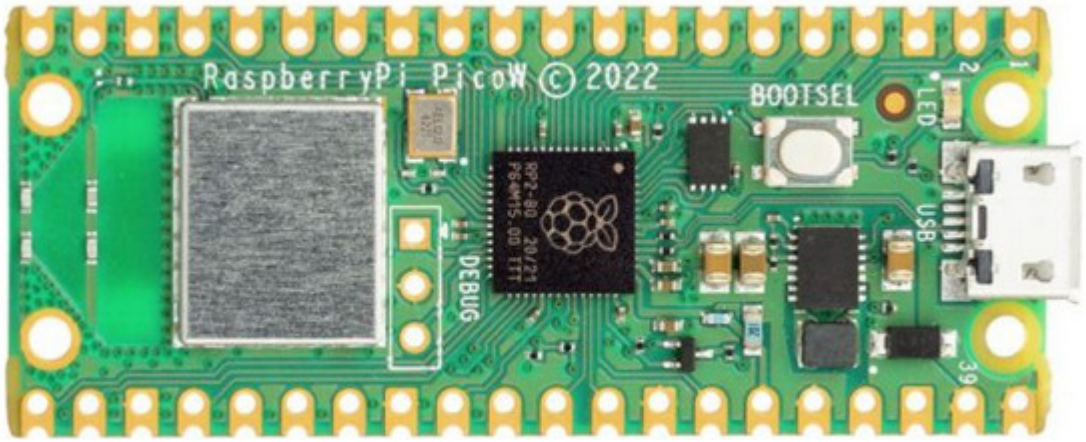
Pico-Flash-Driver



Library allowing your project to save and retrieve data to Pico's flash memory (for example for configuration data)

Firmware Version 2.00
User Guide
Updated September 2nd, 2024

IMPORTANT :

This User Guide is about Pico-Flash-Driver Firmware Version 2.00 from Andre St. Louys.
To get the full potential of this library, you must not simply clone the GitHub repository. Read this User Guide to get the most from this library.

Join our Pico-Flash-Driver discussion group on:
https://github.com/astlouys/Pico-Flash-Driver/discussions

Index

Pico-Flash-Driver User Guide

Introduction

The Pico-Flash-Driver « library » is a C language utility that will allow your Pico's program / firmware to save data in Pico's flash memory and retrieve this data whenever needed, even after a power failure and / or power cycle (Off / On).

I used the term « driver » in the name « Pico-Flash-Driver », but it is not a « driver » in the usual technical terminology. I could have used the word « support » « utility » or « helper », but I decide to use « driver ». I do have other similar utilities, and using « Driver » in the name simply help me to categorize them.

The Pico-Flash-Driver is not a standalone utility but rather a « library » that must be added / linked to your program's source code. Make sure to follow the setup procedure later in this guide to understand how to best configure and use it. Those procedures assume that you followed the usual (« standard ») directory structure configuration to organize your development system. If you did not follow those standards, you will need to read those instructions "mutatis mutandis".

The Pico-Flash-Example.c is an example showing how the Pico-Flash-Driver may be integrated in a project to access Pico's flash memory. It is a small program that makes nothing really useful, if not giving you a starting point on how to use the Pico-Flash-Driver.

Take note that the Pico's flash memory lifecycle is estimated to more or less 100,000 write cycles. To write a configuration and / or save some data once in a while, this is more than enough. However, if you plan to make many write cycles every second, this library may not be a good starting point since there is no wear-leveling algorithm included.

Files description

The Pico-Flash-Driver library contains the following files:

Filename	Description
Pico-Flash-Driver.c	Main library program file.
Pico-Flash-Driver.h	Include file that must be « #included » in your program.
Pico-Flash-Example.c	A simple example program that makes use of the Pico-Flash-Driver library. You may refer to this example to see how the library may be integrated in a program.
baseline.h	Include file that I use in most of my programs. If you have such a program that you use in many of your programs, it may be a good idea to create a « utility » directory in the « pico » directory and simply create a symbolic link to those often used files in every project directories.
CMakeLists.txt	The CMake file used for building the Pico-Flash-Example. It has been used under Visual Studio Code on a Raspberry Pi 5 development system. It is supposed to be relatively standard, but some fine tune may be required depending on your own development environment.
pico_sdk_import.cmake	You will probably wonder why this file is not in the GitHub repository. Of course, it is required to build Pico-Flash-Example. I recommend creating a symbolic link to the original SDK file. This way, every time you update your SDK, you simply need to build you programs since the symbolic link already points to the latest version of the file. From your project directory: ln -s /home/pi/pico/pico-sdk/external/pico_sdk_import.cmake

Setup procedure for the Pico-Flash-Example project

Follow this procedure to properly setup the library and build the example program on your system:

1) Create the directory « /home/pi/pico/Pico-Flash-Driver »
   `mkdir /home/pi/pico/Pico-Flash-Driver`

2) Copy the files « Pico-Flash-Driver.c », « Pico-Flash-Driver.h », « Pico-Flash-Example.c » and « CMakeLists.txt » to the directory you just created.

