APPLICATION NOTES

4D SYSTEMS
TURNING TECHNOLOGY INTO ART

ViSi – Read and Write Data on Diablo16 Flashbanks

DOCUMENT DATE:	**21st May 2019**
DOCUMENT REVISION:	**1.1**

W W W . 4 D S Y S T E M S . C O M . A U

Description

This document is intended to provide fundamental information on how to write and read information to-and-from a Diablo16 FlashBank.

It contains a working example that has been dissected to properly demonstrate the manner by which the process is done. The example for this application note makes use of a text file initially stored on a micro-SD. The file will then be copied to the FlashBank and then accessed to display the information that was contained therein.

It should be noted that this application note is limited to using the currently available flash memory allotted for FlashBanks that is 32Kb.

Before getting started, the following are required:

- 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.

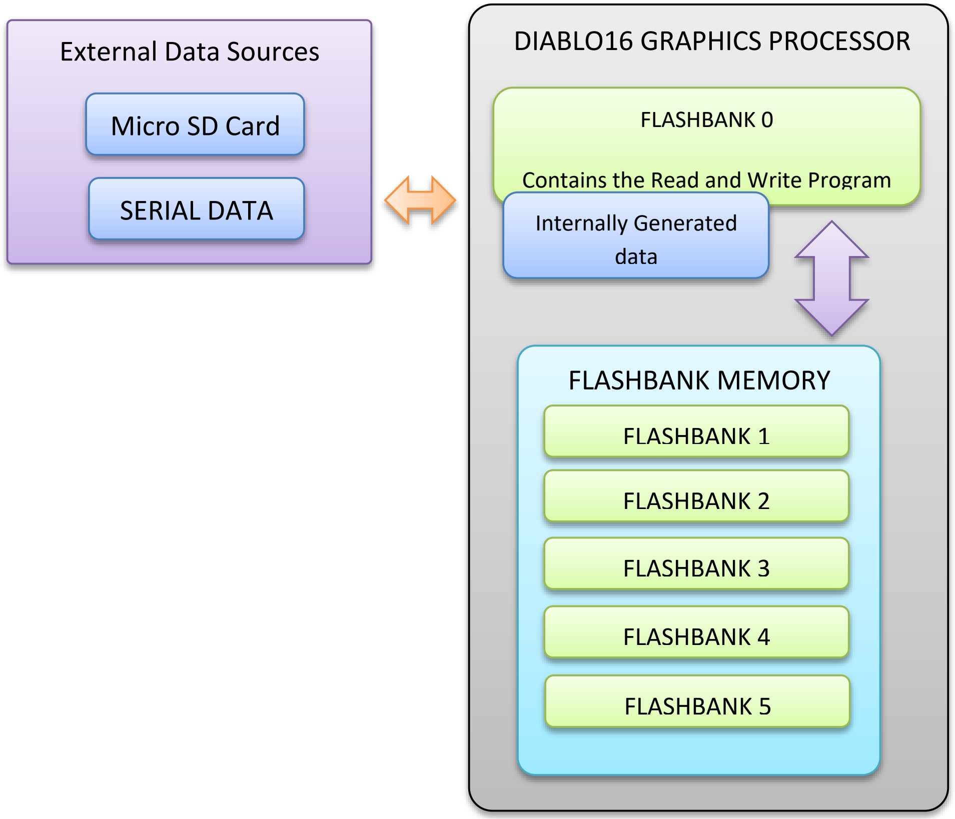
- 4D Programming Cable / µUSB-PA5/µUSB-PA5-II for non-gen4 displays (uLCD-xxx)
- 4D Programming Cable & gen4-IB / gen4-PA / 4D-UPA, for gen-4 displays (gen4-uLCD-xxx)
- Workshop 4 IDE (installed according to the installation document)
- When downloading an application note, a list of recommended application 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

Description 2
Content 3
Application Overview 3
Sources of information 4
Data from a micro-SD 4
The micro-SD source file 4
Copying a file to the FlashBank 5
Accessing the file copied to FlashBank 5
Processing the data stored on FlashBank 5
Conversion of ASCII to HEXADECIMAL 6
Proprietary Information 7
Disclaimer of Warranties & Limitation of Liability 7

Application Overview

The figure below is a representation of how to the process of reading and writing to FLASHBANK Memory. There are 6 available flash memory banks in the Diablo16 graphics processor. The primary program to handle the main program is assigned to FLASHBANK 0. For demonstration purposes, the data source used in this application note is a micro-SD card and FLASHBANK 1 was used as the target for saving/storing information.



Sources of information

Data information that can be stored in to the flash memory banks can be retrieved from several sources. Internal data is the first to be presented. During the course of a running program, there are often times a need to save information that are generated within the program such as calibration information, settings, and several others. Writing the data to a particular FlashBank will save time, effort, and eliminates the process of having to re-set up a device during boot-up.

On some other instances, there are also information that needs to be saved into the graphics processor that are required to be done during runtime. These kinds of information can be saved on a removable devices such as the micro-SD and then transferred/copied onto the graphics processor.

