MCGILL UNIVERSITY

Java Virtual Machine

ECSE 426: Microprocessor Systems

GROUP # 3

4/17/2008

Eugene Savchenko 260184770
Ionut Georgian Ciobanu 260242300
Stanislav Soukhanov 260130916
David Grigorian 260171569
Saroj Gidda 260129887

Table of Contents

INTRODUCTION

The final Microprocessor project consists of the design and implementation of a virtual machine for the IJVM, a subset of the Java Virtual Machine (JVM). Our main hardware consists of one or more McGump's boards and a conventional personal computer. The objective is to create a flexible parallel system which can be used for any IJVM application. Some of the features of the implemented Java Virtual Machine are:

- Additional instructions were implemented to the virtual machine using special parameters passed to the IN and OUT functions in order to both increase its flexibility and to accelerate computationally intensive operations by performing them in hardware on a CPLD.
- The addition of a second board through the secondary McGump's serial port header allowed two (or more) IJVM virtual machines to execute IJVM code in parallel, and accelerate the application further.
- A menu -based loader program should be created to allow a user to upload, execute, and view the result s of IJVM program execution over the serial port.
- Application inputs will be taken from the keypad, and outputs shown thro ugh the terminal.

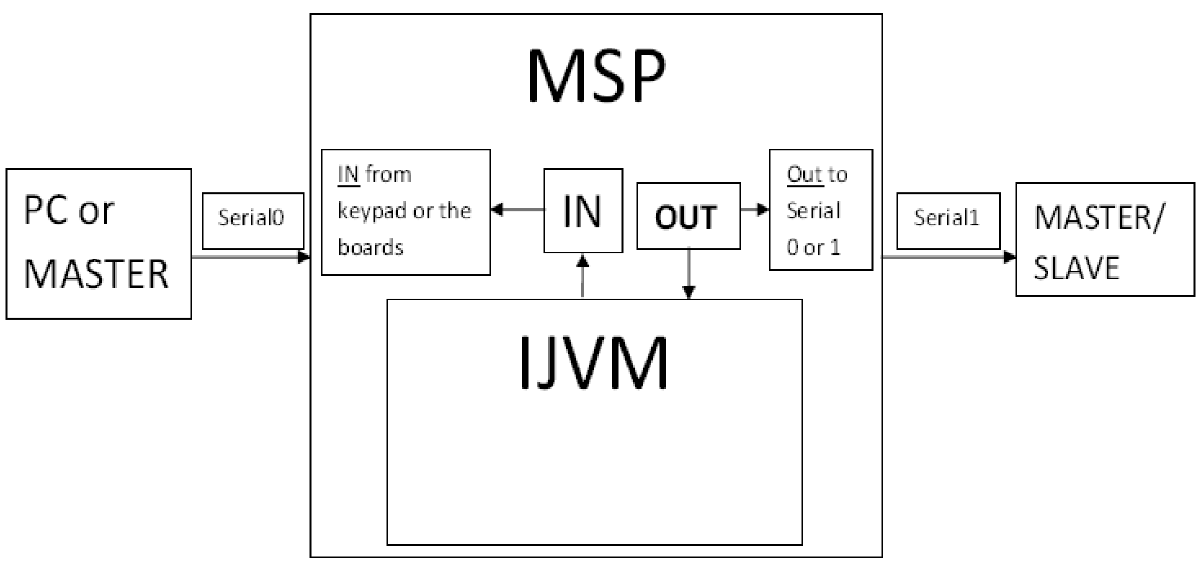

Figure 1: Subsystem connections in the main system

This report covers all aspects of the design in a comprehensive and technical fashion. We have organized the report as asked by the requirements from the previous labs, with 2 sub-classifications for each section:

Functional Specifications: This is essentially aimed at giving the user a concise but accurate idea of the basic working of the system. There are minimum details present in this section; the contents comprise of the list of functions as written in the code itself.
The user will be able to know exactly how what these functions do without having to understand any implementation details.
Implementation: The details of the functions that were left out in the previous section are detailed here as necessary. Technicalities are covered, and a general discussion of the solution is given. This includes the hardware that was used, any software design decisions that were made (flags, etc.), relevant parameter configuration, and the handling of interrupts. The flow of the program is implicitly laid in every explanation presented.

Diagrams have been added where deemed necessary; their purpose is to ease the understanding of the processes and provide a visual representation of modules. A conclusion wraps up the document by summarizing important results and the *performance analysis* of our system as a whole, providing discussion on any relevant topics that were only briefed during the course of this report.

COMPILER

The compiler function (compiler.c) is a multi-pass compiler that processes the ijvm code of the program several times. Each pass parses the input (source code) file, and writes the correct bytecode directly in the compiled file Any step can set an error flag if needed due to a syntax error. After each step, the global error flag is checked. If set, the compilation process is stopped and, if in debug mode, an appropriate error message is sent generated.

Functional Specifications:

Function: `int ijvmcompilerRun(void)`
Purpose: opens the file through the function from filesys.c for further I/O calls.
If valid: parses them
Inputs: none
Returns: 1 if success, or 0 if error

Function: `int WriteExecutable(unsigned short addr, unsigned char data)`
Purpose: function that allows us to randomly write in FLASH, the compiler file
Inputs: addr, data
Returns: 1 for success, 0 for write error

Function: `int BuildMethodTable(struct Symbol table[MAX_CONSTANTS])`
Purpose: Build a table of all the methods and stores the address and the name of each method
Inputs: takes in table with the variable parameter of Max constants
Returns: size of table, sets error flag if needed

Function: `int BuildLabelTable(struct Symbol table[MAX_CONSTANTS])`
Purpose: Builds a table of all the labels and stores the address and name of each label
As soon as we reach the end, replace the jumps with the offset label. This helps to check syntax.
Inputs: Takes in the table with a variable parameter of max constants.
Returns: returns 1

Function: `int BuildConstantTable(struct Symbol table[MAX_CONSTANTS])`
Purpose: Builds a table of all the contants and stores the address and name of each constant
Inputs: takes in the table with a variable parameter of max constants
Returns: size

Function: `int ReplaceConstants(struct Symbol table[MAX_CONSTANTS], char tableSize)`
Purpose: Replaces constants in the compiled code with their actual values from the constants table.

Inputs: table, tableSize
Returns: size of table, sets error flag if needed

Function: `int ReplaceMethods(struct Symbol table[MAX_CONSTANTS], char tableSize)`
Purpose: Replaces method calls with the absolute address of the method – taken from the method table.
Inputs: table, tableSize
Returns: 1 for success, 0 if an error has occured

Function: `int ReplaceLabels(struct Symbol table[MAX_CONSTANTS], char tableSize)`
Purpose: Replaces jumps with the offset (from the jump) of the target label - taken from the label table.
Inputs: table, tableSize
Returns: 1 for success, 0 if an error has occured

Function: `int BuildVariableTable(struct Symbol table[MAX_CONSTANTS])`
Purpose: Builds the table of variables and stores the address and name of each variable. As soon as we reach the end, replace the jumps with the offset label. This helps to check syntax.
Inputs: table
Returns: size of table, sets error flag if needed

Function: `int ReplaceVariables(struct Symbol table[MAX_CONSTANTS], char tableSize)`
Purpose:
Inputs: table, tableSize
Returns: 1 for success, 0 if an error has occured

Function: `int parseConstants(void), int parseMethods(void), int parseLabels(void), int parseVariables(void)`
Purpose: Executes above-described functions.
Inputs: none
Returns: 1 for success, 0 if an error has occured

Function: `int parseTokens(char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE], int tokenCount, int *codeIndex, char *isLabel, int *offset)`
Purpose: Parses a series of string tokens and generates offset information in the compiled code; performs primary syntax check. Sets the error flag if needed.
Inputs: Array of strings to be parsed, number of strings to be parsed.
Returns: Code information – opcode, offset in compiled code, whether there is a label or not.

Function: `int value(char *token)`
Purpose: Parses the **byte** parameter given to IINC or BIPUSH. Sets the error flag if needed (syntax check).

Inputs: String token to be parsed
Returns: Value of token as defined in specifications (converts from binary, hexa or ASCII to decimal) as needed.

Function: `int searchSymbol(char symbol[MAX_TOKEN_SIZE], struct Symbol *table, int symbolCount)`
Purpose: Find a symbol in the symbol table
Inputs: Symbol to find, symbol table, size of table
Returns: position, -1 if not found

Function: `int AddSymbol(char* name, int address, struct Symbol *table, int *tableIndex)`
Purpose: Add a new symbol to the symbol table
Inputs: Symbol to add, table, table size
Returns: 1 for success, 0 if an error has occured

Implementation:

Given the solution architecture, a multi-pass compiler was the most elegant way to cope with our limitations of available RAM memory. Thus, we had to trade speed (we perform more operations) for space. It makes 4 passes: for labels, methods, constants, and variables. On each pass it replaces symbols with their value or offset in memory: constants are replaced with their value, variables are replaced with their offset from the current frame pointer, methods are replaced with their relative address in memory, the number of local variables and parameters passed is added at the method location to ease the work at run-time, and jumps to labels (conditional and unconditional jumps) are replaced with an offset from the current position. Thus, the offset can be either positive or negative.

Syntax Checking

There is syntax checking at 4 levels:
1. The *Token Parser* reports errors if any given line of input is not conforming to the language syntax
2. The Value function returns an error if a value fed to an instruction is malformed
3. Each pass checks the existence of the symbols it tries to retrieve/add to the symbol table and generates two possible errors: symbol not defined/symbol already exists
4. Scope checks: the compiler checks that no variables/constants are defined outside the .var/.const section, methods are correctly formed, etc

Limits

The compiler is limited in the number of constants per program, methods per program, variables per method (main is considered a method), and labels per method. Using the MAX_CONSTANTS and MAX_METHODS, these can be easily changed depending on the size of RAM available.
The maximum size of a string token (variable name, method name, constant name, label name, keyword) is defined by MAX_TOKEN_SIZE. This can also be adjusted according to the size of RAM available

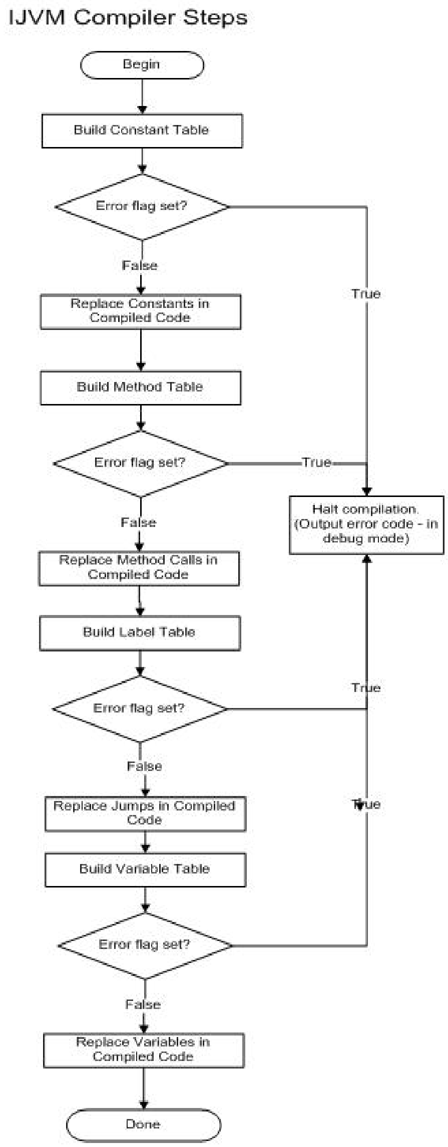

Figure 2: IJVM Compiler

UART Module:

In asynchronous mode, the USART connects the MSP430 to an external system via two external pins, URXD and UTXD. UART mode is selected when the SYNC bit is cleared.

UART mode features independent transmit and receive shift registers, separate transmit and receive buffer registers, programmable baud rate with modulation for fractional baud rate support, status flags for error detection and suppression and address detection and independent interrupt capability for receive and transmit.

Our JVM uses the UART0 buffered interrupt-driven reception and transmission. Current UART settings are fixed to 19200-N-8, but can easily made reconfigurable. Figure ## shows the control and status registers of USART0.

Table 13−3. USART0 Control and Status Registers

Register	Short Form	Register Type	Address	Initial State
USART control register	U0CTL	Read/write	070h	001h with PUC
Transmit control register	U0TCTL	Read/write	071h	001h with PUC
Receive control register	U0RCTL	Read/write	072h	000h with PUC
Modulation control register	U0MCTL	Read/write	073h	Unchanged
Baud rate control register 0	U0BR0	Read/write	074h	Unchanged
Baud rate control register 1	U0BR1	Read/write	075h	Unchanged
Receive buffer register	U0RXBUF	Read	076h	Unchanged
Transmit buffer register	U0TXBUF	Read/write	077h	Unchanged
SFR module enable register 1†	ME1	Read/write	004h	000h with PUC
SFR interrupt enable register 1†	IE1	Read/write	000h	000h with PUC
SFR interrupt flag register 1†	IFG1	Read/write	002h	082h with PUC

† Does not apply to '12xx devices. Refer to the register definitions for registers and bit positions for these devices.

Figure 3: UART controls

Functional Specifications:

Function: `int serialModuleInit()`
Purpose: this function initializes the serial module. It must be called before any of the module functions are called. It initializes the UART registers for the communication parameters, sets the UART clock, and initializes the GPIO pins used as RX and TX.
Inputs:
Returns:

Function: `int serialWrite()`
Purpose: this function writes the charecter to the transmit queue and // forces the transmit interrupt to call ISR.
Inputs: SerialPort port {COM_1, COM_2}, char c
Returns: int - returns 1 if when successful, returns 0 if not.

Function: `int serialWriteBuffered()`
Purpose: This function writes the character to the transmit queue, but forces the transmit interrupt to call ISR _ONLY_ when the queue is full (unlike serialWrite). To make sure that the data is completely transferred, the transmit buffer needs to be flushed with serialFlushWriteBuffer when finished.
Inputs: SerialPort port {COM_1, COM_2}, char c
Returns: int - returns 1 if when successful, returns 0 if not.

Function: `int serialWriteString()`
Purpose: this function copies the charecter string to the transmit queue and forces the transmit interrupt to call ISR.
Inputs: SerialPort port - serial port, char buffer[] - character buffer to transmit, unsigned int bufferSize - the size of buffer to transmit
Returns: int - returns 1 if when successful, returns 0 if not.

IJVM Input/Output

This module is the proxy that controls access to the hardware IO functions: serial ports and keypad. It works by directing IO streams to the current IO consumer: either the virtual machine-level modules (like the console) or the ijvm-application level (IJVM applications running inside the virtual machine).

Functional Specifications:

Function: `int ijvmioModuleInit(void)`
Purpose: initialize the module before using any of its functions
Inputs: none
Returns: return: 1 on success, 0 on failure

Function: `int ijvmioProcessFsm(void)`
Purpose: process the ijvmio proxy module. Needs to be called as often as possible. Currently, it only processes the keypad module.
Inputs: none
Returns: 1 on success, 0 on failure

Function: `int ijvmioSetIOConsumer( IO Consumer Cons)`
Purpose: set the current IO consumer that will have access to the IO facilities.
Inputs: IO consumer Cons
Returns: 1 on success, 0 on failure

Function: `ijvmioSerialWrite, int ijvmioSerialWriteBuffered, int ijvmioSerialRead, int ijvmioSerialReadString, int` `ijvmioSerialWriteString, int ijvmioSerialFlushReadBuffer, int ijvmioSerialFlushWriteBuffer`

Purpose: Proxy functions for serial access

Inputs: refer to UART module

Returns: refer to UART module

Function: `KeyType ijvmioKeypadRead(IOConsumer cons)`

Purpose: Proxy function for keypad access

Inputs: IOConsumer cons, refer to keypad module

Returns: refer to keypad module

Flash Memory

Flash self-programmability is becoming increasingly important. However, when accessing a flash-memory array for an erase/program operation, the CPU cannot simultaneously execute the code in the flash array. Thus, a microcontroller with only a single on-chip flash cannot execute code and modify its flash-memory contents at the same time. The problem can be solved in two ways:
(1) Instructions to erase/program flash memory are copied into RAM for execution by the CPU, and
(2) The CPU is sent into an idle state while the flash memory erase/program process is being completed.

Flash memory is organized in the MSP430 in the following fashion (please see Figure 4):

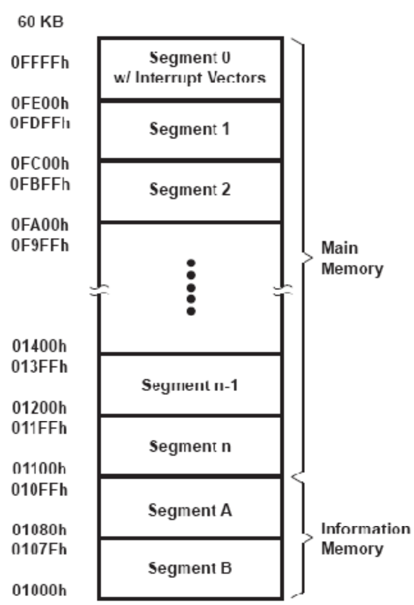

Figure 4: Flash Memory Organization in the MSP430149

The memory is divided into Main Memory and Information Memory; the former contains 512 byte segments, totaling to 60 KB. The latter has two segments ('A' and 'B') of 128 bytes each, thus totaling 256 B. Our info. is stored from address 01400h onwards (towards Segment 1).

Functional Specifications:

Function:	`int flashModuleInit(void)`
Purpose:	initialize the module before calling any of its functions
Inputs:	none
Returns:	1on success, 0 on failure

Function:	`int flashWriteByte(unsigned short addr, unsigned char data)`
Purpose:	fast-write of a single byte at the raw flash address. The target location must be erased beforehand.
Inputs:	unsigned short addr - target raw flash address, unsigned char data - the byte to be written
Returns:	1 on success, 0 on failure

Function:	`int flashReadSegment(unsigned short addr, char buffer[], unsigned int length)`
Purpose:	read from the flash segment into a memory buffer. (NOTE: this is done for uniform interface; the read from flash is actually equivalent to reading from memory). The extra overhead is justified for maintainability and interface cleanliness purposes.
Inputs:	unsigned short addr - source raw flash segment start address , char buffer[] - target RAM buffer , unsigned int length - number of bytes to copy from flash to the buffer
Returns:	1 on success, 0 on failure

Function:	`int flashWriteSegment(unsigned short addr, char buffer[], unsigned int length)`
Purpose:	read into the flash segment from the memory buffer. The target segment must be erased beforehand by calling flashEraseSegment.
Inputs:	unsigned short addr - target raw flash segment start address , char buffer[] - source RAM buffer , unsigned int length - number of bytes to copy from the buffer to flash
Returns:	1 on success, 0 on failure

Function:	`int flashEraseSegment(unsigned short addr)`
Purpose:	Erase the segment starting at 'addr'
Inputs:	unsigned short addr - flash segment-to-be-erased start address
Returns:	1 on success, 0 on failure

Implementation:

The flow of the flash processes is very straightforward. Essentially, everything operates through the controlled use of pointers and flags. A flash pointer is simply a pointer of type int (16 bits) or char (8 bits) that points to an address in flash memory. It is important to note that, at the beginning of any flash operation, the first step is to disable interrupts and the watchdog. This is because reading or writing to flash memory while it is being programmed or erased is prohibited. At the end of the read/write/erase, interrupts and the watchdog are enabled again. The typical procedures have been expounded below individually for the convenience of the reader:

Flash Writing

Writing to flash may be done by byte, word, or by 'Block'. The block write can be used to accelerate the flash write process when many sequential bytes or words need to be programmed. However, a block write cannot be initiated from within flash memory. The block write must be initiated from RAM only. And anyhow, for our purposes, word writes are considerably fast, and therefore the extra improvement in speed is not really required. For these reasons, we implement byte and word writes only. The general steps are shown below.
_ Disable Interrupts and WatchDog
_ The 'LOCK' bit is first cleared.
_ The 'WRT' bit is then set. This effectively enables us to write information hereafter.
_ The write is made simply by initializing the data in the pointer to the data we wish to write (the data is of type int for a word write, and type char for a byte write).
_ The WRT bit is then cleared, and the 'LOCK' bit is set again.
_ Enable Interrupts and WatchDog

Flash Reading

Since the default mode is Read mode, no flags need to be set or unset during this activity.
The rest of the operation remains essentially the same.
_ Disable Interrupts and WatchDog
21
_ The read is made simply by initializing the data in the pointer to the data we wish
to read.
_ Enable Interrupts and WatchDog

Flash Erasing

There are three modes in which flash memory may be erased: Segment Erase, Mass Erase (all main memory segments), and Erase all Flash Memory (main and information segments). In our project, we only deal with Segment Erase, because we are only interested in erasing small portions of information at a time, and the smallest portion which may be erased is a segment. The following steps assume the Segment Erase mode is being used.
_ Disable Interrupts and WatchDog
_ The 'LOCK' bit is first cleared.
_ The 'ERASE' bit is then set. This effectively enables us to erase information hereafter.
_ The deletion is made simply by making a 'dummy write'; we initialize the data in the pointer to any data value (such as '0'). The contents of that address, and subsequently that entire segment in which the data was stored, is cleared.
_ The WRT bit is then cleared, and the 'LOCK' bit is set again.
_ Enable Interrupts and WatchDog

Writing Several Characters

Most of the times, we need to write/read many words at a time. For these purposes, the data is streamed directly from flash whilst doing so. The data is received via UART; therefore, every character received generates an interrupt. The interrupt causes the data to be written to flash, and the flash pointer is incremented at the end of the write. The next time the character comes in, it is written to the next address. In this fashion, long string of data is written quickly and effectively.

Console

This module implements the user interface to the IJVM - the serial console. When the virtual machine is not running, it presents a prompt on the master serial port, and processes all commands received at this port, like upload a new assembly source code, compile it, and start the execution of the IJVM etc. It also prints the error and status messages.

The user can choose to run the machine on either one board, or more than one board. Accordingly, the user must follow the following steps in order to execute the ijvm source code:

Scenario 1: Use of 1 board
Step 1: At the prompt, type ‘fmt’ , which is the command to format the file
Step 2: Then type in ‘enum’ for enumeration
Step 3: Then type out the command to upload the master ‘U0’
Step 4: Type ‘compile’, which starts the compilation of the code
Step 5: After successful compilation, type in start at the command prompt to start the execution of your ijvm source code.

Scenario 2: Use of 2 boards
Step 1: At the prompt, type ‘fmt’ to format the file.
Step 2: Then type in ‘enum’ for the enumeration process to begin.
Step 3: Then type in ‘U0’, the command to upload the master.
Step 4: Then type ‘U1’ to upload the slave.
Step 5: Type ‘compile’, which starts the compilation of the code
Step 6: After successful compilation, type in start at the command prompt to start the execution of your ijvm source code.
The user is also able to view the source code that has been downloaded on the master or the slave. By typing ‘D0’ or ‘D1’ at the command prompt.

Functional Specifications:

Function:	int consoleModuleInit(void)
Purpose:	initialize this module before using any of its functions. It initializes all related modules: file system, IO proxy, and timers
Inputs:	none

Returns: 1 on success, 0 on failure

Function: int consoleProcessFsm(void)
Purpose: this function does one processing pass of the console FSM. It is not blocking, and should be called as often as possible.
Inputs: none
Returns: 1 on success, 0 on failure

Function: static int beginChainEnumerate(ConsoleState processState)
Purpose: this function forwards the internal board enumerate request to further slave down the device chain, assigning to it the next board ID in the sequence.
Inputs: ConsoleState processState - the next FSM state to assign when done
Returns: 1 on success, 0 on failure

Implementation:

A detailed description of the console code is depicted in figure 5 .

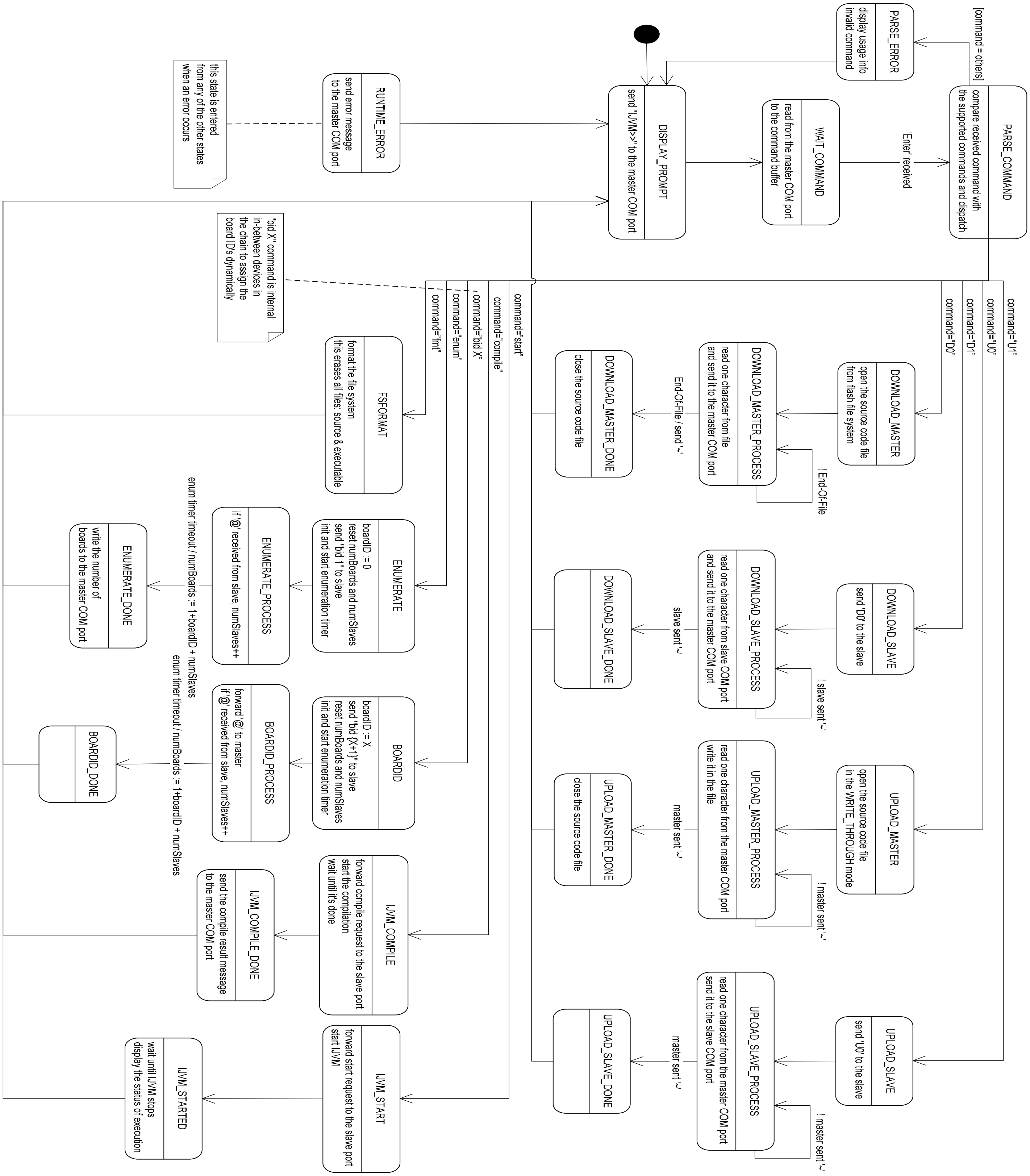

Figure 5: Console Implementation

IJVM Execution code

This is the ijvm processor module. It executes ijvm code precompiled by the ijvmcompiler module.

