VITRAL - A text mode window manager for real-time embedded kernels

Manuel Coutinho
IST-UTL*
mabeco@comp.ist.utl.pt

Carlos Almeida
IST-UTL
cra@comp.ist.utl.pt

José Rufino
FCUL†
ruf@di.fc.ul.pt

Abstract

This paper presents VITRAL, a multicolor text mode window manager designed for embedded real-time systems. Therefore, timeliness requirements and resource constraints are the main concern. Even though complex graphical environments are not supported, the application can use VITRAL to build a powerful, dependable environment that reports the system state and still provide a friendly interface to the user.

1. Introduction

Most embedded control applications consist of several different tasks that need to be executed in a concurrent fashion and usually have real-time requirements. In some cases these applications need to interact with the real world, performing I/O operations through a set of devices such as sensors and actuators, but they may also need to interact with human users.

Due to the basic requirements of these applications (concurrent tasks, real-time, I/O event handling), multitasking real-time kernels are a fundamental component to support their development.

Standard distributions of most real-time multitasking kernels only have a generic console for user interaction. In that console, there is only one window to which all the tasks must perform their keyboard input/screen output.

In situations where there are several different tasks interfacing with a human user, a friendlier interface is desired. With this objective, we developed VITRAL[1]. The VITRAL (Portuguese word for Stained Glass Window) driver is a simple yet reliable multiple text windows manager for real-time kernels. It is compatible with standard I/O functions (stdio library) where, associated with each window, we can have read operations from the keyboard and write operations to the screen (Figure 1).

Figure 1. Aspect of VITRAL - a simple window manager

Embedded control systems and applications,interacting with the real-world to perform I/O operations (e.g. through sensors/actuators), exhibit, in general, a set of stringent real-time and dependability requirements. This means the system must be able to preserve timeliness even in the presence of disturbing factors, such as the occurrence of faults or transient overload [1].

The console driver is a standard kernel component which calls for the use of I/O temporal protection mechanisms, at least whenever dependability and timeliness are a must. Furthermore, console I/O operations must not jeopardize overall timeliness constraints, thus requiring time bounded functions.

In the console driver, the input is the keyboard and the output the monitor. The keyboard is typically an asynchronous device that produces events when a user presses a key. If the number of events increases dramatically, due to a failure, accident or intentional action provoked by a malicious user, the constant processing of these events may jeopardize application timeliness. In order to prevent this potentially fatal overload, temporal protection mechanisms must be added to the system.

The VITRAL window manager addresses all these issues while minimizing the impact on existing applications. It replaces the original kernel console driver, and incorporates a temporal protection mechanism associated with console input.

*Instituto Superior Técnico - Universidade Técnica de Lisboa, Avenida Rovisco Pais, 1049-001 Lisboa, Portugal. Tel: +351-21-8418397 - Fax: +351-21-8417499. This work was partially supported by FCT, through Project POSC/EIA/56041/2004 (DARIO) WWW Page - http://pandora.ist.utl.pt/projects/dario.

†Faculdade de Ciências da Universidade de Lisboa, Campo Grande - Bloco C8, 1749-016 Lisboa, Portugal. Tel: +351-21-7500254 - Fax: +351-21-7500084. Navigators Home Page: http://www.navigators.di.fc.ul.pt.

[1]A preliminary version of VITRAL was presented at a national conference [2]

2. VITRAL

The VITRAL manager is a basic window manager designed for embedded real-time systems. It provides a multicolor text based environment (not a graphic design). The next section presents a general view of VITRAL and its integration with the application.

2.1 The VITRAL approach

VITRAL is a multicolor text mode window manager. Using VITRAL, the application can multiplex its I/O through several windows to improve the overall quality of data presentation and provide a friendlier user interface.

An application task can control window creation/destruction with the definition of some attributes (such as foreground/background color, window dimensions, window title, etc). The task can dynamically associate itself to a window to direct its I/O operations. This way, a task can choose its window at runtime. For example: in an alarm situation the task will redirect its output from its "normal" window to an alarm window. Also, several tasks can be associated to the same window.

The application task can also change dynamically some window attributes, for instance, char color, blinking characteristics, etc, to emphasize alarm or abnormal situations. VITRAL also provides some additional operations, like hide/show window capabilities and specification of user-level control and input parameters.

The presence of a window selection bar provides added flexibility to the user when managing windows, like a hide/show operation or selecting the input window.

For increased compatibility with existent software and decrease in cost development, the application can use the standard I/O library (like printf/scanf functions) without any changes. For data output, the use of a printf (or related) function will automatically direct the string to the window associated with the calling task. Likewise, if a scanf is performed, VITRAL will block the task until a key is directed to the window associated to the task. To improve resource management, VITRAL allows the specification of some window input capabilities. So, if the application does not require input operations for a given window, VITRAL will not reserve the necessary resources.

