interface to 8 **(default)**

stop	1	• set the number of stop bits processed per data transmission through the RS232 interface to 1 bit **(default)**
	2	• set the number of stop bits processed per data packet through the RS232 interface to 2 bits
	15	• set the number of stop bits processed per data packet through the RS232 interface to **1.5** bits
parity	O (Odd)	• set the parity bit of the data sent through RS232 to odd • if unsure what parity is refer to the glossary of this guide
	E (Even)	• set the parity bit of the data sent through RS232 to even
	N (None)	• remove the parity bit data sent through the RS232 **(default)**
	M (Mark)	• set the parity bit of the data sent through RS232 to a mark??
	S (Space)	• set the parity bit of the data sent through RS232 to a space??
rxi	Y	• enable the receive interface of the RS232 interface
	N	• disable the receive interface of the RS232 interface **(default)**
	C	• set the receive interface of the RS232 interface as a command processing source
	D	• set the receive interface of the RS232 interface to receive debugging data
proc	• termination characters can be specified to generate an interrupt to process a string of data • NB when sending commands (in command mode where rxi = C) to the module, processing only occurs when \\0D or 0D hex is received e.g. TEXT(mytext, “hello world”);; \\0D	
	all;	• trigger on all received characters **(default)**
	CRLF;	• trigger on a carriage return (CR) followed by line feed (LF = 0Dh 0A)
	CR;	• trigger on carriage return (CR = 0dH) in command mode where rxi = C;
	LF;	• trigger on line feed (LF = 0Ah)
	NUL;	• trigger on NUL (00h)
	\\xx;	• trigger on xxh (specify hex value)
	"ABCD";	• string in format defined by SYSTEM encode parameter
	"\\xx\\xx";	• string in format defined by SYSTEM encode parameter
procDel	Y or N	• keep (Y) or remove (N) the termination character(s) before processing **(default = N)**
procNum	0 to 16,780,000	• interrupt on n bytes received as alternative to proc and procDel parameters **(default = 0)**
rxb	1 to 16,780,000	• specify the size of the receive buffer in bytes **(default = 8192 bytes)**
txi	Y	• enable the transmit interface of the RS232 interface
	N	• disable the transmit interface of the RS232 interface **(default)**
	C	• set the transmit interface of the RS232 interface as a command processing source
	E	• set the transmit interface of the RS232 interface to echo command processing mode
txb	1 to 16,780,000	• specify the size of transmit buffer in bytes **(default = 8192 bytes)**
encode	• set the data encode mode to suit the purpose of the RS232 interface	
	s	• set data encode to 8 bit ASCII data • codes 00-1F and 80-FF are converted to ASCII "\\00" - "\\1D" and "\\80" - "\\FF" respectively **(default)**

	sr	• set data encode to 8 bit ASCII raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as ASCII+ data
	sd	• set data encode to 8 bit ASCII raw data bytes • all bytes are processed as raw data
	w	• set data encode to UNICODE – HEX Char x 4 = U16 (Most significant hex-pair first) "ABCD" -> \\ABCD
	wr	• set data encode to 8 bit UNICODE raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UNICODE+ data??
	wd	• set data encode to 8 bit UNICODE raw data bytes • all bytes are processed as raw data
	m	• set data encode to 8 bit UTF8 raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UTF8+ data??
	mr	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	md	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	D8M	• set data encode to 8 bit data with U16's, U32's etc output **most** significant byte first • the same as data encode sd
	D8L	• set data encode to 8 bit data with U16's, U32's etc output **least** significant byte first
	D16M	• set data encode to 16 bit data with bytes processed as **most** significant byte first • interrupt occurs after two bytes • the same as data encode wd
	D16L	• set data encode to 16 bit data with bytes processed as **least** significant byte first • interrupt occurs after two bytes
	D32M	• set data encode to 32 bit data with bytes processed as **most** significant byte first • interrupt occurs after four bytes • the same as data encode md
	D32L	• set data encode to 32 bit data with bytes processed as **least** significant byte first • interrupt occurs after four bytes
	sh or h8m or h8l	• set data encode to ASCII – HEX Char x 2 = U8 e.g. "A8" -> \\A8
	h16m	• set data encode to ASCII – HEX Char x 4 = U16 (**most** significant hex-pair first) e.g. "ABCD" -> \\ABCD
	h16l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) e.g. "ABCD" -> \\CDAB
	h32m	• set data encode to ASCII – HEX Char x 4 = U32 (**most** significant hex-pair first) "12345678" -> \\12345678
	h32l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) "12345678" -> \\78654321
flow	N	• set the handshaking to NONE **(default)**
	H	• set the flow for handshaking to hardware RTS/CTS or DTR/DSR
	S	• set the flow for handshaking to software XON/XOFF

Fig. 4.9 *Table defining the RS232 interface setup parameters*

The value of the *Interface* parameter has to be changed to RS2 when setting up the RS232 interface in iDev as seen in the example.

```
//FILENAME: TU480a.mnu

SETUP(RS2)      //
{
baud = 34850;   //
data = 7;       //
stop = 2;       //
parity = N;     //
rxi = C;        //
proc = all;     //
procDel = Y;    //
procNum = 5;    //
rxb = 5250;     //
txi = E;        //
txb = 5250;     //
encode = sr;    //
flow = N;       //
}

SETUP(RS2)         //or a quick setup combination

{
set = "768NC";     //
}

SETUP(RS2)        //or a mixture of both setup types
{
set = "768NC";    //
data = 5;         //
proc = all;       //
encode = mr;      //
}
```

Fig. 4.10 *Example code showing how the RS232 interface is setup is done in iDev*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated. It is possible to use this method to change the *set* parameter for a quick setup interface but it is not recommended, as this would cause a lot of grief to the developer to set the sub parameter values.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updrs2func)        //
{
LOAD(RS2.proc,"CRLF";   //
LOAD(RS2.flow,"H";      //
LOAD(RS2.encode,"m";    //
}
```

Fig. 4.11 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for the RS232 Interface*

Due to the complexity, the required hardware changes and peripheral using the RS232 interface it would not be possible to create a 'basic' example project that can be included in this guide. For an example project that uses the RS232 interface, download the example

project from the website (link here) named 'RS232 Raw Data and Pointer Demonstration Project'. There are new iDev commands in this example project that are used in manipulating interfaces such as interrupts. The *INT* (for Interrupts*)* command is introduced and explained in Chapter 4.7 of this guide and processing data is explained in Chapter 4.8.

4.2. RS422/RS485 INTERFACE

The RS422/485 interface is only available on the TFT module with suffix-**K611XXX**. The communication of the RS422/RS485 interface in iDev can be set to either Full Duplex(RS422) or Half Duplex(RS485) (If unsure on what Full Duplex or Half Duplex is, refer to the Glossary in Chapter 10 of this guide). This is achieved by soldering certain pins on the back of the module in the jumper highlighted below.

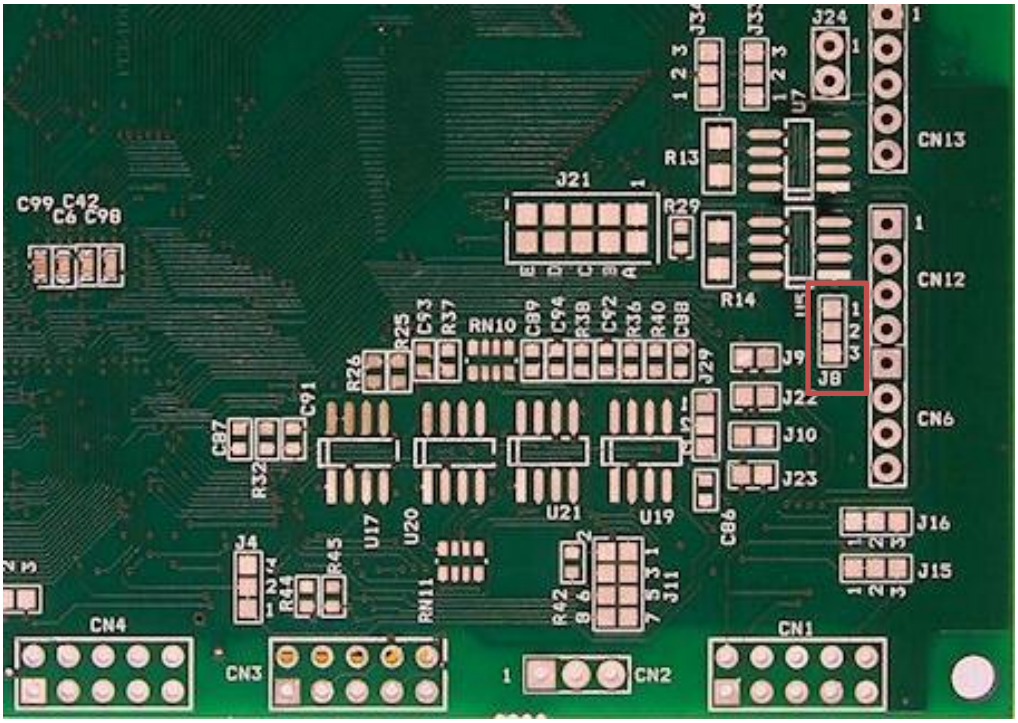


Fig. 4.12 *Picture of the back of a 4.3" TFT module indicating the location of the jumper to set the communication mode of the RS485 interface*

The location of the jumper J8 varies in different PCB boards but most of them are located in the bottom right part on the back of the module as shown above. The location of the jumper link J8 is highlighted in RED in the diagram in Fig 4.12. For **Full Duplex (RS422)** communication mode, solder **pin 1** and **pin 2** together and for **Half Duplex (RS485)** communication mode, solder **pin 2** and **pin 3** together in jumper J8. The default communication mode is **Full Duplex** so pin 1 and pin 2 are already linked or connected on all the TFT modules. If the developer requires the use of **Half Duplex** mode then the solder links between pin 1 and 2 should be removed first before soldering pin 2 and pin 3 together. This arrangement of jumper pins applies to all the sizes of TFT. If unsure about the definition of full duplex or half duplex then refer to the Glossary in Chapter 10 of this guide.

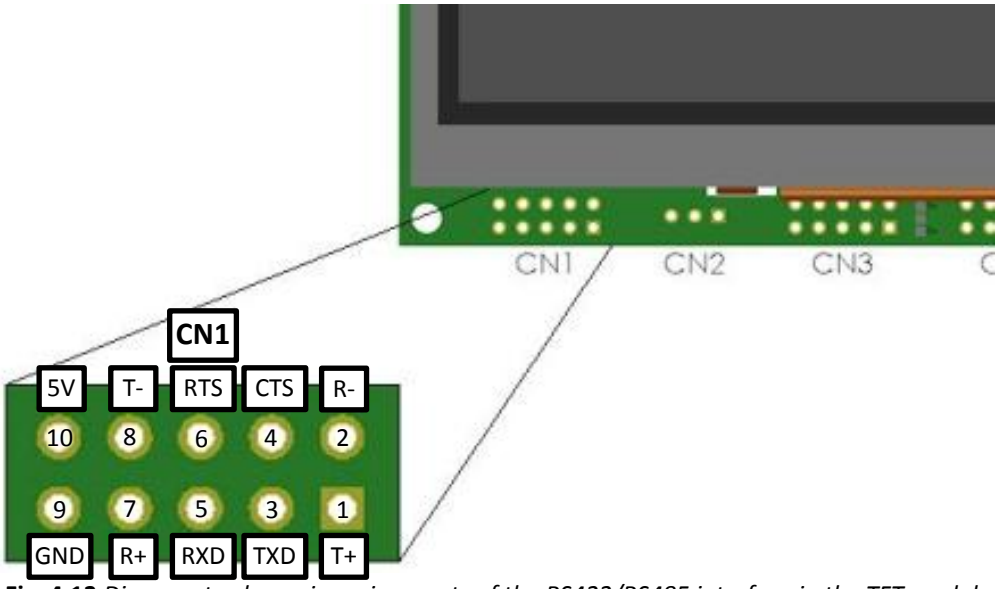


Fig. 4.13 *Diagram to show pin assignments of the RS422/RS485 interface in the TFT module*

RS422/RS485 (CN1) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	T+	Non-Inverting Transmit Data for Half Duplex communication mode	Output
2	R-	Inverting Receive Data	Input
3	TXD	Transmit Data	Output
4	CTS	Clear To Send	Input
5	RXD	Receive Data	Input
6	RTS	Request To Send	Output
7	R+	Non-Inverting Receive Data for Half Duplex communication mode	Input
8	T-	Inverting Receiver Data	Output
9	GND	Common Ground	Input/Output
10	5V	5V power	Input/Output

Fig. 4.14 *Table describing the pin assignments of the RS422/RS485 interface*

The Itron SMART TFT modules are powered by 5V and so any 5V pin in any connectors can act as a power source for the module and the same is applied to the GND (Common Ground) pin. However, if the TFT module is already powered by 5V then the rest of the 5V pins in the other connectors act as 5V outputs and the same with the GND (Common Ground) pins. A typical RS422/RS485 connection application between an external host module (DTE-Data Terminal Equipment) and the TFT display module (DCE-Data Communications Equipment) diagram is found in Fig 4.15 and Fig 4.12. The Itron SMART TFT is referred to as the 'Display System' in the diagrams.

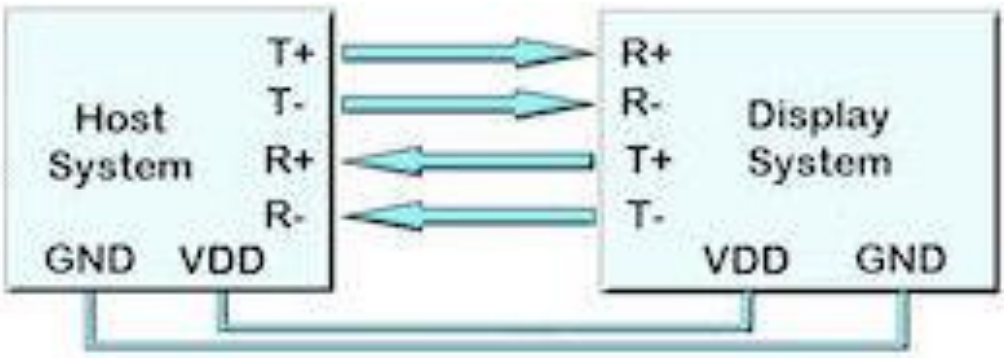


Fig. 4.15 *Diagram displaying how the RS422 interface of the TFT module is connected to a typical external module*

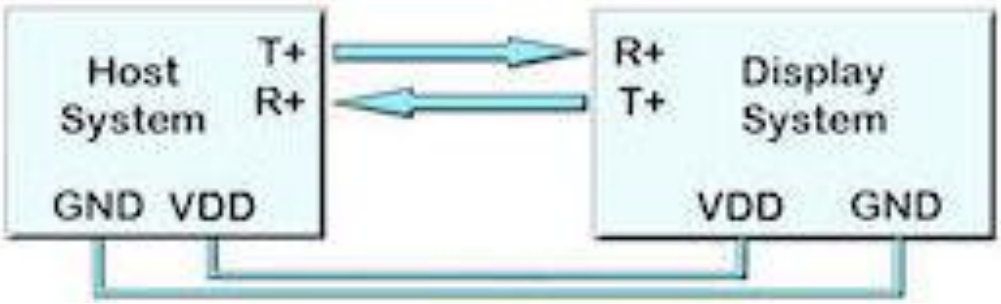


Fig. 4.16 *Diagram displaying how the RS485 interface of the TFT module is connected to a typical external module*

It is important to remember that the diagrams in Fig 4.15 and 4.16 is only meant to represent a general application when using the RS422/RS485 interface; some connections may be different depending on the how pins were assigned on the external module and its purpose. Once all the hardware side of the interface is finished, then the software side has to be prepared. The settings for the RS422/RS485 interface can be altered using the *SETUP* command in iDev. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

Similar to the RS232 interface, there is also a quick setup command format that is used for the RS485 interface in iDev. This allows the developer to change the baud rate, the parity and the communication mode of the RS422/RS485 interface using one line. Other specific parameters can also be added after the quick setup line provided that the quick setup line is the first line in the *Setup Body.*

SETUP command format for quick setup of the RS422/RS485 interface:
Setup Header
SETUP(RS4)
Setup Body
{
set = "BaudParityCommunicationMode";
}

When using the quick setup type, there are expected values for each sub-parameter. Each one is defined in the table below.

Sub-parameter	Expected Values	Definition
Baud	48	set the baud rate for the RS422/RS485 interface to 4800
	96	set the baud rate for the RS422/RS485 interface to 9600
	192	set the baud rate for the RS422/RS485 interface to 19200
	384	set the baud rate for the RS422/RS485 interface to 38400
	768	set the baud rate for the RS422/RS485 interface to 76800
	1150	set the baud rate for the RS422/RS485 interface to 115000
Parity	N (None)	remove the parity bit data sent through the RS422/RS485 **(default)**
	O (Odd)	set the parity bit of the data sent through RS422/RS485 to odd
	E (Even)	set the parity bit of the data sent through RS422/RS485 to even
Communication Mode	Y	enable the receive and transmit interface of the RS422/RS485 interface (i.e. rxi = Y; and txi = N;)
	N	disable the receive and transmit interface of the RS422/RS485 interface (i.e. rxi = N; and txi = N;)
	C	set the receive and transmit interface of the RS422/RS485 interface as a command processing source (i.e. rxi = C; and txi = C;)
	E	set the receive interface of the RS422/RS485 interface as a command processing source and transmit interface of the RS422/RS485 interface to echo command processing mode (i.e. rxi = C and txi = E;)
	D	set the receive interface of the RS422/RS485 interface to receive debugging data and transmit interface of the RS422/RS485 interface to echo command processing mode (i.e. rxi = D and txi = E;)

Fig. 4.16 *Table defining the sub-parameter values when using the quick setup command for the RS422/RS485 interface*

When using any interfaces such as the RS422/RS485 interface, it is important to enable the interface first by using the *SETUP* command in the TU480.mnu file. Similar on how the style parameters are defined, when a setup parameter is not defined in the main menu file, then the default value of that particular setup parameter is assumed and used.

RS422/RS485 setup parameters		
Parameter	**Expected Values**	**Definition**
baud	110 to 6,250,000	• set the baud rate value for the RS422/RS485 interface **(default = 19200)** • any value can be set to allow trimming for deviating clocks i.e. 34850 • if unsure what baud rate is, then refer to the Glossary (Chapter 10) of this guide for further explanation
data	5	• set the number of data bits processed per data transmission through the RS422/RS485 interface to 5
	6	• set the number of data bits processed per data transmission through the RS422/RS485 interface to 6
	7	• set the number of data bits processed per data transmission through the RS422/RS485 interface to 7
	8	• set the number of data bits processed per data transmission through the RS422/RS485 interface to 8 **(default)**
stop	1	• set the number of stop bits processed per data transmission through the RS422/RS485 interface to 1 bit **(default)**
	2	• set the number of stop bits processed per data packet through the RS422/RS485 interface to 2 bits
	15	• set the number of stop bits processed per data packet through the RS422/RS485 interface to **1.5** bits
parity	O (Odd)	• set the parity bit of the data sent through RS422/RS485 to odd • if unsure what parity is refer to the glossary of this guide
	E (Even)	• set the parity bit of the data sent through RS422/RS485 to even
	N (None)	• remove the parity bit data sent through the RS422/RS485 **(default)**
	M (Mark)	• set the parity bit of the data sent through RS422/RS485 to a mark??
	S (Space)	• set the parity bit of the data sent through RS422/RS485 to a space??
rxi	Y	• enable the receive interface of the RS422/RS485 interface
	N	• disable the receive interface of the RS422/RS485 interface **(default)**
	C	• set the receive interface of the RS422/RS485 interface as a command processing source
	D	• set the receive interface of the RS422/RS485 interface to receive debugging data
proc	• termination characters can be specified to generate an interrupt to process a string of data • NB when sending commands (in command mode where rxi = C) to the module, processing only occurs when \\0D or 0D hex is received e.g. TEXT(mytext, "hello world");; \\0D	
	all;	• trigger on all received characters **(default)**
	CRLF;	• trigger on a carriage return (CR) followed by line feed (LF = 0Dh 0A)
	CR;	• trigger on carriage return (CR = 0dH) in command mode where rxi = C;
	LF;	• trigger on line feed (LF = 0Ah)
	NUL;	• trigger on NUL (00h)
	\\xx;	• trigger on xxh (specify hex value)
	"ABCD";	• string in format defined by SYSTEM encode parameter
	"\\xx\\xx";	• string in format defined by SYSTEM encode parameter

procDel	Y or N	• keep (Y) or remove (N) the termination character(s) before processing **(default = N)**
procNum	0 to 16,780,000	• interrupt on n bytes received as alternative to proc and procDel parameters **(default = 0)**
rxb	1 to 16,780,000	• specify the size of the receive buffer in bytes **(default = 8192 bytes)**
txi	Y	• enable the transmit interface of the RS422/RS485 interface
	N	• disable the transmit interface of the RS422/RS485 interface **(default)**
	C	• set the transmit interface of the RS422/RS485 interface as a command processing source
	E	• set the transmit interface of the RS422/RS485 interface to echo command processing mode
txb	1 to 16,780,000	• specify the size of transmit buffer in bytes **(default = 8192 bytes)**
encode	• set the data encode mode to suit the purpose of the RS485 interface	
	s	• set data encode to 8 bit ASCII data • codes 00-1F and 80-FF are converted to ASCII "\\00" - "\\1D" and "\\80" - "\\FF" respectively **(default)**
	sr	• set data encode to 8 bit ASCII raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as ASCII+ data
	sd	• set data encode to 8 bit ASCII raw data bytes • all bytes are processed as raw data
	w	• set data encode to UNICODE – HEX Char x 4 = U16 (Most significant hex-pair first) "ABCD" -> \\ABCD
	wr	• set data encode to 8 bit UNICODE raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UNICODE+ data??
	wd	• set data encode to 8 bit UNICODE raw data bytes • all bytes are processed as raw data
	m	• set data encode to 8 bit UTF8 raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UTF8+ data??
	mr	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	md	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	D8M	• set data encode to 8 bit data with U16's, U32's etc output **most** significant byte first • the same as data encode sd
	D8L	• set data encode to 8 bit data with U16's, U32's etc output **least** significant byte first
	D16M	• set data encode to 16 bit data with bytes processed as **most** significant byte first • interrupt occurs after two bytes • the same as data encode wd
	D16L	• set data encode to 16 bit data with bytes processed as **least** significant byte first • interrupt occurs after two bytes

	D32M	• set data encode to 32 bit data with bytes processed as **most** significant byte first • interrupt occurs after four bytes • the same as data encode md
	D32L	• set data encode to 32 bit data with bytes processed as **least** significant byte first • interrupt occurs after four bytes
	sh or h8m or h8l	• set data encode to ASCII – HEX Char x 2 = U8 e.g. "A8" -> \\A8
	h16m	• set data encode to ASCII – HEX Char x 4 = U16 (**most** significant hex-pair first) e.g. "ABCD" -> \\ABCD
	h16l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) e.g. "ABCD" -> \\CDAB
	h32m	• set data encode to ASCII – HEX Char x 4 = U32 (**most** significant hex-pair first) "12345678" -> \\12345678
	h32l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) "12345678" -> \\78654321
flow	N	• set the handshaking to NONE **(default)**
	H	• set the flow for handshaking to hardware RTS/CTS or DTR/DSR
	S	• set the flow for handshaking to software XON/XOFF
duplex	H (Half Duplex)	• set the communication mode to Half Duplex (RS485) • Half Duplex uses connector CN1(Connector 1) with pins 1 and 8
	F(Full Duplex)	• set the communication mode to Full Duplex (RS422) **(default)**

Fig. 4.17 *Table defining the RS422/RS485 interface setup parameters*

```
//FILENAME: TU480a.mnu

SETUP(RS4)          //
{
baud = 34850;       //
data = 7;           //
stop = 2;           //
parity = N;         //
rxi = C;            //
proc = all;         //
procDel = Y;        //
procNum = 5;        //
rxb = 5250;         //
txi = E;            //
txb = 5250;         //
encode = sr;        //
flow = N;           //
duplex = F;         //
}

SETUP(RS4)          //or a quick setup combination
{
set = "768NC";      //
}

SETUP(RS4)          //or a mixture of both setup types
{
set = "768NC";      //
data = 5;           //
proc = all;         //
encode = mr;        //
}
```

Fig. 4.18 *Example code showing how the RS422/RS485 interface setup is done in iDev*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated. It is possible to use this method to change the set parameter for a quick setup interface but it is not recommended, as this would cause a lot of grief to the developer to set the sub parameter values.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updrs4func)        //
{
LOAD(RS4.proc,"CRLF";   //
LOAD(RS4.flow,"H";      //
LOAD(RS4.encode,"m";    //
}
```

Fig. 4.19 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for the RS422/RS485 Interface*

Unfortunately there is no example project that uses the RS422/RS485 interface from the Itron TFT website but the RS232 and RS422/RS485 interface are really similar and so the example project that uses the RS232 interface can be altered as a template or a starting point for a beginner developer. The example project can be downloaded from the website (link here) named 'RS232 Raw Data and Pointer Demonstration Project'. There are new iDev commands in this example project that are used in manipulating interfaces such as interrupts. The *INT* (for Interrupts*)* command is introduced and explained in Chapter 4.7 of this guide and processing data is explained in Chapter 4.8.

4.3. CMOS ASYNCHRONOUS INTERFACE (AS1, AS2, DBG)

There are three CMOS Asynchronous interfaces available in iDev, namely AS1, AS2 and DBG (debugging). All of which operate at 3.3V logic level (definition in Chapter 10) but the AS1 has the option to work at both 3.3V and 5V logic levels. The AS1 interface is located in connector 3 of the module as shown below.

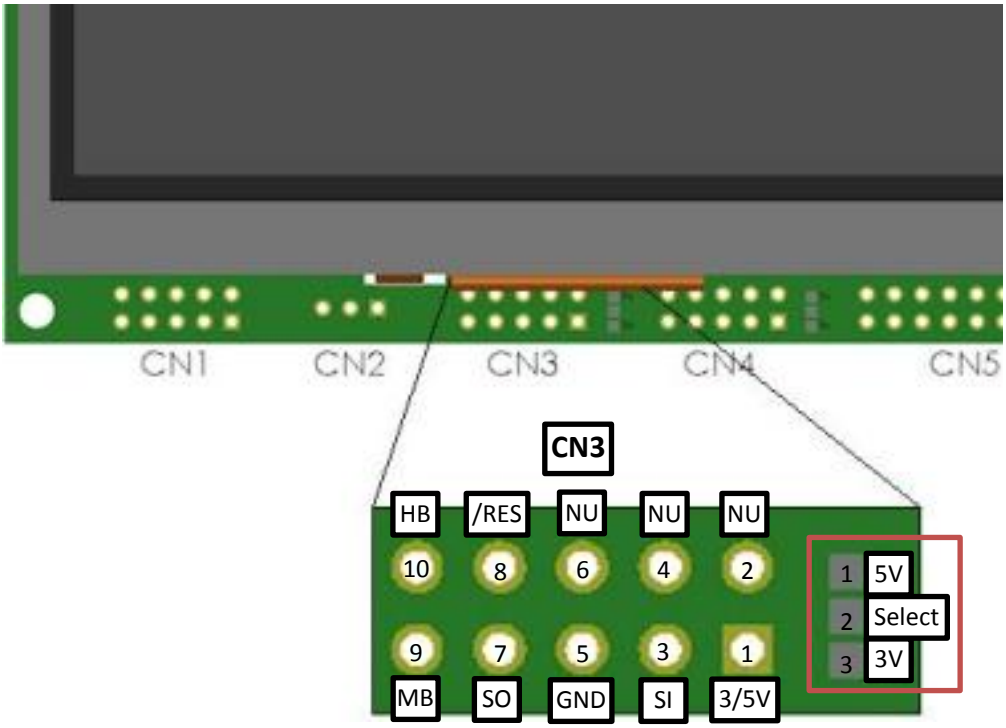


Fig. 4.20 *Diagram to show pin assignments of the AS1 interface in the TFT module with the jumper link highlighted*

The jumper link highlighted in RED, allows the developer to select the voltage output level for pin 1 (3V/5V) in connector 3(**CN3**). The developer must either link or solder together the **Select pad** (pad 2) of the jumper to **5V pad** if 5V is required to come out of pin 1(in **CN3**) or the **3V pad** to the **Select pad** (pad 2 of the jumper) for 3.3V output. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions and sizes do not have this jumper link. This means that the voltage level input/output at pin 1 in connector 3(**CN3**) is fixed at 5V. The default logic level for the AS1 interface that all TFT sizes come with is 3.3V. If however, the model number suffix of the TFT module is **K6XXXXXXXA** then this means that the AS1 interface logic level of that particular TFT module is 5V. The Itron SMART TFT modules are powered by 5V and so any 5V pin in any connectors can act as a power source for the module and the same is applied to the GND (Common Ground) pin. However, if the TFT module is already powered by 5V then the rest of the 5V pins in the other connectors act as 5V outputs and the same is applied with the GND (Common Ground) pins; an external module that is connected to the TFT module can either be powered by 3.3V or 5V from the module in pin 1.

AS1 (CN3) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	3/5V	3.3V (Output) or 5V (Input or Output) power	Input/Output
2	NU	Not Used (Do not connect to anything)	Input
3	SI	Serial In	Input
4	NU	Not Used (Do not connect to anything)	Input
5	GND	Common Ground	Input/Output
6	NU	Not Used (Do not connect to anything)	N/A
7	SO	Serial Out	Output
8	/RES	Reset Pin (Active Low)	Input
9	MB	Module Busy	Input
10	HB	Host Busy	Output
Jumper Link (Highlighted in RED) for AS1 in CN3			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 1 in **CN3**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3.3V output at pin 1 in **CN3**

Fig. 4.21 *Table describing the pin and pad assignments of the AS1 interface*

There is another Asynchronous Interface called AS2 available on the Itron SMART TFT module but the pins for the AS2 are found in connector 7 (**CN7**) and connector 4 (**CN4**).

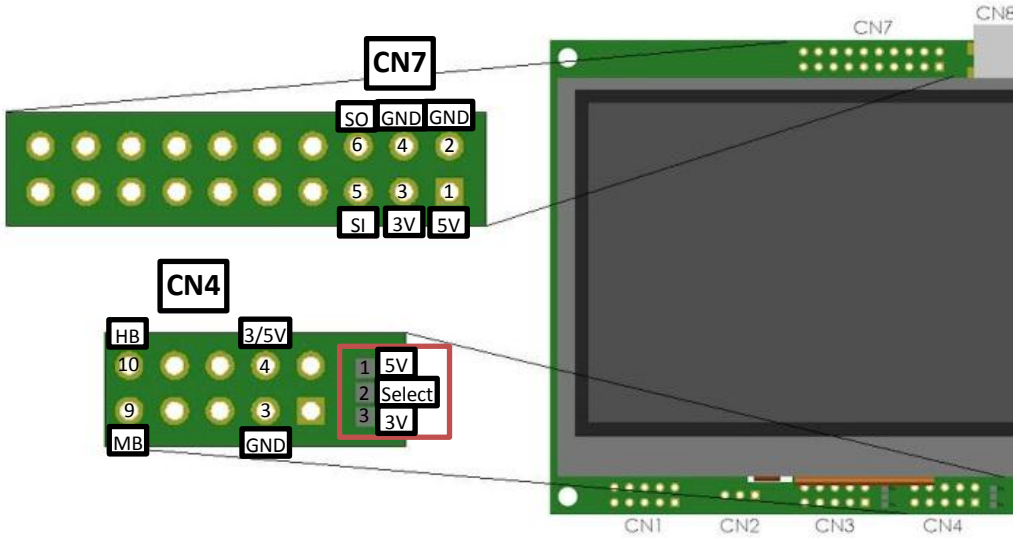


Fig. 4.22 *Diagram to show pin assignments of the AS2 interface in the TFT module with the jumper link highlighted*

Similar to the connector layout for AS1, the AS2 also has a jumper link to set the voltage output levels on pin 4 in connector 4 (**CN4**) to either 5V or 3.3V. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions do not have this jumper link have the voltage level input/output at pin 4 in connector 4(**CN4**) fixed at 5V. Unlike the AS1 interface, the AS2 interface has a **fixed logic level** operating at **3.3V**.

AS2 (CN7) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	5V	5V power	Input/Output
2	GND	Common Ground	Input/Output
3	3V	3.3V (Output) power	Output
4	GND	Common Ground	Input/Output
5	SI	Serial In	Output
6	SO	Serial Out	Input
AS2 (CN4) Pin Assignment Definition			
3	GND	Common Ground	Input/Output
4	3/5V	3.3V (Output) or 5V (Input or Output) power	Input/Output
9	HB	Host Busy	Input
10	MB	Module Busy	Output
Jumper Link (Highlighted in RED) for AS2 in CN7 and CN4			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 4 in **CN4**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3V output at pin 4 in **CN4**

Fig. 4.23 *Table describing the pin and pad assignments of the AS2 interface*

The AS2 interface does not have an allocated connector set of pins to it so unfortunately the pin assignments are found in connector 4(**CN4**) and connector 7(**CN7**) as seen in Fig. 4.23. Having a separate connector set however, enables the developer to use the AS1 and AS2 interfaces simultaneously. As mentioned before, the TFT module is powered by 5V and so pin 4 in connector 4(**CN4**) can also act as an input to power the TFT module or an output to power an external module. The Itron SMART TFT is referred to as the 'Display System' in the diagram below.

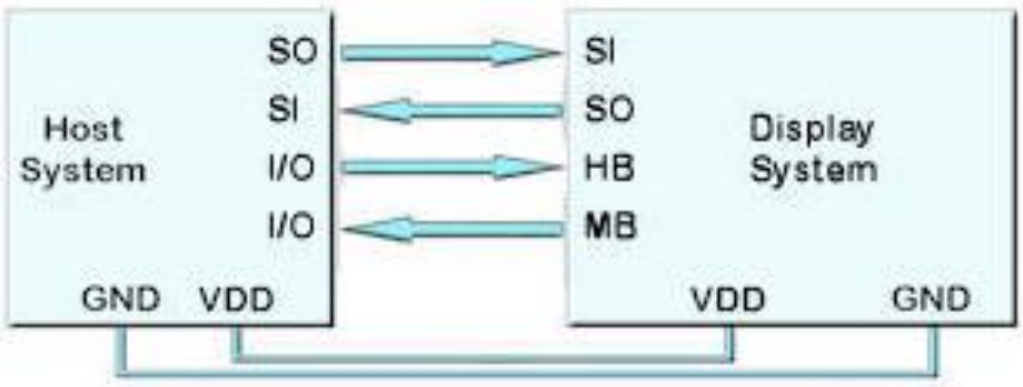


Fig. 4.24 *Diagram displaying how the AS1/AS2 interface of the TFT module is connected to a typical external module*

The diagram in Fig. 4.24 represents a general application when using the AS1 and AS2 interface; some connections may be different depending on how the pins were assigned on the external module and its purpose. The host busy line (HB) stops the module from sending data to the host. (More Module busy and Host busy explanation here). The use of the HB and MB busy lines are optional and can be connected together if not required. It is important to remember that the AS1 and AS2 interface can be used simultaneously due to the separate

locations of the pin connectors. The Itron SMART TFT module has an interface allocated for debugging purposes that solely consist of receive data and transmit data lines. The developer can use this interface to check if the TFT module is correctly communication with an external module connected to it just using the two basic transmission lines. The connector 6 (**CN6**) for the DBG (debug) interface is located in the bottom right part on the back of the Itron SMART TFT module.