Functional Specifications:

Function: int ijvmModuleInit(void)
Purpose: initializes all required modules
Inputs: none Returns: 1 if success, 0 if not

Function: static int ijvmFetch(IjvmByte *data)
Purpose: fetches the byte from the executable file at the current program counter index and increments it
Inputs: IjvmByte *data - pointer to the fetched data
Returns: 1 if success, 0 if not

Function: static int ijvmLoad(void)
Purpose: initializes all VM registers for the new execution.
Inputs: none
Returns: 1 if success, 0 if not

Function: static int ijvmGetVar(IjvmVarIndex index, IjvmVarData *var)
Purpose: looks up a variable in the variables stack that belongs to the current frame and returns its value.
Inputs: IjvmVarIndex index - the index of the local variable within current frame
IjvmVarData *var - the pointer to hold the value
Returns: 1 if success, 0 if not

Function: static int ijvmSetVar(IjvmVarIndex index, IjvmVarData var)
Purpose: looks up a variable in the variables stack that belongs to the current frame and sets its value.
Inputs: IjvmVarIndex index - the index of the local variable within current frame
IjvmVarData var - new variable value
Returns: 1 if success, 0 if not

Function: static int ijvmPrepareNewFrameContext(void)
Purpose: initializes the new frame upon the new function invokation. At this point the program counter must point at the function prefix in the executable file, containing two bytes - the number of arguments, and the number of local variables. These are used to create the new frame record, and reserve space for the local variables on the variable stack. Also the call arguments are pushed off the main program stack and stored as local variables.
Inputs: none
Returns: 1 if success, 0 if not

Function: static int ijvmStackPush(IjvmStackData x)
Purpose: push a new data onto the IJVM data stack, and increment the stack pointer
Inputs: IjvmStackData x - a new value to be pushed
Returns: 1 if success, 0 if not

Function: static int ijvmStackPop(IjvmStackData *x)
Purpose: decrement the stack pointer and pop the top data from the IJVM data stack.
Inputs: IjvmStackData *x - pointer to hold the popped value
Returns: 1 if success, 0 if not

Function: IjvmStackData ijvmAluSqrt(IjvmStackData x)
Purpose: the MCU implementation of the square root. It is faster than the math.h sqrt implementation
Inputs: IjvmStackData x - the argument to the square root
Returns: IjvmStackData - the square root of x

Function: int ijvmExecuteOut(OutInstructionType type)
Purpose: this function executes the particular type of the OUT instruction. The argument(s) of the instruction should be stored on the data stack.
Inputs: OutInstructionType type - the subtype of the OUT instruction
Returns: 1 on success, 0 on failure

Function: int ijvmExecuteIn(InInstructionType type)
Purpose: this function executes the particular type of the IN instruction. The argument(s) of the instruction should be stored on the data stack.
Inputs: InInstructionType type - the subtype of the IN instruction
Returns: 1 on success, 0 on failure

Function: int ijvmProcessFsm(void)
Purpose: process the FSM of the virtual machine module. It's either stopped, or running and executing the IJVM compiled bytecode.
Inputs: none
Returns: 1 on success, 0 on failure

Function: int ijvmStartExecution(void)
Purpose: this function is called by external modules (console) to request the start of the execution. Usually, this means that a precompiled image has been created.
Inputs: none
Returns: 1 on success, 0 on failure

IMPLEMENTATION:

The implementation of the code, for the sake of simplicity is shown as a pictorial representation.

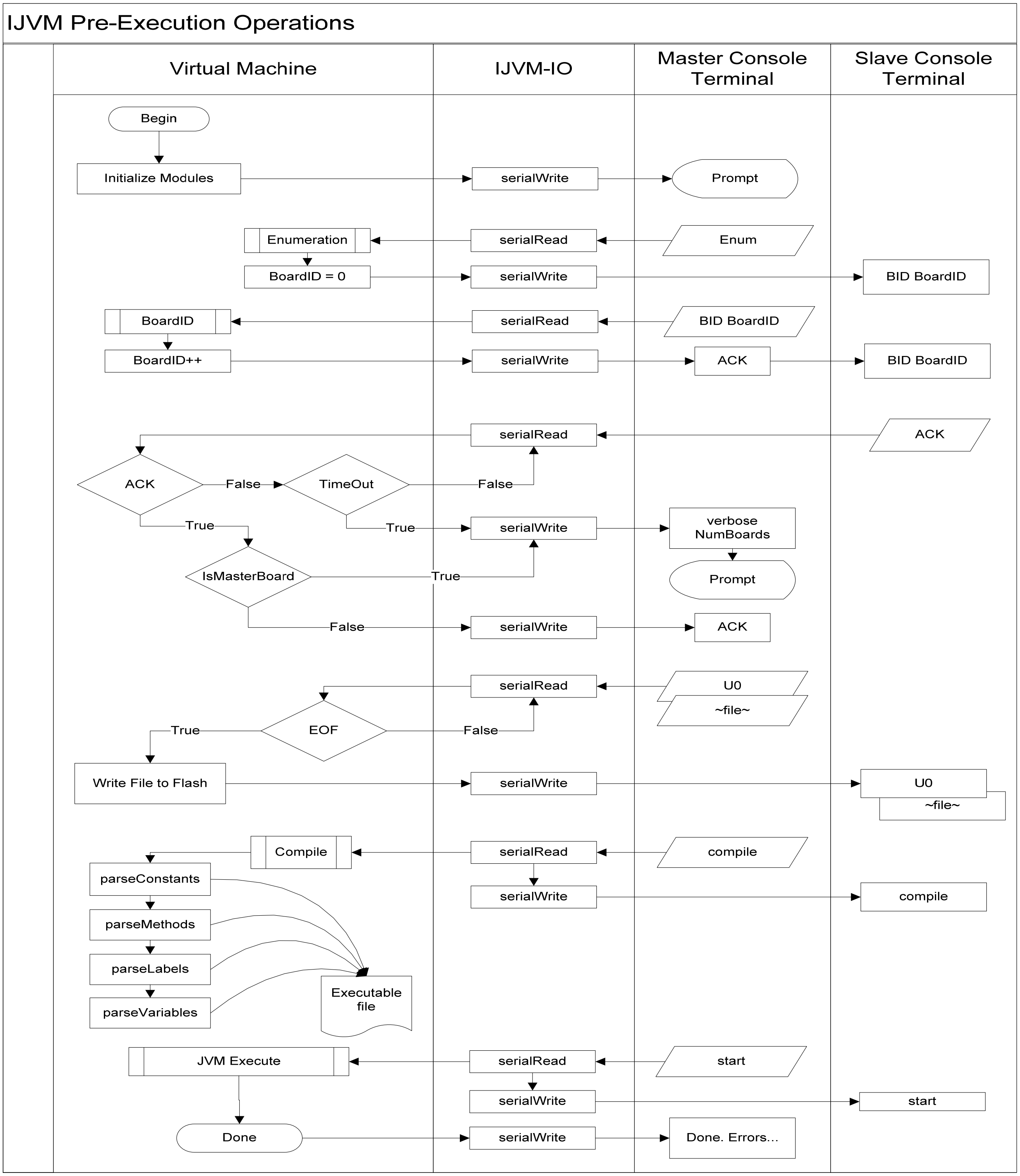

Figure 6: IJVM Pre-Execution Operations

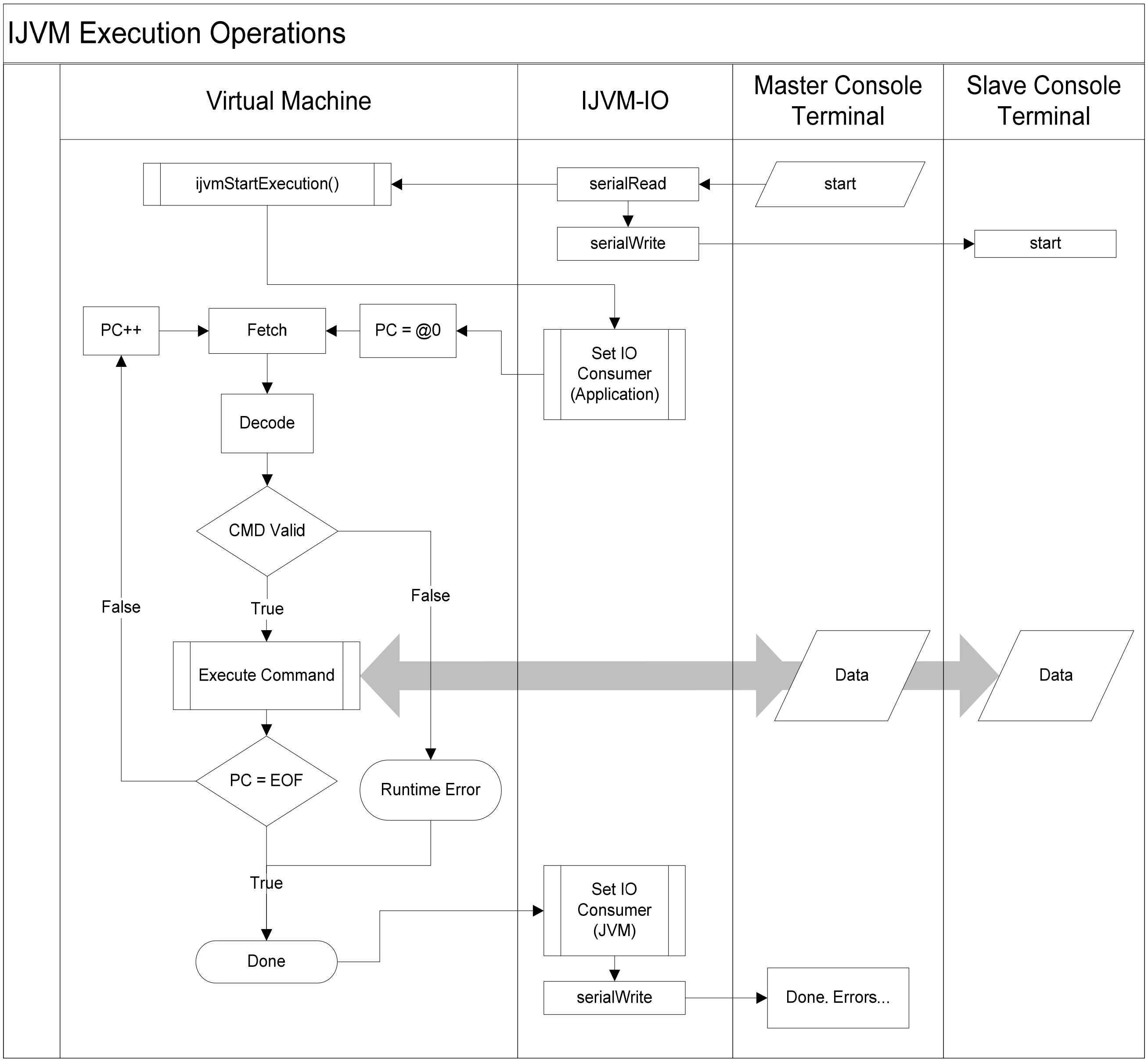

Figure 7: IJVM Execution Operation

File System:

This module provides the familiar file system interface to the underlying flash memory. Current implementation limits the number of files to two; the maximum sizes of the files, and their start and end addresses in flash must specified at compile time. This is transparent to the users of this module, though, who only need to specify the file index (equivalent to a file name) to open the file, and then use any of the available file IO functions.

Functional Specifications:

Function: unsigned int filesysFileMaxSize(unsigned int fileIndex)
Purpose: query the filesystem for the current size limits per file with ID=index.
Inputs: unsigned int fileIndex - file index
Returns: unsigned int - max size allowable for that file

Function: static int filesysFlushCurrentSegment(void)
Purpose: private function that reconciles RAM with flash by flushing the currently cached buffer to its associated segment.
Inputs: none
Returns: 1 if success, 0 if not

Function: static int filesysSetCurrentSegment(unsigned short addr)
Purpose: private function that caches the new flash segment into the RAM segment buffer
Inputs: unsigned short addr - the start address of the flash segment
Returns: 1 if success, 0 if not

Function: int filesysModuleInit(void)
Purpose: initialize current module before using any of its functions. This reads the flash to determine if there's a valid file system. if there is one, then the RAM filesys structures are initialized; if there is no valid file system, then a new one is created.
Inputs: none
Returns: 1 if success, 0 if not

Function: int filesysModuleDeinit(void)
Purpose: flushes the registry into flash to ensure consistency (just in case).
Inputs: none
Returns: 1 if success, 0 if not

Function: int filesysFormat(void)
Purpose: create the new file system by writing a fresh registry record into flash.
Inputs: none
Returns: 1 if success, 0 if not

Function: FileID filesysFileOpen(unsigned int fileIndex, FileOpenMode mode)
Purpose: open a file and obtain a handle to be used in further file I/O calls.
Inputs: unsigned int fileIndex - the file index within FS (equivalent to a filename)
FileOpenMode mode - the open mode (see filesys.h)
Returns: FILE_ID_INVALID if failure, otherwise a valid file handle FileID

Function: int filesysFileClose(FileID id)
Purpose: close a file and flush all the pending caches to flash
Inputs: FileID id - the file handle obtained from filesysFileOpen
Returns: 1 on success, 0 on failure

Function: int filesysFileRead(FileID id, char* data)
Purpose: read one byte from the open file at the current read pointer
Inputs: FileID id - the file handle obtained from filesysFileOpen
char* data - pointer to hold the read value
Returns: 1 on success, 0 on failure (EOF or other)

Function: int filesysFileReadLine(FileID id, char buf[], unsigned int bufSize)
Purpose: read a string of characters from the open file until the newline (\r) character into the buffer. The newline is not copied in the buffer. The null (\0) is appended to the buffer. The file pointer points past the newline character when done.
Inputs: FileID id - the file handle obtained from filesysFileOpen
char buf[] - the target buffer to hold the string
bufSize - the size of the buffer
Returns: 1 on success, 0 on failure (EOF or other)

Implementation:

SIZE STATS:

1) ratio of the flash segment _used_ to the flash segment _size_, for the file system: 128 / 512. That is, each flash segment has 128 bytes usable out of 512.
2) assembly source code file (ID 0): 6400 bytes
3) compiled executable file (ID 1): 3328 bytes

These limits are easily modifiable depending on the file sizes to be used in any application. We _do_ know that the compiled code requires significantly less than the half of the source. We didn't want to tailor the file sizes for current project too much: the justification is that the file system is GENERIC, not hardwired to the IJVM project.

Primary Search Algorithm:

Our code was implemented based on the *simplest primality test* called the Naïve Test.

Functional Specifications:

Algorithm: Given an input number n, we see if any integer m from 2 to $\sqrt{n}$ divides n. If n is divisible by any m then n is composite, otherwise it is prime.
We can also improve the efficiency by skipping all even m except 2, since if any even number divides n then 2 does. We can further improve by observing that all primes are of the form 6k ± 1, with the only exceptions of 2 and 3. This is because all integers can be expressed as (6k + i) for some k and for i = −1, 0, 1, 2, 3, or 4; 2 divides (6k + 0), (6k + 2), (6k + 4); and 3 divides (6k + 3). We first test if n is divisible by 2 or 3, then we run through all the numbers of form 6k ± 1 .

Implementation:

The boards know from the enumeration process which board ID each has and the total number of boards. Based on this, they have a common step value and each board is initialized using a different index, such that the values they check for are interleaved (please see figure 9).

The Master board is responsible for assembling the range given the number it receives from the keypad and sending it to the slaves. When boards are done computing numbers, they send the result to their master. Each board, upon receiving information from it’s slave, forwards these data to it’s master.

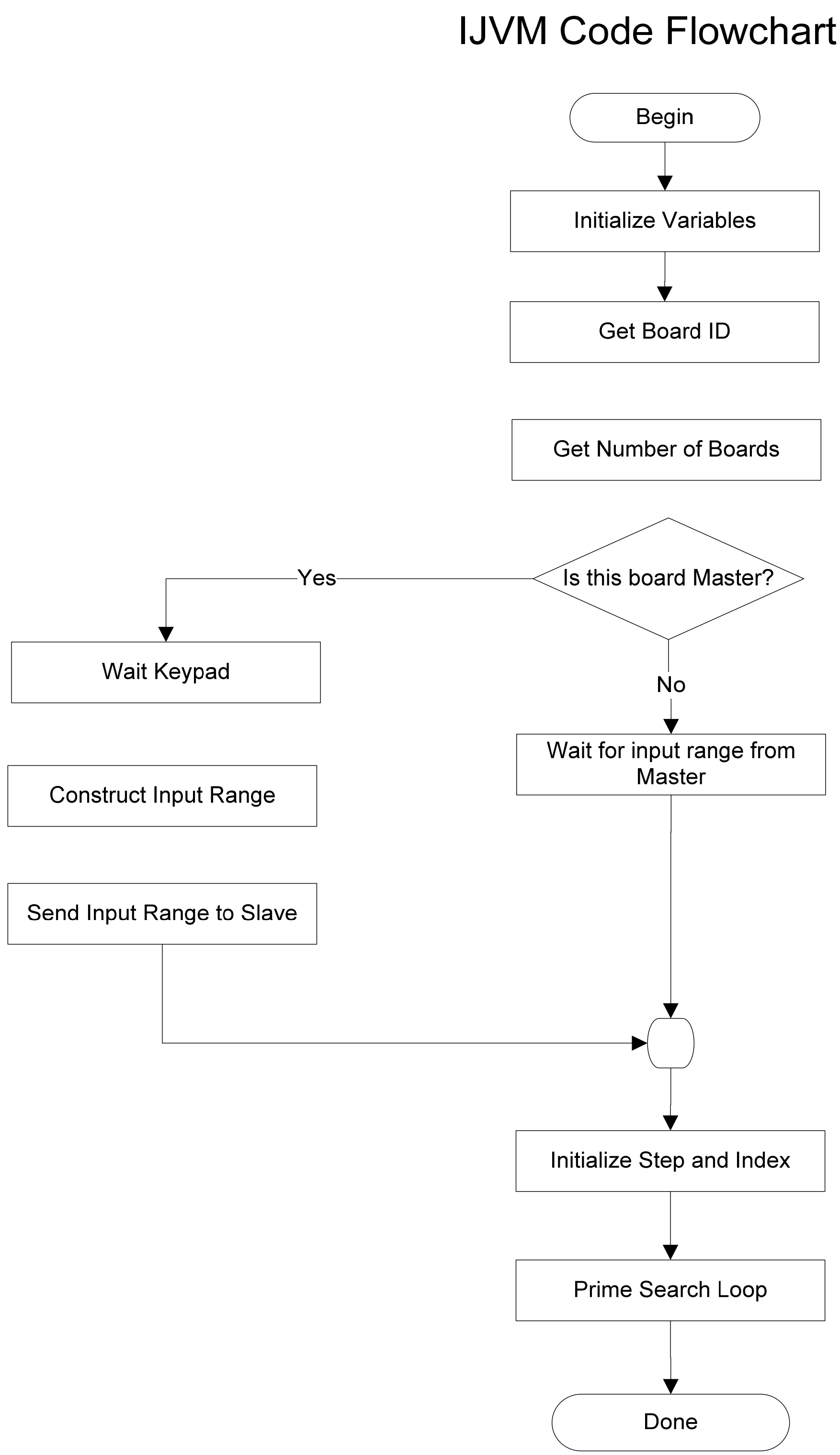

Figure 8: IJVM Code flowchart

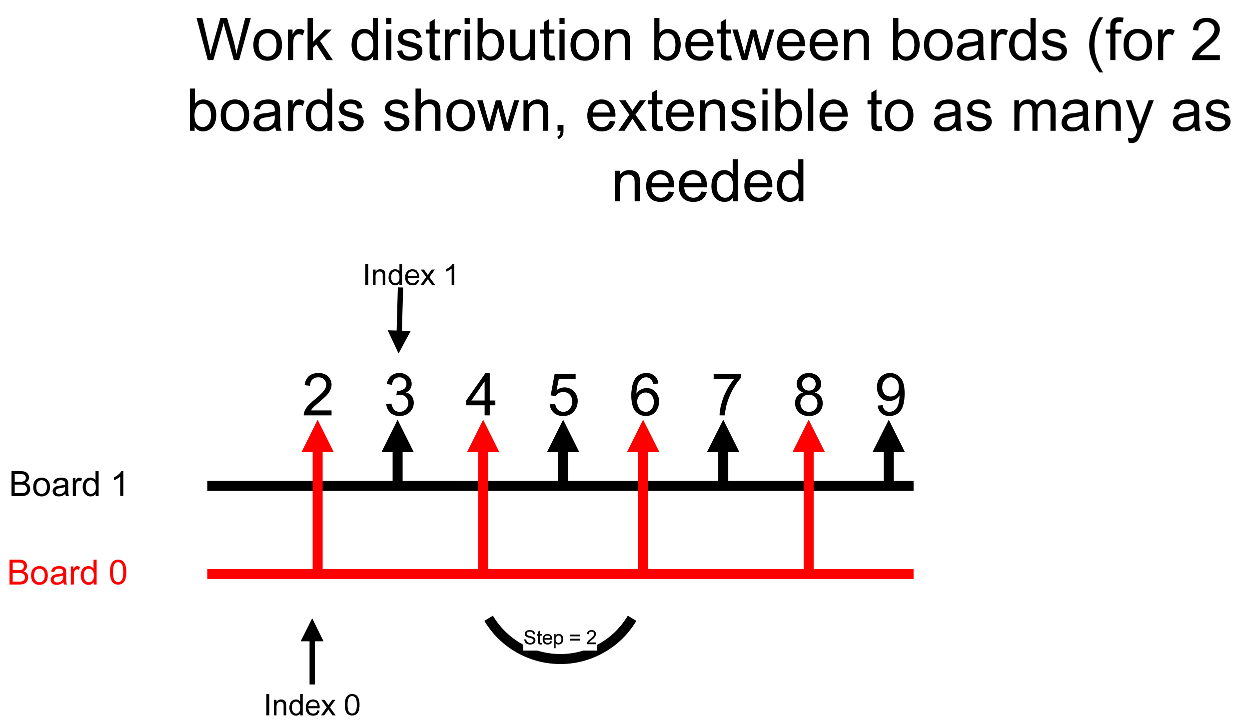

Figure 9: Work Distribution between the boards

CPLD Acceleration

To accelerate the performance of the virtual machine, instructions were implemented in hardware that were executed in parallel with the execution of the ijvm source code. Our primality test takes in the range, and the code is required to manipulate the range by performing operations such as square root, division and mod. We implemented the *square root* function in VHDL as our accelerated instruction.

Functional Specifications:

Function: `entity calculate`
Purpose: calculates the square root a number
Inputs: takes in the integer
Returns: square root of the integer

Implementation:

We tried to implement more functions such as division, mod, multiplication etc, but the limitation we encountered was in terms of exceeding the maximum number of cells on the CPLD.

Performance

Performance Table		
Number of Boards	Time	Accelerated SQRT function
1	10minutes 27seconds	NO
1	10minutes 2seconds	YES
2	5minutes 47seconds	NO
2	5minutes 30seconds	YES

The code took about 10 minutes and 27 seconds to run the whole range on one board and 5 minutes and 47 seconds to run it on two boards. The CPLD acceleration saved about 25 seconds on one board and 17 when running on two.

The rationale behind separating components, and the extra overhead and large code (the whole project compiled into 20KB) is to create a *generic* set of components (modules) not tied to the IJVM. Among all used components, all drivers (serial, timer, keypad, flash) are generic, filesys is generic and can be reused in other projects DIRECTLY AS IS. Console has a very similar structure to the previous lab and can be reused with slight modifications.

This needs to be mentioned, because our solution is slower than similar solutions from other teams and LARGE at the expense of being GENERIC, ROBUST, MAINTAINABLE, and EXTENDABLE. So in short, good software architecture at the expense of performance.

Bonus Points:

Extra requirements we satisfy:
We implemented all the extra features required for bonus marks:

1. Our boards dynamically determine their number and the total number of boards using a trace route protocol and
2. The exact same MSP430 and IJVM code is on all boards, thus allowing for easy extension to a large number or parallel boards.

Source Code:

