Abstract

Wind River Systems, Inc. has implemented VxWorks on the TRON-architecture Gmicro/200 microprocessor. VxWorks is an interactive development environment and real-time kernel in which the real-time target is linked with networked hosts. This paper describes how VxWorks was ported to the Gmicro microprocessor, demonstrates the role of interactive development tools, and discusses VxWorks in light of the TRON concept.

1. Introduction

VxWorks, an interactive development environment and real-time kernel, has been ported to the TRON-specification Gmicro/200 (TRONCHIP).

The interactive development environment portion of VxWorks is network-based. Network hosts provide cross-development services: remote terminal communications, file service, program development tools, and utilities. Although these facilities are currently provided by UNIX™, VxWorks is host-independent and may be hosted on any distributed environment providing similar services.

The real-time kernel portion of VxWorks, called wind, supports multitasking with pre-emptive priority scheduling, intertask synchronization and communications facilities; exception handling, and memory management. The network interface may be removed after development is complete, or retained to implement application network servers.

The design philosophy of VxWorks emphasizes connectivity among development environment components, an extensible kernel, and consistent tools. "Connectivity" implies both distributed data and distributed control.

The wind kernel derives significant benefits from the Gmicro architecture, achieving more elegant and faster code. At a higher level, the VxWorks design complements the TRON concepts of interactive development and distributed processing.

2. VxWorks development environment

Real-time target systems usually don't have many development resources. Standard host computer environments are well-equipped with development tools, storage, and utilities.

The VxWorks environment allows development on target systems to use the full range of services available on host systems, at the minimal cost of a standard network interface added to the target hardware.

2.1 VxWorks environment components

The VxWorks development environment provides resources for the job of real-time target development:

Module loader. Cross-developed object modules are transparently and incrementally loaded to the target.

Target System Symbol Table. Symbols are dynamically defined by incremental loads and by interactive definition.

Shell. VxWorks provides a C-interpreter interface that allows interactive execution of most C language expressions, VxWorks and user-loaded functions, and manipulation of defined symbols. In addition, new symbols may be defined and accessed interactively.

Debugging Tools. VxWorks provides source-level debugging, a symbolic disassembler, symbolic C-subroutine traceback, task-specific breakpoints and single-stepping, system status displays, and exception handling.

Performance Evaluation. An execution timer, configurable to monitor a single routine or a group of
routines, and CPU utilization monitors support performance evaluation.

I/O System. VxWorks implements a UNIX-compatible I/O system on the target, including UNIX standard buffered I/O.

Bus Interface. Multiple processors sharing a common bus communicate via shared memory mechanisms.

I/O Drivers. Target I/O drivers support serial I/O devices, remote file access via a network driver, intertask communications via a pipe driver, a RAM disk and SCSI hard-disk access.

Remote File System. Applications may access Network File System (NFS) facilities to transparently access files on any attached NFS server. Additional facilities are provided for accessing non-NFS servers.

Network Interface. VxWorks implements UNIX source-compatible sockets, remote command execution, remote login, Remote Procedure Call (RPC), and remote file access via standard Ethernet connections.

Board Support Package. Most of the customization for VxWorks target hardware is done in a single target-specific source library. This code includes routines for hardware initialization, interrupt handling and generation; hardware clock and timer management; local and bus memory mapping; memory sizing, and related functions.

BootROM Package. A standard package of utilities to support target CPU initialization from the network, including a minimal-function debug monitor.

3.0 The real-time kernel

The VxWorks Real-Time Kernel, called wind, is based on the principle of providing a minimum core of required real-time services with the option of user extensibility. The functions and components of the kernel include:

Multitasking. The kernel provides a multitasking environment. The kernel maintains context information and defines the following states for each task:

* ready — the task is waiting only for CPU time
* pending — the task is blocked because some resource is unavailable
* delayed — the task is inactive for a pre-determined time interval
* suspended — the task is prevented from becoming active

Scheduling. The kernel implements priority-based pre-emptive scheduling, which may be augmented by round-robin selection. Under pre-emptive scheduling, each task is assigned a priority. The kernel activates the highest priority task that is in the ready state. Hardware interrupts always have overriding priority.

