

# Technical Note

## Software Device Drivers for Micron® P8P Parallel Phase Change Memory

---

### Introduction

This technical note describes the library source code in C for Micron's P8P parallel phase change memory (PCM) device using the P8P software device driver.

This technical note also includes an overview of the programming model for the P8P device. It describes the operation of the memory device and provides a basis for understanding and modifying the accompanying source code.

The source code is available from your Micron distributor. The P8P.c and P8P.h files contain libraries for accessing the P8P parallel PCM device. The source code is written to be as platform independent as possible, and requires minimal changes to compile and run.

This technical note explains how to modify the source code for target hardware. The source code contains comments throughout that explain how it is used and why it has been written the way it has.

This technical note does not replace the P8P parallel PCM device data sheet. It refers to it throughout, and it is necessary to have a copy of it to follow some of the explanations.

The software supplied with this documentation has been tested on a target platform and is usable in C and C++ environments. It is small in size and can be applied to any target hardware.

### Programming Model

The 128Mb parallel PCM device can be electrically erased and programmed through special coded command sequences on most standard microprocessor buses. PCM combines the benefits of traditional floating gate Flash, with some of the key attributes of RAM and EEPROM. Like NOR Flash and RAM technology, PCM offers fast random access times. Like NAND Flash, PCM has the ability to write moderately fast. PCM is also similar to RAM and EEPROM in that it supports bit-alterable writes (overwrite). Unlike Flash, no separate erase step is required to change information from 0 to 1 and 1 to 0. However, unlike RAM, PCM technology is nonvolatile, with data retention comparable to that of NOR Flash.

The device has an asymmetrically-blocked architecture, is divided into four 32KB parameter blocks (in top or bottom configuration) and 128KB main blocks. Each block can be erased individually, and can be programmed and erased over 100 million cycles.

PROGRAM and ERASE commands are written to the command interface of the memory. An on-chip program/erase controller (P/E.C.) handles the timings necessary for PROGRAM and ERASE operations. The end of a PROGRAM or ERASE operation can be detected, and any error conditions identified. The command set required to control the memory is consistent with JEDEC standards.

An ERASE can be suspended to either READ from or PROGRAM to any other block and then be resumed. PROGRAM can be suspended to read data in any other block and then be resumed.

Each block can be protected individually against PROGRAM and ERASE operations. P8P device block locking enables block locking/unlocking and permanent locking. Permanent block locking provides enhanced security for boot code. The combination of these two locking features provides a complete locking solution for code and data.

**Note:** Data with the current CPU data bus width is referred to as elements throughout the document unless otherwise specified. Due to the flexibility of the software device driver, the size of an element depends on the current configuration (user change area).

## Bus Operations and Commands

Most P8P device functionality is available via the two standard bus operations: READ and WRITE. READ operations retrieve data or status information from the device. WRITE operations are interpreted by the device as commands that modify the data stored or the behavior of the device.

Only certain special WRITE operation sequences are recognized as commands by the P8P memory device. These commands are listed in the commands table provided in the corresponding data sheet. The main commands are classified as:

- Read
  - READ ARRAY
  - READ STATUS REGISTER
  - READ ID CODE
  - READ QUERY
  - CLEAR STATUS REGISTER
- Program
  - PROGRAM SETUP
  - ALT SETUP
  - BIT ALTERABLE WRITE
  - BUFFERED PROGRAM
  - BIT ALTERABLE BUFFERED WRITE
  - BUFFER PROGRAM ON ALL 1S
  - BUFFERED WRITE CONFIRM
- Erase
  - BLOCK ERASE SET-UP
  - ERASE CONFIRM
- Suspend/Resume
  - WRITE OR ERASE SUSPEND
  - SUSPEND RESUME
- Block locking
  - LOCK SETUP
  - LOCK BLOCK
  - UNLOCK BLOCK
  - LOCK DOWN
- Protection
  - PROTECTION PROGRAM SETUP
- Serial interface
  - WRITE ENABLE
  - WRITE DISABLE
  - READ IDENTIFICATION
  - READ STATUS REGISTER
  - WRITE STATUS REGISTER
  - READ DATA BYTES
  - FAST READ DATA BYTES
  - PAGE PROGRAM LEGACY
  - PAGE PROGRAM BIT ALTERABLE
  - SECTOR ERASE

The READ MEMORY ARRAY command returns the device to read mode where it behaves as a ROM. In this mode, a READ operation outputs the data stored at the specified device address onto the data bus.

