

# Technical Note

## Extending PCM Temperature and Data Retention Ranges: Software Refresh Procedure for Micron® P5Q Serial PCM and P8P Parallel PCM Devices

### Introduction

This technical note describes a software procedure for refreshing data in Micron® P5Q serial phase change memory (PCM) and P8P parallel PCM devices. It also summarizes how data is stored in PCM, temperature considerations, and our software REFRESH solution. The information provided is intended for users with a working knowledge of the principles of programming embedded systems. This solution enables customers to use Micron PCM devices beyond current data sheet specifications for temperature and data retention to support the specific requirements of embedded applications.

### Overview of PCM

PCM is a type of memory device that stores information through a reversible structural phase change in a chalcogenide material. The material can change its own properties, both electrical and optical, when changing from the amorphous (disordered) to the polycrystalline (regularly ordered) state. The states are stable when electrical power is absent, which makes PCM devices nonvolatile.

The PCM storage element consists of a thin film of chalcogenide contacted by a resistive heating element. In PCM, the phase change is induced in the memory cell by highly localized joule heating caused by an induced current at the material junction. During this process, a small volume of the chalcogenide material changes phases. Changing phases is a reversible process that is modulated by the magnitude of injected current, the applied voltage, and the duration of the heating pulse.

### Attributes

PCM combines the benefits of traditional floating gate Flash (NOR and NAND) with some of the key attributes of RAM and EEPROM.

PCM offers fast random access times like NOR Flash and RAM technology and also has the ability to write moderately fast like NAND Flash. In addition, PCM is 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. And different from RAM, PCM technology is nonvolatile. However, data retention compares with NOR Flash devices. Currently, PCM technology has WRITE cycle endurance that is better than that of NAND or NOR Flash, but less than that of RAM.

P5Q serial PCM and P8P parallel PCM devices provide a proper emulation of legacy NOR Flash and also provide a set of features that designers can use to exploit the specific capabilities of the PCM technology. This is intended to encourage the spread of P5Q serial PCM and P8P parallel PCM devices into existing hardware and software development platforms.

## PCM Temperature Considerations and a Solution

Because state transitions are mainly generated by temperature changes, huge temperature increases to the data stored in the memory array could cause a change to the disordered state, causing the data to be unreadable. For technical details about the relationship between data retention and temperature, refer to TN-310054: PCM Reliability Considerations. Contact your Micron representative for assistance.

P8P parallel PCM and P5Q serial PCM device reliability across nominal temperature ranges is superior to that of floating gate Flash, even though the devices may be subject to thermally activated disturbs at higher temperatures. In this section, we outline a software procedure for refreshing data in PCM devices.

The frequency of the refresh algorithms will be driven by temperature because the higher the temperature the faster the data corruption. When the data pattern is valid, refreshing the data on itself restores the threshold of the single bit on the readable level, increasing the timeline during which the data is correctly readable. Thus, all cells in the memory array must be periodically refreshed in a timely fashion, especially when high temperatures are detected.

Refresh is accomplished by reading and rewriting each word on itself in the array. Writing one word at a time would be inefficient on such memory devices. To enhance performance, the refresh procedure refreshes at least one buffer at a time or a multiple of buffers at a time. The most suitable size of the buffer used by the procedure depends on the device type and geometry, and it is usually stated as a configuration requirement. One or multiple buffers are refreshed at the time of the PCM array because this is the minimum size of a REFRESH cycle. The actual number of operations required to refresh the entire array depends on the buffer width (as a minimum) and the configuration of the algorithm.

### Configuring Refresh Algorithms

There are different ways to configure a refresh algorithm. One solution is to address when the REFRESH operation is performed in a single burst, which means that the entire array is refreshed without interruption. When a refresh procedure is ongoing, the user cannot access the PCM memory array. If overheating persists, choosing to refresh at the wrong time could lead to a deadlock condition.