Control. The kernel supports task creation, initialization, activation; task suspension, resumption, restarting, and delay; task ID, option selection, and status query.

Intertask Communications. The kernel uses a shared-memory model for local intertask communications. Semaphores provide basic mutual exclusion and synchronization. Message queues and pipes provide intertask message passing within a CPU. Sockets and Remote Procedure Calls (RPCs) provide network-transparent intertask communications. Task exceptions are handled by suspending the task, recording task state, and displaying a console message.

Extensibility. The wind kernel may be extended — without rebuilding— by using “hooks” provided. User code may be invoked whenever a task is created, when a task context switch occurs, and when a task is deleted. In addition, spare fields in the task control block are available to extend the context of each task. An “unhook” mechanism is provided to remove extensions individually.

4.0 Porting VxWorks to Gmicro/200

This section describes the steps followed when porting VxWorks to Gmicro/200. VxWorks facilities were used extensively during the process.

VxWorks is structured to localize target-specific code and other information in specific files. The port was done by making very careful changes to these files in a staged order. Each stage established a stable environment and set of tools to support the next stage.

For porting VxWorks to Gmicro, the cross-development software tools were the GNU C compiler, GCC, version 1.37.1; the GNU Gmicro/200 assembler, GAS, version 1.36; and the GNU Gmicro/200 loader, GLDL version 1.35. The development environment was a network of Sun Microsystems SPARCstations all running SunOS UNIX.

A Hitachi H32/200 ASE in-circuit emulator was used to do low-level debugging, for checking generated code, and for examining the behavior of the Gmicro/200 device.
Figure 1. Gmicro port development configuration

The target was the Gmicro/200 microprocessor on the Hitachi H32SBC, a VME-bus board. The H32SBC is equipped with EMS (Executive Monitor System) firmware, which supports debugging and serial image crossloading.

The development configuration is shown in Figure 1.

4.1 Porting Step 1: portable kernel modifications

The portable wind kernel was modified to provide the minimal necessary functions. A prototype Board Support Package was implemented, using serial communications for host and terminal interfaces -using polling only- via EMS monitor calls. No interrupts were supported.

This step required assembler and C code changes to the architecture-dependent portions of VxWorks. Most of the C code in VxWorks is architecture-independent.

Care was taken at this and subsequent steps to generate assembly instructions common to Gmicro/100, Gmicro/200, and Gmicro/300.

4.2 Porting Step 2: establishing the VxWorks shell environment

The minimally-modified portable kernel image was generated and transferred to the target hardware via serial link under control of the EMS monitor. This image was debugged using the in-circuit emulator and EMS instruction and memory access breakpoint facilities.

The result was a functional -though slow- VxWorks shell environment.

4.3 Porting Step 3: adding Gmicro-specific debug functions

The standard VxWorks debug library was modified so that breakpoints and single-stepping control could be performed by shell commands. Some Gmicro-specific additions were made; for example, an memory-access breakpoint command interface was added.

The result was a fully-functional VxWorks shell environment, suitable for development and debugging.

4.4 Porting Step 4: adding board-specific functions

The hardware-specific portion of the port, the Board Support package, was implemented. The BSP supports hardware interrupts, real-time clocks, DMA, and similar functions.
4.5 Porting Step 5: adding network support

The following step added support for VME Ethernet boards, namely CMC ENP10/L and Excelan EXOS 202. These drivers support the BSD networking suite from `rlogin` to Network File System (NFS). Afterwards, VxWorks network data transfers occurred at full speed.

4.6 Porting Step 6: building BootROM

The next-to-last step was to modify the portable BootROM package for the H32SBC and Ethernet configuration. Once the BootROM package was complete, VxWorks booted automatically on the target hardware using Ethernet.

