Super Light Virtual Machine (SLVM) – Virtual machine for Microcontrollers with Highly Reduced Resources

S. Rusinov, R. Ilarionov

Key Words: *Virtual machine; SLVM; small devices VM.*

Abstract. *This article presents a concept of a virtual machine able to run on microcontrollers with very small resources, such as PIC18 family of Microchip. The main features of the VM are real-time processing, multi-tasking and memory management which meets the needs of small embedded projects.*

1. Introduction

Nowadays virtual machines technology is becoming more and more popular. This is possible since the computer field is developing very fast and the computers are powerful enough, so we do not have to think how to save computer resources as we used to do in the past. Now other features of software applications are more important, such as:

- Software has to be platform independent.
- The time for software development must be highly reduced.
- Software should be easily reused for different applications.
- Easy software maintenance.
- Software reliability and secure execution must be increased.

The virtual machines technology gives us an easy way to achieve these features. But all good news has its price. The virtual machine requires much more resources in comparison to the platform-dependent solutions. The software execution is slower. The JIT (Just-In-Time) compilation technique provides a solution for this issue by intermediate conversion of the byte code (JAVA) or IL (Intermediate Language – .NET Framework) to native instructions. This technology gives us an optimal code for each platform, but we still have a delay provoked by the compilation process just before code execution. There are several disadvantages caused by JIT:

- JIT requires extra resources for the intermediate code compilation.
- It can be used for Real-Time solutions only if the whole program is compiled before its execution.
- JIT compiler is not available for all platforms.

The virtual machines technology is not widely applied in embedded systems for two main reasons. The first is the limited platform resources and the second is that most of the embedded systems require strong real-time work. Actually there are several solutions for virtual machines implemented in multimedia devices as PDAs and smart phones. In most cases they are reduced versions of JVM or CLR. The variety of virtual machines for very small platforms is really small. Yes, there are some solutions, but in most cases they require at least a hundred Kbytes footprint and tens of Kbytes RAM. Another disadvantage is that in most cases they are not Real-Time. We believe one of the reasons why these solutions are not widely used is that their architecture is taken from platforms where much more resources are available and the application area does not require Real-Time processing.

Another obstacle to the use of JVM and CLR in the professional development of embedded software is that they are using high level object-oriented programming languages, such as C# and JAVA. Object-oriented techniques, such as inheritance of objects, templates and virtualization of objects, are not easily accepted in small embedded projects. Normally the most important goals of the embedded systems are high performance and reliability. The object-oriented programming languages provide an easy way for implementation of the programs, but at the expense of more resources, low performance and dynamic memory usage which is not preferred in the small embedded systems design (more often all required memory is reserved statically before running the program). For the concept described here the ANSI C language is chosen. It has several advantages. First, it is the most commonly used language for embedded projects, and programmers who know it can easily start to implement applications for the virtual machine. Second, a lot of embedded projects are already implemented in C and making them compliant with the SLVM requirements will not be difficult.

Memory management provided by JVM and CLR (usually it is HEAP based) is very flexible and convenient for platforms with huge resources, but for the embedded systems it is not suitable because:

- The memory becomes fragmented after prolonged work.
- Some services for reducing memory fragmenting and garbage collector are started by the virtual machine automatically. The execution of these services causes a delay in the execution of the user program (inadmissible for Real-Time projects).
- The time for execution of a part of the user program is not constant and it depends on the current memory state. At each iteration the program objects can be located in different places in the memory and, respectively, the access time to them will be different.

2. Virtual Machine for Small Microcontrollers

Here we will try to find a solution to the problems described above by introducing a new concept for virtual machines targeted for embedded platforms with very small resources. Our concept virtual machine is called SLVM (Super Light Virtual Machine). This task is a little bit of a challenge, taking into consideration the requirements of such platforms. Our goal is to avail ourselves of the advantages of virtual machines in the specific embedded field. The main objectives of our embedded virtual machine are listed below:

