SHARP

APPLICATION NOTE

SHARP
2nd Generation Flash Memory Software Drivers

LH28F016SA
LH28F016SU
LH28F800SU

July 1995

SHARP

SHARP
2ND GENERATION FLASH MEMORY
SOFTWARE DRIVERS

CONTENTS

SHARP

INTRODUCTION

ABOUT THE CODE

This application note provides example software code for word writing, block erasing, and otherwise controlling Sharp's LH28F016SA, LH28F016SU and LH28F800SU (hereafter referred to as LH28F016SA) 16M/8M-bit symmetrically blocked memory components. Two programming languages are provided: high-level "C" for broad platform support, and more optimized ASM86 assembly. In many cases, the driver routines can be inserted "as is" into the main body of code being developed by the system software engineer. Extensive comments are included in each routine to facilitate adapting the code to specific applications.

The internal automation of the LH28F016SA makes software timing loops unnecessary and results in platform-independent code. The following example code is designed to be executed in any type of memory and with all processor clock rates. C code can be used with many microprocessors and microcontrollers, while ASM86 assembly code provides a solution optimized for Intel microprocessors and embedded processors.

The LH28F016SA, like the LH28F008SA, is divided into 64K-byte blocks. Since the GSR and BSR are defined relative to the nearest preceding block beginning address, I often refer to this "block base" address in the comments.

Assumptions:

- Pointers (in C) or EDI offsets (in ASM86) are four (4) bytes long, providing a flat addressing space over the entire LH28F016SA device. This implies the use of 386 or higher machines. If the code is to be run on a machine with a smaller address space, the code must be modified to include some sort of "windowing" scheme which maps segments of flash into system memory. The Intel 82365 is commonly used for this purpose.
- "Ints" are 16-bit and "longs" 32-bit in C.
- It is assumed that these pointers return a value equal to what they are pointing to. In other words, even though the pointer may be four (4) bytes long, this does not imply that incrementing the pointer by one will move the pointer four (4) bytes in memory. It is entirely dependent upon what the pointer is pointing to that determines how the increment will be effected. In the case of four (4) byte pointers and 16-bit ints, incrementing the pointer by one will effectively move the pointer two (2) bytes.
- There exists a function "set_pin" which can set an individual LH28F016SA pin, given the pin number.
- There exists a function "get_pin" which can return the value of an individual LH28F016SA pin, given the pin number.
- The C code can access a function which derives the corresponding block base address from any given address.
- BYTE# pin on the device determines whether addressing refers to words or bytes. I assume word writes/reads to a single device. With minor modifications this code can be adapted for a pair of LH28F016SAs in Byte mode.
- LH28F016SA commands can be written to any address in the block or device to be affected by that command.

Both the C and ASM86 code in this document contain the following routines, in this order:

CSR_word_byte_writes (compatible with LH28F008SA)
CSR_block_erase (compatible with LH28F008SA)
CSR_erase_suspend_to_read (compatible with LH28F008SA)
lock_block
lock_status_upload_to_BSR
update_data_in_a_locked_block
add_data_in_a_locked_block
ESR_word_write
two_byte_write
ESR_page_buffer_write
ESR_block_erase
ESR_erase_all_unlocked_blocks
ESR_suspend_to_read_array
ESR_automatic_erase_suspend_to_write
ESR_full_status_check_for_data_write
ESR_full_status_check_for_erase
single_load_to_pagebuffer
sequential_load_to_pagebuffer
upload_device_information
RYBY_reconfiguration
page_buffer_swap

The names of these routines have been changed to more closely match the algorithms presented in the LH28F016SA User's Manual. Please see Appendix A for a table documenting these changes.

SHARP

ABOUT THE LH28F016SA

Companion product datasheets for the LH28F016SA should be reviewed in conjunction with this application note for a complete understanding of the device.

The example code makes extensive use of bit-masking when interpreting the status registers. As a quick review, note that any bit in a register with the appropriate power of two. Since all of the bits other than the one being tested are masked out, testing the resulting byte for truth is the same as testing the desired bit for truth. For example, if a register contains 01001010, the test for bit 3 would be ANDing the register with 00001000, or hex 8, and testing the result for truth:

Binary	Hex	
01001010	A4	Register
& 00001000	& 08	Mask for bit 3
= 00001000	= 08	Result

In this case the result byte is true, indicating that bit 3 in the register was a 1.