4.7 Porting Step 7: optimizing and tuning

During prior steps, no attention was given to code efficiency. The final step of the port was code optimization by using both specialized Gmicro instructions and kernel tuning.

The Gmicro instruction set is well-suited to optimizing task priority ordering functions in the wind kernel. The kernel maintains an ordered priority queue so that no sorting time is required at the time of a task switch. The result is task-switching performance that is constant with respect to the number of enqueued tasks.

Three Gmicro instructions were used to optimize this mechanism: the BSET instruction, which sets a bit in a bitfield array; BCLR, which clears a bit; and BSCH, which searches for a specified bit in a bit array.

The wind task manager locates the highest priority task by searching through two levels of priority bit maps. Task are assigned a priority of 0 (highest) to 256 (lowest); there may be one or more, or no tasks waiting at a given priority. The first-level bit map is a 32-bit word, `metaBMap`, within which an asserted bit indicates that an eligible task exists within a priority group. The highest priority task group is located, at the first level, by executing BSCH to locate the most significant asserted bit in `metaBMap`:

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>010010000000000010000000000000001</td>
<td>31</td>
</tr>
</tbody>
</table>
```

Within `metaBMap` an asserted bit indicates that an eligible task exists at a specific priority. The priority value is located by executing BSCH to locate the most significant asserted bit in the indicated byte cell. In this case, the map indicates a task at priority level 14.

The final step is to combine the indexes from both BSCH operations to select one of 255 linked lists, within which the tasks at each level are maintained. Figure 2 illustrates the array of lists. In this example, there are two tasks waiting at priority level 14: the one at the head of this list will be executed first. There is one task waiting at the lowest priority level, 255, and no tasks waiting at the highest priority level.

The orthogonal register set of the Gmicro design provide further benefits to the wind kernel. Task switching performance depends significantly on the time required to save active registers from the current task and retrieve those in the Task Control Block in memory belonging to the new task.

In this illustration, at least one task in group 1 (priority levels 8 through 15) is active, and there are additional tasks in group 4, 14, and 31. The second-level bit map search locates specific priority levels containing at least one active task, and it is conducted through a 32-byte table, `bmap`:

```
<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>07</td>
</tr>
<tr>
<td>00000010</td>
<td>07</td>
</tr>
<tr>
<td>00000000</td>
<td>07</td>
</tr>
</tbody>
</table>
```

Within `bmap` an asserted bit indicates that an eligible task exists at a specific priority. The priority value is located by executing BSCH to locate the most significant asserted bit in the indicated byte cell. In this case, the map indicates a task at priority level 14.
Figure 2. Prioritized task data structure

Gmicro's sixteen general purpose registers, along with the Program Counter and Process Status Word, nicely contain all task context information required by the kernel. The frame pointer and stack pointer registers are —by convention— the last two general purpose registers, so the result is a very short instruction sequence for loading a new task context:

```
push @(WIND TCB PC, r0)
...  
push @(WIND TCB PSW, r0)  
/* load register set: */  
ldm @(WIND TCB_REGS, r0), (r0-r14)  
/* enter new context: */  
reit
```

The Process Status Word is restored and then the LDM “load multiple” loads 15 registers at once. The program counter is restored by the REIT “return from exception interrupt.”

In addition, the LDCTX “load context” and STCTX “store context” instructions might lead to further kernel performance improvements over register-by-register transfers, but these instructions were not used in the current implementation.

The number of volatile registers is important to performance optimization. The GNU compiler provides the main mechanism for this choice; corresponding adjustments to the kernel are required. The main issue is interrupt latency versus task throughput. For the initial implementation the number of volatiles for Gmicro was set to 6—for comparison, the typical setting for the 680x0 processor family is 4— but future work in this area may be productive. It would be interesting to vary the number of volatile registers, tuning system performance to trade off minimal interrupt latency versus best task throughput.

4.8 Porting Step 8: testing

During this phase, standard target-independent demonstration and training programs were recompiled and tested in this configuration. In addition benchmark programs were run to test kernel and network performance.

4.9 Porting results

A similar development program for Gmicro/100 on the Mitsubishi M32/100 board was completed shortly after the Gmicro/200 port. In the process of doing both these ports, the following issues were noted:
EMS Monitor

The EMS monitor provided very good support. It is especially flexible because it maintains, memory allocations for its own use that can be separated distinctly from user allocations. Using EMS monitor calls, especially for console I/O, was very useful in the early stages of the port. The opportunity to switch between a user bootROM and EMS by changing one jumper was also helpful.

• Gmicro string move instructions

Using the Gmicro SMOV instruction provided a significant performance improvement for block moves. In the code that follows

```c
#if defined(USE_STRING_INSTRUCTION)
    smov/n/f.w
#else
    bra cFwdChk0
    mov.w @r0, @r1
    add #4, r0
    add #4, r1
    cFwdChk0:
    scb #1, r2, #0, cFwdTop0
#endif
```

the alternative that uses the SMOV instruction runs approximately 5 times faster.

• Gmicro Queue Instruction

The Gmicro queue instructions QINS and QDEL were used to implement library functions _insqe —insert a node into a linked list— and _remqe —remove a node from a linked list. The Gmicro QSCH queue-search instruction was not used due to differences between the standard implementation of list terminators in VxWorks and the requirements of the instruction.

5.0 Interactive Development I — Basics

The components of the VxWorks development environment—the kernel and user-supplied host facilities—are controlled from the VxWorks interactive shell. The shell may be accessed directly through a serial console terminal connection, or indirectly via the host network.

The shell shares characteristics of conventional UNIX shells, generalized monitor/debuggers, and the C language. The shell prompt is “->”. The following examples demonstrate the range of the VxWorks shell operations:

```
-> (14 * 9) / 3 (calculation)
value = 42 = 0x2a = 'x'

-> j = 3 (symbol creation)
new symbol "j" added to symbol table

-> k = j + 5 (calculation using variables)
value = 8 = 0x8

-> errnoGet() (invocation of VxWorks function)
value = 0 = 0x0

-> testfnco (invocation of an application function)
value = 25 = 0x19
```

Predefined VxWorks functions and —once they are loaded— application functions are coequal. Invoking a function may be considered as a command to VxWorks or the application, or as a debugging access to a subroutine function, or a combination of both. Any function may also be invoked by a user program or spawned as a task.

The following subsections describe several VxWorks commands and give a detailed example of using the shell. Source code for many of the functions described below is contained in the file `usrLib.c`, which is supplied with VxWorks for the purpose of encouraging user customization.
5.1 *ld* – VxWorks linking loader

The function command *ld* loads object (a.out) modules into target memory. For example, the unlinked Gmicro/200 object module “ex.o” is loaded into target memory by the following command:

```
-> ld() < myhost:/usr/sam/ex.o
```

The file “ex.o” resides on the specified network host. The VxWorks loader

- loads the module’s code and data segments into memory
- relocates the module as required
- resolves external references with the current memory-resident target symbol table
- adds the module’s symbols to the target symbol table.

Symbols in the module –identifying data objects and function routines– become shell-accessible immediately. If the module is changed, recompiled, and reloaded, the latest definitions take precedence; prior definitions are still accessible, however, by address reference.

If “ex_main” is an entry address of a C language routine in “ex.o”, it can be spawned as a task simply by entering

```
-> sp(ex_main)
```

In this case, default parameters are supplied for the task name, priority, and stack size.

5.2 *checkStack* – stack status display command

The function command *checkStack* displays a snapshot summary of task stacks. For example,

```
-> checkStack
```

generates a display of the form shown in Figure 3. This command examines the task stack usage.

The display references each defined task by name, entry point, and Task IDentifier (TID). For each task, it gives the task total stack allocation size (SIZE), the current number of bytes used (CUR), the maximum number of stack bytes used (HIGH), and the number of bytes never used (MARGIN).

The shell provides a command, *period*, to repeat commands every n seconds. The command

```
-> period 5, checkStack
```

spawns a copy of checkStack every 5 seconds. Alternatively, the system call call used in *checkSruck* to get information about a task, *taskInfoGet()*, may be directly invoked at the shell level, or called by a user program, or invoked by a program running on a network node via a Remote Procedure Call.

5.3 *spy* – task activity monitor command

The function command *spy* generates a dynamic report of task status. For example, the command

```
-> spy 10, 200
```

generates a report such as given in Figure 4.

![Table of Task Statistics](image)

<table>
<thead>
<tr>
<th>NAME</th>
<th>ENTRY</th>
<th>TID</th>
<th>SIZE</th>
<th>CUR</th>
<th>HIGH</th>
<th>MARGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>tExcTask</td>
<td>_excTask</td>
<td>fc624</td>
<td>2988</td>
<td>148</td>
<td>676</td>
<td>2312</td>
</tr>
<tr>
<td>tLogTask</td>
<td>_logTask</td>
<td>fb0dc</td>
<td>4988</td>
<td>152</td>
<td>233</td>
<td>4755</td>
</tr>
<tr>
<td>tFtpdTask</td>
<td>_ftpdTask</td>
<td>edc30</td>
<td>5528</td>
<td>176</td>
<td>368</td>
<td>5160</td>
</tr>
<tr>
<td>tPortmapd</td>
<td>_portmapd</td>
<td>ef118</td>
<td>4528</td>
<td>332</td>
<td>2612</td>
<td>1916</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td></td>
<td></td>
<td>1000</td>
<td>0</td>
<td>284</td>
<td>716</td>
</tr>
</tbody>
</table>

*Figure 3 checkStack example output*
Based on the command parameters in this example, the report is updated every 10 seconds on the basis of 200 samples/second.

The display references each defined task by name, entry point, and Task IDentifier (TID). For each task, the display gives the task priority setting (PRI), CPU activity since the invocation of spy in percent and in clock ticks, and the CPU activity since the last display update in percent and in clock ticks.

The spy command operates by spawning a task to provide periodic task activity reports. Data is gathered by an interrupt-level routine connected to a hardware-driven clock interrupt.

5.4 malloc – memory allocation

The routine malloc – part of a suite of memory management functions available – allocates a memory block of the requested size and returns a pointer to the block. These functions are equally available to target application programs and to the user at the shell level.

Memory allocation is performed by traversing a queue of free block descriptors until a block of at least the required size is located. In VxWorks, this potentially lengthy process is logically placed in the context of the caller, not the kernel. Figure 5 illustrates the implementation. Access to malloc is by subroutine call. The first step of the subroutine is to obtain the malloc mutual exclusion semaphore (semTake); the final step is to release the semaphore (semGive).

The time spent in the kernel context is critical, since all other processes are locked out during this time. VxWorks provides a method for measuring time intervals corresponding to function calls, and this method can be used for evaluating these important intervals. The first step is to create a dummy semaphore, called “sem”:

```c
-> sem = semCreate()
new symbol "sem" added to symbol table.
sem = 0xe3870; value = 931968 -
0xe3880 = sem + 0x10
```

The timex command function can be used to time a single function invocation, as follows,

```c
-> timex(semGive, sem)
timex: execution time too short to be measured meaningfully in a single execution.
```

but in this case the function executes too quickly to be measured. The timexN command function repeats calls until enough samples are taken to establish timing to an accuracy of better than 2%, as follows,