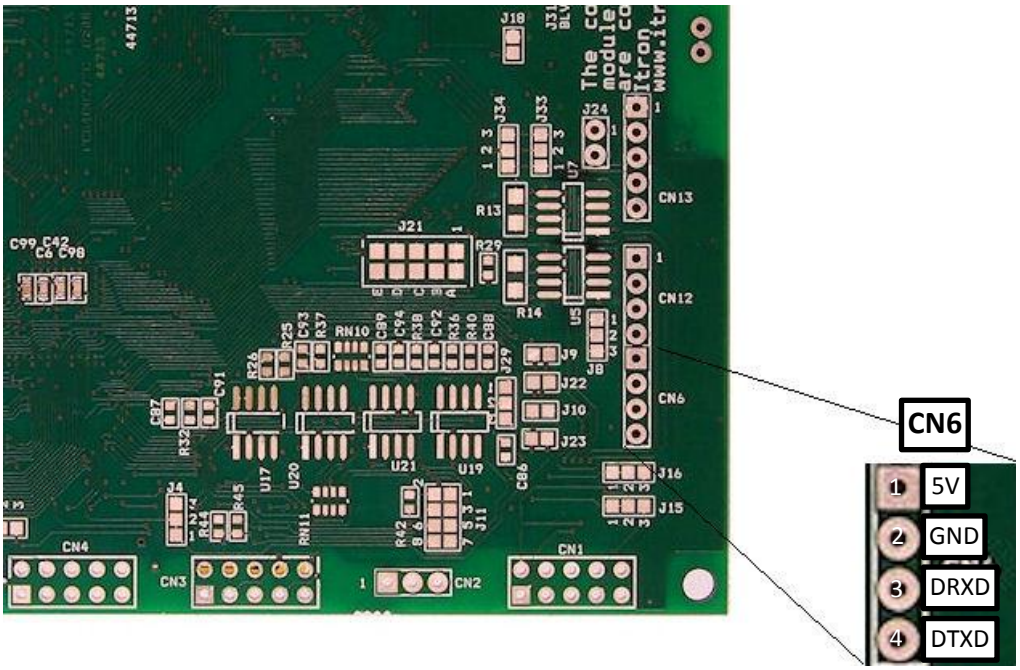


Fig. 4.25 *Diagram to show pin assignments of the DBG interface on the back of the TFT module*

DBG (CN6-back) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	5V	5V power	Input/Output
2	GND	Common Ground	Input/Output
3	DRXD	Debug Receive Data	Input
4	DTXD	Debug Transmit Data	Output

Fig. 4.26 *Table describing the pin assignments of the DBG interface*

The Itron SMART TFT is referred to as the 'Display System' in the diagram below.

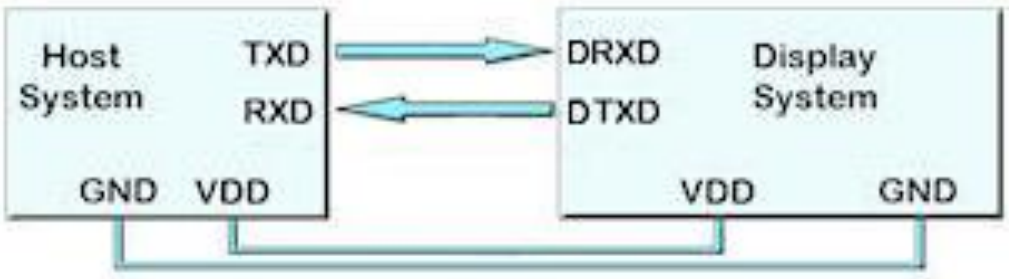


Fig. 4.27 *Diagram displaying how the DBG interface of the TFT module is connected to a typical external module*

When the module is powered on, the micro SD card slot and the NAND memory are scanned for the main menu file (TUXXXa.mnu). If the main menu file and the project files are located in NAND, then the project would always run if there is no micro SD card slot present. The only way to overwrite or control what is stored in NAND is by uploading the main menu file via micro SC card. However, if there is no micro SD card available, the code cannot be changed unless some of the interfaces are enabled to allow data transfer. In iDev, it is possible to initialise some interfaces of the TFT module by applying a temporary link to pin 3 (DRXD) and pin 4 (DTXD) in **CN6** of the module. The link has to be applied before the module is powered on for the initialisation of the interfaces to occur successfully. The interfaces: RS232, AS1, I2C and USB are enabled and the table below describes their parameters when the link is detected. The other parameters that are not specified use the default values.

RS232 setup parameters (see Chapter 4.1)	
Parameter	**Value**
baud	115200
data	8
stop	1
parity	N
flow	H
AS1 setup parameters (see Chapter 4.3)	
Parameter	**Value**
baud	500000
data	8
stop	1
parity	N
flow	H
I2C setup parameters (see Chapter 4.5)	
Parameter	**Value**
addr	\\6E
USB setup parameters (see Chapter 8.5)	
Parameter	**Value**
rxi	C
txi	Y

Fig 4.28 *Table describing the setup parameters of the four interfaces enabled*

Once all the hardware side of the interface is finished, then the software side has to be prepared. To give a better picture on how the AS1/AS2 interface on iDev works, this is a diagram showing how data is processed in the AS1/AS2 interface.

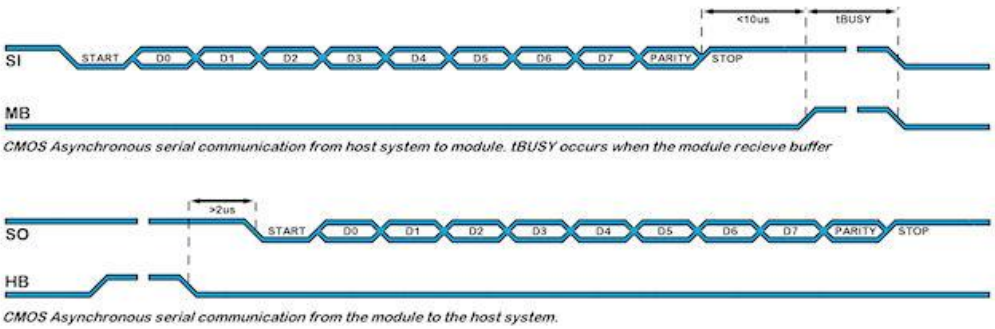


Fig. 4.29 *Diagram showing how data is sent and received through the AS1/AS2 interface in iDev*

The settings for the AS1/AS2/DBG interface can be altered using the *SETUP* command in iDev. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

Similar to the RS232 interface, there is also a quick setup command format that is used for the AS1/AS2 interface in iDev. This allows the developer to change the baud rate, the parity and the communication mode of the AS1/AS2 interface using one line. Other specific parameters can also be added after the quick setup line provided that the quick setup line is the first line in the *Setup Body.*

SETUP command format for quick setup of the AS1/AS2 interface:
Setup Header
SETUP(AS1)
Setup Body
{
set = "BaudParityCommunicationMode";
}

When using the quick setup type, there are expected values for each sub-parameter. Each one is defined in the table below.

Sub-parameter	Expected Values	Definition
Baud	48	set the baud rate for the AS1/AS2/DBG interface to 4800
	96	set the baud rate for the AS1/AS2/DBG interface to 9600
	192	set the baud rate for the AS1/AS2/DBG interface to 19200
	384	set the baud rate for the AS1/AS2/DBG interface to 38400
	768	set the baud rate for the AS1/AS2/DBG interface to 76800
	1150	set the baud rate for the AS1/AS2/DBG interface to 115000
Parity	N (None)	remove the parity bit data sent through the AS1/AS2/DBG
	O (Odd)	set the parity bit of the data sent through AS1/AS2/DBG to odd
	E (Even)	set the parity bit of the data sent through AS1/AS2/DBG to even
Communication Mode	Y	enable the receive and transmit interface of the AS1/AS2/DBG interface (i.e. rxi = Y; and txi = N;)
	N	disable the receive and transmit interface of the AS1/AS2/DBG interface (i.e. rxi = N; and txi = N;)
	C	set the receive and transmit interface of the AS1/AS2/DBG interface as a command processing source (i.e. rxi = C; and txi = C;)

	E	set the receive interface of the AS1/AS2/DBG interface as a command processing source and transmit interface of the RS422/RS485 interface to echo command processing mode (i.e. rxi = C and txi = E;)
	D	set the receive interface of the AS1/AS2/DBG interface to receive debugging data and transmit interface of the AS1/AS2/DBG interface to echo command processing mode (i.e. rxi = D and txi = E;)

Fig. 4.30 *Table defining the sub-parameter values when using the quick setup command for the* AS1/AS2/DBG *interface*

When using any interfaces such as the AS1/AS2/DBG interface, it is important to enable the interface first by using the *SETUP* command in the TU480.mnu file. Similar on how the style parameters are defined, when a setup parameter is not defined in the main menu file, then the default value of that particular setup parameter is assumed and used.

AS1/AS2/DBG setup parameters		
Parameter	**Expected Values**	**Definition**
baud	110 to 6,250,000	• set the baud rate value for the AS1/AS2/DBG interface **(default = 19200)** • any value can be set to allow trimming for deviating clocks i.e. 34850 • if unsure what baud rate is, then refer to the Glossary (Chapter 10) of this guide for further explanation
data	5	• set the number of data bits processed per data transmission through the AS1/AS2/DBG interface to 5
	6	• set the number of data bits processed per data transmission through the AS1/AS2/DBG interface to 6
	7	• set the number of data bits processed per data transmission through the AS1/AS2/DBG interface to 7
	8	• set the number of data bits processed per data transmission through the AS1/AS2/DBG interface to 8 **(default)**
stop	1	• set the number of stop bits processed per data transmission through the AS1/AS2/DBG interface to 1 bit **(default)**
	2	• set the number of stop bits processed per data packet through the AS1/AS2/DBG interface to 2 bits
	15	• set the number of stop bits processed per data packet through the AS1/AS2/DBG interface to **1.5** bits
parity	O (Odd)	• set the parity bit of the data sent through AS1/AS2/DBG to odd • if unsure what parity is refer to the glossary of this guide
	E (Even)	• set the parity bit of the data sent through AS1/AS2/DBG to even
	N (None)	• remove the parity bit data sent through the AS1/AS2/DBG **(default)**
	M (Mark)	• set the parity bit of the data sent through AS1/AS2/DBG to a mark??
	S (Space)	• set the parity bit of the data sent through AS1/AS2/DBG to a space??
rxi	Y	• enable the receive interface of the AS1/AS2/DBG interface
	N	• disable the receive interface of the AS1/AS2/DBG interface **(default)**
	C	• set the receive interface of the AS1/AS2/DBG interface as a command processing source

	D	• set the receive interface of the AS1/AS2/DBG interface to receive debugging data
proc	• termination characters can be specified to generate an interrupt to process a string of data • NB when sending commands (in command mode where rxi = C) to the module, processing only occurs when \\0D or 0D hex is received e.g. TEXT(mytext, “hello world”);; \\0D	
	all;	• trigger on all received characters **(default)**
	CRLF;	• trigger on a carriage return (CR) followed by line feed (LF = 0Dh 0A)
	CR;	• trigger on carriage return (CR = 0dH) in command mode where rxi = C;
	LF;	• trigger on line feed (LF = 0Ah)
	NUL;	• trigger on NUL (00h)
	\\xx;	• trigger on xxh (specify hex value)
	"ABCD";	• string in format defined by SYSTEM encode parameter
	"\\xx\\xx";	• string in format defined by SYSTEM encode parameter
procDel	Y or N	• keep (Y) or remove (N) the termination character(s) before processing **(default = N)**
procNum	0 to 16,780,000	• interrupt on n bytes received as alternative to proc and procDel parameters **(default = 0)**
rxb	1 to 16,780,000	• specify the size of the receive buffer in bytes **(default = 8192 bytes)**
txi	Y	• enable the transmit interface of the AS1/AS2/DBG interface
	N	• disable the transmit interface of the AS1/AS2/DBG interface **(default)**
	C	• set the transmit interface of the AS1/AS2/DBG interface as a command processing source
	E	• set the transmit interface of the AS1/AS2/DBG interface to echo command processing mode
txb	1 to 16,780,000	• specify the size of transmit buffer in bytes **(default = 8192 bytes)**
encode	• set the data encode mode to suit the purpose of the AS1/AS2/DBG interface	
	s	• set data encode to 8 bit ASCII data • codes 00-1F and 80-FF are converted to ASCII "\\00" - "\\1D" and "\\80" - "\\FF" respectively **(default)**
	sr	• set data encode to 8 bit ASCII raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as ASCII+ data
	sd	• set data encode to 8 bit ASCII raw data bytes • all bytes are processed as raw data
	w	• set data encode to UNICODE – HEX Char x 4 = U16 (Most significant hex-pair first) "ABCD" -> \\ABCD
	wr	• set data encode to 8 bit UNICODE raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UNICODE+ data??
	wd	• set data encode to 8 bit UNICODE raw data bytes • all bytes are processed as raw data
	m	• set data encode to 8 bit UTF8 raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UTF8+ data??
	mr	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data

	md	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	D8M	• set data encode to 8 bit data with U16's, U32's etc output **most** significant byte first • the same as data encode sd
	D8L	• set data encode to 8 bit data with U16's, U32's etc output **least** significant byte first
	D16M	• set data encode to 16 bit data with bytes processed as **most** significant byte first • interrupt occurs after two bytes • the same as data encode wd
	D16L	• set data encode to 16 bit data with bytes processed as **least** significant byte first • interrupt occurs after two bytes
	D32M	• set data encode to 32 bit data with bytes processed as **most** significant byte first • interrupt occurs after four bytes • the same as data encode md
	D32L	• set data encode to 32 bit data with bytes processed as **least** significant byte first • interrupt occurs after four bytes
	sh or h8m or h8l	• set data encode to ASCII – HEX Char x 2 = U8 e.g. "A8" -> \\A8
	h16m	• set data encode to ASCII – HEX Char x 4 = U16 (**most** significant hex-pair first) e.g. "ABCD" -> \\ABCD
	h16l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) e.g. "ABCD" -> \\CDAB
	h32m	• set data encode to ASCII – HEX Char x 4 = U32 (**most** significant hex-pair first) "12345678" -> \\12345678
	h32l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) "12345678" -> \\78654321
flow	N	• set the handshaking to NONE **(default)**
	H	• set the flow for handshaking to hardware HB/MB (only applicable to AS1)
	S	• set the flow for handshaking to software XON/XOFF

Fig. 4.31 *Table defining the AS1/AS2/DBG interface setup parameters*

```
//FILENAME: TU480a.mnu

SETUP(AS1)        //
{
baud = 34850;     //
data = 7;         //
stop = 2;         //
parity = N;       //
rxi = C;          //
proc = all;       //
procDel = Y;      //
procNum = 5;      //
rxb = 5250;       //
txi = E;          //
txb = 5250;       //
encode = sr;      //
flow = N;         //
}

SETUP(AS2)        //or a quick setup combination
{
set = "768NC";    //
}

SETUP(DBG)        //or a mixture of both setup types
{
set = "768NC";    //
data = 5;         //
proc = all;       //
encode = mr;      //
}
```

Fig. 4.32 *Example code showing how the AS1/AS2/DBG interface setup is done in iDev*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated. It is possible to use this method to change the *set* parameter for a quick setup interface but it is not recommended, as this would cause a lot of grief to the developer to set the sub parameter values.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updas1func)        //
{
LOAD(AS1.proc,"CRLF";   //
LOAD(AS1.flow,"H";      //
LOAD(AS1.encode,"m";    //
}
```

Fig. 4.33 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for the AS1 Interface*

Due to the complexity, the required hardware changes and peripheral using the AS1/AS2 interface it would not be possible to create a 'basic' example project that can be included in this guide. There are two example projects found on the website that uses the AS1/AS2 interface. The example projects can be downloaded from the website named 'Network Demonstration Project' (link here). This example project exhibits how an external Ethernet module connected to the TFT module through AS1 can be used to communicate with

another TFT module that uses the same setup. Another example project that uses the AS1 interface that can also be downloaded from the website is named 'Transceiver demonstration' (link here). This project uses an external wireless module connected to the TFT modules; this enables the TFT modules to communicate with each other through the AS1 interface wirelessly. These example projects require an advanced level of understanding in iDev and a good knowledge using the AS1/AS2 interface. There are new iDev commands in these example projects that are used in manipulating interfaces such as interrupts. The *INT* (for Interrupts*)* command is introduced and explained in Chapter 4.7 of this guide and processing data is explained in Chapter 4.8.

4.4. SPI (MASTER AND SLAVE) INTERFACE

The SPI (Serial Peripheral Interface) is a synchronous communication interface meaning that it relies on the clock lines of the bus for data transmission to be synchronised?. This interface can either be operated in master and slave mode. Multiple slave devices can be connected and controlled by a single master device. Refer to the Glossary in Chapter 10 for a detailed description of the master and slave mode in interface communication. In the Itron SMART TFT, the SPI interface is enabled by soldering the jumper links on the back of the module. The location of the jumper is always in the bottom right part on the back of the TFT module. This applies to **all** the module sizes and versions except the 3.5" module because there is **no** jumper link J11 on the back of a 3.5" TFT module.

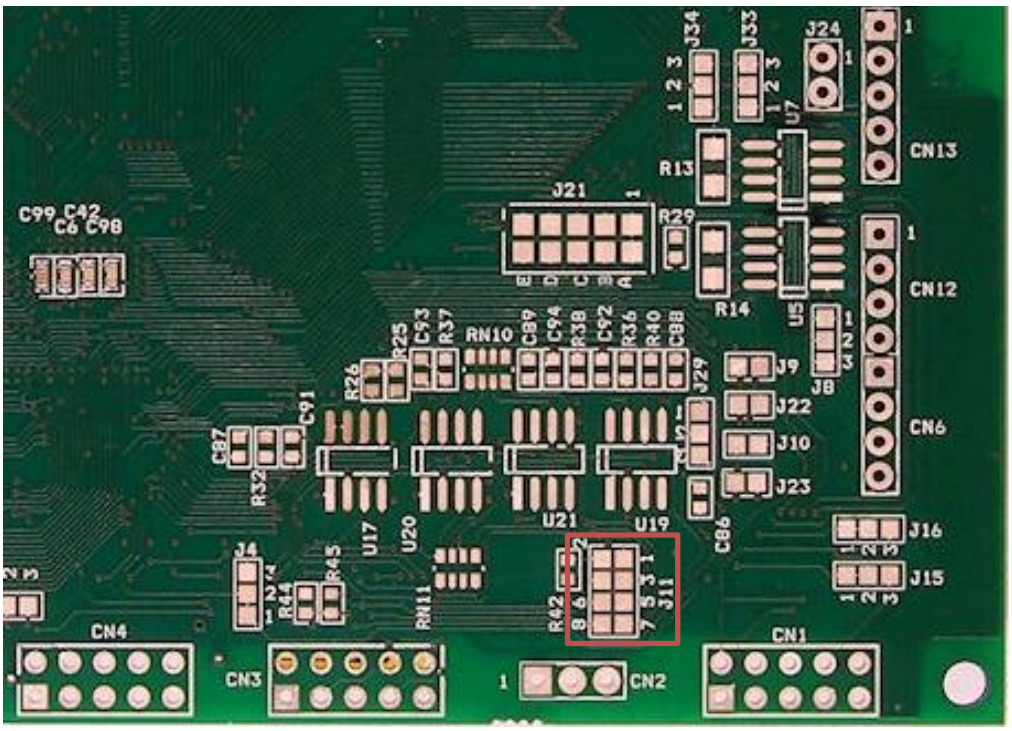


Fig. 4.34 *Diagram indicating the location of the jumper link to enable the SPI interface*

The developer must solder/link pads **1 and 2 together**, **3 and 4 together**, **5 and 6 together**, **7 and 8 together** to enable the SPI interface on the module. The SPI interface would not work properly if these solder links between the pads specified are not done. This jumper pin arrangement applies to all TFT modules except the 3.5" size. The 3.5" module has these jumper links connected internally already. The SPI interface pins are located in connector 3 (**CN3**) of the TFT module as shown in the diagram below.

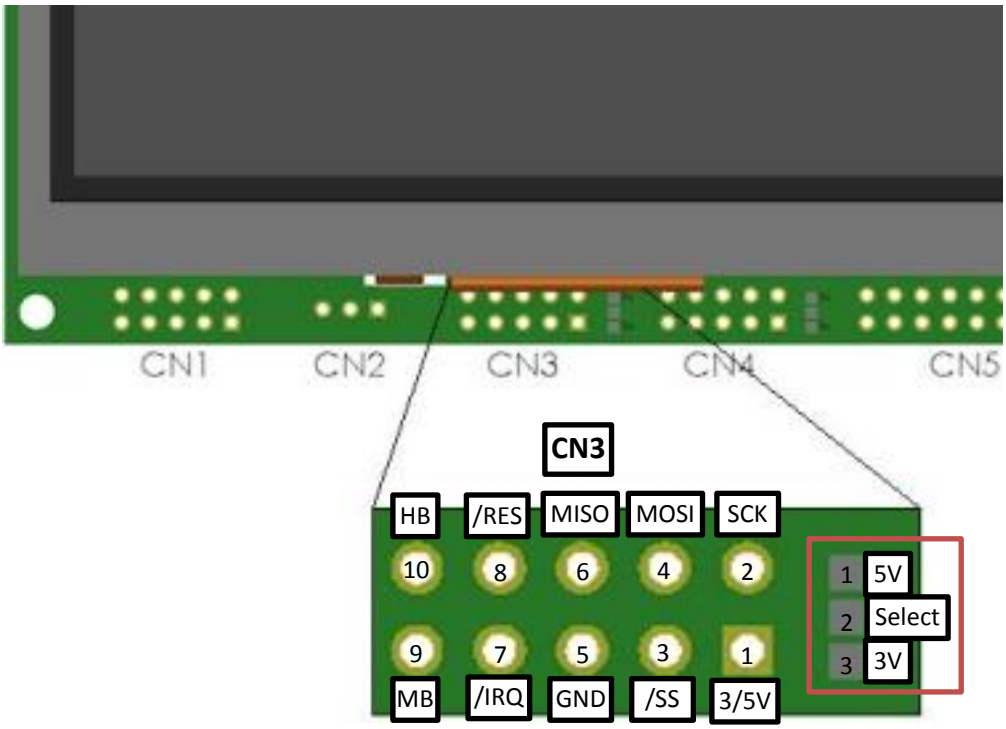


Fig. 4.35 *Diagram to show pin assignments of the SPI interface in the TFT module with the jumper link highlighted*

The jumper link highlighted in RED, allows the developer to select the voltage output level for pin 1 (3V/5V) in connector 3(**CN3**). The developer must either link or solder together the **Select pad** (pad 2) of the jumper to **5V pad** if 5V is required to come out of pin 1(in **CN3**) or the **3V pad** to the **Select pad** (pad 2 of jumper) for 3.3V. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions and sizes do not have this jumper link. This means that the voltage level input/output at pin 4 in connector 3(**CN3**) is fixed at 5V. The default logic level for the SPI interface that all TFT sizes come with is 3.3V. If however, the model number suffix of the TFT module is **K6XXXXXXXS** then this means that the SPI interface logic level of that particular TFT module is 5V. The Itron SMART TFT modules are powered by 5V and so any 5V pin in any connectors can act as a power source for the module and the same is applied to the GND (Common Ground) pin. However, if the TFT module is already powered by 5V then the rest of the 5V pins in the other connectors act as 5V outputs and the same is applied with the GND (Common Ground) pins. This means that an external module that is connected to the TFT module can either be powered by 3.3V or 5V from the module in pin 1.

SPI (CN3) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	3/5V	3.3V (Output) or 5V (Input or Output) power	Input/Output
2	SCK	Serial Clock	Output from Master
3	/SS	Slave Select (Active Low)	Output from Master
4	MOSI	Master Out Slave In	Output from Master
5	GND	Common Ground	Input/Output
6	MISO	Master In Slave Out	Output from Slave
7	/IRQ	Interrupt Request (Active Low)	Output form Slave

8	/RES	Reset Pin (Active Low)	Input to Slave
9	MB	Module Busy	Output from Slave
10	HB	Host Busy	Output from Master
Jumper Link (Highlighted in RED) for AS1 in CN3			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 1 in **CN3**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3V output at pin 1 in **CN3**

Fig. 4.36 *Table describing the pin assignments of the SPI interface*

As mentioned before, SPI interfaces can work as Master or Slave mode. The first scenario is to show the typical connections needed when the Itron SMART TFT is the SPI Slave device and the external module is the SPI Master device. The Itron SMART TFT is referred to as the 'SPI Slave (TFT)' in the diagram below.

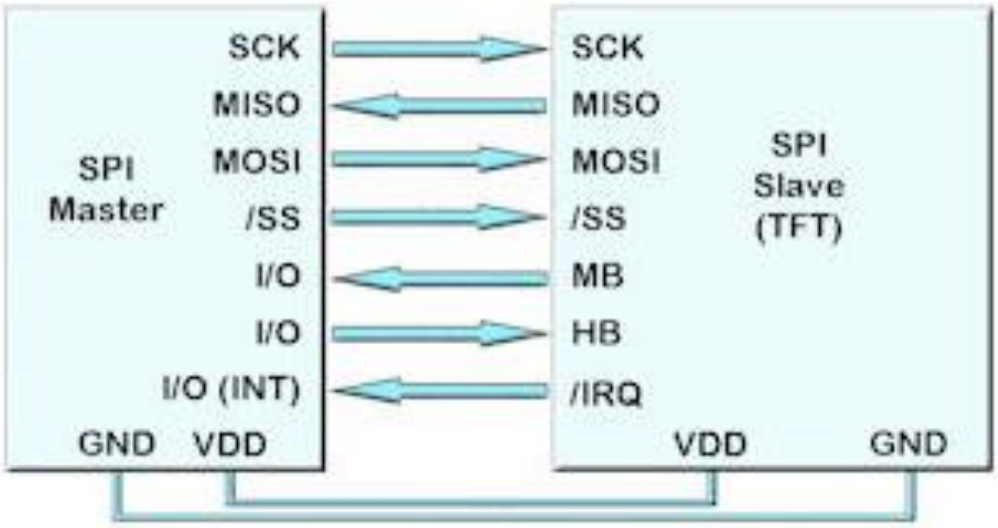


Fig. 4.37 *Diagram to show typical connections between an external device (SPI Master) and the TFT module (SPI Slave)*

The Diagram in Fig 4.37 is meant to represent a general application when using the SPI interface, the TFT module being the **Slave** device; some connections may be different depending on how the pins were assigned on the external module and its purpose. For advanced developers that have used the SPI interface before, the pin connections may be straightforward but for beginners a brief explanation and introduction about the pin connections would help. The **SCK** or Serial Clock is connected to determine and set the speed of serial communication between the master and the slave module. The master specifies the speed or rate of data transmission and the slave has to yield to this speed. The **MISO** and **MOSI** pins are the equivalent of the usual TXD and RXD pins found in other interfaces such as the RS232. The names of these pins are self-explanatory as to how these pins should be connected. The **/SS** pin which stands for Slave Select is connected to begin serial communication between master and slave. This pin in the TFT module is **active LOW**, which means that data transmission starts when the SPI Slave detects this input pin as LOW. The **/SS** pin does not always appear as Serial Select in other SPI devices, this pin is sometimes called /CS (Chip Select) or /STE (Slave Transmit Enable). However, if the SPI Master device

does not have an allocated **/SS** pin, a single digital I/O interface can be used and serve the same purpose. There is also a scenario when the SPI Master device does not have a **/SS** or a usable digital I/O interface, then the **/SS** pin of the TFT module have to be pulled LOW to enable communication between the devices. The **/SS** pin can also be used as and enable pin if the other devices are not connected to the serial line. The HB (Host Busy) pin stops the module from sending data to the host. (More Module busy and Host busy explanation here). The use of the HB and MB busy lines are optional and can be connected together if not required. If the MB (Module Busy) pin is set HIGH, then the input buffer is full or disabled. The **/IRQ** means Interrupt Request and is useful in communication where the master requires notification to allow data coming back to the SPI Master. An Interrupt-triggered digital I/O pin from the SPI Master device normally controls this pin. The **/IRQ** pin is **active LOW** and is driven LOW to signify that data is present in the transmit buffer. If the TFT device is used as the SPI Master device with multiple slaves, a diagram below is created to provide guidance.

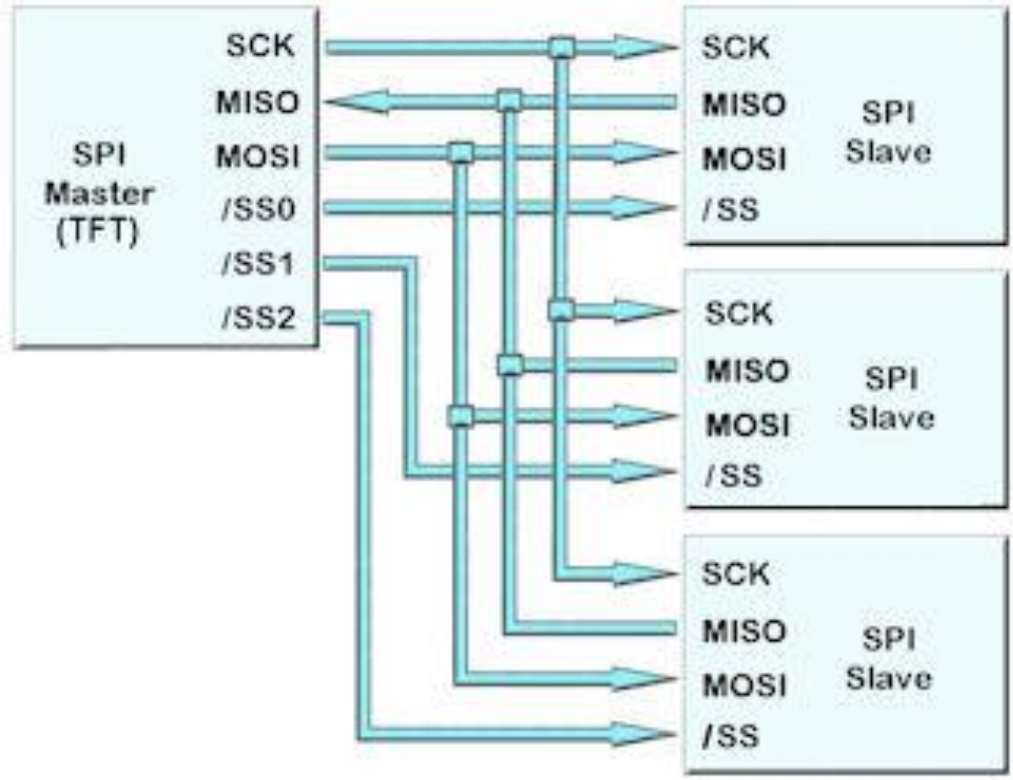


Fig. 4.38 *Diagram to show typical connections between the TFT module (SPI Master) and external modules (multiple SPI Slaves)*

The Diagram in Fig 4.38 is meant to represent a general application when using the SPI interface, the TFT module being the **Master** device; some connections may be different depending on how the pins were assigned on the external module and its purpose. There is no /SS0, /SS1 and /SS2 pins on the TFT module but as mentioned before, the **/SS** pins can be replaced by a simple digital I/O interface from the SPI Master device. The /SS pin in connector 3 (**CN3**) is used as the /SS0 pin and the digital I/O ports found in connector 7 (**CN7**) and connector 4 (**CN4**) can be used as the /SS1 and /SS2 pins. The digital I/O ports of the TFT module are explained properly in Chapter 4.6 of this guide. Once all the hardware side of the interface is finished, then the software side has to be prepared.

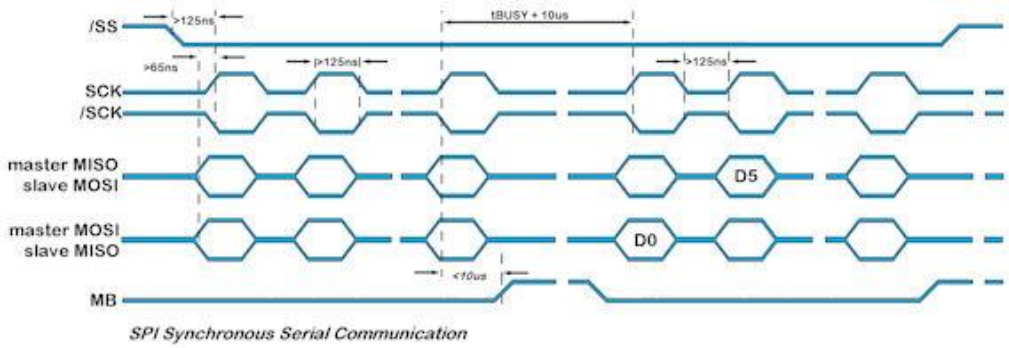


Fig. 4.39 *Diagram showing how data is sent and received through the SPI interface in iDev*

The settings for the SPI interface can be altered using the *SETUP* command in iDev. The *SETUP* command will be used in all the other interfaces as well. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

There is also a quick setup command format that is used for the SPI interface in iDev. This allows the developer to change the SPI mode (master/slave), the idle state and the speed in just one line of code. Other specific parameters can also be added after the quick setup line provided that the quick setup line is the first line in the *Setup Body.*

SETUP command format for quick setup of the SPI interface:
Setup Header
SETUP(SPI)
Setup Body
{
set = "ActiveModeSpeed";
}

When using the quick setup type, there are expected values for each sub-parameter. Each one is defined in the table below.

Sub-parameter	Expected Values	Definition
Active	M (Master)	• set the Itron SMART TFT module as the SPI Master device
	S (Slave)	• set the Itron SMART TFT module as the SPI Slave device
Mode	LR (Low Rising)	• set the idle state of the clock to low rising edge **(default)** • if unsure what idle state is in this context, refer to the glossary in Chapter 10
	LF (Low Falling)	• set the idle state of the clock to low falling edge
	HR (High Rising)	• set the idle state of the clock to high rising edge
	HF (High Falling)	• set the idle state of the clock to high falling edge
Speed	350 to 90000	• set the speed of data transmission value in kilobits/sec(kbs) for SPI Master mode **(default = 1000)** • this sub-parameter is ignored in SPI Slave mode • the maximum recommended practical speed is 1 MHZ (1000) to avoid errors in the SPI communication

Fig. 4.40 *Table defining the sub-parameter values when using the quick setup command for the SPI interface*

When using any interfaces such as the SPI interface, it is important to enable the interface first by using the *SETUP* command in the TU480.mnu file. Similar on how the style parameters are defined, when a setup parameter is not defined in the main menu file, then the default value of that particular setup parameter is assumed and used.

SPI setup parameters		
Parameter	**Expected Values**	**Definition**
active	M (Master)	• set the Itron SMART TFT module as the SPI Master device **(default)**
	S (Slave)	• set the Itron SMART TFT module as the SPI Slave device
	N (None	• disable the SPI interface interface in the Itron SMART TFT module device • useful in instances when the SPI interface have to be turned on and off at certain parts of the code
mode	LR	• set the idle state of the clock to low rising edge **(default)**
	LF	• set the idle state of the clock to low falling edge
	HF	• set the idle state of the clock to high rising edge
	HR	• set the idle state of the clock to high rising edge
speed	350 to 90000	• set the speed of data transmission value in kilobits/sec (kbs) for SPI Master mode **(default = 1000)** • this parameter is ignored in SPI Slave mode • the maximum recommended practical speed is 1 MHZ (1000) to avoid errors in the SPI communication
proc	• termination characters can be specified to generate an interrupt to process a string of data • NB when sending commands (in command mode where rxi = C) to the module, processing only occurs when \\0D or 0D hex is received e.g. TEXT(mytext, “hello world”);; \\0D	
	all;	• trigger on all received characters **(default)**
	CRLF;	• trigger on a carriage return (CR) followed by line feed (LF = 0Dh 0A)
	CR;	• trigger on carriage return (CR = 0dH) in command mode where rxi = C;
	LF;	• trigger on line feed (LF = 0Ah)
	NUL;	• trigger on NUL (00h)

	\\xx;	• trigger on xxh (specify hex value)
	"ABCD";	• string in format defined by SYSTEM encode parameter
	"\\xx\\xx";	• string in format defined by SYSTEM encode parameter
procDel	Y or N	• keep (Y) or remove (N) the termination character(s) before processing **(default = N)**
procNum	0 to 16,780,000	• interrupt on n bytes received as alternative to proc and procDel parameters **(default = 0)**
rxi	Y	• enable the receive interface of the SPI interface
	N	• disable the receive interface of the SPI interface **(default)**
	C	• set the receive interface of the SPI interface as a command processing source
rxb	1 to 16,780,000	• specify the size of the receive buffer in bytes **(default = 8192 bytes)**
rxo	M	• set the receive data order to most significant bit **(default)**
	L	• set the receive data order to least significant bit
rxf	MB	• use hardware MB (module busy) to signify that the receive buffer is full, handshaking?? • this is an equivalent of the flow parameter in other interfaces
	N	• disable hardware MB (module busy)
txi	Y	• enable the transmit interface of the SPI interface
	N	• disable the transmit interface of the SPI interface **(default)**
	C	• set the transmit interface of the SPI interface as a command processing source
	E	• set the transmit interface of the SPI interface to echo command processing mode
txb	1 to 16,780,000	• specify the size of transmit buffer in bytes **(default = 8192 bytes)**
txo	M	• set the transmit data order to most significant bit **(default)**
	B	• set the transmit data order to least significant bit **(default)**
encode	• set the data encode mode to suit the purpose of the SPI interface	
	s	• set data encode to 8 bit ASCII data • codes 00-1F and 80-FF are converted to ASCII "\\00" - "\\1D" and "\\80" - "\\FF" respectively **(default)**
	sr	• set data encode to 8 bit ASCII raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as ASCII+ data
	sd	• set data encode to 8 bit ASCII raw data bytes • all bytes are processed as raw data
	w	• set data encode to UNICODE – HEX Char x 4 = U16 (Most significant hex-pair first) "ABCD" -> \\ABCD
	wr	• set data encode to 8 bit UNICODE raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UNICODE+ data??
	wd	• set data encode to 8 bit UNICODE raw data bytes • all bytes are processed as raw data
	m	• set data encode to 8 bit UTF8 raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UTF8+ data??
	mr	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	md	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data

	D8M	• set data encode to 8 bit data with U16's, U32's etc output **most** significant byte first • the same as data encode sd
	D8L	• set data encode to 8 bit data with U16's, U32's etc output **least** significant byte first
	D16M	• set data encode to 16 bit data with bytes processed as **most** significant byte first • interrupt occurs after two bytes • the same as data encode wd
	D16L	• set data encode to 16 bit data with bytes processed as **least** significant byte first • interrupt occurs after two bytes
	D32M	• set data encode to 32 bit data with bytes processed as **most** significant byte first • interrupt occurs after four bytes • the same as data encode md
	D32L	• set data encode to 32 bit data with bytes processed as **least** significant byte first • interrupt occurs after four bytes
	sh or h8m or h8l	• set data encode to ASCII – HEX Char x 2 = U8 e.g. "A8" -> \\A8
	h16m	• set data encode to ASCII – HEX Char x 4 = U16 (**most** significant hex-pair first) e.g. "ABCD" -> \\ABCD
	h16l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) e.g. "ABCD" -> \\CDAB
	h32m	• set data encode to ASCII – HEX Char x 4 = U32 (**most** significant hex-pair first) "12345678" -> \\12345678
	h32l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) "12345678" -> \\78654321
end	\\(HEX code)	• set the byte sent to host (SPI Master device) when no data left in the display's SPI transmit buffer **(default = \\FF)**
dummy	\\(HEX code)	• set the dummy byte set to Itron SMART TFT module which is ignored so that data can be received by the host (SPI Master device) **(default = 0)**

Fig. 4.41 *Table defining the SPI interface setup parameters*

The *speed* parameter is ignored by the TFT module in Slave mode as this is set by the SPI Master device, so there is no need to set this parameter in Slave mode. Although the clock is capable of 90 Mhz the recommended maximum speed for data transmission is 1 Mhz for external SPI communication. Extensive testing of the implementation of the SPI communication is suggested. An example *SETUP* command for the SPI interface is found below.

