"Want", a Flash Token-Threaded Virtual-Machine and Operating-System for DSPs

EuroForth 2004 - Christophe Lavarenne (cl@ubic.fr - UBIC S.A. - http://www.ubic.fr)

Abstract:
More and more low-cost applications (telecommunications, automotive, metering, alarm systems...) embed, beside a microcontroller unit (MCU), a Digital-Signal-Processor (DSP) to cope with data rates and their processing under hard real-time constraints. For such applications, the UBIC company has developed "WantOS", a multitasking real time operating system with interrupt-driven peripheral drivers, integrated into "WantVM", a compact and efficient token-threaded virtual-machine running several megabytecodes per second directly out of flash, programmable in either "WantC" with a full-featured graphical user interface, or more interactively in "WantForth". This paper presents this software architecture and the savings it has allowed, compared to usual designs, in the hardware architecture and in the design and life cycles of hundreds of thousands of remote alarm systems, and of new real time video-processing projects.

1. Introduction

Real-time embedded applications design is often deeply split into areas, that designers like to call "layers" (analog/digital, hardware/software, system/application, dataflow/control, to give only a few examples), each designed with very different tools requiring very different skills based on quite different knowledge and often using the same vocabulary with different meanings. Hence the difficulty to master several areas, then to communicate clearly between designers skilled in different areas, to the point that they tend to scorn or even distrust each other, and protect their side of the interface against unexpected (misunderstood/misdesigned) behavior from the other side, globally leading to interfaces overcosts.

This paper focuses on the split between the so-called signal-processing (SP) and control-processing (CP) areas.

SP is mathematically founded, originally on continuous signals processed with analog hardware, and more and more on sampled discrete signals processed by DSPs dealing mostly with periodic dataflows and massive amounts of almost repetitive computations with parallel operations and specific post-modify addressing modes (such as circular for delay lines, and bit-reversed for fast block transforms).

In comparison, CP is rather a cook art, so many are the implementation choices for the encoding and the computation of the control state, between combinatory and/or sequential logic, state transition diagrams and/or multitasking, which are far from equivalent in terms of execution speed, memory footprint, and ease of design and debug. But for average CP designers, this seems usually more a matter of personal culture and taste, than an efficiency concern, as is their usual preference for MCU architectures, less efficient but simpler than DSPs, with general purpose addressing modes (such as indirect-indexed for accessing global data structures and stack frames), and integrating on-chip flash memory and more general-purpose multifunction input-output pins, also supporting various serial communication standards (such as UART, I2C, SPI, CAN).

This SP/CP split has led to lots of designs integrating both a DSP and an MCU communicating by some shared link or memory, where the DSP is usually under the control of the MCU, which takes care of the application "high" layers. However, programming such different processors and making them efficiently synchronize and communicate, is not a trivial design activity.

The need existing for a simpler and less expensive solution, several processor manufacturers (DSP-Group with their Teak core, Microchip with their dsPIC, Analog Devices and Intel with their BlackFin) have designed new DSP architectures, with an instruction set enriched to offer all usual MCU instructions, and with family variants enriched to offer all usual MCU peripherals. Then both SP and CP designers are expected to be happy sharing the same processor, and therefore the same communication library relying on some multitasking scheduler.

For very limited SP activity, some MCU manufacturers (such as ARM, or Texas Instrument with their MSP430) also offer a optional multiplier-accumulator unit in their MCU architecture, but this doesn't compete with a full fledged DSP, such as described in the next section.

The UBIC company has thought a different approach, usable on any DSP, and trying to make the best use of each DSP resource. From the obvious observations that:

- a DSP may not be under-dimensioned (or it would not satisfy its application's real-time constraints),
- all its SP is triggered by hardware periodic interrupts,
- its mean unused processing power is still very big compared with an MCU, even during worst case SP,

UBIC decided to "integrate" in software (i.e. emulate) an MCU into the DSP during its SP idle periods (i.e. as the lowest priority task of the DSP), using a portable and compact virtual machine (VM), classifiable as dual-stack token-threaded bytecoded flashed architecture.

Section 2 describes current DSP available resources and their relative costs, driving the implementation choices of the VM and of its multitasking operating system described in section 3. Section 4 presents the available development environments and their original features. Section 5 presents some real applications.

2. DSP resources and costs

DSP architectures are designed to efficiently transfer and process periodically sampled signal data flows, in real-time and at low costs.

The data transfers between the DSP memory and analog-digital converters (either integrated on-chip, or through standardized ports, serial or parallel) is nowadays supported by a dedicated multichannel DMA (Direct Memory Access) controller running in parallel with the CPU (Central Processing Unit).

The CPU is mainly optimized for computing sums of products, the most intensive operation of most signal processing algorithms: on every clock cycle, the CPU is able, in parallel, to fetch an instruction from memory, to decrement a loop counter and jump back to a loop entry, to read up to two data from memory or to write one, to update up to two data pointers (in linear, circular, or bit-reversed addressing), and to combine up to three data (such as multiply two data and sum their product into an accumulator).

Newer DSPs, such as the Blackfin of Analog Devices, are even able to execute two MACs (multiply-accumulate) in parallel per instruction, at up to 800 Mips, for only half a milliwatt per Mips (Mega-Instructions Per Second, of course hardly comparable with Mips of fan cooled general purpose processors) and about two square centimeters of printed circuit board (PCB) space.

The cost of DSPs is almost proportional to their on-chip RAM size (around one cent/kilobit), which typically varies between around a half and one megabits of zero wait-states, partially cacheable program and data memory. In comparison, off-chip memories are easily one to two orders of magnitude cheaper and bigger (around 20 and 5 cents/megabit for 8 Mbit 70ns FLASH and 256 Mbit 7ns SDRAM) against some wait-states.

However, note that SDRAM are hardly found in small sizes, then cost more than the DSP and the FLASH together, require also much more power supply and more PCB space than them, make the PCB harder to route and the memory bus noisier, i.e. the PCB electro-magnetic compliance harder, so avoid SDRAM when possible. In particular, prefer operating systems which are able to live without an SDRAM.

You'll hardly avoid a flash, unless you don't need reconfigurable non-volatile memory and your DSP offers on-chip maskable ROM matching your application needs. Then, if you are still concerned with cost, you'll try to limit the DSP on-chip RAM size, and use instead the flash as much as possible. The UBIC company has followed these guidelines to design the "WantVM" virtual machine.

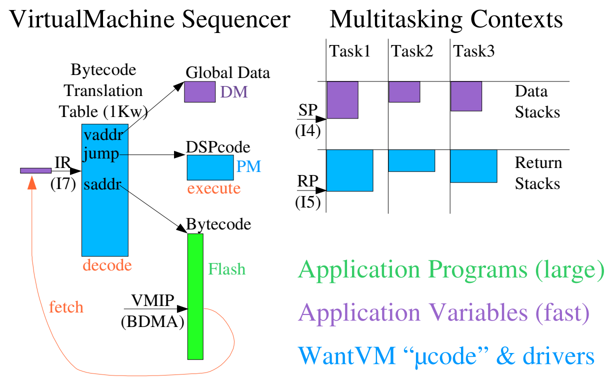
3. WantVM and WantOS architectures

The DSP precious on-chip RAM is reserved for critical highly optimized native code (drivers interrupt and SP compute-intensive subroutines, and VM primitives), and for system buffers and VM tasks contexts (data and return stacks, and global variables).

To avoid allocating RAM for application CP code, the VM executes it "in place" by fetching one bytecode at a time directly from flash when it needs it for execution. As CP activity is sporadic and less critical, its reaction time is dampened by drivers interface buffers to absorb interrupt delays and flash waitstates.

The VM sequencer is pipelined: while interpreting a bytecode, it prefetches the next one, such that interpretation time is mostly masked off by prefetch waitstates. Waitstates are also minimized by compact VM programs, obtained with an open bytecode instruction set, mostly based on 0-operand 1-byte coded instructions using:

- implicit register addresses, thanks to two last-in-first-out register stacks, each sequentially accessed through an implicit stack pointer (instead of the more usual register array, randomly accessed through an explicit index stored in the instruction)
- implicit memory addresses, stored only once in a Bytecode Translation Table (BTT) in RAM, not only for the primitives RAM entry addresses, but also for the subroutines flash entry addresses, and for the RAM buffers base addresses
- 1-operand 2-bytes codes are only used for literals, jump offsets, return stack offsets (for efficient C stack frames support), BTT extensions over 256 bytecodes, and for drivers functions calls.



