A Java Bytecode Compiler for the 8051 Micro-controller

Tsakani Joseph Mbhambhu

Thesis presented in fulfillment of the requirements for the degree of
Master Of Science in Engineering Science
at the University of Stellenbosch

Supervisor: Dr M.M. Blanckenberg

March 2002

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree

Tsakani Joseph Mbhambhu Date

Abstract

This report describes the development of a Java Bytecode Compiler (JBC) for the 8051 micro-controller. Bytecodes are found in the class file generated when a Java source file is compiled with the java compiler (javac). On Java platforms, the Java Virtual Machine (JVM) interprets and executes the bytecodes. Currently existing Java platforms do not support programming the 8051 using Java.

As an 8-bit micro-controller with only 64 KB of total memory, the 8051's word size and memory is too limited to implement a JVM. Moreover, full applications of the 8051 require that it handles hardware interrupts and access I/O ports and special registers.

This thesis proposes a JBC to compile the standard bytecodes found in the class file and generate equivalent assembly code that can run on the 8051.

The JBC was tested on the 8051 compatible AT89C52*44 micro-controller with a program that simulates an irrigation controller. The code generated by the JBC executes correctly but is large in size and runs slower than code of a program written in assembly.

Conclusions drawn are that the JBC can be used to compile Java programs intended for the 8051 and its family of micro-controllers. In particular, it is especially a good tool for people who prefer Java to other languages. The JBC is suitable for smaller programs that do not have efficiency as a major requirement.

Opsomming

Hierdie tesis beskryf die ontwikkeling van `n Java "Bytecode" samesteller (Java Bytecode Compiler, JBC) vir die 8051 mikro beheerder argitektuur. "Bytecodes" is die produk van die standaard Java samesteller "javac" en word deur `n platform spesifieke Java Virtuele Masjien gelees en uitgevoer. Geen JVM is huidig beskikbaar vir die 8051 argitektuur nie.

Die gekose 8–bis 8051 mikro beheerder het `n beperkte interne geheue van 64kB. Hierdie beperking maak dit nie geskik vir `n JVM nie. Daar moet ook voorsiening gemaak word om hardeware onderbrekings te hantering en te kan kommunikeer met die poorte en spesiale registers van die mikro beheerder.

JBC word ontwikkel wat die standaard "Bytecode" kompileer na geskikte masjien kode wat dan op die mikro beheerder gebruik kan word.

Die JBC is ontwikkel en toets en `n eenvoudige besproeiing program is geskryf om op `n Atmel AT89C52*44 te loop. Die kode werk goed maar is nog nie geoptimeer nie en loop onnodig stadig. Optimerings metodes word aanbeveel en bespreek.

Die gevolgtrekking is dat die huidige JBC kan gebruik word om Java kode te skryf vir die 8051 beheerder. Dit maak die hardeware platform nou beskikbaar aan Java programmeerders. Daar moet wel gelet word dat die JBC op die oomblik net geskik is vir klein programme en waar spoed nie die primêre vereiste is nie.

To my daughters Amukelani and Rivoningo

Acknowledgements

I would like to extend my gratitude to all people who have contributed to this work one way or another, and to those whose curiosity kept me on my toes. I would like to thank, in particular, the following people:

My supervisor, Dr M.M. Blanckenberg, for continuous guidance and for sharing his insight, and especially, for his patience during the difficult times of the project. Also thanks to my mentor, Mr. A.N. Rust (and later Mr. H. van der Merwe).

I would also like to thank and extend my gratefulness to my family for their continuous support and motivation and most importantly, for believing in me; and to my classmates, colleagues and friends for having been of help one way or another.

Thanks also to the Department of Communications for financial support.

Lastly, I would like to thank God for helping me stay focussed and determined throughout this research, especially during the difficult times; most of which were not academically related. Thank you for also helping me find my heart.

Contents

List of Figures

List of Tables

Abbreviations

API	-	Application Programming Interface
CLDC	-	Connected Limited Device Configuration
DPH	-	Data Pointer High Byte
DPL	-	Data Pointer Low Byte
DPTR	-	Data Pointer
EEPROM	-	Electrically Erasable Programmable Read Only Memory
EPROM	-	Erasable Programmable Read Only Memory
FPGA	-	Field Programmable Gate Array
FQN	-	Fully Qualified Name
IDC	-	International Data Corporation
IE	-	Interrupt Enable
IP	-	Interrupt Priority
ISR	-	Interrupt Service Routine
JBC	-	Java Bytecode Compiler
JIT	-	Just-In-Time
JVM	-	Java Virtual Machine
KVM	-	K Virtual Machine
LCD	-	Liquid Crystal Display
LED	-	Light Emitting Diode
LVA	-	Local Variable Area
NC	-	Network Computer
OBC	-	On-Board Computer
OS	-	Operand Stack
PC	-	Personal Computer
PCON	-	Power Control
PDA	-	Personal Digital Assistant
RAM	-	Random Accessible Memory
ROM	-	Read Only Memory

RI	-	Receive Flag
SBUF	-	Serial Buffer
SCON	-	Serial Control
SFR	-	Special Function Register
SP	-	Stack Pointer
SUNSAT	-	Stellenbosch University Satellite
TCON	-	Timer Control
TI	-	Transmit Flag
THx	-	Timer x High Byte
TLx	-	Timer x Low Byte
TMOD	-	Timer Mode
UART	-	Universal Asynchronous Receive Transmit
VHDL	-	Very High Speed Integrated Circuit Hardware Description Language

"Diligence is the mother of good luck, and God gives all things to industry…"

- Benjamin Franklin

Chapter 1

Introduction

The impact of technology on everyday lives is increasing by the day. Most of the devices used today depend on computer technology or are themselves computerised in some way. In many of these applications use of embedded micro-controllers is common. An embedded micro-controller is loaded with a program and probably runs that program for its entire life (Schultz, 1998). The 8031 micro-controller, for instance, was used together with the On-Board Computer (OBC) on SUNSAT to determine the state of the satellite by keeping the list of the telecommand signals (Steenkamp, 1999:111). Traditionally, micro-controller programming is supported in only a few languages. The 8051, being one of the members of the MCS-51™ family of micro-controllers, for example, is only supported in assembly, PL/M, C, Forth and Basic (Schultz, 1998:84).

With the introduction of the Java technology, the approach to embedded programming has now extended beyond the traditional micro-controller languages. The Java technology was itself originally intended for embedded systems (Computer Technology Research Corporation, 1998:54). As the Computer Technology Research Corporation indicate, the International Data Corporation (IDC) has predicted that while today's Internet use largely involves PCs, the use of devices such as telephones, Network Computers (NCs), watches, personal digital assistants (PDAs), set-top boxes, and many more will expand the market substantially. Not suprising, the Department of Electrical and Electronic Engineering of the University of Stellenbosch together with the Government's Department of Communications would like to build a satellite programmed in Java.

This research focuses on the 8051 and its family of micro-controllers as a potential device to program in Java on the satellite. In addition to programming devices on the satellite in Java, a solution that can be found for programming the 8051 microcontroller in Java would easily be extendible to general micro-controller applications. Moreover, although Java is a complex language, it is far simpler than assembly language and easier to program correctly than C (Computer Technology Research Corporation, 1998:58). Code written in Java is also small and easy to manage.

1.1 The current Java technology for Embedded Systems

Java source code is compiled by the Java compiler (javac) to a classfile. The classfile is itself a binary file that consists of bytecodes. The bytecodes are interpreted by the Java Virtual Machine (JVM). The JVM then generates executable code that can run on the environment in which it is installed. This concept is shown in figure 1.1 below:

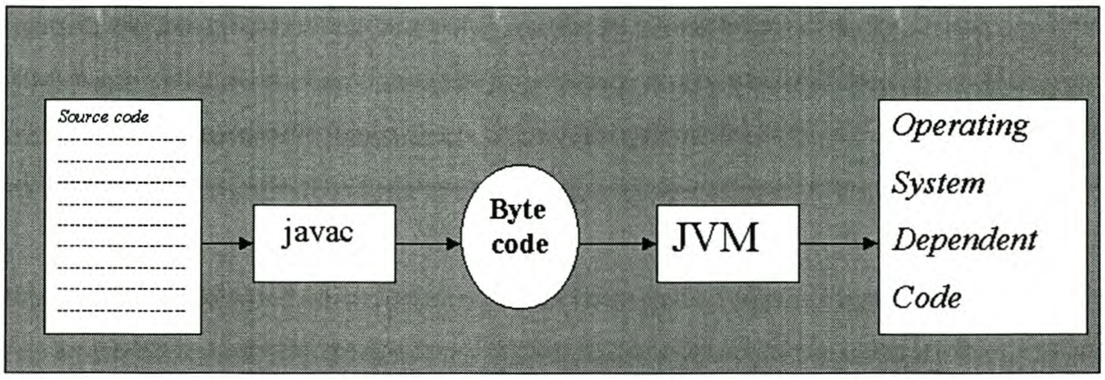


Figure 1.1: Source code generated to executable code

Although the JVM suffices for many applications, other JVMs may contain a Just-In-Time (JIT) native compiler (Lewis, 1998:15) and the adaptive optimizer (Venners, 1999:06). The JIT compiler is responsible for converting the bytecodes into a native executable code. The adaptive optimizer compiles only frequently used bytecodes into native code.

The core to the execution environment for the JVM is the Java Platform. The platform, as Venners (1999:25) indicates, acts as a buffer between the running program and the underlying hardware and operating system. A Java Platform is a runtime system that consists of the JVM and the Java Application Programming Interface (API). The API provides the classes that may be needed by a running program.

Berg and Fritzinger (1999:04) discuss four Java environments, viz., general workstation platform, Personal Java platform, embedded Java platform and JavaCard platform. The general workstation consists of API's defined in Sun Microsystem's *Java 2 Standard Edition*. Since Java was initially planned to be the technology for embedded systems, the general platform was made to be compact so that it can be implemented in software using the resources available to embedded systems (Venners, 1999:27). Venners continues to indicate that because of the limited resources for embedded systems (no hard disk, no graphical display and no display), Sun Microsystems created smaller API's with requirements for embedded and consumer systems. This led to the introduction of the latter four platforms mentioned above.

Sun Microsystem's APIs also indicate that they were meant for processors with larger memory footprint than that of the 8051. Thus, none of the current API development supports the 8051. In an electronic mail correspondence with Lindholm (2000), the Connected Limited Device Configuration (CLDC) was suggested as being the smallest API in Sun Microsystems to date. The CLDC uses a smaller JVM called the K Virtual Machine (KVM). Lindholm, though reluctantly, suggests that the CLDC could be ported to the 8051 and the JIT added to it. The CLDC is, however, meant for 16 bit to 32 bit processors (Sun, 2001b:02).

The hardware requirements in the final CLDC specification release by Sun Microsytems (2000a:2-2) also shows that CLDC's JVM was itself designed for processors with a budget of at least 160 kB to 512 kB of memory. Specifically, the specification assumes that the hardware will have at least 128 kB of non-volatile memory for the JVM and CLDC libraries. It will also have 32 kB volatile memory for Java run time and object memory (Sun Microsytems, 2000a:2-3). In addition to this, the software requirement in the CLDC specification assumes that a minimal host operating system or kernel is available to manage the underlying hardware. This host operating system must provide at least one schedulable entity to run the JVM (Sun Microsytems, 2000:2-3).

1.2. The need for a Java Bytecode Compiler

As can be seen from the discussion in section 1.1 above, all the existing APIs mentioned do not extend the Java technology to memory constrained micro-controllers such as the 8051 or any of the members of its family of micro-controllers. Moreover, 8051 applications include operations on its Special Function Registers (SFR) and handling of interrupts. The task in this research is to find a solution for the 8051 to be able to run Java written programs without changing the architecture of the Java technology and its advantages while maintaining normal 8051 operations (see figure 1.2).

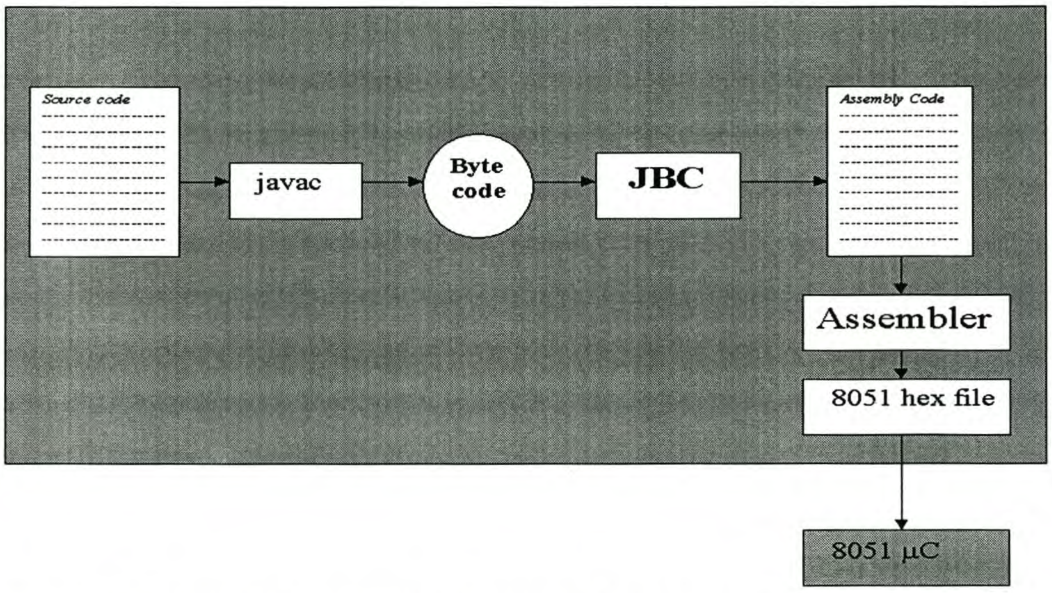


Figure 1.2: Source code generated to executable code

Possible solutions to this problem include using the CLDC as suggested by Lindholm. The problem, however, lies in the fact that the CLDC itself is meant for 16-bit to 32-bit processors. Moreover, hardware requirements for the CLDC are far beyond the 8051's 64 kB memory capacity. The solution is, probably, to compile bytecodes into the 8051's assembly language as shown in figure 1.2.

As can be seen from the figure, the JVM will not have to be ported to the 8051. Similar to the JVM, the JBC will compile the bytecodes and generate an assembly program intended for the 8051. The assembly program will then be compiled using the assembler. The resulting binary file can then be run on the 8051. Section 1.3 gives the outline of the approach to this solution.

1.3. Thesis Layout

The approach to this solution is to study the architecture of the JVM and how it interprets the bytecodes to generate the operating system dependent executable code. The requirements of the JVM to execute the bytecodes are investigated in chapter 2. Section 2.1 discusses the structure of the JVM. The structure of the classfile is discussed in section 2.2.

Chapter 3 discusses the 8051. The architecture of the MCS-51™ family of micro-controllers is discussed in section 3.1. Section 3.2 discusses the limitations of the 8051 with respect to the JVM requirements investigated in chapter 2.

Decisions made in order to implement the JBC are discussed in chapter 4. Section 4.1 discusses how the 8051 limits can be addressed. This includes a discussion as to how the interrupts and the SFRs can be handled in Java. Section 4.2 discusses the further implementation decisions that relate to the run time data areas need to run Java programs.

Chapter 5 discusses the actual implementation of the JBC. The flowcharts of how the JBC generates assembly code from the bytecodes are shown. An example is used to illustrate code that would be generated throughout various stages of implementation. The possible code that is generated from the example given is shown in appendix E.

The JBC is tested in chapter 6 by means of a program written to simulate the irrigation controller. A program written in Java, C and assembly language is also presented in order to compare the size of the executable code that is generated after compiling with the JBC, Keil Software's μVision compiler and asm compiler, respectively. Section 6.1 discusses the program and section 6.2 presents the results of the program. These results are discussed in section 6.3 and evaluated against the implementation decisions made in chapter 4. Issues of speed and efficiency and readability are also discussed.

Chapter 7 gives the conclusions drawn from this research. It culminates by giving recommendations based on the observations made in chapter 6. Most of the recommendations relate to the way in which the JBC can be optimized.

Appendix A shows flowcharts of methods called in the main program described in chapter 6. Appendix B gives some additional tables referred to in chapter 2. Appendix C gives a list of the supported instruction set. The API for the JBC software is given in Appendix D. Appendix E gives the listing of the programs that are mentioned throughout the chapters.

Chapter 2

The JVM and the Java Bytecode

Most programming languages including C and C++ are compiled into an executable machine language. Such languages are called compiled languages. Java, on the other hand, is an interpreted language whose source code is compiled to a binary file known as the class file. The JVM is needed to interpret instructions carried in the bytecodes of the class file and to execute them in accordance to the platform in which it is installed. This is illustrated in figure 2.1 below:

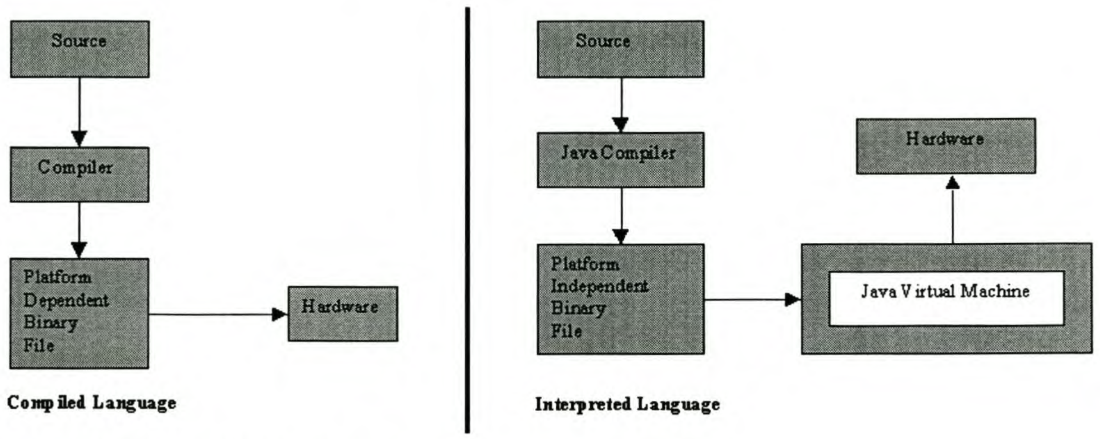


Figure 2.1: Compiled vs. interpreted language

The platform dependent binary file generated for compiled languages is an executable file that runs on a specific platform. This means, as the Computer Technology Research Corporation (1998:33) indicates, that any changes to the program require that it is recompiled on that platform. The Java compiler handles this problem by compiling source code into a platform independent binary file as shown in figure 2.1. This makes, as supported by Winder (2000:11), programs written in Java to be architecturally neutral as their executable representation is independent of the physical and operating system in which they run.

This chapter discusses the JVM and how it interprets the Java bytecode found in the class file. The JVM structure is examined first. The classfile structure will then be discussed.

2.1. The Structure of the JVM

The JVM consists of a class loader and the execution engine. The class loader is responsible for loading into memory the class files and any other classes from the API that may be referenced by the program. The execution engine is responsible for executing the bytecodes found in the class file. Other JVMs may contain a JIT native compiler (Lewis, 1998:15) and the adaptive optimizer (Venners, 1999:06). The JIT compiler is responsible for converting the bytecodes into a native executable code. The adaptive optimizer compiles only frequently used bytecodes into native code. In addition to the class loader and the execution engine, the JVM also consists of a debug interface, an applet security manager and a garbage collector. The applet security manager and the bytecode verifier are responsible for making sure that the loaded program does not violate the security of the host system. The garbage collector is responsible for reclaiming memory occupied by objects that are no longer needed by the program. The garbage collector will be discussed again in chapter 4.

The JVM is not a physical machine. It is rather an abstract (Venners, 1999:136) machine that manipulates various memory areas as it interprets or executes the class file. Run time data areas are needed by the JVM as it executes Java programs. These data areas are needed in order to get and store objects, method parameters, local variables, intermediate results of computations, bytecodes, and so on. Figure 2.2 below is an extract from (Venners, 1999:137). It shows the internal architecture of the JVM.

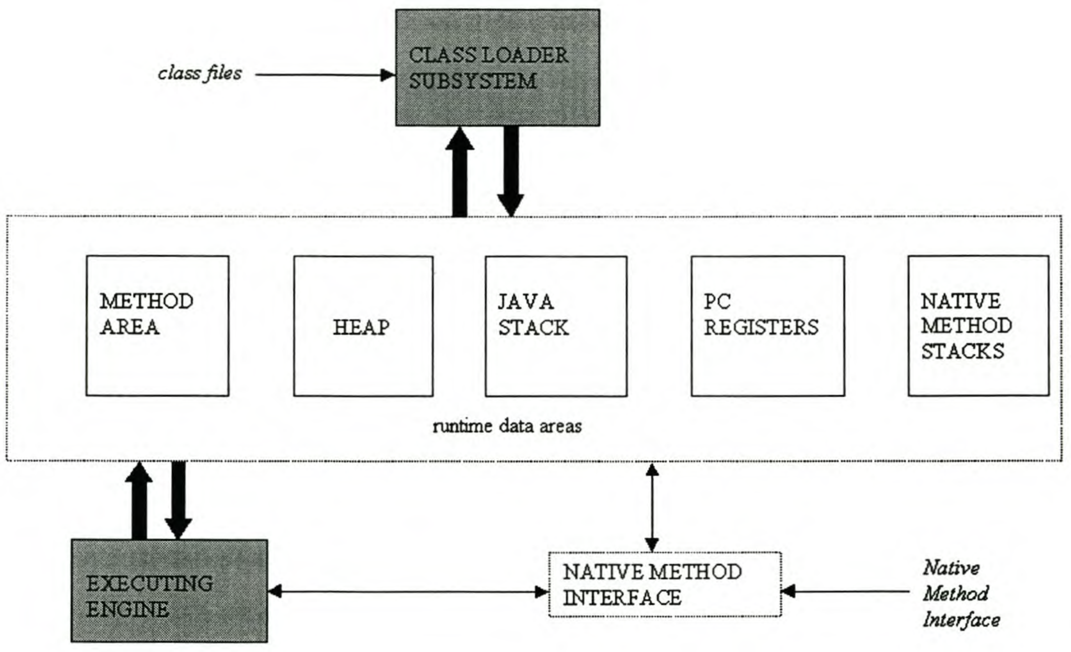


Figure 2.2: Internal Architecture of the JVM (Venners, 1999:137)

As shown in the figure, the JVM has one method area and one heap. Information about the methods is placed in the Method area and objects instantiated are placed on the heap. Each thread formed gets its own PC register and the Java Stack. The PC register points to instructions to be executed.

A Java Stack handles the states of the methods. A state includes local variables, parameters with which the method was invoked, immediate calculations and the state's return value. Method states are represented as frames. A new frame is pushed into a Java Stack when a method is invoked and popped when the method returns. Like the JVM, the JBC should be able to manipulate run time data areas. The next section discusses runtime data areas in detail. Further details for the rest of section 2 can also be obtained from the JVM specification (Lindholm & Yellin, 1999).

2.1.1. Method area

An implementation will need to find data relating to the class. This data is stored in the Method area. Data stored in the Method area includes class structures such as the runtime constant pool, field and method data, and the code for methods and constructors. The memory allocated to the method area can expand and contract as needed.

Other items included in the method area are the method tables and method data. The method data should include the size of local variables and Operand stack (OS) for the Java Stack, method's bytecode and exception table. Data for objects is stored in the heap discussed in the next section.

2.1.2. Heap

A heap is a place for objects. All threads of a single application share one heap. However, different applications have different heaps. The heap's memory is managed by garbage collecting objects no longer in use. The heap's memory may also expand or contract as needed. The heap should have information about class instance variables and all class superclasses.

Given the object reference, an implementation should be able to find the instance data for that object and find a way of accessing this data. Thus memory allocated in the heap should include a pointer to the Method area. Venners (1999:154) discusses various heap designs.

2.1.3. Java Stack

In addition to the method area and the heap, an implementation should also have a Java Stack for running methods. The main component of the Java Stack is a stack frame. The stack frame itself consists of the local variables, Operand stack (OS) and frame data. Information about size is checked from words found in the class data in the method area. A Stack frame can be visualised as shown in figure 2.3 below:

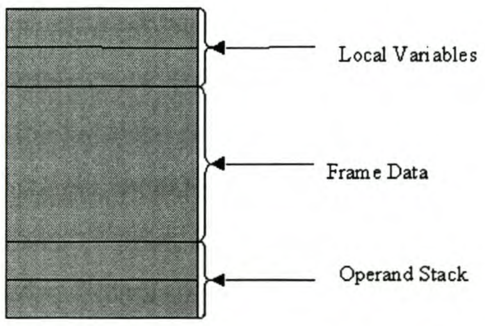


Figure 2.3: Stack Frame (Venners, 1999:169)

The Local Variable Area (LVA) is an array of method parameters. Values occupy one entry (word) in the array, except for *long* and *double* which occupy two words each. Note that the Java word is 32 bits in size. Data types supported by the JVM are given in appendix B3. The OS is also organised as an array of words. Pushing into and popping from the Stack accesses elements on the OS. Venners (1999:167) shows how the operation on the OS can be visualised. The Frame Data consists of information that supports constant pool resolution, normal method invocation and exception dispatch. An implementation may require data to be accessed from the LVA and placed in the OS, and vice versa.

Invoking a method involves switching from the current stack frame to a new frame. When a method is invoked, a new frame is created and pushed onto the Java stack. The frame becomes a current frame – all computations, local variables, parameters and other data are stored and done here. The frame is popped and discarded when the method completes. In order to read the class file, the program needs to know the structural details of the class file. The next section discusses the structure of the class file.

2.2. The Structure of the class File

This section discusses certain items of the class file that will form the core of the JBC operations. The discussion in the section is based on the JVM specification (Lindholm & Yellin, 1999). Further details about the work in this section can also be found from (Venners, 1999).

The class file is made up of a set of 8-bit unsigned bytes. In the class file notation, these bytes are referred to as u1, u2, etc; for unsigned one- and two-byte numbers, respectively. The contents of the class file structure are referred to as items. A class file consists of Java bytecodes that represent a small instruction set. In addition to the bytecodes, the class file consists of a symbol table and other ancillary information. Figure 2.4 below shows items of the class file structure in the order of their appearance.

```
ClassFile{
     u4 magic;
     u2 minor_version;
     u2 major_version;
     u2 constant_pool_count;
     cp_info constant_pool[constant_pool_count];
     u2 access_flags;
     u2 this_class;
     u2 super_version;
     u2 major_class;
     u2 interfaces_count;
     u2 interfaces[interfaces_count];
     u2 fields_count;
     field_info fields[fields_count];
     u2 method_count
     method_info methods[method_count];
     u2 attributes_count;
     attribute_info attributes[attributes_count];
}
```

Figure 2.4: The class file structure (Lindholm & Yellin, 1999:94)

The representation in figure 2.4 is a C-like pseudo code that describes class file structure. Table B3 in Appendix B gives a brief description of each of the class file items of figure 2.4. The class file structure is followed when reading and interpreting the bytecode in the class file. Most of the items shown in figure 2.4 have further structures within themselves. The *method_info* structure, for example, consists of the code attribute that also has further structures. Some of these structures will be discussed in this section.

