APPLICATION NOTES

4D SYSTEMS
TURNING TECHNOLOGY INTO ART

ViSi-Genie Magic 32bit LED Digits

DOCUMENT DATE: 29th May 2019
DOCUMENT REVISION: 1.0

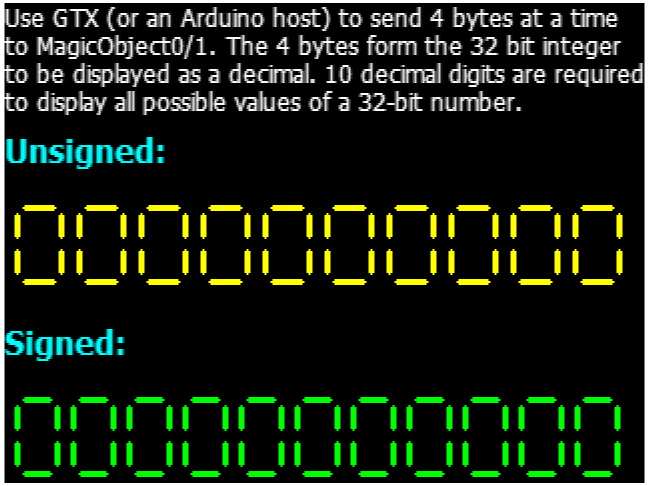
Description

This application note primarily shows how the **Magic Code and Magic Object** objects are used to implement a project that displays 32-bit integer values on the screen using a LED digits object. The implementation further requires the use of the following features and functions in combination with the **Magic Code and Magic Object** objects:

- **String Class Functions**
- **Image Control Functions**

The **String Class functions** and **Image Control functions** are functions native to the Picaso and Diablo16 processors.

Below is a screenshot image of the project used in this application note.



Note 1: Workshop Pro is needed for this application.

Before getting started, the following are required:

- Any of the following 4D Picaso display modules:

  uLCD-24PTU uLCD-32PTU uLCD-43(PT/PCT)
  uLCD-28PTU uLCD-32WPTU

  and other superseded modules which support the ViSi Genie environment

- The target module can also be a Diablo16 **touch** display

  uLCD-35DT uLCD-70DT
  uLCD-43DT uLCD-43DCT

  Visit www.4dsystems.com.au/products to see the latest display module products that use the Diablo16 and Picaso processors.
- 4D Programming Cable or µUSB-PA5
- micro-SD (µSD) memory card
- Workshop 4 IDE (installed according to the installation document)
- 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

Application Overview

The Diablo16 and Picaso are 16-bit processors, and signed number operation with 16-bit integers limits the maximum number that can be displayed by LED digits objects to "32,767". 2^16 equals 65,536. Divide this by two since the first half is used to represent positive numbers; the remaining half is used to represent negative numbers. Thus, attempting to create a 6-digit or more LED digits object or to send to a LED digits object a value beyond the limit results to red "X" marks shown on the object during runtime. To be able to display a value higher than "32,767" in ViSi-Genie, one solution is to use **magic objects**.

With the release of Workshop4 PRO, it is now possible for users to insert 4DGL codes (in the form of magic objects) into Genie projects. This feature allows for more flexibility in the user's project compared to a project created in the standard Genie environment. One of the objects under the Genie Magic pane is the **Magic Object**. This is actually a 4DGL function which allows users to handle bytes received from an external host. The user, for example, can create a **Magic Object** that waits for 4 bytes (which can represent 32-bit integers) and writes the decimal equivalent value to a LED digits object.

