

# Writing Configuration Parameters to FLASH on the PAC52XX *Power Application Controller™*

**Marc Sousa**  
*Senior Manager, Systems and Firmware*



[www.active-semi.com](http://www.active-semi.com)  
Copyright © 2014 Active-Semi, Inc.

## OVERVIEW

Some applications need to be able to store configuration information in FLASH, which allows the application to customize the system according to a customers need. This configuration information cannot be created at the time the firmware was created, and must be updated by the customer in the field.

The PAC52XX allows this operation, and this application note describes the method by which this can be accomplished.

### NOTE

*FLASH memory has a finite number of write cycles, before it becomes unusable. If the application writes FLASH too frequently, then FLASH memory will become unusable and will be unable to execute any firmware previously written to FLASH.*

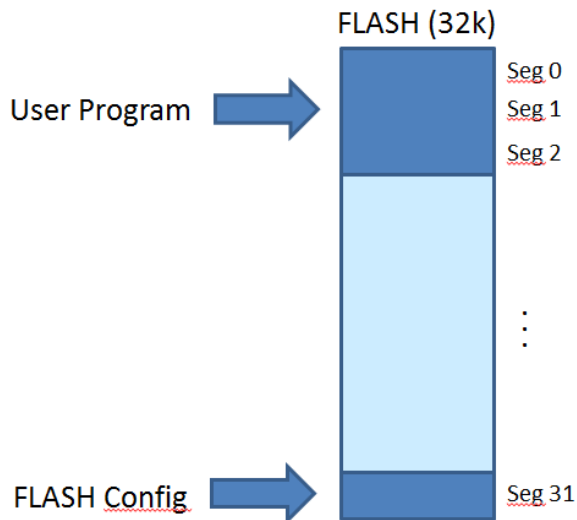
*The application designer should take great care to make sure that this operation executes only for configuration changes, and not during normal operation.*

## FLASH CONFIGURATION

When a user builds a program for the PAC52XX device, FLASH memory stores the program and data in one section of memory. In the PAC52XX, FLASH memory begins at address 0. The user program will start at address 0.

The FLASH memory on the PAC52XX has an available FLASH size of 32k. There are 32 1k FLASH segments.

In this example, the user program will start at address 0, and the FLASH configuration data will be stored in segment 31 (the last segment).



In the PAC52XX, FLASH memory may be erased one segment at a time. So the minimum amount of memory to be erased will be 1k.

For this example, the FLASH configuration information will be kept in segment 31. The user must take care to make sure the user FLASH program does not get any larger than 31 segments (so it does not over-write the FLASH configuration section).

## WRITING FLASH MEMORY

In order to write FLASH memory, the memory must first be erased. FLASH has to be erased each time before it is written. A segment at a time may be erased, and the segment size is 1k. So, the data to write into FLASH must have its segment erased on 1k boundaries before it may be written.

In order to erase FLASH segments, or to write data to erased FLASH memory, the CPU must be executing instructions out of RAM. You cannot both be executing out of FLASH and erasing or writing FLASH memory at the same time.

To instruct the linker to place any function in RAM, you need to do the following.

CooCox:

- Define the symbol "GCC" in Colde

IAR:

- Define the symbol "IAR" in IAR Embedded Workbench

After the tools have had these symbols properly defined, add the following symbol the beginning of the function signature in the source file: **PAC5XXX\_RAMFUNC**. This will tell the linker to place this function in RAM.

For example, the following function would be placed in RAM:

```
PAC5XXX_RAMFUNC void run_this_from_ram(int i)
{
    // This function running in RAM
    ;
}
```