The meanings of the individual bits of these registers is presented here for reference. Note that these are two status register spaces, both of which are distinct from the flash memory array address space. In the CSR space, the CSR is mapped to every address. In the ESR space, the GSR is mapped two words above the base of each 64K-byte block, i.e. to address 2, 8002H, 10002H, etc. (in word mode), while each BSR is similarly mapped one word above the base of each 64K-byte block to locations 1, 8001H, 10001H, etc. (in word mode), each BSR reflecting the status of its own block.

CSR.7	Write State Machine Status	1 = ready 0 = busy
CSR.6	Erase-suspend Status	1 = erase suspended 0 = erase in progress/ completed
CSR.5	Erase Status	1 = error in block erase 0 = successful block erase
CSR.4	Data-write Status	1 = error in data write 0 = successful data write
CSR.3	Vpp Status	1 = Vpp low detect/ operation aborted 0 = Vpp OK when operating
CSR.2	Reserved for future use	
CSR.1	Reserved for future use	
CSR.0	Reserved for future use	

GSR.7	Write State Machine Status	1 = ready 0 = busy
GSR.6	Operation-suspend Status	1 = operation suspended 0 = operation in progress/ completed
GSR.5	Device Operation Status	1 = operation unsuccessful 0 = operation successful or running
GSR.4	Device Sleep Status	1 = device in sleep 0 = device not in sleep
GSR.3	Queue Status	1 = queue full 0 = queue available
GSR.2	Page Buffer Availability	1 = one/two page buffers available 0 = no page buffers available
GSR.1	Page Buffer Status	1 = selected page buffer ready 0 = selected page buffer busy
GSR.0	Page Buffer Select Status	1 = page buffer 1 selected 0 = page buffer 0 selected

BSR.7	Block Status	1 = ready 0 = busy
BSR.6	Block-lock Status	1 = block unlocked for write/erase 0 = block locked to write/erase
BSR.5	Block Operation Status	1 = error in block operation 0 = successful block operation
BSR.4	Block Operation Abort Status	1 = block operation aborted 0 = block operation not aborted
BSR.3	Queue Status	1 = device queue full 0 = device queue available
BSR.2	Vpp Status	1 = Vpp low detected 0 = Vpp OK when operation occurred
BSR.1	Reserved for future use	
BSR.0	Reserved for future use	

LH28F016SA Commands

The LH28F016SA command set is a superset of the LH28F008SA command set, giving existing LH28F008SA code the ability to run on the LH28F016SA with minimal modifications.

LH28F008SA-Compatible Commands

00	invalid/reserved
20	single block erase
40	word/byte write
50	clear status registers
70	read CSR
90	read ID codes
B0	erase suspend
D0	confirm/resume
FF	read flash array

LH28F016SA Performance-Enhancement Commands

0C	page buffer writer to flash
71	read GSR and BSRs (i.e. the ESR)
72	page buffer swap
74	single load to page buffer
75	read page buffer
77	lock bock
80	abort
96,01	RY/BY# enable to level-mode
96,02	RY/BY# pulse on write
96,03	RY/BY# pulse on erase
96,04	RY/BY# pin disable
97	upload BSRs with lock bit
99	upload device information
A7	erase all unlocked blocks
E0	sequential load to page buffer
F0	sleep
FB	two-byte write