Setup Procedure

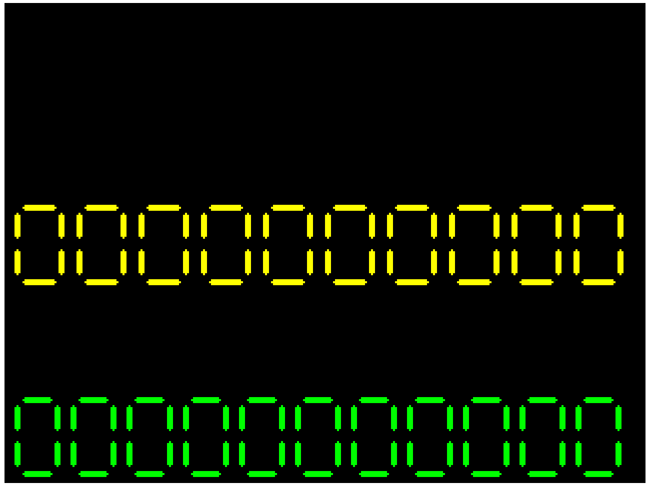
For instructions on how to launch Workshop 4, how to open a ViSi-Genie project, and how to change the target display, kindly refer to the section "**Setup Procedure**" of the application note:

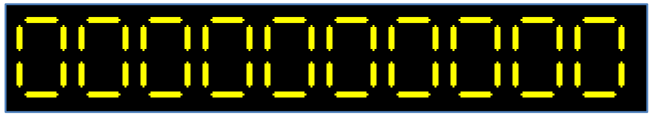
ViSi Genie Getting Started – First Project for Picaso Displays (for Picaso)
or
ViSi Genie Getting Started – First Project for Diablo16 Displays (for Diablo16).

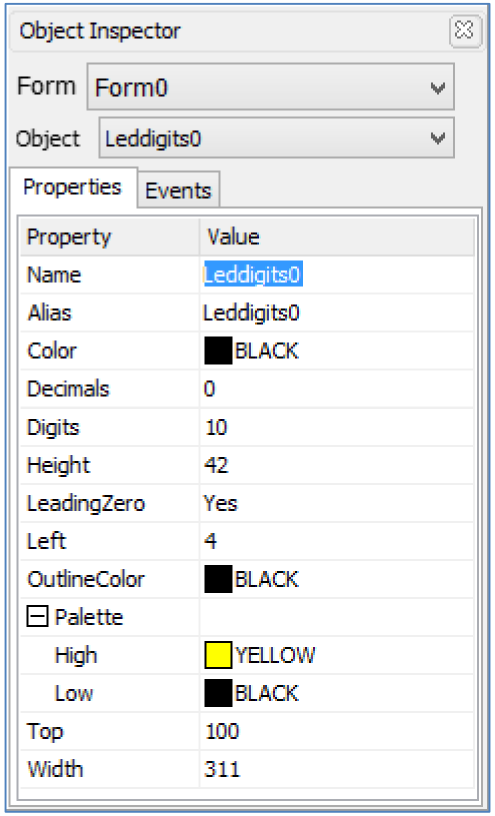
Design the Project

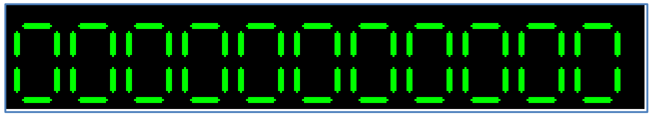
Add Two LED Digits Objects to Form0

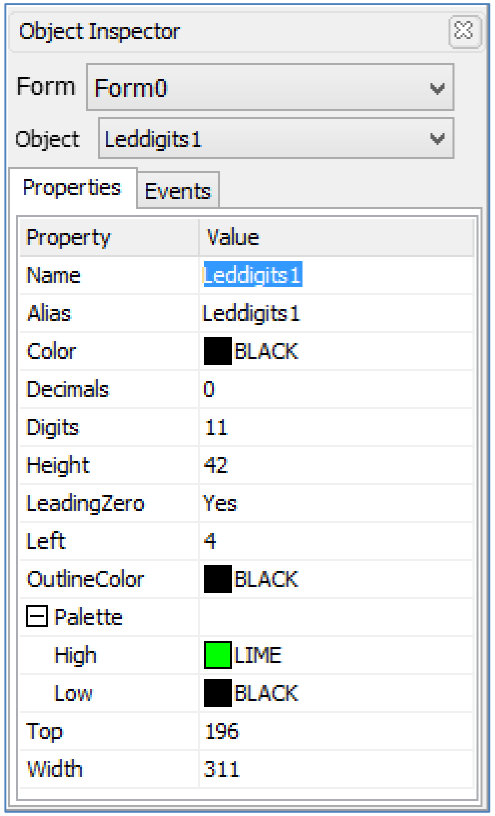
Two LED digits objects – **Leddigits0** and **Leddigits1** – are added to **Form0**.