The READ ELECTRONIC SIGNATURE command enables the user to read the manufacturer code, device code, protection register, and block protection status of the device. Once the command is issued, the data is accessed by reading different addresses.

The BLOCK ERASE command sets all bits to 1 at each memory location in the selected block. All data previously stored in the erased block will be lost. The BLOCK ERASE command takes longer to execute than the other commands because an entire block is erased at one time. Attempts to erase or program a block while the memory is protected (for example,  $V_{PEN} = V_{IL}$ ) generate an error and do not modify the contents of the memory.

The WORD PROGRAM command modifies the data stored at a single device address. Programming larger amounts of data should be performed using the WRITE TO BUFFER AND PROGRAM command.

The WRITE TO BUFFER AND PROGRAM command modifies the data stored at the specified device addresses. Each WRITE TO BUFFER AND PROGRAM command can program up to 32 words at a time. Programming larger amounts of data must be performed one buffer at a time by issuing a WRITE TO BUFFER AND PROGRAM command, waiting for the command to complete, then issuing the next WRITE TO BUFFER AND PROGRAM command, and so on.

Issuing the PROGRAM/ERASE SUSPEND command during a PROGRAM or ERASE operation temporarily places the P8P device in program/erase suspend mode. While a BLOCK ERASE operation is suspended, the blocks not being erased may be read or programmed as if in the reset state of the device. While a PROGRAM operation is suspended, the rest of the device may be read. This enables the user to access information stored in the device immediately, rather than waiting until the program or BLOCK ERASE operation completes. The PROGRAM or BLOCK ERASE operation resumes when the device receives the PROGRAM/ERASE RESUME command.

The READ QUERY command of the device reads data from the common flash interface (CFI), enabling the user to identify the number of blocks in the PCM device and the block addresses. The interface also contains information relating to the typical and maximum PROGRAM and ERASE times. Using this data, the user can implement software timeouts and avoid waiting indefinitely for a defective PCM device to finish programming or erasing. For further information about the CFI, please refer to the CFI specification available at [www.jedec.org](http://www.jedec.org) or from your Micron distributor. Blocks can be protected against accidental or malevolent PROGRAM and ERASE operations that would change their contents. Block protection on the P8P is nonvolatile. After power up or a hardware reset, the block protection remains in the state it was before the power-down or reset. Each block can be protected individually, and protected blocks cannot be unprotected individually.

The STATUS/(READY/BUSY), or STS, pin of the PCM device can be used to identify the program/erase controller status. It can be configured in two modes:

- Ready/Busy: The pin is LOW during PROGRAM and BLOCK ERASE operations, and High-Z when the memory is ready for any operation.
- Status: The pin gives a pulsing signal to indicate the end of a PROGRAM or BLOCK ERASE operation.

## Status Register

While the P8P device is programming or erasing, a READ from the device outputs the status register of the program/erase controller. The status register, which can also be accessed by issuing the READ STATUS REGISTER command, provides valuable information about the most recent PROGRAM or BLOCK ERASE command. The status register

bits are described in the Status Register Bits Table provided in the P8P device data sheet. Their primary use is to determine when programming or erasing is complete and whether it was successful.

Completion of the PROGRAM or ERASE operation is indicated by the program/erase controller status bit (status register bit DQ7) going HIGH (set to 1). Programming or erasing errors are indicated by one or more of the various error bits (status register bits DQ1, DQ3, DQ4, and DQ5) going HIGH. If a failure occurs, the status register error bits remain set until a CLEAR STATUS REGISTER command is issued to the device. This should be done before performing any further operations, or it will not be possible to determine whether subsequent operations are successful.

