



Designer or ViSi 4DGL Strings Print Formats – the Long Hexadecimal Format Specifier

DOCUMENT DATE: 9th MAY 2020
DOCUMENT REVISION: 1.1

Description

There are three hexadecimal format specifiers:

Specifier	Data to be displayed
<code>%x</code>	Hex byte
<code>%X</code>	Hex word
<code>%IX</code>	Hex long

This application note discusses how the **long hexadecimal** format specifier is used with the **str_Printf(...)** function. This application note 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.

Before getting started, the following are required:

- Any of the following 4D Picaso display modules:

[uLCD-24PTU](#) [uLCD-32PTU](#) [uLCD-43\(P/PT/PCT\)](#)
[uLCD-28PTU](#) [uLCD-32WPTU](#) [uVGA-III](#)

and other superseded modules which support the Designer and/or ViSi environments.

- The target module can also be a Diablo16 display

[uLCD-35DT](#)

[uLCD-70DT](#)

Visit www.4dsystems.com.au/products to see the latest display module products that use the Diablo16 processor.

- [4D Programming Cable](#) or [uUSB-PA5](#)
- [micro-SD \(uSD\)](#) memory card
- [Workshop 4 IDE](#) (installed according to the installation document)

Content

Description	2
Content.....	3
Application Overview	3
Setup Procedure	4
Create a New Project.....	4
Design of the Project	4
<i>The Format Specifier “%IX”</i>	<i>4</i>
<i>The Width and Zero Flag Sub-specifiers</i>	<i>6</i>
The Width Sub-specifier	6
The Zero Flag Sub-specifier	6
<i>Dynamic Construction of the Format Specifier</i>	<i>7</i>
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. It also differentiates between word-aligned and byte-aligned pointers. Furthermore, it introduces the use of the function ***str_Printf(...)***.

The application note [Designer or ViSi 4DGL Strings Print Formats – the String and Character Format Specifiers](#) shows how the string and character format specifiers (“%s” and “%c”, respectively) are used. Also, it covers the topics “**Automatic Advancing of the Pointer**” and “**Dynamic Construction of the Format Specifier**”.

This application note now further explains the use of the ***str_Printf(...)*** function together with the long hexadecimal format specifier.

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 of the Project

The Format Specifier “%IX”

The format specifier “%IX” is used for displaying long hexadecimal numbers. A long hexadecimal in 4DGL is a 32-bit (or 4-byte) value, the range of which is from *0x0000 0000* to *0xFFFF FFFF* (0 to $2^{32}-1$). Consider the code snippet shown below.

```
var val32[2];
var ptr;

gfx_ScreenMode(LANDSCAPE) ;           // change manual

umul_1616(val32, 500, 2000);
ptr := str_Ptr(val32);

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

print("val32: ");
str_Printf(&ptr, "%IX");

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

The output of the above code is:

```
ptr old: 16
val32: F4240
ptr new: 20
```

The function *umul_1616(...)* performs an unsigned multiplication of two 16-bit values, placing the 32-bit result in a two-word array. In this example, the two 16-bit values are 500 and 2000. When multiplied together the product

of these is **1000000** or **0xF4240**. If we print the contents of the word array **val32** in hexadecimal format,

```
print("\n");
print("val32[0]: ", [HEX]val32[0]);
print("\n");
print("val32[1]: ", [HEX]val32[1]);
```

we get,

```
ptr old: 16
val32: 1000000
ptr new: 20
val32[0]: 4240
val32[1]: 000F
```

We analyse the contents of the word array **val32**.

element	val32[0]		val32[1]	
byte	high	low	high	low
address	17	16	19	18
Hex	42	40	00	0F

Low word High word

0x000F 4240 = 1000000

Note also that the pointer was advanced by four bytes after the long hexadecimal value was printed.

element	val32[0]		val32[1]			
byte	high	low	high	low	high	low
Hex	42	40	00	0F	-	-
address	17	16	19	18	21	20

ptr old ptr new

Therefore, the long hexadecimal format specifier, **"%lX"**, causes the **str_Printf(...)** function to get four bytes from the address starting at that pointed to by the byte-aligned pointer. **str_Printf(...)** then treats these four bytes as a 32-bit value and prints it in hexadecimal.

The 32-bit data found starting at address **16**, in this example, is “**0x000F4240**”, the decimal equivalent of which is “**1000000**”.

The Width and Zero Flag Sub-specifiers

The Width Sub-specifier

Consider the output below.

```
val32: F4240
val32:   F4240
```

The code for the first line is


```
print("\n\n");
ptr := str_Ptr(val32);
print("val32: ");
str_Printf(&ptr, "%1X");
```

The code for the second line is

```
print("\n");
ptr := str_Ptr(val32);
print("val32: ");
str_Printf(&ptr, "%101X");
```

Note that in the second line, the number has five spaces preceding it. This is because the width specifier was used in the *str_Printf(...)* function.

```
str_Printf(&ptr, "%101X");
```




Here the width specifier value is **10**, so the field width of the printed figure is ten digits, and since the number is only five hexadecimal digits wide, it is preceded by five space characters.

The Zero Flag Sub-specifier

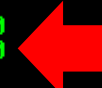
Suppose we want the number to be preceded with zeros rather than spaces, we would write,

```
print("\n");
ptr := str_Ptr(val32);
print("val32: ");
str_Printf(&ptr, "%0101X");
```



Here the width sub-specifier is preceded by the zero flag sub-specifier, which would cause the number to be left-padded with zeros instead of spaces. To illustrate,

```
val32: F4240
val32:   F4240
val32: 00000F4240
```



Therefore, without the zero flag sub-specifier, the default character with which a number, printed with a certain field width, is to be left-padded is the space character. The width and zero flag sub-specifiers can be used with other format specifiers besides the long hexadecimal format specifier.

The Designer project for the discussions on the long hexadecimal format specifier and the width and zero flag sub-specifiers is “**stringsBasics8.4dg**” (attached).

Dynamic Construction of the Format Specifier

As was shown in the application note [Designer or ViSi 4DGL Strings Print Formats – the String and Character Format Specifiers](#), the format specifier argument of the `str_Printf(...)` function can also be a word-aligned string pointer, allowing dynamic construction of the printing format. We will now use dynamically constructed format specifiers to come up with the formatted display output shown below, which is similar to that in the last example.

```
val32: F4240
val32:  F4240
val32: 00000F4240
```

The code snippet for the above output can be implemented using dynamically constructed format specifiers, as shown below.

```
// print a long hexadecimal number as it is
print("\n\n");
print("val32: ");
to(format); print("%lX");
ptr := str_Ptr(val32);
str_Printf(&ptr, format);

// print a long hexadecimal number, 10 digits wide
print("\n");
print("val32: ");
to(format); print("%10lX");
ptr := str_Ptr(val32);
str_Printf(&ptr, format);

// print a long hexadecimal number, 10 digits wide
print("\n");
print("val32: ");
to(format); print("%010lX");
ptr := str_Ptr(val32);
str_Printf(&ptr, format);
```

Where `format` is a word array declared at the start of the code.

```
func main()
var val32[2];
var format[10];
var ptr;
```

The Designer project for the remaining part of this application note is “**stringsBasics8b.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.