- Footprint less than 8Kb.
- Very low RAM usage from the VM kernel – less than 100 bytes.
- Real-Time processing.
- Multi-task support.
- Simplified and high performance memory management.
- The necessity for memory services, such as defragmentation and garbage collector, should be avoided.
- Cooperative work of native user program and programs started on the virtual machines should be possible.
- Development of an easy mechanism for implementation of VM API functions which can be used by the user applications.
- Processing of microcontroller interrupts.
- Support of shared memory between user VM programs and the native program loaded on the microcontroller.
- The virtual machine must be very easily customizable according to the available resources and the peripheral devices available on the microcontroller. This is very important since we must meet the requirement for very small footprint and RAM usage from the VM.
- Dynamic and static loading of the user programs. When static loading is chosen, the whole program is loaded on microcontroller RAM or EEPROM. Program execution from the RAM memory is the fastest, but we are limited to program space, since usually 1 or 2 Kb of RAM are available. Through the dynamic program loading, we are able to execute larger programs at the expense of slower execution, because the programs are loaded and executed particularly on the microcontroller RAM instruction by instruction or function by function.
- The virtual machine should be controlled by the native program loaded on the microcontroller (initialization and controlling VM tasks priority should be implemented in it).
- The VM should be developed in ANSI C in order to facilitate porting on different platforms.
- The program language used for the concept virtual machine is ANSI C. Most of the embedded projects are using this language and it will be easy for the embedded developers to start writing programs for the virtual machine.
- For translation of VM programs from ANSI C language into a SLVM-compatible byte code, a modified SDCC cross compiler is used.

3. SLVM Design

This concept is targeted mainly for 8 bit and 16 bit microcontrollers with minimum 8 Kb of Flash memories and minimum 512 Kb of RAM. Such kinds of devices, for example, are the Microchip microcontrollers from PIC18, PIC24, dsPIC families and Atmel microcontrollers from the AVR family. This concept can be easily applied also on 32 bit platforms with more or equal resources. The architecture of SLVM concept is given in *figure 1*.

It is clear that the byte code interpretation will be hundreds times slower in comparison with the performance which can be achieved if the program was implemented with native instructions. JIT technique is not applicable for two reasons. First, not all microcontrollers are able to execute instructions from the RAM, and second, JIT requires extra resources which are not available on such types of microcontrollers. So we have one choice – the tasks which require fast processing, like interrupts processing, communications, etc. to be implemented in the native program. If necessary, SLVM can control (at high level) these fast tasks by SLVM API functions. Loading of the VM tasks on the microcontroller is a responsibility of the native program too. SLVM provides several API functions for initial loading of the SLVM tasks. That is why the cooperative work of the native program and the VM programs is very important.

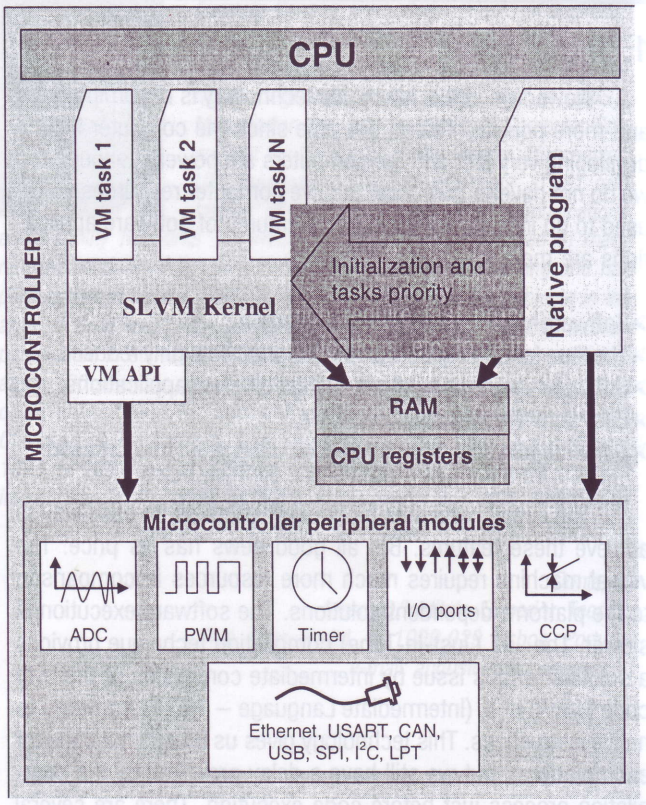


Figure 1. SLVM architecture

4. Memory Management

The memory management algorithm used in SLVM is very simple. Each VM task has its own program stack. Its size is specified statically during the compilation of the SLVM kernel. In the beginning of the program all global variables are defined. They are put in the task stack first and are removed after the program is over. In the beginning of each function all local variables are declared and when it is being called, they are put in the task stack. After the end of the function, all local variables

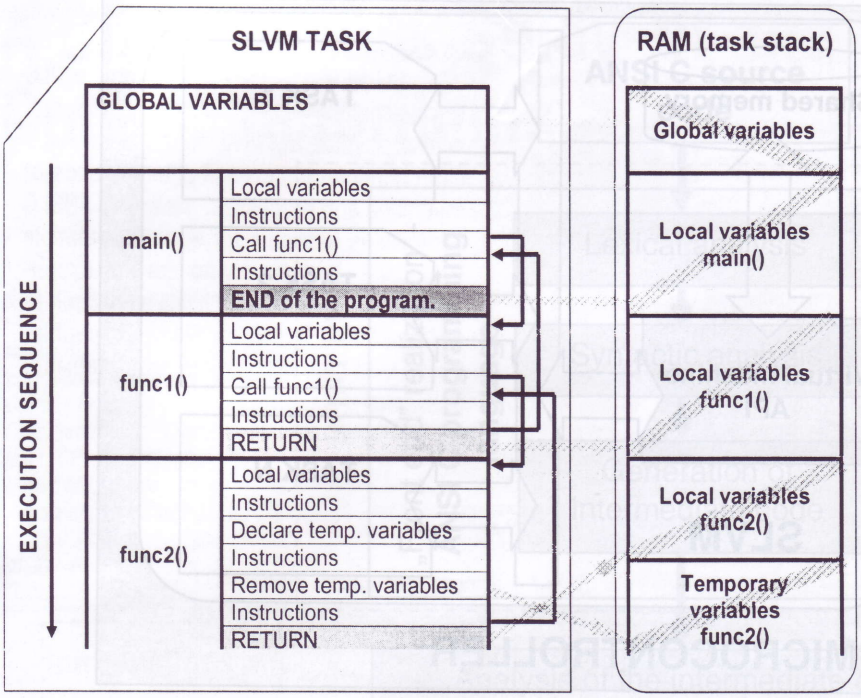


Figure 2. SLVM Memory management

declared for it are removed from the stack. Temporary variables can be specified inside the functions. When the code execution engine meets them, they will be put in the task stack too. They can be removed from the stack by a special instruction. After the end of the main function, all global variables are removed from the task stack.

Figure 2 shows how exactly the memory manager works with a simple program which includes three functions that are called consecutively one from another.

5. Interrupts Handling

Handling microcontroller interrupts is very important for embedded systems. SLVM provides a way of processing interrupts. Each VM task can be registered for interrupts by configuring a specific interrupts table. Each task has its own interrupt table. The configuration of the table can be done by the concrete task or by the native program using SLVM API functions. When an interrupt occurs, the processing routine from the native program will update all interrupt tables of all SLVM tasks which are registered for the current interrupt.

On the next execution of instruction from the task with signaled interrupt, an interrupt handling function will be called, specified in the corresponding interrupt table. Interrupts can be disabled or enabled by calling SLVM API functions. The interrupt processing functions from the SLVM tasks can read some data connected with the signaled interrupt, which is stored in the corresponding interrupt table. This value is put in the table by the native program when an interrupt is processed. This data can be a register value or a state depending on the interrupt type. The interrupts for all platforms should be standardized in order for the SLVM byte code to be platform independent.

Figure 3 shows the basic mechanism for processing interrupts for a single task of SLVM.

6. Shared Memory

The communication between the separate SLVM tasks and the native program is very important. That is why SLVM provides a RAM area which is accessible from all running processes on the platform. This area is named „Shared Memory". Its size is defined statically during SLVM kernel compilation. The access to this memory is performed using SLVM API functions. They can be called both from any running SLVM task and from the native program. It is possible for a part or the whole shared memory to be located at EEPROM if it is available on the platform. This will provide an opportunity for data to be kept after the platform power supply is switched off.

Figure 4 shows how the shared memory works.

