APPLICATION NOTES

Designer or ViSi Strings and Character Arrays

DOCUMENT DATE: **27th APRIL 2019**
DOCUMENT REVISION: **1.1**

Description

This application note shows how string characters are stored in and accessed from memory. This application is intended for use in the Workshop 4 – Designer environment. The 4DGL code of the Designer project can be copied and pasted to an empty ViSi project and it will compile normally. The code can also be integrated to that of an existing ViSi project.

This application note requires:

- Any of the following 4D Picaso and gen4 Picaso display modules:

gen4-uLCD-24PT	gen4-uLCD-28PT	gen4-uLCD-32PT
uLCD-24PTU	uLCD-32PTU	uVGA-III

and other superseded modules which support the ViSi Genie environment

- The target module can also be a Diablo16 display

gen4-uLCD-24D series	gen4-uLCD-28D series	gen4-uLCD-32D series
gen4-uLCD-35D series	gen4-uLCD-43D series	gen4-uLCD-50D series
gen4-uLCD-70D series		
uLCD-35DT	uLCD-43D series	uLCD-70DT

Visit www.4dsystems.com.au/products to see the latest display module products that use the Diablo16 processor. The display module used in this application note is the uLCD-32PTU, which is a Picaso display. This application note is applicable to Diablo16 display modules as well.

- 4D Programming Cable / uUSB-PA5/uUSB-PA5-II for non-gen4 displays(uLCD-xxx)
- 4D Programming Cable & gen4-PA, / gen4-IB / 4D-UPA for gen4 displays (gen4-uLCD-xxx)
- micro-SD (µSD) memory card
- Workshop 4 IDE (installed according to the installation document)
- Any Arduino board with a UART serial port
- 4D Arduino Adaptor Shield (optional) or connecting wires
- Arduino IDE
- When downloading an application note, a list of recommended application notes is shown. It is assumed that the user has read or has a working knowledge of the topics presented in these recommended application notes.

Content

Description 2
Content 3
Application Overview 3
Setup Procedure 4
Create a New Project 4
Design the Project 4
The Function to(outstream) 4
Little Endian Order of Storage 5
Example Code – StringsBasics1.4dg 6
Accessing a Character in the String using Word-Aligned Pointers 6
Print the Characters of a Word Element 7
Print a Low-byte Character 7
Print a High-byte Character 7
Replace a Low-byte Character 7
Replace a High-byte Character 7
Accessing a Character in the String using Byte-Aligned Pointers 8
Print a Character 8
Replace a Character 9
Run the Program 10
Proprietary Information 12
Disclaimer of Warranties & Limitation of Liability 12

Application Overview

At the time of writing of this application note, there exists only one variable data type in 4DGL. This is the ***var*** data type, which is a 16-bit signed integer variable. Since it is 16-bit wide, the ***var*** variable is also a word. It is also possible to declare an array of ***var*** or word variables. Each element of a ***var*** variable array therefore is a 16-bit wide signed integer or a word.

There are no specific variable data types for character arrays and strings. In 4DGL, character arrays and strings are stored inside ***var*** variable arrays. Since a character is one-byte wide, each element of a ***var*** variable array can store two characters.

There are several ways of entering string data into a variable array, two of which are using the functions “**to(…)**” (in combination with the function “**print(…)**”) and “**str_PutByte(…)**”. Another one is direct assignment using word-aligned pointers.

Likewise, there are several ways of accessing string data stored inside a word array. One way is to treat the word array containing the string as it is – a collection of word elements. Another way is to treat it as a region in memory where a string is stored. The former requires the use of word-aligned pointers, while the latter requires the use of byte-aligned pointers and the string class functions.

This application note discusses and elaborates the concepts above through examples.