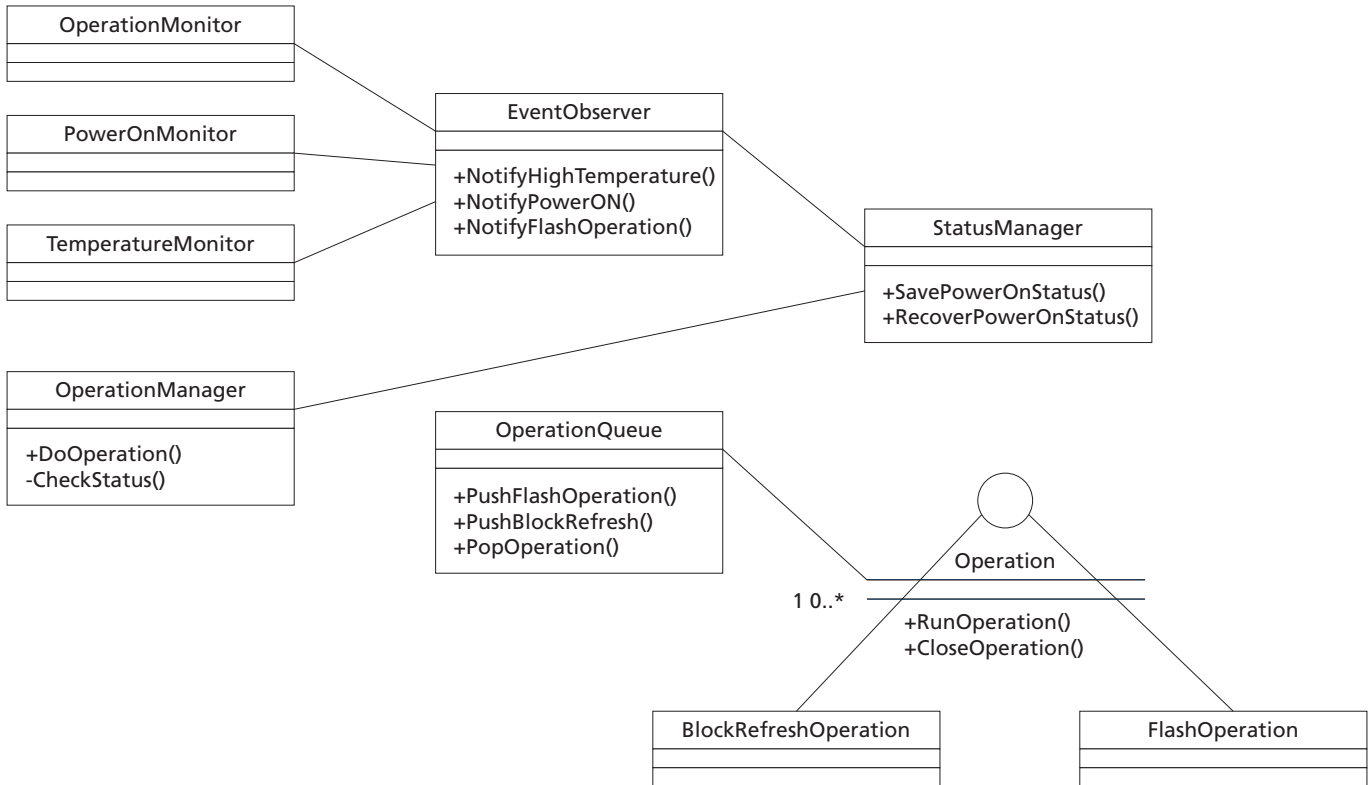
To avoid this condition, REFRESH could be performed in the background. Specifically, one task can evaluate the current state of the PCM and run the user operation or REFRESH. Determining which operation can be performed is based on the priority of the operation. For example, a mechanism can run previously queued jobs. If the user needs frequent access to the PCM device, the refresh time must be chosen as a compromise between the time available for access to the memory and the time required to refresh the device. In this case, REFRESH must be performed by cycle.

Switching between the operations requested by the user and REFRESH could be computationally intensive, so the size of one cycle must be chosen as a compromise between the efficiency of the REFRESH algorithm and the responsiveness perceived by the user. Each cycle is made up of a set of one or more cycles of one or multiple buffer refreshes. For example, there could be one cycle of 10ms for refreshing every 30ms.

## Event Observer Function

The event observer function is in charge of observing when the temperature raises. A mechanism for evaluating the temperature should be added because the device cannot recognize whether the temperature is high.

**Figure 1: Event Observer Function**



## PCM Refresh Procedure

If the PCM device does not have data written to it, no REFRESH is needed. Because the PCM's natural state is crystalline (that is, the 1s of the memory array), the increased temperature does not affect the crystalline state, which means that a word would not need a REFRESH. Only cells with 0s are the cells at risk in this case, and only those cells need to be refreshed.

To optimize the timing access to the device, the minimum granularity of the access size to be refreshed should be set accordingly. For example, this could be the buffer size or a multiple of it. If all bits in the buffer (the size of the buffer is chosen according to the granularity requested by the algorithm) are 1s, the overwrite procedure can be skipped. One of the major issues of a rise in temperature is the need to refresh each memory word continuously when the temperature increases. The procedure should be optimized to ensure refresh completion within the specified time.

