



## Designer or ViSi 4DGL Strings Print Formats – the String and Character Format Specifiers

DOCUMENT DATE: **30<sup>th</sup> APRIL 2019**  
DOCUMENT REVISION: **1.1**

Description

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and other superseded modules which support the ViSi Genie environment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](http://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 .....	2
Application Overview .....	3
Setup Procedure .....	3
Create a New Project.....	3
Design the Project .....	3
<i>The Function str_Printf(...)</i> .....	3
<i>The Format Specifier “%s”</i> .....	4
<i>Automatic Advancing of the Pointer</i> .....	5
<i>The Format Specifier “%c”</i> .....	6
<i>Dynamic Construction of the Format Specifier</i> .....	7
Run the Program .....	8
Proprietary Information .....	9
Disclaimer of Warranties & Limitation of Liability .....	9

## Application Overview

The application note [Designer or ViSi Strings and Character Arrays](#) explains how 4DGL strings and character arrays are stored in and accessed from memory. In that application note, the reader was also introduced to the concept of byte-aligned pointers and the use of the function `str_Printf(...)`.

This application note now further explains the use of the `str_Printf(...)` function together with the “%s” and “%c” format specifiers.

## 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 `str_Printf(...)`

To review the use of the function `str_Printf(...)`, we start by declaring the word array *buffer*, which has a size of ten and the word variable *ptr*.

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

We now stream to *buffer* the literal string constant “1234”.

```
to(buffer); print("1234");
```

We then make the word variable **ptr** a byte-aligned pointer to the string inside **buffer**.

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

To print the string using the pointer, we write:

```
print("buffer: ");
str_Printf(&ptr, "%s");
```

The output of the above code should be:

```
buffer: 1234
```

### The Format Specifier "%s"

The **str\_Printf(...)** function requires the address of the byte-aligned pointer and a format specifier as the arguments. The most commonly used format specifier is "%s". It causes the **str\_Printf(...)** function to print the data pointed to by the byte-aligned pointer as a string of characters. Note that the word array **buffer** in this case would contain the data shown below.

element	buffer[0]		buffer[1]		buffer[2]		buffer[3]	
byte	high	low	high	low	high	low	high	low
Char	2	1	4	3	NULL	NULL	NULL	NULL
Hex	32	31	34	33	0	0	0	0

element	buffer[4]		buffer[5]		buffer[6]		buffer[7]	
byte	high	low	high	low	high	low	high	low
Char	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Hex	0	0	0	0	0	0	0	0

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

The hexadecimal values are the equivalent ASCII values of the characters. For the line

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

to work, the word array (**buffer** in this example) pointed to by the byte-aligned pointer (**ptr** in this example) must contain a null-terminated sequence of ASCII characters.

### Automatic Advancing of the Pointer

Now consider the code snippet shown below.

```
to(buffer); print("1234");

ptr := str_Ptr(buffer);

print("ptr old: ",ptr,"\n");

print("buffer: ");
str_Printf(&ptr, "%s");

print("\n");
print("ptr new: ",ptr,"\n");
```

The output of this would be:

```
ptr old: 16
buffer: 1234
ptr new: 21
```

The initial value of **ptr** is 16. This is the address of the first character in the string.

```
ptr old: 16
```

After the string is printed the value of **ptr** is now **21**.

```
ptr new: 21
```

A change in the value of the pointer occurs because the function **str\_Printf(...)** automatically advances the byte-aligned pointer as it prints the characters. In this case, since the string has four characters, the pointer is now at address **21** after the four characters are printed. So if we would insert another **str\_Printf(...)** command as shown below.

```
to(buffer); print("1234");

ptr := str_Ptr(buffer);

print("ptr old: ",ptr,"\n");

print("buffer: ");
str_Printf(&ptr, "%s");

print("\n");
print("ptr new: ",ptr,"\n");

print("buffer: ");
str_Printf(&ptr, "%s");
```

The output would be

```
ptr old: 16
buffer: 1234
ptr new: 21
buffer:
```

Note that nothing is printed since there are no character bytes starting at address **21** or the low byte of **buffer[2]**. To print the string again therefore, we must first reset the pointer to the beginning address of the string, like as shown below.

```
to(buffer); print("1234");


ptr := str_Ptr(buffer);

print("ptr old: ",ptr,"\n");

print("buffer: ");
str_Printf(&ptr, "%s");

print("\n");
print("ptr new: ",ptr,"\n");

ptr := str_Ptr(buffer);
print("buffer: ");
str_Printf(&ptr, "%s");
```



The Designer project for the discussion on the “%s” format specifier is “stringsBasics3.4dg” (attached).