Setup Procedure

For instructions on how to launch Workshop 4, how to open a **Designer** project, and how to change the target display, kindly refer to the section "**Setup Procedure**" of the application note
[Designer Getting Started - First Project]

For instructions on how to launch Workshop 4, how to open a **ViSi** project, and how to change the target display, kindly refer to the section "**Setup Procedure**" of the application note
[ViSi Getting Started - First Project for Picaso and Diablo16]

Create a New Project

For instructions on how to create a new **Designer** project, please refer to the section "**Create a New Project**" of the application note
[Designer Getting Started - First Project]

For instructions on how to create a new **ViSi** project, please refer to the section "**Create a New Project**" of the application note
[ViSi Getting Started - First Project for Picaso and Diablo16]

Design the Project

The Function to(outstream)

There are several ways of assigning a string to a word array. One way is to stream a literal string constant to the word array using the functions "**to(outstream)"** and "**print(…)**". To store the string "**HELLO WORLD**" for instance, we write,

```
var buffer[10];
```

```
to(buffer); print("HELLO WORLD") ;
```

Note that we had to declare the word array "**buffer**" which has a size of 10. This means that there are 10 word elements in the array. Each word element can contain a pair of bytes. The total number of bytes that ***buffer*** can hold is 20. To know the number of bytes that a word array can hold, simply multiply the array size by two. To illustrate,

Character or byte capacity = array size X 2

The function "**print(…)**" then streams the characters of the literal string constant "**HELLO WORLD**" to the word array ***buffer***. Note that "**HELLO WORLD**" will be automatically terminated with a NULL character.

Little Endian Order of Storage

Inside a word array element there are two bytes of available memory – one of which may be pertained to as the high byte and the other the low byte. To illustrate using the word array "***buffer***",

element	buffer[0]		buffer[1]		buffer[2]		buffer[3]	
byte	*high*	*low*	*high*	*low*	*high*	*low*	*high*	*low*

element	buffer[4]		buffer[5]		buffer[6]		buffer[7]	
byte	*high*	*low*	*high*	*low*	*high*	*low*	*high*	*low*

element	buffer[8]		buffer[9]	
byte	*high*	*low*	*high*	*low*

When characters are streamed to a word array, the processor starts with the first element (***buffer[0]*** in this example). Inside the first element, the low byte is filled first before the high byte (little endian order). After filling both bytes of the first element, the processor proceeds to the second element (***buffer[1]*** in this example). This process is repeated until all of the characters are saved. The string is then terminated with a NULL character. To illustrate,

element	buffer[0]		buffer[1]		buffer[2]		buffer[3]	
byte	*high*	*low*	*high*	*low*	*high*	*low*	*high*	*low*
Char	E	H	L	L	space	O	O	W
Hex	45	48	4C	4C	20	4F	4F	57

element	buffer[4]		buffer[5]		buffer[6]		buffer[7]	
byte	*high*	*low*	*high*	*low*	*high*	*low*	*high*	*low*
Char	L	R	NULL	D	NULL	NULL	NULL	NULL
Hex	4C	52	0	44	0	0	0	0

element	buffer[8]		buffer[9]	
byte	*high*	*low*	*high*	*low*
Char	NULL	NULL	NULL	NULL
Hex	0	0	0	0

Example Code – StringsBasics1.4dg

Below is a screenshot image of the project "**stringsBasics1.4dg**" attached to this application note.

```
1   #platform "uLCD-32PTU"
2
3   #inherit "4DGL_16bitColours.fnc"
4
5   func main()
6       var buffer[10];
7       var n;
8
9       gfx_ScreenMode(LANDSCAPE) ;          // change manually if ori
10
11      to(buffer); print("HELLO WORLD") ;   // this statement
12                                           // saves the HELLO WORLD
13                                           // to the variable array
14
15      while(n<10)                          // while the index n is l
16          print("\n",[HEX]buffer[n]);          // print content of b
17          n++;                             // increment n by 1
18      wend                                 // end the while lop if r
19
20      repeat                               // maybe replace
21      forever                              // this as well
22
23  endfunc
```

Again we declare the word array ***buffer***, which has a size of 10. The word variable ***n*** will be used for indexing into the word array.

```
6       var buffer[10];
7       var n;
```