REFRESH requires global data to store the refresh state in the event of a power loss, which leads to a loss of available space for the user. That data is typically stored in locations within the PCM array, and the user must indicate an address to allow the procedure to store the data; otherwise, REFRESH cannot recognize whether the procedure is complete.

The user must specify the position in the memory array at which the procedure can store data. The procedure stores the progress of the procedure at each cycle, and, in particular, it stores the address of the array that the procedure is refreshing. To prevent cells from prematurely wearing out, the procedure should not use the same location to store the data.

The block that contains the data stored for REFRESH is used as a circular queue. In the first word is the head of the queue, which codes the displacement within the queue. Each location of the queue contains the address of the memory array that the procedure is refreshing. The first word stores information about the displacement coded in base on the number of 1s of the word itself. The head of the queue changes one bit at a time, exploiting the overwrite feature of PCM.

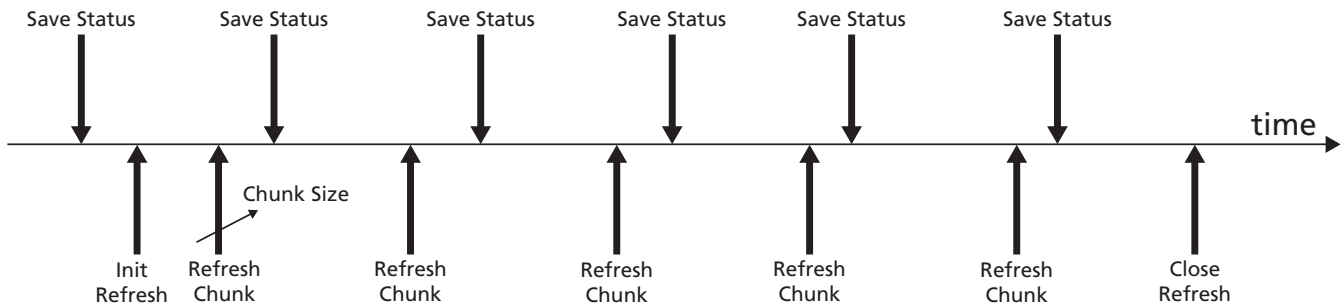
## Refreshing Data in PCM Devices

To understand how the refresh procedure works, it is important to consider what data is stored. The first data that can be stored is user defined and is consistent because it is built into the code itself. Based on this data, the procedure can be initialized.

To initialize the data at startup, the procedure reads the user-defined data from PCM and then reads the data that was most recently written. There is no need to further check for consistency. The time it takes to init the procedure is the time it takes to read less than one block.

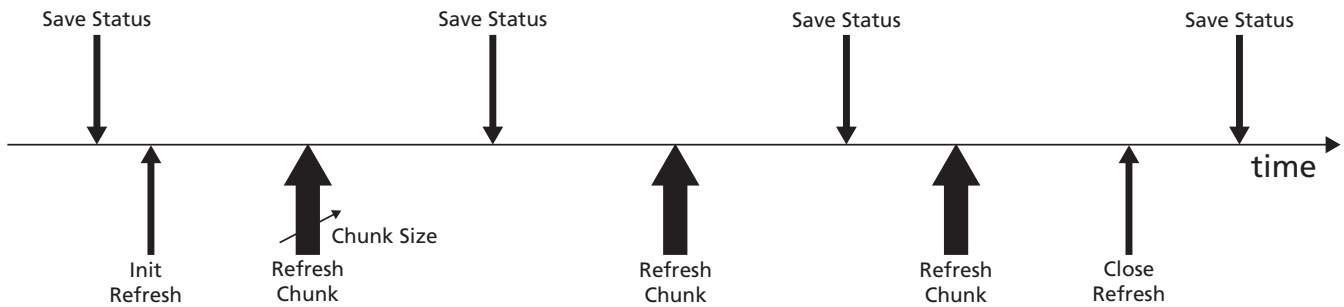
The procedure stores the progress of the refresh at each cycle, as shown in Figure 2.

**Figure 2: Refresh Progress at Each Cycle**



The save status frequency can also be reduced when executing a long operation that cannot be stopped. The intermediate states are not useful in this particular situation. As a result, the granularity of the WRITE operation is tied directly to the size of the chunk being refreshed by the procedure, as shown in Figure 3.