The constant pool is mostly referred to in order to find information about the class, fields and methods. The rest of this section will focus on the constant pool, fields and method structures as they contain most of the information needed when interpreting the bytecodes.

2.2.1. Constant pool

The Constant pool contains constants such as literal strings, final variable values, class names and method names. In addition to these, the Constant pool contains information for symbolic references such as Fully Qualified Names (FQN), Field Names and descriptors and method names and descriptors (see the specification for details). A program may at times need to refer to a particular index of the constant pool to find the contents or reference to the method in that index. Each entry in the constant pool represents a specific constant pool structure. The entry begins with a tag shown in figure 2.5 followed by the structure of the constant pool structure referred to by the tag. The constant pool structure or *cp_info* is shown in figure 2.5.

```
cp_info{
    u1 tag;
    u1 info[];
}
```

Figure 2.5: Constant pool structure (Lindholm & Yellin, 1999:103)

For each entry in the Constant pool, i.e., the *cp_info*, the value of the first *u1* byte gives the JVM information about the contents in the entry.

2.2.2. Fields

A field is a class or instance variable of the class or interface. These are variables that are given in the program during initialisation of the class. Similar to the Constant pool, the Field structure has a specific structure or format that a JVM must follow. Details of the Field structure can be found in the specification.

2.2.3. Methods

The methods written in the Java program are represented in the class file structure by the *method_info* structure. The *method_info* structure of the class file includes items which give information about the access flags of the method (that is, whether the method is public, private, etc), the name of the method, the descriptor and attributes where the code of the method is found. The code attribute contains the bytecode sequence. A large portion of this research focuses on the bytecode sequence that will be produced for each method. These and other items of the *method_info* structure will be explained in this section. The *method_info* or simply the method structure is shown in figure 2.6 taken from the JVM specification.

```
method_info{
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

Figure 2.6: methods structure (Lindholm & Yellin, 1999:114)

The value *u2 access_flags* gives information about the accessing nature of the method. The value of the next two unsigned bytes, *u2 name_index*, specifies the index of the constant pool where to find the name of the method. In other words, at that index, the constant pool entry is a *utf8_string* that represents the name of the method or just another value which specifies where to get the name in the Constant pool. The specification specifies exactly how the name of the method is found from the *u2 name_index*. The *u2 descriptor index* follows the same principle as the *u2 name_index*. The method attributes, *u2 attribute_count* and *the attribute_info attributes[attribute_count]*, give information about additional information for the method. The method attribute has a compulsory attribute – the *Code_attribute*. Other method attributes are *Exception, Synthetic* and *Deprecated* attributes. The specification shows the code attribute shown in figure 2.7.

The explanation of the *Code_attribute* is similar to that of the Method structure and other structures explained in the preceding section. Some areas of interest include the *u2 max_locals*, the value of which gives the number of parameters that have to be passed when this method is called, that is to say, the number of arguments associated with this method. The code attribute structure is shown in figure 2.7 below taken from the JVM specification.

```
Code_attribute{
        u2 attribute_name_index;
        u4 attribute_length;
        u2 max_stack;
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
        {
                u2 start_pc;
                u2 end_pc;
                u2 handler_pc;
                u2 catch_type;
        }exception_table[exception_table_length];
        u2 attribute_count;
        attribure_info attributes[attribute_count];
}
```

Figure 2.7: Code attribute structure (Lindholm & Yellin, 1999:120)

The *u4 code_length*, gives information about the size of the bytecode sequence within this code. The *u1 code[code_length]* is a variable item of the *Code_attribute*. This is the actual bytecode that the JBC will have to interpret and generate the equivalent 8051 assembly code. As mentioned earlier, a large proportion of this research focuses on this part and decisions are made here as to what to do for each bytecode read. Note also that, while the *Code_attribute* is itself a method attribute, it too can have further attributes. The specification defines two *Code_attributes* attributes, viz., *LineNumberTable* and *LocalVariableTable* attributes. These attributes contain debugging information and can be ignored.

2.2.4. Attributes

Attributes can appear in various items of the class file structure, an example is the *Code_attribute* just described, which also has further attributes. The purpose of the attributes is to give further information about a particular item. Attributes appearing in the main class file structure include the *SourceFile* and the *Deprecated* attributes. According to the specification, an implementation is allowed to silently ignore attributes it does not recognise, as long as the overall semantics of the class file is still valid and by the same token, more attributes can be added.

Summary

An implementation of the Java technology should be able to create run time data areas in which to execute the bytecodes found in the class file. The Java run time data areas are organized in the form of a Java stack. A Java stack consists of stack frames in which methods run. The stack frame consists of an OS, Frame Data and the LVA. When a method is called, a new frame is created. The frame is destroyed when the method completes execution.

The bytecodes are found in the code attribute of the method structure. The bytecodes contain instructions that manipulate data in the run time data areas. Data is stored in the run time data areas in the form of 32-bit words. An implementation should be able to interpret these instructions and execute them in accordance with the platform to which it is intended. In chapter 3, limitations imposed by the 8051 on running Java programs are discussed.

Chapter 3

The 8051 and its Limitations

Micro-controllers are found in many day-to-day devices such as calculators, telephones, watches and many more. Their basic function is to allow the device in which they are embedded to intelligently interact with the real world. Typical micro-controller operations involve reading signals or data from the ports, executing appropriate data manipulation instructions depending on the runtime values of these signals, and finally processing the data as output signals (Yeralan & Ahluwalia, 1995:33). The instructions loaded in its Read Only Memory (ROM) are processed and following these instructions, the contents of its Random Access Memory (RAM) is manipulated; giving and getting feedback to and from its outside environment through its peripherals such as the I/O ports.

As mentioned in chapter 1, the 8051 micro-controller, in particular, has been traditionally programmed only in five languages, viz., C, assembly, PL/M, Forth and BASIC (Schultz, 1998:84). The Computer Technology Research Corporation (1998:54), however, indicate that Java was originally intended for embedded systems and that although Java has become popular on the Internet; the possibility exists that Java will appear in a variety of embedded devices. As explained, in chapter 1, various implementations of Java in other environments other than the Internet exist. This study is an attempt to introduce Java as the sixth language that can be used to program the 8051 and its family of micro-controllers.

Since this implementation involves compiling the Java bytecodes explained in chapter 2 to a 8051 compatible executable code, the architecture of the MCS-51™ family and the operation of its various components is investigated. This Chapter will focus on the constraints the 8051 places on the implementation. Firstly the architecture of the 8051 will be discussed to highlight its available resources and lastly the limitations of the 8051 with regard to the JVM requirements will be highlighted.

3.1. The Architecture of the MCS-51™ Family

Almost all sources consulted during this research describe the MCS-51™ family micro-controller in the same way. MacKenzie (1999:17) summarizes the MCS-51™ family micro-controller as an 8-bit micro-controller with features that include,

- 4K bytes ROM
- 128 bytes RAM
- Four 8-bit I/O ports
- Two 16 bit timers
- Serial interface
- 64K external code memory space
- 64K external data memory space
- Boolean processor
- 210 bit-addressable location
- 4 μs multiply/divide

and uses figure 3.1 to illustrate the features mentioned above.

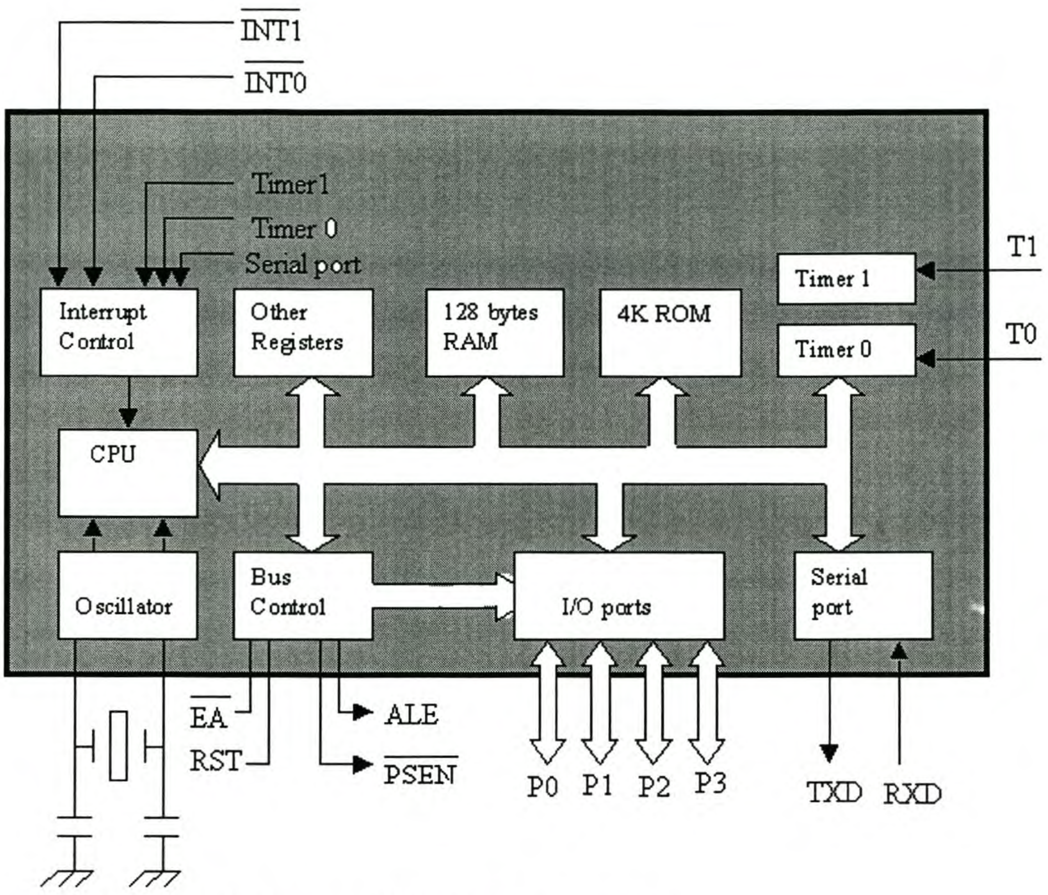


Figure 3.1: 8051 block diagram (MacKenzie, 1999:18)

This family of micro-controllers has both Read Only Memory (ROM) and Random Accessible Memory (RAM). ROM is loaded with actual instructions or a simply a control program that drives the chip. As can be seen from figure 3.1, the available memory for ROM inside the chip itself is only 4 kB. ROM can however be extended up to a maximum of 64 kB using external components (MacKenzie, 1999:22).

The RAM is the memory area for holding data. As depicted in figure 3.1, the 8051 has only 256 bytes of internal data memory. This area is shared with the Special Function Registers. SFR's are used for controlling the function of timers, counters, UART and interrupts.

Information about the MCS-51™ family of micro-controllers is extensively documented. In this section only the SFR's and interrupt requirements will be discussed as they relate to the areas which have to be extensively explored in order to implement Java on the 8051. More details about the 8051 can be obtained from documentation listed in the references.

Special Function Registers

The MCS-51™ family micro-controller has 9 sets of Special Function Registers (SFR) viz.; Program Status Word (PSW), Stack Pointer (SP), Data Pointer (DPTR), B Register, Port Registers, Serial Port Registers, Interrupt Registers and Power Control (PCON) Registers. SFR's are memory locations that control the on-chip peripherals such as ports, timers, and interrupts, as well as other features of the processor (Schultz, 1998:21). This section explores a few of these registers and how they should be approached when implementing Java technology on the 8051.

Stack Pointer (SP)

This is an 8-bit register consisting of the address of data currently on top of the stack. It can be 'pushed', which increments the stack and stores data. It can also be 'popped', which reads data and then decrements the stack. The start of the stack has to be initialised at the beginning of the program. The implementation should take care when handling the Stack Pointer because this is where return addresses are saved when the program branches or during the handling of interrupts. An invalid stack pointer will almost certainly result in program failure.

Port Registers

The 8051 has four Input/Output ports. These ports are available for general use as required by the programmer. Note, however, that some ports may be unavailable due to need for other chip functions, such as external addressing or when other features such as interrupts, serial port, etc, are used. For example, ports such as port 0, 2 and 3 can be used as dual-purpose ports. These ports can operate as I/O lines or as control lines or ports of the address or data bus. The alternate port usage for the AT89S, one of the MCS-51™ family, is illustrated in the table 3.1 in appendix B2.

The basic operations than can be performed on the port registers are writing to and reading from the ports. High level languages such as C have files that specifically define SFR's of the 8051 members (Schultz, 1998:92). As in the C programming language, a Java or JVM implementation on the 8051 will also have to find special ways of operating on the 8051 ports.

Serial Port Registers

The serial port registers provide the 8051 with communication with other serial devices. The serial buffer (SBUF) register is used for reading received data and writing data to be transmitted. Various modes of operation can be chosen by use of the SCON register. SCON is used for selecting the appropriate or required mode of operation of the serial port. These modes are usually chosen during initialisation.

Timer Registers

The 8051 has two 16-bit timers/counters, named timer 0 and timer 1. The timers can be used for interval timing, event counting and baud rate generation. Two SFR registers that are used in the operation of timers are the Time Mode register (TMOD) and the Timer Control register (TCON).

Timers provide real time interfacing with the micro-controller. A program may need to choose timer modes, start and stop a timer and many other various timer operations. As with the I/O and port registers explained thus far, this implementation should provide a way of accessing timer registers and controlling them as the program may demand. In chapter 4 and 5, timer registers will be discussed again in detail.

Interrupt Registers

The 8051 has five sources of interrupt and a two level priority. Two registers are used, viz., the Interrupt Enable (IE) for enabling interrupts and the Interrupt Priority (IP) for handling priorities of the interrupts. An interrupt Service Routine (ISR) takes care of the interrupts that take place.

The implementation has to provide ways of accessing the IE and IP register. In addition to this, the implementation should find ways of setting or clearing specific flags that may be associated with the interrupt and must also be able to specify the beginning of each Interrupt Service Routine. Details of how this implementation will do this is given in Chapter 4 and 5. The limitations that the 8051 imposes on the implementation of the JBC are discussed next in section 3.2.

3.2. Limitations of the 8051

In chapter 2, the run time data areas for the JVM and the file where data is found were discussed. An implementation of the JVM should therefore be able to get data from the class file and allocate it in various memory areas within the device where the Java technology is implemented. Looking at the 8051 and its application, it is apparent that it has many limitations compared to other devices in which Java programs are traditionally run. These limitations are discussed in detail in this section.

3.2.1. Memory

Java programs need run time data areas including the Java stack to manipulate data found in the class file. This means that the device that runs these programs must have enough memory to manipulate and store data. As discussed in section 3.1, the 8051 does not have enough memory. Although it can also be extended to 64 kB, the 8051 only has 128 bytes of RAM. Thus the implementation of Java on the 8051 should find ways around these memory limitations. Chapter 4 discusses how this constraint is handled.

3.2.2. Word Size

As an 8-bit micro-controller, one of the 8051 limitations with regard to running Java is that its instructions are read in bytes. This is because the word size of data and instructions read from the bytecodes is 32 bits. This means that for the 8051 to manipulate data found in the class file, it must use the equivalence of four (32 bits/8 bits) words for each word found in the class file. Since this data has to be stored in the run time data areas, a lot of internal data memory will be needed, essentially out-running the chip's memory. This limitation will also be discussed again in chapter 4 when implementation decisions are made.

3.2.3. I/O ports and External Memory

The 8051 uses registers such as the I/O SFR ports to read and write data. Java does not have methods to access the 8051 ports and thus information to read from or write to ports of the 8051 cannot be read directly from the class file. The implementation should therefore provide means of reading from and writing to 8051 ports in Java. Likewise, means should be provided for accessing external data or code memory of the 8051 using Java.

3.2.5. Interrupts

Finally, it was mentioned in the beginning of this chapter that a micro-controller interfaces with the real world. To do this, it must be able to handle interrupts. An interrupt procedure is written as part of the program that will finally be loaded on the chip. As in the case of I/O ports, Java does not support this. Thus this implementation should find a way around this, so that the program can be able to handle interrupts using the Java language.

Summary

The architecture of the 8051 limits the implementation of the Java technology on the 8051 in various ways. Firstly, the 8051 does not have larger memory for the required run time data areas. Secondly, the data types found in the class file are a minimum 32 bits in size thereby causing the 8051 to require more words for each equivalent data found in the class file. Thirdly, there are no methods for accessing the 8051's I/O ports in Java. Lastly, in order for useful micro-controller programs to run on the 8051, a means of handling 8051 interrupts in Java should be devised. The next chapter looks at the most important decisions that have to be made in order to implement Java technology on the 8051.

Chapter 4

Implementation Decisions

This implementation intends to extend the Java technology to the MCS-51 family of micro-controllers, and in particular, the 8051. In this chapter, implementation decisions that have been made in order to run Java on the 8051 are discussed. The chapter begins by discussing what the JBC will do. A discussion of how the limitations imposed by the 8051 on the implementation will be addressed will also be discussed and, lastly, further decisions that have to be made in order to run Java on the 8051 will be discussed.

4.1. The Java Bytecode Compiler (JBC)

A compiler will be written (in Java) to compile bytecodes found in the class file. Since some compilers generate assembly code (see Aho, Sethi & Ullman, 1986:17 and Appel, 1997:04), the technique used here will be to write a program that will compile the Java bytecodes into an assembly program as shown in figure 4.1

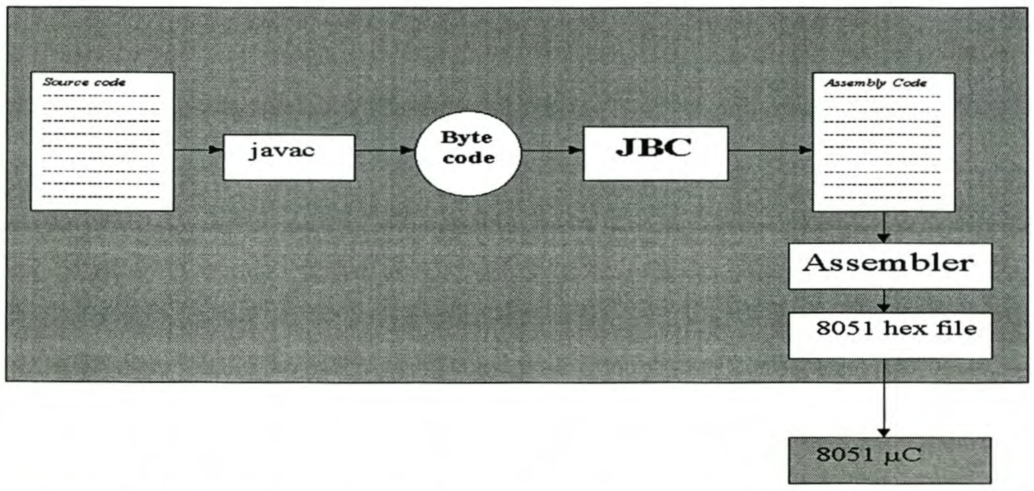


Figure 4.1: Bytecodes compiled to assembly

As shown in the figure 4.1, the assembly code generated will then be recompiled into an executable file that will be loaded on the 8051. The assembly code to be generated is itself a program that must still be compiled to native code. As a result, the code to be appended will also follow a specific algorithm. This algorithm will be in accordance with the supported instruction set (see Appendix C) and also in accordance with the necessary memory designs that will be made on the 8051. The JBC will, however, implement only the subset (shown Appendix C) of the Java language that is necessary for 8051 applications.

4.2. Addressing the 8051 limitations

In chapter 3 specific limitations that the 8051 imposes on the implementation were discussed. These limitations are 8051's word size, the method area and heap requirements, the Java stack requirement, accessing I/O ports and external memory in Java and handling the 8051's interrupts in Java. This section discusses decisions that were taken in order to address these limitations.

4.2.1. Word Size

To solve the problem of the 8051's word size, the JBC will operate on data types bigger than 8 bits in size as if they were bytes. That is, data types such as *int* will be treated as bytes. Thus, the JBC will only support data of values not greater than 255 and will throw an exception if the program contains values not supported. Methods will be supplied for handling values greater than 255. For example, in the case of writing values to registers such as the 16-bit DPTR, a method that will take two parameters, one for the data pointer high byte (DPH) and one for the data pointer low byte (DPL) will be provided.

4.2.2. Method area and Heap

As explained earlier in chapter 2, Java uses the method area and the heap to store data relating to methods and objects, respectively. Implementing the method area and the heap on the 8051's RAM will take up a lot of memory. To address this limitation, this implementation will only support *static* methods. Thus no heap will be necessary. Information regarding the methods will be needed only during the generation of assembly code. This information will be kept in a data structure when reading the class file and can be accessed when needed during the stage of generating assembly code. All methods in the program will be compiled into equivalent assembly code and will be written at the bottom of the code only as subroutines. Thus no method area will be necessary.

4.2.3. Java Stack

Although the Java stack is part of the Heap, it will still be implemented on the 8051 because it deals directly with methods. Since the normal 8051 stack is used to save return addresses for program flow, a second stack will be implemented. Registers R0 and R1 will be used as pointers to the OS and the LVA, respectively. The start of the frame will be above the bit addressable RAM of the 8051. R2 will be used to denote the start of the frame. The value to be loaded in R2 will depend on the size of the Field Area (to be explained in section 4.3.2). If there are no fields, the Java Stack will begin at R2 = 050H. This value will be saved on the 8051's ordinary stack when a new method is invoked and will be popped when the method returns.

Venners (1999: 168) discusses three possible stack designs. This implementation will follow the design shown in figure 4.2. The figure shows the Java Stack design and illustrates a program with method *methodOne(parOne, parTwo)* invoking method *methodTwo(firstPar, secondPar)*. Note that parameters *firstPar* and *secondPar* on the OS of *methodOne()* become the local variables of *methodTwo()*. That is, the OS of *methodOne()* becomes the LVA of *methodTwo()*.

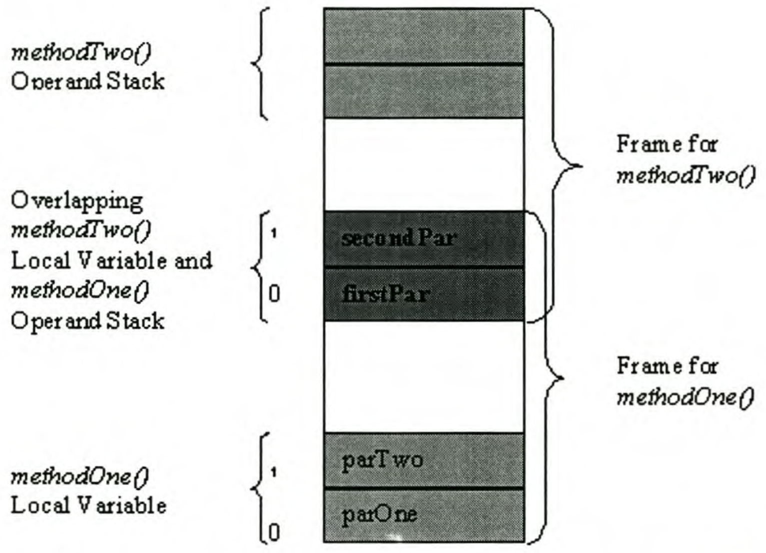


Figure 4.2: Frame overlap stack design (Venners, 1999:171)

This design is preferable over the other two that Venners discusses because, firstly, memory used by the invoking method to store the new method's parameters is the same memory used by the invoked method to access its parameters. Secondly time for copying parameter values from one frame to another is saved. In addition to this way of saving memory, an implementation requires that the frame should be destroyed, thereby claiming memory, when the method completes execution. Note that the Frame Data of the Java Stack will not be implemented on the 8051 Java Stack since this information will be read directly from the class file and stored in the data structure that will be accessible during the generation of assembly code.

4.2.4. I/O Ports and External Memory

The JBC software package will include a set of *static* and *final* methods that will be made known for purposes of reading from and writing to the I/O ports. This will be a class with empty methods. The methods are made *final* so that they cannot be overridden and *static* because this implementation will not support objects. If the JBC finds the name of the method in the class file's *method_info* (section 2.2.3), a subroutine that takes the parameters of the method will be written at the bottom of the code. The same technique will be used for accessing External Memory. These methods will be discussed in chapter 5 and documented in the API.

4.2.5. Interrupts

Default interrupt vectors for all interrupts will be written at the top of the code during initialisation. The address to jump to will be an interrupt service routine (ISR) that corresponds to a specific interrupt. If no interrupt method is written, the ISR at that address will be a default one that simply returns the interrupt.

Special interrupt method names will be reserved for interrupts and will be known to the JBC. When these names are used to name methods, the JBC will replace the default ISRs with the ISRs of the assembly code generated from the bytecodes found in the methods. These ISRs will be written at the bottom of the code.

4.3. Further Implementation Decisions

In addition to the decisions discussed in sections 4.1 and 4.2 above, further decisions have been made with regard to what the JBC will do for each byte found in the *info_structure* of the class file. This section discusses the remaining decisions that were made for this implementation.

4.3.1. Constant pool

Information read from the Constant pool will be stored in a data structure (see also Aho *et al*, 1986:11). This data structure will be accessible during the analysis of the class file structures and the generation of assembly code.

4.3.2. Fields

A separate memory space for fields will be reserved just below the Java stack that will be implemented on the 8051. This memory will be allocated dynamically as need arises. That is, it will be allocated only if the class file being analysed includes class initialisers. Register R3 will be used to denote the beginning of the FieldArea and R1 will be used as a pointer to the FieldArea. The FieldArea will begin at address 050H at all times. The Java Stack will then begin at an address found in R3 plus the number of fields (which can be found from the field count of the *field_info structure* of the class file). Thus, if there are no fields to be included, then there will be no FieldArea and the Java Stack will begin at address 050H. This is illustrated in figure 4.3. The figure shows the Java Stack of figure 4.2. redesigned to include the FieldArea. The corresponding addresses are shown on the left of the figure.

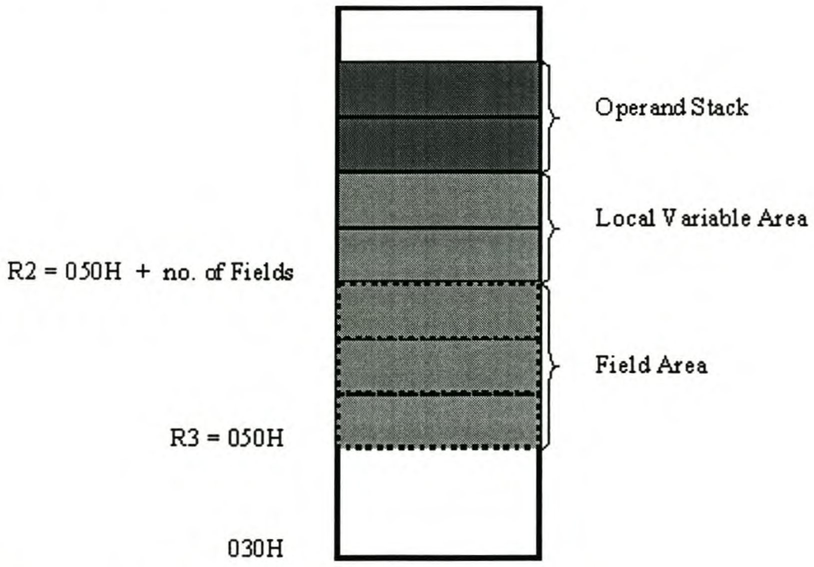


Figure 4.3: Java stack including the FieldArea

4.3.3. Methods

The JBC will read the *method_info* structure of the class file to find information that includes the code attribute information (*code_attribute)*. It will analyse the *code_attribute* structure to get information about the size of the local variables (*max_locals)* and the actual bytecodes (*code[code_length])*. The rest of the attributes (*method_info* attributes) and attributes within the code attribute (*code_attribute* attributes) will be ignored as they only give debugging information. For each bytecode found in the *code[code_length]*, the JBC will follow the instruction set found in the JVM Specification to generate the equivalent 8051 Assembly language. If an unsupported bytecode is encountered, an exception will be thrown. The concept of generating assembly code from bytecode is illustrated in figure 4.4. Note that the ultimate assembly file generated by the JBC will be recompiled into 8051 binary file using *asm51*. This is the actual file that will be loaded onto memory.

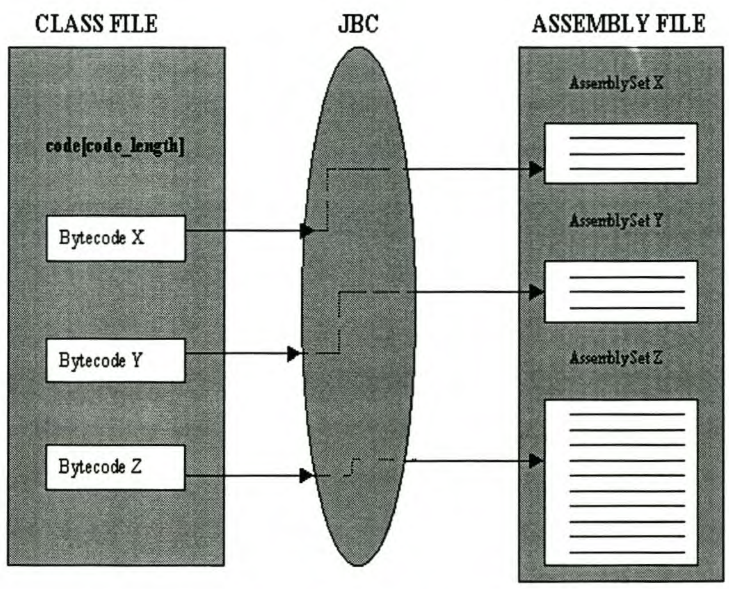


Figure 4.4: **Bytecodes generated into assembly code to make an assembly file**

As can be seen from the figure 4.4 above, the assembly file is made out of a collection of each piece of assembly code generated due to interpretation of the JVM's specification for each bytecode found in the class file. This JVM specification referred to here is the instruction set.

4.3.4. Instruction Set

Due to limitations of the 8051, the instruction set found in the JVM will not be fully implemented. Instructions which involve operation on *doubles*, *floats*, *longs*, arrays, references and objects will not be supported by this compiler. A list of supported instructions is provided in Appendix C. Also, as explained in chapter 1, no known Embedded Java API for the 8051 microcontroller is known to date, so none could be used in this implementation. For supported instructions, the JBC implements the algorithms specified for the JVM.

4.4. Garbage Collection

Due to the decisions taken in section 4.2, this implementation will not support objects. As a result, this implementation will not do any garbage collection as there will be no objects that will be instantiated during run time.

4.5. The JBC Software

The JBC will come as a software package that will include classes to analyse the classfile, classes for *static* methods to access the 8051 I/O ports, serial ports, external memory and to create interrupt procedures. There will also be a class dedicated specifically to reading the *code[code_length]* of the *code_attribute* of the *method_info* structure. A full 8051 Java API will be provided with the package.

For purpose of research, the program will, in addition to analysing the class file and generate assembly code; allow analysis of a constant pool and the generated bytecode. The program will also give an option of compiling the generated assembly file. The program will be referred to as the Java Bytecode Compiler Tool as it does not only compile the class file to a hex file. As just explained, the tool includes a set of other functions which have been included for academic purposes for other researchers who might like to explore this concept further.

Summary

The JBC will compile the bytecodes into equivalent 8051 assembly code. The code will then recompiled into the hex file using the asm51 assembler. Only a subset of the Java language will, however, be supported. All data types will be treated as if they were bytes. Methods will be provided to write values greater than 255 to registers such as the 16-bit DPTR. The method area and the heap will not be implemented as only *static* methods will be supported and thus, no garbage collection will be done. Method area information needed during the generation of assembly code will be stored in the data structure. The Java stack will, however, be implemented at the top of the area allocated dynamically for fields.

Special *static final* empty methods will be written for accessing the SFRs and external memory. When these methods are called, subroutines of their assembly code will be written at the bottom of the code. Special method names will also be reserved for naming interrupt methods. Default ISRs that simply return from the interrupt will be written at the bottom of the code. These ISRs will be replaced with ISRs of the assembly code compiled from the bytecodes found in the methods if the reserved special method names are used.

Chapter 5

Implementation

The JBC is a Java program that reads the class file and generates equivalent 8051 assembly code. This code is then re-compiled into a hex file using the *asm* compiler. The program comprises classes, each of which performs a specific function. The program allows the user to choose between various options. These options include analysing the class file, the constant pool, the opcode, generating the assembly code and compiling the generated assembly code. This chapter will focus on part of the program that deals with the generation of the assembly code and compiling it. It will also show how the program was designed in order to address the limitations raised in chapter 3 and the implementation decisions made in chapter 4.

5.1. Reading the classfile

In order to generate the equivalent 8051 assembly from the class file, the class file is read one byte at a time following the class file format. The assembly code generated follows the algorithm defined by the supported JVM instruction set found in the JVM specification. The supported instruction set is shown in Appendix C. In addition to the instruction set algorithm, the assembly code is generated following the 8051's Java Stack and Field Area memory design done in chapter 4. A string buffer is used to append the equivalent assembly file that is generated as the file is being read. When reading the file is complete, the buffer is written into a file thereby creating a complete assembly file. An example below will be used to show the assembly code that would be generated throughout the stages of the JBC compilation. The listing of the possible JBC assembly code output for this program is shown in Appendix E2.

