Research on the Architecture of Embedded Multitasking Operating Systems Based on Forth Virtual Machine: Postprint

Authors: Dai Hongbing, Zhou Yonglu, An Hongping, Mei Hao

Date: 2018-05-20T00:00:00+00:00

Abstract

Faced with increasingly complex embedded application requirements and numerous challenges in the current embedded operating system research field that urgently need to be addressed—including reconstruction, portability, maintenance, trustworthiness, multi-core, and many-core issues—this work adopts Forth virtual machine technology to explore key technologies for embedded operating systems based on the Forth virtual machine architecture, and proposes an efficient, compact embedded multitasking operating system architecture with favorable extensibility and portability characteristics. This architecture employs categorized memory mapping, Forth vector definitions, and separation of user variables to achieve code sharing and multitask management. Experimental results demonstrate that the Forth virtual machine architecture-based embedded operating system, while leveraging the inherent characteristics of the Forth system, reduces resource consumption and improves system flexibility and operational efficiency.

Full Text

0 Introduction

In the course of embedded system research, we have repeatedly encountered challenges with maintaining systems that require continuous operation. When bugs occur or upgrades become necessary for online embedded firmware, system updates and maintenance typically require shutting down the system for on-site intervention, or even removing equipment from the field for laboratory resolution, followed by reburning the updated firmware. Certain specialized application scenarios demand a fundamentally new type of real-time multitasking operating system that is reconfigurable, extensible, and supports online interaction. For instance, in uninterrupted remote space observation applications,

such an OS would enable the creation of concurrent tasks for observation, control, data acquisition, processing, communication, and monitoring. When an application fault occurs in a particular task, the system should allow forced suspension of that task via remote terminal, completion of online modifications or downloads, and subsequent restoration to ready state for execution—all without system downtime or impact on other tasks.

Further consideration reveals that if the system operates as a dual-state interpret/compile system, code maintenance becomes more direct and expedient. If applications adopt a modular architecture, bug fixes require only replacement of the faulty module rather than reloading the entire program. Does an operating system exist that meets these complex embedded application demands under resource constraints? One answer is the embedded multitask operating system based on Forth virtual machine architecture (FVMOS).

Forth is inherently a process control language and rapid development environment, not merely a programming tool. It possesses strong interactivity, constructability, portability, and self-extensibility capabilities, along with efficient code generation, enabling rapid construction of real-time multitasking operating systems [1]. Since its invention, Forth has evolved through standards including FIG-Forth, Forth-79, Forth-83, ANSI X3.215-1994 [2], ISO/IEC 15145:1997, and FORTH-2012 [3]. Forth language systems are increasingly employed in embedded software and firmware design, with implementations such as Flash-Forth (PIC18Fxxxx), PicForth (PIC16F8xx), hForth (x86 StrongARM), Quartus Forth (Palm Pilot), F68KANS (68K), Forth for the TMS320C50 [4], Mecrisp-Stellaris (ARM-Cortex), PunyForth (ESP8266), and AmForth (Arduino AVR8), covering virtually all mainstream embedded processors. Strictly speaking, these are primarily Forth programming environments. The true milestone in advancing Forth into the operating system domain was the Stand Alone Forth11 developed by the U.S. Kitt Peak Observatory [5], which has remained the benchmark for Forth operating systems.

The Stand Alone Forth11 system is essentially a multitask operating system with rapid I/O capabilities. Its most distinctive feature is the implementation of Forth dictionary sharing through PDP-11 page address mapping. Excluding the ROM monitor bootstrap, the entire system occupies only 40K. The subsequent Stand Alone Forth88 [6] lacked page mapping mechanisms and thus did not achieve code reuse, with each task representing an independent virtual machine occupying 64K of memory. Chuck Moore, building upon research on SEAforth single-chip 40-core and 100-core processors [7–9], introduced Colorforth—the first streamlined embedded Forth operating system. Colorforth embeds a multitask operating system and basic I/O drivers, with only 2K of reentrant core code, now expanded from its original RISC processor to multiple platforms. Colorforth testing demonstrates that Forth code volume is merely 1% of equivalent C code, and that K-level code can accomplish operations traditionally requiring M-level operating systems [10]. The most influential industry implementation is SwiftX from Forth Inc., which also embeds an efficient streamlined Forth multitask

operating system SwiftOS [11], characterized by excellent portability and system reconfigurability. A typical example is that if terminal tasks are unnecessary, the generated target system may exclude the text interpreter entirely.