**Figure 3: WRITE Operation Granularity**



The refresh utility is a sequence of the following operations:

1. An initialization procedure sets the data.
2. A series of WRITE operations are executed to rewrite the data on itself.
3. A close procedure finalizes the refresh procedure.

If the minimum granularity of the procedure is large enough, you will have saved logs that include the refresh not initialized, the refresh existing at each fixed size, and the complete refresh. The size of the chunk can be configured, which is a method for ensuring efficiency and consistency.

Each saved status is a valid log of the refresh procedure, and each intermediate status cannot be recovered. For example, if you have refreshed 20Mb and the chunk size is 15Mb, all data up to 15Mb is correctly refreshed and this status will be available after a potential power loss. As a result, if there is a power failure, the refresh is restored to the most recent saved status. For example, if you want to refresh the entire PCM device during the night, you could set the chunk up to the PCM size, disable the intermediate saving status, and if a power failure occurs, the refresh either will be complete or will not exist. That is, the procedure cannot recognize which part was refreshed and which was not refreshed.

To address the problems associated with an unexpected power loss, REFRESH never erases live data. The new content of the procedure becomes live only when all updates are safely written to the PCM device. REFRESH assures the consistency of the status at each stage:

- During the refresh of each chunk, the procedure is in a stable state. The procedure rewrites one bit on itself, and the user must have no active operations ongoing in this phase and all other activity is suspended.
- When writing the status data in the block defined by the user (as previously stated), the data is managed as a queue. In this state, the order is important because it ensures consistency. As a result, it is optimized:
  - The initial address of the chunk that was just refreshed is written in the queue.
  - The head of the queue modifying one bit at a time is updated. The head represents the displacement to the most recent item recorded into the queue. In this way, we can ensure the consistency of the data at all times.

When REFRESH is complete, the procedure restores the initial condition of the algorithm. To set the initial conditions, the procedure performs a block ERASE and checks whether ERASE was performed successfully. To check the block for consistency, the procedure fills the unused part of the block to store a known pattern. To initialize the procedure at startup, it simply reads the pattern from the PCM device, validates its consistency by reading the pattern, and chooses whether it contains valid data or whether the initialization must be reapplied.

## Conclusion

Because PCM is a memory that stores information through a reversible structural phase change in a chalcogenide material, PCM is sensitive to large temperature increases, and at high temperatures, the data stored in the memory array could change, causing the data to be invalid.

Refreshing the data on itself restores the threshold of the single bit on the readable level, increasing the timeline during which the data is correctly readable. To avoid unnecessary impact on device performance, evaluate the minimum refresh frequency needed to prevent data loss. If REFRESH is not performed on time, the REFRESH itself could be ineffective at preventing data loss. While REFRESH is ongoing, the user cannot access the PCM memory array.

The REFRESH process will ensure the validity of the data for the lifespan of the device. The impact to device operation during this refresh procedure is minimal and is only needed in case of elevated temperatures. For more information, refer to TN-310054: PCM Reliability Considerations.

8000 S. Federal Way, P.O. Box 6, Boise, ID 83707-0006, Tel: 208-368-3900  
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.