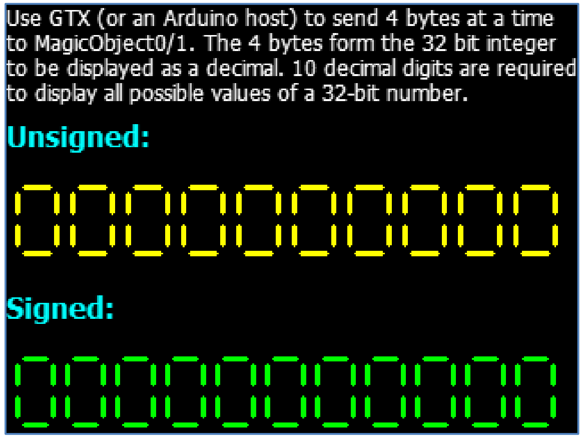


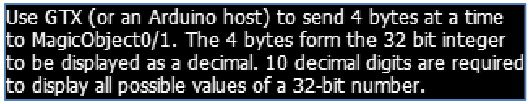
To know more about LED digits objects, their properties, and how they are added to a project, refer to the application note

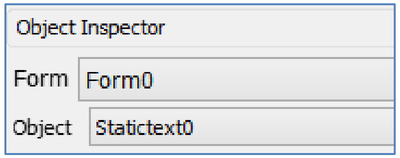
ViSi-Genie Digital Displays

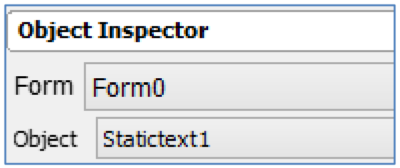
Add Three Static Text Objects to Form0

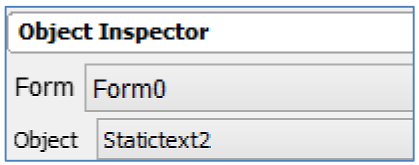
These are **Statictext0, Statictext1,** and **Statictext2**.









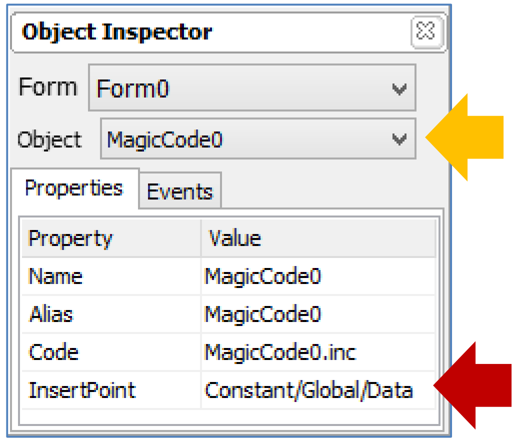


To know more about static text objects, their properties, and how they are added to a project, refer to the application note

[ViSi-Genie Labels, Text, and Strings]

Add a Magic Code Object to Form0

A Magic Code object is added to Form0. This is **MagicCode0**.