```
class Example
{
        static int v = 5;
        static int w = 10;

        public static void main (String[] args)
        {
                int x = 15;
                int y = 20;
                int z = 0;
                int port1;

                z = x|y;
                addd(w,x);
                addd(y,z);

                if (y < x)
                {
                        x = y;
                }
                port1 = IO_8051.readPort_1();
                IO_8051.writePort_3(port1);
        }

        public static int addd(int first, int second)
        {
                return (first + second);
        }
}
```

Figure 5.1 illustrates the overall flowchart for reading the class file. The class file is read by reading each of the class file structures described in chapter 2. As the file is being read necessary assembly code to make an assembly file is appended to the buffer. The magic, version, access flags, this class, super class and interfaces structures of the class file are only read and ignored since this implementation does not deal with the implementation of these structures (also see figure 5.1).

As can be seen from figure 5.1, when reading the class file is complete, the program checks if there are any methods that have to be called from any other class file. If so, the program opens and reads the class file concerned but appends to the buffer only those methods that have been called. Otherwise the program appends special SFR methods, if any. It then invokes the method for appending any interrupt procedures that may be in the program. The "END" directive is then appended to show the end of the assembly program. The assembly code that was appended in the buffer throughout reading the class file is then written to a file, essentially generating an assembly program ready for compilation by the native assembler.

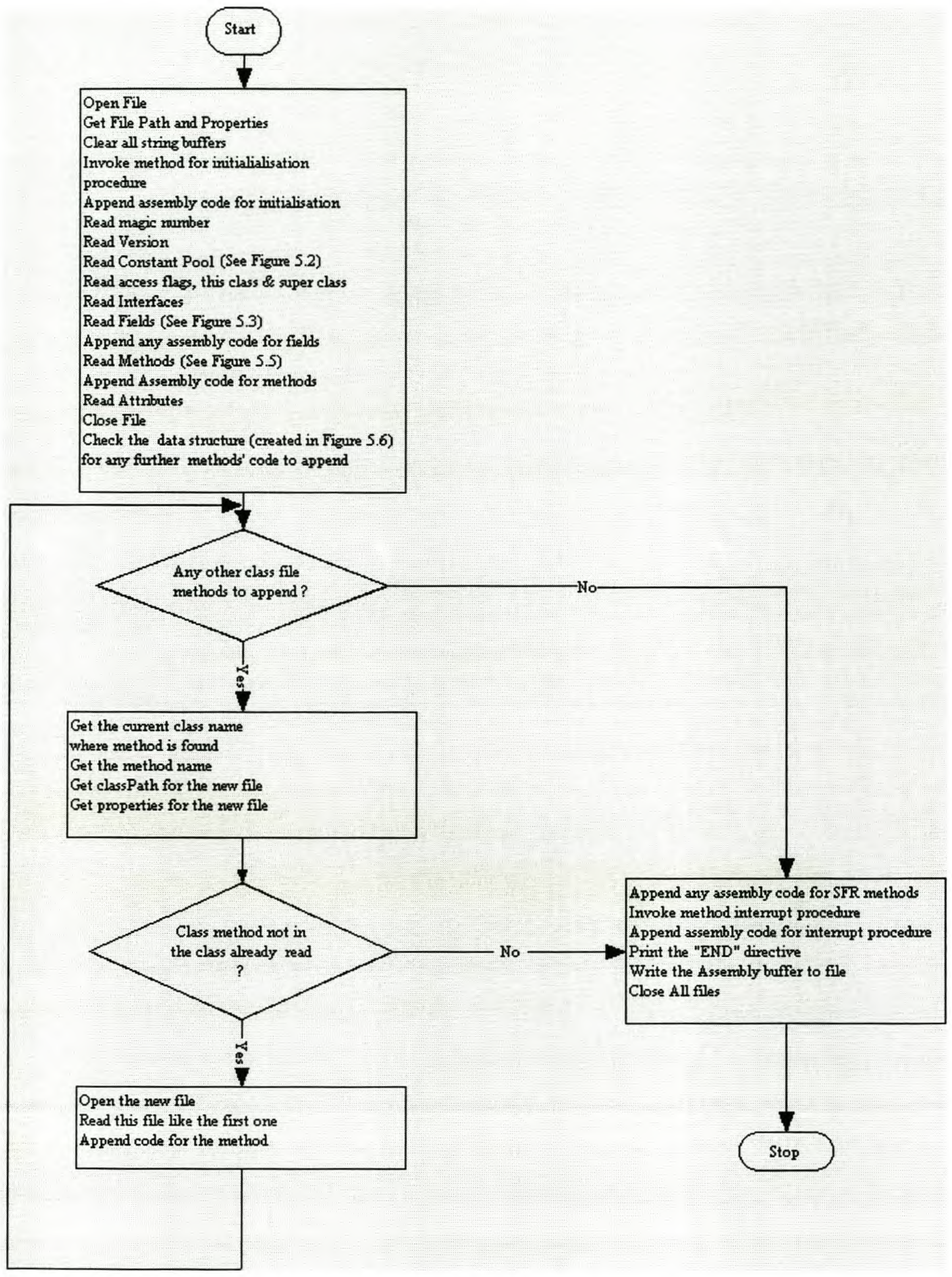

Start
Open File
Get File Path and Properties
Clear all string buffers
Invoke method for initialialisation procedure
Append assembly code for initialisation
Read magic number
Read Version
Read Constant Pool (See Figure 5.2)
Read access flags, this class & super class
Read Interfaces
Read Fields (See Figure 5.3)
Append any assembly code for fields
Read Methods (See Figure 5.5)
Append Assembly code for methods
Read Attributes
Close File
Check the data structure (created in Figure 5.6) for any further methods' code to append
Any other class file methods to append ?
No
Yes
Get the current class name where method is found
Get the method name
Get classPath for the new file
Get properties for the new file
Class method not in the class already read ?
No
Yes
Open the new file
Read this file like the first one
Append code for the method
Append any assembly code for SFR methods
Invoke method interrupt procedure
Append assembly code for interrupt procedure
Print the "END" directive
Write the Assembly buffer to file
Close All files
Stop

Figure 5.1: Flowchart for reading the class file

5.2. Preparing the run time data areas

In order to comply with the design of the FieldArea and the Java stack designed in chapter 4, the JBC must create the runtime data areas information for a particular program. This includes initialising the start of the FieldArea, the Java Stack and any fields that may be in the program. The JBC does this by reading the class file structures and generating assembly code that will accomplish this. This section focuses on the preparation of the run time data areas and its associated data.

5.2.1. Initializing Registers and Data Segments

The flowchart in figure 5.1 shows that the initialising procedure method is called. This method simply appends to the buffer assembly code that defines the usage of registers, the data segment, the code segment and interrupt vectors. Considering the example in section 5.1, after execution of this method, the code in the buffer will look as shown in lines 1 through 46 taken from the listing shown in Appendix E2.

```
1     $MOD51
2     ;------------------------RegisterUsage-----------------------------------
3     ;
4     ; R0 = OperandStack
5     ; R1 = LVAPointer
6     ; R2 = StartOfFrame
7     ; R3 = StartOfFieldArea
8     ; R4 = CalculationValuesStrorage (esp. isub)
9     ;
10    ;------------------------DefineStorage-----------------------------------
11    ;
12    dseg AT 50H
13    FMaxLocals: DS 1       ;To store maximum number of local variable
14    I0MaxLocals: DS 1      ;To store maximum number of Interrupt local variable
15    FMaxStack: DS 1        ;To store maximum size of stack
16    Parameters: DS 1       ;To store number of parameters
17    CompVal: DS 1          ;To store values for comparison purposes
18    PortVal: DS 1          ;To store the value to be written to port
19    PrtVal: DS 1           ;To store the value for bit to be written to port
20    ;
21    ; -----------------------CodeSegment--------------------------------------
22    ;
23    cseg
24    ORG 000H
25    SJMP Start
26    ;
27    ; -----------------------InterruptVectors---------------------------------
28    ;
29    ORG 0003H
30    LJMP intrExternal0