The VM instruction set is limited to 50 primitives to spare RAM and BTT entries, but is completed by a set of 100 inlining macros. It natively supports cooperative multitasking, structured exceptions, in-flash debugging, and a dozen drivers (UART, I2C, RTC+timers, periodically polled I/Os, 8KHz Audio, voice+V32bis Modem, Flash file system with adpcm codec, with configuration and circular log spaces, etc., each with its private interface functions and BTT) totaling 150 functions (among which 30 are bytecoded) making up "WantOS".

WantOS drivers interfaces have been designed to mimic MCU status/control and data I/O registers behavior (instead of the OS-usual central event queue, which multiplexing costs space and time to store, code and decode events identifying tags). But to avoid unnecessary status polling and save DSP power, the *pause* multitasker primitive (which switches the VM task context) and the drivers interrupts and functions cooperate to put asleep and wakeup the VM in the following way:

- *pause* decrements a task-switch counter, and if null puts the VM asleep in the middle of a task-switch by stopping the DSP in low power IDLE mode;

- on an interrupt request, the DSP wakes up and executes the interrupt subroutine; if this one changes its driver's status, it also loads the task-switch counter with the number of tasks;
- on return from interrupt, *pause* checks the task-switch counter: if still null, it keeps the VM asleep by stopping the DSP, otherwise it awakes the VM by letting the DSP execute the next VM instruction;
- then, before the task-switch counter becomes null, each VM task has a chance to poll the status of the driver(s) it takes care of, and if active, to service it; if a driver function modifies the driver's status, it also loads the task-switch counter with the number of tasks.

Therefore, the VM runs until all tasks have polled only inactive driver status, where *pause* puts the VM asleep.

WantOS includes several *loaders*, between a serial link, the flash and RAM, to support DSP-native code and bytecode boot, execution, and interactive development:

- a *boot-loader*, bringing up a serial "native-monitor" (with its own serial-to-RAM loader) for low-level interactive umbilical development in DSP assembler, then looking the flash file system for the VM native-code executable startup file to load;
- a *native-loader*, called to load from flash to RAM the DSP-native-code executable files composing the VM kernel and WantOS drivers (of which the modem is composed of several overlays, sharing the same RAM space during different modem modes, to spare RAM);
- a *bytecode-loader*, called mainly by the VM boot to compute and load into the BTT the flash addresses of the subroutines entry points of compiled bytecode files (which are therefore relocatable, i.e. may be stored anywhere in flash), in which each subroutine is preceded by its size; from an initial bytecode number given in the file header, the compiler and bytecode-loader assign bytecodes and BTT entries in the same order; a null subroutine size triggers the deallocation of the last bytecode and its execution (this supports initializations and startup)
- a *frame-loader*, called mainly by a "bytecode-monitor" (brought up by the VM boot task after some initializations, including spawning a task bytecode-loading the application specific startup file if found), to receive, error-check, and acknowledge (or request retransmission of) HDLC-like frames through a serial driver (such as UART or modem) from a remote computer into a TEMP flash area; depending on the frame's first two bytes, the bytecode-monitor either saves the frame as a file in the flash file system, or executes a system service (such as flash file system listing or update, configuration space dump or update, log space dump, real time clock update, etc.), or executes an application hookable *frame-interpreter* (this supports any frame-oriented application specific communication protocol), or finally executes the frame's bytecode contents (this supports bytecode interactive development, or remote maintenance through precompiled subroutines).

In-flash-debugging is supported by the *stopwm* VM instruction, which looks for its own flash address in a breakpoints table in RAM, and if found suspends VM execution by calling the native-monitor, to allow inspection or modification of the RAM state, or to single-step or resume VM execution, which may also be suspended asynchronously by a *break* condition on the debug serial line, if the UART driver has been configured to do so.

Here are some figures taken from the ADSP-218x implementation: 300 instructions (900 bytes) for the VM core primitives, 2700 instructions for all drivers primitive functions and interrupt subroutines (modem datapumps excluded), and 5 Kbytecodes for WantOS subroutines, altogether in 8 files totaling 12 Kbytes (plus 64 Kbytes in 9 files for the modem datapumps).

4. WantVM development environments

There are 3 development levels:

- WantAsm: DSP code interactive cross-(dis)assembler
- WantForth: VM bytecode interactive cross-compiler
- WantC: VM bytecode C cross-compiler/debugger

The first two are integrated in the same environment, based on Gforth under Linux or on Win32for under Windows (using only the simple usual Forth command line user interface), allowing the programmer to switch between them if needed to debug the hardware and VM.

WantC is a separate environment, based on the open LCC compiler and Wedit (its integrated development editor with a graphical user interface) under Windows, both customized for WantVM. Although WantC is not (yet) incremental, and therefore less interactive from the point of view of a Forth programmer, it is closer than Forth to what the average MCU programmer seems to be used to, and to expect.

WantAsm of course depends on the DSP type; UBIC presently supports Analog Devices 218x (16 bits 64 Mips) and Blackfin (32 bits 800 Mips) fixed-point DSP families. WantAsm may be used either standalone, to generate "Forth-less" DSP code, or to support a subroutine-threaded Forth with inline macros, or to build a VM core, as is the case here.

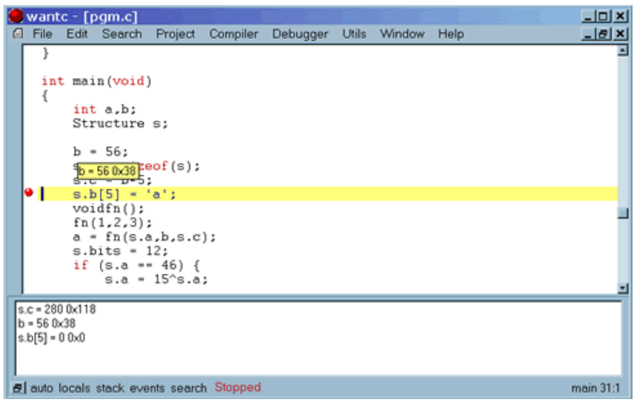
WantForth is a light cross-compiler: instead of the usual juggling with *host* and *target* separate vocabularies containing homonyms, target words are systematically suffixed by a single-quote (this makes macros easier to write and read, such as for example: `: 2*' dup' +' ;`), *target* is defined as a compile loop automatically appending this suffix to each word before its dictionary search (this saves typing the suffix for each target word, and hides non-target words), and *host* is defined as throwing an exception caught by *target* to exit its compile loop and return.

Another deeper simplification is worth to describe, that I apply to all cross-environments I have been developing for the past 15 years: they are interactive without the need for an interpreter (note that so are Lisp and Smalltalk). Forth interactivity usually relies on the execution of a word immediately after its dictionary search when in the *interpret* mode; this requires additional complexity to specify a different behavior when in the *compile* mode, mainly for *immediate* compiling words (non-*immediate* words have a common compile behavior, specified in the compile loop). This complexity explodes when combined with the cross-compilers *target* and *host* modes, to the point that most cross-compilers are not cross-interactive. Interactivity requires user-controlled interleaving between user code input and code execution, but instead of doing it at the word grain level, it may be done at the subroutine grain level, with *anonymous* subroutines: named and anonymous subroutines end with a semicolon, but a named subroutine begins with a colon and a name (added as key in the dictionary to retrieve the subroutine entry point for later reference), whereas an anonymous subroutine doesn't, therefore the only useful thing that can be done before forgetting it, is to execute it immediately. More accurately in a cross-environment, when the host terminates the cross-compilation of an anonymous subroutine, it first downloads into the target all the code cross-compiled since the previous download, and then requests the target to execute the anonymous subroutine. This works as well for WantAsm as for WantForth, as it will some day for WantC, where an *anonymous* function is limited to a function body between {curly braces}, or to a single C-statement (well, the compiler must complain if execution may follow any unresolved forward reference).