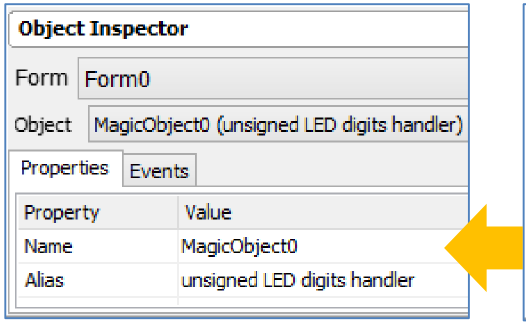


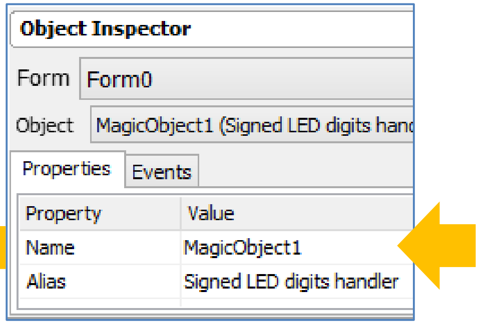
You may open **MagicCode0** of the attached project and copy the 4DGL code to your new project. To know more about the Magic Code objects, their properties, how they are added to a project, and how their codes are opened and edited, refer to the application note

[ViSi-Genie How to Add Magic Objects]

Add Two Magic Object Objects to Form0

Two Magic Object objects – **MagicObject0** and **MagicObject1** – are added to **Form0**. Take note that each has its own alias.





You may open **MagicObject0** and **MagicObject1** of the attached project and copy the 4DGL codes to the appropriate objects in your new project. To know more about the Magic Object objects, their properties, how they are added to a project, and how their codes are opened and edited, refer to the application note

[ViSi-Genie How to Add Magic Objects]

Diagrams

Attached is a PDF file (**programFlow.pdf**) containing several diagrams that attempt to help the user understand how the application works. Knowledge of 4DGL strings is the key to understanding the diagrams.

Diagram A – Program Flow 1

In Diagram A there are two classifications of processes – internal and external. Internal processes are those that are performed inside Genie and are hidden from the user. External processes are those that are defined by the user through the use of the magic objects. Of course, all of the processes are actually inside Genie when the entire project is compiled. The processes are classified as such only to facilitate this discussion. Note that the function "**writeToLeddigits(…)**" is a function defined in **MagicCode0**.

When the Genie program receives a message of the type "**WRITE_MAGIC_BYTES**", it checks the index of the **Magic Object** object for which the message was intended and then calls on the appropriate **Magic Object** object routine. Several processes are performed inside **MagicObject0** and **MagicObject1**, as will be shown later. Note that each Magic Object object eventually calls the function "**writeToLeddigits(…)**" before returning the control to the Genie program.

Diagram B – Program Flow 2

Diagram B shows the key processes that are performed inside the Magic Object objects **MagicObject0** and **MagicObject1** and the function "**writeToLeddigits(…)**". These key processes will be discussed in more detail later.

Diagram C – Program Flow 3

Diagram C is another version of Diagram B – Program Flow 2. Diagram C focuses on the sequence of the key processes that need to be performed. It further shows how these processes are distributed to the objects of the project.

Diagram D – Data Storage Array

We will now discuss how the application works. First we again take note that when the Genie program receives a message of the type "**WRITE_MAGIC_BYTES**", it checks the index parameter then calls on the appropriate **Magic Object** object. Genie internally stores the data contained by the message into an array and provides us, the user, access to this array thru the argument "**var *ptr**", which is the address of the array. In the generated 4DGL code for **MagicObject0**, the argument is indicated below.

```
func rMagicObject0(var action,
                   var object, var newVal, var *ptr)
```

In diagram D, observe how the bytes of the received data are arranged and stored. The addresses shown are just an example.