```
//FILENAME: TU480a.mnu

SETUP(SPI)          //
{
active = S;         //
mode = 7;           //
speed = 2;          //
rxi = C;            //
rxo = M;            //
procDel = Y;        //
procNum = 5;        //
txi = E;            //
txb = 5250;         //
encode = sr;
end = \\AA;
dummy = \\25;
}

SETUP(SPI)          //or a quick setup combination
{
set = "MHF4000";    //
}

SETUP(SPI)          //or a mixture of both setup types
{
set = "MHF4000";    //
txb = 9800;         //
proc = all;         //
end = \\AA;         //
}
```

Fig. 4.42 *Example code showing how the SPI interface setup is done in iDev*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated. It is possible to use this method to change the *set* parameter for a quick setup interface but it is not recommended, as this would cause a lot of grief to the developer to set the sub parameter values.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updspifunc)          //
{
LOAD(SPI.proc,"CRLF";     //
LOAD(SPI.active,"N";      //
LOAD(SPI.encode,"mr";     //
}
```

Fig. 4.43 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for the SPI Interface*

An example project using the SPI interface named 'SPI Demonstration' (link here for Slave mode and here for Master mode). These example projects enable communication between two TFT modules, one operating in Slave mode and the other Master mode. There are iDev commands that are used in this SPI example project for manipulating interfaces such as interrupts(see *INT* (for Interrupts*)* in Chapter 4.7 and handling data in Chapter 4.8).

4.5. I2C/TWI (MASTER AND SLAVE) INTERFACE

The I2C (Inter-Integrated Circuit) or TWI (two-wire interface) interface is an interface that is operated in master and slave mode. This interface only requires two connections namely SCL (Serial Clock) and SDA (Serial Data) lines. If the Itron TFT module is used as the Master device, multiple slave devices can be connected to it. In I2C it is also possible to have multiple master devices controlling multiple slave devices but this is quite complicated and only advanced developers that have used the I2C bus before can use it adequately. It is recommended to do some background reading if the developer is required to use the I2C interface using multiple master modes. Refer to the Glossary in Chapter 10 for a detailed description of the master and slave mode in interface communication. The I2C interface pins are located in connector 3 (**CN3**) of the TFT module as shown in the diagram below.

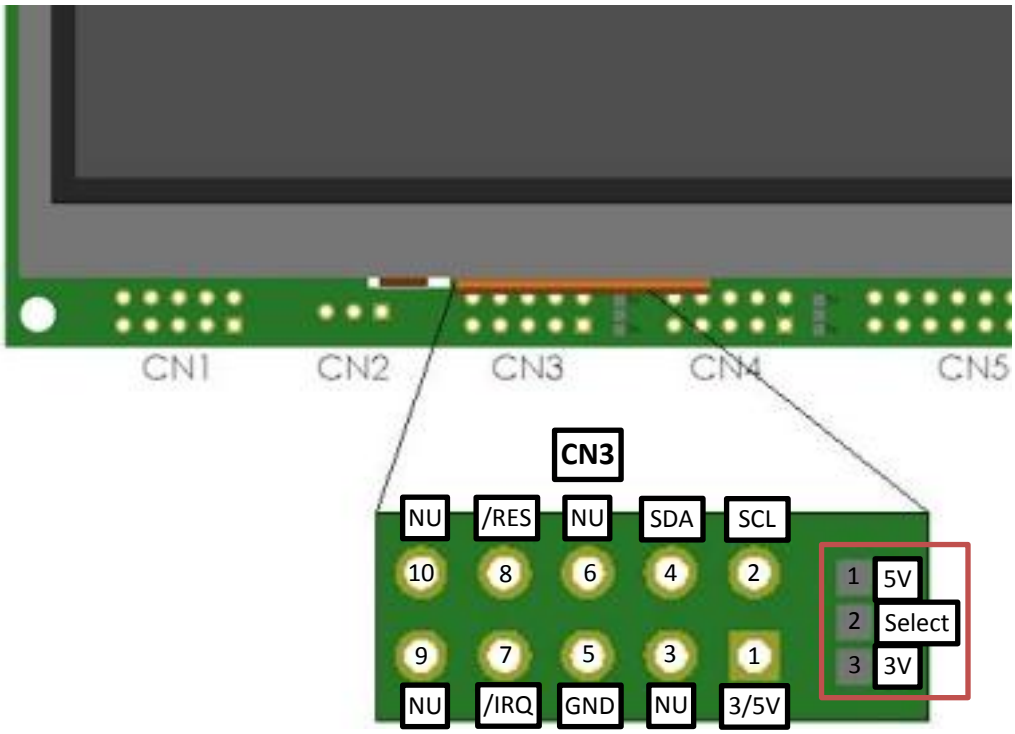


Fig 4.44 *Diagram to show pin assignments of the I2C interface in the TFT module with the jumper link highlighted*

The jumper link highlighted in RED, allows the developer to select the voltage output level for pin 1 (3V/5V) in connector 3(**CN3**). The developer must either link or solder together the **Select pad** (pad 2) of the jumper to **5V pad** if 5V is required to come out of pin 1(in **CN3**) or the **3V pad** to the **Select pad** (pad 2 of jumper) for 3.3V. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions do not have this jumper link. This means that the voltage level input/output at pin 4 in connector 3(**CN3**) is fixed at 5V. The default logic level for the I2C interface that all TFT sizes come with is 3.3V. If however, the model number suffix of the TFT module is **K6XXXXXXXI** then this means that the I2C interface logic level of that particular TFT module is 5V. The Itron SMART TFT modules are powered by 5V and so any 5V pin in any connectors can act as a power source for the module and the same is applied to the GND (Common Ground) pin. However, if the TFT module is already powered by 5V then the rest of the 5V pins in the other connectors act as 5V outputs and the same is applied with the GND (Common Ground) pins. This means that

an external module that is connected to the TFT module can either be powered by 3.3V or 5V from the module in pin 1.

I2C (CN3) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	3/5V	3.3 V (Output) or 5V (Input or Output) power	Input/Output
2	SCL	Serial Clock	Input/Output
3	NU	Not Used (Do not connect to anything)	N/A
4	SDA	Serial Data	Input/Output
5	GND	Common Ground	Input/Output
6	NU	Not Used (Do not connect to anything)	N/A
7	/IRQ	Interrupt Request (Active Low)	Output from Slave
8	/RES	Reset Pin (Active Low)	Input to Slave
9	NU	Not Used (Do not connect to anything)	N/A
10	NU	Not Used (Do not connect to anything)	N/A
Jumper Link (Highlighted in RED) for I2C in CN3			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 1 in **CN3**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3.3V output at pin 1 in **CN3**

Fig. 4.45 *Table describing the pin assignments of the I2C interface*

As mentioned before, I2C interfaces can work as Master or Slave mode. The first scenario is to show the typical connections needed when the Itron SMART TFT is the I2C Slave device and the external module is the SPI Master device. The Itron SMART TFT is referred to as the 'I2C Slave (TFT)' in the diagram below.

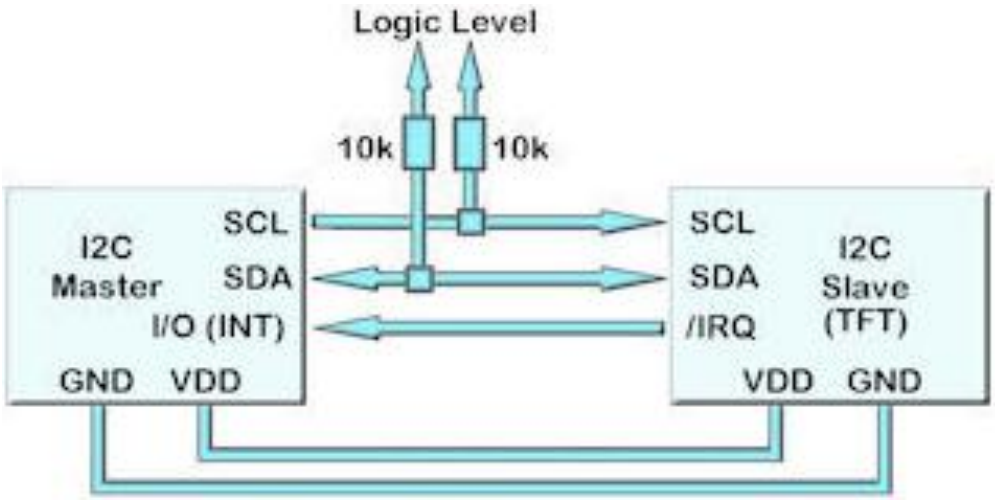


Fig. 4.46 *Diagram to show typical connections between an external device (I2C Master) and the TFT module (I2C Slave)*

The Diagram in Fig 4.46 is meant to represent a general application when using the I2C interface, the TFT module being the **Slave** device; some connections may be different depending on how the pins were assigned on the external module and its purpose. The developer must fit 10kohm pull-up resistors (to Logic Level) to the SDA and SCL lines in the

I2C bus to be able to drive the output of these lines HIGH. It is important to remember that only one set of pull-up resistors for the whole I2C bus interface not for each device involved and used. Some module versions however, have this pull-up resistors fitted internally. All TFT modules have a part number with the version number indicated on the back of the module. The usual suffix **K61XXXXXX vX** for most module sizes, as indicated the module version number is the last part of the part number e.g. A 4.3" module has the part number TU480x272-K612A1TU v12, this is a version 12 module; it has internal-pull up resistors fitted.

Module Version with internal pull-up resistors fitted for I2C bus	
Module Size	**Module Version**
3.5"	v3 onwards
4.3"	v8 onwards
5.7"	v4 onwards
7.0"	v5 onwards

Fig. 4.47 *Table showing which module versions have internal pull-up resistor fitted in I2C bus*

For advanced developers that have used the I2C interface before, the pin connections may be straightforward but for beginners a brief explanation and introduction about the pin connections would help. The SCL (Serial Clock) ensures that data transmission over the I2C bus is synchronised and it sets the speed of serial communication between the master and the slave module. The speed or rate of data transmission is specified by the master and the slave has to yield to this speed. The SCD (Serial Data) is obviously where the data bytes are sent and received. These pins are connected to each other (as in the diagram) to all the devices using the I2C interface. The **/IRQ** means Interrupt Request and is useful in communication where the master requires notification to allow data coming back to the I2C Master. This pin is normally controlled by an Interrupt-triggered digital I/O pin from the I2C Master device. The **/IRQ** pin is **active LOW** and is driven LOW to signify that data is present in the transmit buffer. If the TFT device is used as the I2C Master device with multiple slaves, a diagram below is created to provide guidance.

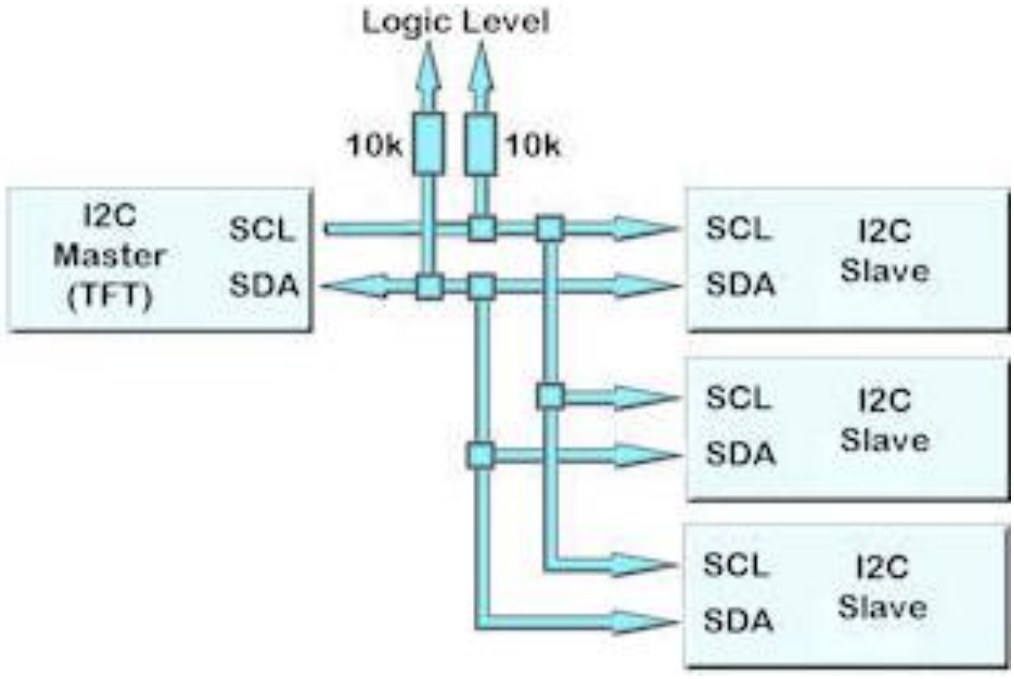


Fig. 4.48 *Diagram to show typical connections between the TFT module (SPI Master) and*

external modules (multiple SPI Slaves)

The Diagram in Fig 4.48 is meant to represent a general application when using the SPI interface, the TFT module being the **Master** device; some connections may be different depending on how the pins were assigned on the external module and its purpose. Once all the hardware side of the interface is finished, then the software side has to be prepared.

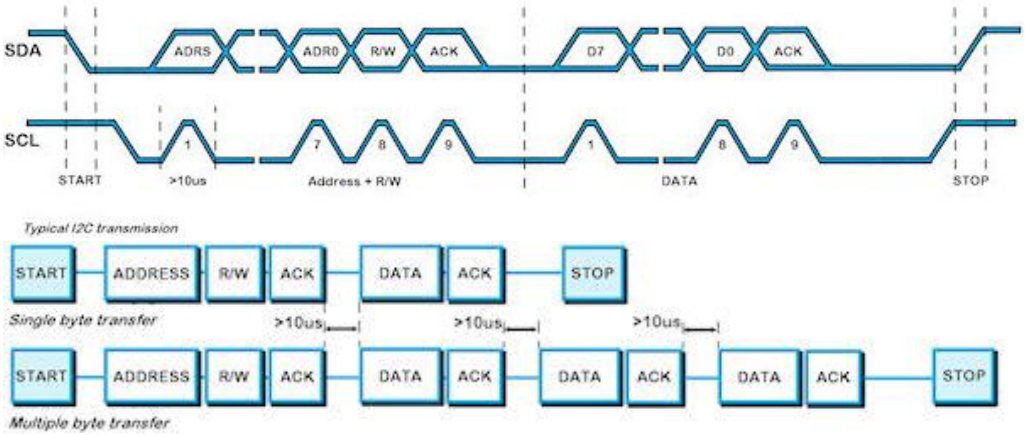


Fig. 4.49 *Diagram showing how data is sent and received through the AS1/AS2 interface*

The I2C Master device regulates the clock line (SCL) and generates when to send START and STOP signals. A START signal is sent by driving the SDA low whilst SCL is high and a STOP signal is sent by driving SDA high while SCL high. After a START condition followed by an address and the +R/W bit (Read and Write bit) is detected by the SLAVE device, the command/data bytes are stored in a buffer size specified in the setup. The address used in I2C bus in iDev is a 7 bit address with the range \\01 to \\7F in HEX code. The +R/W bit is used to determine whether the Master is writing or reading from the Slave. The module will pull SDA low during the 9th clock cycle of a data transfer to acknowledge the receipt of a byte. Additional data bits can then be send provided that the Master has received an ACK bit (Acknowledge bit). If the Master has not detected an ACK bit then the data transfer must be started again by providing a STOP and start condition and address +R/W bit low. This explanation of the I2C communication requires background knowledge in programming and interfaces, some terms used are found in the glossary in Chapter 10.

The settings for the I2C interface can be altered using the *SETUP* command in iDev. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

There is also a quick setup command format that is used for I2C in iDev. This allows the developer to change the communication mode and address bit of the I2C interface using one line of code. Other specific parameters can also be added after the quick setup line provided that the quick setup line is the first line in the *Setup Body.*

SETUP command format for quick setup for the I2C interface:
Setup Header
SETUP(I2C)
Setup Body
{
set = "CommunicationModeAddress";
}

When using the quick setup type, there are expected values for each sub-parameter. Each one is defined in the table below.

Sub-parameter	Expected Values	Definition
Communication Mode	M	set the Itron SMART TFT module as the I2C Master device (i.e. rxi = Y; txi = n; active = M;)
	S	set the Itron SMART TFT module as the I2C Slave device (i.e. rxi = Y; txi = n; active = S;)
	Y	enable the receive and transmit interface of the I2C interface (i.e. rxi = Y; txi = N; active = S;)
	N	disable the receive and transmit interface of the I2C interface (i.e. rxi = N; txi = N; active = S;)
	C	set the receive and transmit interface of the I2C interface as a command processing source (i.e. rxi = C; txi = C; active = S;)
	E	set the receive interface of the I2C interface as a command processing source and transmit interface of the RS422/RS485 interface to echo command processing mode (i.e. rxi = C txi = E; active = S;)
Address	"//01" to "//7F"	set the 7 bit address of the TFT module (Slave mode) in HEX code **(no default, this parameter has to be specified all the time in the setup)** the address specified in this parameter is ignored when in Master mode

Fig. 4.50 *Table defining the sub-parameter values when using the quick setup command for the I2C interface*

When using any interfaces such as the I2C interface, it is important to enable the interface first by using the *SETUP* command in the TU480.mnu file. Similar on how the style parameters are defined, when a setup parameter is not defined in the main menu file, then the default value of that particular setup parameter is assumed and used.

I2C setup parameters		
Parameter	**Expected Values**	**Definition**
addr	"//01" to "//7F"	• set the 7 bit address of the TFT module (Slave mode) in HEX code **(no default, this parameter has to be specified all the time in the setup)** • the address specified in this parameter is ignored when in Master mode
active	M (Master)	• set the Itron SMART TFT module as the I2C Master device
	S (Slave)	• set the Itron SMART TFT module as the I2C Slave device
	N (None	• disable the I2C interface interface in the Itron SMART TFT module device **(default)** • useful in instances when the I2C interface have to be turned on and off at certain parts of the code
speed	20 to 400	• set the speed of data transmission value in kilobits/sec (kbs) for I2C Master mode **(default = 100)** • this parameter is ignored in I2C Slave mode
proc	• termination characters can be specified to generate an interrupt to process a string of data • NB when sending commands (in command mode where rxi = C) to the module, processing only occurs when \\0D or 0D hex is received e.g. TEXT(mytext, "hello world");; \\0D	
	all;	• trigger on all received characters **(default)**
	CRLF;	• trigger on a carriage return (CR) followed by line feed (LF = 0Dh 0A)
	CR;	• trigger on carriage return (CR = 0dH) in command mode where rxi = C;
	LF;	• trigger on line feed (LF = 0Ah)
	NUL;	• trigger on NUL (00h)
	\\xx;	• trigger on xxh (specify hex value)
	"ABCD";	• string in format defined by SYSTEM encode parameter
	"\\xx\\xx";	• string in format defined by SYSTEM encode parameter
procDel	Y or N	• keep (Y) or remove (N) the termination character(s) before processing **(default = N)**
procNum	0 to 16,780,000	• interrupt on n bytes received as alternative to proc and procDel parameters **(default = 0)**
rxi	Y	• enable the receive interface of the I2C interface
	N	• disable the receive interface of the I2C interface **(default)**
	C	• set the receive interface of the I2C interface as a command processing source
	D	• set the receive interface of the I2C interface to receive debugging data
rxb	1 to 16,780,000	• specify the size of the receive buffer in bytes **(default = 8192 bytes)**
txi	Y	• enable the transmit interface of the I2C interface
	N	• disable the transmit interface of the I2C interface **(default)**
	C	• set the transmit interface of the I2C interface as a command processing source
	E	• set the transmit interface of the I2C interface to echo command processing mode
txb	1 to 16,780,000	• specify the size of transmit buffer in bytes **(default = 8192 bytes)**
encode	• set the data encode mode to suit the purpose of the SPI interface	

	s	• set data encode to 8 bit ASCII data • codes 00-1F and 80-FF are converted to ASCII "\\00" - "\\1D" and "\\80" - "\\FF" respectively **(default)**
	sr	• set data encode to 8 bit ASCII raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as ASCII+ data
	sd	• set data encode to 8 bit ASCII raw data bytes • all bytes are processed as raw data
	w	• set data encode to UNICODE – HEX Char x 4 = U16 (Most significant hex-pair first) "ABCD" -> \\ABCD
	wr	• set data encode to 8 bit UNICODE raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UNICODE+ data??
	wd	• set data encode to 8 bit UNICODE raw data bytes • all bytes are processed as raw data
	m	• set data encode to 8 bit UTF8 raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UTF8+ data??
	mr	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	md	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	D8M	• set data encode to 8 bit data with U16's, U32's etc output **most** significant byte first • the same as data encode sd
	D8L	• set data encode to 8 bit data with U16's, U32's etc output **least** significant byte first
	D16M	• set data encode to 16 bit data with bytes processed as **most** significant byte first • interrupt occurs after two bytes • the same as data encode wd
	D16L	• set data encode to 16 bit data with bytes processed as **least** significant byte first • interrupt occurs after two bytes
	D32M	• set data encode to 32 bit data with bytes processed as **most** significant byte first • interrupt occurs after four bytes • the same as data encode md
	D32L	• set data encode to 32 bit data with bytes processed as **least** significant byte first • interrupt occurs after four bytes
	sh or h8m or h8l	• set data encode to ASCII – HEX Char x 2 = U8 e.g. "A8" -> \\A8
	h16m	• set data encode to ASCII – HEX Char x 4 = U16 (**most** significant hex-pair first) e.g. "ABCD" -> \\ABCD
	h16l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) e.g. "ABCD" -> \\CDAB
	h32m	• set data encode to ASCII – HEX Char x 4 = U32 (**most** significant hex-pair first) "12345678" -> \\12345678
	h32l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) "12345678" -> \\78654321
end	\\(HEX code)	• set the byte sent to host (SPI Master device) when no data left in the display's SPI transmit buffer **(default = \\FF)**

Fig. 4.51 *Table defining the I2C interface setup parameters*

The address specified in the *SETUP* is only processed and used when the TFT module is in Slave mode. On the other hand, the address of the Slave device that the TFT module (Master) is trying to communicate to is specified in using the *LOAD* command format. An example code that uses the *SETUP* command for the I2C interface is created.

```
//FILENAME: TU480a.mnu

SETUP(I2C)     //
{
addr = \\3E;   //
active = S;    //
proc = all;    //
procDel = Y;   //
procNum = 5;   //
rxi = C;       //
rxb = 5250;    //
txi = E;       //
txb = 5250;    //
encode = sr;   //
end = \\FE;    //
}

SETUP(I2C)     //or a quick setup combination
{
set = "C3E";   //
}

SETUP(I2C)     //or a mixture of both setup types
{
set = "C3E";   //
procDel = Y;   //
proc = all;    //
encode = mr;   //
end = \\0D;    //
}
```