The API functions in the SDK that configure the memory controller must also let the linker know to put these functions in RAM. To do this, the `pac5xxx_driver_config.h` file must enable the `PAC5XXX_DRIVER_MEMORY_RAM` symbol, as shown below:

```

/*
 *
 */

#ifndef PAC5XXX_DRIVER_CONFIG_H
#define PAC5XXX_DRIVER_CONFIG_H

// #define ALL_RAM

#ifdef ALL_RAM
#define PAC5XXX_DRIVER_GPIO_RAM /*!< Link GPIO driver functions in RAM
#define PAC5XXX_DRIVER_SPI_RAM /*!< Link SPI driver functions in RAM
#define PAC5XXX_DRIVER_SOCBRIDGE_RAM /*!< Link SOC Bridge driver functions in RAM
#define PAC5XXX_DRIVER_TIMER_RAM /*!< Link Timer driver functions in RAM
#define PAC5XXX_DRIVER_I2C_RAM /*!< Link I2C driver functions in RAM
#define PAC5XXX_DRIVER_SYSTEM_RAM /*!< Link System driver functions in RAM
#define PAC5XXX_DRIVER_UART_RAM /*!< Link UART driver functions in RAM
#define PAC5XXX_DRIVER_ADC_RAM /*!< Link ADC driver functions in RAM
#define PAC5XXX_DRIVER_MEMORY_RAM /*!< Link Memory Controller driver functions in RAM
#define PAC5XXX_DRIVER_WATCHDOG_RAM /*!< Link Watchdog Timer driver functions in RAM
#define PAC5XXX_DRIVER_RTC_RAM /*!< Link Real-time Clock driver functions in RAM
#define PAC5XXX_DRIVER_ARM_RAM /*!< Link ARM driver functions in RAM
#define PAC5XXX_DRIVER_TILE_RAM /*!< Link Tile driver functions in RAM
#else
// #define PAC5XXX_DRIVER_GPIO_RAM /*!< Link GPIO driver functions in RAM
// #define PAC5XXX_DRIVER_SPI_RAM /*!< Link SPI driver functions in RAM
// #define PAC5XXX_DRIVER_SOCBRIDGE_RAM /*!< Link SOC Bridge driver functions in RAM
// #define PAC5XXX_DRIVER_TIMER_RAM /*!< Link Timer driver functions in RAM
// #define PAC5XXX_DRIVER_I2C_RAM /*!< Link I2C driver functions in RAM
// #define PAC5XXX_DRIVER_SYSTEM_RAM /*!< Link System driver functions in RAM
// #define PAC5XXX_DRIVER_UART_RAM /*!< Link UART driver functions in RAM
// #define PAC5XXX_DRIVER_ADC_RAM /*!< Link ADC driver functions in RAM
// #define PAC5XXX_DRIVER_MEMORY_RAM /*!< Link Memory Controller driver functions in RAM
// #define PAC5XXX_DRIVER_WATCHDOG_RAM /*!< Link Watchdog Timer driver functions in RAM
// #define PAC5XXX_DRIVER_RTC_RAM /*!< Link Real-time Clock driver functions in RAM
// #define PAC5XXX_DRIVER_ARM_RAM /*!< Link ARM driver functions in RAM
// #define PAC5XXX_DRIVER_TILE_RAM /*!< Link Tile driver functions in RAM
#endif

/* */ /* end of group PAC5XXX_Driver_Config */

```

To write the configuration information to FLASH, be sure to follow the example above in order to make sure the calling function resides in RAM. An example function that is called to update the configuration information could then look like below.

```

// Call this function when user wants to write new config data into FLASH segment 31
// Be sure to not call frequently. Performing too many writes to FLASH may burn out FLASH
// and it will become unusable

```

```

// Also need to make sure that PAC5XXX_DRIVER_MEMORY_RAM symbol in
pac5xxx_driver_config.h is enabled, to
// compile the memory controller functions into RAM (needed to update FLASH)

```

```

// Note that this function *must* be linked into RAM or this operation will not work

```

```

PAC5XXX_RAMFUNC void write_config_area(uint16_t config_1, uint8_t config_2, uint8_t
config_3)
{

```

```

    FlashConfig* fc = (FlashConfig*)FLASH_CONFIG_ADDR;

```

```

    // First, erase segment 31

```

```

    pac5xxx_memctl_flash_page_erase(31);

```

```

    // Wait for erase to complete

```

```

    while (pac5xxx_memctl_page_erase_active() != 0);

```

```
// Prepare to write to erased FLASH segment

pac5xxx_memctl_flash_write_prep();

// Write passed in data to FLASH segment, and write valid key so user knows
data is good
// Hardware will stall each instruction, while write is taking place

fc->config_1 = config_1;
fc->config_2 = config_2;
fc->config_3 = config_3;
fc->valid_key = VALID_KEY;

}
```

In this function the first step is to erase the desired FLASH segment by calling the `pac5xxx_memctl_flash_page_erase(31)` function. This will erase FLASH segment 31.

After calling the function to erase a segment of FLASH, the CPU must wait until the erase is complete before beginning to write to FLASH memory. The user may do this by calling the `pac5xxx_memctl_page_erase_active()` function, to test if this operation has completed.

After the erase is complete, then you must prepare FLASH memory to be written, to try to prohibit unwanted writes to FLASH memory by calling the `pac5xxx_memctl_flash_write_prep()` function.

After this, you may write any FLASH in the segment you erased one time. You may not re-write any FLASH without first erasing the segment as shown above.

## READING CONFIGURATION INFORMATION FROM FLASH

Once FLASH has been written with configuration information as shown above, the firmware may read the contents of it at any time.

The program above writes a special key into FLASH, to indicate that the FLASH configuration has been written. The application firmware may read this key to determine if the contents of FLASH configuration are OK, as shown below.

```
// Set defaults

uint16_t c1 = 0xFACE;
uint8_t c2 = 0xAA;
uint8_t c3 = 0xBB;

int main(void)
{
    FlashConfig *fc = (FlashConfig*)FLASH_CONFIG_ADDR;

    // See if there is a valid key in FLASH, if so load data from FLASH.
    // If the key is not valid, just use default values

    if (fc->valid_key == VALID_KEY)
    {
        c1 = fc->config_1;
        c2 = fc->config_2;
        c3 = fc->config_3;
    }

    while (1);
}
```

In this function, the user program (which may be in FLASH or RAM) checks to see if the valid key is present in configuration FLASH memory. If it is, it uses the configuration values from FLASH.

If the contents of configuration FLASH are not valid, then the default values can be used, as shown above.