Diagram E – Conversion to a 4DGL 32-bit Integer

In 4DGL, 32-bit integers are stored in a manner illustrated on the lower part of Diagram E. When using 4DGL string class functions to print the value of a 32-bit integer, the processor expects that the bytes of the 32-bit integer are properly arranged. Hence, we need to copy the bytes of the received data (from ***ptr***), properly rearrange, and store them to another array (***bytes***). To do this we write,

MagicObject0.inc

```
    // arrange the received bytes
    bytes[1] := ptr[0] << 8 + ptr[1];
    bytes[0] := ptr[2] << 8 + ptr[3];
```

The process is illustrated in Diagram E. For more information on 4DGL strings, refer to the following application notes:

1. Designer or ViSi Strings and Character Arrays
2. Designer or ViSi 4DGL Strings Print Formats – the String and Character Format Specifiers
3. Designer or ViSi 4DGL Strings Print Formats – the Long Decimal Format Specifiers

Diagram F – Print the 32-bit Integer

Print the Value to the Display

After ensuring that the 32-bit integer is stored in ***bytes***, we can now print its decimal equivalent. We can print the decimal equivalent value directly to the screen, as shown below.

MagicObject0.inc

```
    p1 := str_Ptr(bytes);

    str_Printf(&p1, "%10lu");    // print to
```

The code snippet above makes use of a **byte-aligned pointer** and the 4DGL string class function "**str_Printf(…)**" to print the decimal equivalent of a 32-bit integer stored in ***bytes***. For more information on byte-aligned pointers and the use of the str_Printf(…) function, refer to the following application notes:

1. Designer or ViSi Strings and Character Arrays
2. Designer or ViSi 4DGL Strings Print Formats – the String and Character Format Specifiers
3. Designer or ViSi 4DGL Strings Print Formats – the Long Decimal Format Specifiers

Print the Value to an Array

It is also possible to print the decimal equivalent value of the 32-bit integer to an array, such that the array contains the characters of the printed value. The array is therefore essentially a string. This is done by streaming the printed characters to an array.

The process of streaming the printed value to an array is performed by the function "**writeToLeddigits(…)**", which is defined inside the Magic Code object **MagicCode0**. Before returning to main, MagicObject0 calls the function writeToLeddigits(…) as shown below.

MagicObject0.inc

```
    p1 := str_Ptr(bytes);

    str_Printf(&p1, "%10lu");    // print to

    writeToLeddigits(bytes, 0, UNSIGNED);
```

Note that we passed the address of ***bytes*** as the first argument, the index of **Leddigits0** as the second argument, and the constant **UNSIGNED** as the third argument. This will let **writeToLeddigits(…)** know where to get the 32-bit integer (***bytes***), what LED digits object to use (***Leddigits0***), and what format to use (**UNSIGNED**).

Inside the function **writeToLeddigits(…)**, the array ***buffer*** is created. This will contain the characters of the printed decimal equivalent. Note that ***buffer*** here has a size sufficient enough to hold up to 30 characters including the null terminator.

MagicCode0.inc

```
func writeToLeddigits(var address, var ind
    var ledDigitsPr;            // pointer to
    var left, digits, width;
    var i, j, k, c, lb, x;
    var length;
    var buffer[15];
```

To the array ***buffer*** we now stream the decimal equivalent value of the 32-bit integer stored in ***bytes***.

MagicCode0.inc

```

    j := str_Ptr(address);

    if(format == UNSIGNED)
        to(buffer); str_Printf(&j, "%lu");
    else if(format == SIGNED)
        to(buffer); str_Printf(&j, "%ld");
    endif
```

Note that ***j*** is a byte-aligned pointer to ***address***, the address of which in memory is the same as that of ***byte***. At this point, we now have a string stored inside the array ***buffer***, as illustrated in Diagram F.

For more information on the storage of 4DGL strings, byte-aligned pointers, the use of the str_Printf(…) function, and the use of the long signed and long unsigned decimal format specifiers, refer to the following application notes:

1. Designer or ViSi Strings and Character Arrays
2. Designer or ViSi 4DGL Strings Print Formats – the String and Character Format Specifiers
3. Designer or ViSi 4DGL Strings Print Formats – the Long Decimal Format Specifiers

Diagram G – Display the Characters

The last step is now to get all the characters stored in ***buffer***, find the corresponding LED digit for each character, and display the digit at the proper location. To properly display the digits, we have to know where to start displaying (the property "***left***"), the number of digits allowed for a LED digits object (the property "***digit***"), and the width of a single digit (the property "***width***"). We extract these properties from a certain RAM location, the starting address of which is specified by the variable "***oLedDigitsn***". Note that ***oLedDigitsn*** is internal to Genie.