Fig. 4.52 *Example code showing how the I2C interface setup is done in iDev*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated. It is possible to use this method to change the *set* parameter for a quick setup interface but it is not recommended, as this would cause a lot of grief to the developer to set the sub parameter values.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updi2cfunc)          //
{
LOAD(I2C.proc,"CRLF";     //
LOAD(I2C.end,"\\1A";      //
LOAD(I2C.encode,"r";      //
}
```

Fig. 4.53 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for the I2C Interface*

There is an example project that can be downloaded from the website named 'I^2C Sensor Demonstration Project' (link here). This project uses three external devices connected through the I2C interface to the TFT module. The three external devices (accelerometer, light sensor and temperature sensor) are the I2C Slave devices and the TFT module is the I2C Master device. The setup of the three I2C Slave devices connected to the I2C Master device is exactly the same as the diagram in Fig 4.48 in this guide. This example also demonstrates how the ADC ports are used. There are new iDev commands in these example projects that are used in manipulating interfaces such as interrupts. The *INT* (for Interrupts*)* command is introduced and explained in Chapter 4.7 of this guide and processing data is explained in Chapter 4.8.

4.6. Digital Input/Output (I/O) Interface & External Keyboard

There are 31 Digital Input and Output (I/O) lines in an Itron TFT module for all sizes. The digital I/O lines **K00 to K23** operate at a fixed **3.3V logic level**, if the developer requires a different logic level, then an external level shifter IC (definition in Chapter 10) have to be used. The I/O lines K24 to K30 have a variation of logic levels (3.3V or 5V) depending on the module version (see Fig 4.60). The Digital Inputs include an optional pull-up resistor ~50K-120K in value and the Outputs can source ~1mA and sink ~3mA. The I/O ports are initially pulled HIGH at **RESET** or **POWER ON**. An external keyboard or matrix can be connected to the digital I/O pins and perform operations based on which key/button is pressed. These 31 I/O lines can be configured to scan a matrix of buttons with up to 240 keys i.e. a keypad or keyboard with up to 240 keys/buttons can be connected to the TFT module and each key/button pressed can be assigned to perform an action specified in the code. Dual key presses are supported to enable 'Shift' functionality. There are no diodes required to be connected in the key matrix configured for dual key operation. Due to the amount of pins needed for the 31 Digital I/O lines, the locations of these pins are spread out in different connectors. The Digital I/O lines K00 to K15 are found in Connector 7 (**CN7**), K16 to K23 are found in Connector 4 (**CN4**) and K24 to K30 are found in Connector 3 (**CN3**).

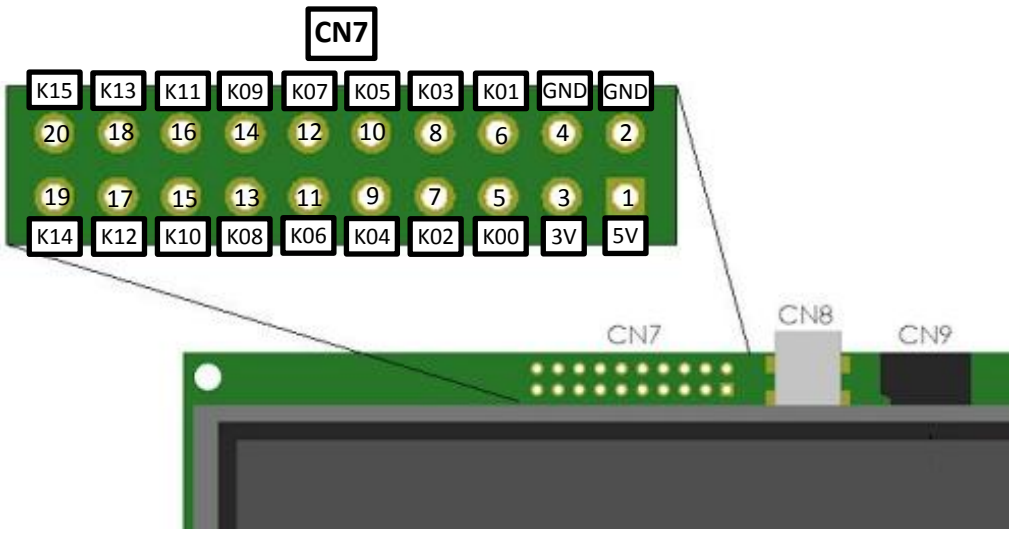


Fig 4.54 *Diagram to show pin assignments of the Digital I/O interface (K00-K15) in CN7 of the TFT module*

Digital I/O (CN7) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	5V	5V power	Input/Output
2	GND	Common Ground	Input/Output
3	3V	3.3 V power	Output
4	GND	Common Ground	Input/Output
5	K00	Digital I/O pin K00	Input/Output
6	K01	Digital I/O pin K01	Input/Output
7	K02	Digital I/O pin K02	Input/Output
8	K03	Digital I/O pin K03	Input/Output
9	K04	Digital I/O pin K04	Input/Output
10	K05	Digital I/O pin K05	Input/Output
11	K06	Digital I/O pin K06	Input/Output
12	K07	Digital I/O pin K07	Input/Output
13	K08	Digital I/O pin K08	Input/Output
14	K09	Digital I/O pin K09	Input/Output
15	K10	Digital I/O pin K10	Input/Output
16	K11	Digital I/O pin K11	Input/Output
17	K12	Digital I/O pin K12	Input/Output
18	K13	Digital I/O pin K13	Input/Output
19	K14	Digital I/O pin K14	Input/Output
20	K15	Digital I/O pin K15	Input/Output

Fig 4.55 *Table describing the pin assignments of the Digital I/O interface (K00-K15) in CN7*

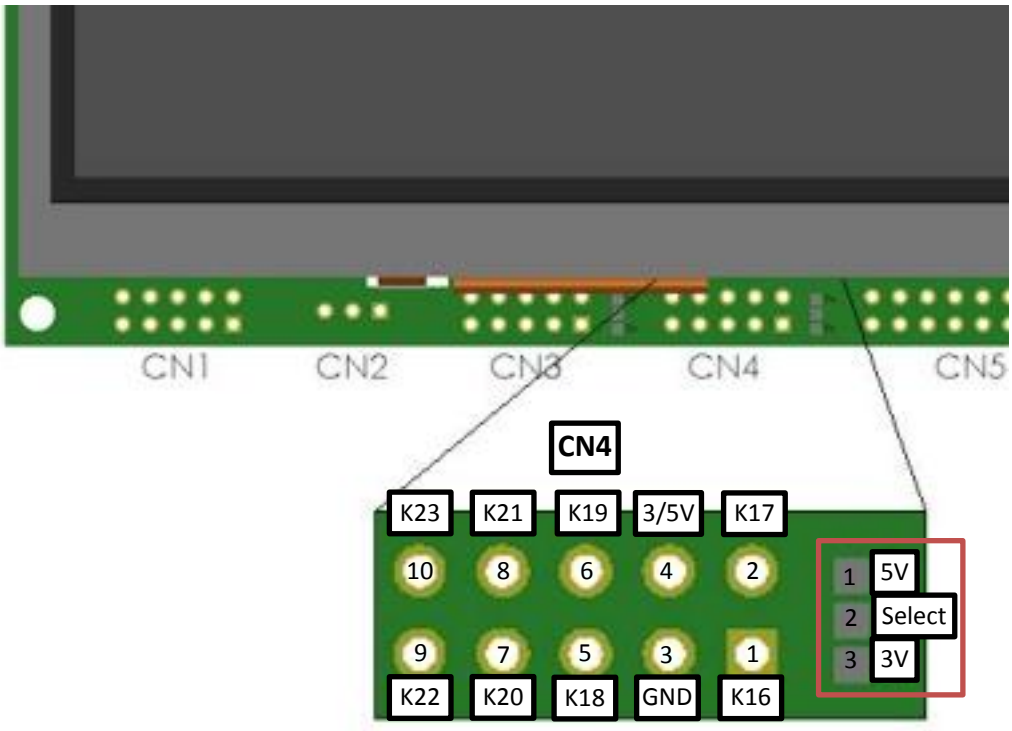


Fig 4.56 *Diagram to show pin assignments of the Digital I/O interface (K16-K23) in CN4 of the TFT module with the jumper link highlighted*

Digital I/O (CN4) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	K16	Digital I/O pin K16	Input/Output
2	K17	Digital I/O pin K17	Input/Output
3	GND	Common Ground	Input/Output
4	3/5V	3.3V (Output) or 5V (Input/Output) power	IntputOutput
5	K18	Digital I/O pin K18	Input/Output
6	K19	Digital I/O pin K19	Input/Output
7	K20	Digital I/O pin K20	Input/Output
8	K21	Digital I/O pin K21	Input/Output
9	K22	Digital I/O pin K22	Input/Output
10	K23	Digital I/O pin K23	Input/Output
Jumper Link (Highlighted in RED) for Digital I/O in CN4			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 4 in **CN4**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3.3V output at pin 4 in **CN4**

Fig 4.57 *Table describing the pin assignments of the Digital I/O interface (K16-K23) in CN4*

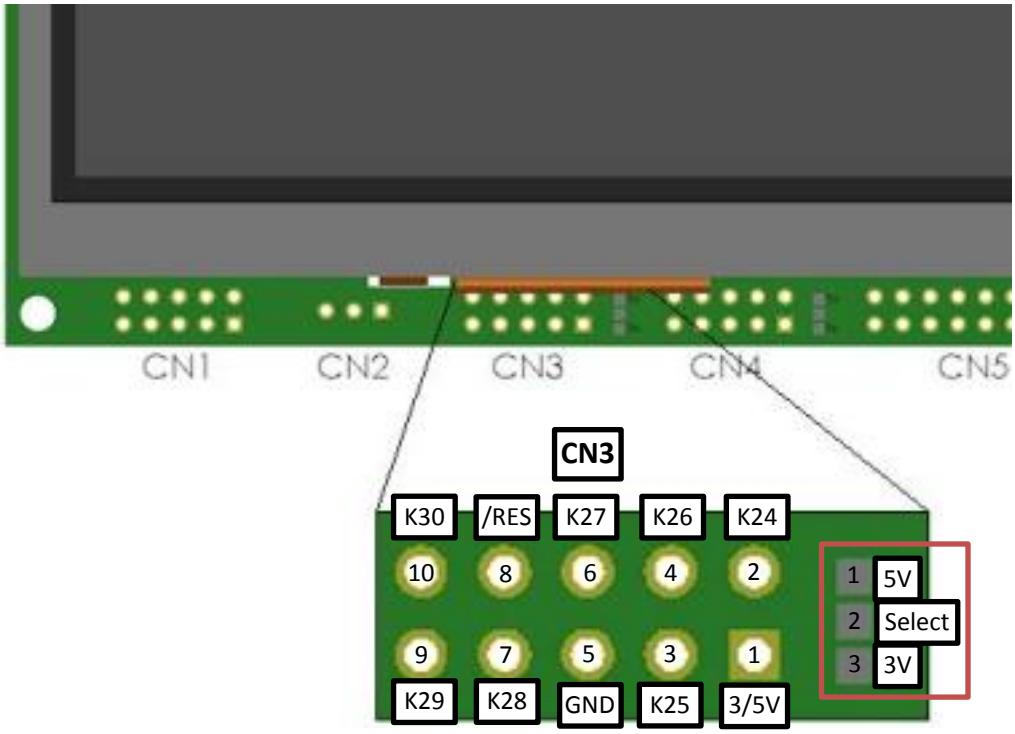


Fig 4.58 *Diagram to show pin assignments of the Digital I/O interface (K24-K30) in CN3 of the TFT module with the jumper link highlighted*

Digital I/O (CN3) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	3/5V	3.3V (Output) or 5V (Input/Output) power	Input/Output
2	K24	Digital I/O pin K24	Input/Output
3	K25	Digital I/O pin K25	Input/Output
4	K26	Digital I/O pin K26	Input/Output
5	GND	Common Ground	Input/Output
6	K27	Digital I/O pin K27	Input/Output
7	K28	Digital I/O pin K28	Input/Output
8	/RES	Reset Pin (Active Low)	Input
9	K29	Digital I/O pin K29	Input/Output
10	K30	Digital I/O pin K30	Input/Output
Jumper Link (Highlighted in RED) for Digital I/O in CN3			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 1 in **CN3**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3.3V output at pin 1 in **CN3**

Fig. 4.59 *Table describing the pin assignments of the Digital I/O interface (K24-K30) in CN3*

The jumper link highlighted in RED in Fig 4.57 and Fig 4.58, allows the developer to select the voltage output level for pin 4 (3V/5V) in connector 4(**CN4**) and pin 1 in connector 3(**CN3**). Both jumper links in **CN4** and **CN3** work the same way, the developer must either link or solder together the **Select pad** (pad 2) of the jumper to **5V pad** if 5V is required to come out of pin 4 in **CN4** and pin 1 in **CN3** or the **3V pad** to the **Select pad** (pad 2 of the jumper) for 3.3V output. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions and sizes do not have this jumper link. This means that the voltage level input/output at pin 4 in **CN4** and pin 1 in **CN3** is fixed at 5V. The default logic level for the Digital I/O interface that all TFT sizes come with is 3.3V. The Itron SMART TFT modules are powered by 5V and so any 5V pin in any connectors can act as a power source for the module and the same is applied to the GND (Common Ground) pin. However, if the TFT module is already powered by 5V then the rest of the 5V pins in the other connectors act as 5V outputs and the same is applied with the GND (Common Ground) pins; an external module that is connected to the TFT module can either be powered by 3.3V or 5V from the module at pin 4 in **CN4** or at pin 1 in **CN3**. The AS1, SPI and I2C interfaces are also located in **CN3** of the module, and the logic levels of these interfaces can either be 3.3V or 5V with a level shifter fitted. TFT modules with the level shifter ICs fitted have certain pins in **CN3** operating at 5V instead of the typical 3.3V. The table below shows which I/O pins operate at 3.3V and 5V with the module number suffix specified.

Module Number suffix	**I/O pins with 3.3V Output**	**I/O pin with 5V Input/Output**
K6XXXXXXXA	4(K26), 6(K27)	2(K24), 3(K25), 7(K28), 9(K29), 10(K30)
K6XXXXXXXS	None	3(K25), 4(K26), 6(K28), 7(K28), 9(K29), 10(K30)
K6XXXXXXXI	3(K25) ,6(K28), 9(K29), 10(K30)	2(K24), 4(K26), 7(K28)

Fig. 4.60 *Table describing which I/O pins operate at 3.3V and 5V*

The basic use of Digital I/O in electronics is a switch/button. These pins can be used to manipulate outputs to turn ON an external LED based on the state of the switch (push button).

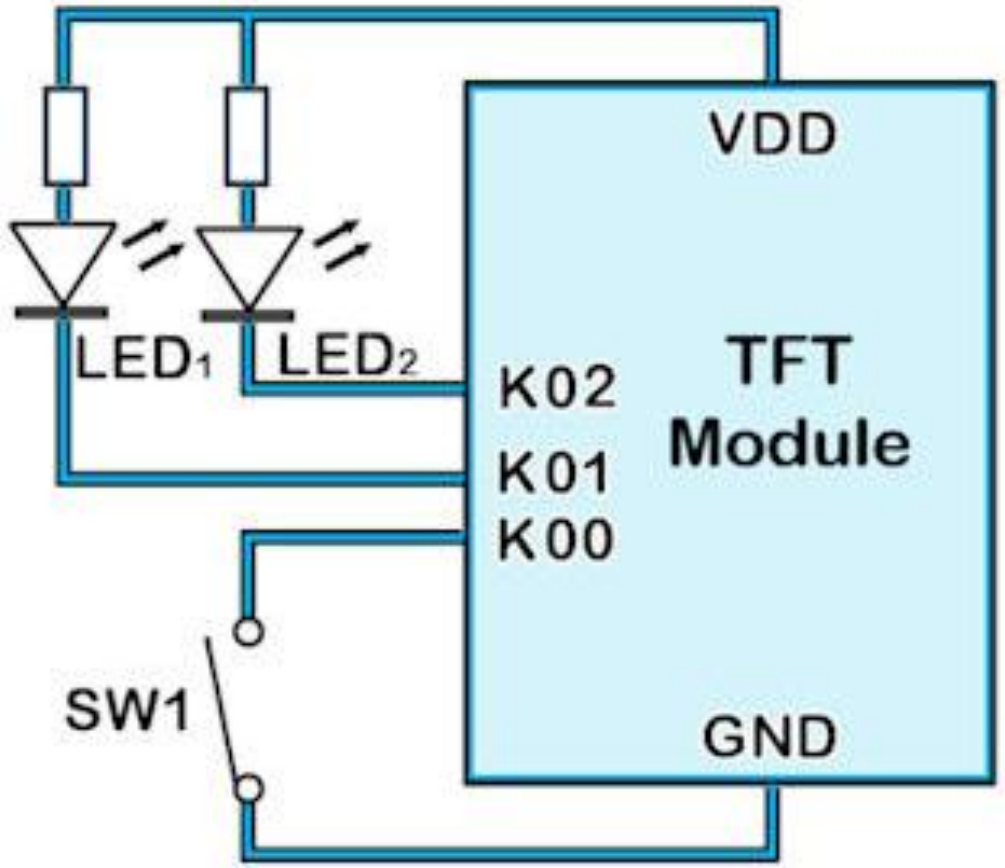


Fig. 4.61 *Diagram to show typical connections between the TFT module and I/O controlled peripherals such as an LED and a push button used as a switch*

There are 31 Digital I/O pins available on the TFT module; in the example above only 3 were used. The diagram is only meant to represent a basic use and application of the I/O ports, it can get complicated if more I/O lines are used. The operation shown above is set such that the I/O K00 is set as an input for the external switch and the I/O K01 and K02 are set as outputs for the two LEDs. When the push button is pressed, the I/O detects a logic LOW. Then it is specified in iDev that when K00 is LOW, set K01 and K02 outputs HIGH which consequently turns ON LED1 and LED2. A different configuration can also be applied so that the opposite happens when K00 is LOW. The Digital I/O control is explained thoroughly in Chapter 4.9. The I/O lines can also be connected to an external keyboard or matrix of keys.

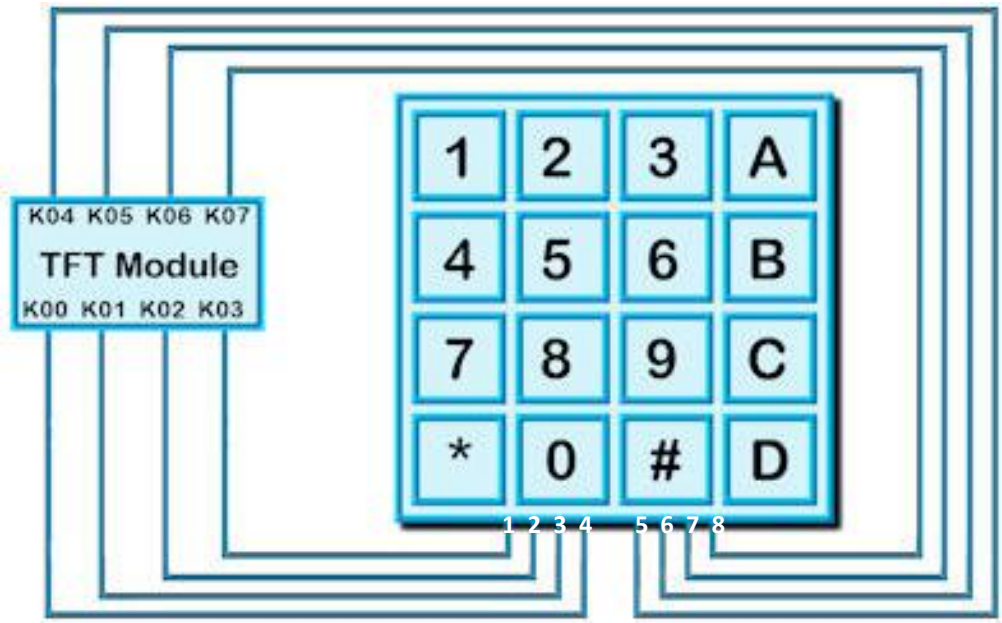


Fig. 4.62 *Diagram to show typical connections between the TFT module and a 4x4 matrix keypad*

In general, a 4x4(16 key) matrix keypad has pin assignments separated by rows and columns. In Fig 4.62, the pins 1 to 4 control the rows and pins 5 to 8 controls the columns. The pin assignments of different keypads are not always like this, so it is recommended to check the datasheet of the keypad being used first before making connections. The I/O K00 to K03 is assigned to control rows and I/O K04 to K07 for the columns. When a button is pressed, a signal input change to the designated I/O pair is detected, of which a function can be called. A more thorough explanation on how to manipulate and control the external keyboard is found in Chapter 4.9.

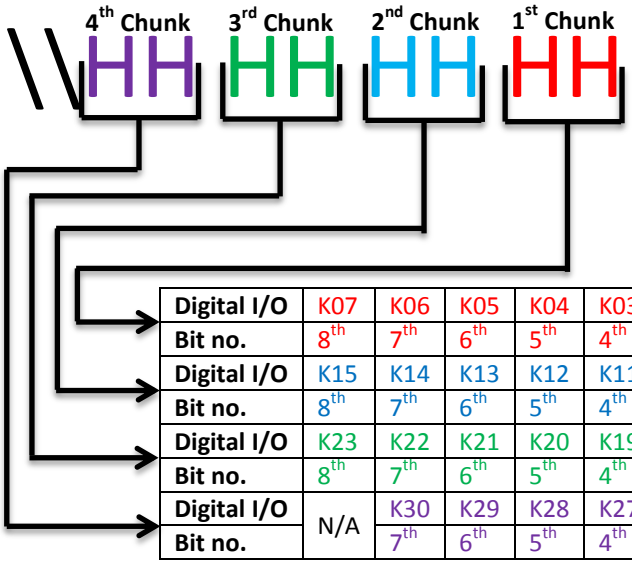
The settings for the Digital I/O interface can be altered using the *SETUP* command in iDev. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.* The *Interface* parameter in the *Setup Header* has to be changed to *KEYIO* for the Digital I/O interface.

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

Digital I/O setup parameters		
Parameter	**Expected Values**	**Definition**
active	\\00000000-\\7FFFFFFF	• specify which digital I/O lines are active • HIGH is active, LOW is inactive • value used is an 8-digit HEX code • **(default = \\00000000)**
inp	\\00000000-\\7FFFFFFF	• specify which digital I/O lines are set as Input or Output • HIGH is input, LOW is ouput • value used is an 8-digit HEX code • **(default = \\00000000)**
trig	\\00000000-\\7FFFFFFF	• specify how the interrupts are triggered in digital I/O lines • HIGH is to trigger interrupt, LOW is to not trigger interrupt • value used is an 8-digit HEX code • **(default = \\00000000)**
edge	\\00000000-\\7FFFFFFF	• specify the idle state of the digital I/O lines • HIGH is rising edge, LOW is falling edge • value used is an 8-digit HEX code • **(default = \\00000000)**
keyb	\\00000000-\\7FFFFFFF	• specify which digital I/O lines are used for external keyboard keys/buttons • HIGH is active I/O for keyboard, LOW is inactive I/O • value used is an 8-digit HEX code • **(default = \\00000000)**
pullup	\\00000000-\\7FFFFFFF	• specify which digital I/O lines should have input pull up resistors (~50k – 120k) activated • HIGH is active, LOW is inactive • value used is an 8-digit HEX code • **(default = \\7FFFFFFF)**

Fig. 4.63 *Table defining the Digital I/O interface setup parameters*

As stated before, there are 31 I/O lines that can be manipulated in the TFT module. It is evident from the table that the expected values for the setup parameters use 8-digit HEX code but it is not obvious what it resembles. The 8-digit HEX code is divided into four 'chunks' of 2-digit HEX value. Each 'chunk' can control 8 I/O lines, but as there's only 31 I/O lines used, the 4th 'chunk' only controls 7 I/O lines. The first 'chunk' controls Digital I/O K00 to K07, second controls Digital I/O K08 to K15, third controls Digital I/O K16 to K23 and lastly the fourth controls Digital I/O 24 to K30.



Digital I/O	K07	K06	K05	K04	K03	K02	K01	K00
Bit no.	8th	7th	6th	5th	4th	3rd	2nd	1st
Digital I/O	K15	K14	K13	K12	K11	K10	K09	K08
Bit no.	8th	7th	6th	5th	4th	3rd	2nd	1st
Digital I/O	K23	K22	K21	K20	K19	K18	K17	K16
Bit no.	8th	7th	6th	5th	4th	3rd	2nd	1st
Digital I/O	N/A	K30	K29	K28	K27	K26	K25	K24
Bit no.		7th	6th	5th	4th	3rd	2nd	1st

Fig. 4.64 *Diagram to show how 'chunks' of 2-bit HEX value and bit number are allocated to the Digital I/O lines*

There are 256 possible values in a 2-digit HEX code and an 8-bit Binary value. The Binary value is used to specify which I/O should be enabled and is converted to a HEX value for the setup parameter. For an introduction and explanation about the binary numbering system, refer to Chapter 3.4. Take the first 'chunk' as an example; it controls the first 8 I/O lines (K00 to K07) and in Binary each 'bit' can either be a 1 or a 0. So if the I/O lines K00 and K03 to K05 are needed to be enabled, the Binary value for the first 'chunk' that is used would have to be *00111001* and in HEX the value is *39* (refer to HEX in Chapter 10 for the conversion table). In order to enable the I/O lines specified, the parameter value used for the *active* parameter is *00000039*.

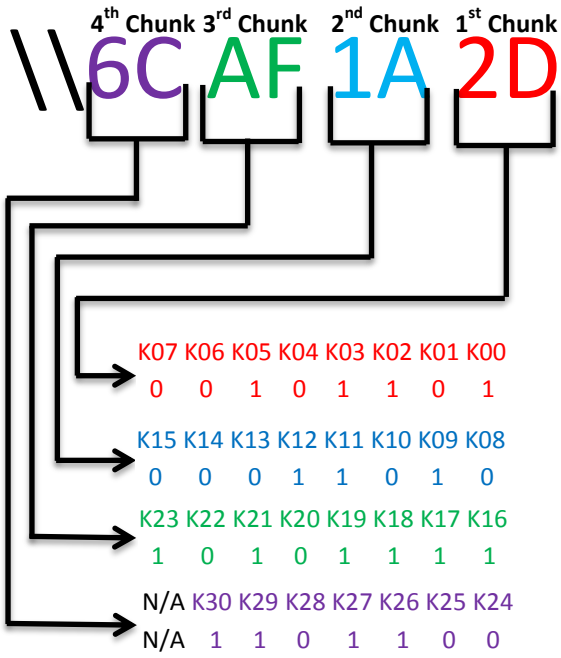


Fig. 4.65 *Diagram with an example 8-digit HEX value, indicating which I/O lines would have a HIGH(1) or LOW(0) value*

```
//FILENAME: TU480a.mnu

SETUP(KEYIO)
{
active = \\6CAF1A2D; //
inp = \\7FFF1A2D;    //
trig = \\6CAF1A2D;   //
keyb = \\00000000;   //
}
```

Fig. 4.66 *Example code showing how the Digital I/O interface setup is done in iDev*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updiofunc)                  //
{
LOAD(KEYIO.inp,"\\0000FFFF";     //
LOAD(KEYIO.trig,"\\6CAFFFFF";    //
}
```

Fig. 4.67 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for the I/O Interface*

4.7. INTERRUPTS

In programming, an interrupt is used to notify or alert the processor (CPU) that an event occurred needs attention and processing. Interrupts are a vital part when using interfaces as it allows the CPU to receive and process data coming in and out of that particular interface. In iDev, the *INT* command is used to specify which interface or I/O buffer is used and the function to be called every time the interrupt event occurs. An interrupt will occur when the interface setup parameters *proc* and *procNum* allow activity within the buffer and the appropriate type of interrupt is set. Serial interfaces can trigger an interrupt on a byte received/transmitted. The I/O interface can trigger an interrupt when an input change is detected. Interrupts in iDev can be stated inside or outside the *Page Body* or even in both as long as they have different interrupt names. If the interrupt is defined **outside** of the *Page Body* this means that this interrupt is a *Global Interrupt*. A Global Interrupt is an interrupt that is active continuously throughout the whole iDev project, so this interrupt can be accessed anywhere in the code. On the contrary, an interrupt that is defined **inside** the *Page Body* is a *Local Interrupt*. This type of interrupt can only be processed when the page that it is defined in is active or displayed. Multiple interrupts can also be created provided that each of them has a unique interrupt name but does not necessarily need to have different interface buffers i.e. different pages can have interrupts from the same interface buffer, provided that each interrupt defined have unique names. An interface buffer is a built-in buffer that processes and handles data in the specified interface every time an interrupt occurs; the data processed by this buffer is handled by the function specified in the *INT* command. The placement of an interrupt in the code is important; it should always be stated **after** the *SETUP* command for the interface that is being used. It is also recommended to state an interrupt after the function associated with it is defined to minimise the possibility of causing errors. As stated in Chapter 2.5.1 and Chapter 2.5.2, the *SHOW* and *HIDE* command can be used to enable and disable an interrupt anywhere in the program.

INT command format to set up Interface interrupts in iDev:
INT(Interrupt Name, Interface Buffer, Function);

Interface Buffer Parameter Expected Values and Definition	
Interface Buffer	**Definition**
RS2RXC	interrupt on each character received for the RS232 interface buffer
RS4RXC	interrupt on each character received for the RS422/RS485 interface buffer
AS1RXC	interrupt on each character received for the AS1 interface buffer
AS2RXC	interrupt on each character received for the AS2 interface buffer
DBGRXC	interrupt on each character received for the DBG interface buffer
I2CRXC	interrupt on each character received for the I2C interface buffer
KXX (where XX is the I/O assignment)	interrupt when an input change in the specified KXX I/O pin is detected

Fig. 4.68 *Table describing the expected values for the Interface Buffer parameter in the INT command in iDev*

An example code is created to show how to use and setup an interrupt for the RS232 interface. As this would require an external module/peripheral that uses the RS232 interface connected to the TFT module, only an example code can be shown without screen shots of the TFT module display.

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}
SETUP command format for quick setup of the RS232 interface:
Setup Header
SETUP(RS2)
Setup Body
{
set = "BaudParityCommunicationMode";
}
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
}

FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
INT command format to set up interrupts in iDev:
INT(Interrupt Name, Interface Buffer, Function);
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

SETUP(RS2)       //
{
set = "768NC";   //
data = 5;        //
proc = all;      //
encode = sd;     //
}

STYLE(homepgst,Page)       //
{
back = white;              //
}
STYLE(Ascii16txst,Text)    //
{
font = Ascii16;            //
col = black;               //
maxRows = 1;               //
maxLen = 32;               //
}

VAR(bufvar,"",TXT);                                    //

PAGE(homepage,homepgst)                                //
{
POSN(240,135);                                         //
TEXT(mybufftxt,"data received here",Ascii16txst);      //
}

FUNC(RS2Eventfunc)                    //
{
LOAD(bufvar,RS2);                     //
TEXT(mybufftxt,bufvar);;              //
}

INT(rs2int,RS2RXC,RS2Eventfunc);      //
SHOW(homepage);                       //
```

Fig. 4.69 *Example code demonstrating how the INT command in iDev is used in a basic level*

This example code in Fig 4.69 would display the data that is received and interpret it as 8 bit ASCII raw data bytes, this interpretation can be changed in the *encode* parameter in the *Setup Body* of the interface. The data is stored in a buffer variable and then placed in a text component called *mybufftxt.* The interrupt is triggered every time a byte or a character is received in the RS232 interface. The frequency and how the interrupt is triggered are

specified in the *proc* and *procNum* parameter in the *Setup Body* of the interface. The function *RS2Eventfunc* is called every time the interrupt is triggered. The buffer's contents are then read using the *LOAD* command. This clears the interrupt which avoids the function to keep getting called. This buffer must be read every time the interrupt is triggered otherwise the function *RS2Eventfunc* will keep getting called. The *LOAD* command for the interfaces is explained thoroughly in Chapter 4.8. The *INT* command is also used for *COUNTERS* and *TIMERS* in iDev (see Chapter 3.7 and Chapter 3.8).

Counter Interrupts in iDev

INT command format to use as wrap-around interrupt for the runtime counter:
INT(Interrupt name, Runtime counter, Function to be called);

Runtime Counter with INT	Definition
INT(myint, CNTMILLI, mymilfunc);	call the function *mymilfunc* every 1000 milliseconds
INT(myint, CNTSECS, mysecfunc);	call the function *mysecfunc* every 60 seconds
INT(myint, CNTMINS, myminfunc);	call the function *myminfunc* every 60 minutes
INT(myint, CNTHOURS, myhrsfunc);	call the function *myhrsfunc* every 24 hours
INT(myint, CNTDAYS, mydayfunc);	call the function *mydayfunc* every 4,294,967,295 days

Fig. 4.70 *Table demonstrating how wrap-around interrupts with runtime counters are used*

Timer Interrupts in iDev

INT command format to setup timer interrupts, where x is the number of timer interrupt being used (TIMER0-TIMER9):
INT(Timer Interrupt name, TIMERx, Function to be called);
LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value);

Timer Manipulation Usage	Definition
LOAD(myvar, TIMERx);	read the remaining time value before TIMERx expires
LOAD(TIMERx, duration);	run TIMERx once based on the duration value specified
LOAD(TIMERx, duration, repeat);	run TIMERx multiple times based on repeat value and the duration value
LOAD(TIMERx, 0);	clear and reset TIMERx

Fig. 4.71 *Table showing how timer interrupts in iDev can be manipulated*

Example Usage	Definition
LOAD(TIMER2,1000);	*TIMER2* runs once and expires after 1000 ms (1 second)
LOAD(TIMER4,500,5);	*TIMER4*'s duration is repeated 5 times with each duration at 500 ms
LOAD(TIMER6,1000,0);	*TIMER9* runs forever, expiring every 1000 ms (1 second)
LOAD(TIMER3,0);	clear and reset TIMER3
LOAD(TIMER7,time);	*TIMER7* runs once and expires after the duration value specified in the variable *time*
LOAD(myvar,TIMER4);	read the remaining time left in *TIMER4* and store it as an integer in variable *myvar*

Fig. 4.72 *Example table demonstrating how manipulation on timer interrupts is applied*

4.8. HANDLING DATA IN INTERFACES – LOAD

In iDev, sending and receiving data through a specified interface can be achieved using the *LOAD* command. The data stored in a variable or an array can be sent through an interface and data received is stored a variable or an array, ready for processing. The data stored in the variable from an interface can be in different formats. It is important that the appropriate *encode* parameter in the interface setup is selected. The buffer specified is read and cleared using the *LOAD* command; this avoids the function related to an interrupt to keep on getting called. The *LOAD* command format below applies to the RS232, RS422/RS485, AS1/AS2 and SPI interfaces.

LOAD command format to **send** data through a specified interface:
LOAD(Interface, Var/Array/"Data");
LOAD command format to **send** multiple data through a specified interface:
LOAD(Interface, Var1/Array1/"Data1", Var2/Array2/"Data2");
LOAD command format to **receive** data through a specified interface:
LOAD(Variable/Array, Interface);

All the data in the list are 'concatenated' when sending multiple data using the *LOAD* command. The *LOAD* command to send data for the I2C interface requires an extra parameter to specify the address and the size of bytes to read after the data has been sent. The *LOAD* command is the same as the others when receiving data through I2C.

LOAD command format to **send** data through I2C:
LOAD(I2C, Device Address, Read Bytes, Var/Array/"Data");
LOAD command format to **send** multiple data through I2C:
LOAD(I2C, Device Address, Read Bytes, Var1/Array1/"Data1", Var2/Array2/"Data2");
LOAD command format to **receive** data through I2C:
LOAD(Variable/Array, I2C);

```
//FILENAME: TU480a.mnu

FUNC(sendAS1func)                    //
{
LOAD(AS1,"temp","setting has",myvar); //
}

FUNC(AS1Eventfunc)          //
{
LOAD(mybuf,AS1);            //
TEXT(mybufftxt,mybuf);;     //
}

FUNC(I2CEventfunc)           //
{
LOAD(I2C,\\28,1,"test");     //
WAIT(20);                    //
LOAD(bufvar,I2C);            //
TEXT(mybufftxt,bufvar);;     //
}
```

Fig. 4.73 *Example code demonstrating how the LOAD command to handle data in an interface is implemented*

4.9. Controlling I/O Interface and External Keyboard (Incomplete)

The Digital I/O interface lines can only have a Logic HIGH or Logic LOW state. The *LOAD* command is used to control the Digital I/O lines and the *KEY* command is used to control an external keyboard.

LOAD command to control Digital Output, where XX is the I/O assignment, 0 (Logic LOW) and 1 (Logic HIGH):
LOAD(KXX, 0/1);
LOAD command to store the Digital I/O state, where XX is the I/O assignment, value stored is either 0 (Logic LOW) and 1 (Logic HIGH):
LOAD(Variable/Arrray, KXX);
In iDev, 8 I/O lines are combined to form different combinations that can be manipulated using one *LOAD* command instead of one for each I/O line. Each combination is assigned to a built-in I/O variable.

8-Bit I/O variable	I/O and Bit Assignment							
	8th	7th	6th	5th	4th	3rd	2nd	1st
KA	K07	K06	K05	K04	K03	K02	K01	K00
KB	K15	K14	K13	K12	K11	K10	K09	K08
KC	K14	K12	K10	K08	K06	K04	K02	K00
KD	K15	K13	K11	K09	K07	K05	K03	K01
KE	K23	K22	K21	K20	K19	K18	K17	K16

Fig. 4.74 *Table describing combinations of I/O lines to which 8-Bit variable*

LOAD command to control Digital Output, where V is the 8-bit I/O variable :
LOAD(KV, \\HEX code);
LOAD command to store the Digital I/O state, where V is the 8-bit I/O variable, value stored is in HEX code:
LOAD(Variable/Arrray, KV);

The value in the *HEX code* parameter is deduced from an 8-bit binary value. The method on which I/O lines are assigned is exactly the same as the 'chunk' explanation in Chapter 4.6. Take KC as an example; to set the I/O lines K14, K08, K00 to logic HIGH then the value used must be \\91. The value \\91 when converted to binary is 10010001. When 10010001 is compared to the table above, it is evident that the 8th (K14), 5th (K08) and 1st (K00) bit has the value 1 and the rest 0. An explanation on the binary numbering system is found in Chapter 3.4.

```
//FILENAME: TU480a.mnu

FUNC(iofunc)
{
LOAD(K05,1);       //set K05 to logic HIGH
LOAD(K21,0);       //set K21 to logic LOW
LOAD(KC,\\91);     //set K14,K08,K00 to logic HIGH & K12,K10,K06,K04,K02 to logic LOW

LOAD(iovar,K05);  //load the I/O K05 state to iovar variable
LOAD(iovar,KC);   //load the 8 bit I/O states to iovar variable
}
```

Fig. 4.75 *Example code showing how the LOAD command is used to manipulate I/O interface*

The external keyboard uses a combination of I/O lines that are specified in the *SETUP* command (see Chapter 4.6). The *type* parameter in the key style has to be changed to *keyio* to enable the external keyboard in iDev, so the built-in key styles cannot be used. The parameter *KXX* and *KYY* specified which I/O line is associated with that particular key component. In iDev, a maximum of 240 keys is supported.

KEY command format for external key:
KEY(Key component name, Function name, KXX, KYY, Key style);
KEY command format using inline commands for external key:
KEY(Key component name, [Inline command1,Inline command2..], KXX, KYY, Key style);

An example using a 2x2 matrix keypad and an LED connected to the TFT module is created. The LED is connected to K04 and the matrix keypad is connected to K00 to K03. The connection setup used is shown below.

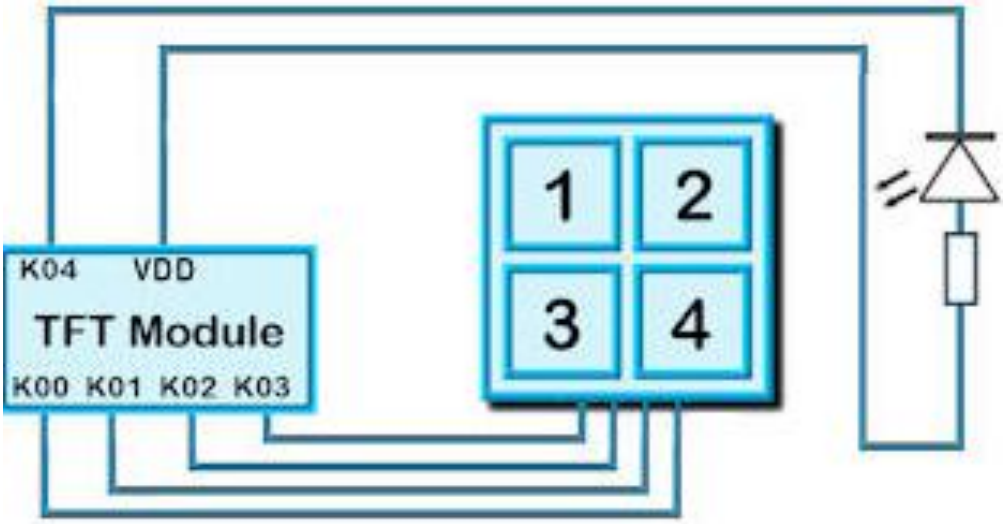


Fig. 4.76 *Diagram showing how the 2x2 matrix keypad and the LED are connected to the TFT module*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
TEXT command format:
TEXT(Text component name, "Text component", Text Style)
KEY command format using inline commands for external key:
KEY(Key component name, [Inline command1,Inline command2..], KXX, KYY, Key style);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
}
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME: TU480a.mnu

SETUP(KEYIO)          //
{
active = \\0000001F; //
keyb = \\0000000F;   //
}

STYLE(extkeyst)             //
{
type = keyio;               //
debounce = 100;             //
delay = 500;                //
repeat = 500;               //
action = D;                 //
}
STYLE(homepgst,Page)        //
{
back = white;               //
}
STYLE(Ascii16txst,Text)     //
{
font = Ascii16;             //
col = black;                //
maxRows = 1;                //
maxLen = 20;                //
}

PAGE(homepage,homepgst)     //
{
POSN(240,135);                                  //
TEXT(extkeytxt,"button is pressed",Ascii16txst);  //
KEY(extkey1,[LOAD(K04,0);],K00,K02,extkeyst); //LED turned on when key '1' is pressed
KEY(extkey2,[LOAD(K04,0);],K00,K03,extkeyst); //LED turned on when key '3' is pressed
KEY(extkey3,[TEXT(extkeytxt,"2rd key pressed");;],K01,K02,extkeyst);
//
KEY(extkey4,[TEXT(extkeytxt,"4th key pressed");;],K01,K03,extkeyst);
//
}
SHOW(homepage);             //
```

Fig. 4.77 *Example code using the KEY and LOAD command to manipulate I/O lines and external keyboard*

When an external keyboard is used, it is important to enable the Digital I/O used in the *active* parameter of the *SETUP* command. The I/O K04 is used as an output in Fig 4.77 hence the value of the *active* and *keyb* parameter. Notice that there is no *POSN* command for the *KEY* command as the key is an external key not a 'touch' key in the screen. Note that there is no built-in debounce in iDev when an I/O line is used as an input that is triggered by an interrupt e.g. K05 is set as an input line with the interrupt set every time falling edge is detected. The developer would have to use this example function in iDevt to implement a debounce (see Chapter 10 for definition).

```
//FILENAME: TU480a.mnu


SETUP(KEYIO)          //
{
active = \\00000020; //
inp = \\00000020; //
trig = \\00000020; //
edge = \\00000000; //
}
VAR(curiostate,0,U8);
VAR(prviostate,0,U8);
VAR(iostate,0,U8);
VAR(lasttime,0,FLT4);
VAR(debouncedel,100,U8);
STYLE(homepgst,Page)
{
back = white;
}

LOOP(debouncelp,FOREVER)
{
LOAD(iostate,K05);
IF(iostate != prviostate?LOAD(lasttime,CNTK05);
CALC(cntcalc,CNTK05,lasttime, "-");
IF(cntcalc > debouncedel?[LOAD(curiostate,iostate);LOAD(curiostate,K05);
LOAD(prviostate,iostate);
}
```

Fig. 4.78 *Example code showing how button debounce is implemented in iDev (incomplete)*

5. CONTROLLING PWM, ADC AND PIEZO BUZZER

The itron SMART TFT module has built-in 3 PWM controllers, 2 ADC ICs and a Buzzer controller. All these built-in peripherals are configured using the *SETUP* command.

5.1. PWM

PWM (Pulse Width Modulation) is a method used to vary and control analogue devices based on the digital outputs e.g. controlling the brightness of an LED. The 3 PWM controllers are located in CN4 (PW1 and PW2) and CN7 (PW3). The PWM controllers on the TFT module are outputs only; it cannot be used to detect PWM levels from external devices. The voltage level of the PWM output is fixed at 3.3V for all module sizes and versions. The other (not PWM) Digital I/O lines in CN4 and CN7 are still usable even though the same connectors are used.

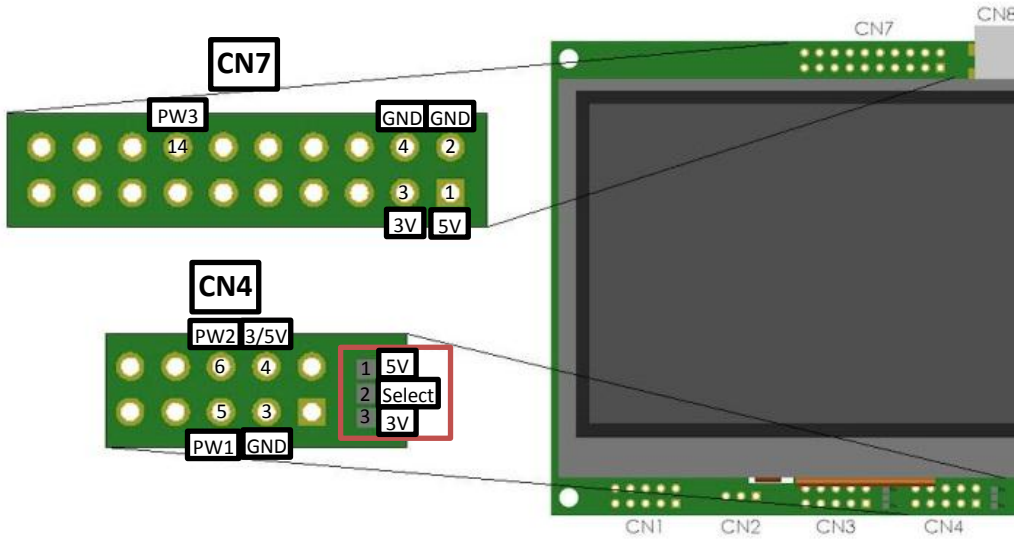


Fig. 5.1 *Diagram to show pin assignments of the AS2 interface in the TFT module with the jumper link highlighted*

The jumper link highlighted in red is used to set the voltage output levels on pin 4 in connector 4 (**CN4**) to either 5V or 3.3V. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions do not have this jumper link have the voltage level input/output at pin 4 in connector 4(**CN4**) fixed at 5V.

PWM3(CN7) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	5V	5V power	Input/Output
2	GND	Common Ground	Input/Output
3	3V	3.3V (Output) power	Output
4	GND	Common Ground	Input/Output
14	PW3	PWM 3 Controller	Output
PWM1 and PWM2 (CN4) Pin Assignment Definition			
3	GND	Common Ground	Input/Output
4	3/5V	3.3V (Output) or 5V (Input or Output) power	Input/Output
5	PW1	PWM1 Controller	Output

6	PW2	PWM2 Controller	Output
Jumper Link (Highlighted in RED) for PWM1 and PW2 CN4			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 4 in **CN4**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3V output at pin 4 in **CN4**

Fig. 5.2 *Table describing the pin and pad assignments of the PWM controller*

The settings for the PWM outputs can be altered using the *SETUP* command in iDev. The *SETUP* command is similar on how it was used in Chapter 1.5 where a system setup is created. It contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (PWM)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

PWM Setup parameters		
Parameter	**Expected Values**	**Definition**
active	123	• enable PWM1,PWM2, and PW3 and synchronise them
	N	• disable all PWM controllers **(default)**
active1	Y or N	• enable (Y) or disable (N) PWM1 output **(default = N)**
active2	Y or N	• enable (Y) or disable (N) PWM2 output **(default = N)**
active3	Y or N	• enable (Y) or disable (N) PWM3 output **(default = N)**
pol1	H or L	• set the polarity to High (H) or Low (L) on first phase of PWM1 **(default = L)**
pol2	H or L	• set the polarity to High (H) or Low (L) on first phase of PWM2 **(default = L)**
pol3	H or L	• set the polarity to High (H) or Low (L) on first phase of PWM3 **(default = L)**
cycle1	1µs(1MHz) to 5000µs(200Hz)	• set the duration of each cycle in microseconds for PWM1 **(default = 1)**
cycle2	1µs(1MHz) to 5000µs(200Hz)	• set the duration of each cycle in microseconds for PWM2 **(default = 1)**
cycle3	1µs(1MHz) to 5000µs(200Hz)	• set the duration of each cycle in microseconds for PWM3 **(default = 1)**
duty1	1% to 99%	• set the duty cycle (see Chapter 10 for definition) value of first phase as a percentage for PWM1 **(default = 1)** • duty cycle values less than 1 are forced to 1 and values greater than 99 are forced to 99 to prevent a DC condition

duty2	1% to 99%	• set the duty cycle value of first phase as a percentage for PWM2 **(default = 1)** • first phase as a percentage for PWM2 • duty cycle values less than 1 are forced to 1 and values greater than 99 are forced to 99 prevent a DC condition
duty3	1% to 99%	• set the duty cycle value of first phase as a percentage for PWM3 **(default = 1)** • first phase as a percentage for PWM3 • duty cycle values less than 1 are forced to 1 and values greater than 99 are forced to 99 prevent a DC condition
delay	0µs to 5000µs	• set the offset/delay between the first phase of PWM1 and PWM2 and first phase of PWM2 and PWM3 in microseconds, for visualisation look at Fig 5.4 **(default = 0)**

Fig. 5.3 *Table defining the PWM controller's setup parameters*

The *cycle* parameter it **not** the frequency of the cycle but the **duration** hence the expected value is in µs and not in Hz. The *duty* parameter is the ratio between the on and off states in one cycle. The diagram below describes how each parameter corresponds to each PWM cycle.

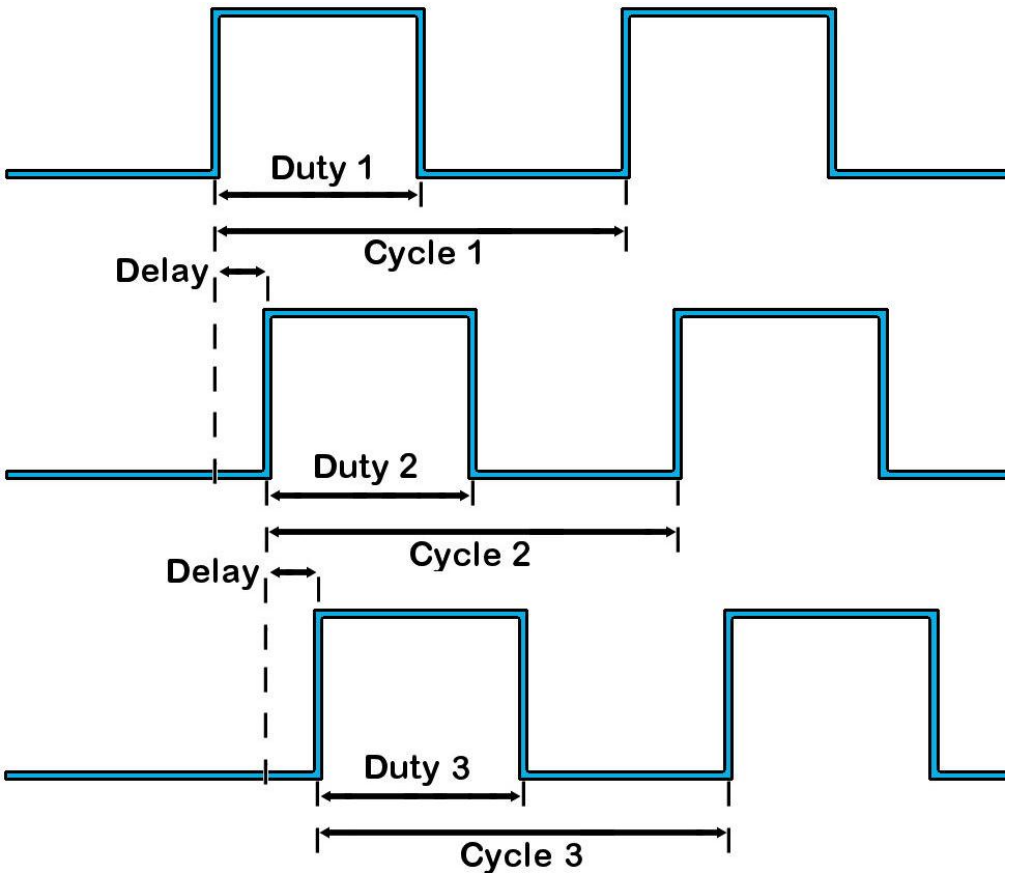


Fig. 5.4 *Diagram to show the relationship of the delay, duty and cycle parameters for PWM1, PWM2 and PWM3*

```
//FILENAME: TU480a.mnu

SETUP(PWM)      //
{
active1 = Y;    //
active2 = Y;    //
pol1 = H;       //
pol2 = H;       //
cycle1 = 300;   //
cycle2 = 300;   //
duty1 = 25;     //
duty2 = 50;     //
}
```

Fig. 5.5 *Example code showing how PWM settings are configured using the SETUP command*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updpwm1func)     //
{
LOAD(PWM.cycle1,25;   //
LOAD(PWM.duty1,76;    //
LOAD(PWM.pol,"L";     //
}
```

Fig. 4.66 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for PWM*

There is an example code named PWM Demonstration (link here) that uses all the PWM controllers. A slide bar is used to control the duty cycle and arrows are used to change the frequency of each cycle. The example code uses a combination of iDev commands to implement the changes in the PWM controllers. The *CALC* command is used to convert the duration of the cycle to frequency.

5.2. ADC

An ADC (Analogue to Digital Converter) is a device in which a signal that is continuously varying is sampled and converted to a digital form e.g. a device producing analogue voltage levels is converted to a digital number that is proportional to the scale of the voltage being measured. The Itron SMART TFT module has 2 built-in ADC ICs (Integrated Circuit) which are capable of converting an **input voltage range of 0V to 3V DC**. The input signal is sampled at 1000 Hz (1000 samples per second). The highest resolution of the ADC is at 10 bit i.e. digital range produced is 0 to 1024.

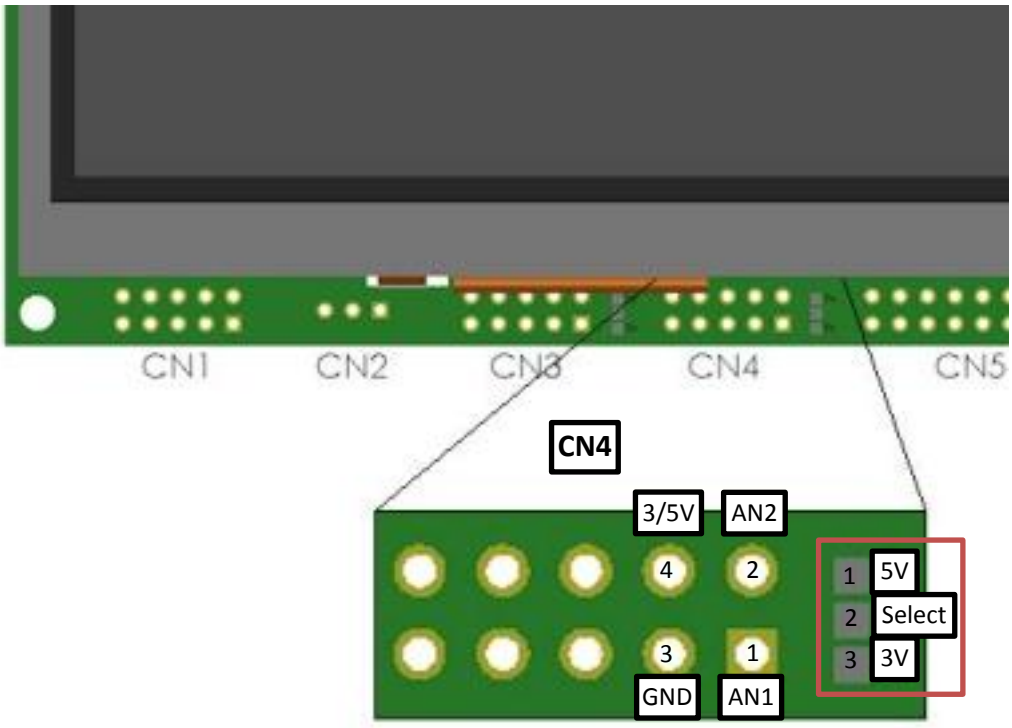


Fig 5.7 *Diagram to show pin assignments of the ADC in CN4 of the TFT module with the jumper link highlighted*

The jumper link highlighted in red is used to set the voltage output levels on pin 4 in connector 4 (**CN4**) to either 5V or 3.3V. This applies to all the module sizes with the similar configuration in the diagram but some TFT versions that do not have this jumper link have the voltage level at pin 4 in connector 4(**CN4**) fixed at 5V.

ADC (CN4) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	AN1	ADC1 with input voltage range 0V to 3V DC max with the reference voltage filtered from 3.3V DC	Input
2	AN2	ADC2 with input voltage range 0V to 3V DC max with the reference voltage filtered from 3.3V DC	Input
3	GND	Common Ground	Input/Output
4	3/5V	3.3V (Output) or 5V (Input/Output) power	IntputOutput

Jumper Link (Highlighted in RED) for Digital I/O in CN4			
Pad Number	**Pad Assignment**	**Definition**	**Link**
1	5V	5V Select	Solder to Select (pad 2) for 5V input/output at pin 4 in **CN4**
2	Select	Select	N/A
3	3V	3.3V Select	Solder to Select (pad 2) for 3.3V output at pin 4 in **CN4**

Fig 5.8 *Table describing the pin assignments of the ADC in CN4*

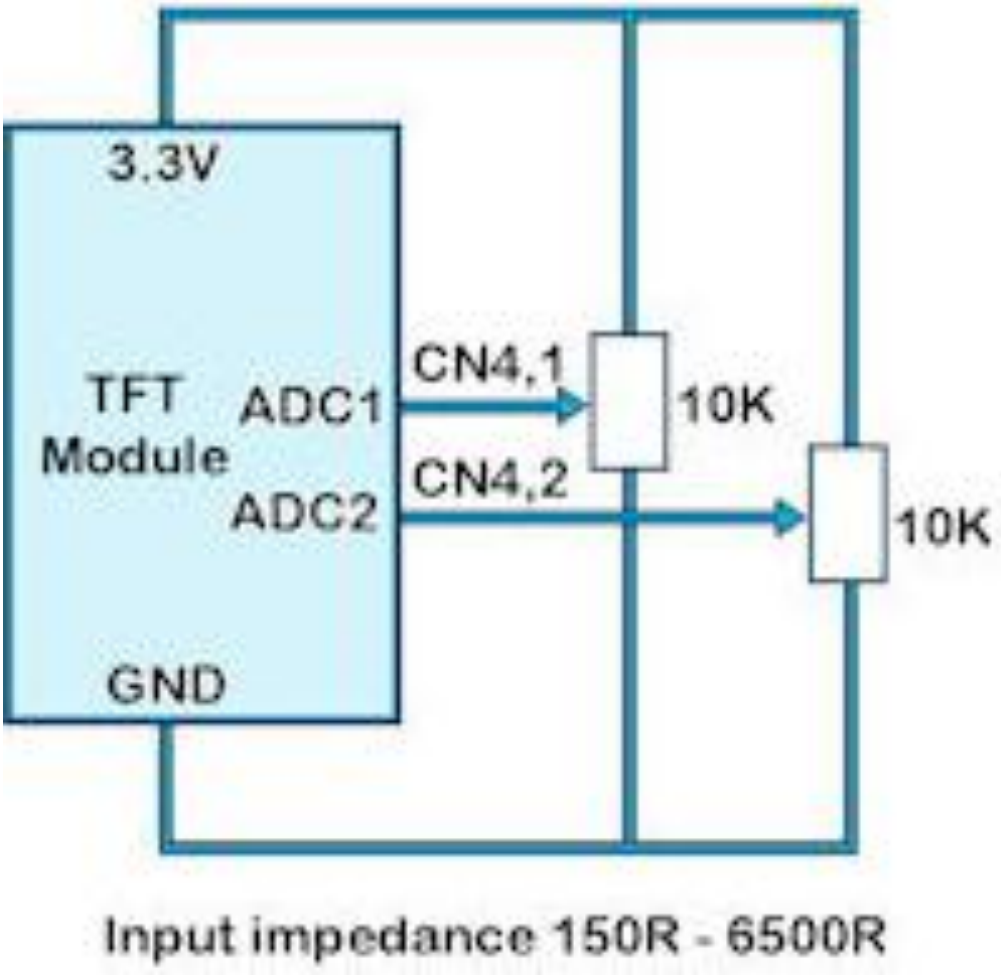


Fig. 5.9 *Diagram showing a typical connection for the ADC inputs*

This diagram is a general representation on how to connect devices to the ADC lines of the TFT module. Some connections may vary and the values also depend on the device's input impedance. The settings for the ADC outputs can be altered using the *SETUP* command in iDev. The *SETUP* contains a *Setup Header* and *Setup Body.*

SETUP command format for any interface used in iDev:
Setup Header
SETUP (ADC)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

ADC setup parameters		
Parameter	**Expected Values**	**Definition**
active	12	• enable ADC1 and ADC2
	1	• enable ADC1 and disable ADC2
	2	• enable ADC2 and disable ADC1
	N	• disable ADC1 and ADC2 **(default)**
calib1	any non-zero floating point value	• set the value used for calibration and scaling of the digital value produced for ADC1 **(default = 1)** • value produced is 1023 × calib1 value = new maximum digital range e.g. 1023 × 0.2 = 204.6 • new value used is 204, value is rounded down
calib2	any non-zero floating point value	• set the value used for calibration and scaling of the digital value produced for ADC2 **(default = 1)** • value produced is 1023 × calib2 value = new maximum digital range e.g. 1023 × 0.2 = 204.6 • new value used is 204, value is rounded down
avg1	1 to 1000	• set the number of samples read and averaged for ADC1, this is **not the sampling rate** • 1ms × avg1 value = average samples taken **(default = 16)**
avg2	1 to 1000	• set the number of samples read and averaged for ADC2, this is **not the sampling rate** • 1ms × avg2 value = average samples taken **(default = 16)**

Fig. 5.10 *Table defining the ADC setup parameters*

```
//FILENAME: TU480a.mnu

SETUP(ADC)      //
{
active = 12;    //
calib1 = 0.4;   //
calib2 = 0.85;  //
avg1 = 150;     //
avg2 = 300;     //
}
```

Fig. 5.11 *Example code showing how ADC settings are configured using the SETUP command*

The structure of the *SETUP* command is similar to the *STYLE* command in iDev. Specific setup parameter's value can be changed or updated using the *LOAD* command dot operator, similar on how specific style parameters are updated.

LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);

```
//FILENAME: TU480a.mnu

FUNC(updadc1func)      //
{
LOAD(ADC.active,2;     //
LOAD(ADC.calib1,2.5;   //
LOAD(ADC.avg1,500;     //
}
```

Fig. 5.12 *Example code demonstrating how the LOAD command with the dot operator is used to update setup parameters for ADC*

The ADC1 input line is used in a graph project named '25 Samples/Second ADC1 Graph Project' (link here) that maps the input voltage to a range of 0 to 3.2V. The digitised value of voltage produced is used to trace points in a voltage against time graph. The current value at different points is also calculated and displayed. The graph colour can be changed by touching the 3 colour boxes in the corner.

5.3. PIEZO

The user can attach a piezo sounder with integrated oscillator or similar low ripple device to provide an audible output or drive an LED indicator. The piezo line in **CN2** is connected to a 30V FET switching to 0V with maximum 200mA.

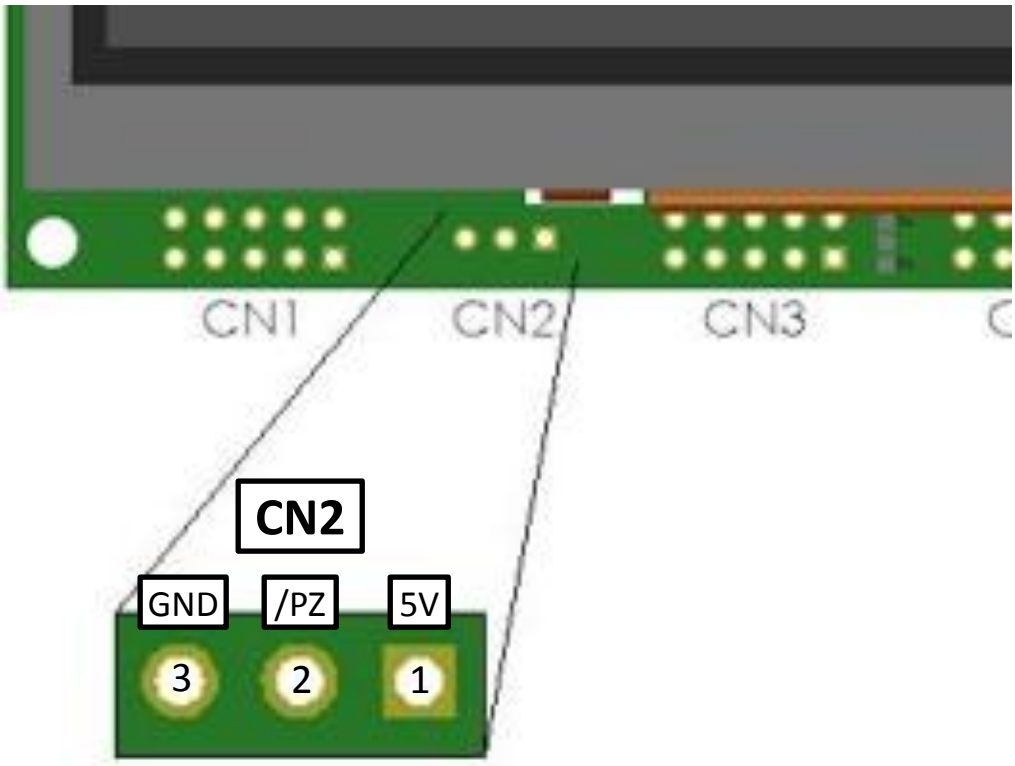


Fig. 5.13 *Diagram to show pin assignments for the Piezo output in CN2*

Piezo (CN2) Pin Assignment Definition			
Pin Number	**Pin Assignment**	**Definition**	**Input/Output**
1	5V	5V power, preferred pin to power the TFT module	Input/Output
2	/PZ (Active Low)	the negative terminal of the device (buzzer) should be connected to this pin	Output
3	GND	Common Ground	Input/Output

Fig. 5.14 *Table describing the pin assignments of the Piezo output in CN2*

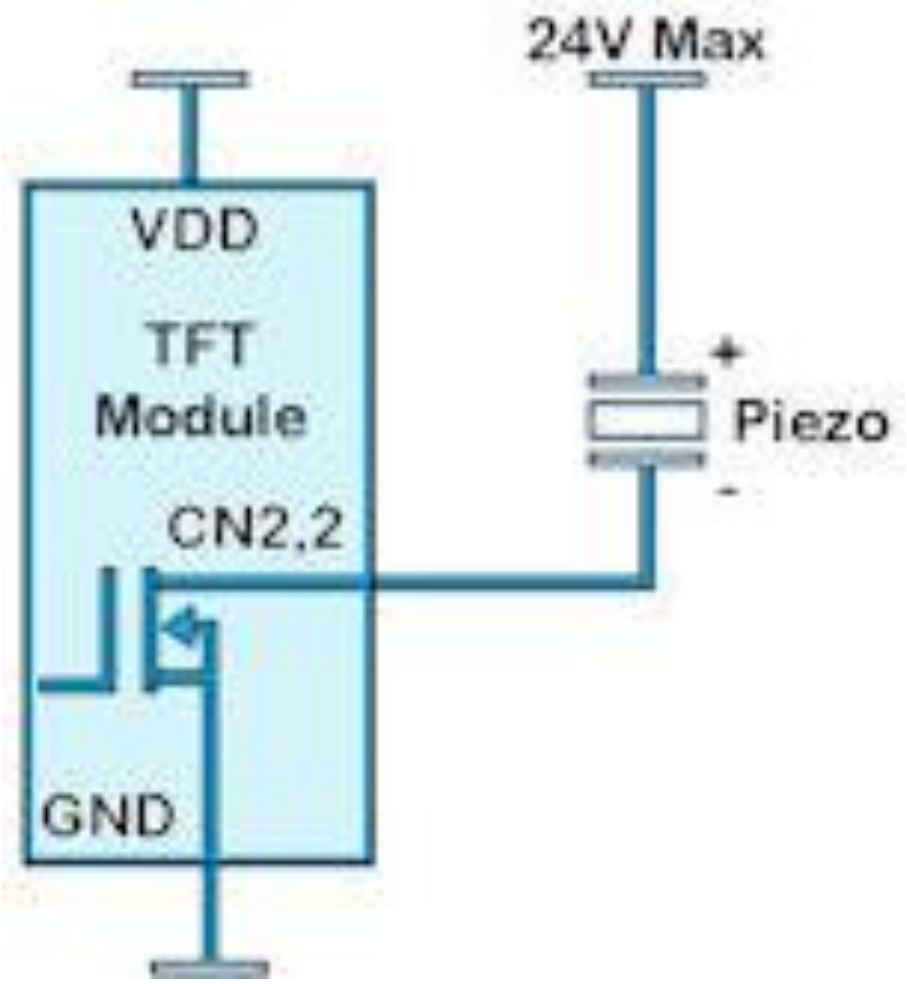


Fig. 5.15 *Diagram showing a typical connection for the Piezo output*

The negative terminal of the device (buzzer) should be connected to the TFT module and the positive terminal to a supply from 5V to 24V DC. Using the LOAD command and the reserved interface word BUZZ controls the Piezo output of the TFT module.

LOAD command to turn the Piezo output ON:
LOAD(BUZZ, ON/OFF);
LOAD command to turn the Piezo output to a specified duration value (in ms) or a value in variable:
LOAD(BUZZ, Duration Value/Variable);

```
//FILENAME: TU480a.mnu

FUNC(buzzfunc)    //
{
LOAD(BUZZ,500);   //
WAIT(1000);       //
LOAD(BUZZ,OFF);   //
}
```

Fig. 5.16 *Example code demonstrating how the Piezo output is manipulated*

6. Real Time Support (RTC and RTA)

6.1. Real Time Clock (RTC)

The RTC requires a battery to be fitted to the read of the module or a 3V DC supply applied via a connector fitted to the rear of the PCB. The RTC is a special type of data that is used to handle time. The *STYLE* command is used to control the different attributes for the RTC. The other data style parameters are the same; refer to Chapter 3.1.1 for better guidance. The default format used is 14 Sep 2012 09:50:04 which can be modified to suit the application. The values for the RTC are changed and manipulated using the *LOAD* command.

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
…
}

RTC format available in iDev	
Format style parameter expected values	**Definition**
d	RTC format for day: day of month with leading zeros = 01-31 **(default = disabled)**
j	RTC format for day: day of month without leading zeros = 1-31
S	RTC format for day: ordinal suffix for day of month = st, nd, rd, th
D	RTC format for day: short textual representation of the day in three letters = Mon-Sun
L	RTC format for day: full textual representation of the day = Monday-Sunday
N	RTC format for day: ISO-8601 numeric representation of the day = 1(Monday), 2(Tuesday)… 7(Sunday)
F	RTC format for month: full textual representation of month = January-December

m	RTC format for month: numeric representation of month with leading zeros = 01-12
M	RTC format for month: short textual representation of month with three letters = Jan-Dec
n	RTC format for month: numeric representation of month without leading zeros = 1-12
Y	RTC format for year: full numeric representation of year with 4 digits = 1900-2099
y	RTC format for year: two digit representation of year = 00-99
a	RTC format for time: lowercase ante meridiem and post meridiem = am, pm
A	RTC format for time: uppercase ante meridiem and post meridiem = AM, PM
g	RTC format for time: 12-hour format of hour without leading zeros = 1-12
G	RTC format for time: 24-hour format of hour without leading zeros = 0-23
h	RTC format for time: 12-hour format of hour with leading zeros = 01-12
H	RTC format for time: 24-hour format of hour with leading zeros = 00-23
i	RTC format for time: format of minutes with leading zeros = 00-59
s	RTC format for time: format of zeros = 00-59

Fig. 5.17 *Table describing different RTC formats that can be used in the STYLE command*

The format of RTC is normally used as a combination of different time attributes. The attributes omitted such as the day of the week doesn't affect what is displayed on the screen.

Format examples	
Format	**Displayed as**
"d M Y H :i:s"	14 Sep 2012 9:50:06
"d/m/y"	14/09/12
"jS F Y g:ia"	14th September 2010 9:50am

Fig. 5.18 *Example of different format parameter values for RTC*

LOAD command to '**read'** the current RTC :
LOAD(Variable, RTC);
LOAD command format to '**set'** RTC using 24-hour time with fixed format:
LOAD(RTC, "YYYY:MM:DD:hh:mm:ss");

Format for the LOAD command to 'set' RTC		
Parameter	**Expected Values**	**Definition**
YYYY	1900 to 2099	set the year
MM	01 to 12	set the month
DD	01 to 31	set the day of month
hh	00 to 23	set the hours
mm	00 to 59	set the minutes
ss	00 to 59	set the seconds

Fig. 5.19 *Table describing the parameters to set RTC using LOAD command*

There are predefined-variables in iDev that can be used to 'read' but **not** 'set' current attributes of the RTC. The value of these variables can be passed into a text variable to be used.

Built-in RTC variables	Definition
RTCYEARS	numeric variable containing year (1900-2099) which can be tested or loaded into a text
RTCMONTHS	numeric variable containing months (1-12) which can be tested or loaded into a text
RTCWEEKDAY	numeric variable containing day of the week (1=Monday-7=Sunday) which can be tested or loaded into a text
RTCDAYS	numeric variable containing days (1-31) which can be tested or loaded into a text
RTCHOURS	numeric variable containing hours (0-23) which can be tested or loaded into a text
RTCMINS	numeric variable containing year (0-59) which can be tested or loaded into a text
RTCSECS	numeric variable containing year (0-59) which can be tested or loaded into a text

Fig. 5.20 *Table describing the pre-defined variables for different RTC attributes*

A really basic example that used the RTC in iDev is created. The screen shot below shows what should be expected when the example code is uploaded on the TFT module.

Fig 5.21 *Screen shot of the TFT module when the code in Fig 5.22 is uploaded*

STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
VAR command format for text/RTC variable:
VAR(Variable name, "Starting text/RTC value", Variable Style);
LOAD command format to '**set**' RTC using 24-hour time with fixed format:
LOAD(RTC, "YYYY:MM:DD:hh:mm:ss");
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body Contents
{
POSN command format:
POSN(x coordinate, y coordinate);
TEXT command format:
TEXT(Text component name, "Text component", Text Style);
LOOP command format:
Loop Header
LOOP(Loop name, Loop duration)
Loop Body
{
LOAD command to '**read**' the current RTC:
LOAD(Variable/Interface Name, RTC);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
}
}
SHOW command format:
SHOW(Page name or page component name);