"C" DRIVERS

```
/*************************************************************************/
/* Copyright Sharp Corporation, 1995                                     */
/* File : stddefs.h                                                      */
/* Standard definitions for C Drivers for the LH28F016SA/SU, LH28F800SU  */
/* Flash memory components                                               */
/*************************************************************************/

/*************************************************************************/
/* pin values                                                            */
/*************************************************************************/
#define LOW                    0
#define HIGH                   1

/*************************************************************************/
/* error codes                                                           */
/*************************************************************************/
#define NO_ERROR               0
#define VPP_LOW                1
#define OP_ABORTED             2
#define BLOCK_LOCKED           3
#define COMMAND_SEQ_ERROR      4
#define WP_LOW                 5

/*************************************************************************/
/* bit masks                                                             */
/*************************************************************************/
#define BIT_0                  0x0001
#define BIT_1                  0x0002
#define BIT_2                  0x0004
#define BIT_3                  0x0008
#define BIT_4                  0x0010
#define BIT_5                  0x0020
#define BIT_6                  0x0040
#define BIT_7                  0x0080

#define LOW_BYTE               0x00FF
#define HIGH_BYTE              0xFF00

/*************************************************************************/
/* RY/BY# enable modes                                                   */
/*************************************************************************/
#define RYBY_ENABLE_TO_LEVEL   1
#define RYBY_PULSE_ON_WRITE    2
#define RYBY_PULSE_ON_ERASE    3
#define RYBY_DISABLE           4

/*************************************************************************/
/* pin numbers                                                           */
/*************************************************************************/
#define WPB 56
/* Write Protect pin (active low) is pin number 56 on standard          */
/* pinout of LH28F016SA.                                                 */

#define VPP 15
/* Vpp Pin is pin number 15 on standard pinout of LH28F016SA.           */
```

```
/*************************************************************************/
/* Copyright Sharp Corporation, 1995                                     */
/* File : drivers.c                                                      */
/* Example C Routines for LH28F016SA/SU,LH28F800SU Flash memory components */
/*                                                                       */
/* NOTE: BYTE# pin on the device determines whether addressing           */
/* refers to words or bytes. I assume word mode.                         */
/* NOTE: A LH28F016SA command can be written to any address in the       */
/* block or device to be affected by that command.                       */
/*************************************************************************/

#include <stdio.h>
#include "stddefs.h"

void set_pin(int pin, int level)
{
/* set_pin is an implementation-dependent function which sets a          */
/* given pin on the standard LH28F016SA pinout HIGH = 1 or LOW = 0       */
}

int get_pin(int pin)
{
/* get_pin is an implementaton-dependent function which returns a        */
/* given pin on the standard LH28F016SA pinout HIGH =1 or LOW = 0        */
}

int *base(int *address)
{
/* base is an implementation-dependent function which takes an           */
/* address in the flash array and returns a pointer to the base          */
/* of that 64K byte block.                                               */
}

char *byte_base(char *address)
{
/* byte version of base function described above                         */
}
```

```
int CSR_word_byte_writes(int *address, int data)
{
/* this procedure writes a byte to the LH28F016SA.                              */
/* It also works with the LH28F008SA.                                            */
int CSR;
/* CSR  variable is used to return contents of CSR register.                    */

*address = 0x1010;
/* Wrod Write command                                                            */
*address = data;
/* Actual data write to flash address.                                           */
while (!(BIT_7 & *address));
/* Poll CSR until CSR.7 = 1 (WSM ready)                                          */

CSR = *address;
/* Save CSR before clearing it.                                                  */
*address = 0x5050;
/* Clear Status Registers command                                                */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.           */
return(CSR);
/* Return CSR to  be checked for status of operation.                           */
}
```

```
int CSR_block_erase(int *address)
{
/* This procedure erases a 64K byte block on the LH28F016SA.                  */
int CSR;
/* CSR variable is used to return contents of CSR register.                   */

*address = 0x2020;
/* Single Block Erase command                                                 */
*address = 0xD0D0;
/* Confirm command                                                            */
*address = 0xD0D0;
/* Resume command, per latest errata update                                   */
while (!(BIT_7 & *address))
/* Poll CSR until CSR.7 = 1 (WSM ready)                                       */
{
/* System may issue an erase suspend command (B0[B0]) here to read data       */
/* from a different block.                                                    */
};

/* At this point, CSR.7 is 1, indicating was is not busy.                     */