3) You can copy directly the « baseline.h » file to the same directory or put it in a « /pico/utility » directory and create a symbolic link so that you can use it through many programs using the same basic variables / definitions. When you update your baseline.h file, it is then automatically updated in all your projects through the symbolic link.

4) Create a symbolic link to the pico_sdk_import.make file:
   `ln -s /home/pi/pico/pico-sdk/external/pico_sdk_import.cmake`

5) You should be able to build the Pico-Flash-Example program at this point.

Setup procedure for your project

Follow this procedure to properly setup your own program to use the Pico-Flash-Driver library. This assumes you followed the recommended directory structure for Pico development and also that you followed the procedure described in the previous section for the Pico-Flash-Driver setup:

1) From you project directory, create the following symbolic links:
   ```
   ln -s ../Pico-Flash-Driver/Pico-Flash-Driver.c
   ln -s ../Pico-Flash-Driver/Pico-Flash-Driver.h
   ln -s ../Pico-Flash-Driver/baseline.h
   ```
   For the file « baseline.h », if you put it in a centralized utility directory, adapt the symbolic link above accordingly.

2) In your project C source file, add the « Pico-Flash-Driver.h » include file:
   ```
   #include "Pico-Flash-Driver.h"
   ```

3) In your project C source file, create the structure that will contain the data you want to save to flash memory. I suggest the structure name « flash_data » and a variable name of « FlashData » for this variable, but any name can do the job. It is a good idea to make it 4096 bytes long since this is the basic sector size that is read / write from / to flash. Simply add placeholders for unused variables (see Pico-Flash-Example.c).

4) What is important, however, is that the last member of this structure must be an unsigned 16-bit variable (« UINT16 ») and must be named « Checksum », or « Crc », or the like. In other words, the last UINT16 of the structure is reserved for internal use (reserved).

5) It is important to remember that writing to flash memory with the Raspberry Pi Pico is limited to a chunk of a maximum of one sector size (which is 4096 bytes or 0x1000). If you want to save more data, you will need to split the data in smaller chunks (maximum of 4096 bytes each).

6) In your project`s CMakeLists.txt file, in the section « add_executable », add the source file « Pico-Flash-Driver.c ».

7) Also in your project CMakeLists.txt file, in the section « target_link_libraries », add the following if they are not already there: « hardware_flash », « hardware_irq » and « pico_multicore ».

8) Reading comments in the Pico-Flash-Example.c and Pico-Flash-Driver.c may also help you to better understand how to use it.

How to use

You will usually make a call to « flash_read_data() » when your program starts, to retrieve the configuration as it has been saved to flash memory (you must plan for the first read, when flash contains garbage – probably all 0xFF). You will have to cast your structure pointer to a unsigned int 8 pointer « UINT8 * » in your call:

```
flash_read_data(FLASH_DATA_OFFSET1, (UINT8 *)&FlashData, sizeof(FlashData))
```

When coming back from the « flash_read_data() » call, you may want to confirm the integrity of the configuration read back. To do so, you will want to make sure that the Checksum read back from flash corresponds to the checksum computed on the actual data. The function « flash_extract_crc() » will return the CRC saved to flash (you can also, more easily, used the UINT16 variable that you declared as the last member of your configuration structure. You will compare this checksum with the one calculated with the configuration data with the help of the « util_crc16() » function. If both checksum match, it means that the configuration read back from flash is OK (it is the same as the one computed from the current structure in RAM). Be aware that when using the « util_crc16() » function, the DataSize must be 2 bytes (one « UINT16 ») less than your structure size, in order to exclude the « CRC » structure member itself from the crc calculation. If both checksum differ, either it is a first read (with garbage in flash), or there has been some type of corruption. In such a case, you may want to assign default values to your variables.

How to use the « flash_save_data() » depends of the algorithms used in your program. You may want to save a new copy of the configuration as soon as something changes in its content. You may also setup a callback function that is called, say, every hour, and that will compute the current checksum of your data structure (on RAM). If something has changed, the computed checksum will not be the same as the one that is the last member of the structure (and that has been read back from flash on the last « flash_read_data() »). If so (if both checksums do not match), it means that something has changed in the current data in RAM and you may want to save the new data. I want to repeat once more that you should not use those functions to write to flash many times a second since no « wear levelling » algorithm has been implemented and you will rapidly reach the expected write cycle life of your Pico (more or less 100,000 writes cycles).

You may refer to the program Pico-Flash-Example.c that is very simple and shows how the Pico-Flash-Driver may be implemented in your code.

« flash_erase() » and « flash_write() » should usually not be used outside of the « flash_driver.c » file (although I didn't make them « static »).