C Code

```
#ifndef FILESYS_H

#define FILESYS_H


/* This is the public header for the filesys module */


//----------------------------------------------------------------------------

// Public filesys data types

//----------------------------------------------------------------------------


//file handle data type

typedef int FileID;


typedef enum {

  OPEN_READ,
```

```
  OPEN_WRITE,        //write in the buffered mode - flash is not updated right away
  OPEN_WRITE_THROUGH //write in the unbuffered mode - flash is updated right away
}
FileOpenMode;


#define FILE_ID_INVALID -1


//-----------------------------------------------------------------------------
// Public filesys module functions
//-----------------------------------------------------------------------------


int filesysModuleInit(void);
int filesysModuleDeinit(void);
int filesysFormat(void);
unsigned int filesysFileMaxSize(unsigned int fileIndex);
FileID filesysFileOpen(unsigned int fileIndex, FileOpenMode mode);
int filesysFileClose(FileID id);
int filesysFileRead(FileID id, char* data);
```

```
int filesysFileWrite(FileID id, char data);

int filesysFileSetReadPos(FileID id, unsigned int pos);

int filesysFileSetWritePos(FileID id, unsigned int pos);

int filesysFileGetReadPos(FileID id, unsigned int *pos);

int filesysFileGetWritePos(FileID id, unsigned int *pos);

int filesysFileReadLine(FileID id, char buf[], unsigned int bufSize);


#endif //FLASH_H




#ifndef IJVMCOMPILER_H

#define IJVMCOMPILER_H


/* This is the public header for the ijvmcompiler module */


//IJVM instructions

typedef enum {

  OP_BIPUSH = 1,
```

```
OP_DUP,        //0x02
OP_ERR,        //0x03
OP_GOTO,       //0x04
OP_HALT,       //0x05
OP_IADD,       //0x06
OP_IAND,       //0x07
OP_IFEQ,       //0x08
OP_IFLT,       //0x09
OP_IF_ICMPEQ,  //0x0A
OP_IINC,       //0x0B
OP_ILOAD,      //0x0C
OP_IN,         //0x0D
OP_INVOKEVIRTUAL, //0x0E
OP_IOR,        //0x0F
OP_IRETURN,    //0x10
OP_ISTORE,     //0x11
OP_ISUB,       //0x12
OP_LDC_W,      //0x13
```

```
  OP_NOP,        //0x14
  OP_OUT,       //0x15
  OP_POP,        //0x16
  OP_SWAP,       //0x17
  OP_WIDE        //0x18
}
IjvmISA;


typedef enum {
  OUT_STD_MASTER = 'M',
  OUT_STD_SLAVE = 'S',
  OUT_ALU_MUL = 'X',
  OUT_ALU_DIV = 'D',
  OUT_ALU_MOD = '%',
  OUT_ALU_SQRT = 'Q'
}
OutInstructionType;
```

```
typedef enum {
  IN_STD_MASTER = 'M',
  IN_STD_SLAVE = 'S',
  IN_KEYPAD = 'K',
  IN_BOARDID = 'B',
  IN_NUMBOARDS = '#'
}
InInstructionType;

//error messages
typedef enum {
  CE_SUCCESS,            //0
  CE_UNKNOWN_ERROR,      //1
  CE_CONST_DEFINED,      //2
  CE_VAR_DEFINED,        //3
  CE_METHOD_DEFINED,     //4
  CE_LABEL_DEFINED,      //5
  CE_CONST_NOT_DEFINED,  //6
```

```
  CE_VAR_NOT_DEFINED,     //7
  CE_METHOD_NOT_DEFINED,  //8
  CE_LABEL_NOT_DEFINED,   //9
  CE_IN_VAR,              //10
  CE_IN_CONST,            //11
  CE_IN_MAIN,             //12
  CE_IN_METHOD,           //13
  CE_NOT_IN_VAR,          //14
  CE_NOT_IN_CONST,        //15
  CE_NOT_IN_MAIN,         //16
  CE_NOT_IN_METHOD        //17
}
CompilerError;

int ijvmcompilerRun(void);
CompilerError ijvmcompilerGetLastError(void);

#endif //IJVMCOMPILER_H
```

```
#ifndef COMMON_H

#define COMMON_H


#include <__cross_studio_io.h>

#include <msp430x14x.h>


#define DIM(x) (sizeof(x)/sizeof(x[0]))


extern int initialized;


#endif //COMMON_H



#ifndef JVM_CONSOLE_H

#define JVM_CONSOLE_H


/* This is the public header for the console module */
```

```
int consoleModuleInit(void);
int consoleProcessFsm(void);

#endif //JVM_CONSOLE_H



#ifndef FLASH_H
#define FLASH_H

/* This is the public header for the flash module */

//This is the integer chunk of flash memory that can be manipulated.
#define FLASH_SEGMENT_SIZE  512

//Public flash module functions
int flashModuleInit(void);
int flashWriteSegment(unsigned short addr, char buffer[], unsigned int length);
```

```
int flashEraseSegment(unsigned short addr);

int flashReadSegment(unsigned short addr, char buffer[], unsigned int length);

int flashWriteByte(unsigned short addr, unsigned char data);


#endif //FLASH_H


#ifndef IJVM_H

#define IJVM_H


/* This is the public header for the ijvm module */


#define IJVM_APP_FILE_INDEX      0

#define IJVM_EXEC_FILE_INDEX     1

#define IJVM_MAX_SIZE_CODE_IN_RAM 200

//#define RUN_FROM_RAM


typedef enum {

  STATUS_READY,
```

```
  STATUS_LOADING,
  STATUS_RUNNING,
  STATUS_COMPLETED,
  STATUS_LOAD_ERROR,
  STATUS_RUN_ERROR
}
IjvmStatus;


typedef enum {
  ERROR_NONE,                 //0
  ERROR_CONST_SEG_FAULT,      //1
  ERROR_VAR_SEG_FAULT,        //2
  ERROR_VAR_SEG_OVERFLOW,     //3
  ERROR_STACK_OVERFLOW,       //4
  ERROR_STACK_UNDERFLOW,      //5
  ERROR_IRETURN_LOWEST_FRAME, //6
  ERROR_UNIMPLEMENTED,        //7
  ERROR_APPLICATION,          //8
```

```
  ERROR_INVALID_OPCODE,        //9
  ERROR_CANT_LOAD_EXEC_FILE,   //10
  ERROR_CODE_SEG_FAULT,        //11
  ERROR_EXEC_FILE_INVALID      //12
}
IjvmError;

typedef struct {
  unsigned char boardId;
  unsigned char boardsNum;
  unsigned char slavesNum;
  IjvmStatus status;
  IjvmError lastError;
}
IjvmGlobalData;

//Public module functions
```

```
int ijvmModuleInit(void);

int ijvmProcessFsm(void);

int ijvmStartExecution(void);


//Public globals


extern IjvmGlobalData ijvmGlobalData;


#endif //TIMER_H




#ifndef IJVM_IO_H

#define IJVM_IO_H


#include <serial.h>

#include <keypad.h>


/* This is the public header for the ijvm_io module */
```

```
typedef enum {
  IO_CONSUMER_JVM,
  IO_CONSUMER_APP
}
IOConsumer;

//Public module functions

int ijvmioModuleInit(void);
int ijvmioProcessFsm(void);
int ijvmioSetIOConsumer(IOConsumer cons);
int ijvmioSerialWrite(IOConsumer cons, SerialPort port, char c);
int ijvmioSerialWriteBuffered(IOConsumer cons, SerialPort port, char c);
int ijvmioSerialRead(IOConsumer cons, SerialPort port, char *c);
int ijvmioSerialReadString(IOConsumer cons, SerialPort port, char buffer[], unsigned int bufferSize);
int ijvmioSerialWriteString(IOConsumer cons, SerialPort port, char buffer[], unsigned int bufferSize);
int ijvmioSerialFlushReadBuffer(IOConsumer cons, SerialPort port);
```

```
int ijvmioSerialFlushWriteBuffer(IOConsumer cons, SerialPort port);

int ijvmioSerialRead(IOConsumer cons, SerialPort port, char *c);

KeyType ijvmioKeypadRead(IOConsumer cons);


#endif //IJVM_IO_H



#ifndef IJVMCPLD_H

#define IJVMCPLD_H


// CPLD OUTPUT ON P2

// CPLD INPUT ON P4

// CONTROL ON P5[0..6]


#define OPCODE_LOAD_X_UPPER     0b00000

#define OPCODE_LOAD_X_LOWER     0b00001

#define OPCODE_LOAD_Y_UPPER     0b00010

#define OPCODE_LOAD_Y_LOWER     0b00011
```

```
#define OPCODE_EXECUTE         0b00100


#define SELECT_QUOTIENT_UPPER   0b00000

#define SELECT_QUOTIENT_LOWER   0b01000

#define SELECT_REMAINDER_UPPER  0b10000

#define SELECT_REMAINDER_LOWER  0b11000


#define CPLD_ALU_START        0b100000

#define CPLD_READY_LINE       0b1000000

#define CPLD_ALU_READY  ((P5IN&CPLD_READY_LINE) == CPLD_READY_LINE)


int cpldModuleInit(void);

int cpldAluMod(int x, int y);

int cpldAluDiv(int x, int y);


#endif //IJVMCPLD_H
```

```
#ifndef KEYPAD_H
#define KEYPAD_H

/* This is the external header for the keypad driver module. */

// Data type for keypresses on a 4x3 keypad
typedef enum {
  KEY_NONE,
  KEY_1,
  KEY_2,
  KEY_3,
  KEY_4,
  KEY_5,
  KEY_6,
  KEY_7,
  KEY_8,
  KEY_9,
  KEY_STAR,
```

```
  KEY_0,
  KEY_POUND
}
KeyType;

//The key ring buffer size - the number of keypresses to buffer
#define KEYPAD_BUFFER_SIZE  16

//Functions available for the users of the keypad module
int keypadModuleInit(void);
int keypadProcess(void);
KeyType keypadRead(void);
void keypadISR(void);
char keypadKeyToChar(KeyType key);

#endif //KEYPAD_H
```

```c
#ifndef COMMON_H

#define COMMON_H


#include <__cross_studio_io.h>

#include <msp430x14x.h>


#define DIM(x) (sizeof(x)/sizeof(x[0]))


extern int initialized;


#endif //COMMON_H


#ifndef FILESYS_H

#define FILESYS_H


/* This is the public header for the filesys module */


//---------------------------------------------------------------------------
```

```
// Public filesys data types
//-----------------------------------------------------------------------------


//file handle data type
typedef int FileID;


typedef enum {
  OPEN_READ,
  OPEN_WRITE,        //write in the buffered mode - flash is not updated right away
  OPEN_WRITE_THROUGH //write in the unbuffered mode - flash is updated right away
}
FileOpenMode;


#define FILE_ID_INVALID -1


//-----------------------------------------------------------------------------
// Public filesys module functions
//-----------------------------------------------------------------------------
```

```
int filesysModuleInit(void);

int filesysModuleDeinit(void);

int filesysFormat(void);

unsigned int filesysFileMaxSize(unsigned int fileIndex);

FileID filesysFileOpen(unsigned int fileIndex, FileOpenMode mode);

int filesysFileClose(FileID id);

int filesysFileRead(FileID id, char* data);

int filesysFileWrite(FileID id, char data);

int filesysFileSetReadPos(FileID id, unsigned int pos);

int filesysFileSetWritePos(FileID id, unsigned int pos);

int filesysFileGetReadPos(FileID id, unsigned int *pos);

int filesysFileGetWritePos(FileID id, unsigned int *pos);

int filesysFileReadLine(FileID id, char buf[], unsigned int bufSize);


#endif //FLASH_H


#ifndef IJVMCOMPILER_H
```

```
#define IJVMCOMPILER_H

/* This is the public header for the ijvmcompiler module */

//IJVM instructions
typedef enum {
  OP_BIPUSH = 1,
  OP_DUP,        //0x02
  OP_ERR,        //0x03
  OP_GOTO,       //0x04
  OP_HALT,       //0x05
  OP_IADD,       //0x06
  OP_IAND,       //0x07
  OP_IFEQ,       //0x08
  OP_IFLT,       //0x09
  OP_IF_ICMPEQ,  //0x0A
  OP_IINC,       //0x0B
  OP_ILOAD,      //0x0C
```

```
  OP_IN,         //0x0D
  OP_INVOKEVIRTUAL, //0x0E
  OP_IOR,        //0x0F
  OP_IRETURN,    //0x10
  OP_ISTORE,     //0x11
  OP_ISUB,       //0x12
  OP_LDC_W,      //0x13
  OP_NOP,        //0x14
  OP_OUT,        //0x15
  OP_POP,        //0x16
  OP_SWAP,       //0x17
  OP_WIDE        //0x18
}
IjvmISA;

typedef enum {
  OUT_STD_MASTER = 'M',
  OUT_STD_SLAVE = 'S',
```

```
  OUT_ALU_MUL = 'X',

  OUT_ALU_DIV = 'D',

  OUT_ALU_MOD = '%',

  OUT_ALU_SQRT = 'Q'

}

OutInstructionType;


typedef enum {

  IN_STD_MASTER = 'M',

  IN_STD_SLAVE = 'S',

  IN_KEYPAD = 'K',

  IN_BOARDID = 'B',

  IN_NUMBOARDS = '#'

}

InInstructionType;


//error messages

typedef enum {
```

```
CE_SUCCESS,           //0
CE_UNKNOWN_ERROR,     //1
CE_CONST_DEFINED,     //2
CE_VAR_DEFINED,       //3
CE_METHOD_DEFINED,    //4
CE_LABEL_DEFINED,     //5
CE_CONST_NOT_DEFINED, //6
CE_VAR_NOT_DEFINED,   //7
CE_METHOD_NOT_DEFINED, //8
CE_LABEL_NOT_DEFINED, //9
CE_IN_VAR,            //10
CE_IN_CONST,          //11
CE_IN_MAIN,           //12
CE_IN_METHOD,         //13
CE_NOT_IN_VAR,        //14
CE_NOT_IN_CONST,      //15
CE_NOT_IN_MAIN,       //16
CE_NOT_IN_METHOD      //17
```

```
}

CompilerError;


int ijvmcompilerRun(void);

CompilerError ijvmcompilerGetLastError(void);


#endif //IJVMCOMPILER_H




unsigned char static_comp[] = {

0x00,0x01,0x01,0x00,0x11,0x00,0x0C,0x00,0x02,0x13,0xFF,0xFF,0x0A,0x00,0x16,0x0E,0x00,0x39,0x02,0x08,0x00,0x08,0x0C,0x00,0x01,0x4D,0x1
5,0x16,0x0B,0x00,0x01,0x04,0xFF,0xE7,0x05,0x01,0x00,0x0C,0x00,0x01,0x4D,0x15,0x10,0x00,0x00,0x01,0x00,0x16,0x13,0x00,0x4B,0x0D,0x02,0
x08,0xFF,0xFA,0x10,0x01,0x03,0x0C,0x00,0x08,0x00,0x68,0x0C,0x00,0x01,0x01,0x0A,0x00,0x61,0x0C,0x00,0x01,0x02,0x0A,0x00,0x5D,0x0C,0x0
0,0x01,0x03,0x0A,0x00,0x56,0x0C,0x00,0x01,0x02,0x13,0x00,0x25,0x15,0x08,0x00,0x48,0x0C,0x00,0x01,0x03,0x13,0x00,0x25,0x15,0x08,0x00,0
x3D,0x0C,0x00,0x13,0x00,0x51,0x15,0x11,0x01,0x01,0x05,0x11,0x02,0x0C,0x01,0x0C,0x02,0x12,0x09,0x00,0x2C,0x0C,0x00,0x0C,0x02,0x13,0x0
0,0x25,0x15,0x08,0x00,0x1E,0x0C,0x00,0x01,0x02,0x0C,0x02,0x06,0x02,0x11,0x02,0x13,0x00,0x25,0x15,0x08,0x00,0x0D,0x01,0x04,0x0C,0x02,0
x06,0x11,0x02,0x04,0xFF,0xD5,0x01,0x00,0x10,0x01,0x01,0x10

};


#ifndef JVM_CONSOLE_STRINGS_H

#define JVM_CONSOLE_STRINGS_H
```

```c
/* This private header contains the console user messages, and the valid
 * admin command strings
 */

#define COMMAND_BUFFER_SIZE 40

#define CMD_DOWNLOAD_MASTER   "D0"
#define CMD_DOWNLOAD_SLAVE    "D1"
#define CMD_UPLOAD_MASTER     "U0"
#define CMD_UPLOAD_MASTER_HEX "u0"
#define CMD_UPLOAD_SLAVE      "U1"
#define CMD_IJVM_START        "start"
#define CMD_IJVM_COMPILE      "compile"
#define CMD_FSFORMAT          "fmt"
#define CMD_ENUMERATE         "enum"
#define CMD_BOARDID           "bid"
#define MAX_CMD_LENGTH        7        //the maximum length of the command
```

```
#define MSG_PROMPT             "IJVM>>"
#define MSG_INVALID_CMD        "Invalid command. Use D0,D1,U0,U1,start,compile,enum,fmt\r\n"
#define MSG_RUNTIME_ERROR      "Runtime error\r\n"
#define MSG_ENUMERATE_OK       "Enumeration done\r\n"
#define ENUMERATION_TIMEOUT    1000       // milliseconds


#define MSG_DL_MASTER_OK       "\r\nDownload from the master done\r\n"
#define MSG_DL_SLAVE_OK        "\r\nDownload from the slave done\r\n"
#define MSG_UL_MASTER_OK       "Upload to the master done\r\n"
#define MSG_UL_SLAVE_OK        "Upload to the slave done\r\n"
#define MSG_IJVM_START_OK      "Execution starting...\r\n"
#define MSG_IJVM_COMPILING     "Compiling...\r\n"
#define MSG_IJVM_COMPILE_OK    "Compilation successful\r\n"
#define MSG_IJVM_COMPILE_ERROR   "Compile error: ID="
#define MSG_IJVM_EXEC_DONE       "\r\nExecution completed.\r\n"
#define MSG_IJVM_EXEC_RUN_ERROR   "Execution runtime error: ID="
#define MSG_FSFORMAT_OK        "File system format OK.\r\n"
```

```
#define MSG_FSFORMAT_ERR        "File system format error.\r\n"

#define STRLEN_CONST(str) (sizeof(str) - 1)

#endif // JVM_CONSOLE_STRINGS_H

// This module implements the user interface to the IJVM - the serial console.
// When the virtual machine is not running, it presents a prompt on the master
// serial port, and processes all commands received at this port, like upload
// a new assembly source code, compile it, start the execution of the IJVM etc.
// It also prints the error and status messages. For the full list of commands
// see console_strings.h

#include <common.h>
#include <string.h>
#include <stdio.h>
#include <console.h>
#include <ijvm.h>
```

```
#include <ijvm_io.h>

#include <ijvmcompiler.h>

#include <filesys.h>

#include <timer.h>

#include "console_strings.h"




//----------------------------------------------------------------------------

// Private data types and macros

//----------------------------------------------------------------------------


//Uncomment to include the extra functionality - uploading a compiled file

//in ASCII HEX form with two ASCII characters per byte and spaces as delimiters

#define UPLOAD_MASTER_HEX


//Private console FSM states

typedef enum {

  STATE_DISPLAY_PROMPT,
```

```
  STATE_WAIT_COMMAND,
  STATE_PARSE_COMMAND,
  STATE_PARSE_ERROR,
  STATE_DOWNLOAD_MASTER,
  STATE_DOWNLOAD_MASTER_PROCESS,
  STATE_DOWNLOAD_MASTER_DONE,
  STATE_DOWNLOAD_SLAVE,
  STATE_DOWNLOAD_SLAVE_PROCESS,
  STATE_DOWNLOAD_SLAVE_DONE,
  STATE_UPLOAD_MASTER,
  STATE_UPLOAD_MASTER_PROCESS,
  STATE_UPLOAD_MASTER_DONE,
#ifdef UPLOAD_MASTER_HEX
  STATE_UPLOAD_MASTER_HEX,
  STATE_UPLOAD_MASTER_HEX_PROCESS,
  STATE_UPLOAD_MASTER_HEX_DONE,
#endif
  STATE_UPLOAD_SLAVE,
```

```
  STATE_UPLOAD_SLAVE_PROCESS,
  STATE_UPLOAD_SLAVE_DONE,
  STATE_FSFORMAT,
  STATE_IJVM_START,
  STATE_IJVM_STARTED,
  STATE_IJVM_COMPILE,
  STATE_IJVM_COMPILE_DONE,
  STATE_BOARDID,
  STATE_BOARDID_PROCESS,
  STATE_BOARDID_DONE,
  STATE_ENUMERATE,
  STATE_ENUMERATE_PROCESS,
  STATE_ENUMERATE_DONE,
  STATE_RUNTIME_ERROR

}
ConsoleState;
```

```
//-----------------------------------------------------------------------------
// Private function prototypes
//-----------------------------------------------------------------------------

static int beginChainEnumerate(ConsoleState ProcessState);
static int processChainEnumerate(ConsoleState DoneState);

//-----------------------------------------------------------------------------
// Private globals
//-----------------------------------------------------------------------------

// This is to track the timeout on the chain enumeration procedure
static TimerId enumerationTimer;

//The FSM current state
static ConsoleState state  = STATE_DISPLAY_PROMPT;

//private character buffer to hold the user input and output
```

```
#define MESSAGE_BUFFER_SIZE   60

static char messageBuffer[MESSAGE_BUFFER_SIZE]; //the buffer space

static int messageLength = 0;                //the size of the buffer used


// The User input format:

// PROMPT >> {command} [{arg1 integer}]


//the command

static char commandString[MAX_CMD_LENGTH + 1];

static int arg1Int;

//the number of the tokens in the command

static int commandNumTokens;


//Flag to track the initialization state of the module

static int moduleInitialized = 0;


//The file handle for the uploaded assembly source code

static FileID appFile = FILE_ID_INVALID;
```

```
//-----------------------------------------------------------------------------
// consoleModuleInit: initialize this module before using any of its functions.
//   It initializes all related modules: file system, IO proxy, and timers
// arguments: none
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int consoleModuleInit(void)
{
  if (moduleInitialized) {
    return 1;
  }
  moduleInitialized = 1;

  if (!ijvmioModuleInit() || !filesysModuleInit() || !timerModuleInit()) {
    moduleInitialized = 0;
    return 0;
  }
```

```
  return 1;
}

//----------------------------------------------------------------------------
// consoleProcessFsm: this function does one processing pass of the console FSM.
//   It is not blocking, and should be called as often as possible.
// arguments: none
// return: 1 if successful, 0 if not.
//----------------------------------------------------------------------------
int consoleProcessFsm(void)
{
  if (!moduleInitialized) {
    return 0;
  }

  switch(state) {
```

```
//-----------------------------------------------------------------------
// This state displays the prompt on the ijvmioSerial terminal
//-----------------------------------------------------------------------
case STATE_DISPLAY_PROMPT:
  if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_PROMPT, STRLEN_CONST(MSG_PROMPT))) {
    ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_1);

    messageLength = 0;          //reset the length for the typed command
    state = STATE_WAIT_COMMAND;
  }
  break;

//-----------------------------------------------------------------------
// This state is reading the user input into the message buffer until the
// user presses Enter. Then the FSM proceeds to STATE_PARSE_COMMAND.
//-----------------------------------------------------------------------
case STATE_WAIT_COMMAND:
  {
```

```
char newChar;

//check if there's a new character received
if (ijvmioSerialRead(IO_CONSUMER_JVM, COM_1, &newChar)) {

  //echo every typed command character
  ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, newChar);

  if (newChar == '\r') {
    //if the user pressed enter (CR), append LF, and discard further input
    ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, '\n');
    ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_1);

    //terminate the message buffer, a command has been received
    messageBuffer[messageLength] = '\0';

    //FSM will parse the received command
    state = STATE_PARSE_COMMAND;
```

```
        break;
      }

      if (messageLength < (MESSAGE_BUFFER_SIZE - 1)) {
        //save user input in the command buffer
        messageBuffer[messageLength++] = newChar;
      }
    }
  }
  break;

//------------------------------------------------------------------------
// This state parses the received string (in the messageBuffer) into the
// command, the string argument and the integer argument. Then if the
// command is valid, the FSM dispatches to the appropriate states to
// execute the command.
//------------------------------------------------------------------------
case STATE_PARSE_COMMAND:
```

```
commandNumTokens = sscanf(
      messageBuffer,
      "%s %d",
      commandString,
      &arg1Int);


if (messageLength == 0 || commandNumTokens == 0) {
  state = STATE_DISPLAY_PROMPT;
}
else if (commandNumTokens == 1 && !strcmp(CMD_DOWNLOAD_MASTER, commandString)) {
  state = STATE_DOWNLOAD_MASTER;
}
else if (commandNumTokens == 1 && !strcmp(CMD_DOWNLOAD_SLAVE, commandString)) {
  state = STATE_DOWNLOAD_SLAVE;
}
else if (commandNumTokens == 1 && !strcmp(CMD_UPLOAD_MASTER, commandString)) {
  state = STATE_UPLOAD_MASTER;
}
```

```
#ifdef UPLOAD_MASTER_HEX
    else if (commandNumTokens == 1 && !strcmp(CMD_UPLOAD_MASTER_HEX, commandString)) {
      state = STATE_UPLOAD_MASTER_HEX;
    }
#endif //UPLOAD_MASTER_HEX
    else if (commandNumTokens == 1 && !strcmp(CMD_UPLOAD_SLAVE, commandString)) {
      state = STATE_UPLOAD_SLAVE;
    }
    else if (commandNumTokens == 1 && !strcmp(CMD_IJVM_START, commandString)) {
      state = STATE_IJVM_START;
    }
    else if (commandNumTokens == 1 && !strcmp(CMD_FSFORMAT, commandString)) {
      state = STATE_FSFORMAT;
    }
    else if (commandNumTokens == 1 && !strcmp(CMD_ENUMERATE, commandString)) {
      state = STATE_ENUMERATE;
    }
    else if (commandNumTokens == 1 && !strcmp(CMD_IJVM_COMPILE, commandString)) {
```

```
  state = STATE_IJVM_COMPILE;
 }
 else if (commandNumTokens == 2 && !strcmp(CMD_BOARDID, commandString)) {
  state = STATE_BOARDID;
 }
 else {
  state = STATE_PARSE_ERROR;
 }
 break;

//-----------------------------------------------------------------------
// This state displays the error message to the terminal and returns to
// display the prompt.
//-----------------------------------------------------------------------
case STATE_PARSE_ERROR:
 if(ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_INVALID_CMD, STRLEN_CONST(MSG_INVALID_CMD))) {
  state = STATE_DISPLAY_PROMPT;
 }
```

```
  else {
    state = STATE_RUNTIME_ERROR;
  }
  break;

//------------------------------------------------------------------------
// This state initiates the execution of the command requesting a download
// of the application code from the master board
//------------------------------------------------------------------------
case STATE_DOWNLOAD_MASTER:
  appFile = filesysFileOpen(IJVM_APP_FILE_INDEX, OPEN_READ);
  if (appFile == FILE_ID_INVALID) {
    state = STATE_DISPLAY_PROMPT;
  }
  else {
    state = STATE_DOWNLOAD_MASTER_PROCESS;
  }
  break;
```

```
//------------------------------------------------------------------------
// This state does the execution of the 'download from master board' command
//------------------------------------------------------------------------
case STATE_DOWNLOAD_MASTER_PROCESS:
 {
   char c;
   if (filesysFileRead(appFile, &c)) {
     ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, c);
     if (c == '\r') {
       ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, '\n');        //append LF
     }
   }
   else {
     //when EOF, send the transmission termination character '~'
     ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, '~');
     state = STATE_DOWNLOAD_MASTER_DONE;
   }
```

```
 }
 break;

//------------------------------------------------------------------------
// This state completes the execution of the 'download from master board' command
//------------------------------------------------------------------------
case STATE_DOWNLOAD_MASTER_DONE:
 filesysFileClose(appFile);
 appFile = FILE_ID_INVALID;

 if(ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_DL_MASTER_OK, STRLEN_CONST(MSG_DL_MASTER_OK))) {
  state = STATE_DISPLAY_PROMPT;
 }
 else {
  state = STATE_RUNTIME_ERROR;
 }
 break;
```

```
//------------------------------------------------------------------------

// This state initiates the execution of the command requesting a download

// of the application code from the slave board

//------------------------------------------------------------------------

case STATE_DOWNLOAD_SLAVE:

 {

  //forward the download command to the slave

  if (ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_2) &&

     ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_2, CMD_DOWNLOAD_MASTER, STRLEN_CONST(CMD_DOWNLOAD_MASTER)) &&

     ijvmioSerialWrite(IO_CONSUMER_JVM, COM_2, '\r')) {


   state = STATE_DOWNLOAD_SLAVE_PROCESS;

  }

  else {

   state = STATE_RUNTIME_ERROR;

  }


 }
```

```
  break;

//------------------------------------------------------------------------
// This state does the execution of the 'download from slave board' command
//------------------------------------------------------------------------
case STATE_DOWNLOAD_SLAVE_PROCESS:
  {
    char c;

    //read the data sent by the slave and forward it to the master that
    //requested the download originally
    if (ijvmioSerialRead(IO_CONSUMER_JVM, COM_2, &c)) {

      //write the received data in the buffered mode so as not to miss
      //any characters sent to us by the slave
      ijvmioSerialWriteBuffered(IO_CONSUMER_JVM, COM_1, c);

      if (c == '~') {
```

```
      //when the slave sends the whole file and the transmission-terminating

      //character, flush the remaining serial transmit buffer holding any

      //pending data to be sent to the master (this is needed since we

      //were buffering that data)


      state = STATE_DOWNLOAD_SLAVE_DONE;

      ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_2);

      ijvmioSerialFlushWriteBuffer(IO_CONSUMER_JVM, COM_1);

     }

   }

  }

  break;


//-----------------------------------------------------------------------

// This state completes the execution of the 'download from slave board' command

//-----------------------------------------------------------------------

case STATE_DOWNLOAD_SLAVE_DONE:
```

```
  if(ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_DL_SLAVE_OK, STRLEN_CONST(MSG_DL_SLAVE_OK))) {
   state = STATE_DISPLAY_PROMPT;
  }
  else {
   state = STATE_RUNTIME_ERROR;
  }
  break;

 //------------------------------------------------------------------------
 // This state initiates the execution of the command requesting an upload
 // of the application code to the master board
 //------------------------------------------------------------------------
 case STATE_UPLOAD_MASTER:
  ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_1);

  //Open the file to hold the upcoming data in the WRITE_THROUGH (!) mode so
  //that writes to the flash are done quickly. If we buffer the data in the
  //file system then we may miss characters every time when the file system
```

```
 //flushes its buffer to flash

 appFile = filesysFileOpen(IJVM_APP_FILE_INDEX, OPEN_WRITE_THROUGH);
 if (appFile == FILE_ID_INVALID) {
  state = STATE_DISPLAY_PROMPT; //TODO error
 }
 else {
  state = STATE_UPLOAD_MASTER_PROCESS;
 }
 break;

//-----------------------------------------------------------------------
// This state does the execution of the 'upload to master board' command
//-----------------------------------------------------------------------
case STATE_UPLOAD_MASTER_PROCESS:
 {
  char newChar;
```

```
    //Every character received from the master is stored in the file until

    //we receive the transmission-terminating '~'


    if (ijvmioSerialRead(IO_CONSUMER_JVM, COM_1, &newChar)) {

      if (newChar == '~') {

        state = STATE_UPLOAD_MASTER_DONE;

      }

      else {

        filesysFileWrite(appFile, newChar);

      }

    }

  }

  break;


//-----------------------------------------------------------------------

// This state completes the execution of the 'upload to master board' command

//-----------------------------------------------------------------------

case STATE_UPLOAD_MASTER_DONE:
```

```
      //close the file
      filesysFileClose(appFile);
      appFile = FILE_ID_INVALID;

      if(ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_UL_MASTER_OK, STRLEN_CONST(MSG_UL_MASTER_OK))) {
        state = STATE_DISPLAY_PROMPT;
      }
      else {
        state = STATE_RUNTIME_ERROR;
      }
      break;

#ifdef UPLOAD_MASTER_HEX
    //------------------------------------------------------------------------
    // This state initiates the execution of the command requesting an upload
    // of the (compiled) application code to the master board
    // (in the ASCII HEX format)
```

```
//-----------------------------------------------------------------------
case STATE_UPLOAD_MASTER_HEX:
  appFile = filesysFileOpen(IJVM_EXEC_FILE_INDEX, OPEN_WRITE_THROUGH);
  if (appFile == FILE_ID_INVALID) {
    state = STATE_DISPLAY_PROMPT;
  }
  else {
    messageLength = 0;
    state = STATE_UPLOAD_MASTER_HEX_PROCESS;
  }
  break;

//-----------------------------------------------------------------------
// This state does the execution of the 'upload to master board' command
// (in the ASCII HEX format)
//-----------------------------------------------------------------------
case STATE_UPLOAD_MASTER_HEX_PROCESS:
  {
```

```
unsigned char byte;
unsigned char upper;
unsigned char lower;

//Keep receiving characters in the buffer until we get a space, then
//the received characters are converted from their ASCII format to a byte
//and stored in the file.
// Ex:  serial: "65 FA 05 33" -> file: 0x65FA0533

if (ijvmioSerialRead(IO_CONSUMER_JVM, COM_1, &messageBuffer[messageLength])) {

  if (messageBuffer[messageLength] == ' ') {
    upper = messageBuffer[messageLength - 2];
    upper = (upper >= 'A') ? (upper - 'A' + 0xA) : (upper - '0');
    lower = messageBuffer[messageLength - 1];
    lower = (lower >= 'A') ? (lower - 'A' + 0xA) : (lower - '0');

    filesysFileWrite(appFile, (upper << 4 | lower));
```

```
        messageLength = 0;
      }
      else if (messageBuffer[messageLength] == '~') {
        state = STATE_UPLOAD_MASTER_HEX_DONE;
        break;
      }
      else {
        messageLength++;
      }
    }
  }
  break;

//------------------------------------------------------------------------
// This state completes the execution of the 'upload to master board'
// command (ASCII HEX format)
//------------------------------------------------------------------------
case STATE_UPLOAD_MASTER_HEX_DONE:
```

```
      filesysFileClose(appFile);
      appFile = FILE_ID_INVALID;

      if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_UL_MASTER_OK, STRLEN_CONST(MSG_UL_MASTER_OK))) {
        state = STATE_DISPLAY_PROMPT; //TODO
      }
      else {
        state = STATE_RUNTIME_ERROR;
      }
      break;

#endif //UPLOAD_MASTER_HEX

    //-----------------------------------------------------------------------
    // This state initiates the execution of the command requesting an upload
    // of the application code to the slave board
    //-----------------------------------------------------------------------
    case STATE_UPLOAD_SLAVE:
```

```
//forward the upload request command to the slave board

if (ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_2) &&
  ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_2, CMD_UPLOAD_MASTER, STRLEN_CONST(CMD_UPLOAD_MASTER)) &&
  ijvmioSerialWrite(IO_CONSUMER_JVM, COM_2, '\r')) {

  ijvmioSerialFlushWriteBuffer(IO_CONSUMER_JVM, COM_2);
  messageLength = 0;
  state = STATE_UPLOAD_SLAVE_PROCESS;
}
else {
 state = STATE_RUNTIME_ERROR;
}
break;

//-----------------------------------------------------------------------
// This state does the execution of the 'upload to slave board' command
```

```
//------------------------------------------------------------------------
case STATE_UPLOAD_SLAVE_PROCESS:
 {
  char newChar;

  //Every character received from our master is forwarded to our slave in
  //the buffered mode so as not to miss any received characters. The
  //forwarding stops when '~' is detected.

  if (ijvmioSerialRead(IO_CONSUMER_JVM, COM_1, &newChar)) {
   ijvmioSerialWriteBuffered(IO_CONSUMER_JVM, COM_2, newChar);

   if (newChar == '~') {
    state = STATE_UPLOAD_SLAVE_DONE;

    //Since we were forwarding data in the buffered mode, we need to
    //flush the buffer to the slave to commit all pending transfers
    ijvmioSerialFlushWriteBuffer(IO_CONSUMER_JVM, COM_2);
```

```
        break;
      }
    }
  }
  break;

//------------------------------------------------------------------------
// This state completes the execution of the 'upload to slave board' command
//------------------------------------------------------------------------
case STATE_UPLOAD_SLAVE_DONE:
  if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_UL_SLAVE_OK, STRLEN_CONST(MSG_UL_SLAVE_OK))) {
    state = STATE_DISPLAY_PROMPT;
  }
  else {
    state = STATE_RUNTIME_ERROR;
  }
  break;
```

```
//------------------------------------------------------------------------
// This state executes the command requesting to start IJVM engine, forwards
// the request to slaves, writes the confirmation to the terminal, and
// returns to displaying the prompt.
//------------------------------------------------------------------------
case STATE_IJVM_START:

  if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_2, CMD_IJVM_START, STRLEN_CONST(CMD_IJVM_START)) &&
      ijvmioSerialWrite(IO_CONSUMER_JVM, COM_2, '\r') &&
      ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_IJVM_START_OK, STRLEN_CONST(MSG_IJVM_START_OK)) &&
      ijvmStartExecution()) {

    state = STATE_IJVM_STARTED;
  }
  else {
    state = STATE_RUNTIME_ERROR;
  }
```

```
  break;

//-----------------------------------------------------------------------
// This state executes while the ijvm has been started. It monitors its
// status until the ijvm stops, and prints the message about the run outcome
//-----------------------------------------------------------------------
case STATE_IJVM_STARTED:
  {
    int success;

    switch (ijvmGlobalData.status) {
      case STATUS_RUN_ERROR:
        snprintf(messageBuffer, MESSAGE_BUFFER_SIZE - 1, "%s%d\r\n", MSG_IJVM_EXEC_RUN_ERROR, ijvmGlobalData.lastError);
        success = ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, messageBuffer, strlen(messageBuffer));
        break;
      case STATUS_COMPLETED:
        success = ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_IJVM_EXEC_DONE, STRLEN_CONST(MSG_IJVM_EXEC_DONE));
        break;
```

```
    default:
      //the VM is still running
      success = 2;
  }
  if (success == 0) {
    //VM is stopped but there was an error writing the message to the serial port
    state = STATE_RUNTIME_ERROR;
  }
  else if (success == 1) {
    //VM was stopped and the status message has been succesfully sent to the serial port
    state = STATE_DISPLAY_PROMPT;
  }
 }
 break;

//------------------------------------------------------------------------
// This state executes the command to request the compilation of the uploaded
// source code. It invokes the ijvmcompiler module to do the compilation.
```

```
//-----------------------------------------------------------------------
case STATE_IJVM_COMPILE:

  //Print the message about the start of the compilation
  if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_2, CMD_IJVM_COMPILE, STRLEN_CONST(CMD_IJVM_COMPILE)) &&
      ijvmioSerialWrite(IO_CONSUMER_JVM, COM_2, '\r') &&
      ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_IJVM_COMPILING, STRLEN_CONST(MSG_IJVM_COMPILING))) {

    ijvmcompilerRun();  //this call blocks until compilation finishes
    state = STATE_IJVM_COMPILE_DONE;
  }
  else {
    state = STATE_RUNTIME_ERROR;
  }
  break;

//-----------------------------------------------------------------------
// This state is activated upon the end of the compilation.
```

```
//------------------------------------------------------------------------
case STATE_IJVM_COMPILE_DONE:
 {
   //Print the message about the end and the result of the compilation
   CompilerError lastCompilerError = ijvmcompilerGetLastError();

   int success = 0;
   if (lastCompilerError == CE_SUCCESS) {
     //compilation successful
     success = ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_IJVM_COMPILE_OK, STRLEN_CONST(MSG_IJVM_COMPILE_OK));
   }
   else {
     //compilation unsuccessful, print the error ID
     snprintf(messageBuffer, MESSAGE_BUFFER_SIZE - 1, "%s%d\r\n", MSG_IJVM_COMPILE_ERROR, lastCompilerError);
     success = ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, messageBuffer, strlen(messageBuffer));
   }

   if (success) {
```

```
      state = STATE_DISPLAY_PROMPT;
    }
    else {
      state = STATE_RUNTIME_ERROR;
    }
  }
  break;

//------------------------------------------------------------------------
// This state executes the command requesting to format the flash file
// system, erasing all filesystem data.
//------------------------------------------------------------------------
case STATE_FSFORMAT:
  {
    int success = filesysFormat();

    if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_2, CMD_FSFORMAT, STRLEN_CONST(CMD_FSFORMAT)) &&
        ijvmioSerialWrite(IO_CONSUMER_JVM, COM_2, '\r') &&
```

```
    ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1,
     success ? MSG_FSFORMAT_OK : MSG_FSFORMAT_ERR,
     success ? STRLEN_CONST(MSG_FSFORMAT_OK) : STRLEN_CONST(MSG_FSFORMAT_ERR))) {

   state = STATE_DISPLAY_PROMPT;
  }
  else {
   state = STATE_RUNTIME_ERROR;
  }
 }
 break;

//-----------------------------------------------------------------------
// This state begins enumerating all devices on the slave chain
// this command is to be invoked only from the PC terminal
//-----------------------------------------------------------------------
case STATE_ENUMERATE:
{
```

```
  if (!beginChainEnumerate(STATE_ENUMERATE_PROCESS)) {
    state = STATE_RUNTIME_ERROR;
  }
}
break;

//-----------------------------------------------------------------------
// This state processes the request to enumerate all devices on the chain
//-----------------------------------------------------------------------
case STATE_ENUMERATE_PROCESS:
{
    if (!processChainEnumerate(STATE_ENUMERATE_DONE)) {
      state = STATE_RUNTIME_ERROR;
    }
}
break;

//-----------------------------------------------------------------------
```

```
// This state completes the enumeration of all devices on the chain
//-----------------------------------------------------------------------
case STATE_ENUMERATE_DONE:
{
  //display the number of boards detected
  snprintf(messageBuffer, MESSAGE_BUFFER_SIZE - 1, "%s: %d board(s)\r\n", MSG_ENUMERATE_OK, ijvmGlobalData.boardsNum);
  if (ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, messageBuffer, strlen(messageBuffer))) {
    state = STATE_DISPLAY_PROMPT;
  }
  else {
    state = STATE_RUNTIME_ERROR;
  }
}
break;

//-----------------------------------------------------------------------
// This state processes the board ID assignement command from the master board
// as part of the enumeration request.
```

```
//---------------------------------------------------------------------------
case STATE_BOARDID:
{
    ijvmGlobalData.boardId = arg1Int;

    //Send a board ID acknowledge to the master board and process further
    //slaves down the chain
    if (!ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, '@') ||
        !beginChainEnumerate(STATE_BOARDID_PROCESS)) {
      state = STATE_RUNTIME_ERROR;
    }
}
break;

//--------------------------------------------------------------------------
// This state processes the enumeration responses from the slaves down the
// chain until the enumeration timeout.
//---------------------------------------------------------------------------
```

```
case STATE_BOARDID_PROCESS:
{
   if (!processChainEnumerate(STATE_BOARDID_DONE)) {
     state = STATE_RUNTIME_ERROR;
   }
}
break;

//-----------------------------------------------------------------------
// This state completes the internal enumeration process
//-----------------------------------------------------------------------
case STATE_BOARDID_DONE:
  state = STATE_DISPLAY_PROMPT;
break;

//-----------------------------------------------------------------------
// This state is an exceptional error state. It displays the error message
// to the console and the FSM returns to displaying the prompt.
```

```
    //---------------------------------------------------------------------------
    case STATE_RUNTIME_ERROR:
      ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_1, MSG_RUNTIME_ERROR, STRLEN_CONST(MSG_RUNTIME_ERROR));
      state = STATE_DISPLAY_PROMPT;
      break;
  }

  return 1;
}

//---------------------------------------------------------------------------
// beginChainEnumerate: this function forwards the internal board enumerate
//   request to further slave down the device chain, assigning to it the next
//   board ID in the sequence.
// arguments: ConsoleState processState - the next FSM state to assign when done
// return: 1 if success, 0 if failure
//---------------------------------------------------------------------------
static int beginChainEnumerate(ConsoleState processState)
```

```
{
 ijvmGlobalData.boardsNum = 0; //reset the total board counter
 ijvmGlobalData.slavesNum = 0; //reset the number of the remaining slaves down the chain

 //Construct the board ID enumeration command with the next board ID in the
 //sequence: {this board's ID} + 1 => {board ID assigned to slave}

 snprintf(messageBuffer, MESSAGE_BUFFER_SIZE - 1, "%s %d\r", CMD_BOARDID, ijvmGlobalData.boardId + 1);

 //Send the board ID command to the slave
 if(!ijvmioSerialFlushReadBuffer(IO_CONSUMER_JVM, COM_2) ||
   !ijvmioSerialWriteString(IO_CONSUMER_JVM, COM_2, messageBuffer, strlen(messageBuffer))) {
  return 0;
 }

 //initialize and start the enumeration timeout timer
 enumerationTimer = timerCreate();
 if ((enumerationTimer == TIMER_ID_INVALID) ||
```

```
    !timerStart(enumerationTimer, ENUMERATION_TIMEOUT)) return 0;

  state = processState;
  return 1;
}

//---------------------------------------------------------------------------
// processChainEnumerate: this function waits for the board ID responses (ACK's)
//   from the slave boards and counts their number, and forwards the ACK's upstream
//   until the enumeration timeout.
// arguments: ConsoleState processState - the next FSM state to assign when done
// return: 1 if success, 0 if failure
//---------------------------------------------------------------------------
static int processChainEnumerate(ConsoleState doneState)
{
  char result;

  //Wait for data on the slave port for slave ACK's
```

```
if (ijvmioSerialRead(IO_CONSUMER_JVM, COM_2, &result)) {

  if (result == '@') {
    // received a slave ACK, increment the number of the slaves downstream
    ijvmGlobalData.slavesNum++;

    //if this board is not the head master board, then forward the
    //received slave ack up the chain
    if (ijvmGlobalData.boardId != 0) {
      if (!ijvmioSerialWrite(IO_CONSUMER_JVM, COM_1, result)) {
        return 0;
      }
    }
  }
}
else if (timerIsExpired(enumerationTimer)) {
  timerDelete(enumerationTimer);
  enumerationTimer = TIMER_ID_INVALID;
```

```
    //When timed out, enumeration is done. We deduce the total number of the boards
    //on the chain from this board's ID and the number of the slaves downstream
    ijvmGlobalData.boardsNum = 1 + ijvmGlobalData.slavesNum + ijvmGlobalData.boardId;

    //advance FSM
    state = doneState;
  }
  return 1;
}

#include <msp430x14x.h>
#include <ijvmcpld.h>


//Flag to track the initialization state of the module
static int moduleInitialized = 0;
```

```
//-----------------------------------------------------------------------------
// cpldModuleInit()
// Description: This function initializes the CPLD controller for accelerating
// DIV and MOD
// Inputs: none
// Outputs: int - returns 1 if the initialization is OK, returns 0 if not.
//-----------------------------------------------------------------------------
int cpldModuleInit(void)
{
  if (moduleInitialized) {
    //The module must be initialized only once
    return 1;
  }

  // control lines
  P5SEL |= 0b1111111;
  P5DIR |= 0b111111;
```

```
  // data output
  P4SEL |= 0xFF;
  P4DIR |= 0xFF;

  // data input
  P2SEL |= 0xFF;
  P2DIR = 0x0;

  moduleInitialized = 1; //set the module initialized flag
  return 1;
}

//---------------------------------------------------------------------------
// CPLD acceleration functions
//---------------------------------------------------------------------------
int cpldAluMod(int x, int y)
{
   int result = 0;
```

```
// load X
P4OUT = 0x0f & (x >> 8);
P5OUT |= CPLD_ALU_START | OPCODE_LOAD_X_UPPER;
P5OUT &= ~CPLD_ALU_START;
// load x
P4OUT = (x&0x0F);
P5OUT = CPLD_ALU_START |OPCODE_LOAD_X_LOWER;
P5OUT &= ~CPLD_ALU_START;


// load Y
P4OUT = 0x0f & (y >> 8);
P5OUT |= CPLD_ALU_START | OPCODE_LOAD_Y_UPPER;
P5OUT &= ~CPLD_ALU_START;
// load y
P4OUT = (y&0x0F);
P5OUT = CPLD_ALU_START | OPCODE_LOAD_Y_LOWER;
P5OUT &= ~CPLD_ALU_START;
```

```
    // execute
    P5OUT = CPLD_ALU_START | OPCODE_EXECUTE;
    P5OUT &= ~CPLD_ALU_START;
    // wait
    while(!CPLD_ALU_READY);

    P5OUT = SELECT_REMAINDER_UPPER;
    // read M
    result = P2IN << 8;

    P5OUT = SELECT_REMAINDER_LOWER;
    // read m
    result += P2IN;

    return result;
}

int cpldAluDiv(int x, int y)
```

```
{
    int result = 0;
    // load X
    P4OUT = 0x0f & (x >> 8);
    P5OUT |= CPLD_ALU_START | OPCODE_LOAD_X_UPPER;
    P5OUT &= ~CPLD_ALU_START;
    // load x
    P4OUT = (x&0x0F);
    P5OUT = CPLD_ALU_START |OPCODE_LOAD_X_LOWER;
    P5OUT &= ~CPLD_ALU_START;


    // load Y
    P4OUT = 0x0f & (y >> 8);
    P5OUT |= CPLD_ALU_START | OPCODE_LOAD_Y_UPPER;
    P5OUT &= ~CPLD_ALU_START;
    // load y
    P4OUT = (y&0x0F);
    P5OUT = CPLD_ALU_START | OPCODE_LOAD_Y_LOWER;
```

```
    P5OUT &= ~CPLD_ALU_START;

    // execute
    P5OUT = CPLD_ALU_START | OPCODE_EXECUTE;
    P5OUT &= ~CPLD_ALU_START;
    // wait
    while(!CPLD_ALU_READY);

    P5OUT = SELECT_QUOTIENT_UPPER;
    // read M
    result = P2IN << 8;

    P5OUT = SELECT_QUOTIENT_LOWER;
    // read m
    result += P2IN;

    return result;
}
```

```
// This module provides the familiar file system interface to the underlying

// flash memory. Current implementation limits the number of files to two; the

// maximum sizes of the files, and their start and end addresses in flash must

// specified at compile time. This is transparent to the users of this module,

// though, who only need to specify the file index (equivalent to a file name)

// to open the file, and then use any of the available file IO functions.


#include <filesys.h>

#include <flash.h>

#include <string.h>


//---------------------------------------------------------------------------

// Private file system macros and mappings into the flash memory

//---------------------------------------------------------------------------


// Fraction of FLASH_SEGMENT_SIZE to be used. This is a tradeoff between using
```

```
// more of the flash segment but requiring a large buffer in RAM, and using
// smaller fractions of the available segment size requiring a small RAM buffer.
// Using the whole FLASH_SEGMENT_SIZE is not practical as it would require a
// buffer 1/4 of the total RAM.

#define FILESYS_SEG_SIZE       128

// Total number of segments used for the file system. Must be set so that the
// file system does not overlap with the code in flash.

#define FILESYS_NUM_FSEGS      76

//the segment dedicated for the filesystem registry

#define FILESYS_REGISTRY_FSEG   0xFC00

#define FILESYS_MAGIC_ID       0xABCD
#define FILESYS_MAX_FILES      2
```

```
//the mappings for file ID=0

#define FILE_ID_0_FSEG_NUM      50

#define FILE_ID_0_FSEG_START    0xFA00

#define FILE_ID_0_FSEG_END      0x9800

#define FILE_ID_0_MAX_SIZE      FILE_ID_0_FSEG_NUM * FILESYS_SEG_SIZE


//the mappings for file ID=1

#define FILE_ID_1_FSEG_NUM      26

#define FILE_ID_1_FSEG_START    0x9600

#define FILE_ID_1_FSEG_END      0x6400

#define FILE_ID_1_MAX_SIZE      FILE_ID_1_FSEG_NUM * FILESYS_SEG_SIZE


//---------------------------------------------------------------------------

// Private file system data types

//---------------------------------------------------------------------------


//A file system structure that resides in the flash as part of the
```

//file system registry to describe the file persistently.

```
typedef struct {
  unsigned char fileUsed;           //flag to mark the file space as taken
  unsigned short fileSize;          //current file size
  unsigned short fileSizeMax;       //maximum file size
  unsigned short segmentStartAddr;  //static mapping: file start seg in flash
  unsigned short segmentEndAddr;    //static mapping: file end seg in flash
  unsigned short segmentNumber;     //static mapping: file max segs in flash
}
FlashFileRecord;
```

//A temporary RAM-residing file system structure describing open files

```
typedef struct {
  int fileOpen;                     //flag to mark file a open
  FileOpenMode fileOpenMode;        //current open mode
  unsigned int fileReadIndex;       //current read file pointer
```

```
  unsigned int fileWriteIndex;       //current write file pointer

  unsigned int fileSize;             //current file size

}

FileIDRecord;


//The file system registry structure stored persistently in flash. Holds

//persistent information on the files in the file system. It is stored in a

//dedicated registry segment FILESYS_REGISTRY_FSEG


typedef struct {


  //identifier for a formatted(valid) filesystem

  unsigned short filesysID;


  //persistent file records for each file

  FlashFileRecord fileRecords[FILESYS_MAX_FILES];

}

FlashRegistry;
```

```
//-----------------------------------------------------------------------------
// Private globals
//-----------------------------------------------------------------------------

//a static buffer to cache the data to/from the flash segment
static char flashSegmentBuffer[FILESYS_SEG_SIZE];

//the address of the flash segment that is buffered by the static buffer
static unsigned short flashSegmentAddress;

//a filesys data structure contained in flash (this is a buffer to load/store it)
static FlashRegistry registry;

//a filesys data structure for opened files
static FileIDRecord descriptors[FILESYS_MAX_FILES];

//Flag to track the initialization state of the module
```

```
static int moduleInitialized = 0;


//-----------------------------------------------------------------------------
// filesysFileMaxSize: query the filesystem for the current size limits per file
//   with ID=index.
// arguments: unsigned int fileIndex - file index
// return: unsigned int - max size allowable for that file
//-----------------------------------------------------------------------------
unsigned int filesysFileMaxSize(unsigned int fileIndex)
{
  switch (fileIndex) {
    case 0: return FILE_ID_0_MAX_SIZE;
    case 1: return FILE_ID_1_MAX_SIZE;
    default: return 0;
  }
}


//-----------------------------------------------------------------------------
```

```
// filesysFlushCurrentSegment: private function that reconciles RAM with flash
//   by flushing the currently cached buffer to its associated segment.
// arguments: none
// return: 1 if success, 0 if failure
//---------------------------------------------------------------------------
static int filesysFlushCurrentSegment(void)
{
  if (flashSegmentAddress != 0) {
    //flush the current segment buffer into flash
    if (!flashEraseSegment(flashSegmentAddress) ||
        !flashWriteSegment(flashSegmentAddress, flashSegmentBuffer, FILESYS_SEG_SIZE)) {
      return 0;
    }
    flashSegmentAddress = 0; //mark flash segment buffer as available
  }
  return 1;
}
```

```
//-----------------------------------------------------------------------------
// filesysSetCurrentSegment: private function that caches the new flash segment
//   into the RAM segment buffer
// arguments: unsigned short addr - the start address of the flash segment
// return: 1 if success, 0 if failure
//-----------------------------------------------------------------------------
static int filesysSetCurrentSegment(unsigned short addr)
{
  if (addr != 0) {
    flashSegmentAddress = addr;
    //read the new segment from flash
    if (!flashReadSegment(flashSegmentAddress, flashSegmentBuffer, FILESYS_SEG_SIZE)) {
      return 0;
    }
  }
  return 1;
}
```

```
//----------------------------------------------------------------------------
// filesysModuleInit: initialize current module before using any of its
//   functions. This reads the flash to determine if there's a valid file system.
//   if there is one, then the RAM filesys structures are initialized; if there
//   is no valid file system, then a new one is created.
// arguments: none
// return: 1 if success, 0 if failure
//----------------------------------------------------------------------------
int filesysModuleInit(void)
{
  if (moduleInitialized) {
    return 1;
  }
  moduleInitialized = 1;

  //initialize the low-level flash memory driver
  if (!flashModuleInit()) {
    moduleInitialized = 0;
```

```
  return 0;
}

//read the file system registry to determine if there's a valid file system
if (!flashReadSegment(FILESYS_REGISTRY_FSEG, (char*)&registry, sizeof(registry))) {
  moduleInitialized = 0;
  return 0;
}

if (registry.filesysID != FILESYS_MAGIC_ID) {
  //The filesystem doesn't exist yet, create it

  if (!filesysFormat()) {
    moduleInitialized = 0;
    return 0;
  }
}
```

```
  flashSegmentAddress = 0;


  return 1;
}


//-----------------------------------------------------------------------------
// filesysModuleDeinit: flushes the registry into flash to ensure consistency
//   (just in case).
// arguments: none
// return: 1 if success, 0 if failure
//-----------------------------------------------------------------------------
int filesysModuleDeinit(void)
{
  if (!moduleInitialized) {
    return 0;
  }
  moduleInitialized = 0;
```

```
  //synchronize FS data with flash, to make sure that everything is consistent
  flashWriteSegment(FILESYS_REGISTRY_FSEG, (char*)&registry, sizeof(registry));
  return 1;
}

//-----------------------------------------------------------------------------
// filesysFormat: create the new file system by writing a fresh registry record
//   into flash.
// arguments: none
// return: 1 if success, 0 if failure
//-----------------------------------------------------------------------------
int filesysFormat(void)
{
  unsigned short addr;

  if (!moduleInitialized) {
    return 0;
  }
```

```
flashSegmentAddress = 0;  //invalidate current flash buffer

memset(&registry, 0, sizeof(registry));

//write the file system identifier and static file-to-flash address mappings
registry.filesysID = FILESYS_MAGIC_ID;
registry.fileRecords[0].fileUsed = 0;
registry.fileRecords[0].fileSize = 0;
registry.fileRecords[0].fileSizeMax = FILE_ID_0_MAX_SIZE;
registry.fileRecords[0].segmentStartAddr = FILE_ID_0_FSEG_START;
registry.fileRecords[0].segmentEndAddr = FILE_ID_0_FSEG_END;
registry.fileRecords[0].segmentNumber = FILE_ID_0_FSEG_NUM;
registry.fileRecords[1].fileUsed = 0;
registry.fileRecords[1].fileSize = 0;
registry.fileRecords[1].fileSizeMax = FILE_ID_1_MAX_SIZE;
registry.fileRecords[1].segmentStartAddr = FILE_ID_1_FSEG_START;
registry.fileRecords[1].segmentEndAddr = FILE_ID_1_FSEG_END;
```

```
  registry.fileRecords[1].segmentNumber = FILE_ID_1_FSEG_NUM;

  //Write registry to flash
  if (!flashEraseSegment(FILESYS_REGISTRY_FSEG) ||
      !flashWriteSegment(FILESYS_REGISTRY_FSEG, (char*)&registry, sizeof(registry))) {
    return 0;
  }

  return 1;
}

//-----------------------------------------------------------------------------
// filesysFileOpen: open a file and obtain a handle to be used in further file
//  I/O calls.
// arguments:
//   unsigned int fileIndex - the file index within FS (equivalent to a filename)
//   FileOpenMode mode - the open mode (see filesys.h)
// return: FILE_ID_INVALID if failure, otherwise a valid file handle FileID
```

```
//------------------------------------------------------------------------------
FileID filesysFileOpen(unsigned int fileIndex, FileOpenMode mode)
{
  unsigned int i;
  unsigned int numSegmentsNeeded;
  unsigned short segmentStartAddress;
  unsigned short segmentEndAddress;
  unsigned short addr;

  FileID id = FILE_ID_INVALID;

  if (!moduleInitialized || fileIndex >= FILESYS_MAX_FILES) {
    return FILE_ID_INVALID;
  }
  if (descriptors[fileIndex].fileOpen) {
    //file is already open
    return FILE_ID_INVALID;
  }
```

```
//initialize the corresponding RAM structure
descriptors[fileIndex].fileOpen = 1;        //mark file as open
descriptors[fileIndex].fileOpenMode = mode;   //store the open mode
descriptors[fileIndex].fileReadIndex = 0;     //reset the read file pointer
descriptors[fileIndex].fileWriteIndex = 0;    //reset the write file pointer



if (!registry.fileRecords[fileIndex].fileUsed && mode == OPEN_READ) {
  //fail if the file was open for reading, while it doesn't exist
  descriptors[fileIndex].fileOpen = 0;
  return FILE_ID_INVALID;
}

if ((mode == OPEN_WRITE) || (mode == OPEN_WRITE_THROUGH)) {

  //first time file creation or truncation for WRITE and WRITE_THROUGH modes
  descriptors[fileIndex].fileSize = registry.fileRecords[fileIndex].fileSize = 0;
```

```
//erase all corresponding flash segments in the static mapping
for (addr = registry.fileRecords[fileIndex].segmentStartAddr;
     addr >= registry.fileRecords[fileIndex].segmentEndAddr;
     addr -= FLASH_SEGMENT_SIZE) {

    if (!flashEraseSegment(addr)) {
      descriptors[fileIndex].fileOpen = 0;
      return FILE_ID_INVALID;
    }
}

//mark file space as used
registry.fileRecords[fileIndex].fileUsed = 1;

//update the registry
if (!flashEraseSegment(FILESYS_REGISTRY_FSEG) ||
    !flashWriteSegment(FILESYS_REGISTRY_FSEG, (char*)&registry, sizeof(registry))) {
```

```
      descriptors[fileIndex].fileOpen = 0;
      return FILE_ID_INVALID;
    }

  }
  else {
    //if open in the READ mode, get the existing file size
    descriptors[fileIndex].fileSize = registry.fileRecords[fileIndex].fileSize;
  }

  id = fileIndex;

  return id; //return the handle to the open file
}

//-----------------------------------------------------------------------------
// filesysFileClose: close a file and flush all the pending caches to flash
// arguments: FileID id - the file handle obtained from filesysFileOpen
```

```
// return: 1 on success, 0 on failure
//---------------------------------------------------------------------------
int filesysFileClose(FileID id)
{
  unsigned int fileIndex = id;

  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
      fileIndex >= FILESYS_MAX_FILES ||
      !descriptors[fileIndex].fileOpen) {

    return 0;
  }

  if (descriptors[fileIndex].fileOpenMode != OPEN_WRITE_THROUGH) {

    //Flush the cached version to flash ONLY if the file was not  open in
    //the WRITE_TROUGH mode, where the contents are written through without any
```

```
  //caching anyways.
  if (!filesysFlushCurrentSegment()) {
    return 0;
  }
}

//mark the file a closed
descriptors[fileIndex].fileOpen = 0;

//update its new size in the registry (synchronize RAM FS and registry structures)
registry.fileRecords[fileIndex].fileSize = descriptors[fileIndex].fileSize;

//update the registry in flash
if (!flashEraseSegment(FILESYS_REGISTRY_FSEG) ||
    !flashWriteSegment(FILESYS_REGISTRY_FSEG, (char*)&registry, sizeof(registry))) {
  return 0;
}
return 1;
```

```
}


//------------------------------------------------------------------------------
// filesysFileRead: read one byte from the open file at the current read pointer
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
//   char* data - pointer to hold the read value
// return: 1 on success, 0 on failure (EOF or other)
//------------------------------------------------------------------------------
int filesysFileRead(FileID id, char* data)
{
  unsigned int fileIndex = id;
  unsigned int segmentIndex;
  unsigned int segmentOffset;
  unsigned short newSegmentAddress;


  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
```

```
    fileIndex >= FILESYS_MAX_FILES ||
    !descriptors[fileIndex].fileOpen) {

  return 0;
}

if (descriptors[fileIndex].fileReadIndex >= descriptors[fileIndex].fileSize) {
  //EOF
  return 0;
}

//compute the flash segment index and the offset from the current file pointer
segmentIndex = descriptors[fileIndex].fileReadIndex / FILESYS_SEG_SIZE;
segmentOffset = descriptors[fileIndex].fileReadIndex % FILESYS_SEG_SIZE;

//calculate the new segment address in flash from the segment index
newSegmentAddress = registry.fileRecords[fileIndex].segmentStartAddr - 0x200 * segmentIndex;
```

```
  //read from flash directly (without caching)
  *data = *(char*)(newSegmentAddress + segmentOffset);

  //increment the file read pointer
  descriptors[fileIndex].fileReadIndex++;

  return 1;
}

//-----------------------------------------------------------------------------
// filesysFileReadLine: read a string of characters from the open file until the
//   newline (\r) character into the buffer. The newline is not copied in the
//   buffer. The null (\0) is appended to the buffer. The file pointer points
//   past the newline character when done.
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
//   char buf[] - the target buffer to hold the string
//   bufSize - the size of the buffer
```

```
// return: 1 on success, 0 on failure (EOF or other)
//-----------------------------------------------------------------------------
int filesysFileReadLine(FileID id, char buf[], unsigned int bufSize)
{
  unsigned int i;
  char c;

  for (i = 0; i < (bufSize - 1); i++) {
    if (!filesysFileRead(id, &c)) return 0;
    if (c == '\r') break;
    buf[i] = c;
  }
  buf[i] = '\0';
  return 1;
}

//-----------------------------------------------------------------------------
// filesysFileWrite: write one byte to the open file at the current write pointer
```

```
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
//   char data - the value to be written
// return: 1 on success, 0 on failure
//---------------------------------------------------------------------------
int filesysFileWrite(FileID id, char data)
{
  unsigned int fileIndex = id;
  unsigned int segmentIndex;
  unsigned int segmentOffset;
  unsigned short newSegmentAddress;

  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
      fileIndex >= FILESYS_MAX_FILES ||
      !descriptors[fileIndex].fileOpen) {

    return 0;
```

```
}

//compute the flash segment index and the offset from the current file pointer
segmentIndex = descriptors[fileIndex].fileWriteIndex / FILESYS_SEG_SIZE;
segmentOffset = descriptors[fileIndex].fileWriteIndex % FILESYS_SEG_SIZE;

//calculate the new segment address in flash from the segment index
newSegmentAddress = registry.fileRecords[fileIndex].segmentStartAddr - 0x200*segmentIndex;

//make sure that the file will not exceed the allocated segment space
if (newSegmentAddress < registry.fileRecords[fileIndex].segmentEndAddr) {
  //file is too big;
  return 0;
}

if (descriptors[fileIndex].fileOpenMode == OPEN_WRITE_THROUGH) {
   //if the file has been opened in the WRITE_THROUGH mode, then
   //write directly into the flash
```

```
    flashWriteByte(newSegmentAddress + segmentOffset, (unsigned char)data);
  }
  else {
   //if the file has been opened in the WRITE mode, then buffer all writes

   //check if the write target location falls within currently cached segment
   if (newSegmentAddress != flashSegmentAddress){

    //writeIndex doesn't fall within the current flash segment

    //(1) flush the current cached segment
    if (!filesysFlushCurrentSegment()) {
     return 0;
    }

    // (2) cache the new segment
    if (!filesysSetCurrentSegment(newSegmentAddress)) {
     return 0;
```

```
      }
    }
    //write the value in the buffer
    flashSegmentBuffer[segmentOffset] = data;
  }

  //increment the write pointer
  descriptors[fileIndex].fileWriteIndex++;

  //update the file size in the RAM file descriptor structure
  if (descriptors[fileIndex].fileSize < descriptors[fileIndex].fileWriteIndex)
    descriptors[fileIndex].fileSize = descriptors[fileIndex].fileWriteIndex;

  return 1;
}

//---------------------------------------------------------------------------
// filesysFileSetReadPos: set the read pointer at a specific position in the file
```

```
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
//   unsigned int pos - the new file read position
// return: 1 on success, 0 on failure
//----------------------------------------------------------------------------
int filesysFileSetReadPos(FileID id, unsigned int pos)
{
  unsigned int fileIndex = id;

  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
      fileIndex >= FILESYS_MAX_FILES ||
      !descriptors[fileIndex].fileOpen ||
      pos >= descriptors[fileIndex].fileSize) {

    return 0;
  }
```

```
  descriptors[fileIndex].fileReadIndex = pos;

  return 1;
}

//---------------------------------------------------------------------------
// filesysFileSetWritePos: set the write pointer at a specific position in the file
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
//   unsigned int pos - the new file write position
// return: 1 on success, 0 on failure
//---------------------------------------------------------------------------
int filesysFileSetWritePos(FileID id, unsigned int pos)
{
  unsigned int fileIndex = id;

  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
```

```
    fileIndex >= FILESYS_MAX_FILES ||
    !descriptors[fileIndex].fileOpen) {

   return 0;
  }

  //check if the new file write position would exceed the max allowable file size
  if (pos >= registry.fileRecords[fileIndex].fileSizeMax) return 0;

  descriptors[fileIndex].fileWriteIndex = pos;

  return 1;
}

//----------------------------------------------------------------------------
// filesysFileGetReadPos: get the current read pointer
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
```

```
//   unsigned int *pos - the pointer to hold the current read pointer value
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int filesysFileGetReadPos(FileID id, unsigned int *pos)
{
  unsigned int fileIndex = id;

  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
      fileIndex >= FILESYS_MAX_FILES ||
      !descriptors[fileIndex].fileOpen) {

    return 0;
  }

  *pos = descriptors[fileIndex].fileReadIndex;
  return 1;
}
```

```
//-----------------------------------------------------------------------------
// filesysFileGetWritePos: get the current write pointer
// arguments:
//   FileID id - the file handle obtained from filesysFileOpen
//   unsigned int *pos - the pointer to hold the current write pointer value
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int filesysFileGetWritePos(FileID id, unsigned int *pos)
{
  unsigned int fileIndex = id;

  if (!moduleInitialized ||
      id == FILE_ID_INVALID ||
      fileIndex >= FILESYS_MAX_FILES ||
      !descriptors[fileIndex].fileOpen) {

    return 0;
```

```
  }

  *pos = descriptors[fileIndex].fileWriteIndex;
  return 1;
}

// This module is the driver for the flash memory. It provides routines for
// reading/erasing/writing to/from flash. This is a low-level component. It is
// used by the filesys module.

#include <common.h>
#include <flash.h>

//Flag to track the initialization state of the module
static int moduleInitialized = 0;

//----------------------------------------------------------------------------
// flashModuleInit: initialize the module before calling any of its functions
```

```
// arguments: none
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int flashModuleInit(void)
{
  if (moduleInitialized) {
    return 1;
  }
  moduleInitialized = 1;

  if (initialized) _DINT();
  FCTL2 = FWKEY + FSSEL_1 + FN4;  // MCLK / 17 => 8Mhz / 17
  if (initialized) _EINT();

  return 1;
}

//-----------------------------------------------------------------------------
```

```
// flashWriteByte: fast-write of a single byte at the raw flash address. The
//   target location must be erased beforehand.
// arguments:
//   unsigned short addr - target raw flash address
//   unsigned char data - the byte to be written
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int flashWriteByte(unsigned short addr, unsigned char data)
{
  if (!moduleInitialized) {
    return 0;
  }

  if (initialized) _DINT();

  //----------------------------------------------------------
  // Write new value (the location must be erased beforehand!)
  //----------------------------------------------------------
```

```
  FCTL3 = FWKEY;                    //clear lock
  FCTL1 = FWKEY + WRT;              //enable write
  *((unsigned char *) addr) = data; //write the byte
  FCTL1 = FWKEY;                    //done, clear WRT
  FCTL3 = FWKEY + LOCK;             //done, set lock

  if (initialized) _EINT();

  return 1;
}

//-----------------------------------------------------------------------------
// flashReadSegment: read from the flash segment into a memory buffer. (NOTE:
// this is done for uniform interface, the read from flash is actually
// equivalent to reading from memory). The extra overhead is justified for
// maintainability and interface cleanliness purposes.
//
```

```
// arguments:
//   unsigned short addr - source raw flash segment start address
//   char buffer[] - target RAM buffer
//   unsigned int length - number of bytes to copy from flash to the buffer
// return: 1 on success, 0 on failure
//----------------------------------------------------------------------------
int flashReadSegment(unsigned short addr, char buffer[], unsigned int length)
{
  unsigned int i;

  if (!moduleInitialized || length > FLASH_SEGMENT_SIZE) {
    return 0;
  }

  for (i = 0; i < length; i++) {
    *buffer++ = *((char *)addr++); // Read values from flash
  }
```

```
  return 1;
}

//-----------------------------------------------------------------------------
// flashWriteSegment: read into the flash segment from the memory buffer. The
//   target segment must be erased beforehand by calling flashEraseSegment !
//
// arguments:
//   unsigned short addr - target raw flash segment start address
//   char buffer[] - source RAM buffer
//   unsigned int length - number of bytes to copy from the buffer to flash
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int flashWriteSegment(unsigned short addr, char buffer[], unsigned int length)
{
  unsigned int i;

  if (!moduleInitialized || length > FLASH_SEGMENT_SIZE) {
```

```
  return 0;
}

if (initialized) _DINT();

//---------------------------------------------------
// Write segment
//---------------------------------------------------
FCTL3 = FWKEY;                // Clear Lock bit
FCTL1 = FWKEY + WRT;          // Set WRT bit for write operation

for (i=0; i < length; i++)
{
  *((unsigned char*)addr++) = *buffer++;        // Write value to flash
}

FCTL1 = FWKEY;                // Clear WRT bit
FCTL3 = FWKEY + LOCK;         // Reset LOCK bit
```

```
  if (initialized) _EINT();

  return 1;
}

//---------------------------------------------------------------------------
// flashEraseSegment: Erase the segment starting at 'addr'
// arguments:
//   unsigned short addr - flash segment-to-be-erased start address
// return: 1 on success, 0 on failure
//---------------------------------------------------------------------------
int flashEraseSegment(unsigned short addr)
{
  unsigned int i;

  if (!moduleInitialized) {
    return 0;
```

```
  }

  if (initialized) _DINT();

  //--------------------------------------------------

  // Erase segment

  //--------------------------------------------------

  FCTL3 = FWKEY;                     // Clear Lock bit

  FCTL1 = FWKEY + ERASE;             // Set Erase bit

  *((unsigned short *) addr) = 0xABAB;     // Dummy write to erase segment

  FCTL3 = FWKEY + LOCK;         // Reset LOCK bit

  if (initialized) _EINT();

  return 1;

}
```

```
// This module is the proxy that controls access to the hardware IO functions:
// serial ports and keypad. It works by directing IO streams to the current IO
// consumer: either the virtual machine-level modules (like the console) or the
// ijvm-application level (IJVM applications running inside the virtual machine).

#include <common.h>
#include <ijvm_io.h>
#include <serial.h>
#include <keypad.h>

//Flag to track the initialization state of the module
static int moduleInitialized = 0;

//the current user of the hardware: serial ports and keypad
static IOConsumer currentIOConsumer;

//---------------------------------------------------------------------------
// ijvmioModuleInit: initialize the module before using any of its functions
```

```
// arguments: none
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int ijvmioModuleInit(void)
{
  if (moduleInitialized) {
    return 1;
  }
  moduleInitialized = 1;

  if (!serialModuleInit() || !keypadModuleInit()) {
    moduleInitialized = 0;
    return 0;
  }

  currentIOConsumer = IO_CONSUMER_JVM; //default IO user
  return 1;
}
```

```
//--------------------------------------------------------------------------------
// ijvmioProcessFsm: process the ijvmio proxy module. Needs to be called as often
//   as possible. Currently, it only processes the keypad module.
// arguments: none
// return: 1 on success, 0 on failure
//--------------------------------------------------------------------------------
int ijvmioProcessFsm(void)
{
  if (!moduleInitialized) {
    return 0;
  }

  keypadProcess();
  return 1;
}

//--------------------------------------------------------------------------------
```

```
// ijvmioSetIOConsumer: set the current IO consumer that will have access to the
//   IO facilities.
// arguments: IOConsumer cons
// return: 1 on success, 0 on failure
//---------------------------------------------------------------------------
int ijvmioSetIOConsumer(IOConsumer cons)
{
  if (!moduleInitialized) {
    return 0;
  }
  currentIOConsumer = cons;
  return 1;
}

//---------------------------------------------------------------------------
// proxy functions for serial access:
//   ijvmioSerialWrite
//   ijvmioSerialWriteBuffered
```

```
//   ijvmioSerialRead
//   ijvmioSerialReadString
//   ijvmioSerialWriteString
//   ijvmioSerialFlushReadBuffer
//   ijvmioSerialFlushWriteBuffer
// arguments: see serial.c
// return: see serial.c
//----------------------------------------------------------------------------
int ijvmioSerialWrite(IOConsumer cons, SerialPort port, char c)
{
  if (!moduleInitialized) {
    return 0;
  }
  return (cons == currentIOConsumer) ? serialWrite(port, c) : 0;
}
//----------------------------------------------------------------------------
int ijvmioSerialWriteBuffered(IOConsumer cons, SerialPort port, char c)
{
```

```
  if (!moduleInitialized) {
    return 0;
  }
  return (cons == currentIOConsumer) ? serialWriteBuffered(port, c) : 0;
}
//---------------------------------------------------------------------------
int ijvmioSerialRead(IOConsumer cons, SerialPort port, char *c)
{
  if (!moduleInitialized) {
    return 0;
  }
  return (cons == currentIOConsumer) ? serialRead(port, c) : 0;
}
//---------------------------------------------------------------------------
int ijvmioSerialReadString(IOConsumer cons, SerialPort port, char buffer[], unsigned int bufferSize)
{
  if (!moduleInitialized) {
    return 0;
```

```
  }
  return (cons == currentIOConsumer) ? serialReadString(port, buffer, bufferSize) : 0;
}
//---------------------------------------------------------------------------
int ijvmioSerialWriteString(IOConsumer cons, SerialPort port, char buffer[], unsigned int bufferSize)
{
  if (!moduleInitialized) {
    return 0;
  }
  return (cons == currentIOConsumer) ? serialWriteString(port, buffer, bufferSize) : 0;
}
//---------------------------------------------------------------------------
int ijvmioSerialFlushReadBuffer(IOConsumer cons, SerialPort port)
{
  if (!moduleInitialized) {
    return 0;
  }
  return (cons == currentIOConsumer) ? serialFlushReadBuffer(port) : 0;
```

```
}
//---------------------------------------------------------------------------
int ijvmioSerialFlushWriteBuffer(IOConsumer cons, SerialPort port)
{
  if (!moduleInitialized) {
    return 0;
  }
  return (cons == currentIOConsumer) ? serialFlushWriteBuffer(port) : 0;
}
//---------------------------------------------------------------------------
// proxy function for keypad access:
//   ijvmioKeypadRead
// arguments: see keypad.c
// return: see keypad.c
//---------------------------------------------------------------------------
KeyType ijvmioKeypadRead(IOConsumer cons)
{
  if (!moduleInitialized) {
```

```
   return 0;
  }
  return (cons == currentIOConsumer) ? keypadRead() : KEY_NONE;
}

/* This is the keypad driver module, for a 4x3 keypad. It uses P3[0..3] pins
 * configures as outputs to drive the 4 rows, and P1[1..3] pins configured
 * as inputs from the 3 columns. The inputs are configured in the interrupt
 * mode to enable interrupt-driven detection of key presses and releases.
 * The keypad is structured as an FSM that need to be processed by calling
 * keypadProcess() as often as possible by the caller code. This module
 * uses the timer functionality for debouncing as provided by the timer
 * module. Detected keypresses are stored in a ring buffer by the FSM,
 * and the caller code needs to use keypadRead() function to retrieve the
 * processed keypresses.
 */

#include <msp430x14x.h>
```

```
#include <keypad.h>

#include <timer.h>

#include <common.h>


//Ring buffer to store detected and processed keypresses

static KeyType keyBuffer[KEYPAD_BUFFER_SIZE];


//Read and write indices to the ring buffer

static unsigned int keyReadIndex;

static unsigned int keyWriteIndex;


//Internal FSM states

typedef enum {

  STATE_INIT_WAITING_FOR_PRESS,

  STATE_WAITING_FOR_PRESS,

  STATE_KEY_PRESS_DETECTED,

  STATE_DEBOUNCING_PRESS,

  STATE_SCAN_KEYPAD,
```

```
  STATE_WAITING_FOR_RELEASE,

  STATE_INIT_WAITING_FOR_RELEASE,

  STATE_KEY_RELEASE_DETECTED,

  STATE_DEBOUNCING_RELEASE

}

KeypadModuleState;

static volatile KeypadModuleState moduleState;


//Flag to track the initialization state of the module

static int moduleInitialized = 0;


//Internal masks & macros for the input (column) and output (rows)

//pins. This is specific for the ports used.

#define KEYPAD_P1MASK  0x0E

#define KEYPAD_INT_ENABLE     P1IE |= KEYPAD_P1MASK

#define KEYPAD_INT_DISABLE    P1IE &= ~KEYPAD_P1MASK

#define KEYPAD_INT_RISING     P1IES &= ~KEYPAD_P1MASK

#define KEYPAD_INT_FALLING    P1IES |= KEYPAD_P1MASK
```

```
#define KEYPAD_INT_RESET      P1IFG &= ~KEYPAD_P1MASK  //reset interrupt flags

#define ROW1_MASK   0x01

#define ROW2_MASK   0x02

#define ROW3_MASK   0x04

#define ROW4_MASK   0x08

#define KEYPAD_DISABLE_ROWS(mask) (P3OUT &= ~mask)

#define KEYPAD_ENABLE_ROWS(mask) (P3OUT |= mask)

#define KEYPAD_READ_COLS()  ((KEYPAD_P1MASK & P1IN) >> 1)


//The internal timer object (from the timer module) used for debouncing

static TimerId debounceTimer;


//The internal scancodes to key types lookup table

static KeyType scanCodeToKeyLookupTable[] =

{

  KEY_STAR,

  KEY_0,

  KEY_POUND,
```

```
  KEY_7,
  KEY_8,
  KEY_9,
  KEY_4,
  KEY_5,
  KEY_6,
  KEY_1,
  KEY_2,
  KEY_3
};

//-----------------------------------------------------------------------------
// keypadKeyToChar()
// Description: this function does conversion from the KeyType to the ASCII
// Inputs: KeyType key - the key to be conterted to ASCII
// Outputs: char - the converted ASCII value.
//-----------------------------------------------------------------------------
char keypadKeyToChar(KeyType key)
```

```
{
  char ch = '\0';

  switch(key) {
    case KEY_0:     ch = '0'; break;
    case KEY_1:     ch = '1'; break;
    case KEY_2:     ch = '2'; break;
    case KEY_3:     ch = '3'; break;
    case KEY_4:     ch = '4'; break;
    case KEY_5:     ch = '5'; break;
    case KEY_6:     ch = '6'; break;
    case KEY_7:     ch = '7'; break;
    case KEY_8:     ch = '8'; break;
    case KEY_9:     ch = '9'; break;
    case KEY_STAR:  ch = '*'; break;
    case KEY_POUND: ch = '#'; break;
  }
```

```
  return ch;
}

//----------------------------------------------------------------------------
// getKeyFromBuffer()
// Description: this is an internal function to retrieve the oldest keypress
// from the keypress ring buffer. It is called by keypadRead().
// Inputs: none
// Outputs: KeyType - the retrieved key. If the ring buffer is empty, it returns
// KEY_NONE
//----------------------------------------------------------------------------
static KeyType getKeyFromBuffer(void)
{
  KeyType key = KEY_NONE;

  //Check is the ring buffer is not empty
  if (keyReadIndex != keyWriteIndex) {
```

```
    //Retrieve the oldest keypress
    key = keyBuffer[keyReadIndex];


    //Increment the read index of the ring buffer
    keyReadIndex = (keyReadIndex + 1) % KEYPAD_BUFFER_SIZE;
  }
  return key;
}


//-----------------------------------------------------------------------------
// putKeyToBuffer()
// Description: this is an internal function to put the newes keypress into
// the keypress ring buffer. It is called by keypadProcess() FSM when a new
// keypress is detected.
// Inputs: KeyType key - the new key. If the ring buffer is full, the new key is
// discarded.
// Outputs: none
//-----------------------------------------------------------------------------
```

```
static void putKeyToBuffer(KeyType key)
{
  if (key == KEY_NONE) return;

  if (((keyWriteIndex + 1) % KEYPAD_BUFFER_SIZE) != keyReadIndex) {
    keyBuffer[keyWriteIndex] = key;
    keyWriteIndex = (keyWriteIndex + 1) % KEYPAD_BUFFER_SIZE;
  }
}

//----------------------------------------------------------------------------
// keypadModuleInit()
// Description: this function initializes the keypad module. It must be called
// before any of the module functions are called.
// Inputs: none
// Outputs: int - returns 1 if the initialization is OK, returns 0 if not.
//----------------------------------------------------------------------------
int keypadModuleInit(void)
```

```
{
 if (moduleInitialized) {
  //The module must be initialized only once
  return 1;
 }

 moduleInitialized = 1; //set the module initialized flag

 //Init the ring buffer : reset the ring buffer indices.
 keyReadIndex = 0;
 keyWriteIndex = 0;

 //Initialize the timer module to be able to use the debounce timer
 if(!timerModuleInit() ||
  (debounceTimer = timerCreate()) == TIMER_ID_INVALID){

  moduleInitialized = 0;
  return 0;
```

```
  }

  //Initialize the IO ports for the columns and the rows, and source the rows
  P3DIR |= 0x0F;  //init P3[0..3] as outputs
  KEYPAD_ENABLE_ROWS(ROW1_MASK | ROW2_MASK |ROW3_MASK |ROW4_MASK);
  P1DIR &= ~0x0E;  //init P1[1..3] as inputs

  //Disable the keypress interrupts
  KEYPAD_INT_DISABLE;

  //Initialize FSM
  moduleState = STATE_INIT_WAITING_FOR_PRESS;

  return 1;
}

//-----------------------------------------------------------------------------
// keypadProcess()
```

```
// Description: this is the FSM of the keypad module. It follows the detection
// of a single keypress. It must be called as often as possible to service the
// keypress interrupts. It is non-blocking.
// Inputs: none
// Outputs: int - returns 1 if successful pass through the FSM , returns 0 if not.
//-----------------------------------------------------------------------------
int keypadProcess(void)
{
  unsigned int scanLine;



  if (!moduleInitialized) {
    return 0;
  }

  //Do a single pass of the FSM
  switch (moduleState) {
```

```
//------------------------------------------------------------------------
// Prepare to wait for a new keypress
//------------------------------------------------------------------------
case STATE_INIT_WAITING_FOR_PRESS:

  //Turn on all the four rows, enable the rising edge interrupts to detect
  //a new keypress
  KEYPAD_ENABLE_ROWS(ROW1_MASK | ROW2_MASK |ROW3_MASK |ROW4_MASK);
  KEYPAD_INT_RISING;
  KEYPAD_INT_RESET;
  KEYPAD_INT_ENABLE;
  moduleState = STATE_WAITING_FOR_PRESS;
  break;

//------------------------------------------------------------------------
// Waiting for a new keypress. When a keypress occurs, the port IO ISR will
// advance the FSM to the next state, STATE_KEY_PRESS_DETECTED
//------------------------------------------------------------------------
```

```
case STATE_WAITING_FOR_PRESS:
  break;

//------------------------------------------------------------------------
// The ISR has detected a keypress. Now to debounce it we start the debounce
// timer.
//------------------------------------------------------------------------
case STATE_KEY_PRESS_DETECTED:
  timerStart(debounceTimer, 40);
  moduleState = STATE_DEBOUNCING_PRESS;
  break;

//------------------------------------------------------------------------
// Wait for the debounce timer to expire. Then proceed to scan the keypad
//------------------------------------------------------------------------
case STATE_DEBOUNCING_PRESS:
  if (timerIsExpired(debounceTimer)) {
    moduleState = STATE_SCAN_KEYPAD;
```

```
  }
  break;

//------------------------------------------------------------------------
// The detected key has been debounced. Now scan the keypad by activating
// each row in turn and reading the columns to form a 3x4=12 bit scan code
//------------------------------------------------------------------------
case STATE_SCAN_KEYPAD:
  {
    KeyType newKey = KEY_NONE;
    int i;

    // First turn off all the rows
    KEYPAD_DISABLE_ROWS(ROW1_MASK | ROW2_MASK | ROW3_MASK | ROW4_MASK);

    //Scan each row in turn. The scancode will contain a '1' for each key
    //that is down a '0' for each key that is not down.
```

```
scanLine = 0; //the scan code
KEYPAD_ENABLE_ROWS(ROW1_MASK);
scanLine |= KEYPAD_READ_COLS();
KEYPAD_DISABLE_ROWS(ROW1_MASK);

scanLine <<= 3;
KEYPAD_ENABLE_ROWS(ROW2_MASK);
scanLine |= KEYPAD_READ_COLS();
KEYPAD_DISABLE_ROWS(ROW2_MASK);

scanLine <<= 3;
KEYPAD_ENABLE_ROWS(ROW3_MASK);
scanLine |= KEYPAD_READ_COLS();
KEYPAD_DISABLE_ROWS(ROW3_MASK);

scanLine <<= 3;
KEYPAD_ENABLE_ROWS(ROW4_MASK);
scanLine |= KEYPAD_READ_COLS();
```

```
    KEYPAD_DISABLE_ROWS(ROW4_MASK);

    //Lookup the key corresponding to the scancode, up to the first match
    for (i = 0; i < DIM (scanCodeToKeyLookupTable); i++) {
      //check each key button bit in the scan code
      if(scanLine & 0x01) {
        newKey = scanCodeToKeyLookupTable[i];
        break;
      }
      scanLine >>= 1;
    }

    putKeyToBuffer(newKey); //Add the new key to the ring buffer
    moduleState = STATE_INIT_WAITING_FOR_RELEASE;
  }
  break;

//-----------------------------------------------------------------------
```

```
// Prepare to wait for the key release
//-----------------------------------------------------------------------
case STATE_INIT_WAITING_FOR_RELEASE:
  //Turn on all the four rows, enable the falling edge interrupts to detect
  //a new key release
  KEYPAD_ENABLE_ROWS(ROW1_MASK | ROW2_MASK | ROW3_MASK | ROW4_MASK);
  KEYPAD_INT_FALLING;
  KEYPAD_INT_RESET;
  KEYPAD_INT_ENABLE;
  moduleState = STATE_WAITING_FOR_RELEASE;
  break;

//-----------------------------------------------------------------------
// Waiting for a new keyrelease. When a keyrelease occurs, the port IO ISR
// will advance the FSM to the next state, STATE_KEY_RELEASE_DETECTED
//-----------------------------------------------------------------------
case STATE_WAITING_FOR_RELEASE:
  break;
```

```
//------------------------------------------------------------------------
// The ISR has detected a keyprelease. Now to debounce it we start the
// debounce timer.
//------------------------------------------------------------------------
case STATE_KEY_RELEASE_DETECTED:
  timerStart(debounceTimer, 40);
  moduleState = STATE_DEBOUNCING_RELEASE;
  break;

//------------------------------------------------------------------------
// Wait for the debounce timer to expire. Then go back to waiting for the
// new keypress. NOTE: no need to do a second scan to check that the key is
// actually released. Tested thoroughly.
//------------------------------------------------------------------------
case STATE_DEBOUNCING_RELEASE:
  if (timerIsExpired(debounceTimer)) {
    moduleState = STATE_INIT_WAITING_FOR_PRESS;
```

```
      }
      break;
    }
  return 1;
}

//-----------------------------------------------------------------------------
// keypadRead()
// Description: this function returns a new key from the keypress ring buffer,
// if any press has been detected. It is non-blocking.
// Inputs: none
// Outputs: KeyType - returns the new key, or KEY_NONE if no key has been pressed
//-----------------------------------------------------------------------------
KeyType keypadRead(void)
{
  if (!moduleInitialized) {
    return KEY_NONE;
  }
```

```
  //try to get the key from the key ring buffer
  return getKeyFromBuffer();
}

//-----------------------------------------------------------------------------
// keypadISR()
// Description: this is the ISR that gets called for each keypress or keyrelease.
// It advances the module FSM.
// Inputs: none
// Outputs: none
//-----------------------------------------------------------------------------
void keypadISR(void)
{
  //Disable further keypad interrupts, until reenabled by the FSM
  KEYPAD_INT_DISABLE;
  //Clear the interrupt flags
  KEYPAD_INT_RESET;
```

```
if (!moduleInitialized) {
  return;
}


switch (moduleState) {

  //------------------------------------------------------------------------
  //The FSM is currently waiting for a keypress. A rising edge interrupt must
  //have occured. Notify and advance the FSM for further processing.
  //------------------------------------------------------------------------
  case STATE_WAITING_FOR_PRESS:
    moduleState = STATE_KEY_PRESS_DETECTED;
    break;

  //------------------------------------------------------------------------
  //The FSM is currently waiting for a keyrelease. A falling edge interrupt
```

```
    //must have occured. Notify and advance the FSM for further processing.
    //------------------------------------------------------------------------
    case STATE_WAITING_FOR_RELEASE:
      moduleState = STATE_KEY_RELEASE_DETECTED;
      break;



  }
}
/* This is the uart driver module, for MSP430. It uses UART0 with buffered
 * interrupt-driven reception and transmission. Current UART settings are
 * fixed to 19200-N-8, but can be easily made reconfigurable.
 */

#include <msp430x14x.h>
#include <serial.h>

//Private serial module variables
```

```
typedef struct {
  char txBuffer[SERIAL_TX_BUFFER_SIZE];  //the transmit ring FIFO
  volatile int txReadIndex;            //the transmit FIFO read pointer
  volatile int txWriteIndex;           //the transmit FIFO write pointer
  volatile int txNumPending;           //the number of chars in tx FIFO
  char rxBuffer[SERIAL_RX_BUFFER_SIZE];  //the receive ring FIFO
  volatile int rxReadIndex;            //the receive FIFO read pointer
  volatile int rxWriteIndex;           //the receive FIFO write pointer
  volatile int rxNumPending;           //the number of chars in rx FIFO
}
SerialPortData;

static SerialPortData ports[2]; //two ports: COM_1 and COM_2

//Private FIFO macros
#define TX_BUFFER_EMPTY(port) (port.txNumPending == 0)
#define RX_BUFFER_EMPTY(port) (port.rxNumPending == 0)
#define TX_BUFFER_FULL(port) (port.txNumPending == (SERIAL_TX_BUFFER_SIZE - 1))
```

```
#define RX_BUFFER_FULL(port) (port.rxNumPending == (SERIAL_RX_BUFFER_SIZE - 1))

//Flag to track the initialization state of the module
static int moduleInitialized = 0;

//------------------------------------------------------------------------------
// serialModuleInit()
// Description: this function initializes the serial module. It must be called
// before any of the module functions are called. It initializes the UART
// registers for the communication parameters, sets the UART clock, and
// initializes the GPIO pins used as RX and TX.
// Inputs: none
// Outputs: int - returns 1 if the initialization is OK, returns 0 if not.
//------------------------------------------------------------------------------
int serialModuleInit(void)
{
  if (moduleInitialized) {
    return 1;
```

```
}
moduleInitialized = 1;

//Reset the rx & tx queue pointers
ports[COM_1].txReadIndex = 0;
ports[COM_1].txWriteIndex = 0;
ports[COM_1].rxReadIndex = 0;
ports[COM_1].rxWriteIndex = 0;
ports[COM_1].txNumPending = 0;
ports[COM_1].rxNumPending = 0;
ports[COM_2].txReadIndex = 0;
ports[COM_2].txWriteIndex = 0;
ports[COM_2].rxReadIndex = 0;
ports[COM_2].rxWriteIndex = 0;
ports[COM_2].txNumPending = 0;
ports[COM_2].rxNumPending = 0;

P3SEL |= 0xF0;  //Select the alt finction (UART0&1) for GPIO pins P3.4 to P3.7
```

```
P3DIR |= 0x50;  //Select the GPIO directions, TX is out, RX is in

//Setup the UART0

U0CTL |= SWRST;   //hold reset while reconfiguring
U0CTL |= CHAR;    //8-N-1 format
U0TCTL |= SSEL1;  //SMCLK = 8MHz
U0BR1 = 0x01;     //the baud rate 19200 settings
U0BR0 = 0xA0;     //the baud rate 19200 settings
U0MCTL = 0;       //the baud rate 19200 settings

U0ME |= UTXE0;    //enable the transmit module
U0ME |= URXE0;    //enable the receive module
U0CTL &= ~SWRST;  //release reset while reconfiguring

//Setup the UART1

U1CTL |= SWRST;   //hold reset while reconfiguring
```

```
  U1CTL |= CHAR;    //8-N-1 format
  U1TCTL |= SSEL1;  //SMCLK = 8MHz
  U1BR1 = 0x01;     //the baud rate 19200 settings
  U1BR0 = 0xA0;     //the baud rate 19200 settings
  U1MCTL = 0;       //the baud rate 19200 settings

  U1ME |= UTXE1;    //enable the transmit module
  U1ME |= URXE1;    //enable the receive module
  U1CTL &= ~SWRST;  //release reset while reconfiguring

  return 1;
}

//-----------------------------------------------------------------------------
// serialWrite()
// Description: this function writes the charecter to the transmit queue and
// forces the transmit interrupt to call ISR.
// Inputs: SerialPort port {COM_1, COM_2}, char c
```

```
// Outputs: int - returns 1 if when successful, returns 0 if not.
//---------------------------------------------------------------------------
int serialWrite(SerialPort port, char c)
{
  if (!moduleInitialized) {
    return 0;
  }

  if (TX_BUFFER_FULL(ports[port])) {
    switch(port) {
      case COM_1: IE1 &= ~UTXIE0; break;
      case COM_2: IE2 &= ~UTXIE1; break;
    }
    return 0;
  }

  ports[port].txBuffer[ports[port].txWriteIndex] = c;
  ports[port].txWriteIndex = (ports[port].txWriteIndex + 1) % SERIAL_TX_BUFFER_SIZE;
```

```
ports[port].txNumPending++;


switch(port) {
   case COM_1:
     while ((IFG1 & UTXIFG0) == 1);
     while (!(U0TCTL&TXEPT));
     IFG1 |= UTXIFG0;  //this forces a tx interrupt
     IE1 |= UTXIE0;
     break;


   case COM_2:
     while ((IFG2 & UTXIFG1) == 1);
     while (!(U1TCTL&TXEPT));
     IFG2 |= UTXIFG1;  //this forces a tx interrupt
     IE2 |= UTXIE1;
     break;
}
```

```
  return 1;
}


//---------------------------------------------------------------------------
// serialWriteBuffered()
// Description: this function writes the charecter to the transmit queue, but
//   forces the transmit interrupt to call ISR _ONLY_ when the queue is full
//   (unlike serialWrite). To make sure that the data is completely transfered,
//   the transmit buffer needs to be flushed with serialFlushWriteBuffer when
//   finished.
// Inputs: SerialPort port {COM_1, COM_2}, char c
// Outputs: int - returns 1 if when successful, returns 0 if not.
//---------------------------------------------------------------------------
int serialWriteBuffered(SerialPort port, char c)
{
  if (!moduleInitialized) {
    return 0;
  }
```

```
if (TX_BUFFER_FULL(ports[port])) {
  switch(port) {
    case COM_1: IE1 &= ~UTXIE0; break;
    case COM_2: IE2 &= ~UTXIE1; break;
  }
  return 0;
}

ports[port].txBuffer[ports[port].txWriteIndex] = c;
ports[port].txWriteIndex = (ports[port].txWriteIndex + 1) % SERIAL_TX_BUFFER_SIZE;
ports[port].txNumPending++;

if (TX_BUFFER_FULL(ports[port])) {
  switch(port) {
    case COM_1:
      while ((IFG1 & UTXIFG0) == 1);
      while (!(U0TCTL&TXEPT));
```

```
      IFG1 |= UTXIFG0;  //this forces a tx interrupt
      IE1 |= UTXIE0;
      break;

    case COM_2:
      while ((IFG2 & UTXIFG1) == 1);
      while (!(U1TCTL&TXEPT));
      IFG2 |= UTXIFG1;  //this forces a tx interrupt
      IE2 |= UTXIE1;
      break;
   }
  }

  return 1;
}
//----------------------------------------------------------------------------
// serialWriteString()
// Description: this function copies the charecter string to the transmit
```

```
// queue and forces the transmit interrupt to call ISR.
// Inputs: SerialPort port - serial port
//         char buffer[] - character buffer to transmit
//         unsigned int bufferSize - the size of buffer to transmit
// Outputs: int - returns 1 if when successful, returns 0 if not.
//-----------------------------------------------------------------------------
int serialWriteString(SerialPort port, char buffer[], unsigned int bufferSize)
{
  int i;
  if (!moduleInitialized) {
    return 0;
  }

  if (TX_BUFFER_FULL(ports[port])) {
    return 0;
  }

  if ((SERIAL_TX_BUFFER_SIZE - ports[port].txNumPending - 1) < bufferSize) {
```

```
  return 0;
}
for (i = 0; i < bufferSize; i++) {
  ports[port].txBuffer[ports[port].txWriteIndex] = buffer[i];
  ports[port].txWriteIndex = (ports[port].txWriteIndex + 1) % SERIAL_TX_BUFFER_SIZE;
  ports[port].txNumPending++;
}

switch(port) {
   case COM_1:
     while ((IFG1 & UTXIFG0) == 1);
     while (!(U0TCTL&TXEPT));
     IFG1 |= UTXIFG0;  //this forces a tx interrupt
     IE1 |= UTXIE0;
     break;

   case COM_2:
     while ((IFG2 & UTXIFG1) == 1);
```

```
      while (!(U1TCTL&TXEPT));
      IFG2 |= UTXIFG1;  //this forces a tx interrupt
      IE2 |= UTXIE1;
      break;
  }
  return 1;
}

//----------------------------------------------------------------------------
// serialRead()
// Description: this function reads the charecter from the receive queue.
// Inputs:  SerialPort port - serial port
//          char *c - the pointer to hold the received character.
// Outputs: int - returns 1 if when successful, returns 0 if not(nothing received).
//----------------------------------------------------------------------------
int serialRead(SerialPort port, char *c)
{
  if (!moduleInitialized) {
```

```
    return 0;
  }

  switch(port) {
    case COM_1: IE1 |= URXIE0; break;
    case COM_2: IE2 |= URXIE1; break;
  }

  if (RX_BUFFER_EMPTY(ports[port])) {
    return 0;
  }

  *c = ports[port].rxBuffer[ports[port].rxReadIndex];
  ports[port].rxReadIndex = (ports[port].rxReadIndex + 1) % SERIAL_RX_BUFFER_SIZE;
  ports[port].rxNumPending--;

  return 1;
}
```

```
//----------------------------------------------------------------------------
// serialFlushReadBuffer()
// Description: this function flushes (clears) the receive queue
// Inputs: SerialPort port - serial port
// Outputs: int - returns 1 if when successful, returns 0 if not.
//----------------------------------------------------------------------------
int serialFlushReadBuffer(SerialPort port)
{
  if (!moduleInitialized) {
    return 0;
  }
  switch(port) {
    case COM_1: IE1 &= ~URXIE0; break;
    case COM_2: IE2 &= ~URXIE1; break;
  }

  ports[port].rxReadIndex = ports[port].rxWriteIndex;
```

```
  ports[port].rxNumPending = 0;

  switch(port) {
    case COM_1: IE1 |= URXIE0; break;
    case COM_2: IE2 |= URXIE1; break;
  }

  return 1;
}

//-----------------------------------------------------------------------------
// serialFlushReadBuffer()
// Description: this function flushes (forces the transmission of) all characters
// in the transmit queue.
// Inputs: SerialPort port - serial port
// Outputs: int - returns 1 if when successful, returns 0 if not.
//-----------------------------------------------------------------------------
int serialFlushWriteBuffer(SerialPort port)
```

```
{
  if (!moduleInitialized) {
    return 0;
  }

  if (TX_BUFFER_EMPTY(ports[port])) {
    return 1;
  }
  _DINT(); //TEST

  while(ports[port].txNumPending > 0) {
    switch(port) {
      case COM_1:
        while ((IFG1 & UTXIFG0) == 0);
        U0TXBUF = ports[port].txBuffer[ports[port].txReadIndex];
        break;

      case COM_2:
```

```
        while ((IFG2 & UTXIFG1) == 0);
        U1TXBUF = ports[port].txBuffer[ports[port].txReadIndex];
        break;
    }

    ports[port].txReadIndex = (ports[port].txReadIndex + 1) % SERIAL_TX_BUFFER_SIZE;
    ports[port].txNumPending--;
  }
  _EINT(); //TEST

  return 1;
}

//-----------------------------------------------------------------------------
// serialModuleDeinit()
// Description: this function deinitializes the serial module. It disables RX&TX
// interrupts and stops the UART
// Inputs: none
```

```
// Outputs: int - returns 1 (deinitialization is always OK)
//---------------------------------------------------------------------------
int serialModuleDeinit(void)
{
  if (!moduleInitialized) {
    return 1;
  }
  moduleInitialized = 0;

  while (!(U0TCTL&TXEPT));
  IE1 &= ~URXIE0;
  IE1 &= ~UTXIE0;
  U0CTL |= SWRST;
  U0ME &= ~UTXE0;
  U0ME &= ~URXE0;

  while (!(U1TCTL&TXEPT));
  IE2 &= ~URXIE1;
```

```
  IE2 &= ~UTXIE1;

  U1CTL |= SWRST;

  U1ME &= ~UTXE1;

  U1ME &= ~URXE1;

  return 1;

}


//-----------------------------------------------------------------------------

// serialUart0TxISR()

// Description: this is the uart0 transmit ISR. Every time it is called it

// transmits

// a character from the transmit queue. If there's nothing to transmit it

// disables further transmit interrupts

// Inputs: none

// Outputs: none

//-----------------------------------------------------------------------------

void  serialUart0TxISR(void)

{
```

```
  if (moduleInitialized) {

    if (!TX_BUFFER_EMPTY(ports[COM_1])) {
      U0TXBUF = ports[COM_1].txBuffer[ports[COM_1].txReadIndex];  //transmit one character from TX queue

      //Increment the TX ring queue pointer
      ports[COM_1].txReadIndex = (ports[COM_1].txReadIndex + 1) % SERIAL_TX_BUFFER_SIZE;
      ports[COM_1].txNumPending--;
    }
    else{
      IE1 &= ~UTXIE0; //disable further TX interrupts
    }
  }
}

//-----------------------------------------------------------------------------
// serialUart0RxISR()
// Description: this is the uart0 receive ISR. Every time it is called it writes a
```

```
// received character from the UART into the receive queue. If the receive
// buffer is full the new character is discarded, and further receive interrupts
// are disabled.
// Inputs: none
// Outputs: none
//-----------------------------------------------------------------------------
void  serialUart0RxISR(void)
{
  if (moduleInitialized) {

    if(!RX_BUFFER_FULL(ports[COM_1])) {
      ports[COM_1].rxBuffer[ports[COM_1].rxWriteIndex] = U0RXBUF;
      ports[COM_1].rxWriteIndex = (ports[COM_1].rxWriteIndex + 1) % SERIAL_RX_BUFFER_SIZE;
      ports[COM_1].rxNumPending++;
    }
    else {
      IE1 &= ~URXIE0; //disable further RX interrupts
    }
```

```
  }
}

//---------------------------------------------------------------------------
// serialUart1TxISR()
// Description: this is the uart1 transmit ISR. Every time it is called it
// transmits
// a character from the transmit queue. If there's nothing to transmit it
// disables further transmit interrupts
// Inputs: none
// Outputs: none
//---------------------------------------------------------------------------
void  serialUart1TxISR(void)
{
  if (moduleInitialized) {

    if (!TX_BUFFER_EMPTY(ports[COM_2])) {
      U1TXBUF = ports[COM_2].txBuffer[ports[COM_2].txReadIndex];  //transmit one character from TX queue
```

```
      //Increment the TX ring queue pointer
      ports[COM_2].txReadIndex = (ports[COM_2].txReadIndex + 1) % SERIAL_TX_BUFFER_SIZE;
      ports[COM_2].txNumPending--;
    }
    else{
      IE2 &= ~UTXIE1; //disable further TX interrupts
    }
  }
}

//----------------------------------------------------------------------------
// serialUart1RxISR()
// Description: this is the uart1 receive ISR. Every time it is called it writes a
// received character from the UART into the receive queue. If the receive
// buffer is full the new character is discarded, and further receive interrupts
// are disabled.
// Inputs: none
```

```
// Outputs: none
//----------------------------------------------------------------------------
void  serialUart1RxISR(void)
{
  if (moduleInitialized) {

    if(!RX_BUFFER_FULL(ports[COM_2])) {
      ports[COM_2].rxBuffer[ports[COM_2].rxWriteIndex] = U1RXBUF;
      ports[COM_2].rxWriteIndex = (ports[COM_2].rxWriteIndex + 1) % SERIAL_RX_BUFFER_SIZE;
      ports[COM_2].rxNumPending++;
    }
    else {
      IE2 &= ~URXIE1; //disable further RX interrupts
    }
  }
}

#ifndef __compiler
```

```
#define  __compiler

#include "myGlobalDefines.h"

int value(char *token);
int parseTokens(char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE], int tokenCount, int *codeIndex, char *isLabel, int *offset);
int searchSymbol(char symbol[MAX_TOKEN_SIZE], struct Symbol *table, int symbolCount);

int AddSymbol(char* name, int address, struct Symbol *table, int *tableIndex);

int ReplaceLabels(struct Symbol table[MAX_CONSTANTS], char tableSize);

int BuildMethodTable(struct Symbol table[MAX_CONSTANTS]);
int WriteExecutable(unsigned short addr, unsigned char data);


int BuildLabelTable(struct Symbol table[MAX_CONSTANTS]);
int BuildConstantTable(struct Symbol table[MAX_CONSTANTS]);
```

```
int ReplaceConstants(struct Symbol table[MAX_CONSTANTS], char tableSize);

int ReplaceMethods(struct Symbol table[MAX_CONSTANTS], char tableSize);

int ReplaceLabels(struct Symbol table[MAX_CONSTANTS], char tableSize);

int ReplaceVariables(struct Symbol table[MAX_CONSTANTS], char tableSize);

int BuildVariableTable(struct Symbol table[MAX_CONSTANTS]);


int parseConstants(void);

int parseMethods(void);

int parseLabels(void);

int parseVariables(void);


int parseTokens(char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE], int tokenCount, int *codeIndex, char *isLabel, int *offset);

int value(char *token);


int searchSymbol(char symbol[MAX_TOKEN_SIZE], struct Symbol *table, int symbolCount);

int AddSymbol(char* name, int address, struct Symbol *table, int *tableIndex);


#endif
```

```
#ifndef __mparser

#define __mparser


#include "myGlobalDefines.h"


//this functions splits a line of IJVM code into tokens

//For methods, is returns each parameter as a token

int parse(char *line, char *delimiters, char strings[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE]);

#endif


#include <stdio.h>

#include <string.h>

#include "myGlobalDefines.h"

#include "parser.h"


//this functions splits a line of IJVM code into tokens

//For methods, is returns each parameter as a token
```

```
int parse(char *line, char *delimiters, char strings[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE]){

        int pCount = 0;
        char *tmp, *uncomment;

        //remove comments
        if (line != NULL){
                if (strlen(line) > 1){
                        /*if ( uncomment = strstr(line, "//")){
                                uncomment[0] = '\0';
             }*/
          for (tmp = line; *tmp != '\0'; tmp++) {
           if (*tmp == '/' && *(tmp + 1) == '/') {
            *tmp = '\0';
            break;
           }
          }
         }
```

```
        if (strlen(line) > 0){
            tmp = line;
            pCount = 0;
            strcpy(tmp, line);
            strtok(tmp, delimiters);

            while ( tmp != NULL && pCount < MAX_TOKEN_COUNT){
                if ( sscanf(tmp, "%s", strings[pCount]) > 0 ){

                    pCount++;
                    }
                tmp = strtok(NULL, delimiters);
                }
            }
        }
    return pCount;
}
```

```
#include <ijvmcompiler.h>

#include <common.h>

#include <stdio.h>

#include <string.h>

#include <filesys.h>

#include <ijvm.h>

#include "myGlobalDefines.h"

#include "compiler.h"

#include "parser.h"


//char code[500];

static int codeIndex = 0;   //index that keeps track of the current position in COMPILED code

static int codeSize;

static int inConst = 0;     //flags

static int inMethod = 0;

static int inVar = 0;


static FileID srcFile = FILE_ID_INVALID;
```

```
static FileID execFile = FILE_ID_INVALID;

static CompilerError lastErrorId;


CompilerError ijvmcompilerGetLastError(void)

{

  return lastErrorId;

}



//the compiler is a multi-pass compiler

//The reason to make is multi-pass is to fit everything in the available RAM memory

//Thus, we had to trade speed (we perform more operations) for space.


//it makes 4 passes: for labels, methods, constants and variables

//on each pass it replaces symbols with their value or offset in memory

//constants are replaced with their value

//variables are replaced with their offset from the current frame pointer

//methods are replaced with their relative address in memory and
```

//the number of local variables and parameters passed is added at the method location

//to ease the work at run-time

//jumps to labels (conditional and unconditional jumps) are replaced with an offset from the

//current position - the offset can be positive or negative

//Syntax Checking

//There is syntax checking at 4 levels:

//1.TokenParser reports errors if any given line of input is not

//conforming to the laguage syntax

//2.The Value function returns an error if a value fed to an instruction is malformed

//3. Each pass checks the existance of the symbols it tries to retrieve/add to the

//symbol table and generates two possible errors: symbol not defined/symbol already exists

//4. Scope checks: the compiler checks that no variables are defined

//outside the .var section, methods are correctly formed, etc

//Limits

//The compiler is limited in the number of constants per program,

//methods per program, variables per method (main is considered a method)

```
//labels per method using the MAX_CONSTANTS and MAX_METHODS defines

//These can be easily changed depending on the size of RAM available.


//The maximum size of a string token (variable name, method name, constant name,

//label name, keyword) is defined by MAX_TOKEN_SIZE

//This can also be adjusted depending on the size of RAM available


//Given the value of MAX_CONSTANTS = MAX_METHODS = 15 and

//MAX_TOKEN_SIZE = 15, the compiler uses about 200 bytes of RAM for each pass




int ijvmcompilerRun(void)

{

  char line[MAX_LINE];

  char strings[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];


  lastErrorId = CE_UNKNOWN_ERROR;    //initialize the default error ID
```

```
srcFile = FILE_ID_INVALID;    //source file initialization
execFile = FILE_ID_INVALID;   //compiled file initialization

srcFile = filesysFileOpen(IJVM_APP_FILE_INDEX, OPEN_READ);
if (srcFile == FILE_ID_INVALID) goto error;

execFile = filesysFileOpen(IJVM_EXEC_FILE_INDEX, OPEN_WRITE);
if (execFile == FILE_ID_INVALID) goto error;

parseConstants();
codeIndex = 0;
if (!filesysFileSetWritePos(execFile, 0)) goto error;
parseMethods();
codeIndex = 0;
if (!filesysFileSetWritePos(execFile, 0)) goto error;
parseLabels();
codeIndex = 0;
```

```
  if (!filesysFileSetWritePos(execFile, 0)) goto error;
  parseVariables();
  codeIndex = 0;
  if (!filesysFileSetWritePos(execFile, 0)) goto error;

  while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){
    parse(line, CODE_DELIMITERS, strings);
  }

  if (!filesysFileClose(srcFile)) goto error;
  srcFile = FILE_ID_INVALID;
  if (!filesysFileClose(execFile)) goto error;
  execFile = FILE_ID_INVALID;

  lastErrorId = CE_SUCCESS;
  return 1; //success

error:
```

```
  if (srcFile != FILE_ID_INVALID) filesysFileClose(srcFile);

  if (execFile != FILE_ID_INVALID) filesysFileClose(execFile);

  return 0;

}


//function that allows us to randomly write in FLASH

//the compiler file

int WriteExecutable(unsigned short addr, unsigned char data)

{

  if (!filesysFileSetWritePos(execFile, addr) ||

      !filesysFileWrite(execFile, data)) {


    return 0;

  }

  return 1;

}


//build a table of all the methods
```

```
//remembering the address and the name of each method

int BuildMethodTable(struct Symbol table[MAX_CONSTANTS])

{

	char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];

	char line[MAX_LINE];

	char tokenCount = 0, isLabel = 0, opcode = -1, locals = 0;

	int size = 0, offset = 0;


	if (!filesysFileSetReadPos(srcFile, 0)) return 0;


	while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){

		if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){

			opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);

			if (opcode == METHOD){

				inMethod = 1;


				if (tokenCount > 1)	//not main

		    WriteExecutable(codeIndex - 2, tokenCount - 2);	//local params saved in code
```

```
                else
        WriteExecutable(codeIndex - 2, 0);   //no local params for main


                AddSymbol(tokens[1], codeIndex - 2, table, &size);
            } else if (opcode == ENDMETHOD ){
                inMethod = 0;
    locals = 0;
            } else if (opcode == ENDVAR ){
                WriteExecutable(codeIndex - 1, locals);
    inVar = 0;
            } else if (inMethod) {                 //we're inside a method
                if (inVar && (tokenCount == 1))
                    locals++;
                //note: order of these two IFs matters
                if (opcode == VAR)
                    inVar = 1;
            }
        }
```

```
    }
    return size;
}



int BuildLabelTable(struct Symbol table[MAX_CONSTANTS])
{
    unsigned int savedReadPos;
    char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
    char line[MAX_LINE];
    char tokenCount = 0, isLabel = 0, opcode = -1, locals = 0;
    int size = 0, offset = 0, savedCodeIndex = 0;

    if (!filesysFileSetReadPos(srcFile, 0)) return 0;

  while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){
        if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
```

```
        opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);

        if (opcode == METHOD){
                inMethod = 1;
        } else if (opcode == ENDMETHOD ){

                if (!filesysFileGetReadPos(srcFile, &savedReadPos)) return 0;
                savedCodeIndex = codeIndex;

                //we need to replace the local labels
if (!filesysFileSetReadPos(srcFile, 0)) return 0;

                codeIndex = 0;
if (!filesysFileSetWritePos(execFile, codeIndex)) return 0;
//as soon as we reach the end of a method we replace the jumps with
//the label offset
//This helps both for syntax checking: just labels defined
//in the local scope will exist in the table
```

```
                //and we keep the table small
                ReplaceLabels(table, size);



                        if (!filesysFileSetReadPos(srcFile, savedReadPos)) return 0;
                        codeIndex = savedCodeIndex;

                if (!filesysFileSetWritePos(execFile, codeIndex)) return 0;

                        inMethod = 0;
                        size = 0;//reset the table of labels
                    } else if (inMethod){
                        if (isLabel){
                            AddSymbol(tokens[0], codeIndex - offset, table, &size);
                        }
                    }
            }
    }
```

```
	return 1;
}

int BuildConstantTable(struct Symbol table[MAX_CONSTANTS])
{
		char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
		char line[MAX_LINE];
		char tokenCount = 0, isLabel = 0, opcode;
		char size = 0;
		int offset = 0;

		if (!filesysFileSetReadPos(srcFile, 0)) return 0;

		while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){
		 if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
					opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);
					if (opcode == CONSTANT){
						inConst = 1;
```

```
            } else if (opcode == ENDCONSTANT){
                inConst = 0;
                return size;
            } else if (inConst) {          //we're reading a constant
                strcpy (table[size].name, tokens[0]);
                table[size].value = value(tokens[1]);
                size++;
            }
        }
    }
    return size;    //no constants defined
}



int ReplaceConstants(struct Symbol table[MAX_CONSTANTS], char tableSize)
{
```

```
    char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
    char line[MAX_LINE];
    char tokenCount = 0, isLabel = 0, opcode, size = 0, inConst = 0, pos = 0 ;
    int offset = 0, tmp = 0;

    if (!filesysFileSetReadPos(srcFile, 0)) return 0;

while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){
        if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
            opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);
            if (isLabel)
                pos = 1;
            else pos = 0;

            if (opcode == OP_LDC_W ){
                WriteExecutable(codeIndex - 3, opcode);

                if ( (tmp = searchSymbol(tokens[pos+1], table, tableSize)) != -1){
```

```
                    WriteExecutable(codeIndex - 1, (char) table[tmp].value);
                    WriteExecutable(codeIndex - 2, (char) (table[tmp].value >> 8));
                }
            }
        }
    }
    return 1;

}

int ReplaceMethods(struct Symbol table[MAX_CONSTANTS], char tableSize)
{
    char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
    char line[MAX_LINE];
    char tokenCount = 0, isLabel = 0, opcode, size = 0, inConst = 0, pos = 0;
    int offset = 0, tmp = 0;

    if (!filesysFileSetReadPos(srcFile, 0)) return 0;
```

```
while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){
        if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
                opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);
                if (isLabel)
                        pos = 1;
                else pos = 0;
                if (opcode == OP_INVOKEVIRTUAL ){

                        WriteExecutable(codeIndex - 3, opcode);

                        if ( (tmp = searchSymbol(tokens[pos+1], table, tableSize)) != -1){
                                WriteExecutable(codeIndex - 1, (char) table[tmp].value);
                                WriteExecutable(codeIndex - 2, (char) (table[tmp].value >> 8));
                        }
                }
        }
    }
```

```
	return 1;
}



int ReplaceLabels(struct Symbol table[MAX_CONSTANTS], char tableSize)
{
		char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
		char line[MAX_LINE];
		char tokenCount = 0, isLabel = 0, size = 0, inConst = 0, pos = 0;
	IjvmISA opcode;
		int offset = 0, tmp = 0;

		if (!filesysFileSetReadPos(srcFile, 0)) return 0;

	while ( filesysFileReadLine(srcFile, line, MAX_LINE) ){
			if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
```

```
opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);
if (isLabel)
        pos = 1;
else pos = 0;
if (opcode == OP_GOTO || opcode == OP_IFEQ || opcode == OP_IFLT || opcode == OP_IF_ICMPEQ ){

        WriteExecutable(codeIndex - 3, opcode);

        if ( (tmp = searchSymbol(tokens[pos+1], table, tableSize)) != -1){
                WriteExecutable(codeIndex - 1, (char) (table[tmp].value - codeIndex + offset) );

                WriteExecutable(codeIndex - 2, (char) ( (table[tmp].value - codeIndex + offset) >> 8));
        }
} else if (opcode <= INSTRUCTION_COUNT && opcode > 0) {
        WriteExecutable(codeIndex - offset, opcode);

        if (opcode == OP_BIPUSH){                   //treat the last special case
                WriteExecutable(codeIndex - offset + 1, value(tokens[pos+1]));   //the argument
        }
```

```
                }
            }
        }
    return 1;
}


int BuildVariableTable(struct Symbol table[MAX_CONSTANTS]){
        char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
        char line[MAX_LINE];
        char tokenCount = 0, isLabel = 0, opcode;
        int size = 0, savedReadPos = 0;
        int offset = 0, savedCodeIndex = 0;
        char varOffset = 0, t = 0;

        if (!filesysFileSetReadPos(srcFile, 0)) return 0;

        while (filesysFileReadLine(srcFile, line, MAX_LINE)){
```

```
if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
        opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);
        if (opcode == METHOD){
                if (tokenCount > 1){        //not in main
   for (t = 2; t < tokenCount; t++){
       AddSymbol(tokens[t], varOffset, table, &size);
                        varOffset++;        //params are local variables
   }

   size += varOffset;
                }
        } else if (opcode == VAR){
                inVar = 1;
        } else if (opcode == ENDVAR && inVar){



 //as soon as we reach the end of a method we replace the jumps with
  //the label offset
```

```
            //This helps both for syntax checking: just labels defined
            //in the local scope will exist in the table
            //and we keep the table small
                    ReplaceVariables(table, size);

            //codeIndex = savedCodeIndex;
                    inVar = 0;
                    size = 0;//reset the table of labels
                    varOffset = 0;

                } else if (inVar) {        //we're reading a variable
                    AddSymbol(tokens[0], varOffset, table, &size);
                    varOffset++;
                }
            }
        }
        return size;    //no constants defined
}
```

```
int ReplaceVariables(struct Symbol table[MAX_CONSTANTS], char tableSize){
        char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE];
        char line[MAX_LINE];
        char tokenCount = 0, isLabel = 0, opcode, size = 0, inConst = 0, pos = 0;
        int offset = 0, tmp = 0;
        //no need to rewind the file as the variable can only be used from here on

        while (filesysFileReadLine(srcFile, line, MAX_LINE)){
                if ( (tokenCount = parse(line, CODE_DELIMITERS, tokens)) != 0){
                        opcode = parseTokens(tokens, tokenCount, &codeIndex, &isLabel, &offset);
                        if (isLabel)
                                pos = 1;
                        else pos = 0;
             if (opcode == ENDMETHOD){
              return 0;
             } else if (opcode == OP_IINC || opcode == OP_ILOAD || opcode == OP_ISTORE){
```

```
					if (opcode == OP_IINC){
						if ( (tmp = searchSymbol(tokens[pos+1], table, tableSize)) != -1){
				WriteExecutable(codeIndex - offset + 1, table[tmp].value);
				WriteExecutable(codeIndex - offset + 2, value(tokens[pos+2]));
						}
					} else {
						if ( (tmp = searchSymbol(tokens[pos+1], table, tableSize)) != -1){
				WriteExecutable(codeIndex - offset + 1, table[tmp].value);
						}
					}
				}
			}
		}
}

int parseConstants(void)
{
	struct Symbol table[MAX_CONSTANTS];
```

```
        char size = 0;
        //create a table of constants
        //then replace all the references with the values from the table



        size = BuildConstantTable(table);


        codeIndex = 0;
    if (!filesysFileSetWritePos(execFile, codeIndex)) return 0;


        if (size > 0)
                ReplaceConstants(table, size);


        codeSize = codeIndex;
    return 1; //TODO check return value - is success at this point??
}


int parseMethods(void)
```

```
{
        struct Symbol table[MAX_METHODS];
        char size = 0;
        //create a table of methods
        //then replace all the references with the values from the table

        size = BuildMethodTable(table);

        codeIndex = 0;
    if (!filesysFileSetWritePos(execFile, codeIndex)) return 0;

        if (size > 0)
                ReplaceMethods(table, size);

        codeSize = codeIndex;
    return 1; //TODO check return value - is success at this point??
}
```

```
int parseLabels(void)

{

        struct Symbol table[MAX_METHODS];

        char size = 0;

        //create a table of constants

        //then replace all the references with the values from the table


        BuildLabelTable(table);


        codeIndex = 0;

        if (!filesysFileSetWritePos(execFile, codeIndex)) return 0;


        codeSize = codeIndex;

    return 1; //TODO check return value - is success at this point??

}



int parseVariables(void)
```

```
{
        struct Symbol table[MAX_METHODS];
        char size = 0;
        //create a table of constants
        //then replace all the references with the values from the table
        codeIndex = 0;

        BuildVariableTable(table);
        if (!filesysFileSetWritePos(execFile, codeIndex)) return 0;

    return 1;
}



//This function advances the write position in the compiled code
//so that direct write is possible - this way it also keeps track of
//the compiled size of the code
```

```
//It also does syntax checks for each line
int parseTokens(char tokens[MAX_TOKEN_COUNT][MAX_TOKEN_SIZE], int tokenCount, int *codeIndex, char *isLabel, int *offset)
{

	char opcode = -1;
	char pos = 0;
	int initial = *codeIndex;

	(*isLabel) = 0;

	if (tokenCount > 0){

		if (tokens[0][ strlen(tokens[0]) -1 ] == ':'){ //It is a label
			tokens[0][strlen(tokens[0]) -1 ] = '\0';	//remove the ':'
			(*isLabel) = 1;
			pos++;
		}
```

```
if (strcmp(tokens[pos], sBIPUSH) == 0){
        opcode = OP_BIPUSH;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sDUP) == 0){
        opcode = OP_DUP;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sERR) == 0){
        opcode = OP_ERR;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sGOTO) == 0){
        opcode = OP_GOTO;
        (*codeIndex)++;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sHALT) == 0){
        opcode = OP_HALT;
        (*codeIndex)++;
```

```
} else if (strcmp(tokens[pos], sIADD) == 0){
        opcode = OP_IADD;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIAND) == 0){
        opcode = OP_IAND;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIFEQ) == 0){
        opcode = OP_IFEQ;
        (*codeIndex)++;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIFLT) == 0){
        opcode = OP_IFLT;
        (*codeIndex)++;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIF_ICMPEQ) == 0){
        opcode = OP_IF_ICMPEQ;
```

```
        (*codeIndex)++;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIINC) == 0){
        opcode = OP_IINC;
        (*codeIndex)++;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sILOAD) == 0){
        opcode = OP_ILOAD;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sijIN) == 0){
        opcode = OP_IN;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sINVOKEVIRTUAL) == 0){
        opcode = OP_INVOKEVIRTUAL;
        (*codeIndex)++;
```

```
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIOR) == 0){
        opcode = OP_IOR;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sIRETURN) == 0){
        opcode = OP_IRETURN;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sISTORE) == 0){
        opcode = OP_ISTORE;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sISUB) == 0){
        opcode = OP_ISUB;
        //replace var with offset
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sLDC_W) == 0){
        opcode = OP_LDC_W;
```

```
        (*codeIndex)++;
        (*codeIndex)++;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sNOP) == 0){
        opcode = OP_NOP;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sijOUT) == 0){
        opcode = OP_OUT;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sPOP) == 0){
        opcode = OP_POP;
        (*codeIndex)++;
} else if (strcmp(tokens[pos], sSWAP) == 0){
        opcode = OP_SWAP;
        (*codeIndex)++;
}       else if (strcmp(tokens[pos], sWIDE) == 0){
        opcode = OP_WIDE;
        (*codeIndex)++;
```

```
}           else


//Keywords

if (strcmp(tokens[pos], sCONSTANT) == 0){

            opcode = CONSTANT;

} else if (strcmp(tokens[pos], sENDCONSTANT) == 0){

            opcode = ENDCONSTANT;

} else if (strcmp(tokens[pos], sMAIN) == 0){        //we treat main and methods identically

            opcode = METHOD;

            (*codeIndex)++;                        //number of params

            (*codeIndex)++;                        //number of local vars - excluding params

} else if (strcmp(tokens[pos], sENDMAIN) == 0){

            opcode = ENDMETHOD;

} else if (strcmp(tokens[pos], sMETHOD) == 0){

            opcode = METHOD;

            (*codeIndex)++;                        //number of params

            (*codeIndex)++;                        //number of local vars - excluding params

} else if (strcmp(tokens[pos], sENDMETHOD) == 0){
```

```
                opcode = ENDMETHOD;
            } else if (strcmp(tokens[pos], sENDVAR) == 0){
                opcode = ENDVAR;
            } else if (strcmp(tokens[pos], sVAR) == 0){
                opcode = VAR;
            }
        }
        *offset = (*codeIndex) - initial;
        return opcode;
}



int value(char *token){
        int value;

        //if (token[0] == '\'')  //only if needed
            //return (int)token[1];
```

```
    sscanf(token, "%i", &value);



    return value;
}

//Find a symbol in a symbol table
int searchSymbol(char symbol[MAX_TOKEN_SIZE], struct Symbol *table, int symbolCount)
{
        int i;
        for(i = 0; i < symbolCount; i++){
                if (strcmp(table[i].name, symbol) == 0){
                        return i;
                }
        }
        return -1;
}
```

```
//Add a symbol to a symbol table

int AddSymbol(char* name, int address, struct Symbol *table, int *tableIndex)

{

        strcpy(table[*tableIndex].name, name);

        table[*tableIndex].value = address;

        (*tableIndex)++;

    return 1; //TODO check return value - is success at this point??

}


#ifndef __myGlobalDefines

#define __myGlobalDefines


//Constants for string sizes

#define MAX_TOKEN_COUNT 7

#define MAX_TOKEN_SIZE 15

#define MAX_LINE 80

#define MAX_CONSTANTS 15

#define MAX_METHODS 15
```

```
//Delimiter string for strtok
#define CODE_DELIMITERS " \n\t(),"

#define INSTRUCTION_COUNT 23

//Section states: .var, .method, .main, .constant
//enum SECTIONS {INVAR = 2,INMETHOD = 4, INMAIN = 8, INCONSTANT = 16};

#define sBIPUSH     "BIPUSH"    //byte        Push a byte onto stack
#define sDUP        "DUP"       //N/A         Copy top word on stack and push onto stack
#define sERR        "ERR"       //N/A  Print an error message and halt the simulator
#define sGOTO       "GOTO"      //label name        Unconditional jump
#define sHALT       "HALT"      //N/A Halt the simulator
#define sIADD       "IADD"      //N/A   Pop two words from stack; push their sum
#define sIAND       "IAND"      //N/A  Pop two words from stack; push Boolean AND
#define sIFEQ       "IFEQ"      //label name    Pop word from stack and branch if it is zero
#define sIFLT       "IFLT"      //label name        Pop word from stack and branch if it is less than zero
```

```
#define sIF_ICMPEQ      "IF_ICMPEQ"    //label name       Pop two words from stack and branch if they are equal

#define sIINC           "IINC"       //variable name, byte       Add a constant value to a local variable

#define sILOAD          "ILOAD"     //variable name            Push local variable onto stack

#define sijIN           "IN"        //N/A       Reads a character from the keyboard buffer and pushes it onto the stack.

                                                                                        //If no character is available, 0 is pushed

                                                                                        //for consistency with ijOUT

#define sINVOKEVIRTUAL          "INVOKEVIRTUAL"     //method name      Invoke a method

#define sIOR            "IOR"      //N/A        Pop two words from stack; push Boolean OR

#define sIRETURN                    "IRETURN"   //N/A         Return from method with integer value

#define sISTORE         "ISTORE"    //variable name           Pop word from stack and store in local variable

#define sISUB           "ISUB"      //N/A       Pop two words from stack; push their difference

#define sLDC_W          "LDC_W"      //constant name       Push constant from constant pool onto stack

#define sNOP            "NOP"     //N/A       Do nothing

#define sijOUT           "OUT"     //N/A      Pop word off stack and print it to standard out

//OUT is a Crossworks macro

#define sPOP            "POP"     //N/A       Delete word from top of stack

#define sSWAP           "SWAP"    //N/A    Swap the two top words on the stack

#define sWIDE           "WIDE"    //N/A      Prefix instruction; next instruction has a 16-bit index
```

```
//IJVM keywords - starting from 90 to make sure they are different from instructions
enum KEYWORDS { CONSTANT = 90, ENDCONSTANT, MAIN, ENDMAIN,
                METHOD, ENDMETHOD, VAR, ENDVAR};

#define sCONSTANT       ".constant"
#define sENDCONSTANT    ".end-constant"
#define sMAIN           ".main"
#define sENDMAIN        ".end-main"
#define sMETHOD             ".method"
#define sENDMETHOD      ".end-method"
#define sVAR            ".var"
#define sENDVAR             ".end-var"

//Segment sizes
#define MAX_CODE_SIZE   600
```

```
typedef struct Symbol{
        int value;
        char name[MAX_TOKEN_SIZE];
};

/*extern char code[500];
extern int codeIndex;
extern int codeSize;
extern int inConst;
extern int inMethod;
extern int inVar;
*/
#endif

// This is the ijvm processor module. It executes ijvm code precompiled by
// the ijvmcompiler module.
```

```
#include <common.h>

#include <ijvm.h>

#include <ijvmcompiler.h>

#include <serial.h>

#include <filesys.h>

#include <ijvm_io.h>

#include <string.h>

#include <stdio.h>

#include <ijvmcpld.h>


//---------------------------------------------------------------------------

// Public globals

//---------------------------------------------------------------------------

IjvmGlobalData ijvmGlobalData;


//---------------------------------------------------------------------------

// Private data types and macros

//---------------------------------------------------------------------------
```

```
#define IJVM_STACK_SIZE        128   //number of IjvmStackData entries

#define IJVM_MAX_FRAMES        5     //number of context frames

#define IJVM_MAX_VARS          20    //max number of vars throughout all frames


//Uncomment this to use CPLD-accelerated instructions

//#define CPLD_ACCELERATION


// ALU operations

#define ALU_SUB(x,y)       (x - y)

#define ALU_ADD(x,y)       (x + y)

#define ALU_MUL(x,y)       (x * y)

#define ALU_AND(x,y)       (x && y)

#define ALU_SQRT(x)        (ijvmAluSqrt(x))

#define ALU_OR(x,y)        (x || y)

#define ALU_EQ(x,y)        (x == y)

#define ALU_EQZ(x)         (ALU_EQ(x,0))

#define ALU_LT(x,y)        (x < y)
```

```
#define ALU_LTZ(x)        (ALU_LT(x,0))

#ifndef CPLD_ACCELERATION
 #define ALU_DIV(x,y)       (x / y)
 #define ALU_MOD(x,y)       (x % y)
#else
 #define ALU_DIV(x,y)       (cpldAluDiv(x,y))
 #define ALU_MOD(x,y)       (cpldAluMod(x,y))
#endif

//a macro to suppress debug output
#define debug_printf(x)   {}

//internal IJVM basic types
typedef unsigned char     IjvmByte;
typedef unsigned short    IjvmReg;
typedef signed short      IjvmAddrOffset;
typedef unsigned short    IjvmStackData;
```

```
typedef signed short      IjvmStackDataSigned;

typedef short             IjvmVarData;

typedef unsigned char     IjvmVarIndex;


//data type to hold frame-specific data

typedef struct {

  IjvmReg PC;     //program counter of the current frame

  IjvmReg LV;     //pointer to the start of local frame vars in the global vars stack

  IjvmReg NV;     //number of local variables for the current frame

}

IjvmFrameData;


//data type to store global info for current execution

typedef struct {

  IjvmReg SP;              //stack pointer reg

  IjvmReg FP;              //current frame pointer

}

IjvmRegs;
```

```
//internal ijvm FSM states
typedef enum {
  STATE_STOPPED,
  STATE_LOADING,
  STATE_PREPARE_RUN,
  STATE_RUNNING,
  STATE_HALTED,
  STATE_RUN_ERROR
}
IjvmState;

//---------------------------------------------------------------------------
// Private functions
//---------------------------------------------------------------------------

static int ijvmPrepareNewFrameContext(void);
static int ijvmGetVar(IjvmVarIndex index, IjvmVarData *var);
```

```
static int ijvmSetVar(IjvmVarIndex index, IjvmVarData var);

static int ijvmStackPush(IjvmStackData x);

static int ijvmStackPop(IjvmStackData *x);


//----------------------------------------------------------------------------

// Private globals

//----------------------------------------------------------------------------

static IjvmStackData stack[IJVM_STACK_SIZE];     //the data stack for IJVM

static IjvmVarData vars[IJVM_MAX_VARS];          //the variable buffer

static IjvmFrameData frames[IJVM_MAX_FRAMES];    //ijvm function frames


static IjvmRegs regs;                    //current execution data

static IjvmState state;                  //ijvm module current FSM state

static FileID execFile = FILE_ID_INVALID;    //file descriptor of the executable


//Flag to track the initialization state of the module

static int moduleInitialized = 0;
```

```
//-----------------------------------------------------------------------------

// ijvmModuleInit: initializes all required modules

// arguments: none

// return: 1 if success, 0 if not

//-----------------------------------------------------------------------------

int ijvmModuleInit(void)

{

  if (moduleInitialized) {

    return 1;

  }

  moduleInitialized = 1;


  if (!ijvmioModuleInit() || !filesysModuleInit() || !cpldModuleInit()) {

    moduleInitialized = 0;

    return 0;

  }


  //clear global info shared with other modules
```

```
  memset(&ijvmGlobalData, 0, sizeof(ijvmGlobalData));

  //default VM state stopped
  state = STATE_STOPPED;

  return 1;
}

//--------------------------------------------------------------------------
// ijvmFetch: fetches the byte from the executable file at the current program
//   counter index and increments it.
// arguments: IjvmByte *data - pointer to the fetched data
// return: 1 if success, 0 if not
//--------------------------------------------------------------------------
static int ijvmFetch(IjvmByte *data)
{
  //check if the executable file has been opened
  if (execFile == FILE_ID_INVALID) return 0;
```

```
 //read from the executable file. ensure that a valid location is read
 if (!filesysFileSetReadPos(execFile, frames[regs.FP].PC) ||
     !filesysFileRead(execFile, (char *)data)) {

     ijvmGlobalData.lastError = ERROR_CODE_SEG_FAULT;
     return 0;
 }

 frames[regs.FP].PC++; //increment current program counter
 return 1;
}

//---------------------------------------------------------------------------
// ijvmLoad: initializes all VM registers for the new execution.
// arguments: none
// return: 1 if success, 0 if not
//---------------------------------------------------------------------------
```

```
static int ijvmLoad(void)
{
  IjvmByte lo,hi;
  unsigned int i;

  if (execFile == FILE_ID_INVALID) return 0;

  regs.SP = 0;
  regs.FP = 0;
  frames[0].LV = 0;
  frames[0].NV = 0;
  frames[0].PC = 0;

  //prepare the new frame context (arguments & local vars).
  //the main function has no args but it may have local vars
  if (!ijvmPrepareNewFrameContext()) return 0;
  return 1;
}
```

```
//------------------------------------------------------------------------------
// ijvmGetVar: looks up a variable in the variables stack that belongs to the
//   current frame and returns its value.
// arguments:
//   IjvmVarIndex index - the index of the local variable within current frame
//   IjvmVarData *var - the pointer to hold the value
// return: 1 if success, 0 if not
//------------------------------------------------------------------------------
static int ijvmGetVar(IjvmVarIndex index, IjvmVarData *var)
{
  if (index >= frames[regs.FP].NV) {
    //the variable index is invalid
    ijvmGlobalData.lastError = ERROR_VAR_SEG_FAULT;
    return 0;
  }
  *var = vars[frames[regs.FP].LV + index];
  return 1;
```

```
}

//-----------------------------------------------------------------------------
// ijvmSetVar: looks up a variable in the variables stack that belongs to the
//   current frame and sets its value.
// arguments:
//   IjvmVarIndex index - the index of the local variable within current frame
//   IjvmVarData var - new variable value
// return: 1 if success, 0 if not