The literal string constant "HELLO WORLD" is now streamed to ***buffer***.

```
11      to(buffer); print("HELLO WORLD") ;   // thi
```

The **while-wend** loop eliminates the need for repetitive statements when printing the content of the elements of a word array.

```
15      while(n<10)                          // whi
16          print("\n",[HEX]buffer[n]);          //
17          n++;                             // inc
18      wend                                 // end
```

The above will print the values of all elements of the word array ***buffer*** (from ***buffer[0]*** to ***buffer[9]***). Each element value will be printed on a new line, and the contents will be printed as hexadecimal vales. The section "**Run the Program**" shows the output of the example code "**stringsBasics1.4dg**".

Accessing a Character in the String using Word-Aligned Pointers

One way of accessing a character of a string saved to a word array is to use a word-aligned pointer, such as that shown in line 16 of the code "**stringsBasics1.4dg**".

```
        print("\n",[HEX]buffer[n]);          // prin
```

Here ***buffer*** is actually a pointer that holds the starting address of the memory location of the actual word array, and ***n*** is the offset of the pointer into that array.

Print the Characters of a Word Element

The line

```
print("\n",[HEX]buffer[0]);
```

would print “4548”.

The line

```
print("\n",[HEX]buffer[1]);
```

would print “4C4C”.

Print a Low-byte Character

To print only the low-byte character, we use the number format specifier **“[HEX2]”** with the function “**print(...)**”. This limits the length of the printed value to two hexadecimal digits.

```
print("\n",[HEX2]buffer[0]);          // print low b
```

Print a High-byte Character

To print only the high-byte character, we perform a bit-shifting operation then print the result up to two hexadecimal digits only.

```
print("\n",[HEX2]buffer[0]>>8);      // print high
```

Replace a Low-byte Character

With each increment of ***n***, the pointer offsets into the next word element of the array, hence the term “word-aligned”. To change the first character of the string from ‘H’ to ‘X’ using a word-aligned pointer, we could write,

```
    //change 'H' to 'X'
    buffer[0] := buffer[0] & 0xFF00;
    buffer[0] := buffer[0] | 0x0058;
```

Since we know that ‘H’ is stored in the low byte of ***buffer[0]***, we clear the low byte of ***buffer[0]*** while at the same time preserving the high byte by ANDing ***buffer[0]*** with ***0xFF00***. Then we set the value of the low byte while preserving the value of the high byte by ORring ***buffer[0]*** with ***0x00nn***, where **nn** is the desired value.

Replace a High-byte Character

To replace the space character with a ‘+’, we could write,

```
    //change 'H' to 'X'
    buffer[0] := buffer[0] & 0xFF00;
    buffer[0] := buffer[0] | 0x0058;
```

To replace the last character ('D') with 'A', we could write,

```
28      //change 'D' to 'A'
29      buffer[5] := 0x0041;
```

Here a literal constant word value is directly assigned to ***buffer[5]***. This shorthand method can be also be used when assigning a character to the high or low byte of a word element, provided of course that the other byte is known.

To print the new value of the string, we write,

```
39      print([STR]buffer);
```

The final value of the string should be "**XELLO+WORLA**". Open the attached Designer project "**stringsBasics1b.4dg**" to see the complete code for this section (**Accessing a Character in the String using Word-Aligned Pointers**).

Accessing a Character in the String using Byte-Aligned Pointers

As you may have already observed in the previous section, accessing a character inside a string stored in a word array using a word-aligned pointer is not straightforward. Note also that there are no data types (as of the time of writing of this application note) specific for characters (or other one-byte wide variables), such that we had to deal with characters using word-aligned pointers. The string class functions in 4DGL allow us to handle strings in an easier (or the normal) way, as will be shown in the following discussions.

Print a Character

Assuming that the word array ***buffer*** now contains the string "**HELLO WORLD**", we first need to declare a word variable ***ptr***.

```
    var n, ptr;
```

Then we assign to ***ptr*** the starting address of the string stored in the word array ***buffer*** by writing

```
    ptr := str_Ptr(buffer);
```

The word variable ***ptr*** now points to the address of letter '**H**', which is the first character of the string in the word array ***buffer***. To access a character we can use **str_GetByte(…)**, which is another 4DGL string class function. For instance, to print '**H**' in hexadecimal, we write,

```
  print([HEX]str_GetByte(ptr + 0),"\n");
```

To print the character as it is we write,

```
  print([CHR]str_GetByte(ptr + 0),"\n");
```

Note the difference between the format specifiers.

To print the seventh character, we write,

```
print([CHR]str_GetByte(ptr + 6),"\n");
```

Note that offsetting the pointer ***ptr*** by 6 advances it to the address of the seventh character. Offsetting ***ptr*** by 1 advances it to the address of the second character or byte, hence the term "**byte-aligned**". Compare this to the "ordinary" word-aligned pointers of the previous section, which advance by increments of two bytes or one word when being offset by a certain value. Note that grouping of the characters of the string by pairs and performing masking and bit-shifting were not needed when accessing the characters. Lastly, note also that we didn't have to worry about the little-endian order of storage of the string inside the word array ***buffer***.

Replace a Character

To put a character into a string, we use the function "**str_PutByte(…)**". To replace the first character in ***buffer*** with 'X' for example, we write,

```
//change 'H' to 'X'
str_PutByte(ptr + 0, 'X');
```

To replace the space character with '+' we write,

```
//change ' ' (the space character) with
str_PutByte(ptr + 5, '+');
```

To replace 'D' with 'A' we write,

```
//change 'D' with 'A'
str_PutByte(ptr + 10, 'A');
```

To print the modified string, we could write

```
print([STR]buffer);
```

or, since we have a byte-aligned pointer, we could write,

```
str_Printf(&ptr, "%s");
```

The Designer project for this section (**Accessing a Character in the String using Byte-Aligned Pointers**) is "**stringsBasics2.4dg**".

Run the Program

For instructions on how to save a **Designer** project, how to connect the target display to the PC, how to select the program destination, and how to compile and download a program, please refer to the section "**Run the Program**" of the application note

[Designer Getting Started - First Project]

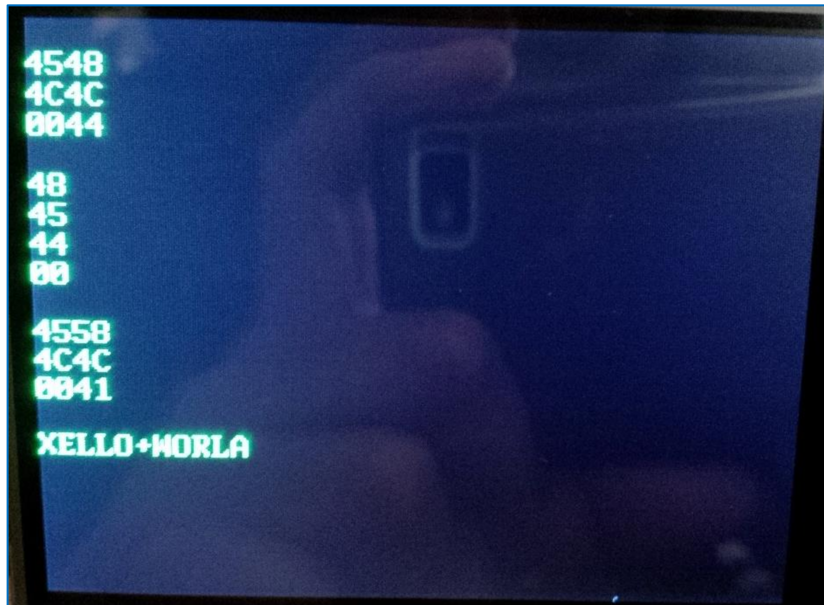
For instructions on how to save a **ViSi** project, how to connect the target display to the PC, how to select the program destination, and how to compile and download a program, please refer to the section "**Run the Program**" of the application note