/* Note that we are still reading from CSR by default.                        */
CSR = *address;
/* Save CSR before clearing it.                                               */
*address = 0x5050;
/* Clear Status Registers command                                             */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.         */
return(CSR);
/* Return CSR to be checked for status of operation.                          */
}
/* If a write has been queued, an automatic erase suspend occurs to write     */
/* to a different block.                                                      */
```

```
int CSR_erase_suspend_to_read(int *read_address, int *erase_address, int *resu
lt)
{
/* This procedure suspends an erase operation to do a read.                 */
int CSR;
/* CSR variable is used to return contents of CSR register.                 */

/* Assume erase is underway in block beginning at erase_address.            */
*erase_address = 0xB0B0;
/* Erase Suspend cmmand                                                     */
while (!(BIT_7 & *erase_address));
/* Poll CSR until CSR.7 = 1 (WSM ready)                                     */
if (BIT_6 & *erase_address) {
/* If CSR.6 = 1 (erase incomplete)                                          */
        *erase_address = 0xFFFF;
        /* Read Flash Array command                                         */
        *result = *read_address;
        /* Do the actual read. Any number of reads can be done here.        */
        *erase_address = 0xD0D0;
        /* Erase Resume command                                             */
} else {
        *erase_address = 0xFFFF;
        /* Read Flash Array command                                         */
        *result = *read_address;
        /* Do the actual read. Any number of reads can be done here.        */
}
*erase_address = 0x7070;
/* Read CSR command                                                         */
CSR = *erase_address;
/* Save CSR before clearing it.                                             */
*erase_address = 0x5050;
/* Clear Status Registers command                                           */
return(CSR);
/* Return CSR to be checked for status of operation.                        */
}
```

```
int lock_block(int *lock_address)
/* This procedure locks a block on the LH28F016SA.                                    */
{
int ESR;
/* ESR variable is used to return contents of GSR and BSR.                            */

int *block_base = base(lock_address);
/* Find pointer to base of block being locked.                                        */

*lock_address = 0x7171;
/* Read Extended Status Registers command                                             */
while (BIT_3 & *(block_base + 2));
/* Poll GSR until GSR.3 = 0 (queue available).                                        */
set_pin(WPB, HIGH);
/* Disable write protection by setting WPB high.                                      */
set_pin(VPP, HIGH);
/* Enable Vpp, wait for ramp if necessary in this system.                             */
*lock_address = 0x7777;
/* Lock Block command                                                                 */
*lock_address = 0xD0D0;
/* Confirmation command                                                               */
*lock_address = 0x7171;
/* Read Extended Status Registers command                                             */
while (!(BIT_7 & *(block_base + 2)));
/* GSR is 2 words above 0; poll GSR  until GSR.7 = 1 (WSM ready).                     */

ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
/* Put GSR in top byte and BSR in bottom byte of return value.                        */
*lock_address = 0x5050;
/* Clear Status Registers command                                                     */
*lock_address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.                 */
return(ESR);
}
```

```
int lock_status_upload_to_BSR(int *address)
/* This procedure uploads status information into the BSR from                */
/* nonvolatile status bits.                                                    */
{
int ESR;
/* ESR variable is used to return contents of GSR and BSR.                     */
int *block_base = base(address);
/* Find pointer to base of 32K word block.                                     */

*address = 0x7171;
/* Read Extended Status Registers command                                      */
while (BIT_3 & *(block_base + 2));
/* Poll GSR until GSR.3 = 0 (queue available).                                 */
*address = 0x9797;
/* Lock-status Upload command                                                  */
*address = 0xD0D0;
/* Confirmation command                                                        */
*address = 0x7171;
/* Read Extended Status Registers command                                      */
while (!(BIT_7 & *(block_base + 2)));
/* Poll GSR until GSR. 7 = 1 (WSM not busy)                                    */

ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
/* Put GSR in top byte and BSR in  bottom byte of return value.                */
*address = 0x5050;
/* Clear Status Registers command                                              */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.          */
return(ESR);
}
```

```
int update_data_in_a_locked_block()
{
/* This routine is implemented a pseudo-code to provide an example of        */
/* impementing the flowchart Updating Data in a Locked Block from the         */
/* LH28F016SA User's Manual                                                   */

set_pin(WPB, HIGH);
/* set WP# high                                                               */
block_erase_with_CSR(block_address);
/* erase block                                                                */
set_pin(WPB, LOW);
/* set WP# low                                                                */
WriteNewData();
/* Use one of Word/Byte Write, Two-Byte Write or Page Buffer Write to Flash */

lock_block(block_address);
/* lock block if desired                                                      */
}