```
//FILENAME TU480a.mmu

STYLE(myRTCst,Data)        //
{
type = text;               //
length = 32;               //
format = "jS F Y g:i:sa"; //
}
STYLE(homepgst,Page)       //
{
back = white;              //
}
STYLE(Ascii32txst,Text)    //
{
font = Ascii32;            //
col = red;                 //
maxRows = 1;               //
maxLen = 32;               //
}

VAR(myRTCvar,"",myRTCst);           //
LOAD(RTC,"2012:09:12:10:21:52");    //

PAGE(homepg,homepgst)               //
{
POSN(240,130);                      //
TEXT(mytime,myRTCvar,Ascii32txst); //
LOOP(updatetimelp,FOREVER) //
{
LOAD(myRTCvar,RTC);      //
TEXT(mytime,myRTCvar);; //
}
}
SHOW(homepg);
```

Fig. 5.22 *Example code demonstrating how RTC is used in iDev*

The example code is a basic project that displays the current time and date. The starting RTC value is set in the code using the *LOAD* command. The developer can use a variable defined to display page that allows the user to change the current stored time and date (RTC). This method would only work provided that the user has stated variables named *yearvar, monthcvar, hourvar, minvar, secvar* in their code. The user can change the value stored in these variables by the use of 'plus or minus buttons' and a *SAVE* 'button' would use this *LOAD* command. There is a fully functional clock example project named 'Analogue Clock Project' (link here) that uses the RTC in iDev properly. The user can change and set the current time and also set an alarm.

LOAD command using variables to allow user to change RTC stored:
LOAD(RTC, yearvar,":",monthvar, ":",dayvar, ":",hourvar, ":",minvar, ":"secvar);

6.2. Real Time Clock Alarm (RTA)

The RTA is used to set the duration, time or time and date of the alarm. The RTA can be configured using the same format to 'set' RTC. The RTA does not support the *years* parameter and is ignored when setting the alarm. An alarm can be set every 20 seconds at 17:45 every day or on the 15th March at 12:52 each year.

LOAD command to '**read'** the current RTA:
LOAD(Variable, RTA);
LOAD command format to '**set'** RTA using 24-hour time with fixed format:
LOAD(RTA,":MM:DD:hh:mm:ss");

Format for the LOAD command to 'set' RTA		
Parameter	**Expected Values**	**Definition**
MM	01 to 12	set the month
DD	01 to 31	set the day of month
hh	00 to 23	set the hours
mm	00 to 59	set the minutes
ss	00 to 59	set the seconds

Fig. 5.20 *Table describing the parameters to set RTA using LOAD command*

Only the populated values are used to set the alarm, therefore alarms can be set every minute, hour, hour:minute:second, day or month etc…

RTA manipulation example	
Example LOAD commands for RTA	**Definition**
LOAD(RTA,":5:26:14:7:03");	alarm is set every year on 26th of May at 14:07:03
LOAD(RTA,":::13:15:");	alarm is set to occur every day at 13:15:00
LOAD(RTA,":::",hourvar, ":",minvar, ":",secvar);	alarm is set to occur at *hourvar:minvar:secvar* where *hourvar,minvar,secvar* are variables that stored values for the appropriate RTA attributes
LOAD(RTA,":::::20");	alarm is set to occur every 20 seconds past the minute
LOAD(RTA,0);	clear the alarm stored
LOAD(RTA,":::::");	clear the alarm stored
LOAD(myalarm,RTA);	'**read**' the current alarm '**set**' by the user

Fig. 5.21 *Table demonstrating how RTA attributes are managed*

Similar to the RTC, RTA also has built-in variables to be able to access certain attributes of the RTA set. If the RTA has not been set then a value of '-1' is returned.

Built-in RTA variables	Definition
RTAMONTHS	numeric variable containing months (1-12) which can be tested or loaded into a text
RTADAYS	numeric variable containing days (1-31) which can be tested or loaded into a text
RTAHOURS	numeric variable containing hours (0-23) which can be tested or loaded into a text
RTAMINS	numeric variable containing year (0-59) which can be tested or loaded into a text
RTASECS	numeric variable containing year (0-59) which can be tested or loaded into a text

Fig. 5.22 *Table describing the pre-defined variables for different RTA attributes*

It is possible to create an interrupt triggered by the alarm (RTA) set; the function set is called at the alarm point.

INT command format to set an interrupt triggered by RTA:
INT(Interrupt Name, RTA, Function);

Since the day of the week is not supported in RTA then to set an alarm that triggers every Thursday at 16:00, the example underneath can be used

```
//FILENAME: TU480a.mnu

INT(RTA,alarmfunc);       //
LOAD(RTA,":::16:00:00"); //

FUNC(alarmfunc)           //
{
IF(RTCWEEKDAY != 4? [EXIT(alarmfunc);]);  //
//insert function contents to be processed for Thursday here..
}
```

Fig. 5.23 *Code template to set an alarm triggered interrupt that occurs on the day set*

There is a fully functional clock example project named ‘Analogue Clock Project’ (link here) that uses the RTA in iDev properly. The user can change and set the current time and also set an alarm.

7. FILE HANDLING FOR SD/MICRO SD CARD AND NAND – FILE (INCOMPLETE)

These commands support the file input/output operations, such as file open, file read, file write, file close, file delete, etc. File and path names can be supplied either as immediate strings or via text variable entities. More complicated file names can be constructed with concatenation of text, strings, pointers, numbers etc... using the file *MKFN* command (see Chapter 7.8). Details on directory name and file name construction can be found below in File and Directory Names.

Also the use of the file object, used to maintain an association with an open "stream" when reading from and writing to the SD/SDHC card can be found below in File Object Variable. Data is read from and written to files in multiples of bytes. The order the data bytes are read/written and the conversion of the bytes (eg ASCII or binary) can be specified during setting up of the File Object Variable.

To allow the user to manage file error conditions without causing a system error, every *FILE()* command returns an optional file result (fileRes) of the file action, eg "File Not Found" or "Access Denied". This is returned as an error number or a configurable text string. See more about fileRes below.

FILE command format for general use:
FILE("Method",Parameter1, Parameter2, Parameter3…);

File command Methods Summary	
Method	**Definition**
APPEND	Append data to specified ‘filename’
CLOSE	close an open file on SD/micro SD card
COPY	copy a specified file
DATE	get the file time and date
DELETE	delete a specified file
EXISTS	check whether a file or directory exists
GETPOS	get the current read/write position in the file
MKFN	make a file name from a list of entities and strings
OPEN	open a file on SD/micro SD card
READ	read data from an open file on SD/micro SD card
READALL	read data from specified ‘filename’
RENAME	rename/move a file
SAVE	save media (e.g. image) to a file
SETPOS	set new/ read/write position in the file
SIZE	get file size
WRITE	write data to an open file on SD/micro SD card
WRITEALL	write data to specified ‘filename’

Fig. 7.1 *Different methods for the File command*

7.1. APPEND

Append data to a file on SD/micro SD card. Opens a file and adds the data to the end of the file then closes the file.

FILE command format for APPEND method:
FILE("APPEND",fileRes,fileObj,fileName,numWritten,numToWrite,data1,data2…);

FILE – APPEND parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable	File object variable See File Object Variable.
fileName	Immediate string or text variable of file name.	File name. See File and Directory Names
numWritten	Entity Name of numeric variable (U32 etc).	Number of bytes written to file **(optional parameter)**
numToWrite	Entity Name of numeric variable (U32 etc).	Maximum number of bytes to write to file **(optional parameter)**
data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	One of more sets of data to write to file

Fig. 7.2 *Table describing the FILE APPEND parameters*

```
//FILENAME: TU480a.mmu

//inside a function
FILE("APPEND",txtResvar,Fileobjvar,"SDHC/logs/log.txt",,,"Hello",Countvar,"\\0a");
//
FILE("APPEND",,FileObjvar,FileNamevar,numbytesvar,1024,Datavar);
//
```

Fig. 7.3 *Example code demonstrating how FILE APPEND is used*

7.2. CLOSE

Close an open file on SD/micro SD card.

FILE command format for CLOSE method:
FILE("CLOSE", fileRes, fileObj);

FILE – CLOSE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
fileObj	Entity name of file object variable	File object variable. See File Object Variable.

Fig. 7.4 *Table describing the FILE CLOSE parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("APPEND",txtResvar,Fileobjvar1); //
FILE("APPEND",,FileObjvar);           //
```

Fig. 7.5 *Example code demonstrating how FILE CLOSE is used*

7.3. COPY

Copy a file from SD/micro SD card to SD/micro SD card or between SD/micro SD card to NAND

FILE command format for COPY method:
FILE("COPY", fileRes, dstfileName, srcFileName);
FILE command format for COPY method with overwrite enabled:
FILE("COPY+O", fileRes, dstfileName, srcFileName);

FILE – COPY parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
dstfileName	Immediate string or text variable of file name.	Destination file name. See File and Directory Names.
srcFileName	Immediate string or text variable of file name.	Source file name. See File and Directory Names.

Fig. 7.6 *Table describing the FILE COPY parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("COPY",txtResvar,"NAND/redleaf.bmp","SDHC/images/greenleaf.bmp");
//
FILE("COPY+O",,DstNamevar,SrcNamevar);  //
```

Fig. 7.7 *Example code demonstrating how FILE COPY is used*

7.4. DATE

Get the file time and date of specified file in the SD/micro SD card or NAND.

FILE command format for DATE method:
FILE("DATE", fileRes, dstfileName, srcFileName);

FILE – DATE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
dst	Immediate string or text variable of file name.	Destination file name. See File and Directory Names.
format	Immediate string or text variable of format	Format of time and date. See Date Format. Default formatting will apply if not supplied. **(optional parameter)**
fileName	Immediate string or text variable of file name.	File name See File and Directory Names.

Fig. 7.8 *Table describing the FILE DATE parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("DATE",Resvar, Txtvar,"d M Y H:i:s","NAND/greenleaf.bmp");
//
FILE("DATE",,Txtvar,"Y",FileNamevar);  //
```

Fig. 7.9 *Example code demonstrating how FILE DATE is used*

7.5. DELETE

Delete a specified file in the SD/micro SD card or NAND.

FILE command format for DELETE method:
FILE("DELETE", fileRes, fileName);

FILE – DATE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
fileName	Immediate string or text variable of file name.	File name See File and Directory Names.

Fig. 7.10 *Table describing the FILE DELETE parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("DELETE",Resvar,"SDHC/greenleaf.bmp"); //
FILE("DELETE",,FileNamevar);                //
```

Fig. 7.11 *Example code demonstrating how FILE DELETE is used*

7.6. EXISTS

Check whether a file or directory exists in the SD/micro SD card or NAND.

FILE command format for EXISTS method:
FILE("EXISTS", fileRes, dst, fileName);

FILE – DATE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
dst	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file existence check. 0 = File/path does not exist. 1 = File/path exists. **(optional parameter)**
fileName	Immediate string or text variable of file/path name.	File name or path name. See File and Directory Names.

Fig. 7.12 *Table describing the FILE EXISTS parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("EXISTS",Resvar, Existsvar,"SDHC/music/jazzmusic.wav");
//
FILE("EXISTS",,Existsvar,FileNamevar);  //
```

Fig. 7.13 *Example code demonstrating how FILE EXISTS is used*

7.7. GETPOS

Get the current read/write position in the file opened on SD/micro SD card.

FILE command format for GETPOS method:
FILE("GETPOS", fileRes, fileObj, posn);

FILE – CLOSE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
fileObj	Entity name of file object variable	File object variable. See File Object Variable.
posn	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Current absolute read/write position from start of file.

Fig. 7.14 *Table describing the FILE GETPOS parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("GETPOS",txtResvar,Fileobjvar2,posnVar); //
FILE("GETPOS",,Fileobjvar2,posnVar);          //
```

Fig. 7.15 *Example code demonstrating how FILE GETPOS is used*

7.8. MKFN

Make a file name from a list of entities and strings specified.

FILE command format for APPEND method:
FILE("MKFN", fileRes, filename, data1, data2…);

FILE – MKFN parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileName	Immediate string or text variable of file name.	Destination for concatenated file name. See File and Directory Names.
data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	One of more sets of data to write to file

Fig. 7.16 *Table describing the FILE MKFN parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("MKFN",txtResvar,Filenamevar,"SDHC/",Dirvar,BaseNamevar,".txt");
//
FILE("MKFN",,Filenamevar,"Nand/file",Numvar,".bmp");
//
```

Fig. 7.17 *Example code demonstrating how FILE MKFN is used*

7.9. OPEN

Open a file on SD/micro SD card for read/write/append/overwrite. If the file does not exist then it is created. This *FILE* method is not supported in NAND. Use this command with a file object variable and *READ* or *WRITE* and *CLOSE,* if the developer needs to read or modify the file by small blocks of data. See Chapter 7.11, Chapter 7.17 and Chapter 7.1 for details of using *READALL, WRITEALL* and *APPEND* file command methods.

FILE command format for OPEN for Read method from start of file (same as OPEN+R):
FILE("OPEN",fileRes,fileObj,fileName,fileName…);
FILE command format for OPEN for Read method from start of file (same as OPEN):
FILE("OPEN+R",fileRes,fileObj,fileName,fileName…);
FILE command format for OPEN for Write method to truncate file to zero length:
FILE("OPEN+W",fileRes,fileObj,fileName,fileName…);
FILE command format for OPEN for Append method to start at end of file:
FILE("OPEN+A",fileRes,fileObj,fileName,fileName…);
FILE command format for OPEN for Overwrite method to overwrite from start of file:
FILE("OPEN+O",fileRes,fileObj,fileName,fileName…);

FILE – OPEN parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable	File object variable See File Object Variable.
fileName	Immediate string or text variable of file name.	File name to open. Can be a concatenation of strings See File and Directory Names

Fig. 7.18 *Table describing the FILE OPEN parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("OPEN+A",txtResvar,Fileobjvar,"SDHC/file.txt");
//
FILE("OPEN+A",txtResvar,Fileobjvar1,"SDHC/info/",BaseNamevar,".txt");
//
FILE("OPEN+W",,Fileobjvar3,FileNamevar);
//
```

Fig. 7.19 *Example code demonstrating how FILE OPEN is used*

7.10. READ

Read data from an open file on SD/micro SD card. The file must already be opened for read access using *FILE OPEN+R* method.

FILE command format for READ method:

FILE("READ",fileRes,fileObj,numRead,numToRead,data);

FILE – READ parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable.	File object variable. See File Object Variable.
numRead	Entity Name of numeric variable (U32 etc).	Number of bytes read from file. **(optional parameter)**
numToRead	Entity Name of numeric variable (U32 etc).	Maximum number of bytes to read from file. This parameter is optional. Either the size of data is read or all remaining bytes (whichever is smaller) if this omitted.
data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	Destination of bytes read in.

Fig. 7.20 *Table describing the FILE READ parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("READ",txtResvar,Fileobjvar1,,Datavar,1024");
//
FILE("MKFN",,Fileobjvar1,NumBytesvar,Datavar);
//
```

Fig. 7.21 *Example code demonstrating how FILE READ is used*

7.11. READALL

Read data from specified 'filename' on SD/micro SD card or NAND. Opens file, reads data, closes the file.

FILE command format for READALL method:
FILE("READALL",fileRes,fileObj,numRead,numToRead,data);

FILE – READALL parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable.	File object variable. See File Object Variable.
numRead	Entity Name of numeric variable (U32 etc).	Number of bytes read from file. **(optional parameter)**
numToRead	Entity Name of numeric variable (U32 etc).	Maximum number of bytes to read from file. This parameter is optional. Either the size of data is read or all remaining bytes (whichever is smaller) if this omitted.
data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	Destination of bytes read in.

Fig. 7.21 *Table describing the FILE READALL parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("READALL",txtResvar,Fileobjvar1,"SDHC/logs/log.txt",,Datavar,1024");
//
FILE("READALL",,Fileobjvar1,Filenamevar,NumBytesvar,Datavar);
//
```

Fig. 7.22 *Example code demonstrating how FILE READALL is used*

7.12. RENAME

Rename/move a file on SD/micro SD card or NAND from SD/micro SD card to SD/micro SD card or between SD/micro SD card and NAND.

FILE command format for RENAME method:
FILE("RENAME", fileRes, dstfileName, srcFileName);
FILE command format for COPY method with overwrite enabled:
FILE("COPY+O", fileRes, dstfileName, srcFileName);

FILE – COPY parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text.	Result of file operation. See File Result. **(optional parameter)**
dstfileName	Immediate string or text variable of file name.	Destination file name. See File and Directory Names.
srcFileName	Immediate string or text variable of file name.	Source file name. See File and Directory Names.

Fig. 7.23 *Table describing the FILE RENAME parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("RENAME",txtResvar,"NAND/redleaf.bmp","SDHC/images/greenleaf.bmp");
//
FILE("RENAME+O",,DstNamevar,SrcNamevar);  //
```

Fig. 7.24 *Example code demonstrating how FILE RENAME is used*

7.13. SAVE

Save media entity (image,audio,etc…) to a file on SD/micro SD card or NAND in the appropriate file format.

FILE command format for SAVE method and sends an error report if file exists:
FILE("SAVE", fileRes, fileObj, fileName, numWritten,numToWrite,data);
FILE command format for SAVE+O method and overwrites fie if it already exists:
FILE("SAVE+O", fileRes, fileObj, fileName, numWritten,numToWrite,data);

FILE – SAVE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable.	File object variable. See File Object Variable.
fileName	Immediate string or text variable of file name.	Destination file name, required if file not already open. SeeFile and Directory Names.
numWritten	Entity Name of numeric variable (U32 etc).	Number of bytes written to file. This parameter is optional.
numToWrite	Entity Name of numeric variable (U32 etc).	Maximum number of bytes to write to file. This parameter is optional.

data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	Destination of bytes read in.

Fig. 7.25 *Table describing the FILE SAVE parameters*

File extension types	
File extension	Definition
".tri"	TFT raw image TFT specific image file designed for fast loading time
".tra"	TFT raw audio TFT specific audio file designed for fast loading time
".trl"	TFT raw layer TFT specific layer file designed for fast loading time
".bmp"	R8G8B8 bitmap standard bitmap

Fig. 7.26 *Table describing custom TFT specific files and bitmap*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("SAVE",txtResvar,Fileobjvar1,"NAND/tree.tri",,,ImgTreelib");
//
FILE("SAVE+O",,Fileobjvar1,,,,SrcNamevar);
//
```

Fig. 7.27 *Example code demonstrating how FILE SAVE is used*

7.14. SETPOS

Set a new Read/Write position in the open file on SD/micro SD card.

FILE command format for SETPOS method to set new absolute read/write position (same as SETPOS+A):
FILE("SETPOS", fileRes, fileObj, actPosn, reqPosn);
FILE command format for SETPOS+A method to set new absolute read/write position (same as SETPOS):
FILE("SETPOS+A", fileRes, fileObj, actPosn, reqPosn);
FILE command format for SETPOS+R method to set new relative read/write position from current position:
FILE("SETPOS+R", fileRes, fileObj, actPosn, reqPosn);

FILE – SETPOS parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable.	File object variable. See File Object Variable.
actPosn	Entity Name of numeric variable (U8, U16 etc), text variable, text.	New read/write position after move - absolute position from start of file. This parameter is optional.

reqPosn	Immediate number or entity name of numeric variable (U8, S8, U16, S16 etc).	Required read/write position. For relative positions, negative numbers move the position towards the start of the file and positive numbers move the position towards the end of the file.

Fig. 7.25 *Table describing the FILE SETPOS parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("SETPOS",txtResvar,Fileobjvar1,,Posnvar);
//
FILE("SETPOS+R",,Fileobjvar1, NewPosnvar,-10);
//
```

Fig. 7.27 *Example code demonstrating how FILE SETPOS is used*

7.15. SIZE

Get the size in bytes of specified file on SD/micro SD card or NAND.

FILE command format for SIZE method:
FILE("SIZE", fileRes, dst, fileName);

FILE – MKFN parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
dst	Entity Name of numeric variable (U8, U16 etc), text variable, text	File size in bytes.
fileName	Immediate string or text variable of file name.	File name. See File and Directory Names.

Fig. 7.20 *Table describing the FILE SIZE parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("SIZE",txtResvar,Sizevar,"NAND/tree.bmp");
//
FILE("SIZE",,Sizevar,Filenamevar);
//
```

Fig. 7.21 *Example code demonstrating how FILE SIZE is used*

7.16. WRITE

Write data to an open file on SD/micro SD card. The file must already be opened for write/append/overwrite access using *OPEN+W, OPEN+A* or *OPEN+O.*

FILE command format for WRITE method:
FILE("WRITE", fileRes, fileObj, numWritten, numToWrite, data1, data2…);

FILE – SAVE parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**

fileObj	Entity name of file object variable.	File object variable. See File Object Variable.
numWritten	Entity Name of numeric variable (U32 etc).	Number of bytes written to file. This parameter is optional.
numToWrite	Entity Name of numeric variable (U32 etc).	Maximum number of bytes to write to file. This parameter is optional.
data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	One or more sets of data to write to file.

Fig. 7.25 *Table describing the FILE WRITE parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("WRITE",txtResvar,Fileobjvar1,,,"Hello",Countvar,"\\0ae");
//
FILE("WRITE",,Fileobjvar1,NumBytesvar,1024,Datavar);
//
```

Fig. 7.27 *Example code demonstrating how FILE WRITE is used*

7.17. WRITEALL

Write data to an open file on SD/micro SD card. The file must already be opened for write/append/overwrite access using *OPEN+W, OPEN+A* or *OPEN+O.*

FILE command format for WRITEALL method for files in SD/micro SD card:
FILE("WRITEALL", fileRes, fileObj, fileName, numWritten, numToWrite, data1, data2…);
FILE command format for WRITEALL method for files in NAND:
FILE("WRITEALL", fileRes, fileObj, fileName, numWritten, numToWrite, data);

FILE – WRITEALL parameters		
Parameter	**Expected Value**	**Definition**
fileRes	Entity Name of numeric variable (U8, U16 etc), text variable, text	Result of file operation See File Result. **(optional parameter)**
fileObj	Entity name of file object variable.	File object variable. See File Object Variable.
fileName	Immediate string or text variable of file name.	File name. See File and Directory Names.
numWritten	Entity Name of numeric variable (U32 etc).	Number of bytes written to file. This parameter is optional.
numToWrite	Entity Name of numeric variable (U32 etc).	Maximum number of bytes to write to file. This parameter is optional.
data	Immediate strings or numbers plus entity names of text or numeric variables or pointers to variables.	One or more sets of data to write to file. NAND only supports one data source.

Fig. 7.25 *Table describing the FILE WRITEALL parameters*

```
//FILENAME: TU480a.mnu

//inside a function
FILE("WRITEALL",txtResvar,Fileobjvar1,"SDHC/logs/log.txt",,,"Hello",Countvar,"\\0a");
//
FILE("WRITEALL",,Fileobjvar1,Filenamevar,NumBytesvar,1024,Datavar);
//
```

Fig. 7.27 *Example code demonstrating how FILE WRITEALL is used*

7.18. FILE OBJECT VARIABLE

When a file is to opened, read or modified in stages and then closed, the file system associates these actions with a "stream" that can identified in future actions by a file object. The user must create a unique file object variable for each open file, though the variable can be reused once a file has been closed.
You can create a file object variable with built in data styles FILEASC or FILEBIN.

```
//FILENAME: TU480a.mnu

VAR(name,0,FILEASC);  //name is a file object variable with data read/write as ASCII
VAR(name,0,FILEBIN);  //name is a file object variable with data read/write as
                      //binary (raw)
```

You can also create your own file styles to alter the way data bytes are read/written. The encode property determines how this data is managed. Refer to the data styles for the VAR() command for the options. The predefined styles are:

```
//FILENAME: TU480a.mnu

STYLE(FILEASC,data)   //predefined style for FILE handling
{
type = file;          //set type to file
encode = sr;          //store a U8 var = 12 as two byes \\31\\32
}

STYLE(FILEBIN,data)   //predefined style for FILE handling
{
type = file;          //set type to file
encode = sd;          //store a U8 var = 12 as one byte \\0C
}
```

The visibility property can be checked to see if the file object variable is currently assigned to a file stream. An assigned variable is "visible", and unassigned variable is not "visible".

CALC(res, varFileObj, "EVIS"); // res is 1 if varFileObj is in use.

7.19. FILE RESULT

Every FILE() command returns an optional file result (fileRes) of the file action. This allows the user to manage file error conditions without causing a system error. System errors will still occur for syntax errors and file system unrelated errors.
If fileRes is specified and is a numeric variable then an error number is returned. If fileRes is a text entity or text variable entity then a text string for the error is returned. These strings are preconfigured in text variable entities but can be changed by the user (eg to support different languages). The following table summarises the file results.

Number	Default Text String	Entity Name	Definition
0	"OK"	FILERES0	OK. No error - the function succeeded

1	"Disk I/O Layer Error"	FILERES1	Disk Error. An unrecoverable error occurred in the lower layer (disk I/O functions)
2	"Assertion Failed"	FILERES2	Internal Error. Assertion failed
3	"Physical Drive Error"	FILERES3	Not Ready. The physical drive cannot work
4	"File Not Found"	FILERES4	File Not Found. Could not find the file
5	"Path Not Found"	FILERES5	Path Not Found. Could not find the path
6	"Invalid Path Name"	FILERES6	Filename Error. The path name format is invalid
7	"Access Denied"	FILERES7	Access Denied. Access denied due to prohibited acce
8	"File Already Exists"	FILERES8	File Exists. Any object that has the same name is already existing
9	"Invalid File Object"	FILERES9	Not Object. The file/directory object is invalid
10	"Write Protected"	FILERES10	Write Protected. The physical drive is write protected
11	"Invalid Drive"	FILERES11	Invalid Drive. The logical drive number is invalid
12	"No Work Area"	FILERES12	Not Enabled. The volume has no work area
13	"No FAT Filesystem"	FILERES13	Not FAT. There is no valid FAT volume
14	"Make Filesystem Failed"	FILERES14	MKFS Aborted. (Not supported)
15	"Timeout"	FILERES15	Timeout. Could not get a grant to access the volume within defined period
16	"Locked Out"	FILERES16	Locked. The operation is rejected according to the file sharing policy
17	"LFN Buffer Overflow"	FILERES17	No Buffer. LFN working buffer could not be allocated
18	"Too Many Open Files"	FILERES18	Too Many Open Files. Number of open files > 5
19	"Invalid Parameter"	FILERES19	Invalid Parameter. Given parameter is invalid

The error strings in FILERES0 to FILERES19 are variables of type TXT and have a maximum length of 32 characters. They can be accessed and changed, as any other text variable. For example:

```
LOAD( FILERES4, "File was not found!" ); // Change default "File Not Found" message
TEXT( txtName, FILERES1 );; // Display the Disk Error message on the screen
```

7.20. File and Directory NAmes

SD/SDHC card

Supports long file names and unlimited directories with the following limitations:

Maximum file name length is 256 characters,
Maximum path name length is 8191 characters (including the SDHC/),
Directory depth is unlimited but must fit in pathname length.
The pound '£' symbol is not supported.

File names take the format:

"SDHC/dir1/dir2/longFileName.ext"

NAND

Supports 8.3 file names only. Subdirectories are not supported in the NAND.
File names take the format:
"NAND/filename.ext" for automatic placement in NAND (menu files in NANDMNU, others in NANDLIB).
"NANDLIB/filename.ext" for placement in NANDLIB area (where files are expected to be rarely updated).
"NANDMNU/filename.ext" for placement in NANDMNU area (where files are expected to be frequently updated).

7.21. Potential Future Commands (not yet supported)

These commands are not implemented but could be made available in future development depending on customer demand.

FILE("DIR", *[fileRes]*, dst, *[format]*, pathName); // Formatted directory listing
FILE("APPENDLN", *[fileRes]*, fileObj, fileName, *[numWritten]*, data *[, data [, ...]]*); // Open, Append Line, Close File (given fileName)
FILE("READLN", *[fileRes]*, fileObj, *[numRead]*, data *[, numToRead]*); // Read Line (given open fileObj)
FILE("STATUS", *[fileRes]*, fileName); // Get File Status
FILE("WRITELN", *[fileRes]*, fileObj, *[numWritten]*, data *[, data [, ...]]*); // Write Line (given open fileObj)
FILE("WRITELN", *[fileRes]*, fileObj, fileName, *[numWritten]*, data *[, data [, ...]]*); // Open, Write Line, Close File (given fileName)

7.22. File Examples

Example 1 - Simple Logging

```
VAR( varFileObj, 0, FILEASC ); // Create a file object variable
FUNC( fncLogStatus )
{
   FILE( "APPEND", varRes, varFileObj, "SDHC/logs/log1.txt", varTime, ": ", varTemperature1, "\\0a" );
   IF( varRes != 0 ? fncReportError );
}
```

Example 2 - Running a Log with Dated File Name

```
VAR( varFileObj, 0, FILEASC ); // Create a file object variable
VAR( varDay, -1, S16 ); // Variable to store time last log was made
VAR( varS32Tmp, 0, S32 ); // Variable for temporary storage
VAR( varTxtRes, "", TXT ); // Text variable for file result
VAR( varTxtTmp, "", TXT ); // Text variable for temporary storage
FUNC( fncTimerExpired )
{
```

```
   LOAD( varS32Tmp, RTCHOURS );
   IF( varDay != varS32Tmp ? fncOpenLogFile );
   FILE( "WRITE", varTxtRes, varFileObj, varLogData1, varLogData2, "\\0a" );
   IF( varTxtRes != "OK" ? [ LOAD( RS2, "Log File Write Error: ", varTxtRes, "\\0d\\0a" ); ] );
}
FUNC( fncOpenLogFile )
{
   LOAD( varDay, RTCHOURS );
   CALC( varS32Tmp, varFileObj, "EVIS" );
   IF( varS32Tmp == 1 ? [ FILE( "CLOSE", , varFileObj ); ] );
   FILE( "MKFN", varTxtRes, varTxtTmp, "SDHC/logs/log", varDay, ".txt" );
   IF( varTxtRes != "OK" ? [ LOAD( RS2, "Log File Name Error: ", varTxtRes, "\\0d\\0a" ); ] );
   FILE( "OPEN+W", varTxtRes, varFileObj, varTxtTmp );
   IF( varTxtRes != "OK" ? [ LOAD( RS2, "Log File Open Error: ", varTxtRes, "\\0d\\0a" ); ] );
}
INT( intTmr0, TIMER0, fncTimerExpired ); // Call function every minute
LOAD( TIMER0, 60000, 0 ); // Create a 1 minute repetitive timer
```

8. File Transfer and Memory

8.1. Transfer via micro SD Card

The most convenient way to transfer an iDev project is via the micro SD card. The TFT module supports micro SD cards with 1 GB capacity in FAT16 or FAT32 format and 4GB, 8GB, 16GB and 32GB capacity in FAT32 format. The reliability of the micro SD cards however depends on the brand and the class. The sizes 1GB, 4GB and 8GB all work reliably regardless of the micro SD card's brand and class. When a firmware update is required then a 1GB micro SD card must be used.

Fig. 8.1 *Image of a 1GB micro SD card inserted into CN9 of the TFT module*

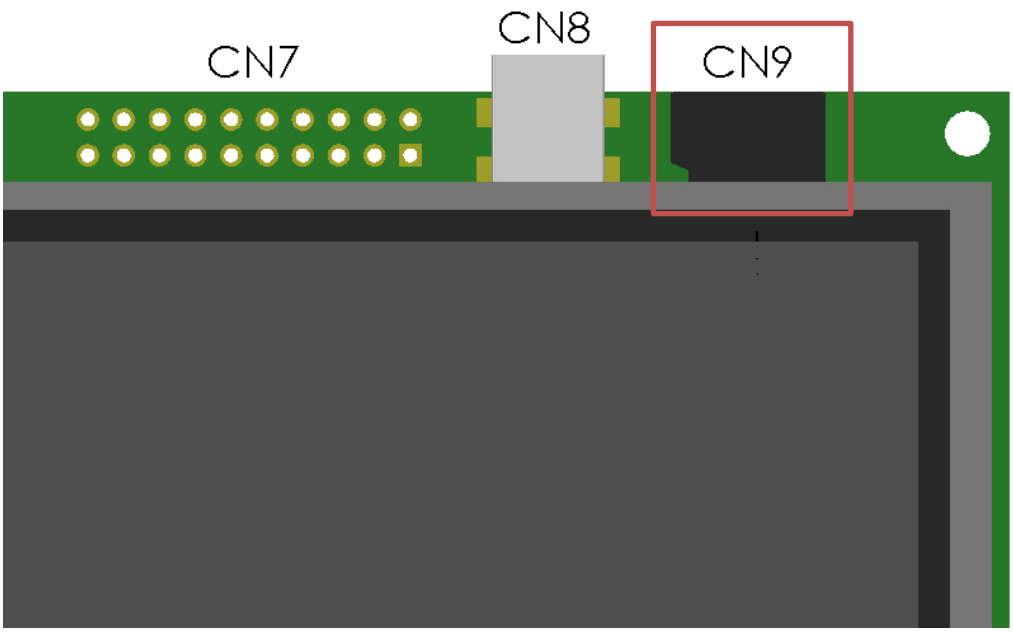


Fig. 8.2 *Picture of the front of a TFT module indicating the location of the micro SD card slot*

All Itron SMART TFT module sizes and versions have a micro SD card slot located in **CN9** which is on the top right hand corner (highlighted in red in Fig 8.1) of the TFT module. The main menu file (*TU320a.mnu*, *TU480a.mnu*, *TU640a.mnu* or *TU800a.mnu*), other menu files, images and other contents of an iDev project have to be copied in the root folder of the micro SD card. The filenames of each file can have a maximum of 8 characters and it must start with a **letter** or **_** (8.3 file naming system). Also the file type should have three characters e.g. image.bmp. The micro SD card slot in the TFT module works on a push (push to insert)-pull (pull to remove) operation.