Integration of existent applications within the VITRAL context has also been addressed. By means of a full-screen window, equivalent to the default console in color and dimensions, **the system automatically sets** the full-screen as each task initial associated window. The output will be identical to the use of a standard console. If the OS(Operating System) does not provide this functionality then it is possible to disable this option, rendering output operations useless until the task is associated to a window.

Along with the offered functionality, VITRAL is designed for real-time embedded systems, which means that it must provide bounded time limits (for example, to create a window, direct a pressed key to a task, write a key to the screen, etc) while using the few resources available. To allow a deterministic time response, overlapped windows are not allowed because of the added complexity of a char writing. In most applications this is not a main concern since VITRAL can easily support more than ten non-overlapping windows simultaneously.

2.2 VITRAL concepts

In this section we present some concepts related with the implementation and design of VITRAL. The most basic element of VITRAL is the **character**. The character is the fundamental printable element. It is also a **cell** attribute, composed as well by a background and foreground colors, brightness definition and a screen location - **coordinate**. The **screen** represents the grouping of all cells.

A **basic window** is a subspace of the screen with a rectangular form, Figure 2. Each basic window has some default attributes (defined at window creation), like the background/foreground colors, which can be changed dynamically (for example, to present an important message) and restored to its original value.

A **window** is formed by two basic windows (Figure 2): one for the title (optional) and another for the body. A task will perform its "normal" output to the basic window representing the body. Figure 2 illustrates these two concepts.

Internally, the VITRAL manager receives several messages containing information to be presented to the screen and decides which of the available data is going to be written first. Consequently, only one window is being updated at any given time. This window corresponds to the **current window**.

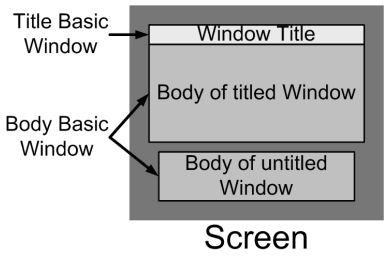

Figure 2. Window and Basic Window concepts

Likewise, only one window at a time is able to receive pressed keys from the keyboard: **active window**. The user can select the active window using a combination of "special keys" designated by **hot keys**. Hot keys allow the selection of the active window and other operations like hide/show a window.

2.3 VITRAL Functional Architecture

Using the standard console, the application typically interfaces with the OS and underlying hardware through the standard I/O library - the ANSI C Library. While the primitives provided by this library are sufficient for

text based applications, window management is not incorporated. A programming extension (VITRAL API) is needed to provide support for application development. Figure 3 illustrates the integration of the OS and VITRAL with the application, while maintaining the existent ANSI C functionality.

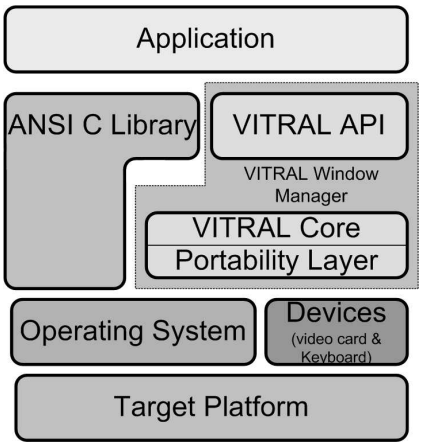

Figure 3. VITRAL functional architecture

The application interfaces with the standard I/O library to perform regular I/O functions, such as a printf or scanf call, and accesses window management through the VITRAL API. Through a portability layer, the VITRAL core is capable of interfacing with several distinct OS and devices, as long as they provide some basic functionality.

3. VITRAL Architecture

The VITRAL manager is implemented at a driver level - Figure 4. Even though a driver level implementation requires a more complex system design and overall knowledge, it increases the application compatibility with existent libraries, like the standard I/O library (stdio), and allows a more controlled environment needed in real-time systems.

The ANSI C library provides a standard interface to the OS core through which the application can access the console driver (VITRAL or another driver) through "well-known" functions, such as printf/scanf. However, due to the lack of flexibility of this library, VITRAL provides an API from which the application can perform an added set of functions, such as window creation and task/window association.