int add_data_in_a_locked_block()
{
/* This routine is implemented as pseudo-code to poovide an example of       */
/* impementing the flowchart Updating Data in a Locked Block from the         */
/* LH28F016SA User's Manual                                                   */
set_pin(WPB, HIGH);
/* set WP# high                                                               */
WriteNewData();
/* Use one of Word/Byte Write, Two-Byte Write or Page Buffer Write to Flash */
set_pin(WPB, LOW);
/* set WP# low                                                                */
}
```

```
int ESR_word_write(int *write_address, int *data, int word_count)
/* This procedure writes a word to the LH28F016SA.                          */
{
int counter, ESR;
/* counter is used to loop through data array                               */
/* ESR variable is used to return contents of GSR and BSR.                  */
int *block_base = base(write_address);

for (counter = 0, counter < word_count; counter++) {
        *write_address = 0x7171;
        /* Read Extended Status Registers command                           */
        while (BIT_3 & *(block_base + 1));
        /* Poll BSR until BSR.3 of target address = 0 (queue available).   */
        /* BSR is 1 word above base of target block in status reg space.   */
        *write_address = 0x1010;
        /* Write word command                                               */
        *write_address = data[counter];
        /* Write actual data.                                               */
}

*write_address = 0x7171;
/* Read Extended Status Registers command                                   */
while (!(BIT_7 & *(block_base + 1))):
/* Poll BSR until BSR.7 of target address = 1 (block ready).                */
ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
/* Put GSR in top byte and BSR in bottom byte of return value.              */
*write_address = 0x5050;
/* Clear Status Registers command                                           */
*write_address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.       */
return(ESR);
}
```

```
int two_byte_write(char *address, char *data, int byte_count)
/* This routine is used when BYTE# is low, i.e. the LH28F016SA               */
/* is in byte mode, to emulate a word write.                                 */
/* Because of this, commands are given as 0x00XY instead of 0xXYXY as in     */
/* the rest of the code presented here.                                      */
/* *data is a byte array containg the low byte, high byte consecutively of */
/* each word.                                                                */
{
int counter, ESR;
/* ESR variable is used to return contents of GSR and BSR.                  */
char *block_base = byte_base(address);
/* Find pointer to base of block.                                            */

for (counter = 0; counter < byte_count; counter++) {
        *address = 0x0071 ;
        /* Read Extended Status Registers command                            */
        while (BIT_3 & *(block_base + 2));
        /* Poll BSR until BSR.3 of target address = 0 (queue available).    */
        *address = 0x00FB;
        /* Two-byte Write command                                            */
        *address = data[counter++];
        /* Load one byte of data register; A0 = 0 loads low byte, A1 high  */
        *address = data[counter];
        /* LH28F016SA automatically loads alternate byte of data register  */
}

/* Write is intiated. Now we poll for successful completion.                */
*address = 0x0071;
/* Read Extended Status Registers command                                    */
while (!(BIT_7 & *(block_base + 2)));
/* Poll BSR until BSR. 7 of target address = 1 (block ready).               */
/* BSR is 1 word above base of target block in status reg space.            */

ESR = (*(block_base + 4) << 8) + (*(block_base + 2) & LOW_BYTE);
/* Put GSR  in top byte and BSR in bottom byte of return value.             */
*address = 0x0050;
/* Clear Status Registers command                                            */
*address = 0x00FF;
/* Write FFH after last operation to reset device to read array mode.      */
return(ESR);
}
```

```
int ESR_pagebuffer_write(int *address, int word_count)
/* This procedure writes from page buffer to flash.                            */
{
/* This routine assumes page buffer is already loaded.                          */
/* Address is where in flash array to begin writing.                            */
/* Low byte of word count word_count must be 127 or fewer, high must be 0.     */
/* High byte of word count exists for future Page Buffer expandability.        */
int ESR;
/* ESR variable is used to return contents of GSR and BSR.                      */
int *block_base = base(address);
/* Find pointer to base of block to be written.                                 */

*address = 0x7171;
/* Read Extended Status Registers command                                       */
while (BIT_3 & *(block_base + 1));
/* Poll BSR until BSR.3 of target address = 0 (queue available).               */
*address = 0x0C0C;
/* Page Buffer Write to Flash command                                           */
*address = word_count;
/* high byte is a don't care, write the low byte                                */
*address = 0;
/* write high byte of word_count which must be 0 (reserved for future use) */
*address = 0x7171;
/* Read Extended Status Registers command                                       */
while (!(BIT_7 & *(block_base + 1)));
/* Poll BSR until BSR.7 of target address = 1 (block ready).                   */

ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
/* Put GSR in top byte and BSR in bottom byte of return value.                  */
*address = 0x5050;
/* Clear Status Registers command                                               */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.          */
return(ESR);
}
```

```
int ESR_block_erase(int *erase_address)
/* This procedure erases a block on the LH28F016SA.                    */
{
int ESR;
/* ESR variable is used to return contents of GSR and BSR.             */
int *block_base = base(erase_address);
/* Find address of base of block being erased.                         */

*erase_address = 0x7171;
/* Read Extended Status Registers command                              */
while (BIT_3 & *(block_base + 1));
/* Poll BSR until BSR.3 of erase_address = 0 (queue available).        */
/* BSR is 1 word above base of target block in ESR space.              */

*erase_address = 0x2020;
/* Single Block Erase command                                          */
*erase_address = 0xD0D0;
/* Confirm command                                                     */
*erase_address = 0xD0D0;
/* Resume command, per latest errata update                            */
*erase_address = 0x7171;
/* Read Extended Status Registers command                              */
while (!(BIT_7 & *(block_base + 1)));
/* Poll BSR until BSR.7 of target erase_address = 1 (block ready).     */

ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
/* Put GSR in top byte and BSR  in bottom byte of return value.        */
*erase_address = 0x5050;
/* Clear Status Registers command                                      */
*erase_address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.  */
return (ESR);
}
```

```
int ESR_erase_all_unlocked_blocks(int *device_address, long *failure_list)
/* This procedure erases all the unlocked blocks on a LH28F016SA.                  */
{
int GSR;
/* Return value will contain GSR in both top and bottom byte.                      */
/* 32 bit long pointed to by failure_list is used to return map                    */
/* of blocck failures, each bit representing one block's status.                   */
/* device_address points to base of chip.                                          */
int block;
/* block is used to hold block count for loop through blocks.                      */
long power = 1;

*failure_list = 0;
/* Initialize all 32 bits of failure list long to 0.                               */
*device_address = 0x7171;
/* Read Extended Status Registers                                                  */
while (BIT_3 & *(device_address + 1));
/* Poll BSR until BSR.3 of target address = 0 (queue available).                   */
*device_address = 0xA7A7;
/* Full_chip erase command                                                         */
*device_address = 0xD0D0;
/* Confirm command                                                                 */
*device_address = 0x7171;
/* Read Extended Status Registers command                                          */
while (!(BIT_7 & *(device_address + 2)));
/* Poll GSR until GSR. 7 = 1 (WSM ready)                                           */

for (block = 0; block < 0x0020; block++)
{
        /* Go through blocks, looking at each BSR. 5 for operation failure */
        /* and setting appropriate bit in long pointed to by failure list. */
        if (BIT_5 & *(device_address + block * 0x8000 + 1))
        /* Multiply block by 32K words to get to the base of each block.   */
                *failure_list += power;
                /* If the block failed, set that bit in the failure list.  */
        power = power << 1;
        /* Increment to next power of two  to access next bit.             */
}

GSR = *(device_address + 2);
*device_address = 0x5050;
/* Clear Status Registers command                                                  */
*device_address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.              */
return(GSR);
}
```

```
int ESR_suspend_to_read_array(int *address, int *result)
/* This procedure suspends an erase on the LH28F016SA.                          */
{
/* Address is assumed to point to location to be read.                          */
/* result is used to hold read value until proecedure is complete.              */
int ESR;
/* ESR variable is used to return contents of GSR and BSR.                      */
int *block_base = base(address);

*address = 0x7171;
/* Read Extended Status Registers command                                       */
while (!(BIT_7 & *(block_base + 1)));
/* Poll BSR until BSR.7 of target address = 1 (block ready).                    */
/* BSR is 1 word above base of target block in ESR space.                       */
*address = 0xB0B0;
/* Operation Suspend command                                                    */
*address = 0x7171;
/* Read Extended Status Registers command                                       */
while (!(BIT_7 & *(block_base + 2)));
/* Poll GSR until GSR.7 =1 (WSM ready).                                         */
if (BIT_6 & *(block_base + 2)) {
/* GSR.6 = 1 indicates an operation was suspended on this device,              */
        *address = 0xFFFF;
        /* Read Flash Array command                                             */
        *result = *address;
        /* Read the data                                                        */
        *address = 0xD0D0;
        /* Resume the operation.                                                */
} else {
        *address = 0xFFFF;
        /* Read Flash Array command                                             */
        *result = *address;
        /* Read the data.                                                       */
}

*address = 0x7171;
/* Read Extended Status Registers command                                       */
ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
/* Put GSR in top byte and BSR in bottom byte of return value.                  */
*address = 0x5050;
/* Clear Status Registers command                                               */
return(ESR);
}
```

```
int ESR_automatic_erase_suspend_to_write(int *write_address, int *erase_addres
s, int data)
/* This procedure writes to one block while another is erasing.            */
{
int ESR;
/* ESR variable is used to return contents of GSR and BSR.                 */
int *block_base = base(erase_address);
/* Find pointer to base of block being erased.                             */

*erase_address = 0x7171;