## A Specific Example

The Commands Table provided in the P8P device data sheet describes the write sequences recognized as valid commands by the program/erase controller.

As an example, to program 9465h to the address 03E2h for the P8P in 16-bit mode, the user must write the following C sequence:

```
* (NMX_uint16*) (0x0000) = 0x00E8; /* 1st cycle (any block
address)
*/
* (NMX_uint16*) (0x0000) = 0x0000; /* 2nd cycle: data length */
* (NMX_uint16*) (0x03E2) = 0x9465; /* 3rd cycle: address and data
*/
* (NMX_uint16*) (0x0000) = 0x00D0; /* final cycle: confirm and
start
*/
```

where NMX\_uint16 is defined as the following 16-bit value:

```
typedef unsigned short NMX_uint16
```

The first two addresses (0000h) are arbitrary. However, they must be located inside the block where the data is to be programmed. This example assumes that address 0000h in the P8P device is mapped to address 0000h in the microprocessor address space. In practice, the PCM device is likely to have a base offset that must be added to the address.

While the device is programming to the specified address, READ operations access the status register bits. Status register bit DQ7 will be 0 during programming and switch to 1 on completion.

If either of the status register bits DQ1, DQ3, or DQ4 is set on completion, the PROGRAM command has failed. Once programmed, the data at address 03E2h cannot be modified until a BLOCK ERASE operation is used to erase the block. Also, it will not be possible to reliably program to any of the addresses between 03E0h and 03E3h until a BLOCK ERASE operation is issued to erase the block.

## Using the Software Driver

### General Considerations

The software driver described in this technical note is intended to simplify the process of developing application code in C for Micron's P8P parallel PCM device. This software driver supports the device driver interface, and as a result, future device changes will not necessarily lead to code changes in application environments.

This technical note gives a summary description of the new device driver interface. A complete description is available from your Micron distributor.

With the software driver interface, users can focus on writing the high-level code required for their particular applications. The high-level code accesses the PCM device by calling the low-level code. This means that users do not have to be concerned with the details of the special command sequences. The resulting source code is both simpler and easier to maintain.

Code developed using the provided drivers can be broken down into three layers:

- Hardware-specific bus operations
- Low-level code
- High-level code written by the user

The low-level code requires hardware-specific READ and WRITE bus operations to communicate with the P8P device. The implementation of these operations is hardware platform dependent as it depends on the microprocessor on which the C code runs and on the location of the memory device in the microprocessor's address space.

The user must write the C code suitable for the current hardware platform.

The low-level code takes care of:

- Issuing the correct WRITE operation sequences for each command.
- Interpreting the information received from the device during programming or erasing.

The high-level code written by the user will access the memory device by calling the low-level code. As a result, the code implemented by the user is simple and easy to maintain. Another consequence is that the user's high-level code is easier to apply to other Micron PCM devices.

When developing an application, the user is advised to proceed as follows:

1. Write a simple program to test the low-level code provided and verify that it operates as expected in the user's target hardware and software environments.
2. Write the high-level code for the desired application. It will access the PCM device by calling the low-level code provided.
3. Thoroughly test the complete application source code.

## Porting the Drivers to the Target System (User Change Area)

All changes to the software driver that the user must consider can be found in the header file. A designated area called the user change area contains the items required to port the software driver to the new hardware.

## Basic Data Types

Check whether the compiler to be used supports the following basic data types as described in the source code, and change it where necessary:

```
typedef    unsigned    char    NMX_uint8;    (8 bits)
typedef                    char    NMX_sint8;    (8 bits)
typedef    unsigned    short   NMX_uint16;   (16 bits)
typedef                    short   NMX_sint16;  (16 bits)
typedef    unsigned    int     NMX_uint32;   (32 bits)
typedef                    int     NMX_uint32;  (32 bits)
```

## Device Type

Choose the correct device by using the appropriate define statement:

```
#define USE_P8P_8 (P8P used in 8 Bit Mode)
#define USE_P8P_16 (P8P used in 16 Bit Mode)
```

## PCM Device Location

BASE\_ADDR is the PCM device start address. It must be set according to the target system to access the PCM device at the correct address. This value is used by the FlashRead() and FlashWrite() functions. The default value is set to zero, and must be adjusted appropriately:

```
#define BASE_ADDR ((volatile uCPUBusType*)0x00000000)
```

## PCM Configuration

Choose the correct PCM device configuration:

- The following define statement supports a board configuration containing a CPU with an external 16-bit memory bus with a single 16-bit PCM device connected to it:

```
#define USE_16BIT_CPU_ACCESSING_1_16BIT_FLASH
```

- The following define statement supports a board configuration containing a CPU with an external 32-bit memory bus with two 16-bit PCM memory devices connected to it:

```
#define USE_32BIT_CPU_ACCESSING_2_16BIT_FLASH
```

- The following define statement supports a board configuration containing a CPU with an external 16-bit memory bus with two 8-bit PCM memory devices connected to it:

```
#define USE_16BIT_CPU_ACCESSING_2_8BIT_FLASH
```

- The following define statement supports a board configuration containing a CPU with an external 32-bit memory bus with two 8-bit PCM memory devices connected to it:

```
#define USE_32BIT_CPU_ACCESSING_4_8BIT_FLASH
```

## Timeout

Timeouts are implemented in the loops of the code to provide an exit for operations that would otherwise never terminate. There are two possibilities.

1. The ANSI library functions declared in `time.h` exist:

If the current compiler supports `time.h`, the define statement `TIME_H_EXISTS` should be activated. This prevents any change in timeout settings due to the performance of the current evaluation hardware.

```
#define TIME_H_EXISTS
```

2. The `COUNT_FOR_A_SECOND` option:

If the current compiler does not support `time.h`, the define statement `TIME_H_EXISTS` cannot be used. In this case, the `COUNT_FOR_A_SECOND` value must be defined so as to create a one-second delay. For example, if 100,000 repetitions of a loop are needed to give a time delay of one second, then `COUNT_FOR_A_SECOND` should have the value 100,000.

```
#define COUNT_FOR_A_SECOND (chosen value)
```

**Note:** This delay depends on the hardware performance and should be updated every time the hardware is changed.

This driver has been tested with a certain configuration, and other target platforms may have other performance data. It may therefore be necessary to change the `COUNT_FOR_A_SECOND` value. It is up to the user to implement the correct value to prevent the code from timing out too early and accommodate correct completion.

## Additional Subroutines

```
#define VERBOSE
```

In this software driver, the `VERBOSE` define statement is used to activate the `FlashErrStr()` function to generate a text string describing the return code from the PCM device.

## Additional Considerations

The access timing data for the PCM device can sometimes be problematic, and it may be necessary to change the `FlashRead()` and `FlashWrite()` functions if they are not compatible with the timings of the target hardware. These problems can be solved with a logic state analyzer.

The programmer must take extra care when the device is accessed during an interrupt service routine. When the device is in read mode, interrupts can freely read from the device. Interrupts that do not access the device may be used during all functions.

## C Library Functions

The software library described in this technical note provides the user with source code for the functions listed in Table 1.

Flash() is used to access all device functions and acts as the main PCM device interface. Unsupported PCM device functionality can be detected and malfunctions can thus be avoided.

**Note:** Note: Other functions are listed to offer a second-level interface when enhanced performance is required. Within the driver, the functions are always used in the same way. This means that the function interface (names, return codes, parameters, data types) remains unchanged, regardless of the PCM device.