31    ORG 000BH
```

```
32      LJMP intrTimer0

33      ORG 0013H
34      LJMP intrExternal1

35      ORG 001BH
36      LJMP intrTimer1

37      ORG 0023H
38      LJMP intrSerialPort

39      ORG 002BH
40      LJMP intrTimer2

41      ; -----------------------MainProgram-------------------------------------
42      ;
43      Start:
44      mov SP, #30H           ;Initialising the stack
45      mov R3, #60H           ;Initialising the start of fields area
46      mov A, R3
```

Lines 12 to 19 show the data segment initialised at 40H and defining various other variables that may be needed at runtime. Interrupt vectors are defined in lines 23 through 40. All interrupts are defined because it is not known in advance as to which interrupts the program may need. Lines 43 to 46 initialise the stack pointer to 30H and the start of the FieldArea to 50H as designed in chapter 4. Note that the code in lines 1 through 46 is constant and stays the same irrespective of the file being read.

5.2.2. Constant pool Information

When the JBC gets to the constant pool table, it creates an array of data structures comprising information found in the constant pool. This data structure is needed in order to store any information that may be needed during the generation of the assembly code. Each of the constant type (see chapter 2) is stored in this structure and can be accessed anytime throughout the rest of the program. This information represents all the information that would be stored onto the Method area and the Heap. Information about the constant pool is not appended to the assembly code buffer as it is only needed as a reference data structure. Flowchart 2 in figure 5.2 illustrates how the Constant pool is read.

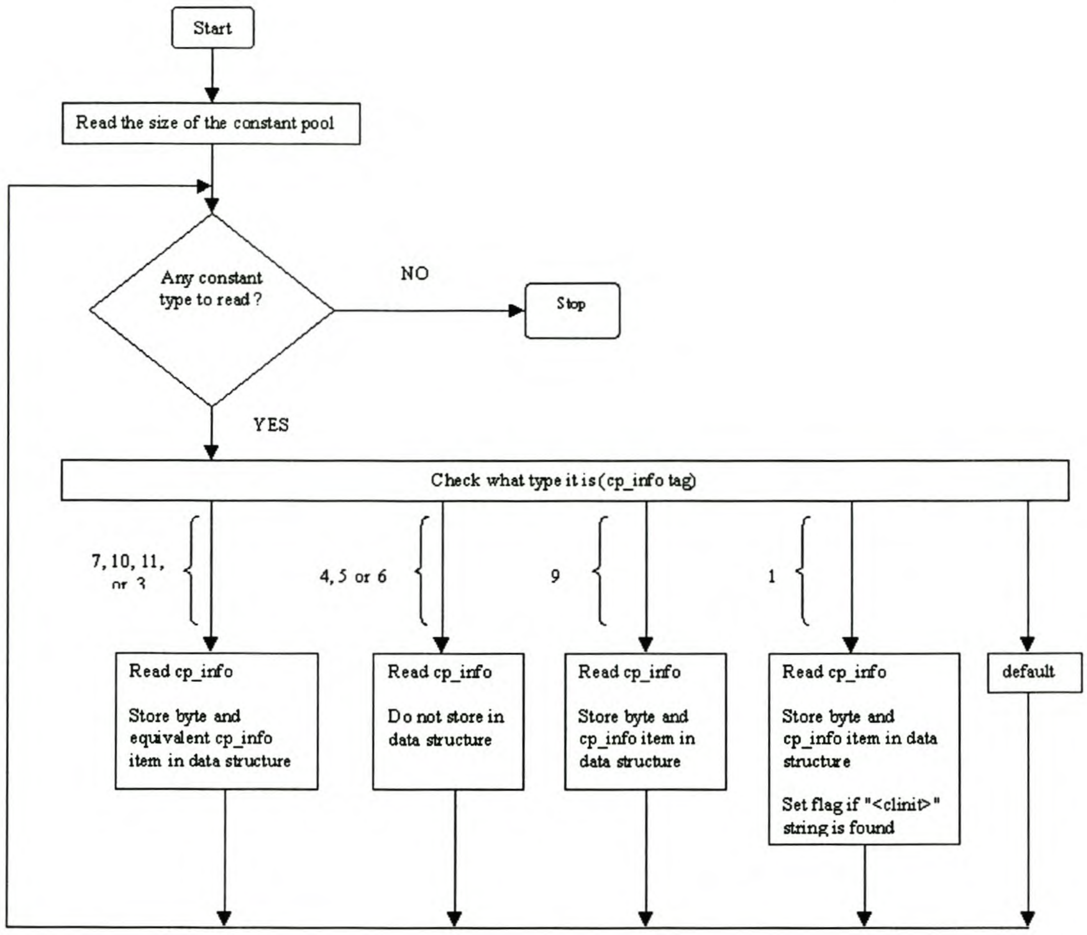

Start
Read the size of the constant pool
Any constant type to read ?
NO
Stop
YES
Check what type it is (cp_info tag)
7, 10, 11, or 3
4, 5 or 6
9
1
Read cp_info
Store byte and equivalent cp_info item in data structure
Read cp_info
Do not store in data structure
Read cp_info
Store byte and cp_info item in data structure
Read cp_info
Store byte and cp_info item in data structure
Set flag if "<clinit>" string is found
default

Figure 5.2: Flowchart for reading the constant pool

Also done when reading the constant pool is checking the strings generated (case 1) to find if a *javac* generated method "<clinit>" was found. This method shows that there are fields in the class and will be needed when reading the fields. The method is responsible for initialising the fields.

5.2.3. Initialising Fields and the Java Stack

To read the field, the program follows the *field_info* structure (figure 5.4). It starts by finding the size of the FieldArea (if any) from the *field_count* of the *field_info* structure. If there are any fields to read, the program creates an array of field names and continues to read the *access_flags* and the *name_index* for the field. The field array is created so that the FieldArea can be created because when the bytecode is generated, reference to the fields is made from the constant pool.

The problem with referencing directly from the constant pool is that constant pool entries are string representations that would take a lot of the memory if they were to be stored in the 8051's RAM. Hence, the JBC has to map the fields in the Constant pool data structure created in section 5.2.2 with the FieldArea that can be represented on the 8051's memory. This concept is illustrated in figure 5.3. below:

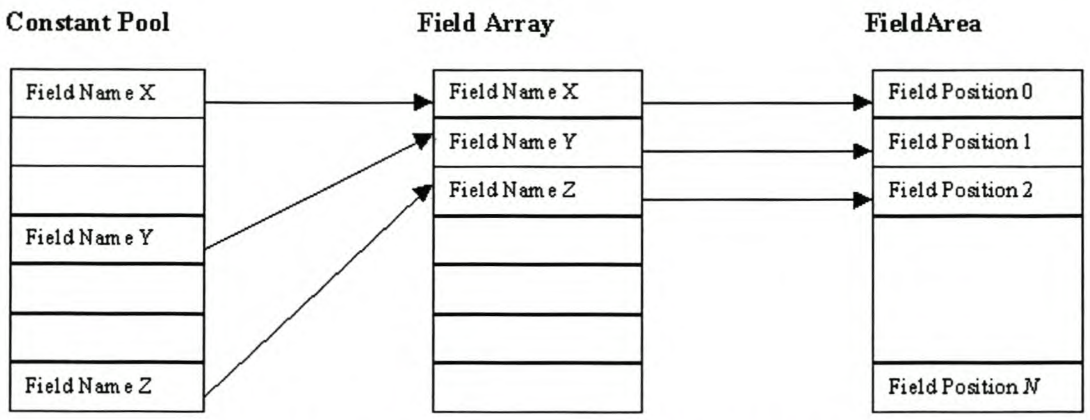


Figure 5.3: Mapping of fields in constant pool with the FieldArea in 8051's memory

The FieldArea was initialised to 50H using register R2 when the initialisation procedure method was called (section 5.2.1). As shown in figure 5.3, the field name is found from the data structure made when reading the constant pool. This name is stored in the field array just created. The indices to the field array are made to correspond to indices to the FieldArea so that when a field is referenced from the constant pool during the generation of assembly, the field array can be scanned to give the position in the Field Area.

To initilise the fields, assembly code for initialising each field to zero (with reference to its position in the field array) is appended to the buffer. The appended code gets the value of the start of the FieldArea (R3) plus the position of the field and write it to the pointer (R1) so that it now points at the position of the field. Zero is then written to that position.

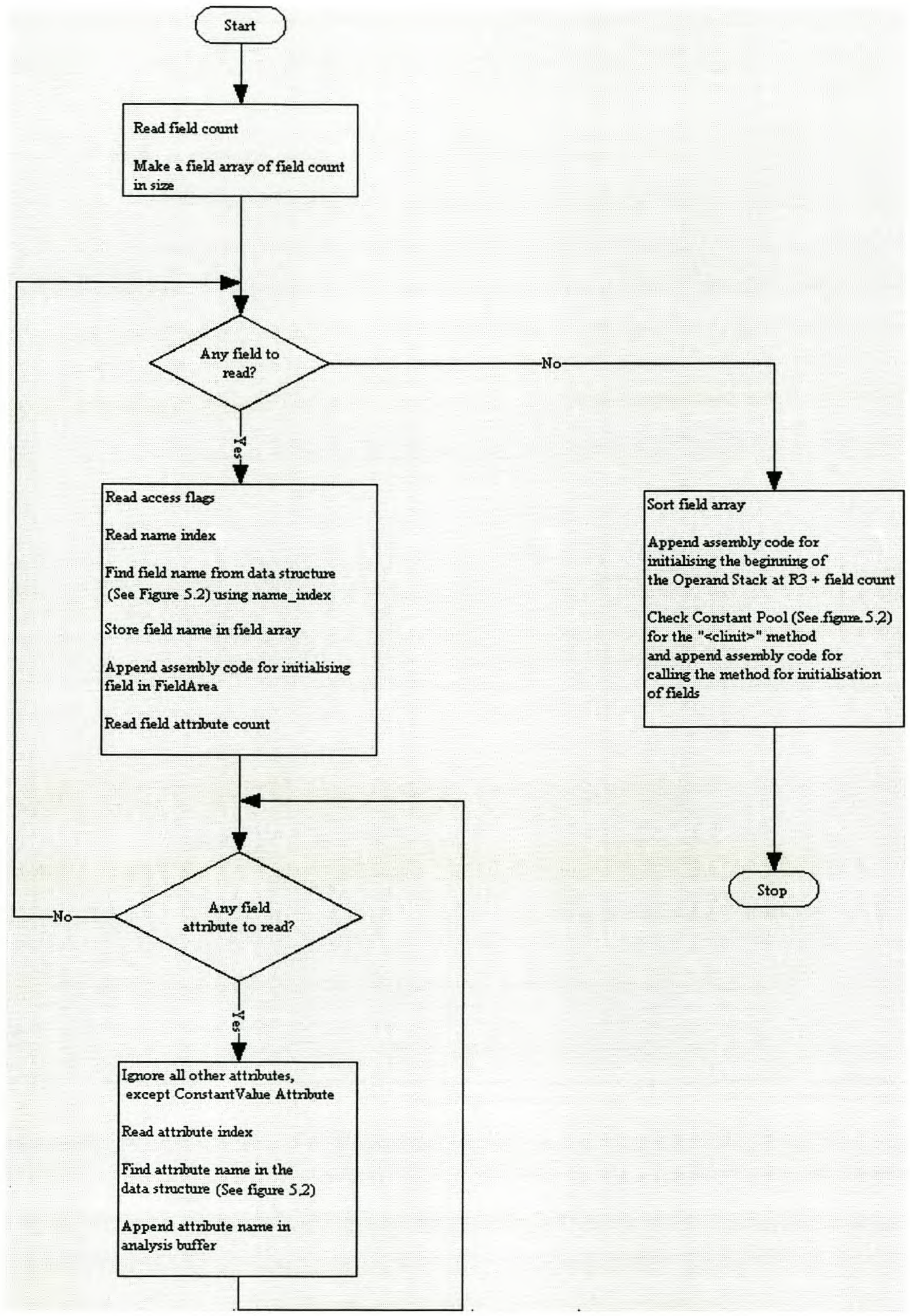

Start
Read field count
Make a field array of field count in size
Any field to read?
No
Yes
Read access flags
Read name index
Find field name from data structure (See Figure 5.2) using name_index
Store field name in field array
Append assembly code for initialising field in FieldArea
Read field attribute count
Any field attribute to read?
No
Yes
Ignore all other attributes, except ConstantValue Attribute
Read attribute index
Find attribute name in the data structure (See figure 5.2)
Append attribute name in analysis buffer
Sort field array
Append assembly code for initialising the beginning of the Operand Stack at R3 + field count
Check Constant Pool (See figure 5.2) for the "<clinit>" method and append assembly code for calling the method for initialisation of fields
Stop

Figure 5.4: Flowchart for reading the Fields structure

Looking at the example in section 5.1, fields *v* and *w* would be initialised as shown in lines 47 through 60 in the listing in appendix E2.

Field attributes are then read. All attributes, except the constant value attribute that defines the attribute length and constant value index, are ignored. To initialise the Java Stack, assembly code for initialising the start of the LVA is appended to the buffer. The LVA starts above the FieldArea at R3 + the number of fields (See section 4.3.2, figure 4.3). Lines 62 to 69 in the Listing in appendix E2 shows the initialisation code for the LVA of the Java Stack of the example in section 5.1. Note that the OS has not yet been initialised. Initialisation of the OS will be discussed when initialisation for methods is discussed because OS initialisation depends on the number of local variables found in the method. The number of local variables is known only when methods are being read.

Once the fields are initialised, the program then checks in the Constant pool (figure 5.2) to establish if there is a need to call the method (<clinit>) for assigning actual field values. If so, assembly code for calling such a method is appended to the buffer. This method is supplied by the java compiler (javac) and will be appended during reading methods. Line 71 in appendix E2 shows the assembly code for calling the field assigning method for the example in section 5.1.

5.2.4. Initialisation for Methods

To read the methods, the program follows the *method_info* structure. A methods buffer is used to append assembly code relating to methods. This buffer is then appended to the buffer containing the rest of assembly accumulated so far (See figure 5.1). Reading the methods is illustrated in the flowchart in figure 5.5.

The program begins by clearing the method buffer of any assembly code it may have been accumulated when reading other classes. If there are any methods to read, the program reads each of the method's access flags and the name index but this information is not appended on the buffer. The program then finds the name of the method from the data structure made when reading the constant pool (figure 5.2). The name of the method together with its class name is appended to the methods buffer to identify the methods and to differentiate them from methods with similar names from other classes. For example the method name for the *main (String[] args)* method in the example in section 5.1 is shown in line 72 of in Appendix E2.

To initialise the OS, the return type (descriptor_index) and the number of attributes (method_attribute_count) are read. If there are any attributes to read, then the program ignores all other attributes except the code attribute. To read the code attribute, the length of the attribute (attribute_length) is read, followed by the size of the LVA (section 4.3.2, figure 4.3) and the size of the stack (max_stack). The LVA information, is appended to the methods buffer. This is stored in a *FmaxLocals* variable created when initialising registers in figure 5.1 (section 5.2.1). The size the LVA is needed when for initialising the OS to a value above the local variables when a method is invoked. Using the example in section 5.1, the associated assembly code (including the method name) for storing the size of the LVA in the data segment is shown in lines 74 to 78 in Appendix E2.

The program then continues and reads the length of the bytecode in the code (code_length). If there is bytecode to read, the program calls a method that generates the assembly code from the bytecode. This method will be explained fully in section 5.3 and figure 5.6 when generating of assembly code from the bytecode is discussed. The generated assembly code returned by the method just mentioned is then appended to the methods buffer. The assembly code for jumping to the "END" assembly language directive (Example in section 5.1, line 320 in the listing Appendix E2) is appended to the

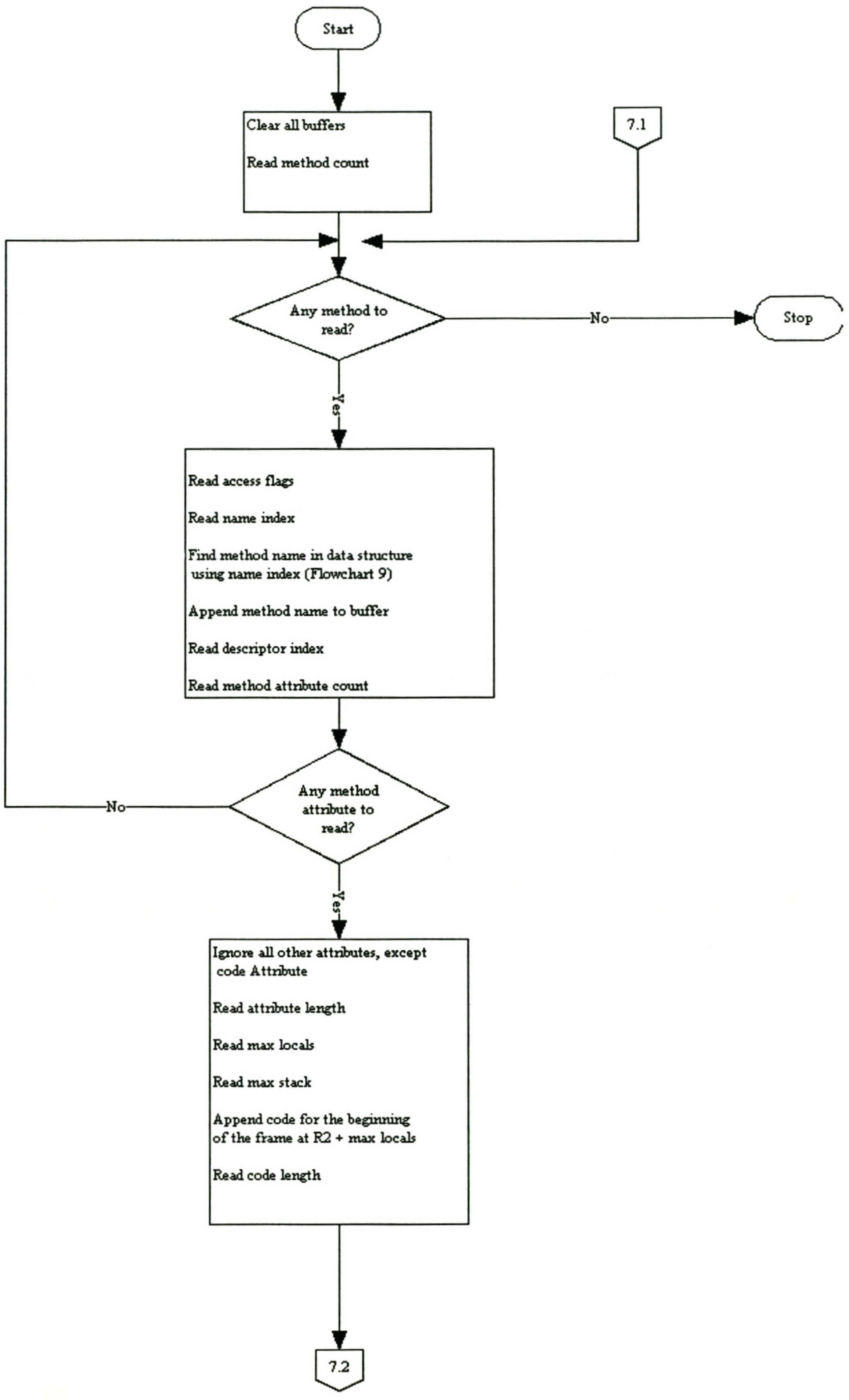

Start
Clear all buffers
Read method count
7.1
Any method to read?
No
Stop
Yes
Read access flags
Read name index
Find method name in data structure using name index (Flowchart 9)
Append method name to buffer
Read descriptor index
Read method attribute count
Any method attribute to read?
No
Yes
Ignore all other attributes, except code Attribute
Read attribute length
Read max locals
Read max stack
Append code for the beginning of the frame at R2 + max locals
Read code length
7.2

Figure 5.5: Flowchart for reading methods

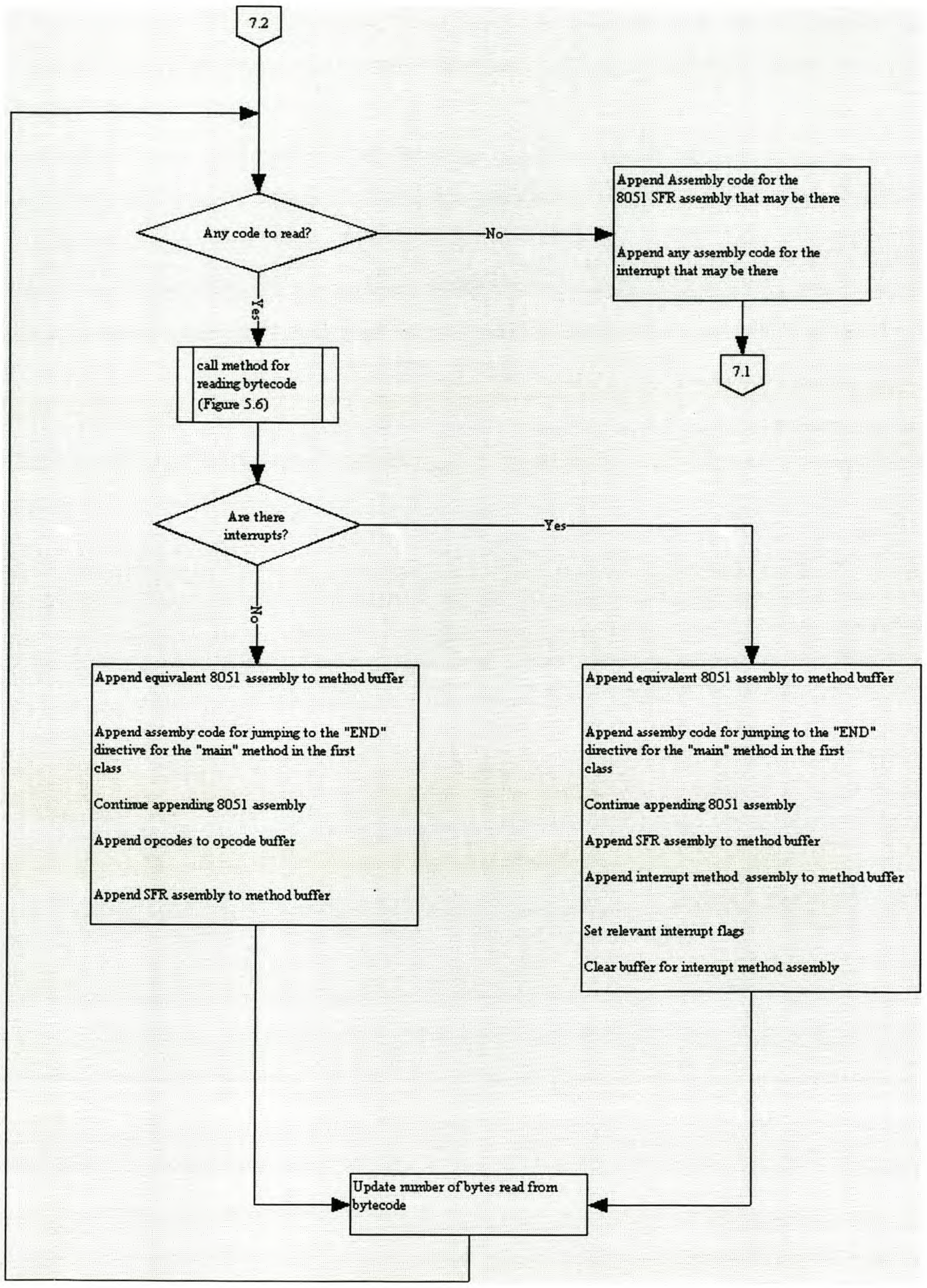

7.2
Any code to read?
No
Yes
Append Assembly code for the 8051 SFR assembly that may be there
Append any assembly code for the interrupt that may be there
7.1
call method for reading bytecode (Figure 5.6)
Are there interrupts?
Yes
No
Append equivalent 8051 assembly to method buffer
Append assemby code for jumping to the "END" directive for the "main" method in the first class
Continue appending 8051 assembly
Append opcodes to opcode buffer
Append SFR assembly to method buffer
Append equivalent 8051 assembly to method buffer
Append assemby code for jumping to the "END" directive for the "main" method in the first class
Continue appending 8051 assembly
Append SFR assembly to method buffer
Append interrupt method assembly to method buffer
Set relevant interrupt flags
Clear buffer for interrupt method assembly
Update number of bytes read from bytecode

Figure 5.5 (cont.)

methods buffer if the method being read is the "main" method of the first class to read. This directive is appended so that the code for subroutines that may be appended next are not executed unless they are called. The *Exit* shown in the listing is used here as a label.

The assembly for SFR and other special methods (section 5.4) is also appended to the methods buffer if its buffer is not empty. If the method is the one of the reserved interrupt method names (see section 5.4), assembly code for interrupts is also appended to the methods buffer in addition to the assembly already in the buffer. The generation of the SFR and interrupt assembly code will be explained in detail in section 5.4.

When appending the assembly code to the methods buffer is done, the total number of bytes read when reading each bytecode and its associated operands is updated. This is done because during the generation of assembly code from the bytecodes (section 5.3), some bytecodes read are accompanied by one or more operands. Thus the program may encounter an *end of file* exception if these operands are not taken into account (see figure 5.5). Any attribute that may be in the code attribute is ignored as it relates to debugging information. If there is no bytecode to read, the program appends the SFR and interrupt assembly code if it exists. If there is no more methods to read, the program moves on to reading the attribute structure for the class. At this stage, all necessary initialisation is now done. The program can now generate assembly file from the bytecodes.

5.3. Generating Assembly Code from Bytecodes

Assembly code is generated from the bytecodes by reading the code array of the method's code attribute. A method, introduced in section 5.2.3, is called. This method is dedicated at looking at the bytecode. The method does this by appending the equivalent 8051 assembly code for each bytecode read as illustrated in figure 4.4 section 4.3.3. the flowchart of the method is described in figure 5.6.

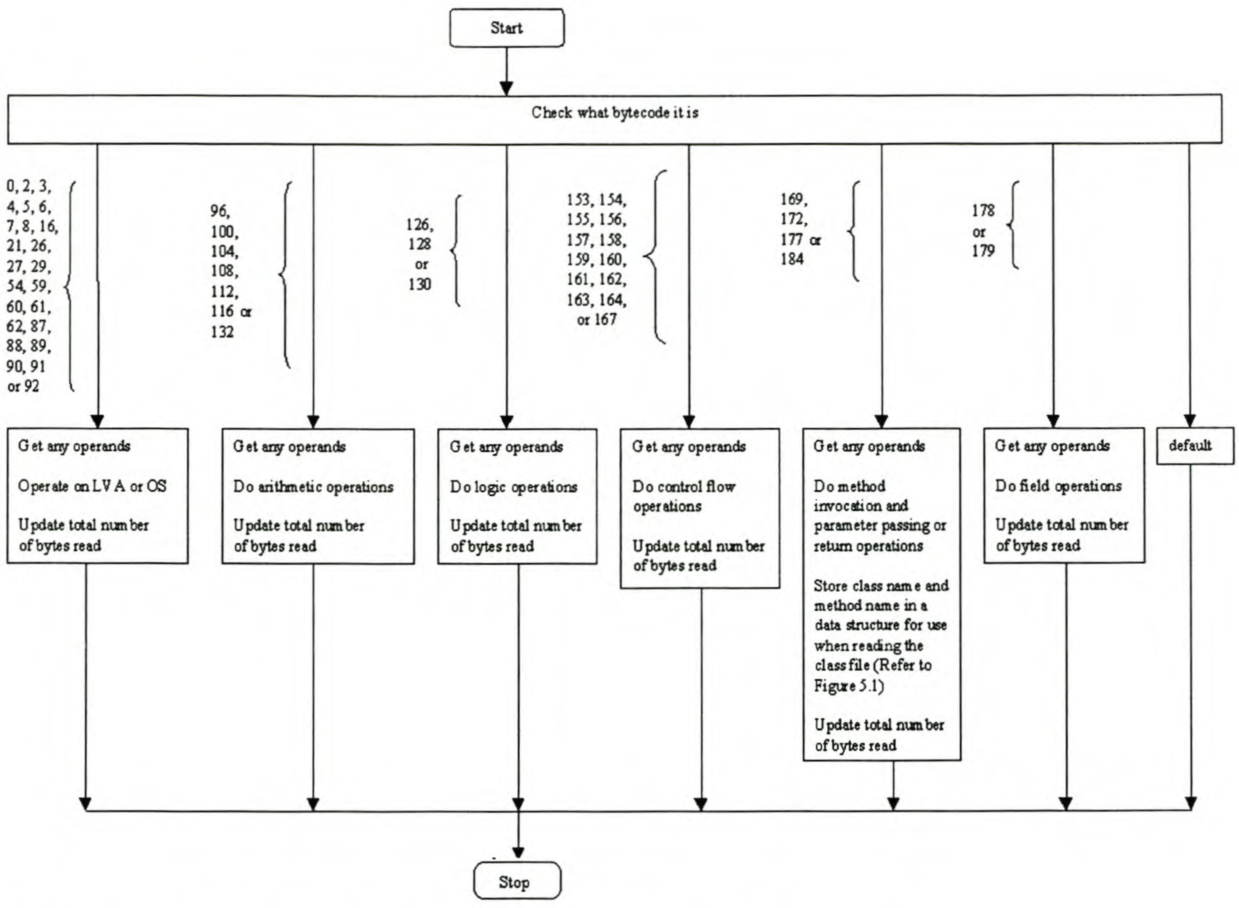

Start
Check what bytecode it is
0, 2, 3, 4, 5, 6, 7, 8, 16, 21, 26, 27, 29, 54, 59, 60, 61, 62, 87, 88, 89, 90, 91 or 92
96, 100, 104, 108, 112, 116 or 132
126, 128 or 130
153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, or 167
169, 172, 177 or 184
178 or 179
Get any operands
Operate on LVA or OS
Update total number of bytes read
Do arithmetic operations
Do logic operations
Do control flow operations
Do method invocation and parameter passing or return operations
Store class name and method name in a data structure for use when reading the class file (Refer to Figure 5.1)
Do field operations
default
Stop

Figure 5.6: Generation of assembly code from reading the bytecodes

The instruction set to be implemented by the JBC can be divided into six types of operation, namely, LVA and the OS operations, arithmetic operations, logic operations, control flow operations, method invocation and return and fields operations. Figure 5.6 describes the flowchart of how the JBC generates assembly code from these instructions. In each case, the JBC treats both the opcodes (mnemonic or bytecode) as being word sizes of a byte as explained in section 4.2.1. This section deals with each of the different operation types. The JVM specification should be consulted to see what the JVM does for each instruction. A full list of generated assembly code corresponding to each bytecode is given in appendix C.

5.3.1. LVA and OS Operations

The size of the LVA is obtained from reading the size of the LVA (max_locals) in section 5.2.3. Instructions that operate on this memory area and implemented by this implementation are *bipush, nop, pop, pop2, dup, dup2, dup_x1, dup_x2, iload, iload_0, iload_1, iload_3, istore, istore_0, istore_1* and *istore_3*. These instructions involve operations such as pushing or popping a value on to the OS and storing or getting a value in the LVA, with an exception of *nop* which does nothing. Pushing the value onto the OS is done by storing the value where R0 is pointing and incrementing it. Likewise, popping is done by decrementing R0 and getting the contents of where R0 is pointing. To store a value in the LVA, the value of R2 (start of frame) is first obtained. The value of the index in the local variable is added to R2 and the result is written to R1 so that R1 now points at an index of R2 + the value added.

Considering the example in section 5.1 in section 5.2 for the *x* variable, lines 79 to 82 in Appendix E2 shows the assembly code that would be appended to the buffer for a *bipush* instruction. Storing this value in the LVA that corresponds to variable *x* involves popping the value from the OS and storing it in the first array of the LVA (*istore_1*). Equivalent assembly for this opcode is shown in lines 83 through 101 in Appendix E2. Note that the value of where to store the value in the LVA is supplied in the class file. In some instructions such as *istore, index* (bytecode 54); this value is the value of *index*. Also note that in each case for the generated assembly, the code begins with a label such as *Opcode8*. This label is necessary to uniquely identify each instruction for purposes of jumping to correct instruction when a control flow instruction is encountered. This will become clear when control flow instructions are discussed later in the section.

In a similar way to *bipush* and *istore* instructions above, the *iload_1* instruction for the same *x* variable in the example in section 5.1 would yield the code listed in lines 129 to 137 in appendix E2. The rest of the bytecodes that deal with the operation on the LVA (see flowchart 5.6) and the OS also behave in a similar manner and will not be discussed individually.

5.3.2. Arithmetic Operations

Arithmetic operations also operate on the OS in that they also push and pop values to and from the OS. They, however, do not store or get them from the LVA. They are also different to stack operations in that they do specific arithmetic operations that cannot be done by the instructions explained in section 5.3.1. As is the case with LVA and OS instructions, the JBC appends the equivalent 8051 assembly code to the buffer for each arithmetic operation it reads. Again a certain algorithm which is in accordance with the Java Stack design is followed. For example, the *addd(y,z)* method in section 5.1, adding *y* and *z* would pop *y* and *z* off the OS, do the addition and push the value on the OS. In accordance with the Java Stack design of this implementation, the assembly code is appended to the buffer is shown in lines 358 through 365 in Appendix E2. Note that the values being popped would have already been pushed on to the OS. A full output assembly code for the *addd(y,z)* method is shown in lines 333 through 365 in Appendix E2. Note also that the *addd()* name is chosen purposely as *add* is a reserved word in assembly language.

In a similar way to the *iadd* instructions, other arithmetic instructions do similar operations to the *iadd* instruction. The rest of the arithmetic instructions implemented by this implementation are *iinc, isub, imul, idiv, irem* and *ineg*.

5.3.3. Logic Operations

Logic operations implemented by this implementation are *iand, ior* and *ixor*. Their operation is similar to that of arithmetic operations just described. They also follow the JVM specification as a guideline and the algorithm for the assembly code to be appended depends on the Java stack designed for this implementation.

As an illustration, consider the code for computing *z* in line 12 of the example in section 5.1. The code *int x = 15* and *int y = 20* will yield code to push firstly the value of *x* onto the OS. The value is then popped and stored in the LVA index 1 corresponding to *x*. Secondly, the value of *y* is pushed on to the OS and then popped and stored in the LVA position 2. The *int z = x|y* instruction causes the two values to be popped from the OS and ored. The result is pushed back onto the stack and the result is stored in the LVA position corresponding to *z*. The assembly code to be appended on to the buffer for this process is shown in lines 129 to 163 in appendix E2.

5.3.4. Control Flow Operations

Three groups of unconditional branching instructions are implemented. These are instructions generated when *if, if-else, while, do-while* or *for* statements are used in a program. The first group comprises *ifeq, ifne, iflt, ifle, ifgt* and *ifge* instructions. These instructions pop a value from the OS and compare it against zero. The algorithm of the assembly code used follows the design of the Java Stack.

The next group of instructions implemented in this implementation comprises of *if_icmpeq, if_icmpne, if_icmplt, if_icmple, if_icmpgt* and *if_icmpge*. These instructions pop two values from the OS and compare them against one another. Again the algorithm followed when appending the assembly code follows the JVM specification and the design of the Java Stack on the 8051.

The last group of the instructions implemented by this implementation comprises of the *goto* instruction. The JBC treats this instruction in the same way as it treats the latter two, except that there are no values to compare. The general algorithm followed for the assembly code to be appended to the buffer when these instructions are encountered is explained in the JVM specification

It was also mentioned in section 5.3.1 that assembly code for each instruction is appended with its corresponding unique label. This is done so that the program can branch to the correct offset calculated from the operands of the branching instructions. To illustrate this further, consider the *if* statement in line 16 of the example in section 5.1. Following the specification, the JBC calculates the offset from the operands and appends the assembly code according to the Java Stack design. The assembly code for the *if_icmpge* instruction read is shown below in line 261 to 272 in Appendix E2. In this code, *opcode39* is the address to jump to. To generate the label, a count of how many bytes have been read with each instruction is done. After each instruction, the total number of bytes read is updated. For example, in the code above, this instruction is the 36th opcode. Two bytes (operands) are then read. The next instruction to go to if comparison does not succeed is therefore *Opcode39*, i.e., 36 + 3. Appending this is done by keeping the "Opcode" string constant while making the number a variable, i.e., *"Opcode" + count + ":"* is appended as a string.

5.3.5. Method Invocation, Parameter Passing and Return Operations

The *javac* compiler provides invocation bytecodes when methods are invoked. Programs written for this implementation will only yield an *invokestatic* (184) bytecode because this implementation only handles *static* methods as explained in section 4.2.2. Methods other than those in the class file being read may also be invoked. When this happens, the JBC opens the class where the method is found and appends equivalent assembly code for that method as a subroutine. This will be explained in detail later in the section. The JBC implements method invocation as specified in the JVM specification and according to the design of the Java Stack.

In addition to method invocation of methods found in class files, the Java program being read may have other special methods such as predefined methods for handling the 8051 SFR's. In this regard, the JBC calls appropriate methods responsible for appending the 8051 SFR assembly code. Methods such as these will be explained in detail in section 5.4. This section discusses separately method invocation, parameter passing and return operations. The flowchart in figure 5.7 illustrates the operation of the JBC when the invokestatic bytecode is encountered.

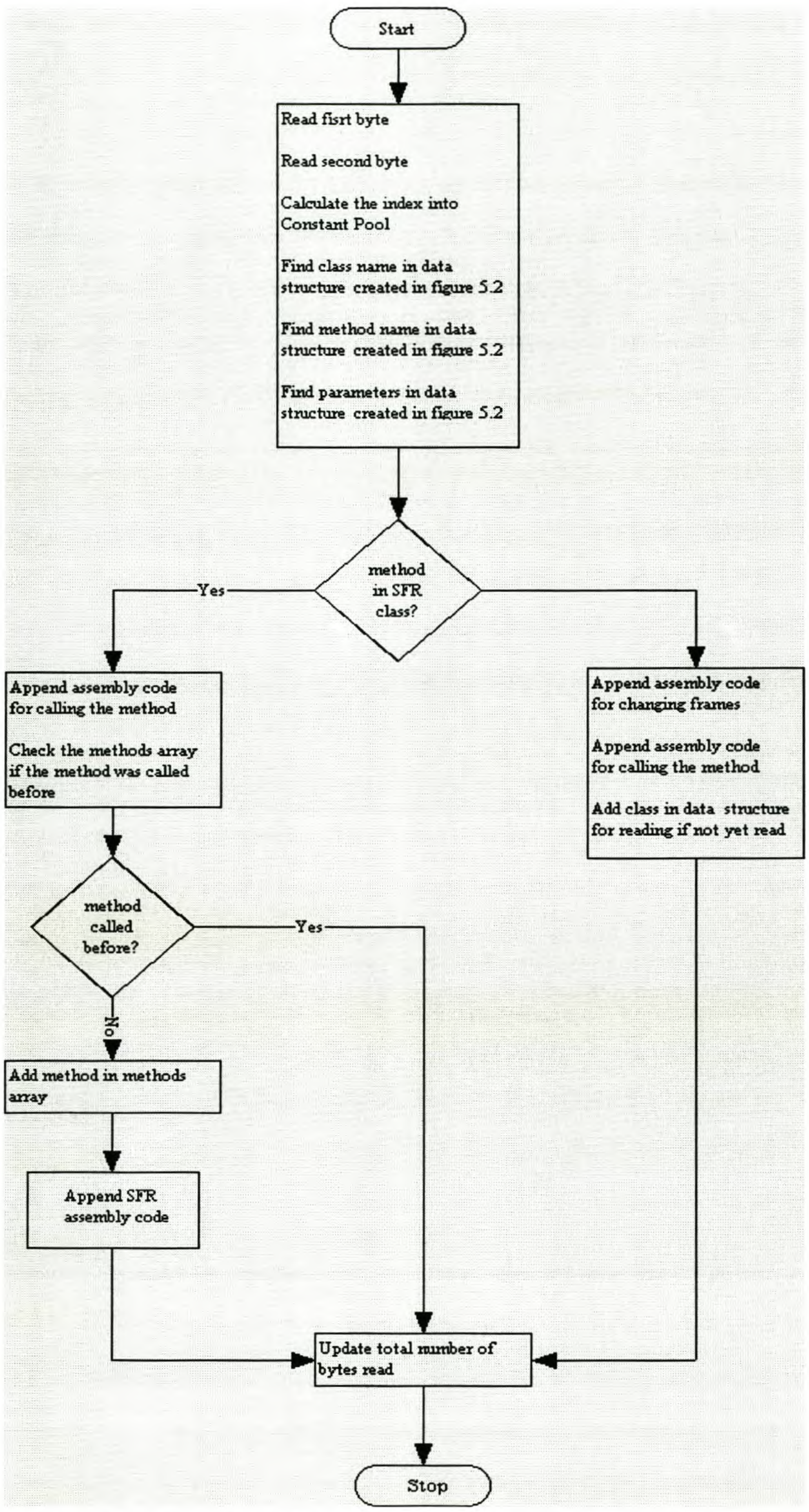

Start
Read fisrt byte
Read second byte
Calculate the index into Constant Pool
Find class name in data structure created in figure 5.2
Find method name in data structure created in figure 5.2
Find parameters in data structure created in figure 5.2
method in SFR class?
Yes
Append assembly code for calling the method
Check the methods array if the method was called before
Append assembly code for changing frames
Append assembly code for calling the method
Add class in data structure for reading if not yet read
method called before?
Yes
No
Add method in methods array
Append SFR assembly code
Update total number of bytes read
Stop

Figure 5.7: Flowchart for the *invokestatic* bytecode

Method Invocation

When the *invokestatic* bytecode is encountered, the program reads the next two bytes. These are the operands of this bytecode. It uses these bytes to calculate the index into the constant pool (figure 5.2). This index is used to find the name of the class, the name of the method and the number of parameters passed with the method. If the method being called is in the class file being read or any other class file except the one for SFR assembly, then the assembly code for changing frames is appended to the buffer (refer to figure 5.7). This assembly code follows the algorithm that is in accordance with the design of the Java Stack in section 4.2.3 and illustrated in figure 4.3 section 4.2.2 and also according to the JVM requirements discussed in the JVM specification for the *invokestatic* instruction and in section 2.1.3.

Going back to the example in section 5.1, the assembly code which will be generated for the method *addd(w,x)* due to the *invokestatic* instruction is shown lines 181 to 198 in Appendix E2. The code begins by saving the value of the current frame on to the normal 8051 stack. The program then gets the number of parameters being passed. The OS is also re-initialised to a value just above the LVA. When this is done, the new method, in this case *addd(w,x)*; is now called. Again note that this method is called with its class name for identification if there is a method with a similar name in another class. When the method has finished execution, the start of the frame for the previous method is obtained again, effectively reclaiming memory by going back to the previous frame. At this stage the JBC updates the number of bytes read and exits.

The algorithm for the assembly code to be appended on the buffer if the method being called is in the class containing SFR subroutines, depends on whether the method is being called for the first time or not. If it is called for the first time the assembly for calling the method, e.g., "*call Classmethod*" is appended to the buffer and the JBC calls the method for appending SFR subroutines. This method will be explained in detail in section 5.3. Also done here, is keeping the names of the methods called so that their code is not appended again if they are called more than once. If the method is not called for the first time, only assembly code for calling the method is appended to the buffer since its subroutine assembly has already been appended to the buffer (refer to figure 5.7). When this is done, the JBC updates the total number of bytes read and exits.

Parameter Passing