*/ Read Extended Status Register command                                   */
while (BIT_3 & *(block_base + 1));
*/ Poll BSR until BSR.3 of targer address = 0 (queue available).           */
*/ BSR is 1 word above base of target block in ESR space.                  */
*erase_address = 0x2020;
*/ Erase Block command                                                     */
*erase_address = 0xD0D0;
*/Confirm command                                                          */
*erase_address = 0x7171;
*/ Read Extended Status Register command                                   */
While (BIT_3 & *(block_base + 1));
*/ Poll BSR until BSR.3 of target address = 0 (queue available).           */
*/ BSR is 1 word above base of target block in ESR space.                  */
*write_address = 0x4040;
*/ Word write command                                                      */
*write_address = data;
*/ Write actual data.                                                      */
*/ Erase suspends, write takes place, then erase resumes.                  */
*erase_address = 0x7171;
/* Read Extended Status Registers command                                  */
while (!(BIT_7 & *(block_base + 1)));
*/ Poll BSR until BSR.7 of erase address = 1 (block ready).                */
*/ BSR is 1 word above base of target block in status reg space.           */

ESR = (*(block_base + 2) << 8) + (*(block_base + 1) & LOW_BYTE);
*/ Put GSR  in top byte and BSR  in bottom byte of return value.           */
*block_base = 0x5050;
*/ Clear Statue Registers command                                          */
*block_base = 0xFFFF;
*/ Write FFH after last operation to reset device to read array mode.      */
return(ESR);
}
```

```
int ESR_full_status_check_for_data_write(int *device_address)
{
int errorcode;

*device_address = 0x7171;
/* Read Extended Status Registers command                                   */
while (!(BIT_7 & *(device_address + 2)));
/* Poll GSR until GSR.7 = 1 (WSM ready)                                     */
/* to make sure data is valid                                               */

if (*(device_address + 1) & BIT_2) errorcode = VPP_LOW;
/* BSR.2 = 1 indicates a Vpp Low Detect                                     */
else if (*(device_address + 1) & BIT_4) errorcode = OP_ABORTED;
/* BSR.4 = 1 indicates an Operation Abort                                   */

else if (get_pin(WPB) == LOW) errorcode = NO_ERROR;
else if (*(device_address + 1) & BIT_6) errorcode = BLOCK_LOCKED;
/* BSR.6 = 1 indicates the Block was locked                                 */
else errorcode = NO_ERROR;

while (!(BIT_7 & *(device_address + 2)));
/* Poll GSR until GSR.7 = 1 (WSM ready)                                     */
/* make sure chip is ready to accept command                                */

*device_address = 0x5050;
/* Clear Status Registers                                                   */

return errorcode;
}
```

```
int ESR_full_status_check_for_erase(int *device_address)
{
int errorcode = NO_ERROR;

*device_address = 0x7171;
/* Read Extended Status Registers command                                     */
while (!(BIT_7 & *(device_address + 2)));
/* Poll GSR until GSR.7 = 1 (WSM ready)                                       */
/* make sure command completed                                                */
if (*(device_address + 1) & BIT_2) errorcode = VPP_LOW;
/* BSR.2 = 1 indicates a Vpp Low Detect                                       */
else if (*(device_address + 1) & BIT_4) errorcode = OP_ABORTED;
/* BSR.4 = 1 indicates an Operation Abort                                     */
else if (get_pin(WPB) == LOW && !(*(device_address + 1) & BIT_6))
        errorcode = BLOCK_LOCKED;
/* BSR.6 = 0 indicates the Block was locked                                   */
if (errorcode == NO_ERROR) {
        *device_address = 0x7070;
        /* Read Compatible Status Register                                    */

        if ((*device_address & BIT_4) && (*device_address & BIT_5))
        /* CSR.4 and CSR.5 == 1 indicate a command sequence error             */
                errorcode = COMMAND_SEQ_ERROR;
}

while (!(BIT_7 & *(device_address + 2)));
/* Poll GSR until GSR.7 = 1 (WSM ready)                                       */
/* make sure device is ready                                                  */

*device_address = 0x5050;
/* Clear Status Registers command                                             */

return errorcode;
}
```

```
void single_load_to_pagebuffer(int *device_address, char *address, int data)
/* This procedure loads a single byte or word to a page buffer.             */
/* device_address points to base of chip.                                   */
{
*device_address = 0x7171;
/* Read Extended Status Registers command                                   */
while (!(BIT_2 & *(device_address + 2)));
/* Poll GSR until GSR.2 = 1 (page buffer available)                         */
*device_address = 0x7474;
/* Single Load to Page Buffer command                                       */
*address = data;
/* Actual write to page buffer                                              */
/* This routine does not affect status registers.                           */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.      */
}
```

```
void sequential_load_to_pagebuffer(int *device_address, char *start_address, i
nt word_count, int* data)
/* This procedure loads multiple words to a page buffer.                      */
/* device_address points to base of chip.                                     */
{
/* Low byte of word_count must be 127 or fewer, high must be 0.               */
/* word_count is zero-based counting, i.e word_count == 0 loads 1 word,       */
/* word_count == 1 loads 2 words etc.                                         */
/* High byte of word_count exists for future Page Buffer expandability.       */
char counter;
/* counter is used to keep track of words written.                            */

*device_address = 0x7171;
/* Read Extended Status Registers command                                     */
while (BIT_2 & *(device_address + 2));