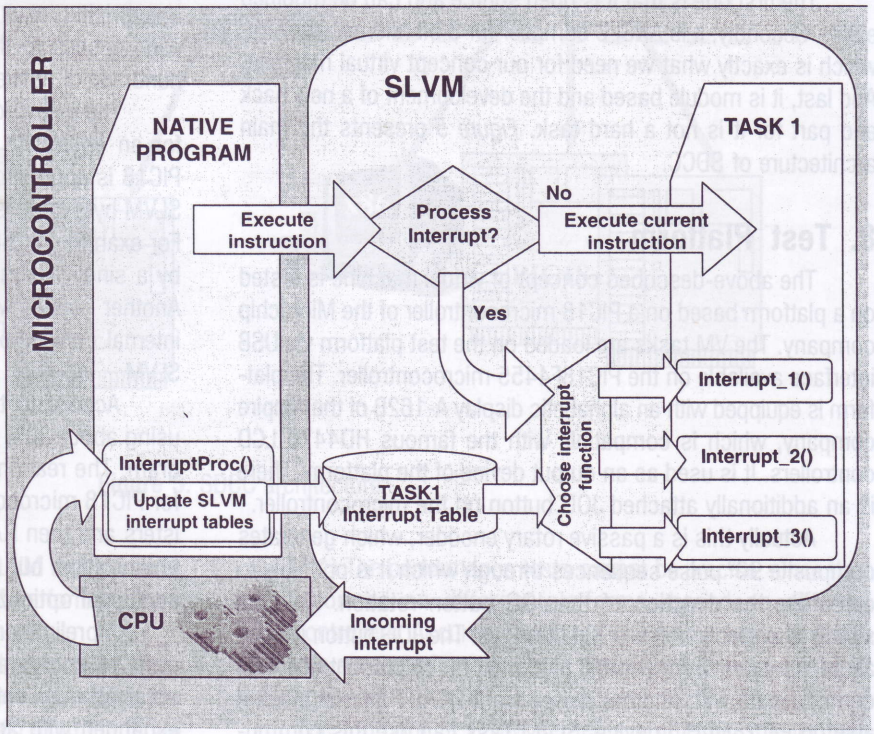


Figure 3. Handling CPU interrupts

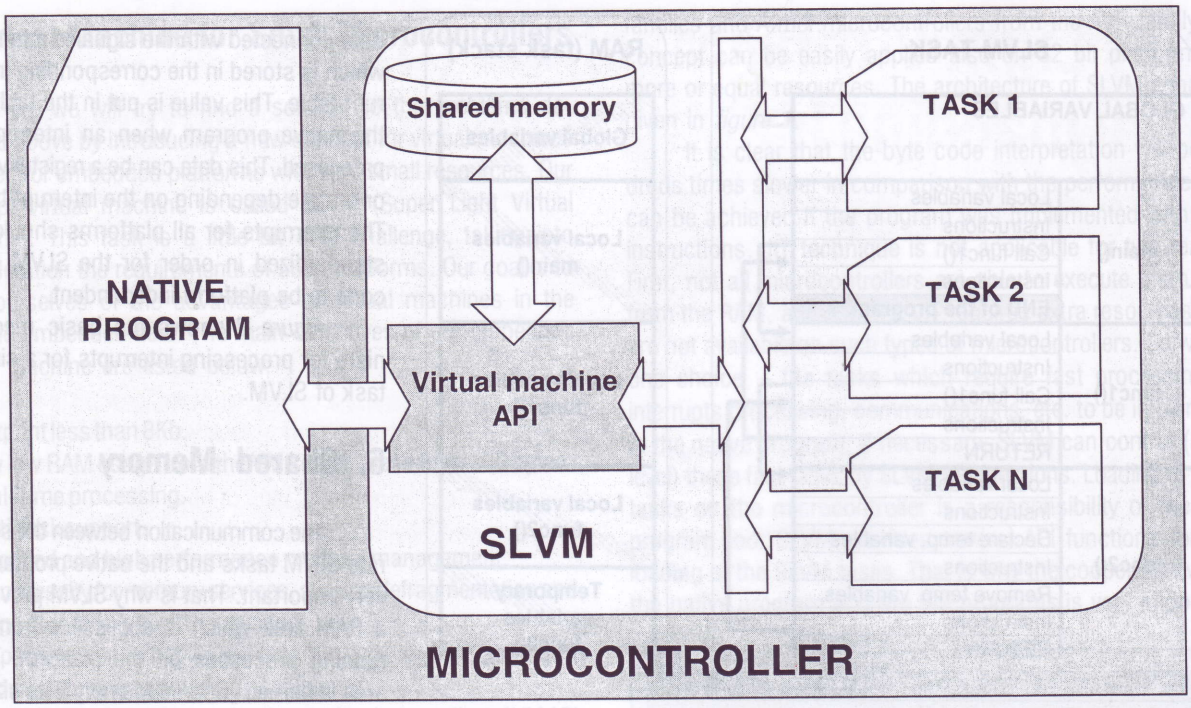


Figure 4. Using SLVM shared memory

7. Byte Code Generation

It is not useful to write user programs for the SLVM directly in byte code. That is why a modified version of the open source cross-compiler SDCC (Small Devices C Compiler) has been developed, which is able to generate SLVM compatible byte code. There are several reasons why this compiler has been chosen.