### The Format Specifier “%c”

The format specifier “%c” causes the **str\_Printf(...)** function to go to the byte address pointed to by the pointer and print the character inside that byte memory location. To illustrate using our previous example, we write,


```
to(buffer); print("1234");

ptr := str_Ptr(buffer);

print("ptr old: ",ptr,"\n");

print("buffer: ");
str_Printf(&ptr, "%c");

print("\n");
print("ptr new: ",ptr,"\n");
```



The output of this code snippet would be:

```
ptr old: 16
buffer: 1
ptr new: 17
```

We see here again that the pointer is automatically advanced. To print the three remaining characters we write,

```

to(buffer); print("1234");

ptr := str_Ptr(buffer);

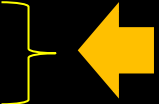
print("ptr old: ",ptr,"\n");

print("buffer: ");
str_Printf(&ptr, "%c");

print("\n");
print("ptr new: ",ptr,"\n");

str_Printf(&ptr, "%c");
str_Printf(&ptr, "%c");
str_Printf(&ptr, "%c");

```



The output of the above snippet would be:

```

ptr old: 16
buffer: 1
ptr new: 17
234

```

The Designer project for the discussion on the “%c” format specifier is “stringsBasics4.4dg” (attached).

### Dynamic Construction of the Format Specifier

The format specifier argument of the *str\_Printf(...)* function can also be a word-aligned string pointer, allowing dynamic construction of the printing format. To illustrate using one of our previous examples, we write,

```

var buffer[10];
var ptr;
var format[10];

gfx_ScreenMode(LANDSCAPE) ;           // change manu

to(buffer); print("1234");
to(format); print("%s");

ptr := str_Ptr(buffer);

print("buffer: ");
str_Printf(&ptr, format);

```

Note that another word array, *format*, was declared.

```

var format[10];

```

To this the literal string constant “%s” was streamed.

```

to(format); print("%s");

```

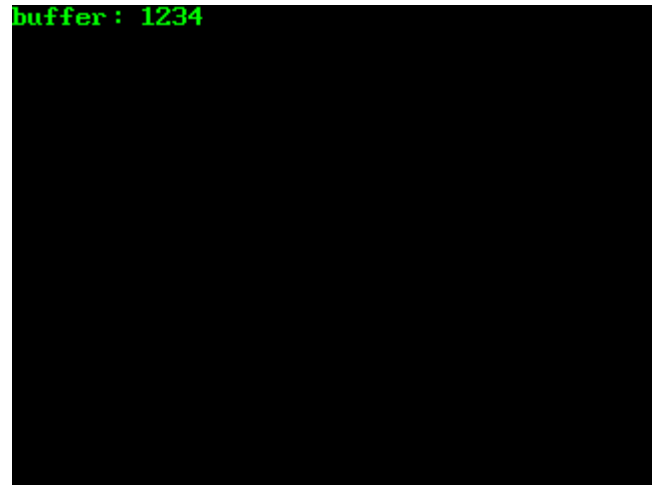
And then *format*, which is essentially a word-aligned pointer to the stored string “%s”, was used as the second argument of the *str\_Printf(...)* function.

```

str_Printf(&ptr, format);

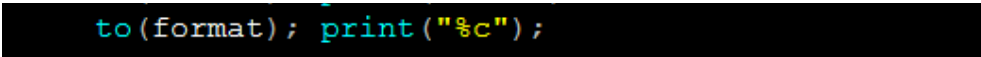
```

The output of the above code snippet would be:




```
buffer: 1234
```

If we were to stream “%c” instead of “%s” to *format*,



```
to(format); print("%c");
```

The output would be:



```
buffer: 1
```

The Designer project for the discussion on dynamic construction of the format specifier is “**stringsBasics5.4dg**” (attached). Although the examples are simple, the ability to construct a format specifier dynamically can be a powerful tool.

## 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.

## 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.