Having known the possible sources of information, it is a must that the user understands the limitation of the process. A FlashBank can only contain up to a maximum of 32Kb of data. Storing information that exceeds the pre-defined limit will bring about problems and loss of data. So, it is a must that all information written are limited to the data size mentioned.

Data from a micro-SD

The micro-SD source file

Prior to use, the micro-SD card was loaded with a file that contains an array of data. File sources that are to be copied to the micro-SD can have user-defined filenames as long as it is limited to having 8 characters. For extension filenames it can be saved as *.txt or *.dat. For this example a file was saved to the micro-SD with this filename – sample.dat. The content is formatted in the following manner.

sample[2500]={FF60, FF60, FF60, FF60, FF60, FF60,…….

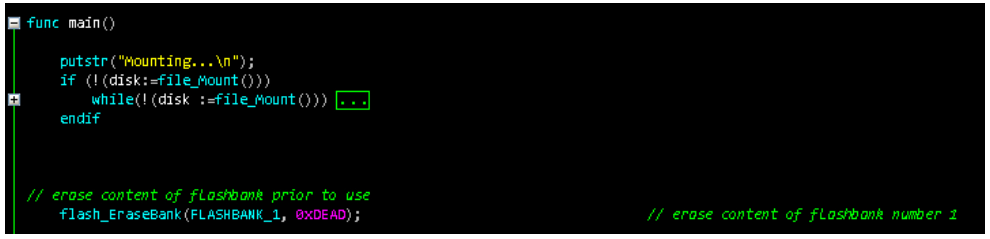
Notice the content of the text file, first is the name of the array followed by the size. This is one way of identifying the size of the contained array so tracking of the content is also easy. Furthermore, the data is saved with 16-bits of information but does not follow the conventional standard for hexadecimal writing, that is – 0xFFFF.

When information source are obtained from a file, it would mean that the content is considered to be written in ASCII form. This means that ‘A’ is not equal to 10, rather ‘A’ is equal to 0x41. This is according to the ASCII values tables. So if the user intends to use the information for retrieving alphanumeric characters then there would be not need to make the conversion.

On the other hand, if there is a need to make use of the information in the file like in this application then a conversion is required. The process conversion will be shown on the following sections.

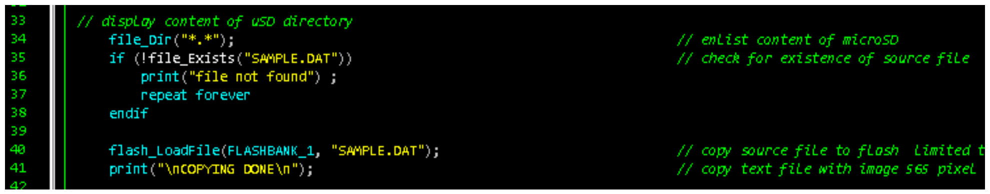
Copying a file to the FlashBank

A file can easily be copied to the FlashBank using the built-in functions of the graphics processor. Consider the following statements. First the micro-SD is initialized and then the target FlashBank is prepared to accept the file through erasing the content.



```
func main()

    putstr("Mounting...\n");
    if (!(disk:=file_Mount()))
        while(!(disk :=file_Mount())) ...
    endif


// erase content of flashbank prior to use
    flash_EraseBank(FLASHBANK_1, 0xDEAD);                       // erase content of flashbank number 1
```

Following the preparation of the FlashBank, the content of the micro-SD is checked. The source filename is checked for presence in the micro-SD. If the file is existing then the file copying proceeds.



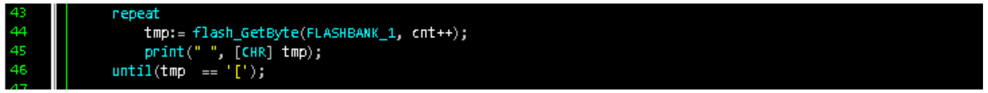
```
// display content of uSD directory
    file_Dir("*.*");                                        // enlist content of microSD
    if (!file_Exists("SAMPLE.DAT"))                         // check for existence of source file
        print("file not found") ;
        repeat forever
    endif

    flash_LoadFile(FLASHBANK_1, "SAMPLE.DAT");              // copy source file to flash  limited t
    print("\nCOPYING DONE\n");                              // copy text file with image 565 pixel
```

The main program will load the source file "SAMPLE.DAT" to the target-FlashBank_1. After the copying process it will give a simple notification that the copying is done.