The first one is that it is open source and can be modified easily. Secondly, it is ANSI C compiler for 8 bit microcontrollers, which is exactly what we need for our concept virtual machine. And last, it is module based and the development of a new back end part for it is not a hard task. *Figure 5* presents the main architecture of SDCC.

8. Test Platform

The above-described concept of virtual machine is tested on a platform based on a PIC18 microcontroller of the Microchip company. The VM tasks are loaded on the test platform via USB interface available on the PIC18F4455 microcontroller. The platform is equipped with an alphabetic display A-162B of the Ampire company, which is compatible with the famous HD4470 LCD controllers. It is used as an output device of the platform. There is an additionally attached JOG button on the microcontroller.

Actually this is a passive rotary encoder, which generates composite 90^0 pulse sequences through which it is possible to determine the direction of the JOG button rotation. An axial switch is available on the JOG button too. The JOG button is used as an input device for our test platform. The microcontroller can communicate with another device via a USART interface available on PIC18F4455 device. SLVM Task can use this communication interface to send messages to another device (for example PC) by calling SLVM API function. The block diagram of the test platform is given in *figure 6*.

9. Results Obtained

Several tests have been performed on the specified tests platform based on a PIC18 microcontroller.

A performance test has shown that SLVM is approximately 600 times slower than the native code execution. This is not a surprising result, taking into consideration that, for example, each access to the program stack requires the execution of hundreds of native instructions.

The byte code size of SLVM compared to the native code for an equivalent program generated with a C compiler for PIC18 is approximately 25% smaller. This is normal since the SLVM byte code is more complex than the PIC18 instructions. For example, in SLVM the calling of a function is represented by a single instruction in which all parameters are specified. Another feature which reduces the byte code size is that the internal conversion of the operand types is supported by the SLVM.

According to the memory used (task stack), SLVM is using about 30% more memory that an equivalent native program. The reason for this issue is that the C compilers used for PIC18 microcontrollers are optimized to use first CPU registers and then RAM memory. SLVM cannot use such kind of optimization but in feature can be improved by introducing additional optimizations in SDCC.

A preliminary test was performed with SLVM platform in order to compare it with other virtual machines. The results obtained may not be accurate since in feature SLVM will be expanded with additional features and the indicators Performance, Footprint and RAM usage may become worse.