On the bottom of the architecture, VITRAL provides a portability layer that allows its integration with several OS and devices (as long as they support some basic functionality). The ANSI C library, more specifically the stdio, communicates with the console driver through a standard interface supplied by the OS. This interface includes driver search functionality and a set of primitives through which read/write operations are performed. An OS portability layer translates the information to be recognizable by the VITRAL Core. The VITRAL Core transfers the

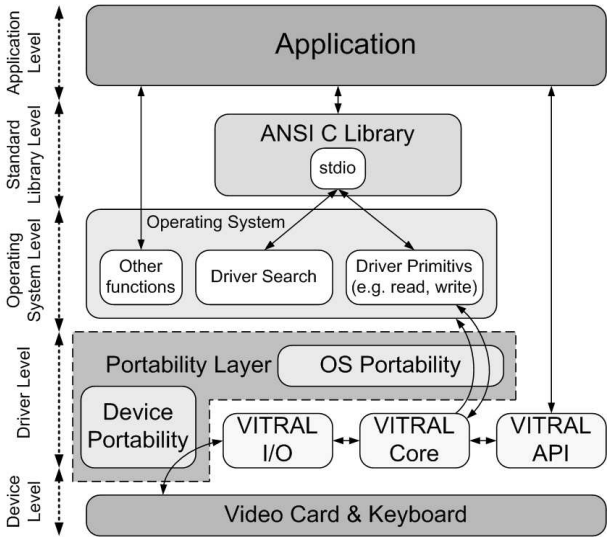

Figure 4. VITRAL architecture and integration

information to the I/O module which communicates with the hardware platform through a device portability layer, that directly accesses the video card memory or the keyboard hardware. The following section describes in more detail the VITRAL Core.

3.1 VITRAL Core

The VITRAL core integrates several components that manage window functionality. It establishes data output management functionality and window creation; synchronizes hot keys events; provides input management; supports interrupt overload detection and recovery module. This section is divided into these four aspects.

Data Output Policies

As mentioned earlier, a task is associated, at a given time, to a window. When the application tasks perform output operations (such as a printf call), the VITRAL manager receives several messages and decides which window is going to be updated - current window.

To synchronize the output from several different tasks, a possible solution would be to create a server task which receives messages and sends (or not) a confirmation that the output was written to the screen, Figure 5. The current window corresponds to the window being processed by the server task. This solution increases the system modularity, since all messages arrive at one point, but has some disadvantages:

- High Complexity
- Low Efficiency
- Absence of Relative Urgency between Messages

The need for inter-process communication increases the complexity of the system while reducing the efficiency

because of the context switch overhead. Also, since messages are treated in a FIFO order, an urgent message may have its response delayed due to a temporary burst of low **priority** messages.

Generally, the system designer desires that messages have a degree of urgency directly related with the task priority. This can be implemented by assigning a priority to each message, which requires complex message systems. Another solution would increase the server priority using inheritance protocols [10], making it similar to a shared resource. Either way, in the worst case, urgent messages have to wait for one low priority message to finish (similar to a non-preemptive system).

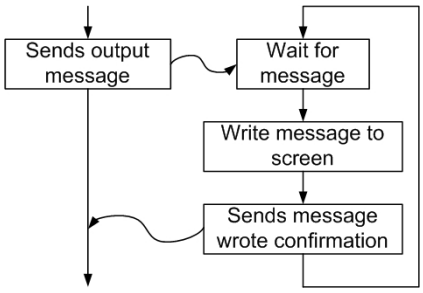

Figure 5. Output server task

Instead of creating a server task, VITRAL implements a decentralized approach, were the output is performed in the context of the calling task. Therefore, higher priority messages can preempt lower priority outputs. Under this implementation the current window is the window associated with the running task, making the OS scheduler the current window scheduler as well. Note that while the output is done in the context of the calling task, there are other operations, described later, that will require the use of a coordinator task.

The output server approach considers all the memory of the video card to be a multiple access variable. In the decentralized approach, by dividing the video card memory into regions, defined by the windows dimensions, each task has a reserved memory space to where output operations are performed [2]. This is an advantage of building such a manager at driver level, since it is possible to analyze the problem at a much lower design level. This solution allows the output to have the same priority as the calling task and has no need for the server task, which lowers the required resources and improves performance since no task context switches are performed.

When the cursor arrives at the last window position, a scroll operation is needed. VITRAL allows windows to shift all lines upward (like a standard window) or to replace the first line and start downwards from there. This increases the system flexibility since a shift operation can be very time consuming (most video cards do not have hardware speedups for a scroll operation unless it is for a full-screen window).

[2]When two (or more) tasks are associated to the same window, synchronization between them may be necessary, but it is the responsibility of the application to maintain a coherent output.

Window Management

The creation of windows and management of hot keys needs to be synchronized since hot keys need the information of what windows they can manipulate (hide/show window, change active window, etc). Hot key processing can not be done inside the keyboard ISR **(Interrupt Service Routine)** because it can take a long time - a hot key can reach $9.4ms$ of processing time. Hence, a coordinator task which controls window management and hot key events is needed.

The implementation of a coordinator task arises some decisions: coordinator task priority; relative priorities between hot keys and creation of window messages. For example, making the coordinator task priority too low will cause a high hot key response time, while making it too high may cause a high blocking time for lower priority tasks. Furthermore, since the application can define a low priority for this task, a priority inversion scenario may emerge [10]. