The Moncky firmware

Kris Demuynck

17th August 2022

Abstract

This document describes the Moncky firmware. This code is available to the programmer and can be linked to programs. This document is a work in progress and will be updated regularly. The home of the Moncky project is

https://hackaday.io/project/181269-the-moncky-project

and

https://gitlab.com/big-bat/moncky

for all source code.

Copyright

Kris Demuynck

The Moncky firmware

Contents

Chapter 1

Introduction

After the boot sequence is done, control is handed over to the code in 'MONCKYOS'. This can be whatever the programmer wants it to be. However, some functionality has already been implemented and is available in the repository on gitlab in the 'firmware' folder. These files can be linked to your own code by the assembler.

The firmware is automatically linked with any code that is produced by the Moncky compiler (described in a separate document).

The home page for the Moncky project is:

https://hackaday.io/project/181269-the-moncky-project

All code and documentation can be found in gitlab:

https://gitlab.com/big-bat/moncky

Chapter 2

System setup

The system starts at address 0x0000. The following commands are set here ('system.asm'):

```
.def :startOfStack 0xFFFE
.def :system_screenConfig 0x7D00
.org 0x0000
        di
        li sp, :startOfStack
        lih sp, ::startOfStack
        li r0, :start_system
        lih r0, ::start_system
        jp [r0]
.org 0x0008
        reset
        reset
        reset
        reset
        reset
        reset
        reset
        reset
.org 0x0010
        push r0 ; save all registers and flags
        push r1
        push r2
        sflags r2
        li r0, :system_screenConfig
        lih r0, ::system_screenConfig
        in r1, (r0)
        xori r1, 2 ; toggle bit
        out r1, (r0)
        rflags r2 ; restore flags and registers
        pop r2
        pop r1
        pop r0
        reti
```

Basically the processor will set the top of stack and jump to the start of the system, which is located further away (at least after the interrupt handler).

When the ROM disables the ROM shadowing, it is executing the following instructions:

```
.org 0x0008
li r0, 0xFF
lih r0, 0xFF
li r1, 0xAA
lih r1, 0xAA
st r1, (r0)
reset
```

This means that the processor will execute the 'reset' instruction at address 0x000D. In order for this to work, there should also be a reset instruction in this location in the operating system. This is why there are reset instructions from address 0x0008 till 0x000F.

At address 0x0010 the hardware interrupt is located. In the default code, the interrupt handler will toggle the second bit in the screen configuration. This bit is not used by the hardware. It can be used to detect the vertical retrace in software.

Important: the interrupt handler must always save all registers that are used and also the flags! Otherwise the code will not run properly when the processor returns to the point the interrupt occurred!

Chapter 3

Graphics

The graphics library ('graphics.asm') contains functionality to control the pixels on the screen.

The following subroutines are available (in C syntax):

- void graphics_init(): initialises the screen to default values (does not clear the screen)
- void graphics_setBgCol(uint8 col): sets the background colour to the specified value
- void graphics_clrscr(): clears the screen (with the background colour)
- void graphics_setCol(uint8 col): sets the current colour (used by plot and line)
- void graphics_setScreenBuffer(uint1 buffer): sets the current buffer to be displayed on the screen (happens after a vertical retrace)
- void graphics_waitVertRetrace(): waits until the next vertical retrace happened. Remark: the interrupts should be enabled for this to function!
- void graphics_setDrawBuffer(uint1 buffer): sets the current buffer to draw on (used by plot and line)
- uint8 graphics_getDrawBuffer(): gets the current buffer to draw on
- void graphics_plot(uint16 x, uint16 y): plots a point on the given coordinates with the current colour on the current buffer
- void graphics_line(uint16 x0, uint16 y0, uint16 x1, uint16 y1): draws a line between 2 given points with the current colour on the current buffer
- void graphics_drawHorizontalEven(uin16 x, uint16 y, uint16 width): draws a horizontal line from the given point to the right. Both x and width are supposed to be even
- void graphics_drawHorizontal(uin16 x, uint16 y, uint16 width): draws a horizontal line from the given point to the right. The width is assumed to be at least 2
- void graphics_drawVertical(uin16 x, uint16 y, uint16 height): draws a vertical line from the given point towards the bottom
- void graphics_drawRectangle(uin16 x1, uint16 y1, uint16 width, uint16 height): draws a rectangle from the top-left corner with the given width and height
- void graphics_fillRectangle(uin16 x1, uint16 y1, uint16 width, uint16 height): fills a rectangle from the top-left corner with the given width and height

- void graphics_scrollUp(uint16 amount): scrolls the screen up by the given amount of lines. The bottom lines are filled with the background colour
- void graphics_drawSprite(uint8 sprite[16][16], uint8 buffer[16][16], uint16 x, uint16 y): draws a sprite at the given location. The pixels that are overwritten are saved in the specified buffer
- void graphics_restoreSprite(uint8 buffer[16][16], uint16 x, uint16 y): restores the saved pixels from the buffer, erasing the sprite

Chapter 4

Text

The text library ('text.asm') contains functionality to draw text on the screen. It uses the graphics library to draw pixels. Characters are drawn in a 8x8 pixel grid. The library currently contains a font for all 128 ASCII characters which is based on the font of the CPC464 computer from Amstrad. The first 32 characters are special:

- 0x01: a heart character (to show the authors love to his partner :-))
- 0x02: an 8x8 square that can be used to display a cursor
- 0x03, 0x04, 0x05, 0x06, 0x07, 0x0B, 0x0C, 0x0E, 0x0F: characters to draw the BigBat logo
- 0x08: back space
- 0x09: tab
- 0x0A: new line
- 0x0D: carriage return
- 0x11: arrow left
- 0x12: arrow right
- 0x13: arrow up
- 0x14: arrow down

The following subroutines are available:

- void text_init(): sets cursor at (0,0)
- void text_setCursor(uint8 x, uint8 y): sets the cursor at the given coordinates (x<80 and y<25)
- void text_drawCursor(): draws the cursor at the current cursor position while saving the overwritten part in memory
- void text_eraseCursor(): restores the part of the screen that was overwritten by the cursor
- void text_drawChar_alt(uint16 x, uint16 y, uint8 char, uint16* fonttable): draws a character at the given (x,y) coordinates. When pixels are 0, the background colour is used. x must be even (otherwise the lsb is erased so it becomes even). The given font table is used.

- void text_drawChar_transparent_alt(uint16 x, uint16 y, uint8 char, uint16* fonttable): draws a character at the given (x,y) coordinates. When pixels are 0, the background is preserved. The given font table is used.
- void text_drawChar(uint16 x, uint16 y, uint8 c): draws a character (0-127) at the given coordinates (x<320 and y<200). The internal font is used.
- void text_printChar(uint8 c): draws a character at the actual cursor position. This function will also move the cursor and scroll the screen up when necessary. It recognises the arrow left, right, up, and down commands as well as the backspace and newline characters.
- void text_printString16(uint16* string): prints a null-terminated string on the screen. There is one 16-bit word per character.
- void text_printString(char* string): prints a null-terminated string on the screen. There are two characters per 16-bit word: the LSB is the first character and the MSB is the second. This allows for more compact storage of large strings.
- void text_printHex(uint16 value): prints the given 16-bit value on the screen in 4 hexadecimal digits (with leading zeroes)
- void text_printDecimal(uint16 value): prints the given 16-bit value on the screen in decimal digits (without leading zeroes). The value is supposed to be positive. If negative, the 10's complement is printed

Chapter 5

Keyboard

The keyboard library ('keyboard.asm') contains functionality to read the next key from the keyboard. It interacts with the hardware keyboard buffer. The keyboard buffer will emit 'key codes' which are 10 bit values. The 9 least significant bits are the value of the key and the 10th bit indicates if the key was released. The 8th bit indicates an 'extended key code'.

When a key code is translated to ASCII the following special keys are defined:

- 0x01: F1
- 0x02: F2
- 0x03: F3
- 0x04: F4
- 0x05: F5
- 0x06: F6
- 0x07: F7
- 0x08: backspace
- 0x09: tab
- 0x0A: enter
- 0x0B: F8
- 0x0C: F9
- 0x0D: carriage return
- 0x0E: F10
- 0x0F: F11
- 0x10: F12
- 0x11: arrow left
- 0x12: arrow right
- 0x13: arrow up

- 0x14: arrow down
- 0x15: delete
- 0x16: insert
- 0x17: page up
- 0x18: page down
- 0x19: home
- 0x1A: end
- 0x1B: esc
- 0x1C: menu

The following subroutines are available:

- void keyboard_init(): will erase the keyboard buffer in the keyboard controller
- uint16 keyboard_readKeyCode(): will return the key code of the next key in the buffer. If no key was pressed, zero is returned.
- uint16 keyboard_waitKey(uint16 timeout): will wait until a key is pressed or released. It will return 0 if a timeout occurs (time is in milliseconds). If the timeout is 0 the computer will wait indefinitely.
- char keyboard_toChar(uint16 keycode): will return the ASCII code that corresponds with the given key code. It translates key codes to ASCII. If the shift key was pressed the corresponding value will be altered accordingly.
- char keyboard_readChar(): will return the ASCII code that corresponds with the key pressed on the keyboard (this is readKeyCode followed by toChar)
- uint16 keyboard_waitChar(uint16 timeout): wait until a key is pressed and return the corresponding ASCII code (this completely masks key releases, only presses are detected).

Chapter 6

Utilities

The utilities library ('utilities.asm') contains some extra utilities that can be handy.

The following subroutines are available:

- void util_log(message, numberOfParameters): this is a special function. It does not follow the normal calling convention. The parameters are passed as r1 and r2 respectively. The function is called right after the 'push bp, set bp, sp' sequence. It will display the current time (milliseconds), a message and all parameters on screen. A programmer can use this to debug software
- void util_memcopy(uint16* source, uint16* destination, uint16 length): this function copies a given amount of 16-bit numbers from source to destination. The data is copied from lowest to highest address
- void util_memclear(uint16* destination, uint16 length, uint16 value): this function fills a given amount of 16-bit numbers with a given value. The data is copied from highest to lowest address
- void util_sleepMillis(uint16 time): this function will wait for the specified amount of milliseconds. This is a blocking wait.
- void util_getMillis(): uint32: this function returns the number of milli seconds since startup
- void util_getMicros(): uint32: this function returns the number of micro seconds since startup
- void util_error(uint16 code): this function will display an error on the screen, with the given code in hexadecimal
- void util_enableInterrupts(): this function will enable the interrupts
- void util_disableInterrupts(): this function will disable the interrupts

Chapter 7

SPI

The SPI library ('spi.asm') contains functionality to initialise and use the SPI interface.

The following subroutines are available:

- void spi_init(): this will set the slave select pins to high, SPI mode to 0, and speed to maximum.
- void spi_enable(uint8 slave): this will put the selected slave select pin to 0 and the others to 1
- void spi_disable(): this will put all slave select pins to 1
- void spi_setSpeed(uint8 speed): this will put the speed of the SPI interface to the given value (the value is calculated as indicated in section **??**).
- uint8 spi_transfer(uint8 byte): this will transfer a byte to the SPI slave. The return value is the received value.

Chapter 8

SD card

The SD card library ('sd.asm') contains functionality to initialise and read blocks from an SD card. Writing is currently not implemented.

The SD card is supposed to be connected to the SPI interface. A nice introduction to SD cards can be found at: http://www.rjhcoding.com/avrc-sd-interface-1.php.