The flowchart in figure 5.7 shows that the index into the Constant pool is used to find the number of parameters. The technique used is to find the descriptor index that consists of the return type string. For integers, treated here as bytes, the return type is *I*. A method with two integer parameters will have two *I*'s. Counting the number of *I*'s from the descriptor index will therefore give the number of parameters.

Parameters are needed in this implementation so that frames can be switched during method invocation. Subtraction of the parameters (line 189, Appendix E2) is done in order to switch the current frame's OS and make it the new frame's LVA as explained in section 4.2.3 and illustrated in figure 4.2. The number of parameters is also used to determine the size of the new frame's LVA.

Return Operations

The return instructions implemented by this implementation are *ireturn* and *return*. When these are encountered, the JBC appends assembly code for returning from a method. The *return* instruction, however, does not append anything if the method returned is one of the interrupt methods. This is so because the interrupt method should have its own assembly code return instruction, *reti*. Referring again to the example in section 5.1 for the method *main (String[] args)*, the general assembly code appended (except if the method is the interrupt method) is shown in lines 316 to 320 in Appendix E2. Note that the *ireturn* instruction would also yield the same assembly code because all return types are treated the same in this implementation. When the method returns, it also makes sure that it gets the start of the calling method's frame as can be seen from the code. This makes sure that the value returned is returned to the calling method.

5.3.6. Field Operations

Field instructions refer to instructions that operate on the class variables or fields. Field instructions implemented in this implementation are *getstatic* and *putstatic*. Their operation can involve getting or putting the value in the FieldArea. Fields were discussed in sections 2.3.2 and 4.3.2. Reading fields has been discussed in section 5.2.2 when initialising the fields was discussed. The algorithm for the FieldArea design in section 4.3.2 together with the JVM specification for the instructions is followed when appending the equivalent assembly code.

When these instructions are encountered, the JBC reads the two operands that come with the instruction. It uses these operands to calculate the index into the Constant pool. The index is then used to find the name of the field. Once the name is found, the index into the array of fields corresponding to the field name is found (see figure 5.4). This index will be used to reference to the corresponding index in the FieldArea designed in section 4.3.2 and created in section 5.2.2. The JBC then appends the equivalent assembly code.

Looking at variable *w* for method *addd(w,x)* in the example in section 5.1, the equivalent assembly code to be appended to the buffer is shown in lines 164 to 171 in Appendix E2. The code gets the start of the FieldArea and then adds the position to the index where it will get the value. R3 is used as the start of the FieldArea and R1 as a pointer to the FieldArea. The value obtained is then pushed on to the OS. In a similar way, assembly code for *putstatic* puts a value on to the Field Area.

The operations describe thus far relate to instructions found in the bytecode. To fully handle the operations of the 8051, interrupts and writing and reading to and from I/O ports has to be done. The section 5.4 discusses special methods that handle these operations.

5.4. Special Methods

Special methods are used to handle operations that involve writing to or reading from the I/O ports of the 8051. These methods are recognised by the JBC if they are used in a program. The JBC then provides necessary assembly code associated with these methods. In addition to these methods, the JBC should be able to handle interrupts. This is also achieved by use of special method names known to the JBC. The JBC, however, does not provide special assembly code for these methods. The code for these methods is determined by the bytecode in the method. All the JBC needs is to recognise the method name so that it knows that it refers to the interrupt method. This section discusses how the SFR and external addressing methods and the interrupt methods.

5.4.1. SFR and External Addressing Methods

A special class of 113 empty *static final* methods has been written in order to address assembly code for 8051 SFR's and external addressing discussed in section 3.1.1. These methods are made *final* so that they cannot be overridden and *static* because this implementation only supports *static* methods. Methods that deal with writing a value return *void* and those that deal with reading the value return a 0 integer. Methods may take parameters or not depending on what they are for. If they take parameters, the parameter values will be loaded on to the OS in a similar way the other methods discussed throughout the chapter do. In other words, the class file will come with instructions for loading these values on to the OS. Operations on the OS of these methods are also similar to those of methods already discussed and they also operate on the Java Stack designed in section 4.2.3.

A second class that contains a method dedicated to appending corresponding SFR assembly code has also been written. When this method is called, it appends the relevant 8051 SFR assembly code to the SFR buffer. This buffer may now be appended to the buffer containing the rest of assembly when appropriate. Typically, the program would append the SFR buffer towards the end of the program just before interrupt procedures are appended. The SFR assembly code is appended to the buffer as a subroutine with the class name and method name as the label.

As with other methods in this implementation, the SFR methods are called at *invokestatic*. The JBC then follows the *invokestatic* algorithm (figure 5.7). It then appends the assembly code for calling the method to the buffer. If the method has not been called before, the method for appending SFR assembly code is called. This method simply checks the name of the method being called and appends the corresponding predetermined assembly. If the method deals with writing value(s) to the SFR, the values are obtained from the OS and written to the SFR. If the method deals with reading the SFR, the value read is pushed on to the OS. To illustrate this consider lines 20 and 21 in the example in section 5.1 that deal with reading a value from port 1 and writing the read value to port 3. The section of the output of the JBC for this program is shown in lines 273 to 315 in Appendix E2.

The code shows that the "main" method calls the method for reading port 1 (Opcode39) and stores the value in LVA position 4 (Opcode44) corresponding to variable *port1* in the example in section 5.1. The value is now loaded on to the OS in preparation to writing it to port 3 (Opcode46). The method for writing to port 3 is called and the "main" returns. The called subroutines are appended after this method. The *IO_8051readPort_1* subroutine follows the algorithm for reading ports. It gets the value of *P1*(port 1) and pushes it on the OS and then returns. The *IO_8051readPort_1* then gets this value from the OS and writes it to port 3. It then returns.

Appending the rest of the methods also follows a similar algorithm as the one just explained. External memory is also accessed in a similar way. The method for writing to external memory takes two parameters, one for the memory address to write and one for the value to write. Reading from external memory is also done in a similar way. Details of the SFR and external methods are documented in the API.

5.4.2. Interrupts

Special method names are reserved for handling interrupts. Unlike the method names for SFR's explained in section 5.4.1, these interrupt names are only known to the JBC and are recognised when reading the class file. These methods should be declared *static* and must return *void*. They should also not take any parameters.

The JBC provides default interrupt service routines (ISRs) for all five 8051 interrupts. These ISRs are appended towards the end of the program (see figure 5.1). The assembly code appended for these ISRs simply has an interrupt label and a return instruction for interrupts as shown in lines 352 through 375 in Appendix E2. If the program does not have a specific interrupt procedure; the interrupt will simply return should it occur.

For programs with one or more interrupt procedures, the JBC replaces the relevant default procedure with the procedure in the program. In order to do this, the JBC looks for the special interrupt method name when reading the methods. The equivalent assembly code for the bytecodes belonging to the interrupt method is appended to the buffer that will be appended to replace the default ISR. The JBC also sets a specific flag to show that a certain interrupt method was found. This flag will be used to tell the method that deals with appending buffers of interrupt assembly code that the default ISR should be replaced with the assembly in the interrupts buffer.

5.5. Compiling

In the beginning of this chapter, it was mentioned that the compiler software allows the user to choose an option of compiling. The assembly code generated by the JBC actually starts with the assembler directive, *$MOD51* (see the listing in Appendix E2). When the *compile* option is chosen, the program writes a batch file that contains the *asm* directive and the name of the file with the *.tjm* extension. The extension is just for identification purposes. This batch file is then executed and the output of the *asm* assembler is written to file for analysis in case of errors. The assembler creates an executable hex file.

Summary

The process of reading the class file discussed in this chapter involves reading the class structures described in the class file format. Information found in the class file structures is used to initialise the run time data areas that will be needed when the assembly code is generated from the interpretation of the bytecode. The method area and heap are handled by storing their information in a data structure that was discussed thoroughly. This data structure is accessed every time this information is needed. Using *static* methods also solves the complexity of having to find a way of representing objects that may have to instantiated as the program is running. This also saves considerable amount of memory that would otherwise be needed to keep information for the class and its members.

As the program reads the structures in the class file, it keeps track of items of the structures that are necessary to generate assembly code. The generation of the assembly code follows three algorithms, one for the JVM instruction set implemented by this implementation, one for the design of the Java Stack and one for the architecture of the 8051. The SFR methods, in particular, address the limitations of the 8051 and provide a way of accessing the SFR's. Interrupt procedures are handled in a way that allows a programmer to write them as methods using special names.

Chapter 6

Evaluation

This chapter discusses the results of the JBC implementation. To evaluate the JBC, programs were written and tested on the provided tutor board shown in figure 6.1. In addition to the tutor board, EMILY 51/52 emulator was also used to simulate and analyse the behaviour of the programs.

6.1. Compiling with the JBC

A tutor board that consists of the ATMEL AT89C52*44 micro-controller connected via the Field programmable Gate Array (FPGA) to the 16-character Liquid Crystal Display (LCD) display and the 12-position keypad was used to test the programs described in this section. Figure 6.1 shows a simplified diagram of the tutor board used. The AT89C52*44 micro-controller is compatible with the 8051.

The program simulates the irrigation controller with one valve. The user enters time for setting the clock and for opening the valve from the PC that is connected to the tutor board via the serial port. When the time set matches the clock time, the valve is opened for 3 seconds. P0.3 is connected to the Light Emitting Diode (LED) that simulates the valve. P1.7 is used to clock the control of the LCD and port 2 is used to write data to the LCD. The choice of the program was in order to investigate the run-time data area usage, to investigate the behaviour of the instruction set, special SFR methods and interrupts.

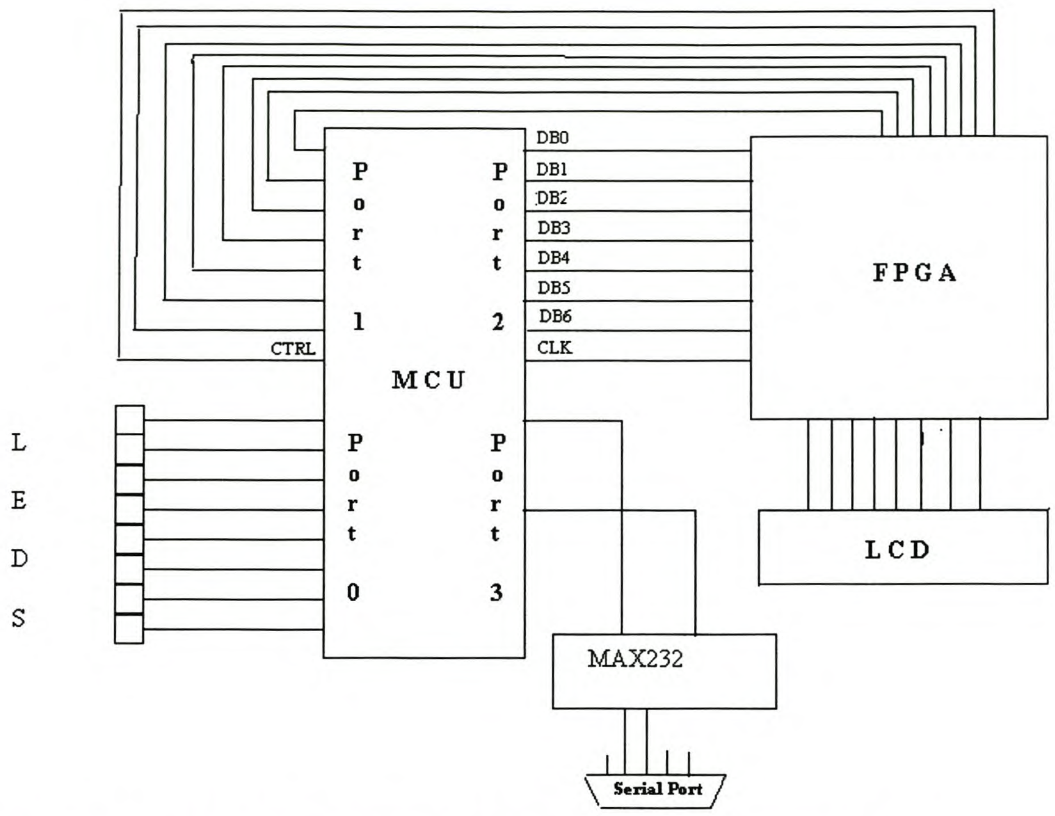


Figure 6.1: A simplified diagram of the tutor board used to test the JBC

The flowchart of the main method of the program is shown in figure 6.2. The source code is shown in Appendix E1. The flowcharts of methods that are called in the main method are shown in Appendix A.

As shown in figure 6.2, the program simply stays in the loop updating and displaying time, and checking for flags that are set when an instruction is received from the serial port. When the instruction for setting the time for opening the valve is received, the methods for displaying the set time and for checking if the time matches with the current time is called. When the instruction for setting the clock is received, the method for setting the clock is called. The new time is then displayed and the program goes back to beginning.

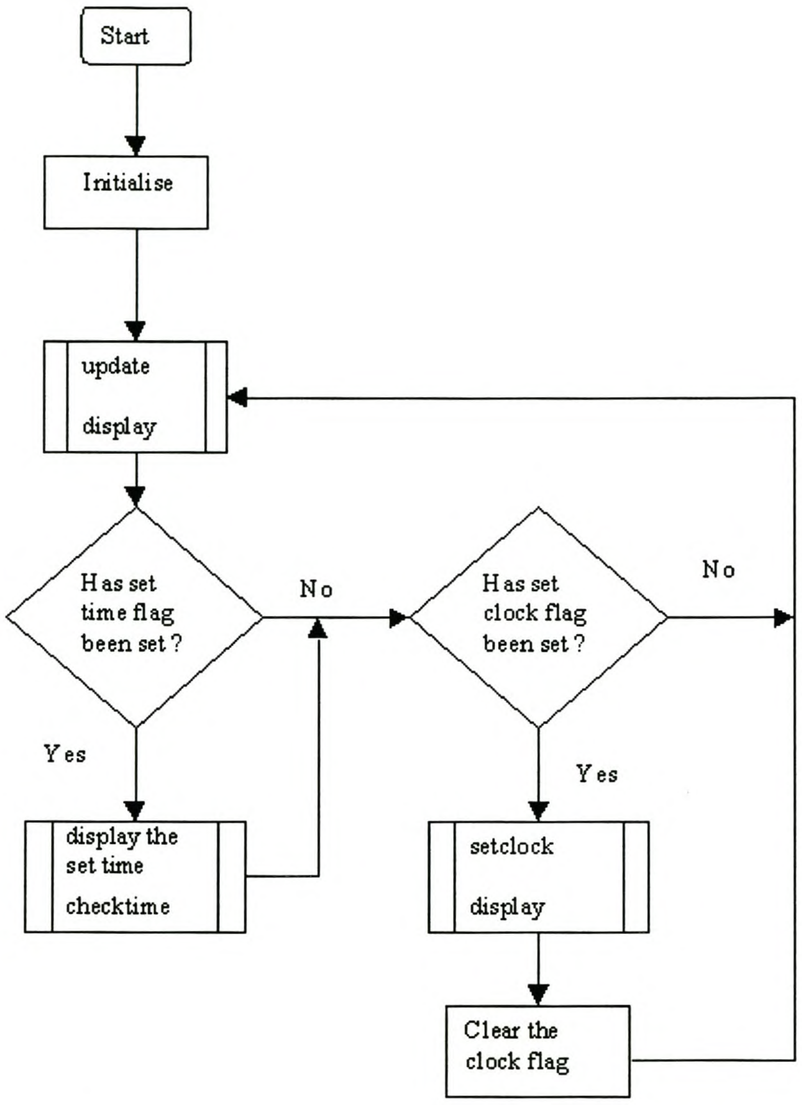


Figure 6.2: An algorithm for the main method of the irrigation controller program

Interrupt procedures for the program are shown in figures 6.3 and 6.4 for the serial port interrupt procedure and timer 0 procedure, respectively. Characters that are used as control instructions for the irrigation controller are received on serial port interrupt. The timer 0 interrupt is used for delay procedures and updating time. Instructions that these procedures cover include those that deal with operation on the SFR registers and control flow of the program.

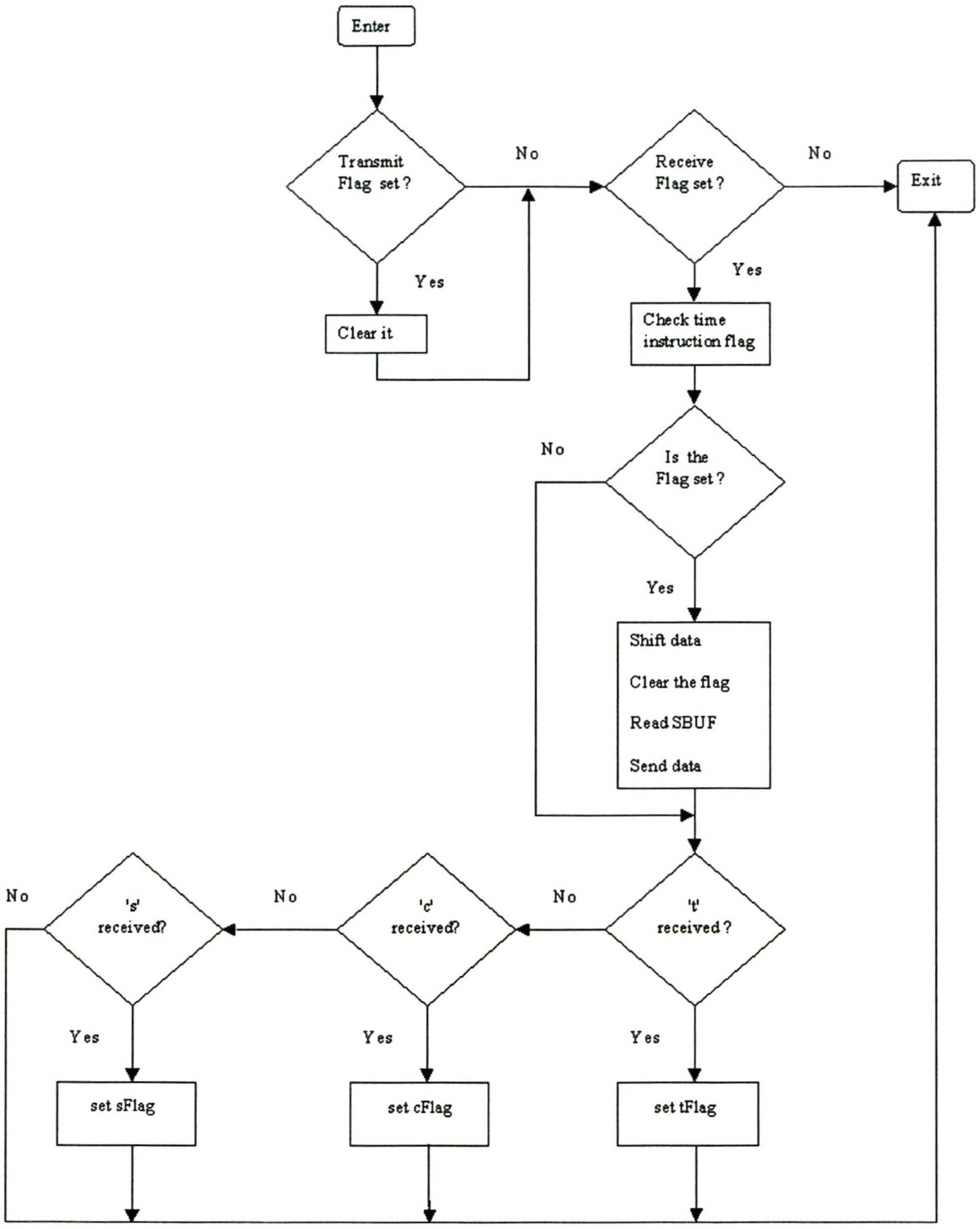


Figure 6.3: Serial port interrupt for the irrigation controller program

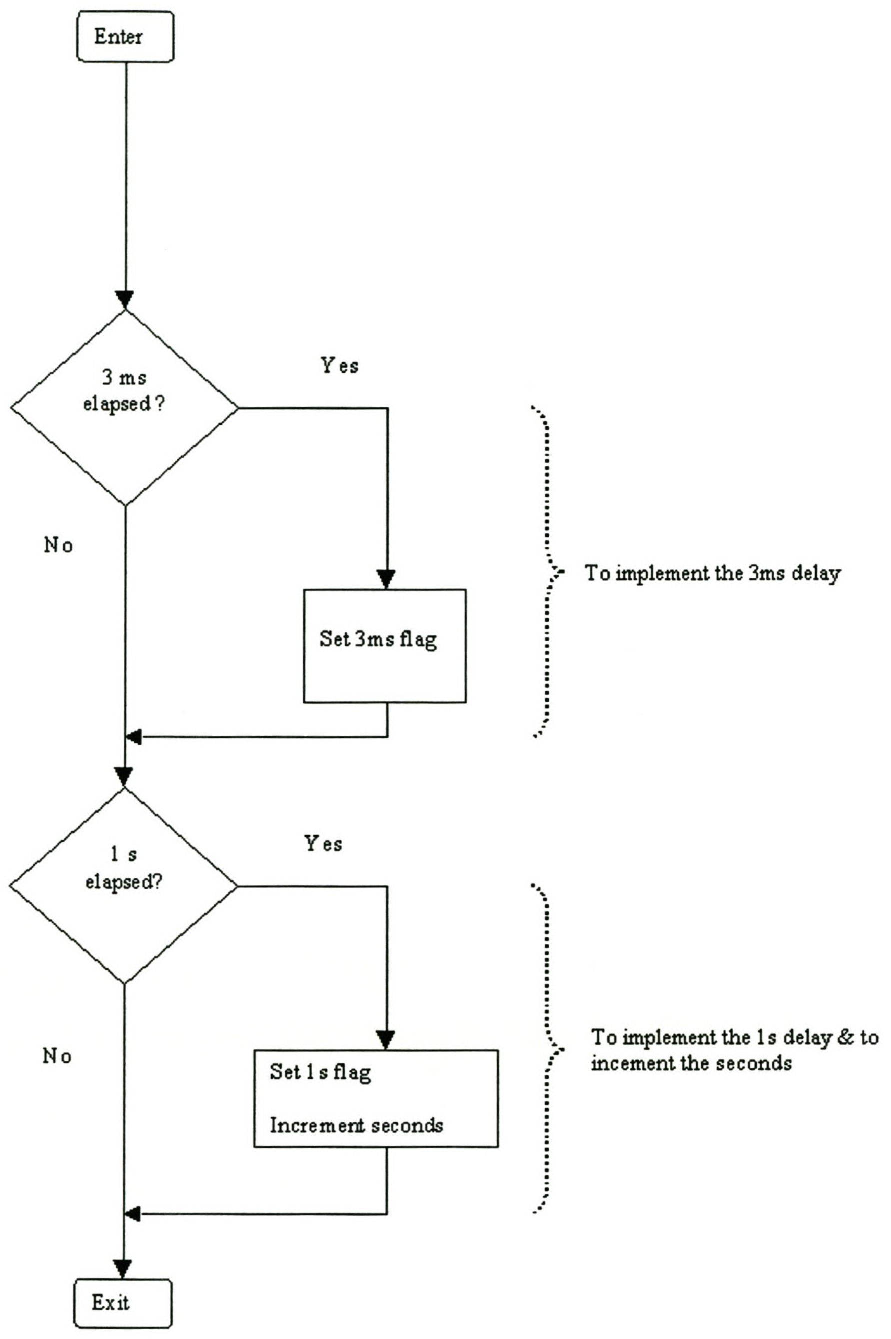


Figure 6.4: Timer 0 interrupt for the irrigation controller program

The *update* method called in the main method (see figure 6.2) updates time when a second has elapsed by calculating the value of the digits from the value of seconds updated on interrupt. Instructions that this procedure covers include those that deal with arithmetic operations. The *update* method is shown in appendix A.

The *display* method, shown in Appendix A, simply displays the value of the clock's digits. It does not take any parameters. Displaying is done by writing to port 2. Bits 2.0 to 2.6 are used as the data bus and bit 2.7 as the latch for the data. Instructions that this procedure covers include those that deal with logic operations.

The *dispset* method (for displaying the set time, figure 6.2) is similar to the *display* method except that it displays the digits of the time set for opening the valve. Both methods start by clearing the display. Clearing the LCD is done by writing the LCD *clear* instruction to port 2 and clocking the LCD control by toggling P1.7 low and high. The flowchart for the *dipset* method is shown in Appendix A.

The *checktime* method, shown Appendix A, compares the current time of the clock with the time set to open the valve. If times match, the valve is opened. P0.3 is used to simulate the valve.

In addition to arithmetic, logic, control flow, interrupt and SFR operations covered by instructions in methods that were described, other instructions involve operation on the OS and LVA, fields and method invocation. To test the concept of parameter passing, the *display* method was altered to take time digits as parameters. In place of *dispset*, *display* was used but with the parameters for the time set for the valve to open. The results of this program are discussed in section 6.2. In addition to this, a program with only one method that takes parameters (shown in appendix E) was written to observe the behaviour of parameter passing in a program with lesser methods than the irrigation controller program.

To evaluate the size of the program compiled using the JBC, a program was written in Java and compiled with the JBC and compared against the program written in assembly and C languages following the same algorithm. The C program was compiled using the Keil Software's μVision compiler. The flowchart of this program is shown in Appendix A. The source code for the three languages is shown in Appendix E. To determine the size of code generated by the JBC to accommodate the SFR's and interrupts, a Java program that only consists of an empty Java application's *main* method was also written. This program was compared against another Java program that also consists of an empty *main* method and empty reserved interrupt methods.

Problems Encountered

The following problems were encountered during the testing of the JBC. These problems necessitated the redesign of the JBC in order to solve the problems.

1. When testing the timer 0 interrupt procedure, the program was found to run for some time and stop. It was found that control flow instructions that involve comparison of values that have to be first stored on the pre-defined variables changed the values stored in the variables on interrupt, causing the program to get wrong values on return from the interrupt. The problem was solved by saving all pre-defined variables and registers on invoking both interrupts and normal methods.
2. A problem was also encountered with the SP that grew into the beginning of the data segment on interrupt. Re-designing the data segment to begin at 40H and the FieldArea to begin at 60H instead of the original 40H and 50H respectively solved this problem.

The results discussed in the next section are for the JBC after the changes just described were made. Programs used to compare the program size and to test parameter passing are presented in Appendix E.

6.2. Results

The irrigation controller program was compiled using the JBC and run on the tutor board that was described. The program consists of 28 fields, 6 methods and 20 SFR methods. The program executed as expected.

Using the EMILY51/52 simulator, the start and the return values for the OS, LVA, FieldArea and SP were tracked. The maximum size of the OS and SP for each methods described in section 6.1 was also tracked. Because of the complexity of the time at which an interrupt can occur, the maximum size of SP and OS on interrupts was tracked by observing the behaviour of the SP and OS when the program is running in the *while* loop.

Observations

1. The LVA starts at 7DH for all methods except for the program methods (*update, display, dipset, checkTime*) that start at 7FH. In all cases it returns to its original address on return from the method.
2. The OS starts at 7EH when the *main* method starts. It is re-intitialised to 7EH plus the number of local variables (2 in this case) of the *main* method on return from the *javac* generated <*clinit*> method for initialsing the fields. The OS start for the SFR methods up to entry into the *while* loop was observed to be 80H except for the program methods that started at 81H.
3. When in the *while* loop, the OS grew to a maximum of 82H on serial interrupt and 84H on timer interrupt.
4. When in the *while* loop, the SP grew to a maximum 46H on serial interrupt and 4EH on timer interrupt.

The summary of these results is presented in table 6.1 below. Also recorded in table 6.1 is the size of the Java source code, the generated assembly code that was compiled with JBC and the hex file that was loaded on the micro-controller.

Table 6.1: Summary of the results of the irrigation controller program

No of fields	28
No of methods	06
No of SFR methods	20
Max stack on interrupt 1	46H (Serial)
Max stack on interrupt 2	4EH(Timer 0)
Max OS on interrupt 1	82H (Serial)
Max OS on interrupt 2	84H(Timer 0)
Program size	Java Source: 8 kb
	Generated : 67 kb
	Hex : 4457 b

6.3. Discussion

The results presented in section 6.2 are evaluated in terms of the run time data areas, the instruction set, special methods, program memory, readability and, code efficiency and execution speed. This section focuses on each one of them.

Run time data areas

The results in section 6.1 show that the JBC creates the run time data areas as designed in chapter 4 with the changes made in section 6.2. The FieldArea begins at the fixed address of 60H at all times. This is above the data segment space reserved for pre-defined variables needed by the JBC to store some data needed at run time. The LVA begins above the FieldArea at the position *FieldArea + 1* on entry. It is re-initialised to 1CH on return from the method for initialising fields. The LVA is re-initiliased to this value because there are 28 fields in this class, i.e., *start of LVA = start of FieldArea(R3) + no of fields = 61H + 1CH = 7DH*. This value may, however, impose problems for the 8051 because it has only 7FH of general purpose RAM. Fields of *boolean* type take much of the RAM because they are represented as full byte whereas they could be represented in the bit-addressable segment (DBIT) in assembly language. The OS is then initialised above the LVA area at *start of OS = start of LVA + no. of local variables = 7EH + 2 = 80H*. This shows that the OS starts above the general purpose RAM. As explained earlier, this may impose problems for the 8051 but works for larger micro-controllers.

The SP is initialised at 30H at all times. As observed when interrupts occurred, the SP may grow into the beginning of the data segment (beginning at 50H). This may also cause the program to run incorrectly. Initialising the SP above the Java stack area requires knowledge of the possible maximum size of the Java stack for all methods.

In general, these results demonstrate the success of the JBC in preparing the necessary run time data areas for running Java programs. The interrupt methods and the methods within the program change frames successfully when they are invoked with the OS of the calling method becoming the LVA of the called method. Thus the JBC complies with the JVM's requirements and is able to reclaim memory used during method invocation.

Instruction Set

The output of the code generated from a code generator must be correct and of high quality (Aho, et al, 1986:513). The Irrigation Controller program described demonstrates success of the operation of all instruction set types discussed in section 5.3. The calculation of time in the *update* method, for instance, is a good example of the ability of the JBC to handle arithmetic operations. With the ORing of data written to port 2, the *display* procedure demonstrates the ability to handle logic operations. The rest of the other instruction types also demonstrate correct execution. Thus the JBC does generate correct code.

The program, however, executed for some time and stopped when the *display* method was changed to take parameters. A simpler program shown in appendix E, however, demonstrated success in parameter passing. The problem with parameter passing for a large program is anticipated to be due to the SP that grows deep as the program enlarges, especially on interrupts. The quality of the code will be discussed when efficiency is discussed.

Special methods

As mentioned when runtime data areas were discussed earlier in the section, special methods have behaved correctly in all circumstances. The subroutines associated with the methods are however large, taking considerable program memory. This is due to the fact that methods have been generalised to get a value from the OS and decide the SFR on which to operate on based on the value received.

The interrupts also behaved correctly but due to the need to save variables and registers on interrupt; the SP grows quite deep as seen in table 6.1.

Program Memory

A program was written in three different languages following the same algorithm. The programs are shown in appendix E. The hex file of the program compiled with the JBC is large (607 bytes) when compared with a similar program written in assembly language (54 bytes) and C (168 bytes). The hex file for the program with an empty *main* method is 56 bytes when no interrupt methods are written and 419 bytes when empty interrupt methods are included.

The observation of the large size of the code generated with the JBC can be attributed to, firstly, SFR subroutines. The SFR subroutines are longer because they are written in such a way that those subroutines that manipulate data accept a value from the OS and decide which SFR to operate on. The SFR subroutines can be improved by breaking them into subroutines of smaller size. Only the required size could be written into code memory as opposed to the generalised subroutine.

Secondly the JBC compiled code is large because the JBC provides default interrupt vectors and their corresponding subroutines. These are provided regardless of whether the program uses interrupt methods or not as has been observed with the program with an empty *main* method that compiled to a hex file of 56 bytes in size. This makes all programs compiled with JBC to have code with an offset of 56 bytes in size. If interrupt methods are included, the size of the hex file enlarges even if the methods are empty. This is because interrupt methods require pre-defined variables to be saved on interrupt. This is done in order to make sure that data used by instructions such as control flow instructions is not lost during handling of the interrupt. A flag could be used to tell the JBC whether to include interrupt methods or not.

Lastly, the code is large because the JBC provides code that will follow the stack based JVM specifications for operating on run time data areas. An operation that could be done in one instruction, for instance, usually requires more than one instruction. For example, to write a value to port 1 would be done in the JBC implementation by taking five instructions as follows:

```
dec R0
mov A, @R0
mov P1, A
mov A, R2
ret
```

whereas it could be done by

```
mov P1, #Value
```

in assembly.

These observations show that while code generated by the JBC is large, it could still be optimised. The programs with empty methods written to demonstrate the offset of the size of the hex file generated show that unnecessary code is generated. This problem could be solved by carefully determining code that is necessary for a particular program.

One approach would be to package code that may be needed into libraries. A reserved statement could be used to tell the JBC of code that is necessary to include in the generated code. Following the stack-based nature of the JVM also leads to large code. Once all variables are known, a second program can be used to eliminate unnecessary code by storing variables at addresses that can be known at run time. In other words, the JBC could be used to generate code from the class file and the second program can be used to optimise code generated by the JBC. This would reduce size of the hex file further.

Readability

For the program written in Java, C and assembly using the same algorithm, it was observed that the program written in Java is more readable, small and easy to follow than the program written in assembly. The program is also comparable but simpler to that written in C. The Java program is easier because of its simplified syntax. This allows use of simplified and more elaborate variables.

Portability

Portability of the class file for programs written for the JBC is still maintained. Programs written for the JBC can still be run on other devices without alteration of the structure of the class file.

Efficiency and Speed

The code generated by the JBC is large and runs slower than the same code written in assembly. For example, an implementation of a Java instruction

```
x++;
```

generates a number of assembly instructions as follows:

```
mov A, R2      ;R2 = start of LVA [12]
mov R1, A      ;prepare to point to the index in the LVA where x is found [12]
add A, #index  ;add the index [12]
mov R1, A      ;R1 (LVA pointer) now points at LVA index for x [12]
mov A, @R1     ;write the contents at that index to the accumulator [12]
add A, #1      ;increment [12]
mov @R1, A     ;write the result to the LVA index for x [24]
```

where, the number in square brackets denotes the number of clock cycles needed to execute the instruction. This instruction could be done in assembly with the following code:

```
mov A, Variable [12]
inc A    [12]
mov Variable, A [12]
```

As can be seen with the clock cycles, the JBC implementation needs a total of 96 cycles to execute this instruction whereas this could be done in 36 cycles with ordinary assembly. Slow execution may pose problems for instructions that are strictly time-dependent such as during handling of interrupts. This is possible, for instance, when updating seconds in the irrigation controller program discussed. If the statement *if (seconds == 60){}* is used, the program might miss the exact time while it is still executing other instructions leaving only one option of using *if (seconds >= 60){}*, instead.

The reasons why the JBC generates code this way are explained in the previous section when program memory was discussed. Note, however, that because of its interpreted nature, the Java language is itself considered relatively slower. Although the JBC generated code is large and slower; the correct behaviour of the irrigation controller program shows that the code is correct and effective but should probably be optimised. Optimization recommendations will be discussed in chapter 7.

Summary

The code generated in the program discussed in this chapter meets the JBC specifications discussed in chapter 4. The run time data areas are designed correctly. The instruction set also meets the specification with an exception of parameter passing for large programs. The special methods and the interrupts are also handled correctly. The code generated by the JBC is however larger than that for programs written in assembly and C languages but can be reduced extensively as recommended in chapter 7. The readability of Java programs compared to that written in assembly is satisfactory. The program written in Java is also simpler and easy to follow than written in C. The problem of efficiency raised in the discussion is attributed to the overall architecture of the Java language that the JBC implementation follows.

Chapter 7

Conclusion and Recommendations

7.1. Specifications

The results discussed in chapter 6 show that the JBC works correctly. The JBC has shown success with the allocation of the runtime data areas defined in the JVM specification. The results also demonstrate that the JBC does generate correct instructions that execute effectively on the 8051 micro-controller. The execution speed of the code could, however, be optimised by carefully reducing the size of the code. Normal 8051 operations such as accessing the I/O ports and interrupts are also handled correctly. Code written in Java is more readable, smaller, portable and easier to correct for mistakes than code written in assembly and C. Thus the specifications of the research have been met. In general, the JBC is a software tool with a full API that can be used successfully to program the 8051 and its family of micro-controllers using Java without compromising its normal operations such as handling interrupts and accessing I/O ports and external memory. The tool is suitable especially for people who prefer programming in Java than other languages. The tool is more suitable for smaller programs that do not require strict code efficiency but can be optimised to suit larger programs as well.

Shortcomings of the JBC include, firstly, that the generated executable is large and takes considerable amount of internal program memory. This can be attributed to longer subroutines associated with the SFR methods, unnecessary inclusion of default interrupt vectors and their ISRs, and the general lengthy structure of code associated with stack-based requirements of the Java language.

Secondly, code generated by the JBC runs slower than code that would have been compiled from a program written in assembly language. This is attributed to the lengthy code structure described when the first shortcoming was discussed. It can also be attributed to the interpreted nature of the Java language that was followed throughout the implementation of the JBC. This problem will be reduced when the first shortcoming is addressed.

Thirdly, the JBC does not support arrays. This problem can also be solved as will be seen when recommendations are made.

Lastly, the SP may grow into the beginning of the RAM reserved for pre-defined variables needed for the JBC operations (in the data segment) for larger programs, especially when interrupts occur. This can be attributed to the need to save variables when interrupts occur. The OS also grows beyond 7FH. While this is not a problem for higher capacity micro-controllers, it may be a problem for the 8051 and smaller micro-controllers. This problem can also be attributed to a lengthy structure of the code generated by the JBC.

7.2. Recommendations

The shortcomings that have been discussed can be solved or at least lessened. Because of the magnitude of the project, lack of documentation on the JVM and compilers and the time constraints, these improvements are only given as recommendations. To optimise the JBC, the following recommendations are given:

1. Instead of generating default code for interrupt vectors and their subroutines, the JBC can determine in advance interrupts that will be needed and generate only the code that is needed for the particular interrupt.
2. SFR subroutines can also be broken down and only the piece that is needed can then be generated instead of having a full subroutine even for operations that are not needed.
3. Frequently used instructions such as *bipush* can be placed at the bottom of the code instead of generating them all the time when they are needed.

4. Code that may be needed for certain operations can be packaged into libraries. A reserved statement such as the *import Package* statement of the Java language could be used to tell the JBC of code that is necessary to include in the code to generate.
5. The process of changing frames can also be written as a subroutine instead of generating code all the time when a method is called.
6. Another program can be written to read the generated assembly code and break it down into a smaller program that eliminates code that is necessary.
7. Arrays can be implemented in a similar way in which fields have been implemented. That is, the Java stack may be designed to include memory for arrays. This may be allocated dynamically only when needed as in the case of fields. Implementing arrays will, however, only reduce the size and improve readability of the Java source code, but will still take up a lot of internal memory.

7.3. Known Error

The following error has been observed. Unfortunately, time did not allow solving it:

1. The *while* loop in the irrigation controller program (section 6.1, Appendix E) uses the variable *loop* as follows:

   while (loop = = true){}

   where *loop* has been made *true*. If the same instruction code is written without the variable *loop*, that is *while (true){}*, the JBC experiences an *out of bounds exception*.

References

1. Appel, A.W. 1997. *Modern compiler implementation in Java*. Preliminary edition. Cambridge: Cambridge University Press
2. Aho, A.V., Sethi, R. & Ullman, J.D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.
3. Berg, D.J. & Fritzinger, J.S. 1999. *Advanced techniques for Java™ developers*. Revised edition. Wiley Computer Publishing: John Wiley & Sons, Inc.
4. Computer Technology Research Corporation. 1998. *High-performance application development for the Internet and Intranets*. 1st edition. Charleston: Computer Technology Research Corporation.
5. Lewis, G., Barber, S. & Siegel, E. 1998. *Programming with Java IDL*. John Wiley.
6. Lindholm, T. & Yellin, F. 1999. *The Java™ virtual machine specification*. 2nd edition. Addison-Wesley.
7. Lindholm, T. 2000. *JVM for a 8031*. E-mail to T.J. Mbhambhu[Online]. 05 Oct., Available E-mail: tjmbhambhu@webmail.co.za [2001, November 26].
8. MacKenzie, I.S. 1999. *The 8051 microcontroller*. 3rd edition. New Jersey: Prentice-Hall.
9. Schildt, H. 2001. *Java™ 2: The complete reference*. 4th edition. Osborne: McGraw-Hill.
10. Schultz, T.W. 1998. *C and the 8051: hardware, modular programming, and multitasking*. 2nd edition. New Jersey: Prentice-Hall, Inc.
11. Steenkamp, N.L. 1999. *Development of the On Board Computer Flight Software for SUNSAT 1*. Master of Electronic Engineering thesis. University of Stellenbosch. Stellenbosch.
12. Sun Microsystems. 2000a. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*. White Paper. Sun Microsystems. USA

13. Sun Microsystems. 2000b. *K Virtual Machine APIs* [online]. Available: http://developer.java.sun.com/developer/technicalAerticles/wireless/midapi/. [2001, August 02].

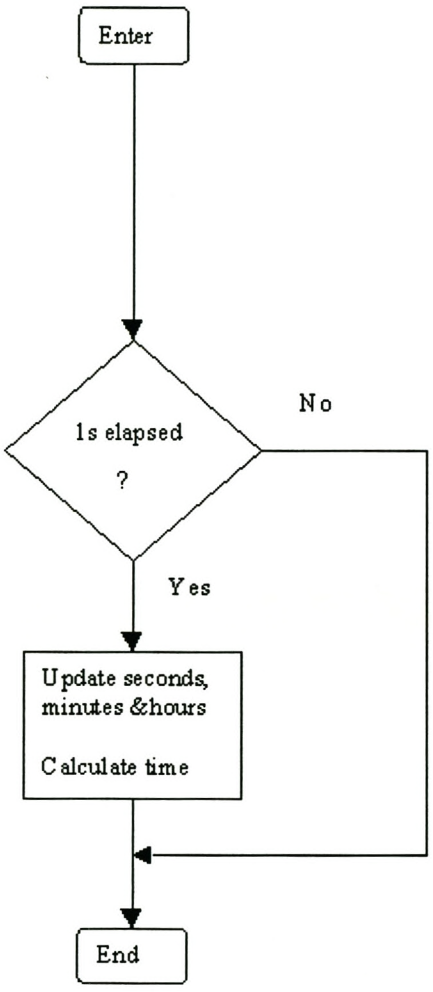
14. Venners, B. 1999. *Inside the Java 2 Virtual Machine*. 2nd edition. McGraw-Hill.

15. Yeralan, S. & Ahluwalia, A. 1995. *Programming and interfacing the 8051 microcontroller*. Addison Wesley.

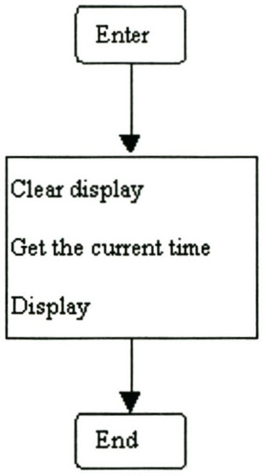
Appendix A

Flowcharts

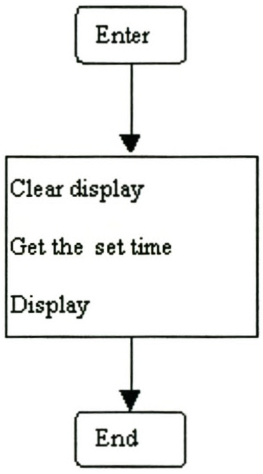
A1 Update Method



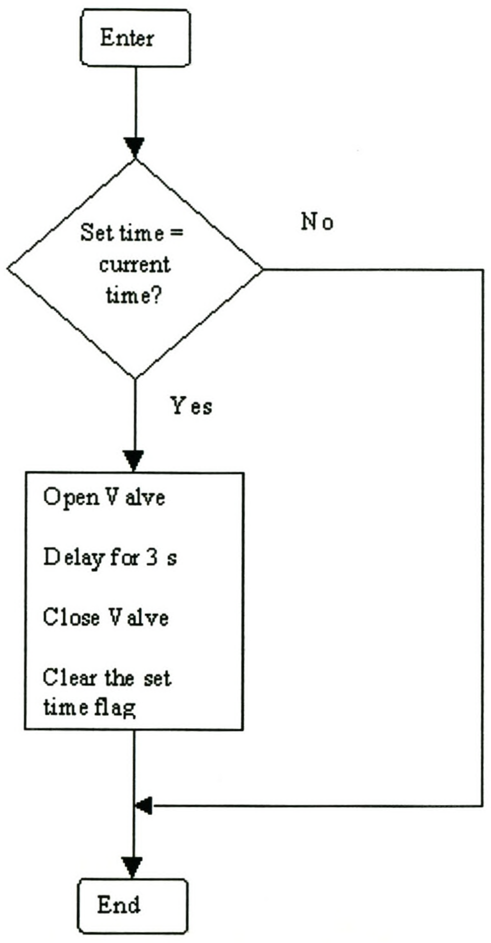
A2 Display Method



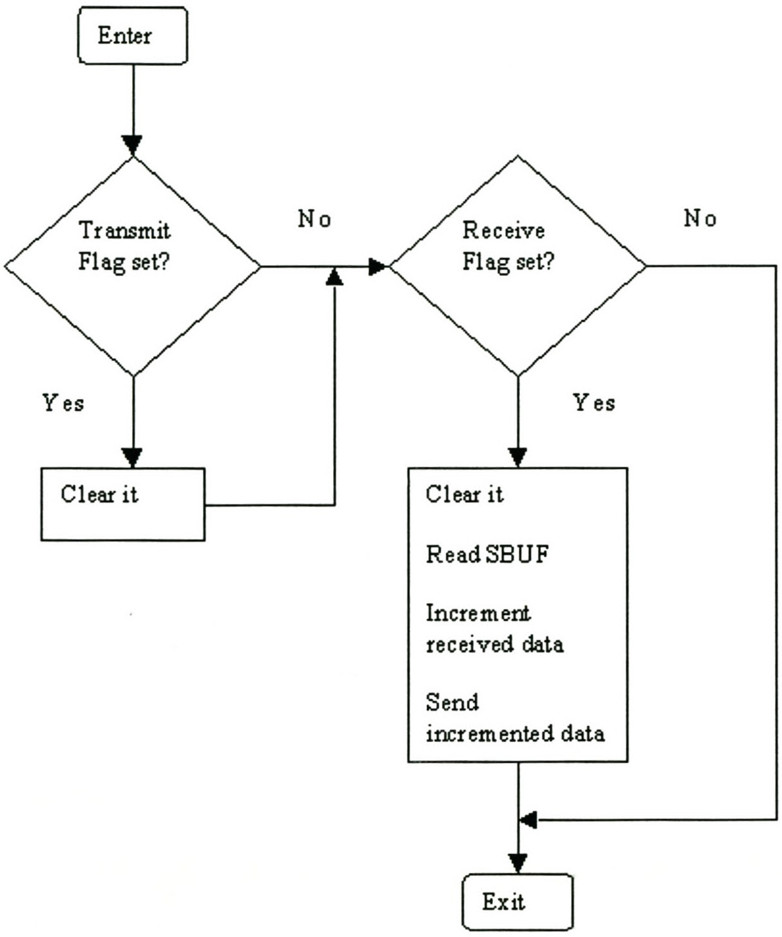
A3 Dipset Method



A4 CheckTime Method



A5 Serial Port program for comparison



Appendix B

Tables

B1 The JVM Type Support

Type	Size (bits)
byte	08
short	16
int	32
long	64
char	16

All types shown above represent signed two's complements except for *char*.

B2 The AT89CS Alternate Pin Usage

PORT PIN	ALTERNATE FUNCTION
P1.0	T2 (Timer/Counter 2 external input)
P1.1	T2EX (Time/Counter 2 Capture/Reload trigger)
P3.0	RXD (Serial Port input)
P3.1	TXD (Serial Port output)
P3.2	INTO (External Interrupt 0)
P3.3	INT1 (External Interrupt 1)
P3.4	T0 (Timer/Counter 0 external input)
P3.5	T1 (Timer/Counter 1 external input)
P3.6	WR (External Data Memory write strobe)
P3.7	RD (External Data Memory read strobe)

B3 Description of class file structures

Entry	Item	Description
1	u4 magic	class file's identification number
2	u2 minor_version	First part of the class file's version number
3	u2 major_version	Second part of the class file's version number
4	u2 constant_pool_count	The number of constant pool entries plus one more
5	cp_info constant_pool[constant_pool_count]	Contains structures representing string constant, class and interface names, field names and other constant referred to in the class file (Lindolm, 1999:95)
6	u2 access_flags	A masked number showing whether a class or interface is public, final, super, interface or abstract. Values are shown in Lindholm(1999:96).
7	u2 this_flags	Gives an index into constant pool for a reference to a class (Lindholm, 1999:97)
8	u2 super_class	It must be zero or refer to the class in the constant pool (Lindholm, 1999:97)
9	u2 interfaces_count	Gives an index into the constant pool for direct superinterfaces of a class or interface type
10	u2 interfaces[interface_count]	An array of interfaces that must correspond to a class representing a direct superinterface
11	u2 fields_count	Gives the number of class or interface fields or variables
12	field_info fields[fileds_count]	Gives complete description about fields
13	u2 methods_count	Gives the number of methods
14	method_info methods[methods_count]	Gives complete description about methods (Lindholm, 1999: 98)
15	u2 attributes_count	Gives the number of attributes of a class
16	attribute_info attributes[attributes_count]	Gives complete description about the class (Lindholm, 1999: 98)

B4 Constant pool Tags

Constant Type	**Value**
CONSTANT Class	7
CONSTANT Fieldref	9
CONSTANT Method ref	10
CONSTANT InterfaceMethod ref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT Float	4
CONSTANT Long	5
CONSTANT Double	6
CONSTANT NameAndType	12
CONSTANT Utf8	1

Appendix C

Supported Bytecode

C1 Arithmetic Instructions

Opcode	Operands and Action	Assembly
iadd	-	*dec R0* *mov A, @R0* *dec R0* *ADD A, @R0* *mov @R0, A* *inc R0*
isub	-	*dec R0* *mov A, @R0* *mov R4, A* *dec R0* *mov A, @R0* *SUBB A, R4* *mov @R0, A* *inc R0*
imul	-	*dec R0* *mov A, @R0* *dec R0* *mov B, @R0* *MUL AB* *mov @R0, A* *inc R0*
idiv	-	*dec R0* *mov B, @R0* *dec R0* *mov B, @R0* *DIV AB* *mov @R0, A* *inc R0*
irem	-	*dec R0* *mov B, @R0* *dec R0* *mov B, @R0* *DIV AB* *mov @R0, B* *inc R0*
ineg	-	*dec R0* *mov A, @R0* *mov R4, A* *mov A, #0* *SUBB A, R4* *mov @R0, A* *inc R0*

R0 = Operand stack pointer

C2 Control Flow Instructions

Opcode	Operands and Action	Assembly
ifeq	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*Dec R0* *Mov A, @R0* *Jnz IfeqEndcount* *IfeqJmpcount : JMP OpcodebranchOffset* *IfeqEndcount : JMP Opcode(count+numRead)*
Ifne	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*Dec R0* *Mov A, @R0* *Jnz IfneJmpcount* *JMP ifneEndcount* *IfneJmpcount : JMP OpcodebranchOffset* *IfneEndcount : JMP Opcode(count+numRead)*
Iflt	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*Dec R0* *mov CompVal, @R0* *mov A, #0* *cjne A, CompVal, IfltJmpccount* *JMP IfltEndcount* *IfltJmpccount : JC IfltJmpcount* *IfltJmpcount : JMP OpcodebranchOffset* *IfltEndcount : JMP Opcode(count+numRead)*
Ifle	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*Dec R0* *mov CompVal, @R0* *mov A, #0* *cjne A, CompVal, IfleJmpccount* *JMP IfleJmpcount* *IfleJmpccount : JC IfleJmpcount* *JMP IfleEndcount* *IfleJmpcount : JMP OpcodebranchOffset* *IfleEndcount : JMP Opcode(count+numRead)*
Ifgt	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *mov A, #0* *cjne A, CompVal, IfgtJmpccount* *JMP IfgtEndcount* *IfgtJmpccount : JNC IfgtJmp count* *JMP IfgtEndcount* *IfgtJmpcount : JMP OpcodebranchOffset* *IfgtEndcount : JMP Opcode(count+numRead)*
Ifge	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *mov A, #0* *cjne A, CompVal, IfgeJmpccount* *JMP IfgeJmpcount* *IfgtJmpccount : JNC IfgeJmp count* *JMP IfgEndcount* *IfgeJmpcount : JMP OpcodebranchOffset* *IfgeEndcount : JMP Opcode(count+numRead)*
If_cmpeq	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *dec R0* *mov A, @R0* *cjne A, CompVal, CmpeqEndcount* *JMP CmpeqJmpcount* *CmpeqJmpcount : JMP OpcodebranchOffset* *CmpeqEndcount : JMP Opcode(count+numRead)*

branchOffset = address to jump to; CompVal = general variable used to store values for comparison; Opcode = label; undelined = a number; count = track of number of bytes read; numRead = number of bytes read after this instruction

Control Flow Instructions (cont..)

Opcode	Operands and Action	Assembly
If_cmpne	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *dec R0* *mov A, @R0* *cjne A, CompVal, CopneJmpcount* *JMP CmpneEndcount* *CmpneJmpcount : JMP OpcodebranchOffset* *CmpneEndcount : JMP Opcode(count+numRead)*
If_cmplt	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *dec R0* *mov A, @R0* *cjne A, CompVal, CmpltJmpccount* *JMP CmpltEndcount* *CmpltJmpccount : JC CmpltJmpcount* *CmpltJmpcount : JMP OpcodebranchOffset* *CmpltEndcount : JMP Opcode(count+numRead)*
If_cmple	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *dec R0* *mov A, @R0* *cjne A, CompVal, CmpleJmpccount* *JMP CmpleJmpcount* *CmpleJmpccount : JC CmpleJmpcount* *CmpleJmpcount : JMP OpcodebranchOffset* *CmpleEndcount : JMP Opcode(count+numRead)*
If_cmpgt	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *dec R0* *mov A, @R0* *cjne A, CompVal, CmpgtJmpccount* *JMP CmpgtEndcount* *CmpgtJmpccount : JNC CmpgtJmpcount* *CmpgtJmpcount : JMP OpcodebranchOffset* *CmpgtEndcount : JMP Opcode(count+numRead)*
If_cmpge	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*dec R0* *mov CompVal, @R0* *dec R0* *mov A, @R0* *cjne A, CompVal, CmpgeJmpccount* *JMP CmpgeJmpcount* *CmpgeJmpccount : JNC CmpgeJmpcount* *JMP CmpgeEndcount* *CmpgeJmpcount : JMP OpcodebranchOffset* *CmpgeEndcount : JMP Opcode(count+numRead)*
goto	*int count = number of bytes already read* *branchbyte1, branchbyte2* *int branchOffset =* *((branchbyte1 << 8 | branchbyte2) + count)* *numRead = 2*	*JMP OpcodebranchOffset*

branchOffset = address to jump to; CompVal = general variable used to store values for comparison;
Opcode = label; undelined = a number; count = track of number of bytes read; numRead = number of bytes read after this instruction

C3 Method Invocation Instructions

Opcode	Operands and Action	Assembly
invokestatic	*indexbyte1, indexbyte2* *int index = indexbyte1 << 8 | indexbyte2* *String methodname = (constantpool[constantpool [constantpool[index].nameandtypeindex].nameindex]. Utf8String)* *String parameters = (constantpool[constantpool [constantpool[index].nameandtypeindex]. descriptorindex]. Utf8String)*	*If SFR method,* *call classmethodname* *else* *push 2* *push 0* *push Parameters* *push FmaxLocals* *mov Parameters, #noofparameters* *mov A, R0* *subb A, Parameters* *mov R2, A* *add A, Parameters* *mov R0, A* *call classmethodname* *pop FmaxLocals* *pop Parameters* *pop 0* *pop 2*
ireturn	-	*mov A, R2* *ret*
return	-	*If method=interrupt method,* *;---------* *else* *mov A, R2* *ret*

R0 = Operand Stack pointer, R2 = start of LVA = FieldArea + local variables, Parameters = variable for no. of parameters, FmaxLocals = maximum no. of local variables, noofparameters = no of parameters, constantpool[] = Constant Pool.

NB - Also refer to chapter 2 for terminology used.

C4 Logic Instructions

Opcode	Operands and Action	Assembly
iand	-	*dec R0* *mov A, @R0* *dec R0* *ANL A, @R0* *mov @R0, A* *inc R0*
ior	-	*dec R0* *mov A, @R0* *dec R0* *ORL A, @R0* *mov @R0, A* *inc R0*
ixor	-	*dec R0* *mov A, @R0* *dec R0* *XRL A, @R0* *mov @R0, A* *inc R0*

R0 = Operand Stack pointer

C4 Fields Instructions

Opcode	Operands and Action	Assembly
getstatic	*indexbyte1, indexbyte2* *int index = indexbyte << 8 | indexbyte 2* *String fieldname= constant pool[index]* *int getField = fieldArray[fieldname]*	*mov A, R3* *add A, #getField* *mov R1,A* *mov A, @R1* *mov @R0, A* *inc R0*
putstatic	*indexbyte1, indexbyte2* *int index = indexbyte << 8 | indexbyte 2* *String fieldname = constant pool[index]* *int getField = fieldArray[fieldname]*	*dec R0* *mov A, R3* *mov R1, A* *add A, #getField* *mov R1,A* *mov A, @R0* *mov @R1, A*

R3 = 60H = Start of FieldArea; fieldArray = array of fields (section 5.2.3)
R0 = Operand Stack pointer; R1 = FieldArea pointer, constantpool[] = Constant Pool

C5 LVA and OS Instructions

Opcode	Operands and Action	Assembly
bipush	*value*	*mov R0, #value*
nop	-	*nop*
Pop	-	*dec R0* *mov A, @R0*
pop2	-	*dec R0* *dec R0* *mov A, @R0*
dup	-	*dec R0* *mov A, @R0* *mov @R0, A* *inc R0* *mov @R0, A* *inc R0*
dup2	-	*dec R0* *dec R0* *mov A, @R0* *inc R0* *inc R0* *mov @R0, A* *dec R0* *mov A, @R0* *inc R0* *mov @R0, A* *inc R0*
dup_x1	-	*dec R0* *mov A, @R0* *dec R0* *dec R0* *mov @R0, A* *inc R0* *inc R0* *inc R0*
dup_x2	-	*dec R0* *mov A, @R0* *dec R0* *dec R0* *dec R0* *mov @R0, A* *inc R0* *inc R0* *inc R0* *inc R0*
iload	*index*	*mov A, R2* *mov R1, A* *add A, #index* *mov R1, A* *mov A, @R1* *mov @R0, A* *inc R0*

R2 = FieldArea + local variables = start of LVA, R0 = Operand Stack pointer, index = index into LVA; value = value to push onto Operand Stack

LVA and OS Instructions (cont..)

Opcode	Operands and Action	Assembly
iload_0	-	*mov A, R2* *mov R1, A* *add A, #0* *mov R1, A* *mov A, @R1* *mov @R0, A* *inc R0*
iload_1	-	*mov A, R2* *mov R1, A* *add A, #1* *mov R1, A* *mov A, @R1* *mov @R0, A* *inc R0*
iload_2	-	*mov A, R2* *mov R1, A* *add A, #2* *mov R1, A* *mov A, @R1* *mov @R0, A* *inc R0*
iload_3	-	*mov A, R2* *mov R1, A* *add A, #3* *mov R1, A* *mov A, @R1* *mov @R0, A* *inc R0*
istore	*index*	*dec R0* *mov A, R2* *mov R1, A* *add A, #index* *mov R1, A* *mov A, @R0* *mov @R1, A*
istore_0	-	*dec R0* *mov A, R2* *mov R1, A* *add A, #0* *mov R1, A* *mov A, @R0* *mov @R1, A*
istore_1	-	*dec R0* *mov A, R2* *mov R1, A* *add A, #1* *mov R1, A* *mov A, @R0* *mov @R1, A*

R2 = FieldArea + local variables = start of LVA, R0 = Operand Stack pointer, index = index into LVA; value = value to push onto Operand Stack

LVA and OS Instructions (cont..)

Opcode	Operands and Action	Assembly
istore_2	-	*dec R0* *mov A, R2* *mov R1, A* *add A, #2* *mov R1, A* *mov A, @R0* *mov @R1, A*
istore_3	-	*dec R0* *mov A, R2* *mov R1, A* *add A, #3* *mov R1, A* *mov A, @R0* *mov @R1, A*

R2 = FieldArea + local variables = start of LVA, R0 = Operand Stack pointer, index = index into LVA; value = value to push onto Operand Stack

Appendix D

Application Programming Interface

D1 Supported Packages

Package Tree	Methods
•java.lang.Object	-
•java.lang.Boolean	booleanValue()

•java.lang.Character	charValue()
	compareTo(Character anotherCharacter)

•java.lang.Math	abs(int a)
	max(int a, intb)
	min(inta, int b)

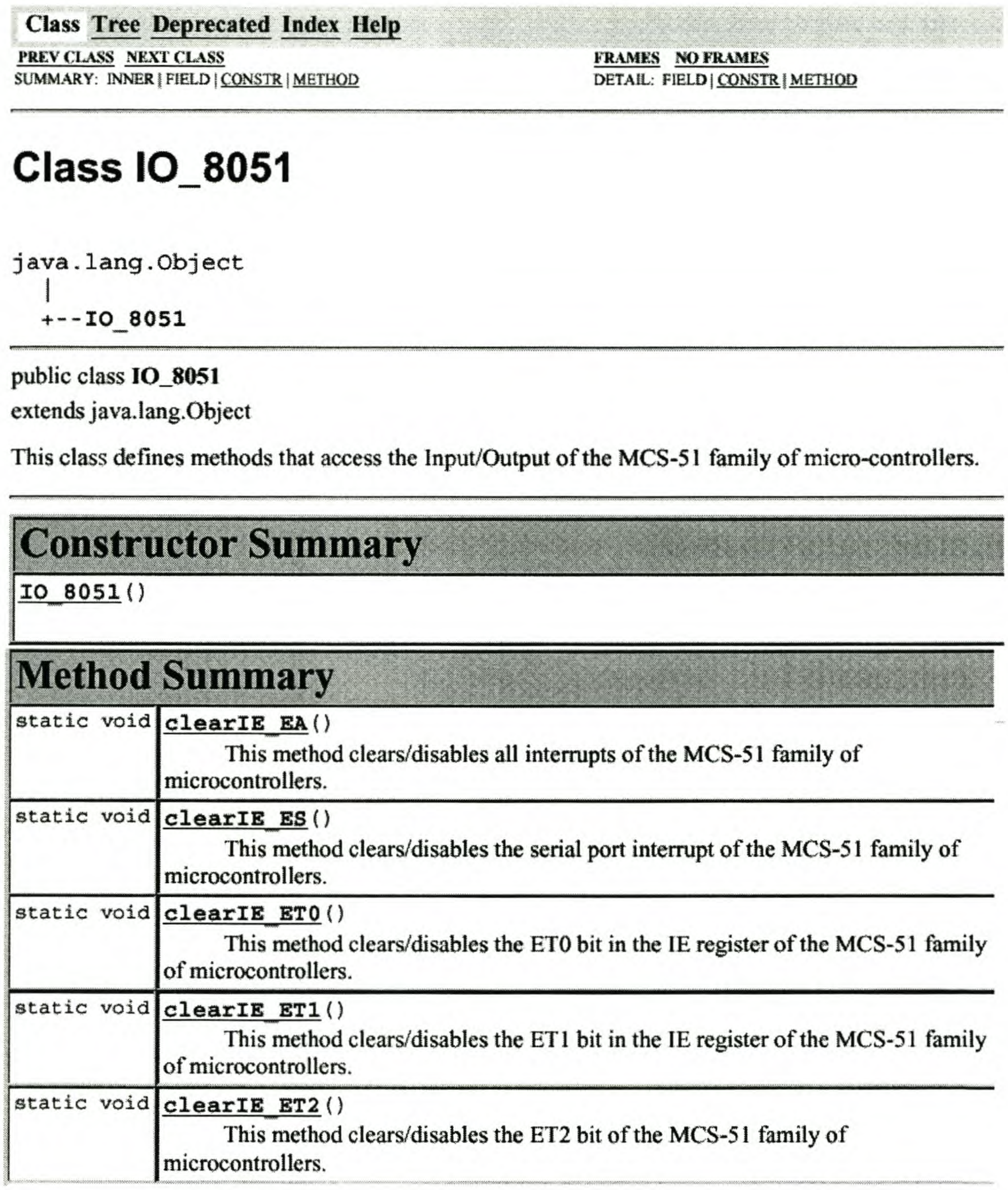
•java.lang.Number	byteValue()
	intValue()
	shortValue()

•java.lang.Byte	byteValue()
	compareTo(Byte anotherByte)
	intValue()
	shortValue()

•java.lang.Integer	byteValue()
	compareTo(Integer anotherInteger)
	intValue()

•java.lang.Short	byteValue()
	compareTo(Short anotherShort)
	intValue()
	shortValue()

D2 Javadoc



Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class IO_8051

```
java.lang.Object
  |
  +--IO_8051
```

public class **IO_8051**
extends java.lang.Object

This class defines methods that access the Input/Output of the MCS-51 family of micro-controllers.

Constructor Summary
`IO_8051()`

Method Summary	
`static void`	`clearIE_EA()` This method clears/disables all interrupts of the MCS-51 family of microcontrollers.
`static void`	`clearIE_ES()` This method clears/disables the serial port interrupt of the MCS-51 family of microcontrollers.
`static void`	`clearIE_ET0()` This method clears/disables the ET0 bit in the IE register of the MCS-51 family of microcontrollers.
`static void`	`clearIE_ET1()` This method clears/disables the ET1 bit in the IE register of the MCS-51 family of microcontrollers.
`static void`	`clearIE_ET2()` This method clears/disables the ET2 bit of the MCS-51 family of microcontrollers.

static void	**clearIE_EX0**() This method clears/disables the EX0 bit in the IE register of the MCS-51 family of microcontrollers.
static void	**clearIE_EX1**() This method clears/disables the EX1 bit in the IE register of the MCS-51 family of microcontrollers.
static void	**clearIE_UNDEFINED**() This IE bit is not defined.
static void	**clearIP_PS**() This method clears/disables the PS bit of the IP register of the MCS-51 family of microcontrollers.
static void	**clearIP_PT0**() This method clears/disables the PT0 bit of the IP register of the MCS-51 family of microcontrollers.
static void	**clearIP_PT1**() This method clears/disables the PT1 bit of the IP register of the MCS-51 family of microcontrollers.
static void	**clearIP_PT2**() This method clears/disables the PT2 bit of the IP register of the MCS-51 family of microcontrollers.
static void	**clearIP_PX0**() This method clears/disables the PX0 bit of the IP register of the MCS-51 family of microcontrollers.
static void	**clearIP_PX1**() This method clears/disables the PX1 bit of the IP register of the MCS-51 family of microcontrollers.
static void	**clearIP_UNDEFINED1**() This bit is undefined
static void	**clearIP_UNDEFINED2**() This bit is undefined
static void	**clearPort_0Bit**(int bit) This method clears a specifies bit of port 0 of the MCS-51 family of microcontrollers.
static void	**clearPort_1Bit**(int bit) This method clears a specifies bit of port 1 of the MCS-51 family of microcontrollers.
static void	**clearPort_2Bit**(int bit) This method clears a specifies bit of port 2 of the MCS-51 family of microcontrollers.

`static void`	`clearPort_3Bit(int bit)` This method clears a specifies bit of port 3 of the MCS-51 family of microcontrollers.
`static void`	`clearScon_RB8()` This method clears/disables the RB8 bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_REN()` This method clears/disables the REN bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_RI()` This method clears/disables the RI bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_SM0()` This method clears/disables the SM0 bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_SM1()` This method clears/disables the SM1 bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_SM2()` This method clears/disables the SM2 bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_TB8()` This method clears/disables the TB8 bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearScon_TI()` This method clears/disables the TI bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	`clearTcon_IE0()` This method clears TCON bit IE0 of the MCS-51 family of microcontrollers.
`static void`	`clearTcon_IE1()` This method clears TCON bit IE1 of the MCS-51 family of microcontrollers.
`static void`	`clearTcon_IT0()` This method clears TCON bit IT0 of the MCS-51 family of microcontrollers.
`static void`	`clearTcon_IT1()` This method clears TCON bit IT1 of the MCS-51 family of microcontrollers.
`static void`	`clearTcon_TF0()` This method clears TCON bit TF0 of the MCS-51 family of microcontrollers.

static void	**clearTcon_TF1**() This method clears TCON bit TF1 of the MCS-51 family of microcontrollers.
static void	**clearTcon_TR0**() This method clears TCON bit TR0 of the MCS-51 family of microcontrollers.
static void	**clearTcon_TR1**() This method clears TCON bit TR1 of the MCS-51 family of microcontrollers.
static void	**intrEnable**(int value) Interrupt Enable method This method writes a value to the IE register of the MCS-51 family of microcontrollers.
static void	**intrPriority**(int value) Interrupt Priority methods This method writes an integer value to the IP register of the MCS-51 family of microcontrollers.
static int	**readConstant**(int value, int rConAddress) Read constant method This method reads a value from memory.
static int	**readExternal**(int rdExAddress) External memory method This method reads a value from the external address.
static int	**readPort_0**() This method reads from port 0 of the MCS-51 family of microcontrollers.
static int	**readPort_0Bit**(int bit) This method reads a specifies bit from port 0 of the MCS-51 family of microcontrollers.
static int	**readPort_1**() This method reads from port 1 of the MCS-51 family of microcontrollers.
static int	**readPort_1Bit**(int bit) This method reads a specifies bit from port 1 of the MCS-51 family of microcontrollers.
static int	**readPort_2**() This method reads from port 2 of the MCS-51 family of microcontrollers.
static int	**readPort_2Bit**(int bit) This method reads a specifies bit from port 2 of the MCS-51 family of microcontrollers.
static int	**readPort_3**() This method reads from port 3 of the MCS-51 family of microcontrollers.
static int	**readPort_3Bit**(int bit) This method reads a specifies bit from port 3 of the MCS-51 family of microcontrollers.

static int	**readSbuf**() SBUF method This method reads a value from the SBUF register of the MCS-51 family of microcontrollers.
static int	**readScon_RB8**() This method reads the RB8 bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_REN**() This method reads the REN bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_RI**() This method reads the RI bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_SM0**() This method reads the SM0 bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_SM1**() This method reads the SM1 bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_SM2**() This method reads the SM2 bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_TB8**() This method reads the TB8 bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readScon_TI**() This method reads the TI bit of the SCON register of the MCS-51 family of microcontrollers.
static int	**readTcon_IE0**() This method reads TCON bit IE0 of the MCS-51 family of microcontrollers.
static int	**readTcon_IE1**() This method reads TCON bit IE1 of the MCS-51 family of microcontrollers.
static int	**readTcon_IT0**() This method reads TCON bit IT0 of the MCS-51 family of microcontrollers.
static int	**readTcon_IT1**() This method reads TCON bit IT1 of the MCS-51 family of microcontrollers.
static int	**readTcon_TF0**() This method reads TCON bit TF0 of the MCS-51 family of microcontrollers.

static int	**readTcon_TF1**() This method reads TCON bit TF1 of the MCS-51 family of microcontrollers.
static int	**readTcon_TR0**() This method reads TCON bit TR0 of the MCS-51 family of microcontrollers.
static int	**readTcon_TR1**() This method reads TCON bit TR1 of the MCS-51 family of microcontrollers.
static void	**scon**(int value) Serial Port mode method This method writes a value to the SCON register of the MCS-51 family of microcontrollers.
static void	**setIE_EA**() This method sets the IE enable/disable bit of the IE register of the MCS-51 family of microcontrollers.
static void	**setIE_ES**() This method sets/enables the serial port interrupt of the MCS-51 family of microcontrollers.
static void	**setIE_ET0**() This method sets/enables the timer 0 overflow of the MCS-51 family of microcontrollers.
static void	**setIE_ET1**() This method sets/enables the timer 1 overflow of the MCS-51 family of microcontrollers.
static void	**setIE_ET2**() This method enables timer 2 overflow of the IE register of the MCS-51 family of microcontrollers.
static void	**setIE_EX0**() This method sets/enables the external interrupt 0 of the MCS-51 family of microcontrollers.
static void	**setIE_EX1**() This method sets/enables the external interrupt of the MCS-51 family of microcontrollers.
static void	**setIE_UNDEFINED**() This bit of the IE register is not defined..
static void	**setIP_PS**() This method sets/enables the PS bit of the IP register of the MCS-51 family of microcontrollers.
static void	**setIP_PT0**() This method sets/enables the PT0 bit of the IP register of the MCS-51 family of microcontrollers.

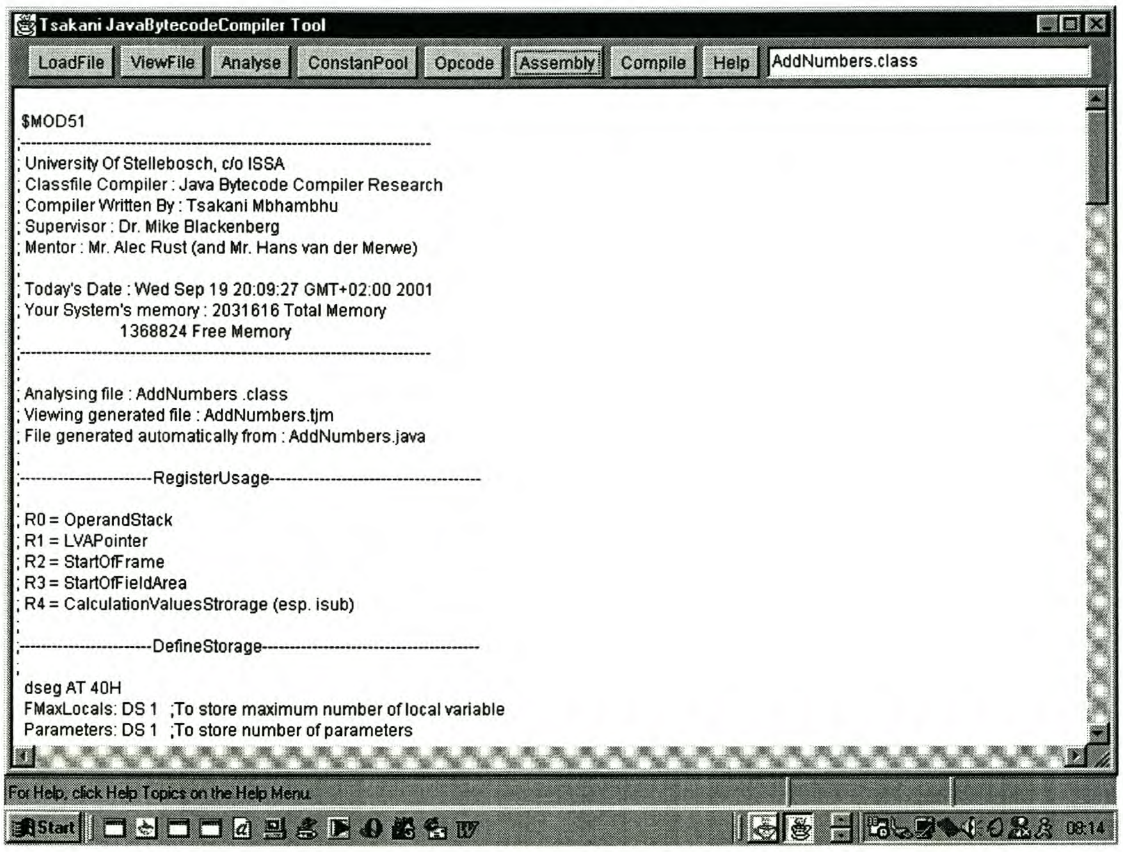
`static void`	**setIP_PT1**() This method sets/enables the PT1 bit of the IP register of the MCS-51 family of microcontrollers.
`static void`	**setIP_PT2**() This method sets/enables the PT2 bit of the IP register of the MCS-51 family of microcontrollers.
`static void`	**setIP_PX0**() This method sets/enables the PX0 bit of the IP register of the MCS-51 family of microcontrollers.
`static void`	**setIP_PX1**() This method sets/enables the PX1 bit of the IP register of the MCS-51 family of microcontrollers.
`static void`	**setIP_UNDEFINED1**() Undefined IP bit
`static void`	**setIP_UNDEFINED2**() Undefined IP bit
`static void`	**setPort_0Bit**(int bit) This method sets a specifies bit of port 0 of the MCS-51 family of microcontrollers.
`static void`	**setPort_1Bit**(int bit) This method sets a specifies bit of port 1 of the MCS-51 family of microcontrollers.
`static void`	**setPort_2Bit**(int bit) This method sets a specifies bit of port 2 of the MCS-51 family of microcontrollers.
`static void`	**setPort_3Bit**(int bit) This method sets a specifies bit of port 3 of the MCS-51 family of microcontrollers.
`static void`	**setScon_RB8**() This method sets/enables the RB8 bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	**setScon_REN**() This method sets/enables the REN bit of the SCON register of the MCS-51 family of microcontrollers.
`static void`	**setScon_RI**() This method sets/enables the RI bit of the SCON register of the MCS-51 family of microcontrollers.

static void	**setScon_SM0**() This method sets/enables the SM0 bit of the SCON register of the MCS-51 family of microcontrollers.
static void	**setScon_SM1**() This method sets/enables the SM1 bit of the SCON register of the MCS-51 family of microcontrollers.
static void	**setScon_SM2**() This method sets/enables the SM2 bit of the SCON register of the MCS-51 family of microcontrollers.
static void	**setScon_TB8**() This method sets/enables the TB8 bit of the SCON register of the MCS-51 family of microcontrollers.
static void	**setScon_TI**() This method sets/enables the TI bit of the SCON register of the MCS-51 family of microcontrollers.
static void	**setTcon_IE0**() This method sets TCON bit IE0 of the MCS-51 family of microcontrollers.
static void	**setTcon_IE1**() This method sets TCON bit IE1 of the MCS-51 family of microcontrollers.
static void	**setTcon_IT0**() This method sets TCON bit IT0 of the MCS-51 family of microcontrollers.
static void	**setTcon_IT1**() This method sets TCON bit IT1 of the MCS-51 family of microcontrollers.
static void	**setTcon_TF0**() Timer 0 overflow flag It is set by hardware when the time/counter overflows This method sets TCON bit TF0 of the MCS-51 family of microcontrollers.
static void	**setTcon_TF1**() Timer 1 overflow flag It is set by hardware when the time/counter overflows This method sets TCON bit TF1 of the MCS-51 family of microcontrollers.
static void	**setTcon_TR0**() Timer 0 control bit It is set/cleared by software to turn the time/counter on/off This method sets TCON bit TR0 of the MCS-51 family of microcontrollers.
static void	**setTcon_TR1**() Timer 1 control bit It is set/cleared by software to turn the time/counter on/off This method sets TCON bit TR1 of the MCS-51 family of microcontrollers.
static void	**writeExternal**(int value, int wrExAddress) External memory method This method writes a value to external address.

static void	**writePcon**(int value) PCON method This method writes an integer value to the PCON register of the MCS-51 family of microcontrollers.
static void	**writePort_0**(int value) This method writes to port 0 of the MCS-51 family of microcontrollers.
static void	**writePort_1**(int value) This method writes to port 1 of the MCS-51 family of microcontrollers.
static void	**writePort_2**(int value) This method writes to port 2 of the MCS-51 family of microcontrollers.
static void	**writePort_3**(int value) This method writes to port 3 of the MCS-51 family of microcontrollers.
static void	**writeSbuf**(int value) SBUF method This method writes an integer value to the SBUF register of the MCS-51 family of microcontrollers.
static void	**writeTH0**(int value) This method writes a value to the TH0 register of the MCS-51 family of microcontrollers.
static void	**writeTH1**(int value) This method writes a value to the TH1 register of the MCS-51 family of microcontrollers.
static void	**writeTimerControl**(int value) This is the timer control (TCON) register method of the MCS-51 family of microcontrollers.
static void	**writeTimerMode**(int value) This is the timer mode (TMOD) register method of the MCS-51 family of microcontrollers.
static void	**writeTL0**(int value) This method writes a value to the TL0 register of the MCS-51 family of microcontrollers.
static void	**writeTL1**(int value) This method writes a value to the TL1 register of the MCS-51 family of microcontrollers.

D3 Java Bytecode Compiler Tool

1. Compiler Outlook



```
$MOD51
;--------------------------------------------------------------------------
; University Of Stellebosch, c/o ISSA
; Classfile Compiler : Java Bytecode Compiler Research
; Compiler Written By : Tsakani Mbhambhu
; Supervisor : Dr. Mike Blackenberg
; Mentor : Mr. Alec Rust (and Mr. Hans van der Merwe)
;
; Today's Date : Wed Sep 19 20:09:27 GMT+02:00 2001
; Your System's memory : 2031616 Total Memory
;                  1368824 Free Memory
;--------------------------------------------------------------------------
;
; Analysing file : AddNumbers .class
; Viewing generated file : AddNumbers.tjm
; File generated automatically from : AddNumbers.java
;
;-----------------------RegisterUsage-----------------------------------
;
; R0 = OperandStack
; R1 = LVAPointer
; R2 = StartOfFrame
; R3 = StartOfFieldArea
; R4 = CalculationValuesStrorage (esp. isub)
;
;-----------------------DefineStorage-----------------------------------
;
 dseg AT 40H
 FMaxLocals: DS 1   ;To store maximum number of local variable
 Parameters: DS 1   ;To store number of parameters
```

2. Button Usage

LoadFile	-	To load the file.
ViewFile	-	To view the loaded file
Analyse	-	To analyse class file structures of the loaded class file
ConstantPool	-	To analyse the constant pool of the loaded class file
Opcode	-	To analyse the opcodes of methods of the loaded classfile
Assembly	-	To generate the assembly for the loaded class file
Compile	-	Generates and compiles assembly into hex file for the loaded class file.

Appendix E

Program Listing

E1 Irrigation Controller Program

```
public class IrgCtrl2
{
	static int countOne = 0, countTwo = 0, sec = 0,
			min = 0, hrs = 0, rsec = 0, rmin = 0,
			rhrs = 0, lsec = 0, lmin = 0,
			lhrs = 0, srmin = 0, slmin = 0,
			srhrs = 0, slhrs = 0, slsec = 0,
			srsec = 0, checkTI = 0, checkRI = 0,
			data = 0, shift = 0;

	static boolean dispFlag = false, cFlag = false,
				tFlag = false, sFlag = false,
				time1s = false, dlayFlag = false,
				loop = true, bFlag = false;

	//Main Program
	public static void main (String[] args)
	{
		//Initialize
		IO_8051.writePcon(0);			//SMOD = 0
		IO_8051.writeTimerMode(0x22);		//Timer 0 and 2 in auto reload mode
		IO_8051.writeTH0(0x00);			//Count down to 255
		IO_8051.writeTH1(-3);			//9600 baud @ 11.0592 MHz
		IO_8051.scon(0x50);			//8-bit UART mode, receiver on
		IO_8051.setTcon_TR0();			//Start the timer0
		IO_8051.setTcon_TR1();			//Start the timer1
		IO_8051.intrEnable(0x92);		//Enable the serial port, timer0
							//and timer1 interrupts

		//initialise port 1
		IO_8051.writePort_1(0xFF);

		//Do this always
		while (loop == true)
		{
			updateTime();
			display();
			if (sFlag == true)
			{
				//dispset();
				checkTime();
			}
			if (cFlag == true)
			{
				setclock();
				display();
				cFlag = false;
			}
//			if (bFlag == true)
//			{
```

```
//                              dispset();
//                              bFlag = false;
//                      }
                }
        }

        public static void checkTime()
        {
                if (((slmin - 48) == lmin) && ((srmin - 48) == rmin) &&
                        ((slhrs - 48) == lhrs) && ((srhrs - 48) == rhrs))
                {
                        IO_8051.clearPort_0Bit(3);
                        delay1s();
                        delay1s();
                        delay1s();
                        IO_8051.setPort_0Bit(3);
                        sFlag = false;
                }
        }

        //Update hrs, min and sec
        public static void updateTime()
        {
                time1s = false;
                while (time1s == false)
                {
                }
                //update minutes
                if (sec == 60)
                {
                        sec = 0;
                        min++;
                }
                //update hrs
                if (min == 60)
                {
                        min = 0;
                        hrs++;
                }
                //reset hrs to zero
                if (hrs == 24)
                {
                        hrs = 0;
                }
                //calculate lsec and rsec
                lsec = ((sec/10));
                rsec = ((sec - (lsec*10)));

                //calculate lmin and rmin
                lmin = ((min/10));
                rmin = ((min - (lmin*10)));

                lhrs = ((hrs/10));
                rhrs = ((hrs - (lhrs*10)));
        }

        //set the clock
        public static void setclock()
        {
                lsec = (slsec - 48);
                rsec = (srsec - 48);
                lmin = (slmin - 48);
                rmin = (srmin - 48);
                lhrs = (slhrs - 48);
                rhrs = (srhrs - 48);
                sec = ((lsec*10) + rsec);
                min = ((lmin*10) + rmin);
                hrs = ((lhrs*10) + rhrs);
        }
```

```
//display standby
public static void display()
{
		//clear display
		clrDisp();

		//display hours
		IO_8051.writePort_2((lhrs + 48)|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);

		delDisp();
		IO_8051.writePort_2((rhrs + 48)|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
		IO_8051.writePort_2(':'|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();

		//display minutes
		IO_8051.writePort_2((lmin + 48)|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
		IO_8051.writePort_2((rmin + 48)|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
		IO_8051.writePort_2(':'|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();

		//display seconds
		IO_8051.writePort_2(((lsec + 48)|0x80));
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
		IO_8051.writePort_2(((rsec + 48)|0x80));
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
		IO_8051.writePort_2(((' ')|0x80));
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
}

//display standby
public static void dispset()
{
		//clear display
		clrDisp();

		//display hours
		IO_8051.writePort_2((slhrs)|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);

		delDisp();
		IO_8051.writePort_2((srhrs)|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
		IO_8051.writePort_2(':'|0x80);
		IO_8051.clearPort_2Bit(7);
		IO_8051.setPort_2Bit(7);
		delDisp();
```

```
                //display minutes
                IO_8051.writePort_2((slmin)|0x80);
                IO_8051.clearPort_2Bit(7);
                IO_8051.setPort_2Bit(7);
                delDisp();
                IO_8051.writePort_2((srmin)|0x80);
                IO_8051.clearPort_2Bit(7);
                IO_8051.setPort_2Bit(7);
                delDisp();
                IO_8051.writePort_2(':'|0x80);
                IO_8051.clearPort_2Bit(7);
                IO_8051.setPort_2Bit(7);
                delDisp();

                //display seconds
                IO_8051.writePort_2(((slsec)|0x80));
                IO_8051.clearPort_2Bit(7);
                IO_8051.setPort_2Bit(7);
                delDisp();
                IO_8051.writePort_2(((srsec)|0x80));
                IO_8051.clearPort_2Bit(7);
                IO_8051.setPort_2Bit(7);
                delDisp();
                IO_8051.writePort_2(((' ')|0x80));
                IO_8051.clearPort_2Bit(7);
                IO_8051.setPort_2Bit(7);
                delay1s();
}

//clear display
public static void clrDisp()
{
        IO_8051.clearPort_1Bit(7);
        IO_8051.writePort_2((0x01)|0x80);
        IO_8051.clearPort_2Bit(7);
        delDisp();
        IO_8051.setPort_2Bit(7);
        IO_8051.setPort_1Bit(7);
}

//delay display method (~3ms)
public static void delDisp()
{
        dispFlag = false;
        while (dispFlag == false)
        {
        }
}

//delay method (~1s)
public static void delay1s()
{
        dlayFlag = false;
        while (dlayFlag == false)
        {
        }
}

//timer 0 interrupt procedure
public static void intrTimer0()
{
        countOne ++;
        if (countOne == 12)
        {
                dispFlag = true;
        }

        if (countOne == 180)
        {
                countOne = 0;
                countTwo++;
                }

                //making roughly a second
                if (countTwo == 20)
                {
                        sec++;
                        rsec++;
                        time1s = true;
                        dlayFlag = true;
                        countTwo = 0;
                }
        }

        //Serial Port interrupt procedure
        public final static void intrSerialPort()
        {
                checkRI = IO_8051.readScon_RI();        //Returns 1 if RI set
                checkTI = IO_8051.readScon_TI();        //Returns 1 if TI set

                if (checkTI == 1)
                {
                        IO_8051.clearScon_TI();         //Clear TI
                }

                if (checkRI == 1)
                {
                        if (tFlag == true)
                        {
                                slhrs = srhrs;
                                srhrs = slmin;
                                slmin = srmin;
                                srmin = slsec;
                                slsec = srsec;
                                srsec = data;
                                shift++;
                                if (shift > 6)
                                {
                                        tFlag = false;
                                }
                        }
                        IO_8051.clearScon_RI();                 //Clear RI
                        data = IO_8051.readSbuf();              //Read receive SBUF
                        IO_8051.writeSbuf(data);                //Transmit back
                        if (data == 't')
                        {
                                shift = 0;
                                tFlag = true;
                        }
                        if (data == 's')
                        {
                                sFlag = true;
                        }
                        if (data == 'c')
                        {
                                cFlag = true;
                        }
                        if (data == 'b')
                        {
                                bFlag = true;
                        }
                }
        }
}
```

E2 Implementation Example Output

```
$MOD51
;------------------------RegisterUsage----------------------------------
;
; R0 = OperandStack
; R1 = LVAPointer
; R2 = StartOfFrame
; R3 = StartOfFieldArea
; R4 = CalculationValuesStrorage (esp. isub)
;
;------------------------DefineStorage----------------------------------
;
dseg AT 50H
FMaxLocals: DS 1        ;To store maximum number of local variable
I0MaxLocals: DS 1       ;To store maximum number of Interrupt local variable
FMaxStack: DS 1         ;To store maximum size of stack
Parameters: DS 1        ;To store number of parameters
CompVal: DS 1           ;To store values for comparison purposes
PortVal: DS 1           ;To store the value to be written to port
PrtVal: DS 1            ;To store the value for bit to be written to port
;
; ------------------------CodeSegment-----------------------------------------
;
cseg
ORG 000H
SJMP Start
;
; ------------------------InterrruptVectors-------------------------------------
;
ORG 0003H
LJMP intrExternal0

ORG 000BH
LJMP intrTimer0

ORG 0013H
LJMP intrExternal1

ORG 001BH
LJMP intrTimer1

ORG 0023H
LJMP intrSerialPort

ORG 002BH
LJMP intrTimer2

; ------------------------MainProgram------------------------------------------
;
Start:
mov SP, #30H            ;Initialising the stack
mov R3, #60H            ;Initialising the start of fields area
mov A, R3

;initializing field 1 at position 0 to zero
mov A, R3
mov R1, A
add A, #0
mov R1, A
mov A, #0
mov @R1, A

;initializing field 2 at position 1 to zero
mov A, R3
mov R1, A
add A, #1
mov R1, A
mov A, #0
```

```
    mov @R1, A

    ;initializing the start of the Java Stack (Frame)
    mov A, R3               ;Start of the Fields Area
    add A, #3               ;Reserving space for fields
    mov R2, A               ;LVA begins at Field Area + Reserved Space

    ;initializing the start of the OS
    mov A, R2               ;Start of Frame above Field Area
    mov R1, A               ;Start of LVA
    add A, #1               ;Going 1 above LVA
    mov R0, A               ;Start of Operand

    ;method for assigning field values in the Fields Area
    call Exampleclinit

    Examplemain:
    ;
    mov A, R2
    mov FMaxLocals, #6
    mov FMaxStack, #3
    add A, FMaxLocals
    mov R0, A

    Opcode6: ;bipush
    ;----------
    mov @R0, #15
    inc R0

    Opcode8: ;istore_1
    ;--------
    dec R0
    mov A,R2
    mov R1,A
    add A, #1
    mov R1, A
  mov A, @R0
  mov @R1, A

  Opcode9: ;bipush
  ;----------
  mov @R0, #20
  inc R0

  Opcode11: ;istore_2
  ;--------
  dec R0
  mov A,R2
  mov R1,A
  add A, #2
  mov R1, A
  mov A, @R0
  mov @R1, A

  Opcode12: ;iconst_0
  ;-------
  mov @R0, #0
  inc R0

  Opcode13: ;istore_3
    ;--------
  dec R0
  mov A,R2
  mov R1,A
  add A, #3
  mov R1, A
  mov A, @R0
  mov @R1, A

  Opcode14: ;iload_1
  ;-------
```

```
mov A, R2
mov R1,A
add A, #1
mov R1, A
mov A, @R1
mov @R0, A
  inc R0

Opcode15: ;iload_2
;-------
mov A, R2
mov R1,A
add A, #2
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode16: ;ior
;------
dec R0
mov A, @R0
dec R0
ORL A, @R0
mov @R0, A
inc R0

Opcode17: ;istore_3
  ;---------
dec R0
mov A,R2
mov R1,A
add A, #3
mov R1, A
mov A, @R0
mov @R1, A

Opcode18: ;getstatic
;-------
mov A, R3
add A, #1
mov R1, A
mov A, @R1
mov @R0, A
  inc R0

Opcode21: ;iload_1
;-------
mov A, R2
mov R1,A
add A, #1
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode22: ;invokestatic...
;----------------
push 2
push 0
push Parameters
push FMaxLocals
mov Parameters, #2
mov A, R0
subb A, Parameters
mov R2, A
mov A, R2
add A, Parameters
mov R0, A
call Exampleaddd
pop FMaxLocals
pop Parameters
pop 0
pop 2

Opcode25: ;pop
;-------
dec R0
mov A, @R0

Opcode26: ;iload_2
;-------
mov A, R2
mov R1,A
add A, #2
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode27: ;iload_3
;-------
mov A, R2
mov R1,A
add A, #3
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode28: ;invokestatic...
;---------------
push 2
push 0
push Parameters
push FMaxLocals
mov Parameters, #2
mov A, R0
subb A, Parameters
mov R2, A
mov A, R2
add A, Parameters
mov R0, A
call Exampleaddd
pop FMaxLocals
pop Parameters
pop 0
pop 2

Opcode31: ;pop
;-------
dec R0
mov A, @R0

Opcode32: ;iload_2
;-------
mov A, R2
mov R1,A
add A, #2
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode33: ;iload_1
;-------
mov A, R2
mov R1,A
add A, #1
mov R1, A
mov A, @R1
mov @R0, A
```

```
inc R0

Opcode34: ;if_icmpge
;-------
dec R0
mov CompVal, @R0
dec R0
mov A, @R0
cjne  A, CompVal, CmpgeJmpc34
JMP CmpgeJmp34
CmpgeJmpc34: JNC CmpgeJmp34
JMP CmpgeEnd34
CmpgeJmp34: JMP Opcode39
CmpgeEnd34: JMP Opcode37

Opcode37: ;iload_2
;-------
mov A, R2
mov R1,A
add A, #2
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode38: ;istore_1
;--------
dec R0
mov A,R2
mov R1,A
add A, #1
mov R1, A
mov A, @R0
mov @R1, A

Opcode39:
;---------------
call IO_8051readPort_1

Opcode42: ;istore...
;----------
dec R0
mov A,R2
mov R1,A
add A, #4
mov R1, A
mov A, @R0
mov @R1, A

Opcode44: ;iload
;-------
mov A, R2
mov R1,A
add A, #4
mov R1, A
mov A, @R1
mov @R0, A
inc R0

Opcode46:
;---------------
call IO_8051writePort_3

Opcode49: ;return
;-------
mov A, R2
ret
JMP Exit

IO_8051readPort_1:
mov A, P1
```

```
Opcode58: ;bipush
;----------
mov @R0, #10
inc R0

Opcode60: ;putstatic
;-------
dec R0
mov A, R3
mov R1, A
add A, #1
mov R1, A
mov A, @R0
mov @R1, A

Opcode63: ;return
;-------
mov A, R2
ret

  ;  Default External 0 Interrupt Procedure
  ;  -------------------------------------

intrExternal0:
reti

  ;  Default Time 0 Interrupt Procedure
  ;  ---------------------------------

intrTimer0:
reti

  ;  Default External 1 Interrupt Procedure
  ;  -------------------------------------

intrExternal1:
reti

  ;  Default Timer 1 Interrupt Procedure
  ;  ----------------------------------

intrTimer1:
reti

  ;  Default Serial Port Interrupt Procedure
  ;  -------------------------------------

intrSerialPort:
reti

  ;  Default Timer2 Interrupt Procedure
  ;  --------------------------------

intrTimer2:
reti

  ;
   Exit:
     END
```

E3 Parameter Passing Program

```
class Parameters
{

        public static void main (String[] args)
        {
                int x = 9;
                int y = 7;
                IO_8051.writePort_0(addd(x,y));
        }


        public static int addd(int first, int second)
        {
                return (first + second);
        }
}
```

E4 Java Serial Program

```
public class SerPort5
{
        static boolean wag = true;
        static boolean yima = true;

        public static void main (String[] args)
        {
                IO_8051.writePcon(0);                   //SMOD = 0
                IO_8051.writeTimerMode(32);             //Timer 1 in auto reload mode
                IO_8051.writeTH1(-3);                   //9600 baud @ 11.0592 MHz
                IO_8051.scon(80);                       //8-bit UART mode, receiver on
                IO_8051.setTcon_TR1();                  //Start the timer
                IO_8051.intrEnable(144);                //Enable the serial port interrupts

                //Wait
                while (wag = true)
                {
                        IO_8051.setPort_1Bit(2);
                        IO_8051.clearPort_1Bit(2);
                }

        }

        //Serial Port interrupt procedure
        public final static void intrSerialPort()
        {
                int data = 0;
                int checkTI;
                int checkRI;

                checkRI = IO_8051.readScon_RI();                //Returns 1 if RI set
                checkTI = IO_8051.readScon_TI();                //Returns 1 if TI set

                if (checkTI == 1)
                {
                        IO_8051.clearScon_TI();                 //Clear TI
                        //IO_8051.setPort_1Bit(2);              //Set P1.2
                        IO_8051.clearPort_1Bit(2);
                }

                if (checkRI == 1)
                {
                        IO_8051.clearPort_1Bit(2);              //Clear P1.2
                        IO_8051.clearScon_RI();                         //Clear RI
                        data = IO_8051.readSbuf();              //Read receive SBUF
                        data++;                                 //Increment data read
                        IO_8051.writeSbuf(data);                //Write incremented data to
                                                                //transmit SBUF
                        IO_8051.clearPort_1Bit(3);
                }
        }
}
```

E5 C Serial program

```
#pragma DEBUG OBJECTEXTEND CODE // pragma lines can contain state C51
#include <stdio.h>              // declarations for I/O functions
#include <ADuC812.h>            // 8052 & ADuC812 predefined symbols



unsigned char c = 0;
/*************************************/
/* Main Function of the Program
/*************************************/
void main (void){
  /* CONFIGURE UART */
  SCON = 0x50 ;                 // 8bit, noparity, 1stopbit
  TMOD = 0x20 ;                 //
  TH1 = 0xFD ;                  // ..for 9600baud..
  TR1 = 1 ;                     // ..(assuming 11.0592MHz crystal)

   IE = 0xD0;

  while(1){
    }

}

/********************************************/
/* SERIAL INTERRUPT SERVICE ROUTINE
/********************************************/
void serial_int(void) interrupt 4 {

   if (RI) {
      RI=0;
      c=SBUF;
      c++;
     SBUF=c;
   }

   if(TI){
      TI=0;

   }
}
```

E6 Assembly Serial Program

```
$MOD 51
;Constants
;PCON   equ     87H                     ;"Power Control Register"
;
;===============================================================================
;
;Data variables
                dseg
                org 30H
CharCnt:        ds 1                    ;count of characters received
;
;===============================================================================
;
;Program code
                cseg
                org 0000H
                jmp Begin               ;Jump to Begin on power reset
                org 0023H
                jmp Receive             ;Serial port interrupt
;
;Main program
;
                org 0030H
Begin:          mov SP,#70H             ;Inialize stack
                mov CharCnt,#0          ;Set CharCnt to zero
                mov PCON,#0             ;SMOD=0
                mov TMOD,#20H           ;Timer 1 in auto reload mode
                mov TH1,#-3             ;9600 baud @ 11.0592 MHz
                mov SCON,#50H           ;8-bit UART mode, receiver on
                setb TR1                ;start the timer
                mov IE,#90H             ;enable serialport interrupts
WAG:            cpl P1.7                ;Toggle test line
                jmp WAG                 ;Wait for interrupts
;
;Serial port interrupt procedure
;
Receive:        clr P1.2                ;Switch LED on
                clr RI                  ;Clear receive flag
                mov A,SBUF              ;Read receive buffer
                inc CharCnt             ;Increment character count
                inc A                   ;Increment character to next ASCII value
                mov SBUF,A              ;Write send buffer
RX1:            jnb TI,RX1              ;Wait until send complete
                clr TI                  ;Clear send flag
                setb P1.2               ;Switch LED off
                reti
;
                end
```