8.2. TRANSFER VIA SD CARD OR MICRO SD CARD ADAPTOR

An SD card or micro SD card adaptor can be connected to the TFT module on **CN5** for all the module sizes except 3.5". The 3.5" TFT module does not have **CN5.** This option to transfer files is useful in cases where the TFT module is placed in an enclosure that restricts access to **CN9**, where the on-board micro SD card slot is located. The SD cards supported are 1GB (FAT16/FAT32) and 4GB, 8GB, 16GB and 32GB in FAT 32 format. The reliability of the micro SD cards however depends on the brand and the class. The sizes 1GB, 4GB and 8GB all work reliably regardless of the micro SD card's brand and class. When a firmware update is required then a 1GB micro SD card must be used. The project files have to be transferred to the root folder of the SD card which follows the 8.3 file naming system. Only one micro SD card or SD card can be connected to the TFT module at a time i.e. it is not possible to connect a micro SD card on **CN9** and an SD card on **CN5** to transfer project files at the same time. The pin assignments for the SD card or micro SD card adaptor are exactly the same.

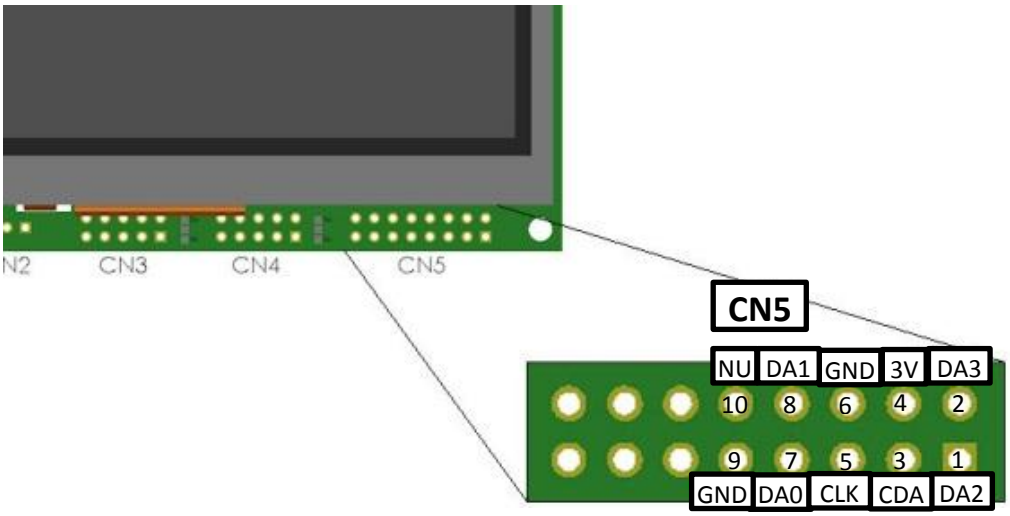


Fig. 8.3 *Diagram to show pin assignments for the SD card/micro SD card adaptor in CN5*

Pin Number	Pin Assignment	Definition
1	DA2	Data Line 2
2	DA3	Data Line 3
3	CDA	Serial Clock and Data
4	3V	3.3V power
5	CLK	Clock
6	GND	Common Ground
7	DA0	Data Line 0
8	DA1	Data Line 1
9	GND	Common Ground
10	NU	Not Used (Do not connect to anything)

Fig. 8.4 *Table defining the pin assignments for SD card/micro SD card adaptor*

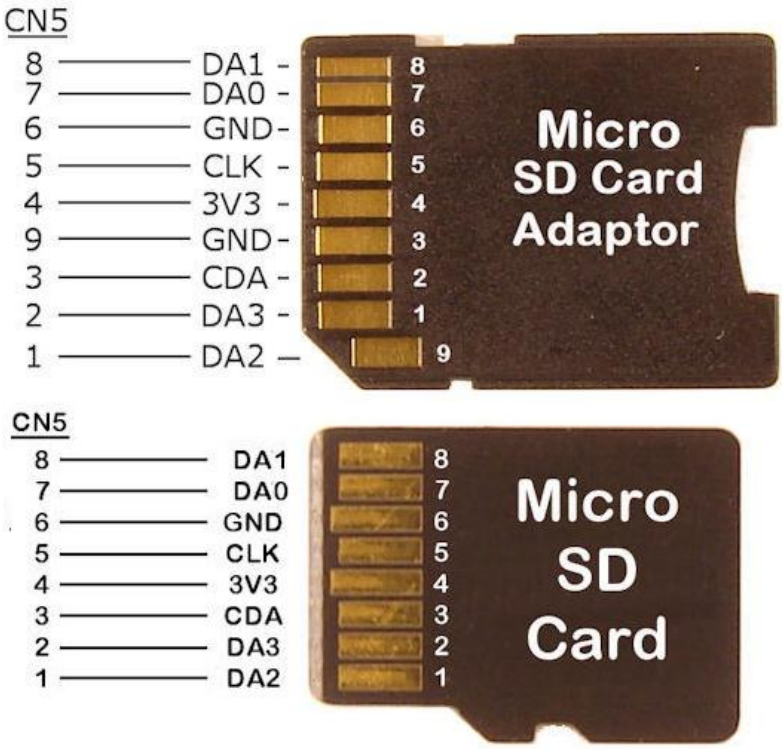


Fig. 8.5 *Diagram to display the pin connections of micro SD card adaptor and micro SD card to CN5*

A typical application to using this method uses an 800mm cable with connectors and cable stubs to evaluate the SD card reading which proved successful when uploading the 88 files of the demonstration software. It is recommended to use screened flat IDC cable of minimum length with a ferrite collar.

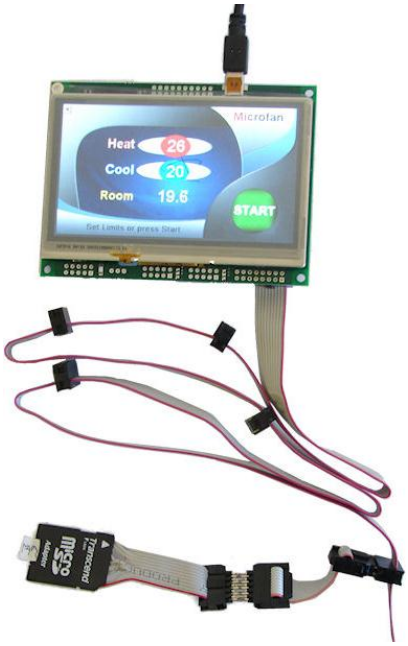


Fig. 8.6 *Image of the SD card adaptor connected via a long cable to CN5*

8.3. TRANSFER TO NAND – FPROG & LOAD

The TFT module has 128 Mbyte NAND flash memory which is organised into 3 drives. The first drive is a protected drive containing boot and operation files which use approximately 4 Mbyte and the other two drives for user accessible menu file drive and image/font/sound files with variable partition to allow large image and fonts to occupy the maximum space of 124 Mbytes. The second and third drive contents can be cleared using the *RESET(NAND)* command.

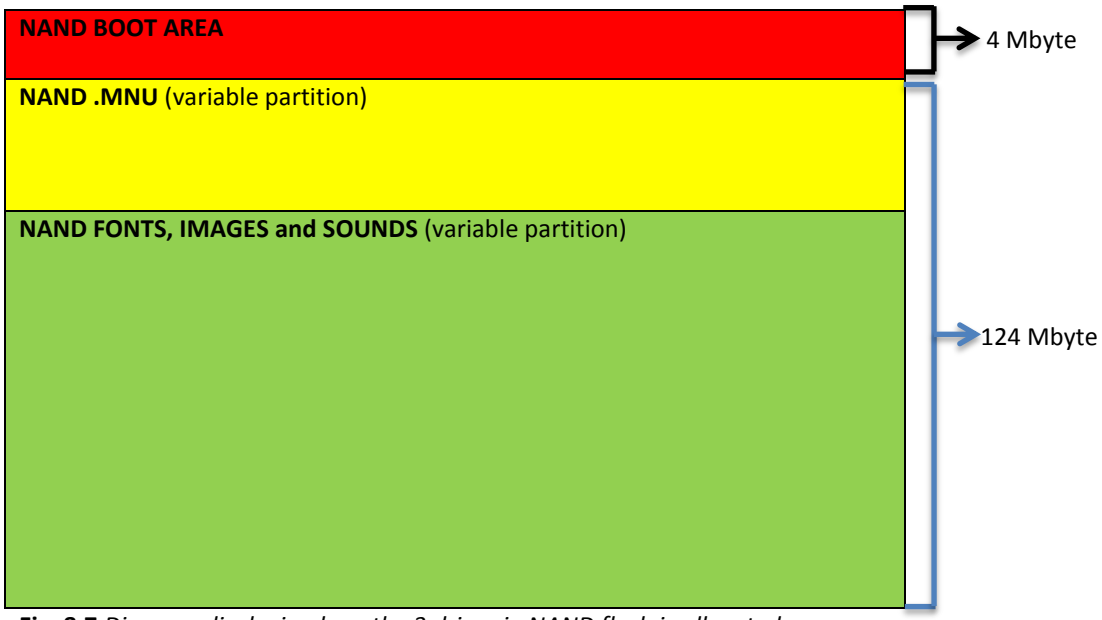


Fig. 8.7 *Diagram displaying how the 3 drives in NAND flash is allocated*

The files can be transferred to the NAND flash memory using the *FPROG* and *FEND* command. These commands are used to program subsequent commands into internal flash memory. If the existing files stored in NAND have to be replaced then the *RESET(NAND)* command have to be used after the *FPROG* command, otherwise the files are added to NAND. The subsequent *LIB* commands transfer images and files from NAND.

FPROG command to transfer:
FPROG;
LOAD(NAND, "SDHC/Filename1");
LOAD(NAND, "SDHC/Filename2");
LOAD(NAND, "SDHC/Filename3");
…
FEND;

If the files have been transferred to NAND, then the *source* parameter in all *LIB* commands that add the files to the iDev project have to be changed to *NAND* e.g. change *LIB(img1,"SDHC/imgfile1.bmp");* to *LIB(img1,"NAND/imgfile1.bmp");*. When the module is turned on, it checks for the correct TUXXXa.mnu file in NAND first then SDHC.

FPROG command to transfer:
FPROG;
LOAD(NAND, "SDHC/Filename1");
LOAD(NAND, "SDHC/Filename2");
LOAD(NAND, "SDHC/Filename3");
…
FEND;
RESET command format:
RESET(Name of iDev property)
INC command format:
For single files
INC("Source/Filename") ;
For multiple files
INC("Source/Filename1", "Source/Filename2", "Source/Filename3"…);
LIB command format for images:
LIB(Library image name, "Source/Filename");
LIB command format:
LIB(Library font/sound name, "Source/Filename");

```
//FILENAME: TU480a.mnu

FPROG;                          //start of FPROG command
RESET(NAND);                    //clear the contents of NAND memory
LOAD(NAND,"SDHC/TU480a.mnu");   //copy the TU480a.mnu file to NAND
LOAD(NAND,"SDHC/Funcs.mnu");    //copy the Funcs.mnu file to NAND
LOAD(NAND,"SDHC/Vars.mnu");     //copy the Vars.mnu file to NAND
LOAD(NAND,"SDHC/imgfile1.bmp"); //copy the imgfile1.bmp image to NAND
LOAD(NAND,"SDHC/imgfile3.bmp"); //copy the imgfile3.bmp image to NAND
LOAD(NAND,"SDHC/fntfile1.fnt"); //copy the fntfile1.fnt font to NAND
FEND;                           //end of FPROG command

INC("NAND/Funcs.mnu","NAND/Vars.mnu"); //add Func.mnu and Vars.mnu to iDev project

LIB(img1,"NAND/imgfile1.bmp");  //add imgfile1.bmp to library
LIB(img2,"SDHC/imgfile2.bmp");  //add imgfile2.bmp to library
LIB(img3,"NAND/imgfile3.bmp");  //add imgfile3.bmp to library
LIB(fnt1,"NAND/fntfile1.fnt");  //add fntfile1.bmp to library
LIB(fnt2,"SDHC/fntfile2.fnt");  //add fntfile2.bmp to library
```

Fig. 8.8 *Example code demonstrating how FPROG is used in iDev*

Once the project files are copied to the NAND then it is not necessary to keep the *FPROG* and *LOAD* commands anymore. However, if the iDev project is getting updated and changed via the micro SD card quite often then it is recommended to leave the *FPROG* and *LOAD* commands so that the necessary files are changed appropriately. Another way of utilising the *FPROG* command is by creating a separate menu file named *FPROG.mnu* and perform all the *FPROG and LOAD* commands there to copy the project files to the NAND memory.

```
//FILENAME: TU480a.mnu

INC("NAND/Fprog.mnu", "NAND/Funcs.mnu","NAND/Lib.mnu");
//add Fprog.mnu,Funcs.mnu,Lib,mnu to iDev project
```

Fig. 8.9 *Main menu file containing INC command to add the menu files to iDev project*

```
//FILENAME: Fprog.mnu

FPROG;                          //start of FPROG command
RESET(NAND);                    //clear the contents of NAND memory
//menu files
LOAD(NAND,"SDHC/TU480a.mnu");   //copy the TU480a.mnu file to NAND
LOAD(NAND,"SDHC/Funcs.mnu");    //copy the Funcs.mnu file to NAND
LOAD(NAND,"SDHC/Lib.mnu");      //copy the Lib.mnu file to NAND
//image files
LOAD(NAND,"SDHC/imgfile1.bmp"); //copy the imgfile1.bmp image to NAND
LOAD(NAND,"SDHC/imgfile2.bmp"); //copy the imgfile2.bmp image to NAND
LOAD(NAND,"SDHC/imgfile3.bmp"); //copy the imgfile3.bmp image to NAND
LOAD(NAND,"SDHC/imgfile4.bmp"); //copy the imgfile4.bmp image to NAND
LOAD(NAND,"SDHC/imgfile5.bmp"); //copy the imgfile5.bmp image to NAND
LOAD(NAND,"SDHC/imgfile6.bmp"); //copy the imgfile6.bmp image to NAND
LOAD(NAND,"SDHC/imgfile7.bmp"); //copy the imgfile7.bmp image to NAND
LOAD(NAND,"SDHC/imgfile8.bmp"); //copy the imgfile8.bmp image to NAND
LOAD(NAND,"SDHC/imgfile9.bmp"); //copy the imgfile9.bmp image to NAND
//font files
LOAD(NAND,"SDHC/fntfile1.fnt"); //copy the fntfile1.fnt font to NAND
LOAD(NAND,"SDHC/fntfile2.fnt"); //copy the fntfile2.fnt font to NAND
LOAD(NAND,"SDHC/fntfile3.fnt"); //copy the fntfile3.fnt font to NAND
LOAD(NAND,"SDHC/fntfile4.fnt"); //copy the fntfile4.fnt font to NAND
LOAD(NAND,"SDHC/fntfile5.fnt"); //copy the fntfile5.fnt font to NAND
LOAD(NAND,"SDHC/fntfile6.fnt"); //copy the fntfile6.fnt font to NAND
FEND;                           //end of FPROG command
```

Fig. 8.10 *Fprog.mnu file demonstrating the use of FPROG and LOAD commands*

```
//FILENAME: Lib.mnu

//image files
LIB(img1,"NAND/imgfile1.bmp");  //add imgfile1.bmp to library
LIB(img2,"NAND/imgfile2.bmp");  //add imgfile2.bmp to library
LIB(img3,"NAND/imgfile3.bmp");  //add imgfile3.bmp to library
LIB(img4,"NAND/imgfile4.fnt");  //add imgfile4.bmp to library
LIB(img5,"NAND/imgfile5.fnt");  //add imgfile5.bmp to library
LIB(img6,"NAND/imgfile6.fnt");  //add imgfile6.bmp to library
LIB(img7,"NAND/imgfile7.fnt");  //add imgfile7.bmp to library
LIB(img8,"NAND/imgfile8.fnt");  //add imgfile8.bmp to library
LIB(img9,"NAND/imgfile9.fnt");  //add imgfile9.bmp to library
//font files
LIB(fnt1,"NAND/fntfile1.fnt");  //add fntfile1.bmp to library
LIB(fnt2,"NAND/fntfile2.fnt");  //add fntfile2.bmp to library
LIB(fnt3,"NAND/fntfile3.fnt");  //add fntfile3.bmp to library
LIB(fnt4,"NAND/fntfile4.fnt");  //add fntfile4.bmp to library
LIB(fnt5,"NAND/fntfile5.fnt");  //add fntfile5.bmp to library
LIB(fnt6,"NAND/fntfile6.fnt");  //add fntfile6.bmp to library
```

Fig. 8.11 *Lib.mnu* file adding all the project files into the project library using the *LIB* command

It is possible to upload and write files to the NAND via a serial interface using the LOAD command format. The correct interface is automatically selected i.e. the interface that receives the *LOAD* command.

LOAD command format to upload and write files to NAND via serial interface:
LOAD("EXT/Filename?size=value&timedate=value&usechecksum=value&useack=value");
CR Filedata ChecksumH ChecksumL

Definition for LOAD command parameters		
Parameter	**Expected Values**	**Definition**
size	value in bytes (maximum is based on available RAM space)	specify the number of bytes to be written and uploaded **(required parameter)**
timedate	YYYY:MM:DD:HH:MM:SS in ASCII format	specify the timestamp of the file being sent **(optional parameter)**
usechecksum	1 or 0	enable checksum **(optional parameter)**
useack	1 or 0	enable acknowledge byte **(optional parameter)**
CR	HEX code	specify the carriage return
Filedata	HEX code/ASCII/Binary	list the contents of the file
ChecksumH	HEX code	specify the high nibble checksum value
ChecksumL	HEX code	specify the low nibble checksum value

Fig. 8.12 *Table describing the parameters for the LOAD command to upload/write files to NAND via serial interface*

All the TFT module versions and sizes have a RAM capacity of 64 MB. If *usechecksum =1*, then the TFT module will expect a 16 bit checksum added to the file data**.** The checksum uses the sum of the checksum itself and the file data. The checksum is validated and the file only written to NAND if correct when compared with the data received. On the other hand if *useack=1,* then the module will send a single byte (on the same serial interface as the command is being received on) to indicate success or failure of the upload/write operation. The value returned is either *06* (ACK) if successful or *15* (NAK) if a failure occurred. Note that the ACK is sent after the file is written to NAND therefore this can be used to inform the host to send the next file if multiple files are being sent.

An example to upload and write a 5-byte file called test.txt containing the values for the word 'hello' using *checksum* and *acknowledge* without the time stamp.

SETUP command format for any interface used in iDev:
Setup Header
SETUP (Interface)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
LOAD command format to send files to NAND via serial interface:
LOAD("EXT/Filename?size=value&timedate=value&usechecksum=value&useack=value");
CR Filedata ChecksumH ChecksumL

```
//FILENAME: TU480a.mnu

SETUP(RS2)       //specify the setup paramaters for RS2
{
baud = 9600;     //set the baud rate to 9600
rxi = C;         //set the receive interface as a command processing source
rxb = 9600;      //set the receive buffer to 8500
txi = Y;         //enable the transmit interface
txb = 9600;      //set the transmit buffer to 8500
encode = s;      //set the encoding of data to 8 bit ASCII
}

FUNC(writeNANDfunc)
{
LOAD("EXT/test.txt?size=5&usechecksum=1&useack=1"); //
0D hello 01 74  //
}
```

Fig. 8.13 *Example code to upload and write to NAND via serial interface using the LOAD command*

It is only possible to upload and write to NAND via serial interface in command mode *(rxi = C; txi = Y;)*. If the interface is in data mode *(rxi = Y;),* there will need to be provision in the user protocol to allow a user command to be received that will change the interface setup using, for example, *LOAD(RS2.rxi, C); LOAD(RS2.txi, Y);* in preparation for uploading the files.

8.4. TRANSFER VIA USB

The connector **CN8** allows users of USB enabled TFT modules to connect directly to a PC using a standard mini-B cable which effectively powers the TFT module. The iDev project files can be uploaded to the internal NAND memory or the micro SD card using terminal software or the iDEVTFT development environment (link here).

Fig. 8.14 *Image of the standard mini-B USB cable connector*

USB Enabled TFT module versions		
Module Size	**K611XXX**	**K612XXX**
3.5”	v2 onwards	v3 onwards
4.3”	v4 onwards	v7 onwards
5.7”	v3 onwards	v3 onwards
7”	v5 onwards	v6 onwards

Fig. 8.15 *Table describing which TFT module versions have internal USB device fitted*

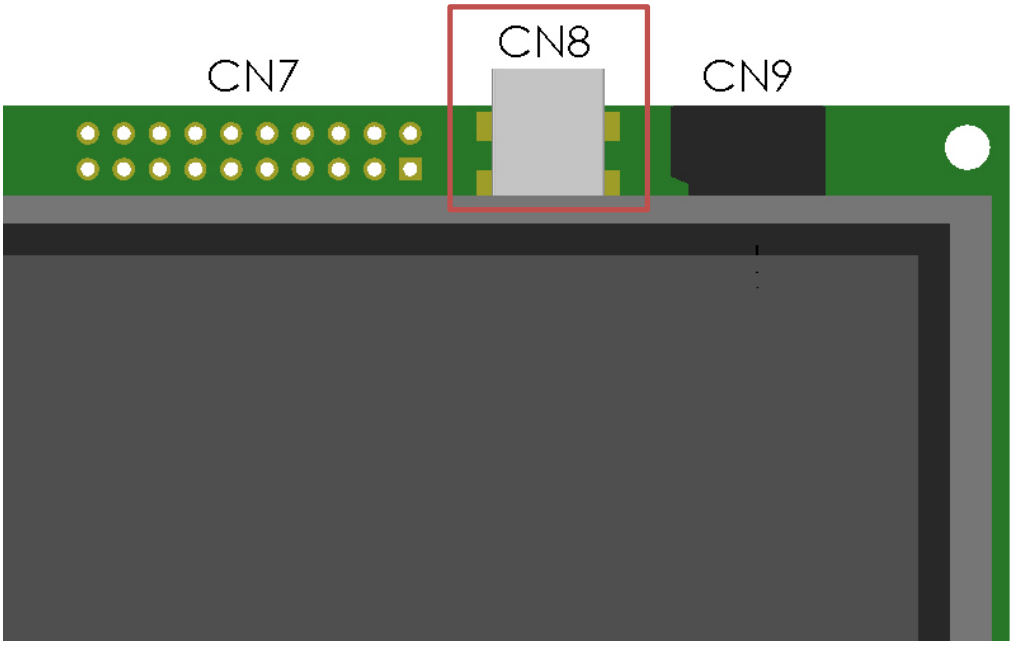


Fig. 8.16 *Picture of the front of a TFT module indicating the location of the internal USB device*

To transfer the project files from the PC to the internal NAND or micro SD card, the internal USB driver have to be installed first (link here). When the TFT module is connected to a PC, a

pop-up message usually appears to indicate that a new device has been detected. Download the USB *.INF* files applicable to the PC's OS and install as directed. There are 2 *.INF* files in *TUXXUSB1.ZIP*, extract the files and locate in an accessible directory on the PC. However, if the pop-up message did not appear, the internal USB device driver can be installed through the computer's 'Device Manager'. The current supported operating systems are Windows XP and Windows 7 in 32/64-bit. If a *TUXXXa.mnu* is not present in the NAND or micro SD card, the TFt module will enable USB communication automatically and driver installation can be achieved using the appropriate *.INF* file. It is possible that administrator rights may be required to install the USB driver. If there is a *TUXXXa.mnu* file in the NAND or micro SD card then setup command have to be included to configure the USB device port. The TFT module has an allocated pin connector for modules without an internal USB device fitted found in **CN5.** All the TFT module sizes and versions have **CN5** except the 3.5" size variant.

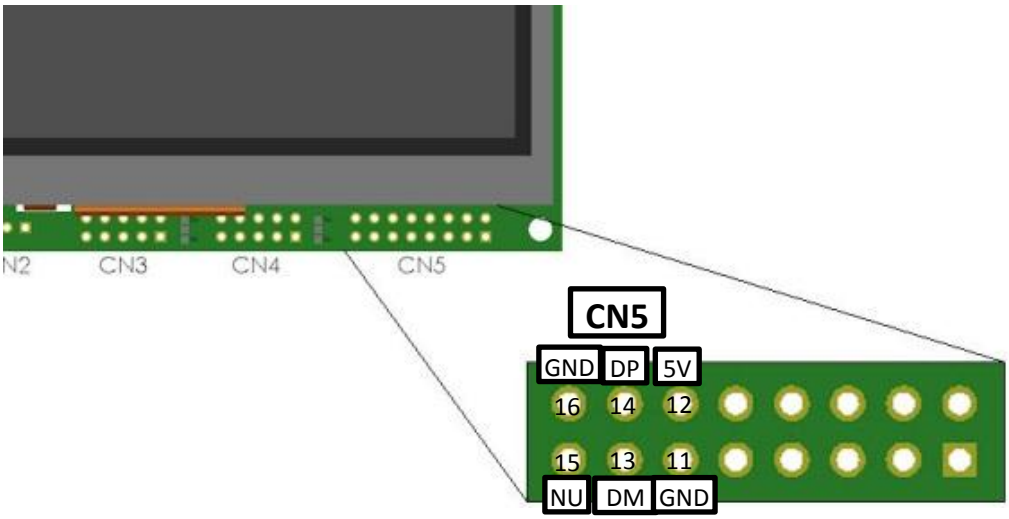


Fig. 8.17 *Diagram to show pin assignments for the USB device port in CN5*

USB port (CN5) Pin Assignment Definition		
Pin Number	**Pin Assignment**	**Definition**
11	GND	Common Ground
12	5V	5V power
13	DM	Data Minus (Data -)
14	DP	Data Plus (Data +)
15	NU	Not Used (Do not connect to anything, TFT module is always the USB Slave device)
16	GND	Common Ground

Fig. 8.18 *Table describing the pin assignments of the USB port in CN5*

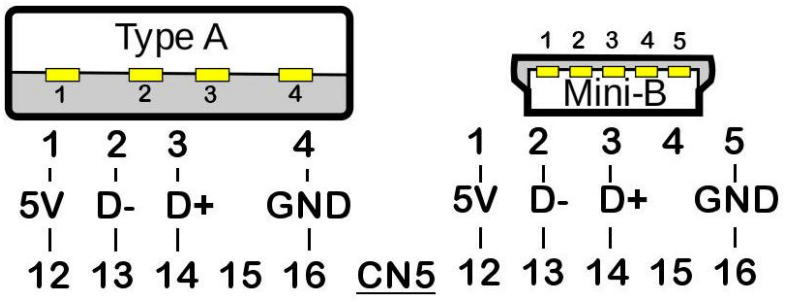


Fig. 8.19 *Diagram showing a typical connection between USB device port in CN5 and a standard Type A and Mini-B USB connector*

This diagram is a general representation on how to connect standard USB Type A and Mini-B devices to the USB port in **CN5** of the TFT module. The pin number 15 on **CN5** is not meant to be connected to anything since the TFT module only operates in USB Slave mode. The settings for the USB device port can be altered using the *SETUP* command in iDev. The *SETUP* contains a *Setup Header* and *Setup Body*.

SETUP command format for any interface used in iDev:
Setup Header
SETUP (USB)
Setup Body
{
setup parameter1 = setup value1;
setup parameter2 = setup value2;
setup parameter3 = setup value3;
…
}

USB port setup parameters		
Parameter	**Expected Values**	**Definition**
rxi	Y	• enable the receive interface of the USB device port
	N	• disable the receive interface of the USB device port
	C	• set the receive interface of the USB device port as a command processing source**(default)**
	D	• set the receive interface of the USB device port to receive debugging data
proc	• termination characters can be specified to generate an interrupt to process a string of data • NB when sending commands (in command mode where rxi = C) to the module, processing only occurs when \\0D or 0D hex is received e.g. TEXT(mytext, “hello world”);; \\0D	
	all;	• trigger on all received characters **(default)**
	CRLF;	• trigger on a carriage return (CR) followed by line feed (LF = 0Dh 0A)
	CR;	• trigger on carriage return (CR = 0dH) in command mode where rxi = C;
	LF;	• trigger on line feed (LF = 0Ah)
	NUL;	• trigger on NUL (00h)
	\\xx;	• trigger on xxh (specify hex value)
	"ABCD";	• string in format defined by SYSTEM encode parameter
	"\\xx\\xx";	• string in format defined by SYSTEM encode parameter
procDel	Y or N	• keep (Y) or remove (N) the termination character(s) before processing **(default = N)**
procNum	0 to 16,780,000	• interrupt on n bytes received as alternative to proc and procDel parameters **(default = 0)**
rxb	1 to 16,780,000	• specify the size of the receive buffer in bytes **(default = 8192 bytes)**
txi	Y	• enable the transmit interface of the USB device port
	N	• disable the transmit interface of the USB device port **(default)**
	C	• set the transmit interface of the USB device port as a command processing source

	E	• set the transmit interface of the USB device port to echo command processing mode
txb	1 to 16,780,000	• specify the size of transmit buffer in bytes **(default = 8192 bytes)**
encode	• set the data encode mode to suit the purpose of the USB device port	
	s	• set data encode to 8 bit ASCII data • codes 00-1F and 80-FF are converted to ASCII "\\00" - "\\1D" and "\\80" - "\\FF" respectively **(default)**
	sr	• set data encode to 8 bit ASCII raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as ASCII+ data
	sd	• set data encode to 8 bit ASCII raw data bytes • all bytes are processed as raw data
	w	• set data encode to UNICODE – HEX Char x 4 = U16 (Most significant hex-pair first) "ABCD" -> \\ABCD
	wr	• set data encode to 8 bit UNICODE raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UNICODE+ data??
	wd	• set data encode to 8 bit UNICODE raw data bytes • all bytes are processed as raw data
	m	• set data encode to 8 bit UTF8 raw text bytes • codes 00-07 are processed as cursor commands • codes 20-FF are processed as UTF8+ data??
	mr	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	md	• set data encode to 8 bit UTF8 raw data bytes • all bytes are processed as raw data
	D8M	• set data encode to 8 bit data with U16's, U32's etc output **most** significant byte first • the same as data encode sd
	D8L	• set data encode to 8 bit data with U16's, U32's etc output **least** significant byte first
	D16M	• set data encode to 16 bit data with bytes processed as **most** significant byte first • interrupt occurs after two bytes • the same as data encode wd
	D16L	• set data encode to 16 bit data with bytes processed as **least** significant byte first • interrupt occurs after two bytes
	D32M	• set data encode to 32 bit data with bytes processed as **most** significant byte first • interrupt occurs after four bytes • the same as data encode md
	D32L	• set data encode to 32 bit data with bytes processed as **least** significant byte first • interrupt occurs after four bytes
	sh or h8m or h8l	• set data encode to ASCII – HEX Char x 2 = U8 e.g. "A8" -> \\A8
	h16m	• set data encode to ASCII – HEX Char x 4 = U16 (**most** significant hex-pair first) e.g. "ABCD" -> \\ABCD
	h16l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) e.g. "ABCD" -> \\CDAB
	h32m	• set data encode to ASCII – HEX Char x 4 = U32 (**most** significant hex-pair first) "12345678" -> \\12345678

	h32l	• set data encode to ASCII – HEX Char x 4 = U16 (**least** significant hex-pair first) "12345678" -> \\78654321

Fig. 8.20 *Table defining the USB port setup parameters*

```
//FILENAME: TU480a.mnu

SETUP(USB)    //
{
rxi = C;      //
rxb = 9600;   //
txi = Y;      //
txb = 9600;   //
encode = s;   //
}
```

Fig. 8.21 *Example code showing how USB port settings are configured using the SETUP command*

8.5. EEPROM

The EEPROM (Electrically Erasable Programmable Read-Only Memory) is a non-volatile memory used to store data that is saved when the device is powered of e.g. variable values. The internal EEPROM of all the TFT modules has 7.5 kbytes of user space and 500 bytes for system parameters such as touch screen calibration and screen orientation. Data variables can be created for storage in EEPROM with the use of *VAR* command (see Chapter 3.1.1). The data stored in EEPROM are protected by checksums and in the event of corruption, the default value assigned to the variable will be used. It may be necessary to clear the EEPROM with the command *RESET(EEPROM);*. After this, the default parameters will be applied for the touch screen calibration and orientation. The process of installing a new boot file *(boot.bin)* also clears the EEPROM.