WantC has several unusual features worth to mention, apart its absence of support for *float* and *double* types. First of all, its compiler backend generates bytecode for a dual-stack engine, which was far from its usual assumption of a register-array based engine; we finally rewrote the backend from scratch.

Then it generates (very quickly) binary executable code (and separate debug-support information in XML format) directly from application and libraries source files, without any need for a separate assembler, archiver, linker, makefile, or for the intermediate files they exchange. Most library functions are defined `_INLINE_ASM` by default, with an `#ifdef` to instead generate a separate function if needed; inline assembler improves speed and saves BTT entries against a few bytes of flash for each call; the "inline-assembler" is a Forth without defining words. Among the compiled functions, are linked together into the executable code only those reachable from the *main* function, by a recursive exploration of the call-tree, including in it assignments to function pointers; during this exploration, a bytecode is assigned to each reachable function, then each function-call may be resolved, call-stack depth is computed, call-recursion is detected, and both are reported along with other allocated runtime resources (RAM, BTT, flash). It is worth to mention that this automatic stripping process not only saves runtime resources by removing dead code, but also simplifies library maintenance and selection: the link-time binary granularity being independent of the source granularity, each library may be maintained as a single simple source file, that the user has only to `#include` in his source code; library source hiding by encryption may be supported on request.

If configured in *debug* mode, WantC generates one *stopwm* instruction for every C statement or source line, whichever includes several others. To start debugging, the host-side cross-debugger communicates with the target-side bytecode-monitor to download the compiled code into the TEMP flash area; the breakpoints table is initially empty, which by default enables every *stopwm*. Then, when an enabled *stopwm* (or a user requested *break* on the debug serial line) suspends the VM execution, the target-side native-monitor signals it to the host-side cross-debugger, which then uploads the VM state, automatically highlights the source line of the reached breakpoint, and displays the variables in the five source lines around it; any variable in scope may also be inspected by simply pointing the mouse over, and waiting for a popup to display its value. To resume VM execution on user request (for a single step or more, or even to stop debugging, each case modifying differently the VM state), the cross-debugger downloads the VM state, and requests the native-monitor to resume VM execution.


wantc - [pgm.c]
File Edit Search Project Compiler Debugger Utils Window Help
b = 56 0x38
auto locals stack events search Stopped
main 31:1

```
}

int main(void)
{
    int a,b;
    Structure s;

    b = 56;
    s[illegible]zeof(s);
    s.c = b*5;
    s.b[5] = 'a';
    voidfn();
    fn(1,2,3);
    a = fn(s.a,b,s.c);
    s.bits = 12;
    if (s.a == 46) {
        s.a = 15^s.a;
```

```
s.c = 280 0x118
b = 56 0x38
s.b[5] = 0 0x0
```

If configured in *install* mode, WantC generates no *stopwm* in the executable bytecode file (typically 8% smaller). To install it, together with other application files (data, sounds, etc.) for standalone execution into the flash file system, the host-side *installer* interacts with the programmer through a dialog box, and communicates with the target-side bytecode-monitor to sequentially:

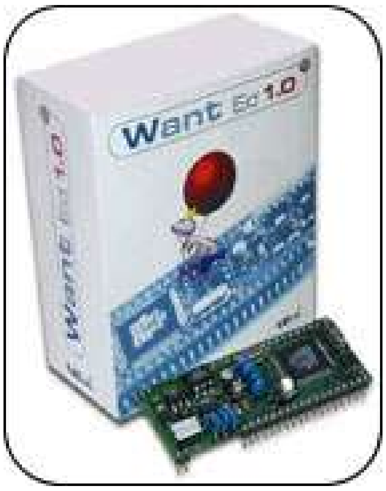
- upload the signatures (4 first and 4 last bytes) of all files in the target flash file system, compare them with the signatures of the files in the host installation directories (one for system files, the other for application files), and display a colored list of all these files, each marked with a symbol indicating its state (new, up-to-date, obsolete, etc.)
- download the files which have changed on the host (then the flash file system contains two copies of each updated file, keeping the flash file system in a consistent state, where WantOS file-search functions only see the elder copy)
- trigger the bytecode-monitor *rmreboot* service, which deletes from flash the requested files (those absent on the host) and the elder copy of duplicate files, and then reboots the VM in the consistent updated state of the flash file system.