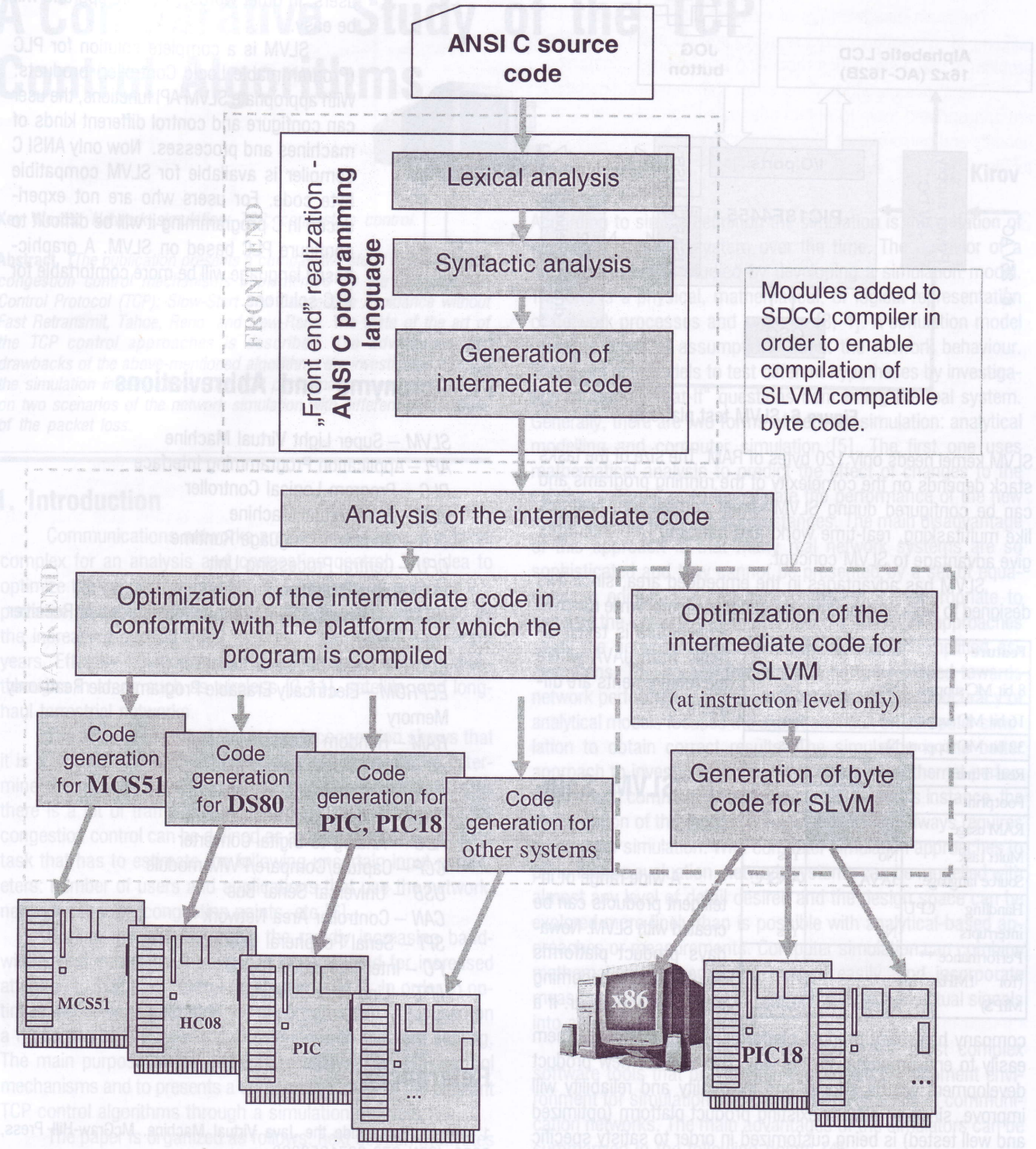


Figure 5. SDCC architecture

The table below is a comparison of SLVM and nanoVM virtual machines. The results for the nanoVM performance are extracted from the WEB site:

http://www.harbaum.org/till/nanovm/index.shtml.

(*) 300 bytes intended for task stack are included in the specified RAM usage. The actual memory used from the SLVM kernel is about 120 bytes.

(**) The performance tests cannot be precise, since they are performed on different platforms, based on different microcontrollers. Accurate results can be read only in case both concepts are tested on the same hardware with equivalent test programs.

The footprint of SLVM kernel is a little bit smaller than nanoVM. The differences are more evident in the RAM usage.

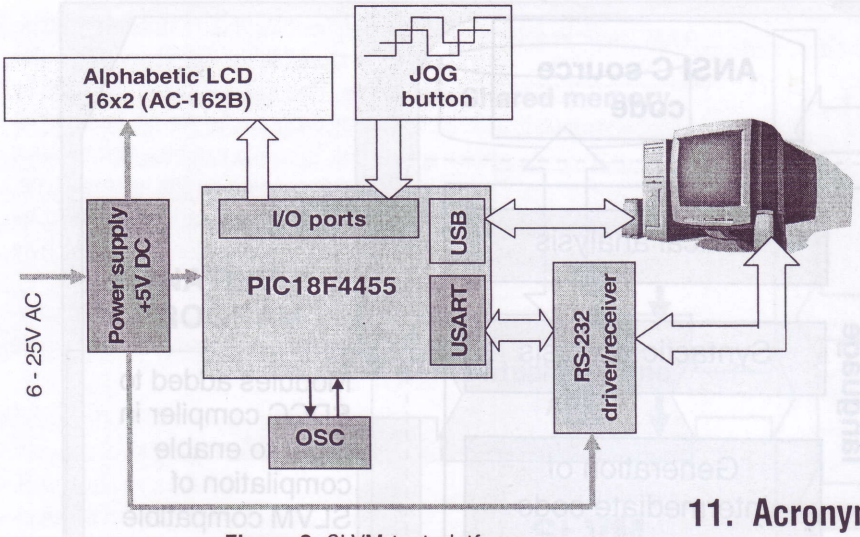


Figure 6. SLVM test platform

SLVM kernel needs only 120 bytes of RAM. The size of the tasks stack depends on the complexity of the running programs and can be configured during SLVM kernel compilation. Features like multitasking, real-time work, and handling CPU interrupts give advantage to SLVM concept.