//-----------------------------------------------------------------------------
static int ijvmSetVar(IjvmVarIndex index, IjvmVarData var)
{
  if (index >= frames[regs.FP].NV) {
    //the variable index is invalid
    ijvmGlobalData.lastError = ERROR_VAR_SEG_FAULT;
    return 0;
  }
  vars[frames[regs.FP].LV + index] = var;
```

```
  return 1;
}

//-----------------------------------------------------------------------------
// ijvmPrepareNewFrameContext: initializes the new frame upon the new function
//   invokation. At this point the program counter must point at the function
//   prefix in the executable file, containing two bytes - the number of
//   arguments, and the number of local variables. These are used to create the
//   new frame record, and reserve space for the local variables on the variable
//   stack. Also the call arguments are pushed off the main program stack and
//   stored as local variables.
// arguments: none
// return: 1 if success, 0 if not
//-----------------------------------------------------------------------------
static int ijvmPrepareNewFrameContext(void)
{
  IjvmByte numArgs;
  IjvmByte numVars;
```

```
IjvmVarIndex i,j;
IjvmVarIndex frameLV, frameNV;

//read the function prefix: # of arguments, # of local vars
if (!ijvmFetch(&numArgs)) return 0;
if (!ijvmFetch(&numVars)) return 0;

//check if our frame is the lowest frame, meaning that the main function's\
//frame is being initialized
if (regs.FP == 0) {
  frames[0].LV = 0;
  frames[0].NV = numVars;
}
else {
  //reserve the local variable space after the space taken by the lower
  //frame
  frames[regs.FP].LV = frames[regs.FP - 1].LV + frames[regs.FP - 1].NV;
  frames[regs.FP].NV = numVars + numArgs;
```

```
}

frameLV = frames[regs.FP].LV;
frameNV = frames[regs.FP].NV;

if ((frameLV + frameNV) >= IJVM_MAX_VARS) {
  //too many variables
  ijvmGlobalData.lastError = ERROR_VAR_SEG_OVERFLOW;
  return 0;
}

for (i = 0; i < numArgs; i++ ) {
  //store the user-pushed arguments for the function as local variables
  if (!ijvmStackPop((IjvmStackData *)&vars[frameLV++])) return 0;
}

for (i = 0; i < numVars; i++ ) {
  vars[frameLV++] = 0;  //clear other local variables
```

```
  }
  return 1;
}

//----------------------------------------------------------------------------
// ijvmStackPush: push a new data onto the IJVM data stack, and increment
//   the stack pointer
// arguments: IjvmStackData x - a new value to be pushed
// return: 1 if success, 0 if not
//----------------------------------------------------------------------------
static int ijvmStackPush(IjvmStackData x)
{
  if (regs.SP < IJVM_STACK_SIZE) {
    stack[regs.SP] = x;
    regs.SP = regs.SP + 1;
    return 1;
  }
  //too many values pushed
```

```
  ijvmGlobalData.lastError = ERROR_STACK_OVERFLOW;
  return 0;
}

//---------------------------------------------------------------------------
// ijvmStackPop: decrement the stack pointer and pop the top data from the IJVM
//   data stack.
// arguments: IjvmStackData *x - pointer to hold the popped value
// return: 1 if success, 0 if not
//---------------------------------------------------------------------------
static int ijvmStackPop(IjvmStackData *x)
{
  if (regs.SP != 0) {
    regs.SP = regs.SP - 1;
    *x = stack[regs.SP];
    return 1;
  }
  //stack is empty
```

```
  ijvmGlobalData.lastError = ERROR_STACK_UNDERFLOW;
  return 0;
}

//----------------------------------------------------------------------------
// ijvmAluSqrt: the MCU implementation of the square root. It is faster than
//   the math.h sqrt implementation
// arguments: IjvmStackData x - the argument to the square root
// return: IjvmStackData - the square root of x
//----------------------------------------------------------------------------
IjvmStackData ijvmAluSqrt(IjvmStackData x)
{
  unsigned short bit = 16;
  unsigned short mask = 0x8000;
  unsigned short root = 0x0000;
  unsigned long acc;

  do
```

```
  {
    acc = root | mask;

    if (acc * acc <= x)
      root |= mask;
    mask >>= 1;
  }
  while (--bit);

  return root;
}

//----------------------------------------------------------------------------
// ijvmExecuteOut: this function executes the particular type of the OUT
//   instruction. The argument(s) of the instruction should be stored on the
//   data stack.
// arguments:  OutInstructionType type - the subtype of the OUT instruction
// return: 1 on success, 0 on failure
```

```
//-----------------------------------------------------------------------------
int ijvmExecuteOut(OutInstructionType type)
{
  IjvmStackData stackData1;
  IjvmStackData stackData2;
  int success = 0;

  switch(type) {
    //-----------------------------------------------------------------------
    // Output to the device connected to the master (upstream) COM port (std out)
    //-----------------------------------------------------------------------
    case OUT_STD_MASTER:
      //pop the value to be OUT'ed from the data stack
      if (!ijvmStackPop(&stackData1)) break;

      if (ijvmGlobalData.boardId  == 0) {
        //if the board is first in the chain, then we need to convert the number
        //to ASCII to be displayed on the terminal connected to the board's
```

```
  //master COM port
  char numbuf[8] = {0};
  snprintf(numbuf, 8, "%u\r\n", stackData1);
  success = ijvmioSerialWriteString(IO_CONSUMER_APP, COM_1, numbuf, strlen(numbuf));
 }
 else {
  //if the board is one of the slaves in the board chain, then the data
  //is transfered in raw binary as two bytes, except for the delimiters
  //added before ('[') and after it (']')
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_1, '[');
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_1, (char)(stackData1 >> 8));
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_1, (char)stackData1);
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_1, ']');
 }
 break;
//-----------------------------------------------------------------------
// Output to the device connected to the slave (downstream) COM port (std out)
//-----------------------------------------------------------------------
```

```
case OUT_STD_SLAVE:
  //pop the value to be OUT'ed from the data stack
  if (!ijvmStackPop(&stackData1)) break;

  //the data is transfered in raw binary as two bytes within the chain,
  //delimited by '[' and ']'
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_2, '[');
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_2, (char)(stackData1 >> 8));
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_2, (char)stackData1);
  success = ijvmioSerialWrite(IO_CONSUMER_APP, COM_2, ']');
  break;

//-----------------------------------------------------------------------
// Output to the multiplication engine (calculations are done synchronously)
//-----------------------------------------------------------------------
case OUT_ALU_MUL:
  //pop the arguments to be multiplied from the data stack
  if (!ijvmStackPop(&stackData1)) break;
```

```
  if (!ijvmStackPop(&stackData2)) break;

  //multiply them and push the result back onto the data stack
  if (!ijvmStackPush(ALU_MUL(stackData2, stackData1))) break;
  success = 1;
  break;

//-----------------------------------------------------------------------
// Output to the division engine (calculations are done synchronously)
//-----------------------------------------------------------------------
case OUT_ALU_DIV:
  //pop the arguments for division from the data stack
  if (!ijvmStackPop(&stackData1)) break;
  if (!ijvmStackPop(&stackData2)) break;

  //perform division and push the result back onto the data stack
  if (!ijvmStackPush(ALU_DIV(stackData2, stackData1))) break;
  success = 1;
```

```
  break;

//----------------------------------------------------------------------
// Output to the modulus engine (calculations are done synchronously)
//----------------------------------------------------------------------
case OUT_ALU_MOD:
  //pop the arguments for modulus from the data stack
  if (!ijvmStackPop(&stackData1)) break;
  if (!ijvmStackPop(&stackData2)) break;

  //perform modulus and push the result back onto the data stack
  if (!ijvmStackPush(ALU_MOD(stackData2, stackData1))) break;
  success = 1;
  break;

//----------------------------------------------------------------------
// Output to the square root engine (calculations are done synchronously)
//----------------------------------------------------------------------
```

```
    case OUT_ALU_SQRT:
      //pop the argument for square root from the data stack
      if (!ijvmStackPop(&stackData1)) break;
      //perform square root and push the result back onto the data stack
      if (!ijvmStackPush(ALU_SQRT(stackData1))) break;
      success = 1;
      break;
  }

  return success;
}

//------------------------------------------------------------------------------
// ijvmExecuteIn: this function executes the particular type of the IN
//   instruction. The argument(s) of the instruction should be stored on the
//   data stack.
// arguments:  InInstructionType type - the subtype of the IN instruction
// return: 1 on success, 0 on failure
```

```
//------------------------------------------------------------------------------
int ijvmExecuteIn(InInstructionType type)
{
  IjvmStackData stackData1;
  KeyType key;
  int success = 0;
  char c1,c2,c3,c4;

  switch(type) {

    //------------------------------------------------------------------------
    // Input from the device connected to the master (upstream) COM port (std in)
    //------------------------------------------------------------------------
    case IN_STD_MASTER:
      stackData1 = 0; //reset the result of the IN operation. Default=0
      success = ijvmioSerialRead(IO_CONSUMER_APP, COM_1, &c1);

      //If there's new data that came from the master port and it is
```

```
//properly delimited then block until all of that delimited data is read
//This should not block for a long time as delimited data is transfered in
//integer chunks, so the impact on the execution is small.
if (success && c1=='[') {

  //after the start delimiter, the next two bytes should be the data,
  //followed by the end delimiter.
  while(!ijvmioSerialRead(IO_CONSUMER_APP, COM_1, &c2));
  while(!ijvmioSerialRead(IO_CONSUMER_APP, COM_1, &c3));
  while(!ijvmioSerialRead(IO_CONSUMER_APP, COM_1, &c4));
  if (c4 == ']') {
    //If the end delimiter is valid, we have a valid word that came from
    //the master. We construct the value to be passed to the IJVM app.
    stackData1 = c2 << 8 | c3;
  }
  success = 1;
}
else {
```

```
  //No new data has been received, or the delimiters are invalid, proceed
  //with the execution. No new data to be passed to the IJVM app.
  success = 1;
 }
 //Push the newly received data, or zero, if nothing's been received
 if (!ijvmStackPush(stackData1)) success = 0;
 break;

//------------------------------------------------------------------------
// Input from the device connected to the slave(downstream) COM port (std in)
//------------------------------------------------------------------------
case IN_STD_SLAVE:
 stackData1 = 0; //reset the result of the IN operation. Default=0
 success = ijvmioSerialRead(IO_CONSUMER_APP, COM_2, &c1);

 //If there's new data that came from the slave port and it is
 //properly delimited then block until all of that delimited data is read
 //This should not block for a long time as delimited data is transfered in
```

```
//integer chunks, so the impact on the execution is small.
if (success && c1=='[') {

  //after the start delimiter, the next two bytes should be the data,
  //followed by the end delimiter.
  while(!ijvmioSerialRead(IO_CONSUMER_APP, COM_2, &c2));
  while(!ijvmioSerialRead(IO_CONSUMER_APP, COM_2, &c3));
  while(!ijvmioSerialRead(IO_CONSUMER_APP, COM_2, &c4));

  if (c4 == ']') {
    //If the end delimiter is valid, we have a valid word that came from
    //the slave. We construct the value to be passed to the IJVM app.
    stackData1 = c2 << 8 | c3;
  }
  success = 1;
}
else {
  //No new data has been received, or the delimiters are invalid, proceed
```

```
    //with the execution. No new data to be passed to the IJVM app.
    success = 1;
  }
  //Push the newly received data, or zero, if nothing's been received
  if (!ijvmStackPush(stackData1)) success = 0;
  break;

//--------------------------------------------------------------------------
// Input from the keypad
//--------------------------------------------------------------------------
case IN_KEYPAD:
  //read the keypad buffer
  key = ijvmioKeypadRead(IO_CONSUMER_APP);

  //if no new key has been pressed, push 0 onto the data stack, otherwise
  //push the ascii value of the key
  stackData1 = (key == KEY_NONE) ? 0 : keypadKeyToChar(key);
  if (!ijvmStackPush(stackData1)) break;
```

```
  success = 1;
  break;

//------------------------------------------------------------------------
// Input the current board ID parameter from the virtual machine
//------------------------------------------------------------------------
case IN_BOARDID:
  //push the board ID onto the data stack for the application to read
  if (!ijvmStackPush(ijvmGlobalData.boardId)) break;
  success = 1;
  break;

//------------------------------------------------------------------------
// Input the number of the boards in the chain from the virtual machine
//------------------------------------------------------------------------
case IN_NUMBOARDS:
  //push the number of the boards onto the data stack for the application to read
  if (!ijvmStackPush(ijvmGlobalData.boardsNum)) break;
```

```
    success = 1;
    break;
  }

  return success;
}

//-----------------------------------------------------------------------------
// ijvmProcessFsm: process the FSM of the virtual machine module. It's either
// stopped, or running and executing the IJVM compiled bytecode.
// arguments:  none
// return: 1 on success, 0 on failure
//-----------------------------------------------------------------------------
int ijvmProcessFsm(void)
{
  //temporary variables used during the execution
  IjvmByte opcode;
  IjvmByte arg1;
```

```
IjvmByte arg2;

IjvmStackData stackData1;

IjvmStackData stackData2;

IjvmVarData varData1;

IjvmReg address;

int busy = 0;


if (!moduleInitialized) {

  return 0;

}


switch (state) {


  //===========================================================================

  // The virtual machine is running, i.e. it is fetching, decoding and

  // executing instructions from the executable compiled image.

  //===========================================================================

  case STATE_RUNNING:
```

```
// (1) Fetch the opcode for the instruction at the current PC

if (!ijvmFetch(&opcode)) {

  state = STATE_RUN_ERROR;

  break;

}


// (2) Decode the opcode, and execute the appropriate instruction

switch (opcode) {

  //--------------------------------------------------------------------

  case OP_NOP:

    debug_printf("NOP\n");

    break;

  //--------------------------------------------------------------------

  case OP_BIPUSH:

    debug_printf("BIPUSH\n");


    //This is a 2-byte instruction. Fetch the argument, stored as the
```

```
  //second byte after the opcode, and push it onto the data stack
  if (!ijvmFetch(&arg1) ||
      !ijvmStackPush((IjvmStackData)arg1)) {
   state = STATE_RUN_ERROR;
   break;
  }
  break;
//---------------------------------------------------------------------
case OP_DUP:
  debug_printf("DUP\n");

  //Pop the current value at the top of the data stack and push it twice
  //This essentially duplicates it.
  if (!ijvmStackPop(&stackData1) ||
      !ijvmStackPush(stackData1) ||
      !ijvmStackPush(stackData1)) {

   state = STATE_RUN_ERROR;
```

```
    break;
  }
  break;
//---------------------------------------------------------------------
case OP_GOTO:
  debug_printf("GOTO\n");

  //This is a 3-byte instruction. Fetch the arguments, stored as the
  //second and third byte after the opcode. These two bytes form the
  //jump offset relative to the address of the opcode of the current
  //instruction.

  if (!ijvmFetch(&arg1) || !ijvmFetch(&arg2)) {
    state = STATE_RUN_ERROR;
    break;
  }

  //The jump is relative to the address of the opcode. After fetching
```

```
 //the instruction arguments, the PC needs to be rewound, so decrement
 //it by 3 to point to the current opcode, and apply the signed offset
 frames[regs.FP].PC = (frames[regs.FP].PC - 3) + (IjvmAddrOffset)((arg1 << 8) | (arg2));
 break;
//--------------------------------------------------------------------
case OP_IADD:
 debug_printf("IADD\n");

 //integer add instruction: arguments are popped from the stack and the
 //result is pushed onto the stack

 if (!ijvmStackPop(&stackData1) ||
     !ijvmStackPop(&stackData2) ||
     !ijvmStackPush(ALU_ADD(stackData1, stackData2))) {

  state = STATE_RUN_ERROR;
  break;
 }
```

```
  break;
//-------------------------------------------------------------------
case OP_IAND:
  debug_printf("IAND\n");

  //boolean (not bitwise) AND: arguments are popped from the stack and
  //the result is pushed onto the stack
  if (!ijvmStackPop(&stackData1) ||
      !ijvmStackPop(&stackData2) ||
      !ijvmStackPush(ALU_AND(stackData1, stackData2))) {

    state = STATE_RUN_ERROR;
    break;
  }
  break;
//-------------------------------------------------------------------
case OP_IFEQ:
  debug_printf("IFEQ\n");
```

```
//This is a 3-byte instruction. Fetch the arguments, stored as the
//second and third byte after the opcode. These two bytes form the
//jump offset relative to the address of the opcode of the current
//instruction.

if (!ijvmFetch(&arg1) ||      //get jump offset upper byte
    !ijvmFetch(&arg2)) {      //get jump offset lower byte
  state = STATE_RUN_ERROR;
  break;
}

//pop the value to compare to zero from the data stack
if (!ijvmStackPop(&stackData1)) {
  state = STATE_RUN_ERROR;
  break;
}
```

```
  if (ALU_EQZ(stackData1)){

    //The value is equal to zero. So we need to jump.
    //The jump is relative to the address of the opcode. After fetching
    //the instruction arguments, the PC needs to be rewound, so decrement
    //it by 3 to point to the current opcode, and apply the signed offset
    frames[regs.FP].PC = (frames[regs.FP].PC - 3) + (IjvmAddrOffset)((arg1 << 8) | (arg2));
  }
  break;
//---------------------------------------------------------------------
case OP_IFLT:
  debug_printf("IFLT\n");

  //This is a 3-byte instruction. Fetch the arguments, stored as the
  //second and third byte after the opcode. These two bytes form the
  //jump offset relative to the address of the opcode of the current
  //instruction.
```

```
if (!ijvmFetch(&arg1) ||      //get jump offset upper byte
    !ijvmFetch(&arg2)) {      //get jump offset lower byte
  state = STATE_RUN_ERROR;
  break;
}

//pop the value to compare to zero from the data stack
if (!ijvmStackPop(&stackData1)) {
  state = STATE_RUN_ERROR;
  break;
}
if (ALU_LTZ((IjvmStackDataSigned)stackData1)){

  //The value is less than zero. So we need to jump.
  //The jump is relative to the address of the opcode. After fetching
  //the instruction arguments, the PC needs to be rewound, so decrement
  //it by 3 to point to the current opcode, and apply the signed offset
  frames[regs.FP].PC = (frames[regs.FP].PC - 3) + (IjvmAddrOffset)((arg1 << 8) | (arg2));
```

```
  }
  break;
//--------------------------------------------------------------------
case OP_IF_ICMPEQ:
  debug_printf("IF_ICMP\n");

  //This is a 3-byte instruction. Fetch the arguments, stored as the
  //second and third byte after the opcode. These two bytes form the
  //jump offset relative to the address of the opcode of the current
  //instruction.

  if (!ijvmFetch(&arg1) ||     //get jump offset upper byte
      !ijvmFetch(&arg2)) {     //get jump offset lower byte
    state = STATE_RUN_ERROR;
    break;
  }

  //pop two values to be compared from the data stack
```

```
  if (!ijvmStackPop(&stackData1) ||
      !ijvmStackPop(&stackData2)) {

    state = STATE_RUN_ERROR;
    break;
  }

  if (ALU_EQ(stackData2,stackData1)){

    //The values are equal. So we need to jump.
    //The jump is relative to the address of the opcode. After fetching
    //the instruction arguments, the PC needs to be rewound, so decrement
    //it by 3 to point to the current opcode, and apply the signed offset
    frames[regs.FP].PC = (frames[regs.FP].PC - 3) + (IjvmAddrOffset)((arg1 << 8) | (arg2));
  }
  break;
//-------------------------------------------------------------------
case OP_IINC:
```

```
debug_printf("IINC\n");

//This is a 3-byte instruction. Fetch the arguments, stored as the
//second and third byte after the opcode. The first one is the index
//of the local variable to be modified, the second is the constant to
//be added to the variable

if (!ijvmFetch(&arg1) ||      //get the local variable offset
    !ijvmFetch(&arg2)) {      //get the constant to be added
  state = STATE_RUN_ERROR;
  break;
}

//Try to look up the local variable by its index, and add the constant
if (!ijvmGetVar((IjvmVarIndex)arg1, &varData1) ||
    !ijvmSetVar((IjvmVarIndex)arg1, ALU_ADD(varData1,(IjvmByte)arg2))) {
  state = STATE_RUN_ERROR;
  break;
```

```
  }
  break;
//---------------------------------------------------------------------
case OP_ILOAD:
  debug_printf("ILOAD\n");

  //This is a 2-byte instruction. Fetch the second byte, that is the
  //index of the local variable to be loaded onto the data stack

  if (!ijvmFetch(&arg1) ||              //get the local variable offset
      !ijvmGetVar(arg1, &varData1) ||     //look up the var by its index
      !ijvmStackPush((IjvmStackData)varData1)) {  // push its value
    state = STATE_RUN_ERROR;
    break;
  }
  break;
//---------------------------------------------------------------------
case OP_INVOKEVIRTUAL:
```

```
debug_printf("INVOKE\n");

//This is a 3-byte instruction. Fetch the arguments, stored as the
//second and third byte after the opcode. These two bytes form the
//absolute address (in the current implementation) of the function
//being called.
//NOTE: This is equivalent to getting the 2-byte index to the constant
//jump table that contains the absolute address. This approach is less
//indirect.

if (!ijvmFetch(&arg1) ||      //get function address upper byte
    !ijvmFetch(&arg2)) {      //get function address lower byte
  state = STATE_RUN_ERROR;
  break;
}

address = arg1 << 8 | arg2;       //compute target address (absolute)
regs.FP++;                        //change the frame
```

```
  frames[regs.FP].PC = address;     //set the new frame's target address


  //prepare the new frame context, i.e. the number of arguments to the
  //function being called and the number of local variables. This info
  //is stored in the function prefix code - 2 bytes at the function
  //target location, preceding the actual function code.


  if (!ijvmPrepareNewFrameContext()) {
    state = STATE_RUN_ERROR;
    break;
  }
  break;
//-------------------------------------------------------------------
case OP_IOR:
  debug_printf("IOR\n");


  //boolean (not bitwise) OR: arguments are popped from the stack and
  //the result is pushed onto the stack
```

```
  if (!ijvmStackPop(&stackData1) ||
      !ijvmStackPop(&stackData2) ||
      !ijvmStackPush(ALU_OR(stackData2, stackData1))) {

    state = STATE_RUN_ERROR;
    break;
  }
  break;
//--------------------------------------------------------------------
case OP_IRETURN:
  debug_printf("IRETURN\n");

  if (regs.FP == 0) {
    //this instruction cannot be used in the lowest
    //(main function) frames
    ijvmGlobalData.lastError = ERROR_IRETURN_LOWEST_FRAME;
    state = STATE_RUN_ERROR;
```

```
  break;
 }

 //change the frame to the lower one: previous context is automatically
 //restored (previous program counter & previous local variable info)
 regs.FP--;
 break;

//-------------------------------------------------------------------
case OP_ISTORE:
 debug_printf("ISTORE\n");

 //This is a 2-byte instruction. Fetch the second byte, that is the
 //index of the local variable to be assigned the value popped from
 //the data stack

 if (!ijvmFetch(&arg1) ||            //get the local variable offset
    !ijvmStackPop(&stackData1) ||    //pop the new var value to assign
```

```
    !ijvmSetVar(arg1, (IjvmVarData)stackData1)) { //assign the var
   state = STATE_RUN_ERROR;
   break;
  }
  break;
//--------------------------------------------------------------------
case OP_ISUB:
  debug_printf("ISUB\n");

  //integer sub instruction: arguments are popped from the stack and the
  //result is pushed onto the stack

  if (!ijvmStackPop(&stackData1) ||
    !ijvmStackPop(&stackData2) ||
    !ijvmStackPush(ALU_SUB(stackData2, stackData1))) {

   state = STATE_RUN_ERROR;
   break;
```

```
 }
 break;
//----------------------------------------------------------------------
case OP_LDC_W:
 debug_printf("LDC_W\n");

 //This is a 3-byte instruction. Fetch the arguments, stored as the
 //second and third byte after the opcode. These two bytes form the
 //_actual_ constant value in the current implementation.
 //NOTE: This is equivalent to getting the 2-byte index to the constant
 //table that contains the 2-byte constant value. This approach is less
 //indirect. Both ways require 2 bytes, so there is no preference.

 if (!ijvmFetch(&arg1) ||      //get constant value upper byte
     !ijvmFetch(&arg2)) {      //get constant value lower byte
  state = STATE_RUN_ERROR;
  break;
 }
```

```
  //construct the value of the constant to be pushed onto the stack
  stackData1 = (arg1 << 8) | (arg2);
  if (!ijvmStackPush(stackData1)) {
    state = STATE_RUN_ERROR;
  }
  break;
//---------------------------------------------------------------------
case OP_POP:
  debug_printf("POP\n");

  //pop (discard) the value at the top of the stack
  if (!ijvmStackPop(&stackData1)) {
    state = STATE_RUN_ERROR;
    break;
  }
  break;
//---------------------------------------------------------------------
```

```
case OP_SWAP:
 debug_printf("SWAP\n");

 //Swap the two topmost values on the stack. The following code pops
 //two values and pushes them back in reverse order.

 if (!ijvmStackPop(&stackData1) ||
    !ijvmStackPop(&stackData2) ||
    !ijvmStackPush(stackData1) ||
    !ijvmStackPush(stackData2)) {

  state = STATE_RUN_ERROR;
  break;
 }
 break;
//--------------------------------------------------------------------
case OP_WIDE:
 //No wide instructions in the current ISA
```

```
  ijvmGlobalData.lastError = ERROR_UNIMPLEMENTED;
  state = STATE_RUN_ERROR;
  break;
//---------------------------------------------------------------------
case OP_HALT:
  //The IJVM application has finished. Execution stops.
  debug_printf("HALT\n");
  state = STATE_HALTED;
  break;
//---------------------------------------------------------------------
case OP_ERR:
  //The IJVM application has finished due to an app error. Execution stops.
  ijvmGlobalData.lastError = ERROR_APPLICATION;
  state = STATE_RUN_ERROR;
  break;
//---------------------------------------------------------------------
case OP_OUT:
  debug_printf("OUT\n");
```

```
  //Execute the OUT instruction. The subtype of the OUT instruction is
  //at the top of the stack.

  if (!ijvmStackPop(&stackData1) ||        //get OUT intruction subtype
      !ijvmExecuteOut(stackData1)) {       //execute a particular OUT
    state = STATE_RUN_ERROR;
    break;
  }

  break;
//--------------------------------------------------------------------
case OP_IN:
  debug_printf("IN\n");

  //Execute the IN instruction. The subtype of the IN instruction is
  //at the top of the stack.
```

```
    if (!ijvmStackPop(&stackData1) ||        //get IN intruction subtype
        !ijvmExecuteIn(stackData1)) {        //execute a particular IN
      state = STATE_RUN_ERROR;
      break;
    }

    break;
   //--------------------------------------------------------------------
   default:
    //The opcode fetched is invalid
    ijvmGlobalData.lastError = ERROR_INVALID_OPCODE;
    state = STATE_RUN_ERROR;
  }
  break;

//==========================================================================
case STATE_STOPPED:
  //The virtual machine is idle, waiting for a request to start execution.
```

```
  break;

//==================================================================================
case STATE_LOADING:
  //The virtual machine has been requested to start execution. Open the
  //pre-compiled executable file.

  ijvmGlobalData.status = STATUS_LOADING;

  execFile = filesysFileOpen(IJVM_EXEC_FILE_INDEX, OPEN_READ);
  if (execFile == FILE_ID_INVALID) {

    //The file cannot be opened (it hasn't been compiled yet, for ex.)
    ijvmGlobalData.lastError = ERROR_CANT_LOAD_EXEC_FILE;
    state = STATE_RUN_ERROR;
  }
  else {
```

```
    //The executable file has been opened, prepare the virtual machine for
    //a new execution
    state = STATE_PREPARE_RUN;
  }
  break;

//==================================================================
case STATE_PREPARE_RUN:

  //Initialize the virtual machine registers. If the executable file
  //has a complex structure with multiple sections, it will be dealt with here.

  if (!ijvmLoad()) {

    //Can't load. The executable file is corrupt.
    ijvmGlobalData.lastError = ERROR_EXEC_FILE_INVALID;
    filesysFileClose(execFile);
    execFile = FILE_ID_INVALID;
```

```
  state = STATE_RUN_ERROR;
  break;
 }

 //redirect the IO streams to the IJVM applications, so only application's
 //IN and OUT instructions will have access to the inderlying IO.
 ijvmioSetIOConsumer(IO_CONSUMER_APP);

 ijvmGlobalData.lastError = ERROR_NONE;      //clear the errors
 ijvmGlobalData.status = STATUS_RUNNING;     //update the status
 state = STATE_RUNNING;
 break;


//==========================================================================
case STATE_RUN_ERROR:
 filesysFileClose(execFile);                //close the executable image file
 execFile = FILE_ID_INVALID;
```

```
  ijvmGlobalData.status = STATUS_RUN_ERROR;   //set error status

  //redirect the IO streams back to the virtual machine needs, the console
  //etc., away from the application that's been stopped.

  ijvmioSetIOConsumer(IO_CONSUMER_JVM);
  debug_printf("Runtime error\n");
  state = STATE_STOPPED;
  break;

//==========================================================================
case STATE_HALTED:
  filesysFileClose(execFile);               //close the executable image file
  execFile = FILE_ID_INVALID;
  ijvmGlobalData.status = STATUS_COMPLETED;   //set success status

  //redirect the IO streams back to the virtual machine needs, the console
  //etc., away from the application that's been stopped.
```

```
      ijvmioSetIOConsumer(IO_CONSUMER_JVM);

      debug_printf("Halted\n");

      state = STATE_STOPPED;

      break;

  }


  return 1;

}


//----------------------------------------------------------------------------

// ijvmStartExecution: this function is called by external modules (console) to

//   request the start of the execution. Usually, this means that a precompiled

//   image has been created.

// arguments: none

// return: 1 if success, 0 if failure

//----------------------------------------------------------------------------

int ijvmStartExecution(void)
```

```
{
  if (!moduleInitialized) {
    return 0;
  }

  if (state == STATE_STOPPED) {
    state = STATE_LOADING;
  }

  return 1;
```

IJVM Code

```
.constant
K_pound  0x23  //pound key ascii code
NumBoards 0x23  //OUT subtype code
BoardID 0x42    //OUT subtype code
Slave 0x53         //IN/OUT subtype code
Master 0x4D     //IN/OUT subtype code
Multiply 0x58   //OUT subtype code
Divide 0x44     //OUT subtype code
```

```
Mod 0x25        //OUT subtype code

Sqrt 0x51       //OUT subtype code

Keypad 0x4B     //IN subtype code

Thousand 1000

TenThousand 10000


.end-constant


.main

.var


index           //the starting number to check primality

step            //every step'th number is checked beginning at index

inputrange      //max number to find primes up to, for all boards

boardID                 //current board ID

numBoards       //total number of boards on the chain


//variables to hold the keypressed digits
```

```
d1
d2
d3
d4
d5
currentkey

.end-var
        //clear vars
        BIPUSH 0
        DUP
        DUP
        DUP
        DUP
        ISTORE index
        ISTORE inputrange
        ISTORE boardID
```

```
ISTORE numBoards

ISTORE currentkey


//init key digits to '0' ascii

BIPUSH 0x30

DUP

DUP

DUP

DUP

ISTORE d1

ISTORE d2

ISTORE d3

ISTORE d4

ISTORE d5


//get the number of boards from IJVM

LDC_W NumBoards

IN
```

```
ISTORE numBoards

//get board ID from IJVM
LDC_W BoardID
IN
ISTORE boardID

//are we master or slave?
ILOAD boardID
IFEQ newkey     // 0 is the master, read keypad for inputrange



//The slave reads inputrange from the master


INVOKEVIRTUAL GetRange


ISTORE inputrange
GOTO msend
```

```
        // The master reads the keypad
        // compose the range from the d's
        // each must be offset by -0x30
        // d5 is Most significant
        // d1 is least significant

newkey:     INVOKEVIRTUAL getKey //call a blocking method, when it returns a new key is on the stack

        // if key is pound, branch
        ISTORE currentkey

        ILOAD currentkey
        LDC_W K_pound
        ISUB
        IFEQ keypad_accept
```

```
// discard digit 5, load
// everything from low to high
ILOAD d4
ISTORE d5

ILOAD d3
ISTORE d4

ILOAD d2
ISTORE d3

ILOAD d1
ISTORE d2

ILOAD currentkey
ISTORE d1
```

```
    GOTO newkey

keypad_accept: ILOAD d1
    BIPUSH 0x30
    ISUB
    ISTORE inputrange

    // d2*10
    ILOAD d2
    BIPUSH 0x30
    ISUB
    BIPUSH 10
    LDC_W Multiply
    OUT

    // inputrange += d2*10
```

```
ILOAD inputrange

IADD

ISTORE inputrange


// d3*100

ILOAD d3

BIPUSH 0x30

ISUB

BIPUSH 100


LDC_W Multiply

OUT


// inputrange += d3*100

ILOAD inputrange

IADD

ISTORE inputrange
```

```
// d4*1000

ILOAD d4
BIPUSH 0x30
ISUB
LDC_W Thousand
LDC_W Multiply
OUT

// inputrange += d4*1000
ILOAD inputrange
IADD
ISTORE inputrange

// d5*10000
ILOAD d5
BIPUSH 0x30
ISUB
```

```
            LDC_W TenThousand
            LDC_W Multiply
            OUT

            // inputrange += d5*1000
            ILOAD inputrange
            IADD
            ISTORE inputrange



msend: NOP
            //master has inputrange -> send to slave
            // or
            //slave receives inputrange -> forward to its slaves

            ILOAD inputrange
            LDC_W  Slave
            OUT
```

```
//initialize index = BoardID*2 + 1

ILOAD boardID

ILOAD boardID

IADD            //BoardID*2

BIPUSH 1

IADD            //BoardID*2 + 1

ISTORE index


//initialize step with 2*numBoards

//we skip even numbers and interleave the

//workload among the boards


ILOAD numBoards

ILOAD numBoards

IADD

ISTORE step
```

```
        //The main primesearch loop

again:  ILOAD index

        //check if we're done computing up to inputrange
        DUP
        ILOAD inputrange
        SWAP
        ISUB
        IFLT done

        //index (on the stack) is the current number to prime-check
        INVOKEVIRTUAL isPrime

        IFEQ nope

        //index is prime, send to master
        ILOAD index
```

```
        BIPUSH 0x4D          //out std master
        OUT

        //index is not prime, now check slave boards
nope:   LDC_W Slave
        IN
        DUP

        //if any of the slaves has sent us a newly found prime
        //number, forward it to our master

        IFEQ sbusy
        DUP
        LDC_W Master
        OUT

sbusy: POP
```

```
        //go to next index to check its primality
        ILOAD step
        ILOAD index
        IADD                    //index += step
        ISTORE index

        GOTO again
done:   HALT
.end-main



//This function waits for the master to
//pas it the keypad-typed range

.method GetRange()
.var
.end-var
        BIPUSH 0                //push dummy
```

```
read:   POP
        LDC_W Master
        IN                      //read master serial port
        DUP
        IFEQ read
        IRETURN

.end-method


//Read keypad (blocking) returns only when key is pressed
.method getKey()
.var
.end-var
        BIPUSH 0                //push dummy
keypad: POP
        LDC_W Keypad
        IN
```

```
        DUP
        IFEQ keypad
        IRETURN
.end-method

//Check x for primality using a naive approach
.method isPrime(x)
.var
top
p
tmp
.end-var

        ILOAD x
        IFEQ not_prime          //0 not prime
        ILOAD x
        BIPUSH 1
        IF_ICMPEQ not_prime   //1 not prime
```

```
ILOAD x
BIPUSH 2
IF_ICMPEQ is_prime      //2 is prime
ILOAD x
BIPUSH 3
IF_ICMPEQ is_prime      //3 is prime

ILOAD x
BIPUSH 2
LDC_W Mod
OUT                     //mod
IFEQ not_prime          //multiple of 2 => not prime

ILOAD x
BIPUSH 3
LDC_W Mod
OUT                     //mod
IFEQ not_prime          //multiple of 3 => not prime
```

```
        ILOAD x
        LDC_W Sqrt
        OUT                 //sqrt
        ISTORE top          //get the max number to check : sqrt(x)


        BIPUSH 5            //start with p=5
        ISTORE p
next:   ILOAD top
        ILOAD p
        ISUB
        IFLT is_prime       //checked all numbers 6k+1 & 6k-i < sqrt(x) => x is prime

        ILOAD x
        ILOAD p
        LDC_W Mod
        OUT                 //mod
```

```
IFEQ not_prime          //divisible by 6k-1 => not prime

ILOAD x
BIPUSH 2
ILOAD p
IADD                    //p+=2 (i.e. p = (6k-1) + 2 = 6p + 1)
DUP
ISTORE p
LDC_W Mod
OUT                     //mod
IFEQ not_prime          //divisible by 6k+1 => not prime

BIPUSH 4
ILOAD p
IADD                    //p+=4 (i.e. next p=6k'-1)
ISTORE p

GOTO next
```

```
not_prime: BIPUSH 0
           IRETURN
is_prime: BIPUSH 1
           IRETURN

.end-method
~
```

CPLD Code

```
LIBRARY ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity calculate is
 port (
  CLK : in std_logic;
  start_op : in std_logic;
  number : in unsigned(7 downto 0);
  result : out unsigned(3 downto 0);
  op_done : out std_logic
 );
end calculate;

architecture sqroot OF calculate IS
```

```
function SquareRoot (Arg: unsigned) return unsigned is

    constant AMSB: integer:= number'length-1;
    constant RMSB: integer:= (number'length/2) - 1;
    variable Root: unsigned(RMSB downto 0);
    variable Test: unsigned(RMSB+1 downto 0);
    variable Rest: unsigned(AMSB+1 downto 0);
-- variable temp: unsigned(number'length downto 0) := number;

begin

    Root := (others => '0');
    Rest := '0' & Arg;

    for i in RMSB downto 0 loop

       Test := Root(RMSB-1 downto 0 ) & "01";
       if Test(RMSB-i+1 downto 0) > Rest(AMSB+1 downto 2*i) then
           Root := Root(RMSB-1 downto 0) & '0';
       else
           Root := Root(RMSB-1 downto 0) & '1';
           Rest(AMSB downto i*2) := Rest(AMSB downto i*2) - Test(RMSB-i+1 downto 0);
       end if;

    end loop;

 op_done <= '1';
    return Root;

end;

begin
```

```
  Uut : process(CLK, start_op, number)
  begin
 if rising_edge(CLK) and start_op='1' then
  op_done <= '0';
--  temp := number;
  result <= SquareRoot(number);
 end if;

  end process;

end sqroot;
```