1.1 Forth Virtual Machine

The hardware platform for FVMOS may be either a hardware Forth stack processor or a register-based processor. Since the operating system runs on the Forth virtual machine, its implementation exhibits considerable uniqueness.

From the Forth system structure perspective, Forth is a dictionary-based system. The dictionary comprises multiple vocabularies, each consisting of Forth definitions (containing NF name field, LF link field, CF code field, and PF parameter field). Each Forth definition corresponds to an independently functional program within the system. All Forth definitions are organized and linked together through a unified data structure (dictionary structure), with LFA pointing to the previous Forth word. Forth definitions follow strict hierarchical calling relationships: high-level Forth definitions can only invoke low-level Forth definitions, with hierarchical relationships completely implicit within the Forth definitions themselves. The lowest-level Forth definitions consist of stack processor instructions or Code definitions composed of assembly instructions, with colon definitions (high-level definitions) at higher levels.

From an operational mechanism standpoint, Forth is a dual-state system merging interpreter and compiler. The Forth virtual machine directly executes stack processor instructions or Code code composed of assembly instructions. For register-based processors, an additional address interpreter (or inner interpreter, machine primitive) called Next controls Code execution. Beyond Next, a main control loop called Quit governs the entire system—this is the interpretive compiler. Under Quit' s control, when a newly input/scanned string is found in the dictionary, its CFA is either compiled into the dictionary (compile state) or transferred to Next for immediate execution (execution state). If not found in the dictionary, it is converted to a number, either compiled into the dictionary (compile state) or pushed onto the stack (execution state).

1.2 FVMOS

Building upon mainstream FVMOS architecture research, this paper proposes the overall FVMOS architecture shown in Figure 1 [Figure 1: see original paper]. The Forth Virtual Machine (FVM) situated above the CPU is essentially the Forth address interpreter NEXT. During runtime, it utilizes the private stack of the currently executing task within its user variable area. At system power-on, the Stask stack is used. As a first-level abstraction, FVM effectively shields hardware details. Its instruction pointer resides in each task' s return stack, requiring only a current user variable area pointer UP for task switching, thereby eliminating the overhead of traditional context switching.

Unlike conventional operating system architectures, FVMOS does not run directly on the hardware processor but rather coexists with user task programs on the Forth virtual machine. Consequently, the FVMOS architecture essentially represents multitask layout, inseparable from multitask management. Although preliminary research proposed an EFOS (Embedded Forth Operating System) framework [12] based on Forth philosophy, it inadequately reflected FVM-based design thinking and left several issues requiring deeper investigation. This paper, grounded in mainstream FVMOS multitask operating system architecture research, targets reconfigurable, extensible, portable, and interactive multitask organization and management, proposing an FVMOS architecture and corresponding multitask management algorithms that satisfy complex embedded application requirements.

2.1 Memory Allocation and Management

Based on prior research and considering embedded storage layout alongside mainstream Forth cooperative scheduling algorithms, this paper optimizes the design by abandoning the x86-inspired approach of dividing Forth memory into four logically independent segments (CS:, DS:, VS:, SS:). Instead, we establish three groups of access operation definitions and corresponding pointers for Flash, RAM, and E2PROM, designated for code, data, and parameter storage respectively.

In conventional layouts, separate Flash and RAM dictionaries would be required, but this approach not only increases dictionary management overhead but also causes loss of dynamically loaded tasks in RAM upon system power failure, reverting the system to its pre-power-on state. Therefore, this architecture stores all Forth reentrant code running on FVM in Flash. The FBS lower layer comprises Code definitions, with high-level definitions above. Built upon FBS is FVMOS, composed of high-level definitions with downward dependencies. Above FVMOS are multiple user tasks belonging to Stask. To streamline target systems, this architecture incorporates optional FBS components; if online interaction is unnecessary for the application system, the generated target system may exclude the text interpreter Interpreter.

Flash stores reentrant FBS and FVMOS, with allocation determined by the DP pointer. Access operation definitions include: `,`, `@I`, `IHERE`, and `IALLOT`. RAM stores all program runtime data, with access operation definitions including: `!`, `@`, `HERE`, and `ALLOT`. The UP pointer indicates the base address of the user variable area for the currently running task (TCB). Most embedded processors lack page address mapping and memory protection mechanisms. To achieve code protection, explicit Flash access operations may be hidden.

Considering the complexity of Forth' s online interactive process that permits dynamic code loading at any time, the DP in Flash is allocated in strict address-incrementing order. Correspondingly, UP in RAM may employ either simple sequential allocation based on actual requirements or dynamic storage man-

agement techniques such as dynamic allocation/recycling algorithms and buffer pools.

2.2 Reentrancy and User Variable Area

Reentrant code requirements dictate that no private variables may exist except constants. The aforementioned storage management model supports system reentrancy transformation—with minor modifications, the entire Forth system can achieve reentrancy. Forth system characteristics determine that most runtime parameters and results can reside in the data stack or return stack, while strings, variables, and arrays requiring processing can be stored in RAM data areas allocated to each task.

Introducing Forth user variables is key to achieving these objectives. To ensure reentrancy, public routines can be defined to address variables in RAM data areas—these variables are Forth user variables. Forth user variables are essentially address variables, with the PF in their definition storing the variable' s address in RAM.

The user variable area is a special region within RAM. While each task can share code such as text interpreters and I/O drivers, each possesses independent operational data defined by user variables. These private data are stored in different regions of the RAM data area—this region is the user variable area.

2.3 Task Control Block

Forth operating systems typically employ cooperative scheduling strategies, meaning each task's activation time is predetermined. Unlike other multitasking approaches, FVMOS' s cooperative scheduling is VM-based, supporting three task types: terminal tasks, background tasks, and interrupt tasks. Although TCB contents differ among the three task types, all connect to the multitask circular linked list (right dashed line).

Context preservation requires only pushing the current return stack pointer RP onto the data stack and saving the current data stack pointer SP to the task' s user variable area. Context restoration requires only recovering SP from the task' s user variable area and storing the stack top value in the RP pointer. The FVMOS Task Control Block (TCB) is a special area within the user variable area related to multitask scheduling, with its structure shown in Table 1 .

Each item in the TCB table is defined through the user variable `USER`. The basic items (first six) are essential for every task, while additional items are specifically designed for terminal tasks. The additional I/O driver item stores the terminal task' s I/O driver vector, related to the specific terminal device connected to that task. The additional interpreter operation item stores the operation vector associated with the terminal task' s text interpreter and state. Notably, the TCB does not retain the return stack pointer; instead, the current return stack pointer resides at the data stack top.

3 FVMOS Multitasking Architecture and Organization Management

FVMOS scheduling primitives include `PAUSE`, with the introduction of swappable Forth vector definitions `PASS` and `WAKE` enabling rapid task switching without TCB table lookup. The following discussion focuses on task management algorithms related to operating system architecture. Considering Forth language and system characteristics, this paper employs the Forth2012 standard for algorithm description.