## Reference Code

```
#include <stdio.h>
#include "P8P.h"

/*****
***** user defined area *****/
*****/
#define BASE_ADDR ((volatile uCPUBusType*)0x00000000)
#define USER_FIXEDBLOCKADDR 0x0 // must be chosen at the beginning of a block
#define USER_ENDFIXEDBLOCKADDR (USER_FIXEDBLOCKADDR+BLOCK_SIZE)
#define ByteAddress 4 // number of bytes of the address
#define udWriteBufferSize 32

#define CurrentQueueAddress(x, y) ( (x *16 + y) * sizeof(uCPUBusType) )

const int num_of_chunk = FLASH_SIZE/BLOCK_SIZE/2;
const int chunk_size = FLASH_SIZE/num_of_chunk;

/***** End user area *****/

typedef unsigned char  NMX_uint8; /* All HW dependent Basic Data Types */
typedef          char  NMX_sint8;
typedef unsigned short NMX_uint16;
typedef          short NMX_sint16;
typedef unsigned int   NMX_uint32;
typedef          int   NMX_sint32;

typedef enum {
    Test_Success,
    Test_Failure,
    Test_BlockProtected,
    Test_DoubleProgramAttempt,
    Test_FunctionNotSupported
} Test_ReturnType;
```

```
typedef NMX_uint16 uCPUBusType;

// the user defines an area that can be used by the procedure to store internal
// data
// to avoid overwriting the same word location in the flash, the area is used
// as circular buffer
// the area is made up of:
// - the first word stores the displacement into the area in which there are
// the valid data
// - the next area contains the actual address of the flash the procedure is
// refreshing
// this area is used as a circular buffer

struct powerOnState {
    NMX_uint32 udByteDisplacement; //
    NMX_uint16 ubBitDisplacement;
    NMX_uint32 udArrayAddress; // the address to be refreshed
    NMX_uint32 udEndAddress;
    NMX_uint32 udBaseAddr; // base address of the flash
    NMX_uint32 udQueueDisplacement; // the displacement into the queue
    NMX_uint16 uwNumber;
} RState, *RSp;

struct ConfigState {
    NMX_uint32 udFixedAddress;
    NMX_uint32 udQueueStartDisplacement;
    NMX_uint32 udQueueEndDisplacement;
    NMX_uint32 udMagicNrStartDisplacement;
    NMX_uint32 udMagicNrEndDisplacement;
};

struct ConfigState CfgState =
{
//udFixedAddress: the user must define the block address
    (USER_FIXEDBLOCKADDR),
//udQueueStartDisplacement: start address of the queue
    (QUEUEDISPLACEMENT + USER_FIXEDBLOCKADDR),
//udQueueEndDisplacement: end address of the queue
    (QUEUEDISPLACEMENT + USER_FIXEDBLOCKADDR),
```

```
//udMagicNrStartDisplacement: check the consistency
    (QUEUEDISPLACEMENT + USER_FIXEDBLOCKADDR + 1),
//udMagicNrEndDisplacement: the address into the queue
    (USER_ENDFIXEDBLOCKADDR),
};

voidDebug_PrintPowerOnStatus(void);

ReturnType Init_PowerOnStatus (void);
ReturnType Check_PowerOn (void);
ReturnType Close_PowerOnStatus (void);
ReturnType CheckOn_Completed (void);
ReturnType Save_PowerOnStatus (void);
ReturnType Refresh_Chunk (void);
ReturnType Calculate_displacement(void);
ReturnType Write_PowerOnStatus(void);
ReturnType Write_Displacement(void);
ReturnType check_magicNumber(void);
ReturnType FlashOverwriteBuffer( NMX_uint32 udAddrOff, NMX_uint32 udNrOfElementsInArray, void *pArray );
ReturnType Micron PCM_refresh_memory(void);

voidDebug_PrintPowerOnStatus(void)
{
    NMX_uint32 udAddrOffset;

    printf("udByteDisplacement:%x - ", RState.udByteDisplacement);
    printf("ubBitDisplacement:%x - ", RState.ubBitDisplacement);
    printf("udArrayAddress:%x - ", RState.udArrayAddress);
    printf("udQueueDisplacement:%x\n", RState.udQueueDisplacement);

    udAddrOffset = CurrentQueueAddress(RState.udByteDisplacement, RState.ubBitDisplacement) + CfgState.udQueueStartDisplacement;
    printf("address to be refreshed:0x%x @addr:%x\n", (FlashRead(udAddrOffset+1)<<16 | FlashRead(udAddrOffset)), CurrentQueueAddress(RState.udByteDisplacement, RState.ubBitDisplacement) );
    printf("\n\n");
}
```

```
ReturnType Micron PCM_refresh_memory(void)
{
    ReturnType rRetVal = Flash_Success; /* Return Value: Initially optimistic */
    NMX_sint32 udRemains;
    udRemains = 0;

    /* *****
    ****          ALL BLOCKS MUST BE UNPROTECT BEFORE STARTING          ****
    ***** */

    Check_PowerOn(); // recover the status: this check if a powerloss occurred, set
                    the refresh-defined variables

    Init_PowerOnStatus(); // recover the status: this set the user-defined vari-
                        ables

    // start refresh
    for (udRemains = 0; udRemains < num_of_chunk; udRemains++)
    {
        if (Flash_Success != Refresh_Chunk())
            printf("Errore in Refresh_Chunk\n");

        Save_PowerOnStatus();

        Debug_PrintPowerOnStatus();
    }

    Close_PowerOnStatus();

    return Flash_Success; // TBD
}

ReturnType CheckOn_Completed(void)
{
    printf("CheckOn_Completed\n");
    return Flash_Success; // TBD
}

ReturnType check_magicNumber(void)
```

```

{
#define FIXEDPATTERN 0xAAAA
NMX_uint32 udCount = (CfgState.udMagicNrEndDisplacement - CfgState.udMagic-
                    NrStartDisplacement) / sizeof(uCPUBusType);
NMX_uint32 udSourceAddr = CfgState.udMagicNrStartDisplacement;

if ( (uCPUBusType)0xFFFFFFFF != FlashRead(CfgState.udMagicNrStartDisplacement)
    )
    return Flash_Success; // There is a Power Loss

for (udCount=0; udCount<udWriteBufferSize; udCount++)
    if ((uCPUBusType) FIXEDPATTERN != FlashRead(udSourceAddr+udCount) )
        return Flash_ProgramFailed;

return Flash_Success; // so far so good
}

```

// recover the status: this check if a powerloss occurred during an erase, set  
the refresh-defined variables

ReturnType Check\_PowerOn(void)

```

{
NMX_uint32 udNrOfElementsInArray;
NMX_uint32 udStartAddr = 0x0;//RState.udByteDisplacement;
NMX_uint32 udSourceAddr = 0x0;
NMX_uint16 ublBlockNr = 0;
NMX_uint32 udAddrOffset = 0;

if (check_magicNumber() == Flash_Success)
{
    if ( (uCPUBusType)(0xFFFFFFFF) == FlashRead(udStartAddr) )
    {
        udNrOfElementsInArray = ByteAddress/sizeof(uCPUBusType);
        udAddrOffset = CurrentQueueAddress(RState.udByteDisplacement,
            RState.ubBitDisplacement ) + CfgState.udQueueStartDisplacement;
        FlashOverwriteBuffer( udAddrOffset, udNrOfElementsInArray,
            &udSourceAddr ) ; // Init the first address to be refreshed
    }
}
}

```

```
    } else
    {
        FlashBlockErase(CfgState.udFixedAddress);
    }

    return Flash_Success;
}

ReturnType Refresh_Chunk (void)
{

    ReturnType rRetVal = Flash_Success; /* Return Value: Initially optimistic */
    uCPUBusType *ucpArrayPointer; /* Use an uCPUBusType to access the array */
    uCPUBusType ucArray[udWriteBufferSize];
    NMX_uint32 udCount, curLength, remains;
    NMX_uint32 udNrOfElementsInArray;
    NMX_uint32 udSourceAddr = RState.udArrayAddress;

    remains = chunk_size;
    do {

        /* Program Buffer size: 32Words */
        curLength = udWriteBufferSize-(udSourceAddr&(udWriteBufferSize-1));
        if (curLength > remains)
            curLength = remains;

        udNrOfElementsInArray = curLength;
        ucpArrayPointer = ucArray; // read one buffer
        for (udCount=0; udCount<udWriteBufferSize; udCount++)
            ucpArrayPointer[udCount] = FlashRead(udSourceAddr+udCount);
        // overwrite the buffer
        rRetVal = FlashOverwriteBuffer( udSourceAddr, udNrOfElementsInArray, ucAr-
            ray );

        udSourceAddr += curLength;
        remains -= curLength;
    } while (remains);/* End while */
```

```
    return rRetVal;
}

ReturnType Save_PowerOnStatus(void)
{
    ReturnType rRetVal = Flash_Success;
    if (Flash_Success != (rRetVal = Write_PowerOnStatus()) )
        return rRetVal; // write the address just refreshed in the queue

    rRetVal = Write_Displacement(); // this finalizes the transition; lower one bit
    return rRetVal;
}

// this function is used to finalize the refresh procedure
//
ReturnType Close_PowerOnStatus(void)
{
    ReturnType rRetVal = Flash_Success; /* Return Value: Initially optimistic */
    uCPUBusType *ucpArrayPointer; /* Use an uCPUBusType to access the array */
    uCPUBusType ucArray[udWriteBufferSize];
    NMX_uint32 udCount;
    NMX_uint32 udNrOfElementsInArray;
    NMX_uint32 udSourceAddr = 0x1000; //RState.udAddress;
    NMX_uint16 ublBlockNr = 0;

    udNrOfElementsInArray = 32; // TODO

    ucpArrayPointer = ucArray; // read one buffer
    for (udCount=0; udCount<udWriteBufferSize; udCount++)
        ucpArrayPointer[udCount] = (uCPUBusType) FIXEDPATTERN;

    // overwrite the buffer
    if (Flash_Success != (rRetVal = FlashOverwriteBuffer( udSourceAddr, udNrOfEle-
        mentsInArray, ucArray ) ) )
        return rRetVal;

#ifdef NOTDEBUG
```

```
    rRetVal = FlashBlockErase( ublBlockNr);
#endif

    return rRetVal;
}

ReturnType Calculate_displacement(void)
{
    //
    RState.udQueueDisplacement = CurrentQueueAddress(RState.udByteDisplacement,
        RState.ubBitDisplacement );

    RState.ubBitDisplacement++;
    if ( 0 == RState.ubBitDisplacement%(8*sizeof(uCPUBusType)))
    {
        RState.ubBitDisplacement = 0;
        RState.udByteDisplacement++;
    }

    return Flash_Success;
}