Accessing the file copied to FlashBank

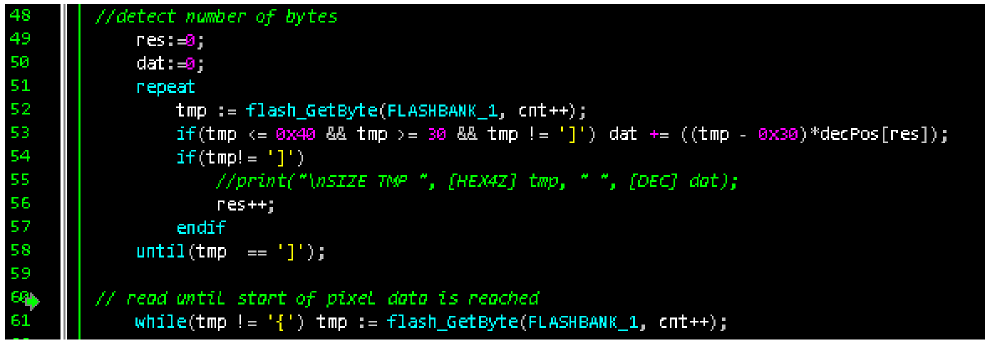
Processing the data stored on FlashBank

The first step on the process of reading the data is to identify the size of the saved array – sample[2500]. In this case the array 'sample' is given to be 2500 but this is not readily available for the program. A portion of the main program must have a simple routine to check and detect the size. For the detection the statements below searches for a particular character within the data file and continuously reads characters until the '[' is read. The reading is done with the use of the flash_GetByte(target flashbank, counter). The counter is used to increment the data location to be read.



```
    repeat
        tmp:= flash_GetByte(FLASHBANK_1, cnt++);
        print(" ", [CHR] tmp);
    until(tmp  == '[');
```

If the control character is read, the next step on the process is extract the size (originally ASCII) and convert this to a decimal value. This is done with the following statements.

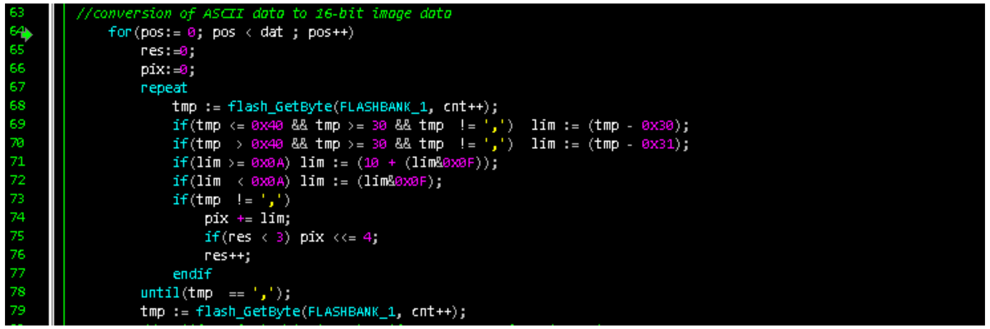


```
//detect number of bytes
    res:=0;
    dat:=0;
    repeat
        tmp := flash_GetByte(FLASHBANK_1, cnt++);
        if(tmp <= 0x40 && tmp >= 30 && tmp != ']') dat += ((tmp - 0x30)*decPos[res]);
        if(tmp!= ']')
            //print("\nSIZE TMP ", [HEX4Z] tmp, " ", [DEC] dat);
            res++;
        endif
    until(tmp  == ']');

// read until start of pixel data is reached
    while(tmp != '{') tmp := flash_GetByte(FLASHBANK_1, cnt++);
```

After the conversion, an extra reading of byte if performed to simply increment the byte location and discard the ‘]’ character until the start of the array is reached as marked by ‘{’ character.

Conversion of ASCII to HEXADECIMAL

As previously mentioned the original content of the file is written in ASCII, so conversion of signal is needed to utilize the data stored in the FlashBank.



```
//conversion of ASCII data to 16-bit image data
   for(pos:= 0; pos < dat ; pos++)
       res:=0;
       pix:=0;
       repeat
           tmp := flash_GetByte(FLASHBANK_1, cnt++);
           if(tmp <= 0x40 && tmp >= 30 && tmp  != ',')  lim := (tmp - 0x30);
           if(tmp  > 0x40 && tmp >= 30 && tmp  != ',')  lim := (tmp - 0x31);
           if(lim >= 0x0A) lim := (10 + (lim&0x0F));
           if(lim  < 0x0A) lim := (lim&0x0F);
           if(tmp  != ',')
               pix += lim;
               if(res < 3) pix <<= 4;
               res++;
           endif
       until(tmp  == ',');
       tmp := flash_GetByte(FLASHBANK_1, cnt++);
```

The first step in the process is to read the group of ASCII characters that represent a decimal value, i.e. FF60, which is 65376 in decimal (0xFF60 HEX). Referring to the ASCII table, number 0~9 are in the range of 0x30 to 0x39, respectively. So with this consideration, conversion of numerical characters into decimal values will require subtraction of 0x30. On the other hand for alpha characters, ‘A’ starts with an ASCII value of 0x41. So to convert the ‘A’ into a decimal value of 10, there is a need to subtract a hex value of 0x31.

After the subtraction of the excess value the bytes are then arranged through bit shifting. Afterwards these bytes are combined to form the original value therefore making the actual hexadecimal value.

The hexadecimal FF60 is followed by a ‘,’. The comma character will mark the end of the group and hence, the next group can be read. Referring back to the statements, an extra reading of byte is done to simply read-out the character ‘space’ in between ‘,’ and the next group of ASCII characters.

** Note: All characters that are existent in the source file, including space and ‘\n’ should be considered when creating a routine to read the data store in a FlashBank.

Proprietary Information

Disclaimer of Warranties & Limitation of Liability