SLVM has advantages in the embedded area, since it is designed to work on real-time embedded systems; while nanoVM architecture is taken directly from JAVA where the requirements are different.

Feature	nanoVM	SLVM
8 bit MC support	Yes	Yes
16 bit MC support	No	No
32 bit MC support	No	Yes
Real-Time	No	Yes
Footprint	8K	7,4K
RAM usage	768	420*
Multi task	No	Yes
Source language	JAVA	ANSI C
Handling CPU interrupts	No	Yes
Performance ** (for 1MHz in MIPS)	0,0025	0,0032

10. SLVM Solutions

A wide range of intelligent products can be created with SLVM. Nowadays product platforms solutions are becoming increasingly popular. If a company has ready product platforms, it can customize them easily to end products. In this way, the time for new product development will be shorter and its quality and reliability will improve, since an already existing product platform (optimized and well tested) is being customized in order to satisfy specific customer requirements. SLVM is very suitable for customization of product platforms.

For example, if a company is producing car alarm systems, and has a platform based on some microcontroller, the use of SLVM can provide a wide range of products with a different number of security zones, logic and peripheries by simply modifying SLVM tasks working at a higher level. So if you have well-designed product platforms, using SLVM can easily provide a wide range of products based on this platform. They will be more reliable because the same firmware will be loaded on them and only the SLVM tasks will be different. If necessary, the byte code of the SLVM task can be updated without much difficulty by the users. In other words, software upgrade will be easy.

SLVM is a complete solution for PLC (Programmable Logic Controller) products. With appropriate SLVM API functions, the user can configure and control different kinds of machines and processes. Now only ANSI C compiler is available for SLVM compatible byte code. For users who are not experienced in C programming it will be difficult to configure PLC based on SLVM. A graphic-based language will be more comfortable for PLC solutions.

11. Acronyms and Abbreviations

SLVM – Super Light Virtual Machine
API – Application Programming Interface
PLC – Program Logical Controller
JVM – JAVA Virtual Machine
CLR – Common Language Runtime
CPU – Central Processing Unit
VM – Virtual Machine
USART – Universal Synchronous Asynchronous Receiver Transmitter
SDCC – Small Device C Compiler
EEPROM – Electrically Erasable Programmable Read-only Memory
RAM – Random Access Memory
JIT – Just-in-time compilation
PDA – Personal Digital Assistant
PWM – Pulse-width modulation
ADC – Analog-to-Digital Converter
CCP – Capture/Compare/PWM module
USB – Universal Serial Bus
CAN – Controller Area Network
SPI – Serial Peripheral Interface
I²C – Inter-Integrated Circuit
LPT – Line Print Terminal

References

1. Veners, Bill. Inside the Java Virtual Machine. McGraw-Hill Press, 2000, ISBN 978-0071350938.
2. Blunden, Bill. Virtual Machines Design and Implemetation in C/C++. Wordware Publishing, 2002, ISBN 978-1556229039.
3. Smith Jim, Nair Ravi. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, 2005, ISBN 978-1558609105.

to be continued on page 24

- Reno: adds modification to Fast Recovery.
- New-Reno: enhanced Reno TCP using a modified version of Fast Recovery.

The comparative study of the above-mentioned algorithms is done. The advantages of the Reno and New Reno algorithms are proved. The simulation results show that when the probability for packet loss is high the New Reno one is the most effective because it has very low probability of retransmission timeouts.

The simulation results will help experts make well-informed decisions on how to manage an Ethernet network and fine-tune the network parameters.