The *installer* is not only a development tool, it may also be used for remote update by modem, of systems in the field, provided the application initiates, or accepts, a modem connection, and redirects the bytecode-monitor to use it: this leaves room for protection against undesired connections.

5. WantVM applications

The first applications supported by a WantVM on ADSP-218x, and until now the biggest in number of units running in the field (several hundreds of thousands) and running hours (they run non-stop), is for a family of burglar remote alarm systems. Compared with previous families based on 8051 microcontrollers and dedicated integrated circuits, the fabrication costs have been considerably lowered, the communication speed has been multiplied by 700 (from 20 bps DTMF to 14400 bps V32bis), allowing remote configuration and software update, and the local and remote user interfaces are improved with vocal messages (generated by concatenating sounds stored in adpcm-coded files in flash). Another family of social remote alarm systems for aged persons has followed the same conversion to WantVM on ADSP-218x, with an even simpler hardware architecture using a low-cost stereo codec, and a lower-cost telephone line interface. These two families of products are developed by the alarm manufacturer in WantForth on top of WantOS, with a few variants between the number and types of drivers. It was quite an investment to teach WantForth and WantOS to the manufacturer developers, who were used to assembly or C low-level programming on 8051 or PIC microcontrollers.

For evaluation purposes, or for smart modem applications in small series, which cannot afford in their design the high level of integration of a Want hardware core (DSP, quartz, voltage regulator, flash, codec, and isolated telephone line interface), UBIC has designed and sells the WantModem stamp (30x70mm) with its development kit, including a devboard (with leds, buttons, I2C thermometer, RS232 connector, and draft area), a power supply, an RS232 cable, and a CD with WantC and a complete technical documentation; for more information and for ordering, visit us at http://www.ubic.fr



Several demonstration applications have been developed under WantC on the WantModem. The biggest one controls a number of digital, analog, and radio-link, plug-and-play I/O modules through an I2C bus, records significant events in the flash log space, automatically calls a telephone number on an alarm event, or answers incoming calls, then guides the remote user with vocal menus and state reports, and lets him navigate in the menus and remotely control some I/Os and configuration parameters with the DTMF tones of his telephone keys. Some figures: this application weights 70 Kbytes of C source code, compiled in 0.1 seconds into 5 Kbytecodes, and is installed together with 48 sound files totaling 120 Kbytes.

New video processing and communication applications and a WantVM are under development on Blackfin, with its WantAsm connected through its hardware UART to boost the development cycle, compared with VisualDSP (Analog Devices integrated development environment), which is hopelessly slow through the USB-JTAG link of its EZ-KIT devboard. The Blackfin is a powerful 14 mm^2 piece of low-cost low-power hardware, that WantVM will make easier to program efficiently and economically by both DSP experts and MCU skilled developers; or at least by the author.

6. Conclusion

Forth trains minds to look for shortcuts.

When I can't avoid using "state-of-the-art" development environments, I feel too often limited by their lack of flexibility and embarrassed by their opaque complexity hidden behind their supposedly "friendly" graphical interfaces. Cross-development of hard real time applications, where moreover the hardware has to be debugged as much as the software, is a complex enough game to encourage the player to shortcut tools which hinder him.

The best shortcuts I have found so far, have been to build my own cross-development environments (for 8086, 8051, RTX2000, PIC, ADSP-218x, MSP430, Blackfin), almost from scratch (well, on top of open Forth systems: thanks to their creators!), and to simplify them ever more.

As presented in this paper, anonymous definitions and quote-suffixing are big simplifications of conventional Forth cross-assemblers/compilers, allowing to make them easily interactive; the open bytecode instruction set is an efficient simplification of token-threaded virtual machines, making their code more compact and easily relocatable; the recursive linker is a straight shortcut across the usual concept of compilation chain, saving lots of intermediate tools, files, and compilation time.

The ant is a simple small insect, but with regard to its size, its power and speed are impressive, and its ability to communicate with its partners makes them together even more powerful. Such is the Want. The initial W ... is another story.