Fig. 8.7 *Diagram displaying how internal EEPROM is allocated in iDev*

9. EXAMPLE CODES (INCOMPLETE)

These projects are found folder called ‘Chapter 9 Example Code’

animation waterfalls
ball cursor
cursor animation
d-pad
d-pad bottles
PWM example
random graph
scroll bars
toggle switches

10. GLOSSARY

;

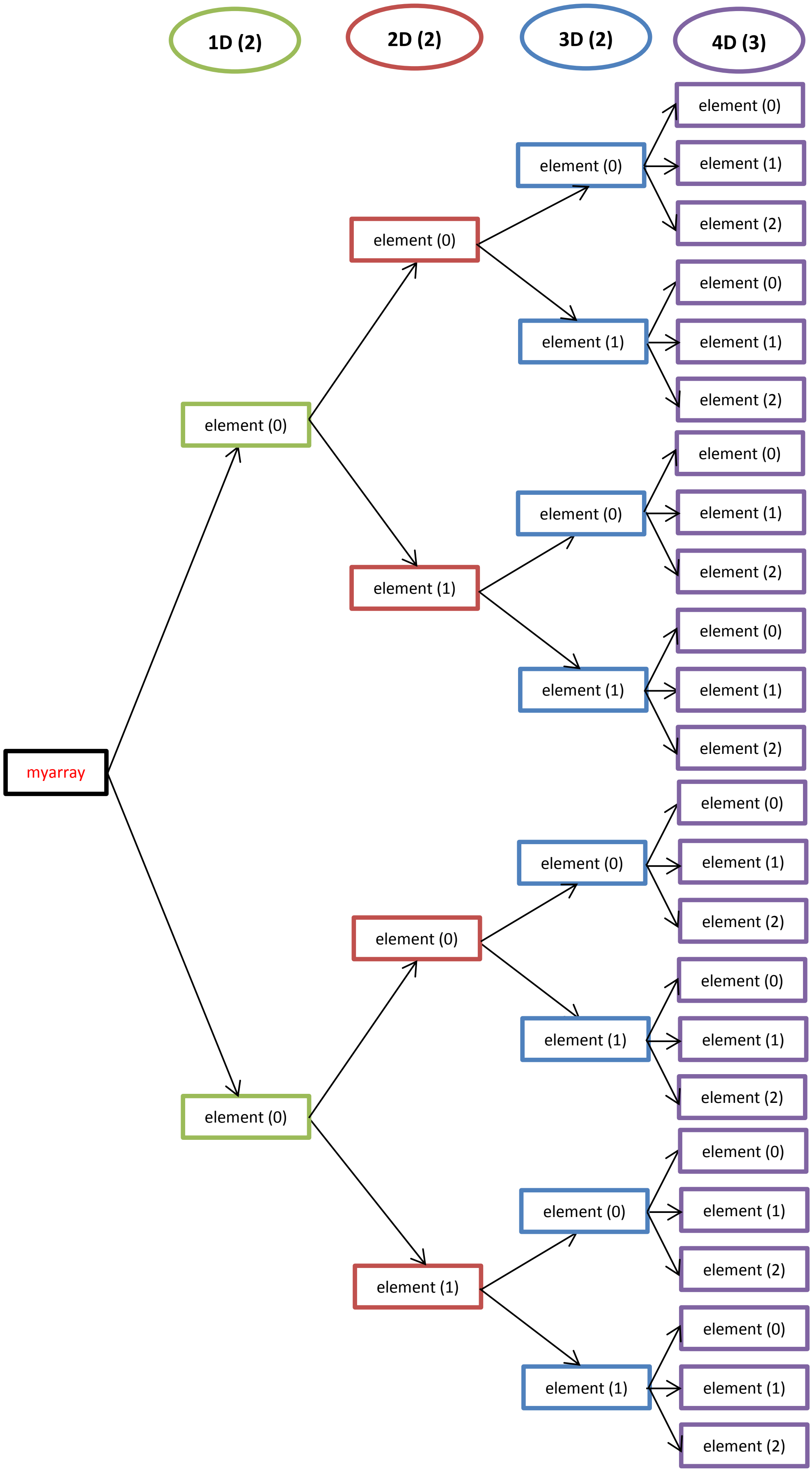
The command termination character ';' is used to signify the end of a command in iDev.

Append

The process of adding or joining data to the end of a pre-existing data e.g. the data 'world' is appended to 'hello', resulting in the data 'helloworld'

Array

In iDev, an array refers to a series of variables all of which are the same type. Each 'item' in an array is called an array element. It is possible to create 1D, 2D, 3D and 4D arrays in iDev. The diagram on the next page represents how array elements are organised in a 4D array.


1D (2)
2D (2)
3D (2)
4D (3)
myarray
element (0)
element (1)
element (2)

Asynchronous communication

The process of data transmission between devices that uses separate clock signals (independent transmit and receive clocks). This method requires a start and stop bit to signify the byte timing of the data. In iDev the AS1, AS2, DBG, RS232 and RS422/RS485 interfaces uses asynchronous communication.

Baud rate

Baud rate refers to the amount of times per second a signal changes state or varies in an interface channel, baud rate in iDev is calculated by this formula reg = ((clk freq/baud rate)+8)/16) where reg is the real baud rate value that the module's processor uses

Buffer

A buffer refers to a variable which is used to temporarily hold data before it is moved to its final destination. For example, the buffer variable *mybuff* is used as a temporary storage for the result of the *CALC(mybuff,20,18,"++);.* The contents of the buffer can then be loaded to another variable or its final destination.

Clock signal

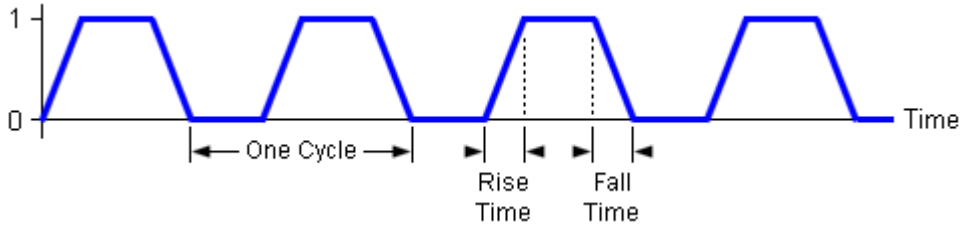
A signal that oscillates between a Logic HIGH and Logic LOW signal back and forth at a specific frequency. The clock signal ensures that the data stream between devices is synchronised.

Concatenate

This is the process of joining multiple data or strings end to end. For example in iDev the string 'I', 'am', 'strong' are concatenated using the *LOAD* command to end up with 'Iamstrong'.

Cycle

This term is used to describe the full signal from the rising edge (going HIGH), through time when the value of clock is (HIGH), through the falling edge (going LOW), the time that the value is zero until the start of the next rising edge.



Debounce

In electronic circuits pressing switches/buttons does not always provide a clean state switch. That is, when the switch is pressed the contacts of the switch does not make a clean contact to change the state but it 'bounces'. This 'bounce' creates an indeterminate state of the switch where it can be detected LOW when it's actually HIGH and vice versa. However, the 'debounce' phenomenon can be avoided in iDev.

DC signal

Refers to Direct Current signal which is either always positive or always negative but the magnitude can vary in time. Most electronic devices (such as the Itron SMART TFT modules) normally require a steady DC supply which is constant at one value.

Duplex
A type of communication operation whereby devices can act as transmitter and receiver at the same time (transceiver). Duplex communication allows data flow in both directions, hence verification and control of data reception/transmission is possible.

Duty cycle
The duty cycle is the proportion of the 'on' state to the 'off' state of the PWM signal. It is usually represented in percentage. A low duty cycle corresponds to low power as the power is off for most of the time and vice-versa

Floating point
A floating point is used to represent real numbers that requires significant accuracy. This allows values to have more decimal points, thus providing more accuracy

Full-duplex
A device that can transmit and receive data at the **same time** is a Full-duplex device. The RS232 and RS422 interface operate in Full-duplex mode.

Half-duplex
A device that can transmit and receive data but **not at the same time** is a Half-duplex device. The RS485 interface operates in Half-duplex mode.

Handshaking
The term 'handshaking' in programming is used to describe the process of one device establishing a successful connection with another device through a specific interface. This usually involves connection verification, speed, acknowledging etc…

HEX code
HEX code or Hexadecimal code is a base 16 notational system for representing numbers. The digits used are 0, 1, 2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,0, A, B, C, D, E and F. There 256 numbers that can be represented using Hexadecimal. The diagram on the next page is a conversion table between Hex, Decimal, Binary and Octal numbering systems.

Conversion Table – Decimal, Hexidecimal, Octol, Binary

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
0	0	000	00000000	16	10	020	00010000	32	20	040	00100000	48	30	060	00110000
1	1	001	00000001	17	11	021	00010001	33	21	041	00100001	49	31	061	00110001
2	2	002	00000010	18	12	022	00010010	34	22	042	00100010	50	32	062	00110010
3	3	003	00000011	19	13	023	00010011	35	23	043	00100011	51	33	063	00110011
4	4	004	00000100	20	14	024	00010100	36	24	044	00100100	52	34	064	00110100
5	5	005	00000101	21	15	025	00010101	37	25	045	00100101	53	35	065	00110101
6	6	006	00000110	22	16	026	00010110	38	26	046	00100110	54	36	066	00110110
7	7	007	00000111	23	17	027	00010111	39	27	047	00100111	55	37	067	00110111
8	8	010	00001000	24	18	030	00011000	40	28	050	00101000	56	38	070	00111000
9	9	011	00001001	25	19	031	00011001	41	29	051	00101001	57	39	071	00111001
10	A	012	00001010	26	1A	032	00011010	42	2A	052	00101010	58	3A	072	00111010
11	B	013	00001011	27	1B	033	00011011	43	2B	053	00101011	59	3B	073	00111011
12	C	014	00001100	28	1C	034	00011100	44	2C	054	00101100	60	3C	074	00111100
13	D	015	00001101	29	1D	035	00011101	45	2D	055	00101101	61	3D	075	00111101
14	E	016	00001110	30	1E	036	00011110	46	2E	056	00101110	62	3E	076	00111110
15	F	017	00001111	31	1F	037	00011111	47	2F	057	00101111	63	3F	077	00111111

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
64	40	100	01000000	80	50	120	01010000	96	60	140	01100000	112	70	160	01110000
65	41	101	01000001	81	51	121	01010001	97	61	141	01100001	113	71	161	01110001
66	42	102	01000010	82	52	122	01010010	98	62	142	01100010	114	72	162	01110010
67	43	103	01000011	83	53	123	01010011	99	63	143	01100011	115	73	163	01110011
68	44	104	01000100	84	54	124	01010100	100	64	144	01100100	116	74	164	01110100
69	45	105	01000101	85	55	125	01010101	101	65	145	01100101	117	75	165	01110101
70	46	106	01000110	86	56	126	01010110	102	66	146	01100110	118	76	166	01110110
71	47	107	01000111	87	57	127	01010111	103	67	147	01100111	119	77	167	01110111
72	48	110	01001000	88	58	130	01011000	104	68	150	01101000	120	78	170	01111000
73	49	111	01001001	89	59	131	01011001	105	69	151	01101001	121	79	171	01111001
74	4A	112	01001010	90	5A	132	01011010	106	6A	152	01101010	122	7A	172	01111010
75	4B	113	01001011	91	5B	133	01011011	107	6B	153	01101011	123	7B	173	01111011
76	4C	114	01001100	92	5C	134	01011100	108	6C	154	01101100	124	7C	174	01111100
77	4D	115	01001101	93	5D	135	01011101	109	6D	155	01101101	125	7D	175	01111101
78	4E	116	01001110	94	5E	136	01011110	110	6E	156	01101110	126	7E	176	01111110
79	4F	117	01001111	95	5F	137	01011111	111	6F	157	01101111	127	7F	177	01111111

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
128	80	200	10000000	144	90	220	10010000	160	A0	240	10100000	176	B0	260	10110000
129	81	201	10000001	145	91	221	10010001	161	A1	241	10100001	177	B1	261	10110001
130	82	202	10000010	146	92	222	10010010	162	A2	242	10100010	178	B2	262	10110010
131	83	203	10000011	147	93	223	10010011	163	A3	243	10100011	179	B3	263	10110011
132	84	204	10000100	148	94	224	10010100	164	A4	244	10100100	180	B4	264	10110100
133	85	205	10000101	149	95	225	10010101	165	A5	245	10100101	181	B5	265	10110101
134	86	206	10000110	150	96	226	10010110	166	A6	246	10100110	182	B6	266	10110110
135	87	207	10000111	151	97	227	10010111	167	A7	247	10100111	183	B7	267	10110111
136	88	210	10001000	152	98	230	10011000	168	A8	250	10101000	184	B8	270	10111000
137	89	211	10001001	153	99	231	10011001	169	A9	251	10101001	185	B9	271	10111001
138	8A	212	10001010	154	9A	232	10011010	170	AA	252	10101010	186	BA	272	10111010
139	8B	213	10001011	155	9B	233	10011011	171	AB	253	10101011	187	BB	273	10111011
140	8C	214	10001100	156	9C	234	10011100	172	AC	254	10101100	188	BC	274	10111100
141	8D	215	10001101	157	9D	235	10011101	173	AD	255	10101101	189	BD	275	10111101
142	8E	216	10001110	158	9E	236	10011110	174	AE	256	10101110	190	BE	276	10111110
143	8F	217	10001111	159	9F	237	10011111	175	AF	257	10101111	191	BF	277	10111111

Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin	Dec	Hex	Oct	Bin
192	C0	300	11000000	208	D0	320	11010000	224	E0	340	11100000	240	F0	360	11110000
193	C1	301	11000001	209	D1	321	11010001	225	E1	341	11100001	241	F1	361	11110001
194	C2	302	11000010	210	D2	322	11010010	226	E2	342	11100010	242	F2	362	11110010
195	C3	303	11000011	211	D3	323	11010011	227	E3	343	11100011	243	F3	363	11110011
196	C4	304	11000100	212	D4	324	11010100	228	E4	344	11100100	244	F4	364	11110100
197	C5	305	11000101	213	D5	325	11010101	229	E5	345	11100101	245	F5	365	11110101
198	C6	306	11000110	214	D6	326	11010110	230	E6	346	11100110	246	F6	366	11110110
199	C7	307	11000111	215	D7	327	11010111	231	E7	347	11100111	247	F7	367	11110111
200	C8	310	11001000	216	D8	330	11011000	232	E8	350	11101000	248	F8	370	11111000
201	C9	311	11001001	217	D9	331	11011001	233	E9	351	11101001	249	F9	371	11111001
202	CA	312	11001010	218	DA	332	11011010	234	EA	352	11101010	250	FA	372	11111010
203	CB	313	11001011	219	DB	333	11011011	235	EB	353	11101011	251	FB	373	11111011
204	CC	314	11001100	220	DC	334	11011100	236	EC	354	11101100	252	FC	374	11111100
205	CD	315	11001101	221	DD	335	11011101	237	ED	355	11101101	253	FD	375	11111101
206	CE	316	11001110	222	DE	336	11011110	238	EE	356	11101110	254	FE	376	11111110
207	CF	317	11001111	223	DF	337	11011111	239	EF	357	11101111	255	FF	377	11111111

Input impedance
The impedance 'seen' by any device connected to its inputs. This is the impedance measured across the inputs.

Integer
An integer refers to a whole number (not a fraction) that can be positive or negative. Integer numbers does not have decimal places unlike Floating Point numbers.

Interface
In programming, interface refers to the point that provides a two-way interaction between two devices.

iDev components
The term iDev components is the term used to describe variables, pages, page components, loops, functions, styles etc… that an iDev developer has 'created' in his/her code. The term 'created' means that the developer has named it and its origin comes from the iDev developer. On the other hand, things such as style parameters, setup parameters aren't 'created' by the developer.

Logic level
Logic level is the current state that a digital signal can have. In digital circuits, there only two possible logic level states are Logic HIGH and Logic LOW. In Binary, the digit 1 represents Logic HIGH and the digit 0 for Logic LOW.

Non-volatile memory
This is a type of memory that keeps the data contents in storage after the power of the device is turned off. An example of non-volatile memory is EEPROM.

Oscillator
An oscillator is a device that produces periodic fluctuations between two things based on changes in energy. Devices such as computers and clocks use oscillators.

Page components
In iDev, page components refer to the things that can be 'added' to a page namely text component, image component, draw component and key component.

Parity bit
The parity bit is used for error correction of data packets. It is a bit added to ensure that the number of bits with the value '1' in a set of bits is even or odd. A parity bit can be added in data packets sent through RS232, RS422/RS485, AS1 and AS2 interfaces in iDev.

Pointer
Pointer is a certain variable type that is used to locate another variable. The use of pointers allows the developer to perform data related operations quicker. In iDev, pointers can be used to point to other pointers, iDev components and page components but in most cases pointers are used to direct to another variable**.**

Pull low/ pull high
Sometimes in electronics, some circuit components are described to be pulled LOW or pulled HIGH. Pull HIGH refers to the use of pull-up resistors to ensure that given no other input, a circuit assumes a

default HIGH value and vice-versa for Pull Low.

Ripple
A ripple is a small unwanted periodic variation of the DC output from a power supply. The ripple can be caused by incomplete conversion of an AC signal to a DC signal within the power supply. An ideal power supply has minimum ripple effects.

Source and sink currents
The source and sink currents describes the direction of current flow. The source current provides a constant source of positive charge carriers (i.e. provides current) and the sink current absorbs the constant flow of positive charge carriers (i.e. absorbs current).

Synchronous communication
The process of data communication where the clock signal between the transmitting device and receiving device is shared. For example, a clock signal is set by the Master device in I2C communication is the same one that the Slave device uses.

Volatile memory
Data stored in volatile memory storage are erased after the device is turned off. The RAM (random access memory) of the TFT module is volatile memory, so anything stored currently in it is erased after the module is powered off.

11. Appendix

References:

http://www.docstoc.com/docs/30167115/Conversion-Table---Decimal-Hexidecimal-Octol-Binary-Conversion-Table---Decimal-Hexidecimal-Octol hexadecimal converter table

http://www.cplusplus.com/reference/clibrary/cstdio/printf/ c printf format parameters

http://www.cprogramming.com/tutorial/lesson2.html if statement

http://www.ad-net.com.tw/index.php?id=62 rs232 rs485

http://www.best-microcontroller-projects.com/i2c-tutorial.html i2c

http://www.robot-electronics.co.uk/acatalog/I2C_Tutorial.html i2c

http://www.barrgroup.com/Embedded-Systems/How-To/PWM-Pulse-Width-Modulation pwm

http://www.intel.com/support/motherboards/desktop/sb/CS-023466.htm usb mini-b image

http://en.wikipedia.org/wiki/Universal_Serial_Bus usb type a and mini b image

http://www.pcguide.com/intro/fun/clockClocks-c.html cycle, rise time fall time, clock signal image

12. Accessories (Incomplete)

12.1. CANBUS Adaptor

12.2. Capacitive Touch

12.3. Rotary Encoder

12.4. Battery Connector

12.5. External Soundcard

13. COMMAND FORMAT ARCHIVE (INCOMPLETE)

INC COMMANDS
INC command format:
For single files
INC("Source/Filename") ;
For multiple files
INC("Source/Filename1", "Source/Filename2", "Source/Filename3"…);

LIB COMMANDS
LIB command format for images:
LIB(Library image name, "Source/Filename");
LIB command format for transparency:
LIB(Library image name, "Source/Filename?back=Colour in HEX");
LIB command format for rotation:
LIB(Library image name, "Source/Filename?rotate=0°, 90°, 180° or 270°");
LIB command format for scaling:
LIB(Library image name, "Source/Filename?scale=value");
LIB command format for multiple transformations:
LIB(Library image name, "Source/Filename?transformation1&transformation2..");
LIB command format for fonts and sounds:
LIB(Library font/sound name, "Source/Filename");
LIB command format for mapping fonts:
LIB(Library font name, "Source/Filename?start=HEX value to be mapped");

SETUP COMMANDS
SETUP command format:
Setup Header
SETUP(SYSTEM)
Setup Body
{
parameter1 = parameter value1;
parameter2 = parameter value2;
parameter3 = parameter value3;
…
}

RESET COMMANDS
RESET command format:
RESET(Name of iDev property)

PAGE COMMANDS
PAGE command format:
Page Header
PAGE(Page name, Page style)
Page Body
{
Page Components…
}

STYLE COMMANDS
STYLE command format:
Style Header
STYLE(Style name, Style type)
Style Body
{
style parameter 1 = style value 1;
style parameter 2 = style value 2;
style parameter 3 = style value 3;
...
}
STYLE command format inherit:
Style Header
STYLE(New Style name, Style name inherit)
Style Body
{
new style parameter 1 = new style value 1;
new style parameter 2 = new style value 2;
new style parameter 3 = new style value 3;
...
}

POSN COMMANDS
POSN command format:
To change cursor position
POSN(x coordinate, y coordinate);
To reposition cursor position based on previous cursor position
POSN(+ /- x coordinate,+ /- y coordinate);
For single Page/Page Component
POSN(x coordinate, y coordinate, Page/Page Component);
For multiple Page/Page Components
POSN(x coordinate, y coordinate, Page1/Page Component1, Page2/Page Component2…);

TEXT COMANDS
TEXT command format:
TEXT(Text component name, "Text component", Text Style);
TEXT command format with text data source from a variable:
TEXT(Text component name, Text variable, Text Style);
TEXT command format using text component and text cursor manipulation:
TEXT(Text component name, "\\HEX CodeText component", Text Style);
TEXT command format to update text component that has been declared before:
TEXT(Text component name, "New text component");;
TEXT command format to pass all array elements to text component:
TEXT(Text component, Array source name);
TEXT command format to pass all array elements in the specified 1st dimension to text component:
TEXT(Text component, Array source name.1D);
TEXT command format to pass all array elements to text component:
TEXT(Text component, Array source name.1D.2D);
TEXT command format to pass array elements in the specified 3rd dimension to text component:

TEXT(Text component, Array source name.1D.2D.3D);
TEXT command format to apply different data format to text component:
TEXT(Text Component,%Data format%Variable Source, Text Style);
TEXT command format to apply printf data format to text component:
TEXT(Text Component,%*Printf Data Format %Variable Source, Text Style);
Printf Data format:
FlagsWidth.PrecisionLengthSpecifier

IMG COMMANDS
IMG command format for image already stored in iDev Library:
IMG(Image component name, Library Image name, Image Style);
IMG command format for image stored in SDHC card:
IMG(Image component name, "Source/Filename", Image Style);

DRAW COMMANDS
DRAW command format:
DRAW(Draw component name, size/coordinate X, size/coordinate Y, Draw style);

KEY COMMANDS
KEY command format:
KEY(Key component name, Function name, X, Y, Key style);
KEY command format using inline commands:
KEY(Key component name, [Inline command1,Inline command2..], X, Y, Key style);
KEY command format for external key:
KEY(Key component name, Function name, KXX, KYY, Key style);
KEY command format using inline commands for external key:
KEY(Key component name, [Inline command1,Inline command2..], KXX, KYY, Key style);

SHOW COMMANDS
SHOW command format:
SHOW(Page name or page component name);
SHOW command format for multiple page components:
SHOW(Page name1/component name1,Page name2/component name2…);

HIDE COMMANDS
HIDE command format to disable interrupts:
HIDE(Interrupt name1, Interrupt name2…);
HIDE command format:
HIDE(Page name or page component name);
HIDE command format for multiple page components:
HIDE(Page name1/component name1,Page name2/component name2…);
HIDE command format to disable interrupts:
HIDE(Interrupt name1, Interrupt name2…);

DEL COMMANDS
DEL command format:
DEL(iDev component name);
DEL command format for multiple iDev components:
DEL(iDev component name1,iDev component name2…);

LOAD COMANDS

LOAD command format to update styles:

LOAD(Style name.Parameter,New Parameter Value);

LOAD command format to change stored text data:

LOAD(Destination variable name, Text data source);

LOAD command format to change stored text data from multiple sources:

LOAD(Destination variable name, Text data source1, Text data source2…);

LOAD command format to change stored integer/ float data:

LOAD(Destination variable name, Int/float data source);

LOAD command format to change stored integer/float data from multiple sources:

LOAD(Destination variable name, Int/float data source1, Int/float data source2…);

LOAD command format for using pointers:

LOAD(Pointer variable name>"Shared destination value", Destination Identifier);

LOAD command format to change single element in one-dimensional array:

LOAD(Array name.1D, New element value/variable);

LOAD command format to transfer single element to variable:

LOAD(Variable name, Array name.1D);

LOAD command format to change all elements with a single value in one-dimensional array:

LOAD(Array name, Single value);

LOAD command format to change multiple elements in one-dimensional array:

LOAD(Array name,1st element value,2nd element value, 3rd element value…);

LOAD command format to pass array elements to serial interface/text variable or another array:

LOAD(Destination of array elements, Array source name);

LOAD command format when array elements come from serial interface (serial buffer):

LOAD(Array name, Serial interface source);

LOAD command format to change single element in two-dimensional array:

LOAD(Array name.1D.2D, New element value/variable);

LOAD command format to transfer single element to variable:

LOAD(Variable name, Array name.1D.2D);

LOAD command format to change multiple elements in 1st dimension of a two-dimensional array:

LOAD(Array name.1D,1st element value,2nd element value,3rd element value…);

LOAD command format to pass array elements in the specified 1st dimension to serial interface/ text variable or another array:

LOAD(Destination of array elements, Array source name.1D);

LOAD command format when all array elements come from serial interface (serial buffer):

LOAD(Array name, Serial interface source);

LOAD command format when 1st dimension array elements come from serial interface (serial buffer):

LOAD(Array name.1D,Serial interface source);

LOAD command format to change single element in three-dimensional array:

LOAD(Array name.1D.2D.3D, New element value/variable);

LOAD command format to transfer single element to variable:

LOAD(Variable name, Array name.1D.2D.3D);

LOAD command format to change multiple elements in specified 2nd dimension of a three-dimensional array:

LOAD(Array name.1D.2D,1st element value,2nd element value,3rd element value…);

LOAD command format to pass array elements in the specified 2nd dimension to serial interface/text variable or another array:

LOAD(Destination of array elements, Array source name.1D.2D);

LOAD command format when all array elements come from serial interface (serial buffer):
LOAD(Array name, Serial interface source);
LOAD command format when 2nd dimension array elements come from serial interface (serial buffer):
LOAD(Array name.1D.2D,Serial interface source);
LOAD command format to change single element in three-dimensional array:
LOAD(Array name.1D.2D.3D.4D, New element value/variable);
LOAD command format to transfer single element to variable:
LOAD(Variable name, Array name.1D.2D.3D.4D);
LOAD command format to change multiple elements in specified 3rd dimension of a three-dimensional array:
LOAD(Array name.1D.2D.3D,1st element value,2nd element value…);
LOAD command format to pass array elements in the specified 3rd dimension to serial interface/text variable or another array:
LOAD(Destination of array elements, Array source name.1D.2D.3D);
LOAD command format when all array elements come from serial interface (serial buffer):
LOAD(Array name, Serial interface source);
LOAD command format when 3rd dimension array elements come from serial interface (serial buffer):
LOAD(Array name.1D.2D.3D,Serial interface source);
LOAD command format to apply different data format to a destination (serial interface/variable):
LOAD(Destination,%Data format%Variable Source);
LOAD command format to apply printf data format to a destination (serial interface/variable):
LOAD(Destination,%*Printf Data Format%Variable Source);
Printf Data format:
FlagsWidth.PrecisionLengthSpecifier
LOAD command format to change value of a variable:
LOAD(Destination Variable, New Value/Variable);
LOAD command format to combine/concatenate values or contents of variables and pass the result to a variable:
LOAD(Text Variable, "New Text"/Text Variable1, " New Text"/ Text Variable2…);
LOAD command format to send contents of a variable or a value through an interface:
LOAD(Interface, New Value/Variable);
LOAD command format to send combined/concatenated contents of a text variable or text data through an interface:
LOAD(Interface, "New Text"/Text Variable1, "New Text"/ Text Variable2,…);
LOAD command format to use a previously defined page as a template for a new page that is created, page refresh is needed to make changes visible:
LOAD(Destination Page, Previously Defined Page);;
LOAD command format to change specific setup parameters:
LOAD(Setup Name.Parameter, New Parameter Value);
LOAD command format to transfer files from SDHC to on-board NAND flash (used with FPROG – Chapter 8.3):
LOAD(NAND, "SDHC/Filename");
LOAD command format to **send** data through a specified interface:
LOAD(Interface, Var/Array/"Data");
LOAD command format to **send** multiple data through a specified interface:
LOAD(Interface, Var1/Array1/"Data1", Var2/Array2/"Data2");
LOAD command format to **receive** data through a specified interface:
LOAD(Variable/Array, Interface);
LOAD command format to **send** data through I2C:

LOAD(I2C, Device Address, Read Bytes, Var/Array/"Data");
LOAD command format to **send** multiple data through I2C:
LOAD(I2C, Device Address, Read Bytes, Var1/Array1/"Data1", Var2/Array2/"Data2");
LOAD command format to **receive** data through I2C:
LOAD(Variable/Array, I2C);
LOAD command to control Digital Output, where XX is the I/O assignment, 0 (Logic LOW) and 1 (Logic HIGH):
LOAD(KXX, 0/1);
LOAD command to store the Digital I/O state, where XX is the I/O assignment, value stored is either 0 (Logic LOW) and 1 (Logic HIGH):
LOAD(Variable/Arrray, KXX);
LOAD command to control Digital Output, where V is the 8-bit I/O variable :
LOAD(KV, \\HEX code);
LOAD command to store the Digital I/O state, where XX is the 8-bit I/O variable, value stored is in HEX code:
LOAD(Variable/Arrray, KV);
LOAD command format to update/change specific setup parameters:
LOAD(Interface.Parameter, New Parameter Value);
LOAD command to turn the Piezo output ON:
LOAD(BUZZ, ON/OFF);
LOAD command to turn the Piezo output to a specified duration value (in ms) or a value in variable:
LOAD(BUZZ, Duration Value/Variable);
LOAD command to '**read'** the current RTC :
LOAD(Variable, RTC);
LOAD command format to '**set'** RTC using 24-hour time with fixed format:
LOAD(RTC, "YYYY:MM:DD:hh:mm:ss");
LOAD command using variables to allow user to change RTC stored:
LOAD(RTC, yearvar,":",monthvar, ":",dayvar, ":",hourvar, ":",minvar, ":"secvar);
LOAD command to '**read'** the current RTA:
LOAD(Variable, RTA);
LOAD command format to '**set'** RTA using 24-hour time with fixed format:
LOAD(RTA,":MM:DD:hh:mm:ss");
LOAD command format to upload and write files to NAND via serial interface:
LOAD("EXT/Filename?size=value&timedate=value&usechecksum=value&useack=value");
CR Filedata ChecksumH ChecksumL

RUN COMMANDS
RUN command format:
RUN(Function Name);
RUN command format with **Inline Function:**
RUN([Function contents]);

FUNC COMMANDS
FUNC command format:
Function Header
FUNC(Function Name)
Function Body
{
Function contents…
}
Inline Function command format :
In the function parameter of the iDev command
[Function contents]

LOOP COMMANDS
LOOP command format:
Loop Header
LOOP(Loop name, Loop duration)
Loop Body
{
Loop contents…
}
EXIT command format:
EXIT();
EXIT command format for a specific loop
EXIT(Loop name);

VAR COMMANDS
VAR command format:
VAR(Variable name, Starting value, Variable Style);
VAR command format for text variable:
VAR(Variable name, "Starting text value", Variable Style);
VAR command format for **pointers**:
VAR(Pointer variable name>"Shared destination value", Pointer type);
VAR command format for **one-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D);
VAR command format for **two-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D,Size2D);
VAR command format for **three-dimensional arrays**:
VAR(Array name, Array initial values, Data type, Size1D,Size2D,Size3D);
VAR command format for **four-dimensional arrays**:
VAR(Array name, Array initial values, Data type,Size1D,Size2D,Size3D,Size4D);
VAR command format to apply different data format to value stored:
VAR(Variable name,%Data format%Starting Value, Variable Style);
VAR command format to apply printf data format to value stored:
VAR(Variable name,%*Printf Data Format %Starting Value, Variable Style);

IF COMMANDS

IF command format to create if statements with just one action, note that a *Operand1* can be a variable and *Operand2* can be a literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:

IF(Operand1 Operator Operand2?Function);

IF command format to create if statements with an else action, note that a *Operand1* can be a variable and *Operand2* can be a variable, literal text string or numeric value but both *Operands* cannot be literal text string or numeric value:

IF(Operand1 Operator Operand2?Function1:Function2);

CASE SELECT

Typical iDev Case Switch/Select method note that *chkstr, input* and *runfnc* are text variables, *caseval* is a S8 integer variable that were predefined:

LOAD(chkstr, ",", input, ",");
CALC(caseval, ", 1AG, 2GQ, 3TE, 4PL, ", chkstr, "FIND");
IF(caseval < 0? case_default: [LOAD(runfnc, "case_", input); RUN(runfnc);]);

CALC COMMANDS

CALC command format for most arithmetic methods:

CALC(Destination Variable, Operand 1, Operand2, "Method");

CALC command format for most text string methods:

CALC(Destination Variable, Operand1, Operand2, Method);

CALC command format for most buffer handling methods:

CALC(Destination Variable, Operand 1, Operand2, Operand3, Method);

CALC command format for NAND directory listing:

CALC(Destination Variable, Operand1, Operand2, Operand3, "DIR");

CALC command format for MCHK iDev Checksums where **Operand1** is the source of the buffer and **Operand2** is the type:

CALC(Destination Buffer, Operand1, Operand2, "MCHK");

This checksum copies the buffer in Operand1 to the destination buffer. If unsure what buffer means then refer to the glossary located at end of this guide. This makes a checksum of the type specified in Operand2 and append to the destination buffer.

CALC command format for TCHK iDev Checksums where **Operand1** is the source of the buffer and **Operand2** is the type:

CALC(Result, Operand1, Operand2, "TCHK");

CALC command format for CRC16 in iDev:

CALC(Destination Variable, Operand1, Operand2, Operand3, "CRC16");

CALC command format for CRC32 in iDev:

CALC(Destination Variable, Operand1, Operand2, Operand3, "CRC32");

CALC command format for deducing text, draw and image component information:

CALC(Destination Variable, Operand1, "Method");

CALC command format for splitting data buffers:

CALC(Destination Pointer, Operand1, Operand2, "MSPLIT");

INT COMMANDS
INT command format to use as wrap-around interrupt for the runtime counter:
INT(Interrupt name, Runtime counter, Function to be called);
INT command format to setup timer interrupts, where x is the number of timer interrupt being used (TIMER0-TIMER9):
INT(Timer Interrupt name, TIMERx, Function to be called);
INT command format to set up Interface interrupts in iDev:
INT(Interrupt Name, Interface Buffer, Function);
INT command format to set an interrupt triggered by RTA:
INT(Interrupt Name, RTA, Function);

WAIT COMMANDS
WAIT command format:
WAIT(Duration);

FPROG COMMANDS
FPROG command to transfer:
FPROG;
LOAD(NAND, "SDHC/Filename1");
LOAD(NAND, "SDHC/Filename2");
LOAD(NAND, "SDHC/Filename3");
…
FEND;

14. IMAGE FILES USED IN GUIDE

Files are found in a folder called 'images used in example codes in iDev'

Chapter	Example code in Fig number	Filename	Image Description
2.3.2	2.8	image1.bmp	sunset
2.3.3	2.9	image1.bmp	sunset
	2.12	image1.bmp	sunset
2.3.5	2.16	image1.bmp	sunset
		image2.bmp	green button
		image3.bmp	red button
2.3.6	2.18	greenbox.bmp	green button
		redbox.bmp	red button
		number1.bmp	number 1 image
		tick.bmp	tick image inside green circle
2.3.9	2.24	sunset.bmp	sunset
2.3.11	2.28	greenbox.bmp	green button
		redbox.bmp	red button
2.3.12	2.32	togoff.bmp	toggle on image
		togon.bmp	toggle off image
2.4	2.35	togoff.bmp	toggle on image
		togon.bmp	toggle off image
	2.36	togoff.bmp	toggle on image
		togon.bmp	toggle off image
2.5.4	2.40	greenbox.bmp	green button
3.5	3.60	Back.bmp	numpad image for different cases
3.6.1	3.67	Backg.bmp	numpad image for calculator