MagicCode0.inc
```

    ledDigitsPr := oLedDigitsn + 10 * index;

    left     := ledDigitsPr[Ofs_Digits_Left];
    digits   := ledDigitsPr[Ofs_Digits_Digits];
    width    := ledDigitsPr[Ofs_Digits_Widthdigit]

```

The extraction of the characters from inside the string is done using the 4DGL string class function "**str_GetByte(…)**".

MagicCode0.inc
```

    x := left + (width * (digits - length));
    for (i := 0; i < length; i++)
        c := str_GetByte(k++);
        if ((c == 0x30) && (lb))        // al
            c := 10 ;
```

The displaying of the frames of a LED digit image is done using the functions "**img_SetWord(…)**" and "**img_Show(…)**".

MagicCode0.inc
```
            else
                c &= 0x0F ;
            endif
            img_SetWord(hndl, uSDidx[index], IMAG
            img_SetWord(hndl, uSDidx[index], IMAG
            img_Show(hndl, uSDidx[index]);
            x += width ;
        next
```

img_SetWord(…)" and "**img_Show(…)**" are examples of 4DGL image control functions. To know more about them, refer to the following application notes.

1. [ViSi Displaying Images from the uSD Card - WYSIWYG FAT16]
2. [ViSi Images and User Images]

Understanding how the remaining part of the **writeToLeddigits(…)** function works is now left to the reader as an exercise.

Build and Upload the Project

For instructions on how to build and upload a ViSi-Genie project to the target display, please refer to the section "**Build and Upload the Project**" of the application note

ViSi Genie Getting Started – First Project for Picaso Displays (for Picaso)
or
ViSi Genie Getting Started – First Project for Diablo16 Displays (for Diablo16).

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

Identify the Messages

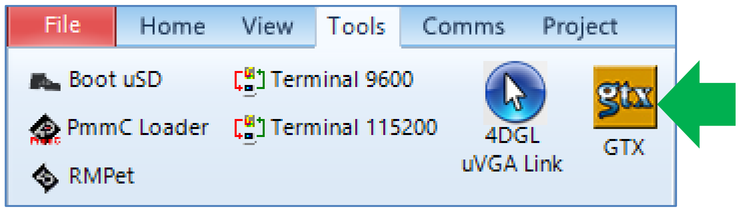
The display module is going to send messages to an external host. This section explains to the user how to interpret these messages. An understanding of this section is necessary for users who intend to interface the display to a host. The ViSi Genie Reference Manual is recommended for advanced users.