**Table 1: C Library Functions Provided**

Function	Description
FlashReset()	Used to reset the device to the read memory array mode. Note: There should be no need to call this function under normal operation, as all other software library functions leave the device in this mode.
FlashReadStatusRegister()	Used to read the status register of the device.
FlashReadDeviceId()	Used to read the device code of the PCM device.
FlashReadManufacturerCode()	Used to read the manufacturer code of the PCM device.
FlashReadCfi()	Used to check whether the PCM device CFI is supported. Then, it reads the CFI value at the specified offset.
FlashWriteStatusRegister()	Used to write the status register.
FlashBlockErase()	Used to erase a block in the device. The blocks cannot be erased when $V_{PEN}$ is LOW, $V_{IL}$ . Attempting to do so generates an error.
FlashBlockProtect()	Used to protect a block in the PCM device. Once protected, the block cannot be programmed or erased until it is unprotected.
FlashCheckCompatibility()	Used to check the PCM device for compatibility.
FlashCheckBlockProtection()	Used to check whether a block is protected.
FlashChipErase()	Used to erase the entire memory device. The memory device cannot be erased when $V_{PEN}$ is LOW, $V_{IL}$ . Attempting to do so generates an error.
FlashProgram()	Used to program arrays of elements to the PCM device. Only previously erased elements can be programmed reliably. Protected blocks cannot be programmed.
FlashProtectionRegisterProgram()	Used to program the user area of the protection register. Programming is not possible after the area has been protected.
FlashResume()	Used to resume a previously suspended operation.
FlashSingleProgram()	Used to program a single element.
FlashSuspend()	Used to suspend the PROGRAM or ERASE operation in progress.
FlashRead()	Used to read a value from the PCM device.
FlashWrite()	Used to write a value to the PCM device.
FlashBitAlterableProgram()	Used to program arrays of elements to the PCM device. Protected blocks cannot be programmed. Note: The state of PCM memory can change from a 0 to 1, or from 1 to 0. There is no need to erase blocks.

The functions provided in the software library rely on the user implementing the hardware-specific bus operations and on access timings to communicate properly with the PCM device. If changes to the software driver are necessary, only the two following functions must be changed:

- FlashRead()
- FlashWrite()

## Getting Started (Example Quicktest)

To test the source code in the target system, start by reading from the P8P device. If it is erased, then only FFFFh data should be read. Then, read the manufacturer and device codes and check that they are correct. If these functions work, the other functions are likely to work, too. However, all functions should be tested thoroughly.

To start, write a function `main()` and include the C file as described in the following example. All PCM functions can be called and executed within the `main()` function.

The following example shows a check of the device identifiers (device code, manufacturer code) and a simple BLOCK ERASE operation.

```
#include P8P.h

void main(void) {
    ParameterType fp; /* Contains all Flash
    Parameters */
    ReturnType rRetVal; /* Return Type Enum */
    Flash(ReadManufacturerCode, &fp);
    printf("Manufacturer Code: %08Xh\r\n",
    fp.ReadManufacturerCode.ucManufacturerCode);
    Flash(ReadDeviceId, &fp);
    printf("Device Code: %08Xh\r\n",
    fp.ReadDeviceId.ucDeviceId);
    fp.BlockErase.ublBlockNr = 10; /* block number 10 will be
    erased */
    rRetVal = Flash(BlockErase, &fp); /* function execution */
} /* EndFunction Main */
```

## Software Limitations

The described software does not implement the full set P8P functionality. When an error occurs, the software simply returns the error message, and it is left to the user to decide what to do. The user can either try the command again or, if necessary, replace the device.

## Conclusion

The P8P 3V supply PCM device is the ideal product for embedded and other computer systems. It can be easily interfaced to microprocessors, and is driven with simple software drivers written in the C language.

The software device driver interface supports changeable memory device configurations, compiler-independent data types, and a unique access mode for a broad range of memory devices.

Moreover, applications supporting the software device standard can use any memory device with the same interface without any code change. Recompiling with a new software driver is all that is needed to control a new device.

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900  
[www.micron.com/productsupport](http://www.micron.com/productsupport) Customer Comment Line: 800-932-4992

Micron and the Micron logo are trademarks of Micron Technology, Inc. All other trademarks are the property of their respective owners.