3.1 Flash Task Definition

Creating a special task definition in the Flash Forth dictionary involves saving the allocated user variable area base address to the CFA code field, and return stack and data stack bottom pointers to the PFA parameter field. The defined CFA and PFA together constitute the Task Information Block (TIB). The task definition algorithm is as follows:

```
: TASK ( ds rs us "name" -- )
   <BUILDS
   HERE ,          \ Store HERE (available TCB address in RAM) to Flash task definition's CF
   (us) &12 + ALLOT   \ Allocate user variable area in RAM (first 6 items)
   (rs) ALLOT HERE ,  \ Allocate return stack, save rp0 to Flash task definition's pfa[0]
   (ds) ALLOT HERE ,  \ Allocate data stack, save sp0 to Flash task definition's pfa[1]
   1 ALLOT            \ Separator
   DOES> ;            \ When executing task, push CFA onto data stack
```

The entry parameters `ds`, `rs`, and `us` represent data stack size, return stack size, and additional user area size respectively. `<BUILDS` creates a task header in Flash, subsequently storing the available address `HERE` in RAM user variable area (corresponding to `IHERE` for Flash operations) into the Flash task definition' s CFA (i.e., TCB base address). It allocates the TCB in RAM user variable area (first 6 items are basic), allocates the return stack, saves `rp0` to the Flash task definition' s `pfa[0]`, allocates the data stack, and saves `sp0` to the Flash task definition' s `pfa[1]`. When executing the task (`DOES>`), the CFA is pushed onto the data stack.

3.2 Memory Area Interoperability

Interoperability between TIB in Flash and TCB in RAM is implemented through the following definitions:

```
: TIB>TCB   ( -- tcb )   @I ;          \ Map TIB to tcb[status]
: TIB>RP0   ( -- rp0 )  I-CELL+ @I ;  \ Map TIB to tcb[rp0]
: TIB>SP0   ( -- sp0 )  I-CELL+ I-CELL+ @I ;  \ Map TIB to tcb[sp0]
: TIB>SIZE  ( -- size ) I-CELL+ I-CELL+ I-CELL+ @I ;  \ Convert TIB to user variable area si
```

Here, `@I` differs from RAM operation `@` by fetching data from Flash. The system retains Forth convention, treating conventional access operators as RAM operations.

3.3 Task User Variable Area Initialization

After task creation, `TASK-INIT` initializes the TCB and stack area in the task's user variable area, sets the default radix to decimal, and places the task in the `PASS` sleep state before linking the task body. The `TASK-INIT` definition algorithm is as follows:

```
: TASK-INIT ( tib -- )
   DUP TIB>TCB OVER TIB>SIZE 0 FILL  \ Initialize TCB and stack area in task user variable a
   DUP TIB>SP0 OVER TIB>TCB &6 + !   \ tcb[sp0]=tib[sp0]
   DUP TIB>SP0 CELL- OVER TIB>TCB &8 + !  \ tcb[TOS]
   DUP TIB>RP0 OVER TIB>TCB &4 + !   \ tcb[rp0]=tib[rp0]
   &10 OVER TIB>TCB &12 + !          \ tcb[base]=10
   TIB>TCB TASK-SLEEP ;              \ tcb[0]=PASS
```

The entry parameter `tib` is the TIB popped by `DOES>` when the task header executes. Through interoperability conversion to TCB address, it clears the TCB and stack area in RAM user variable area, stores the `sp0` saved in Flash task header to `tcb[sp0]`, calculates the stack top and saves it to `tcb[TOS]`, stores the `rp0` saved in Flash task header to `tcb[rp0]`, sets default decimal radix, and finally sets the initial task state to `PASS`.

The multitask storage architecture is illustrated in Figure 2 [Figure 2: see original paper].