```c
-> timexN(semGive, sem)
timex: 58275 reps, time per rep = 11 +/- 0 (0%) microsecs
value = 58 = 0x3a = :'
```

The result is that semGive requires 11 µSec on Gmico/200. (The H32SBC runs at 20 MHz with 1 memory wait state.)

VxWorks allows the combination of up to four functions to be timed. To chose the first function, the timexFunc command is executed as follows:

Figure 4   spy example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>ENTRY</th>
<th>TID</th>
<th>PRI</th>
<th>total % (ticks)</th>
<th>delta % (ticks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tExcTask</td>
<td>_excTask</td>
<td>fbb58</td>
<td>0</td>
<td>0% (0)</td>
<td>0% (0)</td>
</tr>
<tr>
<td>tLogTask</td>
<td>_logTask</td>
<td>fa6e0</td>
<td>0</td>
<td>0% (0)</td>
<td>0% (0)</td>
</tr>
<tr>
<td>KERNEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERRUPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
functions; subsequent unused arguments are filled with dummy zero values.

A defined function sequence is accessed, as a default, by \texttt{timexN()}, as follows:

\begin{verbatim}
  -> timexN()
  timex: 20812 reps, time per rep = 24 +/- 0 (0%) microsecs
  value = 58 = 0x3a = ': I
\end{verbatim}

This indicates that—in this preliminary implementation—the combination of \texttt{semTake} and \texttt{semGive} consumes 24 \( \mu \text{Sec} \). By subtraction, \texttt{semTake} requires 13\( \mu \text{sec} \) on the target Gmicro hardware.

This result shows that other processes are locked out for a very small small amount of time. VxWorks library calls, in general, are implemented similarly.

The problem of “variable system call cost” has received some recent attention in the TRON literature [1] [2], especially pointing out the difficulty of achieving a finite task switching latency, and specifically with respect to evaluating CTRON kernels.
5.5 Using the shell: a shell script

VxWorks command functions can be combined into scripts to perform complex series of commands. During the port to Gmicro/200, the following script was used to download bootROM code images:

```bash
fd = rcmd("tama",514,"hdei", "hdei", "cat ~/vw.new/bootrom", 0)
buf = malloc(100)
read (fd, buf, 32)
size = *(buf + 4) + *(buf + 8)
buf1 = malloc(size)
fioRead(fd, buf1, size)
bcopyBytes (buf1, 0xc00000, size)
close (fd)
free (buf)
free (buf1)
```

Execution of this script takes a few seconds, and results in a code image transfer from the host file server to RAM mapped in from the emulator at address 0xc00000.

The first line of this script establishes the identifier `fd` in the target system symbol table, and assigns to it the result of the function call `rcmd`.

```bash
fd = rcmd("tama",514,"hdei", "hdei", "cat ~/vw.new/bootrom", 0)
```

The `rcmd` call establishes a logical source for the image, in this case, a file on the network host named `tama`. The second line

```bash
buf = malloc(100)
```

creates a temporary buffer in the target, and

```bash
read (fd, buf, 32)
```

reads the first 32 bytes -the `a.out` format execution header- of the image into the buffer. The fourth line

```bash
size = *(buf + 4) + *(buf + 8)
```

access the temporary buffer to find and combine the size of the object text and initialized data to determine the actual object image size. (The second and third 32-bit words in the header indicate the size of the image text and initialized data segments.) The identifier `size` is automatically established and becomes available for subsequent use. The shell always assumes 32-bit integer values.

The following two lines

```bash
buf1 = malloc(size)
fioRead (fd, buf1, size)
```

establish a buffer of the required size for the BootROM image, and transfers the image to that buffer. Then