ReturnType Write_PowerOnStatus(void)
{

    ReturnType rRetVal = Flash_Success; /* Return Value: Initially optimistic */
    uCPUBusType *ucpArrayPointer; /* Use an uCPUBusType to access the array */
    uCPUBusType ucArray[udWriteBufferSize];
    NMX_uint32 udNrOfElementsInArray;
    NMX_uint32 udSourceAddr = RState.udArrayAddress;
    NMX_uint16 ublBlockNr = 0;
    NMX_uint32 udAddrOffset;

    udNrOfElementsInArray = ByteAddress/sizeof(uCPUBusType); // address Size / bus
        width
```

```
udAddrOffset = CurrentQueueAddress(RState.udByteDisplacement, RState.ubBitDis-
    placement ) + CfgState.udQueueStartDisplacement;

rRetVal = FlashOverwriteBuffer( udAddrOffset, udNrOfElementsInArray,
    &udSourceAddr );

    // next address to be refreshed
RState.udArrayAddress += chunk_size;

return rRetVal; // TBD

}

ReturnType Write_Displacement(void) // this finalizes the transition
{

    NMX_uint16 number = RState.uwNumber;
    ReturnType rRetVal = Flash_Success;

    number &= ~(1 << RState.ubBitDisplacement);
    rRetVal = FlashOverwriteBuffer( RState.udQueueDisplacement, 1, &number
        );// 1 word
    if (0==number)
    {
        RState.ubBitDisplacement = 0;
        RState.udQueueDisplacement++;
        RState.uwNumber = FlashRead(RState.udQueueDisplacement);
        RState.udByteDisplacement++;
    } else
    {
        RState.ubBitDisplacement++;
        RState.uwNumber = number;
    }

    return rRetVal;

}
```

```
ReturnType Init_PowerOnStatus (void)
{
    int i,j;

    uCPUBusType number;
    uCPUBusType wordDisplacement = 0;
    NMX_uint32 udAddrOffset;

    RState.udQueueDisplacement = 0x0;
    RState.ubBitDisplacement = 0;
    RState.udArrayAddress = 0x0;
    RState.udEndAddress = 0x0;
    RState.udBaseAddr = 0x0;

    RState.udQueueDisplacement = 0;
    RState.ubBitDisplacement = 0;

    for (j=0;j<num_of_chunk;j++)
    {
        number = FlashRead(RState.udQueueDisplacement);

        for (i=0;i<16;i++)
        {
            if ( number & (1 << i))
            {
                RState.ubBitDisplacement = i;
                RState.uwNumber = number;
                udAddrOffset = CurrentQueueAddress(RState.udByteDisplacement,
                    RState.ubBitDisplacement ) + CfgState.udQueueStartDisplacement;
                RState.udArrayAddress = FlashRead(udAddrOffset-1)<<16 |
                    FlashRead(udAddrOffset-2);
                printf("Init_PowerOnStatus -> recovered %x-%x \n", (FlashRead(udAddrOffset-
                    1)<<16), (FlashRead(udAddrOffset-2)));
                return Flash_Success;
            }
        }
        RState.udQueueDisplacement++;
    }
}
```

```
        RState.udByteDisplacement++;
    }

    return Flash_ProgramFailed;
}
```

/\*\*\*\*\*

```
Function:      ReturnType FlashOverwriteBuffer( NMX_uint32 udAddrOff,
        NMX_uint32 udNrOfElementsInArray, void *pArray )
```

**Arguments:**

udAddrOff is the address offset into the flash to be programmed  
udNrOfElementsInArray holds the number of elements (uCPUBusType) in the array.  
pArray is a void pointer to the array with the contents to be programmed.

**Return Values:**The function returns the following conditions:

- Flash\_Success
- Flash\_OperationTimeOut
- Flash\_AddressInvalid
- Flash\_BlockProtected
- Flash\_ProgramFailed
- Flash\_VppInvalid
- Flash\_FunctionNotSupported

**Description:** This function is used to program an array into the flash. The procedure overwrite

the existing data, no need to erase the block first.  
Any errors are returned without any further attempts to program other addresses  
of the device. The function returns Flash\_Success when all addresses have successfully been programmed.