4 Experimental Evaluation

On the Arduino embedded hardware platform, we implemented the FVMOS basic framework and multitask management algorithms using an open-source Forth virtual machine. In the experimental environment, during system power-on initialization, we designed a test program containing one terminal task `TASK1` and one background task `TASK2`:

```
: MS ( n -- ) PAUSE 0 ?DO 1MS LOOP ;  \ Call PAUSE every n milliseconds

VARIABLE M  \ Global variable M used by TASK1 and TASK2

: INITM ( -- ) 0 M ! ;

$40 $40 0 BACKGROUND-TASK TASK2  \ Create background task TASK2, create task header in Flash

: TASK2-BODY ( -- ) BEGIN 1 M +! &10 MS AGAIN ;  \ Define TASK2 body

: STARTTASKER ( -- )  \ Initialize and start multitasking
```

```
   TASK2 TIB>TCB ACTIVATE TASK2-BODY  \ Link TASK2 body
   ADD-FIRSTTASK  \ Create TCB1 circular linked list
   TASK2 TIB>TCB ADD-NEXTTASK  \ Add TCB2 to circular linked list
   MULTI ;  \ Start multitasking
```

After completing task creation, initialization, and body linking, the multitask TCB circular linked list must be initialized and constructed. `ADD-FIRSTTASK` initializes the first terminal task, sets its `tcb[status]` to `WAKE`, and makes `FOLLOWER` point to itself. The `ADD-FIRSTTASK` definition algorithm is as follows:

```
: ADD-FIRSTTASK
   WAKE STATUS !      \ Set current task's tcb[status] to WAKE
   UP@ FOLLOWER ! ;   \ Make FOLLOWER point to itself
```

`ADD-FIRSTTASK` initializes the first terminal task, setting its `tcb[status]` to `WAKE` and making `FOLLOWER` self-referential.

After system power-on startup, through terminal task `TASK1`, a new background task `TASK3` is established. The following program is dynamically loaded via command line or file:

```
VARIABLE N  \ Global variable N used by TASK1 and TASK3

: INITN ( -- ) 0 N ! ;

$40 $40 0 INTERRUPT-TASK TASK3  \ Create interrupt task TASK3, create task header in Flash,

: TASK3-BODY ( -- ) BEGIN 1 N +! &10 MS AGAIN ;  \ Define TASK3 body

: STARTTASK3 ( -- )  \ Initialize and start TASK3
   TASK3 TIB>TCB ACTIVATE TASK3-BODY
   TASK3 TIB>TCB ADD-NEXTTASK ;
```

The terminal task `TASK1` is actually the system task (Stask), containing both FBS and FVMOS components, encompassing all tasks defined before system power-on and new tasks dynamically loaded by terminal tasks after system startup. Therefore, all tasks can be effectively controlled through terminal tasks using `SLEEP`, `WAKE`, `STOP`, and other operations. Experiments demonstrate stable and reliable system operation, with related algorithms achieving the objectives of reconfigurable, extensible, portable, and interactive multitask organization and management.

Compared with the Stand Alone Forth88 multitask system based on CPU scheduling [6], the proposed FVMOS multitask architecture offers numerous advantages. Regarding embedded limited storage space utilization, the FVM-based architecture reduces TCB size. Task switching requires only preserving or restoring the SP pointer, compressing dozens of CPU image storage items in the TCB to just six units. In terms of reconstruction, extension, and portability, the entire multitask management system is constructed upon the system' s

high-level FBS definitions, achieving hardware independence. Regarding system simplicity, the special scheduling method and streamlined TCB structure eliminate the need for separate task queue creation and maintenance. Compared with Stand Alone Forth88, the code size for task creation at the same level is compressed by at least 10 times, with overall compression reaching 16 times, as shown in Table 2 . In terms of efficiency, besides reducing conventional priority-based round-robin scheduling algorithm complexity, all multitask management algorithm complexities are O(1).