The code is derived from the following C-code:

```
#define SS 7
#define MISO 10
#define MOSI 8
#define SCK 9
#define SD_SUCCESS  0
#define SD_ERROR_CANNOT_GO_TO_IDLE_STATE 1
#define SD_ERROR_CANNOT_DETECT_TYPE 2
#define SD_ERROR_TIMEOUT 3
#define SD_ERROR_CANNOT_GET_OCR 4
#define SD_ERROR_CANNOT_SET_BLOCK_SIZE 5
#define SD_ERROR_CANNOT_READ 6
uint8 sd_error;
uint8 sd_type;
uint8 sd_result[5];
void sd_disable() { spi_transfer(0xFF); spi_disable(); spi_transfer(0xFF); }
void sd_enable() { spi_transfer(0xFF); spi_enable(0); spi_transfer(0xFF); }
void sd_command(uint8 cmd, uint32 arg, uint8 crc) {
    spi_transfer(cmd|0x40);
    spi_transfer(arg >> 24);
    spi_transfer((arg >> 16) & 0xFF);
    spi_transfer((arg >> 8) & 0xFF);
    spi_transfer(arg & 0xFF);
    spi_transfer(crc|0x01);
}
void sd_powerUpSeq() {
    sd_disable();
    util_sleepMillis(1);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
```

```
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
}
uint8 sd_readRes1() {
    uint8 i = 0, res1;
    while((res1 = spi_transfer(0xFF)) == 0xFF) {
        i++;
        if (i > 8) break;
    }
    return res1;
}
uint8 sd_readUntil(uint8 expected, uint16 timeout) {
  uint8 res1;
  uint16 t1 = util_msec_LSB;
  while((res1 = spi_transfer(0xFF)) != expected) {
    if ((util_msec_LSB-t1)>timeout) return 0xFF;
  }
  return res1;
}
uint8 sd_goIdleState() {
    sd_enable();
    sd_command(0, 0, 0x94);
    uint8 res1 = sd_readRes1();
    sd_disable();
    return res1;
}
void sd_readRes3_7(uint8 *res) {
    res[0] = sd_readRes1();
    if (res[0] > 1) return;
    res[1] = spi_transfer(0xFF);
    res[2] = spi_transfer(0xFF);
    res[3] = spi_transfer(0xFF);
    res[4] = spi_transfer(0xFF);
}
void sd_sendIfCond(uint8 *res) {
    sd_enable();
    sd_command(8, 0x0000001AA, 0x86);
    sd_readRes3_7(res);
    sd_disable();
}
void sd_readOCR(uint8 *res) {
    sd_enable();
    sd_command(58, 0, 0);
    sd_readRes3_7(res);
    sd_disable();
}
```

```
uint8 sd_sendApp() {
    sd_enable();
    sd_command(55, 0, 0);
    uint8 res1 = sd_readRes1();
    sd_disable();
    return res1;
}
uint8 sd_sendOpCond() {
    sd_enable();
    sd_command(41, (sd_type==2?0x40000000:0), 0);
    uint8_t res1 = sd_readRes1();
    sd_disable();
    return res1;
}
uint8 sd_sendBlockSize() {
  sd_enable();
  sd_command(16, 512, 0);
  uint8_t res1 = sd_readRes1();
  sd_disable();
  return res1;
}
uint8 sd_init() {
    sd_type = 0;
    uint8 cmdAttempts = 0;
    sd_powerUpSeq();
    while((sd_result[0] = sd_goIdleState()) != 0x01) {
        cmdAttempts++;
        if (cmdAttempts > 10) {
          sd_error = sd_result[0];
          return SD_ERROR_CANNOT_GO_TO_IDLE_STATE;
        }
    }
    sd_sendIfCond(sd_result);
    sd_type = (result[0] & 4 == 0x04)?1:2;
    if (sd_type == 2 && sd_result[4] != 0xAA) {
        sd_error = sd_result[0];
        return SD_ERROR_CANNOT_DETECT_TYPE;
    }     cmdAttempts = 0;
    do {
        if (cmdAttempts > 100) return SD_ERROR_TIMEOUT;
        sd_result[0] = sd_sendApp();
        if(sd_result[0] < 2) {
            sd_result[0] = sd_sendOpCond();
        }
        util_sleepMillis(10);
        cmdAttempts++;
    } while(sd_result[0] != 0);
    sd_readOCR(sd_result);
    if (!(sd_result[1] & 0x80)) return SD_ERROR_CANNOT_GET_OCR;
    sd_type = (sd_result[1] & 0xC0 == 0xC0)?3:sd_type;
    uint8 res = sd_sendBlockSize();
```

```
    if (res != 0) return SD_ERROR_CANNOT_SET_BLOCK_SIZE; else return SD_SUCCESS;
}
uint8 sd_read(uint32 blockNumber, uint16 *destination) {
  sd_enable();
  if (sd_type != 3) blockNumber <<= 9;
  sd_command(17, blockNumber, 0);
  uint8 res1 = sd_readUntil(0xFE, 100);
  if (res1 != 0xFE) return SD_ERROR_CANNOT_READ;
  for(uint16 i = 0; i < 256; i++) {
    *destination = spi_transfer(0xFF) | (spi_transfer(0xFF) << 8);
    destination++;
  }
  // skip CRC
  spi_transfer(0xFF);
  spi_transfer(0xFF);
  sd_disable();
  return SD_SUCCESS;
}
```

The following subroutines are available (only the publicly availabe are shown here):

- uint8 sd_init(): this will detect and initialise the card. If anything goes wrong, an error code is returned. The error codes differ from the ones generated by the ROM and can be seen in the C-file above. This routine should be called after an SD card is inserted.
- uint8 sd_read(uint32 blockNumber, uint16* destination, bool io): will read the specified block (512 bytes) into memory to the specified location. If the Boolean 'io' is true, the data is written to i/o memory in stead of normal RAM. The destination should point to an array containing 256 16-bit words. The words are read LSB first. If an error occurs the code is returned. On success the return value is equal to 0.

Chapter 9

FAT

The FAT library ('fat.asm') contains functionality to interact with a FAT16 file system, located on the first partition of an SD card. This partition must be strictly smaller than 32MB.

The code is derived from the following C-code:

```
#define FAT_OK                 0
#define FAT_ERROR_NOT_FORMATTED       1
#define FAT_ERROR_PARTITION_TOO_BIG      2
#define FAT_ERROR_INVALID_SECTOR_SIZE    3
#define FAT_ERROR_INVALID_NUMBER_OF_FATS 4
#define FAT_ERROR_FILE_NOT_FOUND         5
#define FAT_END_OF_FILE                  6
uint32 fat_startPartition;
uint32 fat_lengthPartition;
uint16 fat_fatPosition;
uint32 fat_rootFolderPosition;
uint16 fat_numberOfSectorsForRootFolder;
uint16 fat_fileStart;
uint32 fat_fileLength;
uint16 fat_currentClusterNumber;
uint8 fat_sectorsPerCluster;
uint8 fat_currentSectorNumber;
uint16 fat_sectorsPerFat;
uint16 fat_sector[256];
uint8 fat_findAndCheckPartition() {
    fat_startPartition = 0;
    fat_lengthPartition = 0;
    sd_read(0, fat_sector); // read MBR
    uint16 signature = fat_sector[255];
    if (signature != 0xAA55) return FAT_ERROR_NOT_FORMATTED;
    fat_startPartition = fat_sector[0xE3] + fat_sector[0xE4] << 16;
    fat_lengthPartition = fat_sector[0xE5] + fat_sector[0xE6] << 16;
    if (fat_lengthPartition >= 65536) return FAT_ERROR_PARTITION_TOO_BIG;
    return FAT_OK;
}
uint8 fat_findAndCheckFAT() {
    fat_fatPosition = 0;
    fat_rootFolderPosition = 0;
```

```
    fat_numberOfSectorsForRootFolder = 0;
    sd_read(fat_startPartition, fat_sector); // VBR
    uint16 bytesPerSector = (fat_sector[5] >> 8)
                          | (fat_sector[6] & 0xFF) << 8;
    if (bytesPerSector != 512) return FAT_ERROR_INVALID_SECTOR_SIZE;
    uint8 numberOfFATs = fat_sector[8] & 0xFF;
    if (numberOfFATs != 2) return FAT_ERROR_INVALID_NUMBER_OF_FATS;
    fat_sectorsPerFat = fat_sector[11];
    fat_fatPosition = fat_sector[7];
    fat_rootFolderPosition = fat_startPartition + fat_fatPosition
                           + 2 * fat_sectorsPerFat;
    uint16 maxNumberOfRootDirEntries = (fat_sector[8] >> 8)
                                     | (fat_sector[9] & 0xFF) << 8;
    fat_numberOfSectorsForRootFolder = maxNumberOfRootDirEntries / 16;
    fat_sectorsPerCluster = fat_sector[6] >> 8;
    return FAT_OK;
}
uint8 fat_openFile(char filename[11]) {
    fat_fileStart = 0;
    fat_fileLength = 0;
    fat_currentSectorNumber = 0;
    fat_currentClusterNumber = 0xFFFF; // end of file
    uint16 i = 0;
    while (i < fat_numberOfSectorsForRootFolder) {
        sd_read(rootFolderPosition + i, fat_sector); // part of root folder
        uint8 entry = 0;
        while (entry < 16) {
            uint16 position = entry * 16;
            char* name = fat_sector + position;
            if (strcmp(name, filename, 11)==0) {
                fat_currentClusterNumber = fat_fileStart
                                         = fat_sector[position + 13];
                fat_fileLength = fat_sector[position + 14]
                               + fat_sector[position + 15] << 16;
                return FAT_OK;
            }
            entry++;
        }
        i++;
    }
    return FAT_ERROR_FILE_NOT_FOUND;
}
uint8 fat_readNextFileSector(uint16 buffer[256]) {
    if (fat_currentClusterNumber >= 0xFFF8) return FAT_END_OF_FILE;
    sd_read(fat_rootFolderPosition + fat_numberOfSectorsForRootFolder
            + (fat_currentClusterNumber-2)*fat_sectorsPerCluster
            + fat_currentSectorNumber, buffer);
            // weird thing: FAT cluster numbers start at 2...
    fat_currentSectorNumber++;
    if (fat_currentSectorNumber >= fat_sectorsPerCluster) {
        fat_currentSectorNumber = 0;
```

```
            uint16 fatSectorNumber = fat_currentClusterNumber >> 8;
            uint16 fatOffset = fat_currentClusterNumber & 0xFF;
            sd_read(fat_startPartition + fat_fatPosition +
                    fat_fatSectorNumber, fat_sector);
            fat_currentClusterNumber = fat_sector[fatOffset];
        }
        return FAT_OK;
    }
```

The following (public) subroutines are available:

- uint8 fat_init(): will find and check the first partition and will localise the FAT16 file system. It returns FAT_OK if everything is ok, otherwise an error code is returned (see code above)
- uint8 fat_openFile(char filename[11]): will try to find the file with the given filename (11 characters are expected). Each two characters are expected to be in a 16-bit word in memory (LSB first). If less than 11 characters are present, spaces should be added so the length is always 11. The string must not be null-terminated. If the file was not found, an error code is returned. On a FAT-16 filesystem all filenames are in capitals.
- uint8 fat_readNextFileSector(uint16 buffer[256], boolean io): will read the next file sector (512 bytes, 256 words) into the given location (the boolean indicates if the buffer is located in i/o space). If there are more sectors available, FAT_OK is returned. A return value FAT_END_OF_FILE means the end of the file has been reached (the last sector has just been read).

Chapter 10

Integer math

The integer math library ('math.asm') contains functionality to do basic calculations with integer values. The following functions are available:

- int16 math_multiply8signed(int8 a, int8 b): multiplies two signed 8-bit integers. The result is put in r0 (16 bit). The operation needs 31 clock cycles. Note: the current implementation is not correct: it will just do an unsigned multiplication. But as int8 will probably be discarded in the future, this may not be a big issue.
- uint16 math_multiply8unsigned(uint8 a, uint8 b): multiplies two unsigned 8-bit integers. The result is an uint16 and is put in r0 (16 bit). The operation needs 31 clock cycles.
- int16 math_multiply16signed(int16 a, int16 b): multiplies two signed 16-bit integers. The result is put in r0. The operation needs 73 clock cycles.
- uint16 math_multiply16unsigned(uint16 a, uint16 b): multiplies two unsigned 16-bit integers. The result is put in r0. The operation needs 73 clock cycles.
- int32 math_multiply32signed(int32 a, int32 b): multiplies two signed 32-bit integers. The result is put in r0 (LSB) and r1 (MSB). The operation needs 426 clock cycles.
- uint32 math_multiply32unsigned(uint32 a, uint32 b): multiplies two unsigned 32-bit integers. The result is put in r0 (LSB) and r1 (MSB). The operation needs 426 clock cycles.
- int32 math_divideByTen16signed(int16 n): divides a signed 16-bit integer by 10. The result is put in r0 (LSB) and the rest (modulo) in r1 (MSB). The operation needs 36 clock cycles.
- uint32 math_divideByTen16unsigned(uint16 n): divides an unsigned 16-bit integer by 10. The result is put in r0 (LSB) and the rest (modulo) in r1 (MSB). The operation needs 29 clock cycles.
- uint32 math_divide8unsigned(uint8 a, uint8 b): divides two unsigned 8-bit integers. The result is put in r0 (LSB) and the remainder (modulo) in r1 (MSB). The operation uses 236 clock cycles. This implementation uses the next function since int8 and uint8 will be deprecated.
- uint32 math_divide16unsigned(uint16 a, uint16 b): divides two unsigned 16-bit integers. The result is put in r0 (LSB) and the remainder (modulo) in r1 (MSB). The operation uses 236 clock cycles.

- uint32 math_divide8signed(uint8 a, uint8 b): divides two signed 8-bit integers. The result is put in r0 (LSB) and the remainder (modulo) in r1 (MSB). The operation uses 264 clock cycles. This implementation uses the next function since int8 and uint8 will be deprecated.
- uint32 math_divide16signed(uint16 a, uint16 b): divides two signed 16-bit integers. The result is put in r0 (LSB) and the remainder (modulo) in r1 (MSB). The operation uses 264 clock cycles.
- uint32 math_divide32unsigned(uint32 a, uint32 b): divides two unsigned 32-bit integers. The result is put in r0/r1. The remainder is put in r2/r3. The operation uses 656 clock cycles.
- uint32 math_divide32signed(int32 a, int32 b): divides two signed 32-bit integers. The result is put in r0/r1. The remainder is put in r2/r3.The operation uses 702 clock cycles.

Chapter 11

Floating point math

The floating point math library ('float.asm') contains functionality to do many different calculations with IEEE 754 float32 floating point numbers.. The following functions are available:

- float32 float_itof(int32 i)
- void float_print(float32 f)
- float32 float_add(float32 a, float32 b)
- float32 float_sub(float32 a, float32 b)
- float32 float_mult(float32 a, float32 b)
- float32 float_div(float32 a, float32 b)

More complex operations with floating point values are also implemented, using the CORDIC algorithm. All these functions use the following general function:

- void cordic(): this performs cordic_precision steps of the algorithm. All calculations are done with 32-bit fixed-point numbers (30 binary digits behind the dot)

This function doesn't have parameters but uses the following global variables instead:

- cordic_mu : int16: this defines the curve to be used (1 = circular, 0 = linear, -1 = hyperbolic)
- cordic_mode : uint16: this defines the mode (0 = rotating, 1 = vectoring)
- cordic_alpha : pointer to array of int32: this is a pointer to the array of pre-calculated values
- cordic_precision : uint16 = 30: this is the number of decimal digits
- cordic_K : uint32 = 0x3F1B74EE // 0.6072529350088814
- cordic_K_hyp : uint32 = 0x3F9A8F44 // 1.207497067763
- cordic_x : int32
- cordic_y : int32
- cordic_z : int32

In circular mode the values for alpha are:

{ 843314856 497837829 263043836 133525158 67021686 33543515 16775850 8388437 4194282 2097149 1048575 524287 262143 131071 65535 32767 16383 8191 4095 2047 1023 511 255 127 63 31 15 8 4 2 1 0 }

In linear mode the values are:

{ 1073741824 536870912 268435456 134217728 67108864 33554432 16777216 8388608 4194304 2097152 1048576 524288 262144 131072 65536 32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1 0 }

In hyperbolic mode the values are:

{0 589812981 274247418 134923406 67196450 33565361 16778581 8388778 4194325 2097154 1048576 524288 262144 131072 65536 32768 16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1 0 }

The implemented algorithm is based on the following Moncky code:

```
fun cordic()
i : uint16
d : int16
oldX : int32
again : uint16
{
  let again = 0
  let i = (cordic_mu & 1)
  while (i <= cordic_precision) {
    let again = ((cordic_mu == -1) & ((~ again) & ((i == 4) | (i == 13))))
    if (cordic_mode == CORDIC_MODE_ROTATION) {
      let d = (?(cordic_z < 0) -1 1)
    } else {
      let d = (?((cordic_x<0)^(cordic_y<0)) 1 -1)
    }
    let oldX = cordic_x
    if (cordic_mu != 0) {
      if ((cordic_mu ^ d) < 0) {
        let cordic_x = (cordic_x + (cordic_y >>> i))
      } else {
        let cordic_x = (cordic_x - (cordic_y >>> i))
      }
    }
    if (d < 0) {
      let cordic_y = (cordic_y - (oldX >>> i))
      let cordic_z = (cordic_z + (^cordic_alpha)[i])
    } else {
      let cordic_y = (cordic_y + (oldX >>> i))
      let cordic_z = (cordic_z - (^cordic_alpha)[i])
    }
    let i = (? again (i + 1) i)
  }
}
```

Based on the cordic algorithm, the following functions are available:

- float32 float_sin(float32 radians)
- float32 float_cos(float32 radians)
- float32 float_tan(float32 radians)
- float32 float_atan(float32 radians)
- float32 float_sqrt(float32 radians)
- float32 float_exp(float32 radians)
- float32 float_ln(float32 radians)