Use the GTX Tool to Analyse the Messages

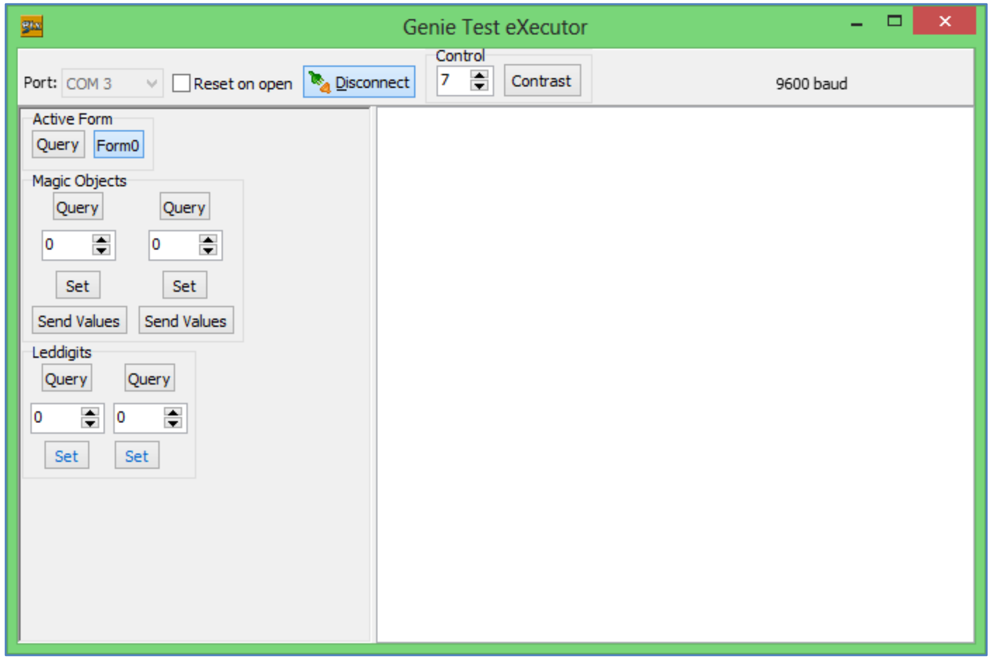
Using the GTX or **Genie Test eXecutor** tool is one option to get the messages sent by the display to the host. Here the PC will be the host. The GTX tool is a part of the Workshop 4 IDE. It allows the user to receive, observe, and send messages from and to the display module. It is an essential debugging tool.

Launch the GTX Tool

Under the Tools menu click on the GTX tool button.

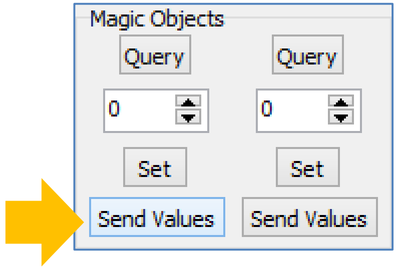


The Genie Test eXecutor window appears.