References

1. Douglas, C. Internetworking with TCP/IP (2). Prentice-Hall, Inc., 1991.
2. Douglas, C. Internetworking with TCP/IP (3). Prentice-Hall, Inc., 1991.
3. Ewerlid, A. Reliable Communication over Wireless Links, in Nordic Radio Symp. (NRS). Sweden, Apr. 2001.
4. Fall, K., S. Floyd.Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. - *Computer Communication Review*, 26 (3), July 1996, 5-21.
5. Firoiu, V., M. Borden. A Study of Active Queue Management for Congestion Control.Proc. IEEE INFOCOM, March 2000.
6. Jacobson, V. Congestion Avoidance and Control. - *Computer Communication Review*, 18 (4), August 1988, 314-329.
7. Jacobson, V. Congestion Avoidance and Control, in Proceedings of SIGCOMM '88 Workshop. ACM SIGCOMM, ACM Press, Stanford, CA, 1988, 314-329.
8. Mo, J. and J. Walrand. Fair End-to-end Window-based Congestion Control. - *IEEE/ACM Trans. Networking*, 8, 5 (Oct. 2000), 556-567.
9. Morris, R. Scalable TCP Congestion Control. IEEE INFOCOM 2000, Tel Aviv.
10. Padhye, J., S. Floyd. On Inferring TCP Behavior. - *Computer Communications Review ACM-SIGCOMM*, 31, August 2001.
11. Schilke, A. TCP over Satellite Links. Seminar Broadband Networking Technology, TU Berlin, 1997.
12. Stevens, W. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, 1999, http://www.faqs.org/rfcs/rfc2001.html.
13. Wang, R., et all. TCP with Sender-side Intelligence to Handle Dynamic, Large, Leaky Pipes. - *IEEE Journal on Selected Areas in Communications*, 23 (2), 2005, 235-248.

Manuscript received on 14.02.2008

Georgi Kirov *received his M. S. in Computer Technologies from the Technical University of Sofia, Bulgaria. He obtained his PhD degree in the field of the Intelligent Technologies and Computer Networks from the Institute of Computer and Communication Systems at the Bulgarian Academy of Sciences. Dr. Kirov is currently a research fellow at the Department of Knowledge Based Control Systems, Institute of Control and System Researches of the Bulgarian Academy of Sciences (BAS) with publications in the fields of intelligent technologies in computer simulation, system researches and communication networks.*

Contacts:
Institute of Control and System Research,
Bulgarian Academy of Sciences,
Acad. G. Bonchev str., bl. 2, P.O.Box 79,
1113 Sofia
e-mail: kirov@icsr.bas.bg

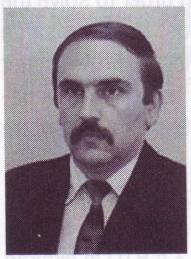
continuation from 16

4. Mac,Ronald. Writing Compilers and Interpreters. Wiley, 1996, ISBN 978-0471113539.
5. Louden, Kenneth C. Compiler Construction: Principles and Practice. Course Technology, 1997, ISBN 978-0534939724.
6. Aho, Alfred V. Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, 2006, ISBN 978-0321486813.
7. Muchnick, Steven. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997, ISBN: 978-1558603202.
8. Appel, Andrew W., Maia Ginsburg. Modern Compiler Implementation in C. Cambridge University Press, 2004, ISBN 978-0521607650.
9. Cooper,Keith, Linda Torczon. Engineering a Compiler. Morgan Kaufmann, 2003, ISBN 978-1558606982.
10. Craig, Iain D. Virtual Machines. Springer, 2005, ISBN 978-1852339692.
11. Lindholm Tim, Frank Yellin. Java(TM) Virtual Machine Specification.The (2nd Edition). Prentice Hall PTR, 1999, ISBN 978-0201432947.

Manuscript received on 13.12.2007

Svetozar Petrov Rusinov *born in 1980. Obtained M. Sc. Degree in Computer Systems and Technologies in 2007 at Technical University of Gabrovo. He worked at Senior R&D Engineer in Johnson Controls Inc. His main research interests comprise the area of software architecture, software engineering, software reliability and embedded and real time software systems.*

Contacts:
e-mail: svetozar_rusinov@mail.bg



Raycho Todorov Ilarionov *(born 1957) is Assoc. Prof. in department Computer systems and technologies of Technical University of Gabrovo. His research interests include Computer Peripherals, Multimedia Systems, Digital Circuit Devices & Microprocessors Devices.*

Contacts:
e-mail: ilar@tugab.bg