/* Poll GSR until GSR.2 = 0 (page buffer available).                          */
*device_address = 0xE0E0;
/* Sequential Page Buffer Load command                                        */
*start_address = word_count;
*start_address = 0;
/* Automatically loads high byte of count register                            */
for (counter = 0; counter <= word_count; counter++)
       *(start_address + counter) = data[counter];
/* Loop through data, writing to page buffer.                                 */
/* This routine does not affect status registers.                             */
*device_address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.         */
}
```

```
int upload_device_information(int *address)
/* This procedure uploads the device revision number to the variable DRC.  */
/* This implementation differs in that it does not loop as in the         */
/* algorithm. This is so the calling routine can have a chance to do      */
/* error checking instead of looping forever waiting for the device       */
/* complete the operation.                                                 */
{
int DRC = 0;
/* DRC variable is used to return device revision status.                  */
int *block_base = base(address);
/* Find pointer to base of ˜32K word block.                                */

*address = 0x7171;-
/* Read Extended Status Registers command                                  */
while ((BIT_3 & *(block_base + 2)) && (!(BIT_7 & *(block_base + 2))));
/* Poll GSR until GSR.3 = 0 (queue available) and GSR.7 = 1                */
/* (WSM available)                                                         */
*address = 0x9999;
/* Device information Upload command                                       */
*address = 0xD0D0;
/* Confirmation command                                                    */
*address = 0x7171;
/* Read Extended Status Registers command                                  */
while (!(BIT_7 & *(block_base + 2)));
/* Poll GSR until GSR.7 = 1 (WSM not busy)                                 */
if (BIT_5 & *(block_base + 2)) return ((*(block_base + 2) & HIGH_BYTE) << 8);
/* if GSR.5 = 1 operation was unsuccessful. Return GSR and 0 in DRC        */
*address = 0x7272;
/* Swap page buffer to bring buffer with status information to top.        */
*address = 0x7575;
/* Read page Buffer command                                                */
DRC = (*(block_base + 2) & 0xFF00 << 8) + (*(block_base + 3) & LOW_BYTE);
/* Put GSR in top byte of return value.                                    */
/* User should check GSR for operation success                             */
/* Put device revision code in bottom byte of return value.                */
/* Note that device revision code was read from word 3 in page buffer.     */
*address = 0x5050;
/* Clear Status registers command                                          */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.      */
return(DRC);
}
```

```
int RYBY_reconfigure(int *address, int mode)
{
/* This procedure changes the RY/BY# configuration mode to the given mode  */
int GSR;
/* the GSR variable is used to return the value of the GSR                  */

*address = 0x7171;
/* Read Extended Registers command                                          */
while (BIT_3 & *(address + 2));
/* Poll GSR until GSR.3 = 0 (queue available)                               */
*address = 0x9696;
/* Enable RY/BY# configuration, next command configures RY/BY#              */
switch (mode) {
        case RYBY_ENABLE_TO_LEVEL:
                *address = 0x0101;
                /* Enable RY/BY# to level mode                              */
                break;

        case RYBY_PULSE_ON_WRITE:
                *address = 0x0202;
                /* Enable RY/BY# to pulse on write                          */
                break;

        case RYBY_PULSE_ON_ERASE:
                *address = 0x0303;
                /* Enable RY/BY# to pulse on erase                          */
                break;

        case RYBY_DISABLE:
        default:
                *address = 0x0404;
                /* Enable RY/BY# to disable                                 */
                break;
}
*address = 0x7171;
/* Read Extended Status Registers command                                   */
while (!(BIT_7 & *(address + 2)));
/* Poll GSR until GSR.7 = 1 (WSM ready)                                     */
GSR = *(address + 2) & LOW_BYTE;
/* put GSR into low byte of return value                                    */
*address = 0x5050;
/* Clear Status registers command                                           */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.       */
return(GSR);
}
```

```
int page_buffer_swap(int *address)
{
/* This routine attempts to swap the page buffers, returning the value of  */
/* the GSR before the operation in the upper byte and the value of the GSR */
/* after the operation in the lower byte for comparison                    */
/* For operation to be successful, the following must be true :            */
/* (before) GSR.0 = (after) !GSR.0                                         */
/* (after) GSR.5 = 0                                                       */
int GSR;
/* GSR variable is used to return contents of GSR before and after         */
/* operation                                                               */

*address = 0x7171;
/* Read Extended Registers command                                         */
GSR = *(address + 2) << 8;
/* Put GSR into upper byte before page buffer swap                         */
*address = 0x7272;
/* Write Page Buffer Swap command                                          */
GSR |= (*(address + 2) & LOW_BYTE);
/* Put GSR after operation into low byte for comparison                    */
*address = 0x5050;
/* Clear Status registers command                                          */
*address = 0xFFFF;
/* Write FFH after last operation to reset device to read array mode.      */
return(GSR);
}
```

Flash, Software, Drivers, Non-Volatile, LH28F016SA, LH28F016SU, LH28F800SU