**Note:** Two program modes are available:

The number of elements (udNumberOfElementsInArray) contained in pArray are programmed directly to the flash starting with udAddrOff.

**Pseudo Code:**

- Step 1: Check if programming mode is valid
- Step 2: Check whether the data to be programmed are within the  
Flash memory

- Step 3: While there is more to be programmed
- Step 4: Determine limits of current buffer (16 elements, or 4 pages)
- Step 5: Program within the next buffer
- Step 6: Decision between direct and single value programming
- Step 7: Wait until the Program/Erase Controller is ready
- Step 8: Check for any errors
- Step 9: Clear Status Register and return to Read Array mode
- Step 10: Return the error condition

```

*****/
ReturnType FlashOverwriteBuffer( NMX_uint32 udAddrOff, NMX_uint32 udNrOfElementsInArray, void *pArray ) {
    ReturnType rRetVal = Flash_Success; /* Return Value: initially optimistic */
    uCPUBusType *ucpArrayPointer; /* Use an uCPUBusType to access the array */
    uCPUBusType ucStatus; /* Holds the Status Register reads */
    NMX_uint32 remains; /* remain numbers to be written in the programming array */
    NMX_uint32 curLength; /* current length need to write to the buffer */
    NMX_uint32 udCount, udArrayCount = 0; /* current length need to write to the buffer */
    NMX_uint32 prevMaskValue;
    NMX_uint32 udInitialAddrOff = udAddrOff;

    ucpArrayPointer = (uCPUBusType *)pArray;
    remains = udNrOfElementsInArray;

do {

    /* Program Buffer size: 32Words */
    curLength = udWriteBufferSize-(udInitialAddrOff&(udWriteBufferSize-1));
    if (curLength > remains)
        curLength = remains;

    /* Step 1: Call FlashWriteProgramBuffer */
    /* Step 2: Program within the next buffer */
    FlashWrite( udInitialAddrOff, CMD(0x0050) ); /* Clear Status Register */
    /* NOTE ! CSR also clears bit 1 BPS as well as bits 3, 4 and 5 */

```

```
FlashWrite( udInitialAddrOff, (uCPUBusType)CMD(0x00EA) ); /* Program
Setup */

FlashTimeOut(0); /* Initialize TimeOut Counter */
do {
    ucStatus = FlashRead(udInitialAddrOff);
    if (FlashTimeOut(5) == Flash_OperationTimeOut) {
        FlashReset(udInitialAddrOff);
        return Flash_OperationTimeOut;
    } /* Endif */

} while( (ucStatus & CMD(0x0080)) != CMD(0x0080) );
/* Wait until every Action is finished (StatusRegister Bit7 = 1) */

FlashWrite( udInitialAddrOff, (uCPUBusType)CMD((curLength - 1)) ); /* Num-
ber of elements */

/* Step 3: Write Content to buffer */ /* Start the main program cycle */
for (udCount = 0; udCount < curLength; udCount++, udArrayCount)
    FlashWrite( (udInitialAddrOff + udCount), ((uCPUBusType *)ucpArray-
Pointer++)[udArrayCount] );

FlashWrite( udInitialAddrOff, (uCPUBusType)CMD(0x00D0) ); /* Confirm pro-
gram */

/* Step 4: Wait until Program/Erase Controller is ready (StatusRegister
Bit7 = 1) */
FlashTimeOut(0); /* Initialize TimeOut Counter */
do {
    ucStatus = FlashRead(udInitialAddrOff);
    if (FlashTimeOut(5) == Flash_OperationTimeOut) {
        FlashReset(udInitialAddrOff);
        return Flash_OperationTimeOut;
    } /* Endif */

} while( (ucStatus & CMD(0x0080)) != CMD(0x0080) );

udInitialAddrOff += curLength;
remains -= curLength;
```

```
} while (remains);/* End while */

/* Step 5: Clear Status Register and return to Read Array mode */
FlashWrite( udInitialAddrOff, CMD(0x0050) ); /* Clear Status Register */
/* NOTE ! CSR also clears bit 1 BPS as well as bits 3, 4 and 5 */
FlashReset(udInitialAddrOff); /* Read Array Command */

/* Step 10: Return the error condition */
return rRetVal;

} /* EndFunction FlashProgram */
```