```bash
bcopyBytes (buf1, 0xc00000, size)
```

moves the image to uninitialized emulator-mapped RAM at physical address 0xc00000.

The final three lines of the script take care of housekeeping chores: closing the logical file and freeing the buffers.

After the script concludes, control returns to the shell command level. In practice, the next step was to spot-check the ROM image at several important addresses to make sure the image was correctly built. At this point, all VxWorks debug facilities are available, including a display memory utility and a disassembler.

Next, the Gmicro/200 EMS monitor was restarted, employed to download the new bootROM RAM image, and used to start execution at its entry point. If the image is correct, the sign-on message of the new, completed BootROM appears.

5.6 Remote debugging with Gmicro/200

During real-time application development, not to mention kernel development, it is assumed that the developers must be in close physical proximity to the development target hardware - if, for no other reason, to be able to use the system hardware reset button when a crash occurs. This necessity is reduced significantly by the combination of extensive exception detection facilities of Gmicro and the facilities in VxWorks to handle exceptions. The result is that the system recovers from errors very well, the number of “hard” crashes is significantly reduced, and it becomes much more practical to consider doing remote software development.

6.0 Interactive development I

The motivation for VxWorks is completely pragmatic: the resources on development targets are often frustratingly inadequate to the problems of developing complex real-time target applications. Since targets are necessarily tailored to application needs, it is usually impractical to add extensive resources directly to targets.

On the other hand, minicomputers, mainframes, and workstations running UNIX provide environments rich in resources and tailored to program development. The availability of cost-effective networking technology
supports a key connectivity component of VxWorks. The minimum hardware addition to target systems—a network card—provides a connection to the resource-rich host environment.

The second key component of VxWorks is the wind real-time kernel. Kernel functions have been deliberately and carefully kept to a bare minimum [3], and these functions are designed as building blocks to support higher level VxWorks functions in a hierarchical design.

The third key component of VxWorks is a set of tools to solve the practical problems of real-time development. Many of the VxWorks tools have been designed to meet specific development needs; many others serve to bind the VxWorks components together with resources in host environments. These resources, including text editors, assemblers, compilers, linkers, and so on, are independent of VxWorks and may be chosen accordingly.

Implicit in the VxWorks design is that interactive development is not just a matter of interacting with the target environment, but of interacting with—ranging and extending—the development environment as well. Further, in VxWorks the distinctions between the target and the host development environment have purposely become blurred.

It might be observed that VxWorks is based on well-known and well-implemented concepts of “distributed data,” and further, that VxWorks implements “distributed control,” with distribution both over network topology and over the time interval of system development.

Why is interactive development so important?

In the TRON design, an interactive model, TULS (TRON Universal Language System) [4], and its implementation, TACL (TRON Application Control-flow Language) [4] [5] [6] overlie the major design components—BTRON, CTRON, ITRON, and MTRON—of the overall concept. Programmable interfaces allow functions to be partitioned among subsystems at run time, i.e., after development has been completed. Programmable interfaces support dynamic load balancing. Without a programmable interface, static interface specifications and version update difficulties will stifle the large and very large networks of cooperating computers envisioned [7].

In particular, these considerations envision creation of a TULS/TACL-based User Interface Management System/User Interface Development System (UIMS/UIDS) that supports prototype development, creation of correct programs that follow system guidelines, and end-user programs.

Under VxWorks, the need to distribute functions among subsystems begins at the inception of the development environment, and extends through the development cycle. The need for dynamic adaptation to static interface specifications and version updates is no less important in the development phase. Sometimes the best way to understand an interface specification is to ask the system to demonstrate it; following specifications is a key job of a developer, but non-interactive tools provide little support for this. Load balancing—for example, when compilations begin to take too long—is just a matter of switching over to a more powerful host.

In other words, the distribution of control over time extends from the development stage through the life of the system, and this control must be maintained by appropriate interactive, programmable mechanisms—or else the system fails to perform effectively for long.

7.0 Conclusions

This paper describes the VxWorks development environment and real-time kernel, demonstrates some of the VxWorks-based development methodology, and draws parallels to the TRON philosophy.

Acknowledgements

The authors wish to thank Toshio Takata of OKI Electric Industry Co., Ltd. (Japan) for valuable support; Hideki Yoshida and Yasushi Saitoh of the Department of Information Science, University of Tokyo, for supplying the GNU tools for Gmicro; Richard Stallman and the Free Software Foundation for supporting the GNU concept, and the staffs of Wind River Systems, Inc. (U.S.) and Wind River Systems, K.K. (Japan).

VxWorks and wind are trademarks of Wind River Systems, Inc. Gmicro is a trademark of the Gmicro Group for the TRON Specification Microprocessor. The GNU tools are publicly licensed products of the Free Software Foundation. UNIX is a registered trademark of AT&T Bell Laboratories.

References