[ViSi Getting Started - First Project for Picaso and Diablo16]

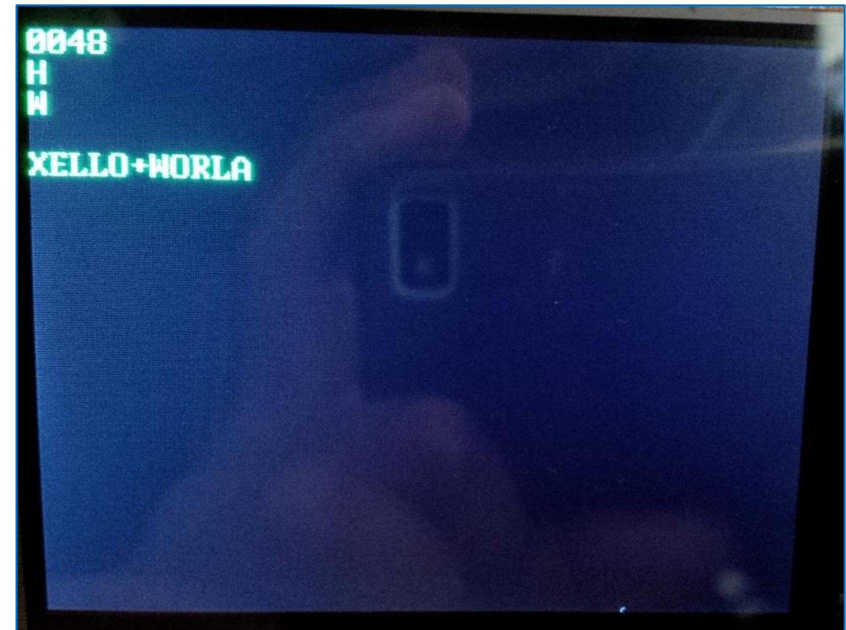
The uLCD-32PTU and uLCD-35DT display modules are commonly used as examples, but the procedure is the same for other displays.

After compiling the code "**stringsBasics1.4dg**" and uploading the program to the display module, the output should look like as shown below.

After compiling the code “**stringsBasics1b.4dg**” and uploading the program to the display module, the output should look like as shown below.



After compiling the code “**stringsBasics2.4dg**” and uploading the program to the display module, the output should look like as shown below.



Proprietary Information

The information contained in this document is the property of 4D Systems Pty. Ltd. and may be the subject of patents pending or granted, and must not be copied or disclosed without prior written permission.
4D Systems endeavours to ensure that the information in this document is correct and fairly stated but does not accept liability for any error or omission. The development of 4D Systems products and services is continuous and published information may not be up to date. It is important to check the current position with 4D Systems.
All trademarks belong to their respective owners and are recognised and acknowledged.

Disclaimer of Warranties & Limitation of Liability

4D Systems makes no warranty, either expresses or implied with respect to any product, and specifically disclaims all other warranties, including, without limitation, warranties for merchantability, non-infringement and fitness for any particular purpose.
Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications.
In no event shall 4D Systems be liable to the buyer or to any third party for any indirect, incidental, special, consequential, punitive or exemplary damages (including without limitation lost profits, lost savings, or loss of business opportunity) arising out of or relating to any product or service provided or to be provided by 4D Systems, or the use or inability to use the same, even if 4D Systems has been advised of the possibility of such damages.
4D Systems products are not fault tolerant nor designed, manufactured or intended for use or resale as on line control equipment in hazardous environments requiring fail – safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines or weapons systems in which the failure of the product could lead directly to death, personal injury or severe physical or environmental damage (‘High Risk Activities’). 4D Systems and its suppliers specifically disclaim any expressed or implied warranty of fitness for High Risk Activities.
Use of 4D Systems’ products and devices in 'High Risk Activities' and in any other application is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless 4D Systems from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any 4D Systems intellectual property rights.