Table 2. Code Size (LOC) Comparison

System	Multitasker
Stand Alone Forth88	
FVMOS	

5 Conclusion

The above analysis reveals that FVMOS is fundamentally different from traditional approaches, lacking strictly distinguished modules, layers, microkernels, or other conventional structures. Although the system possesses inherent reconfigurability and even componentization characteristics, it differs from componentized operating systems. Forth' s unique stack and dictionary structures determine its special architecture and operational mechanisms. On one hand, Forth' s threaded code enables the Forth stack machine' s dictionary structure to serve as a dynamically and interactively extensible open program library. Additionally, the Forth virtual machine runtime environment provides rare cross-platform capabilities. On the other hand, precisely these characteristics introduce challenges for VM-based multitask organization and management. For instance, since all processes (including task scheduling) must operate under Quit control, classical multitask management algorithms cannot be directly applied. Although real-time performance requires further improvement, the FVMOS architecture and multitask management algorithms investigated in this paper offer practical guidance value.

References

[1] Dai Hongbing. Development of a new and efficient microcomputer Forth language [J]. Journal of the Graduate School of the Chinese Academy of Sciences, 1993, 10(1): 62-69.

[2] ANSI X3.215-1994 American National Standards for Information Systems Programming Languages Forth [S]. New York: American National Standards Institute, 1994.

[3] Forth-Standard-Committee. FORTH-2012 [S/OL]. (2015). https://forth-standard.org/.

[4] FORTH. Inc. Featured Forth Applications [J/OL]. (2009). http://www.forth.com/resources/app Notes.

[5] Thomas E. McGuire. Kitt Peak Multi-Tasking FORTH-11 [J]. The Journal of Forth Application and Research, 1984, 2(2): 57-67.

[6] Dai Hongbing. Design and implementation of an efficient microcomputer real-time multitask operating system [J]. Journal of the Graduate School of the Chinese Academy of Sciences, 1993, 10(3): 283-292.

[7] James Rash. Space-related applications of forth [J/OL]. (2006-09). http://forth.gsfc.nasa.gov/.

[8] Caffrey R T. Forth in space: interfacing SSBUV: a scientific instrument, to the space shuttle [J]. ACM Sigforth Newsletter, 1993, 4(3): 1-8.

[9] IntellaSys, A TPL Group Enterprise. SEAforth 40C18 Scalable Embedded Array Processor [EB/OL]. (2008). http://www.intellasys.net/templates/trial/content/SEK_40C18_DataSheet

[10] Mikiten B C, Mikiten S, Orr J L. A Forth-based real-time in-flight monitoring system [C]// Proc of the 2nd & 3rd Annual Workshops on Forth. New York: ACM Press, 1991: 31-34.

[11] Forth Inc. SwiftX Cross Compilers for Embedded Systems Applications [EB/OL]. (2016). https://www.forth.com/embedded/.

[12] Yang Weimin, Dai Hongbing, An Hongping, et al. Research on a new embedded Forth real-time operating system [J]. Journal of Yunnan University: Natural Sciences Edition, 2013(S2): 96-103.

[13] Paul Frenger. Forth and AI Revisited: BRAIN.FORTH [J]. ACM SIGPLAN Notices, 2004, 39(12): 11-16.

[14] Stephen Pelc. Programming Forth [M]. [S.l.]: MicroProcessor Engineering Limited, 2011: 97.

[15] Dai Hongbing, Yang Weimin, Wang Liqing, et al. Research and implementation of a multi-target Forth self-generator [J]. Application Research of Computers, 2014, 31(4): 1109-1114.

[16] The Forth Interest Group. Forth Compilers Page [EB/OL]. (2009). http://www.forth.org/compilers.html.

[17] Frederic P Miller. Colorforth [M]. Alphascript [S.l.], 2010: 28.

[18] Frenger P. Hard Java [J]. Acm Sigplan Notices, 2008, 43(5): 5-9.

[19] Hanna D M, Jones B, Lorenz L, et al. An embedded Forth core with floating point and branch prediction [C]// Proc of IEEE International Midwest Symposium on Circuits and Systems. 2013: 1055-1058.

Note: Figure translations are in progress. See original paper for figures.