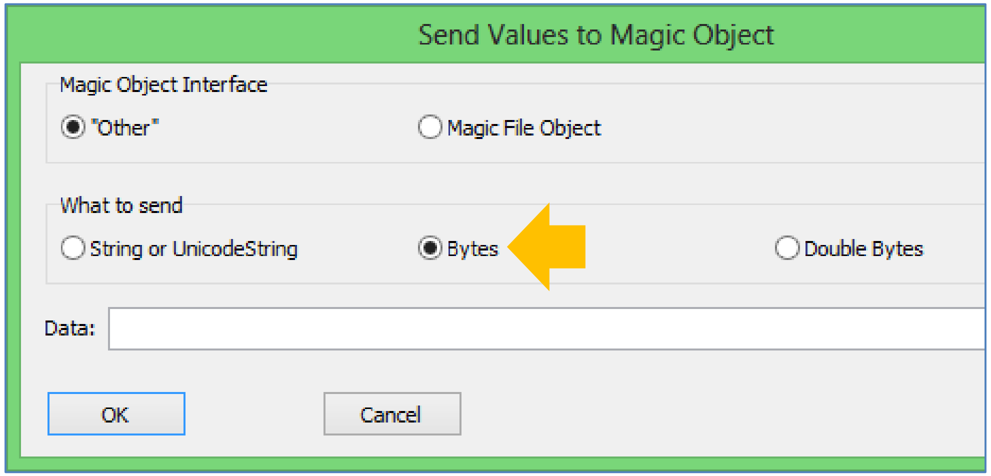


Send a WRITE_MAGIC_BYTES Message

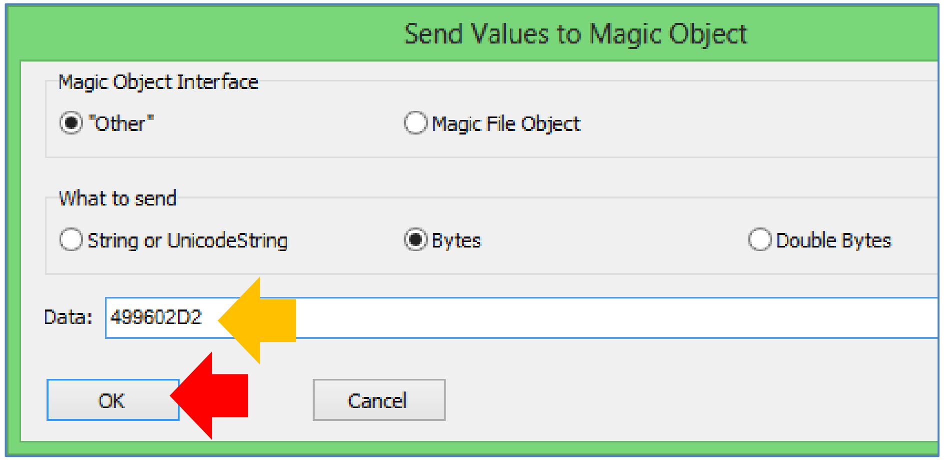
Click the Send Values Button for MagicObject0



Select "Bytes"



Input a 32-bit Hexadecimal Number



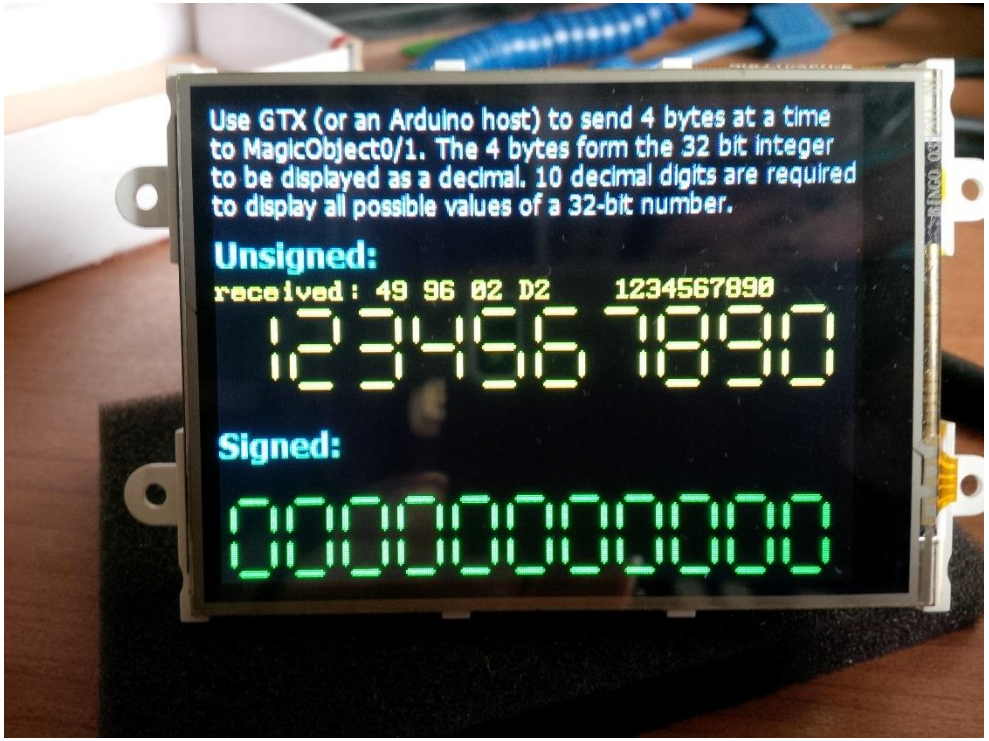
Click OK. To the right part of the window, the message sent is shown in green font.

```
Set MagicObject byte Value 12:39:32.275 [08 00 04 49 96 02 D2 03]
```

The format of the messages is shown below.

Command	Object Index	Length	Byte1	Byte 2	Byte 3	Byte 4	Check sum
0x08	0x00	0x04	0x49	0x96	0x02	0xD2	0x03

Upon receiving the above message, the display module will now display the equivalent decimal value using **Leddigits0**, like as shown below.



Proprietary Information

Disclaimer of Warranties & Limitation of Liability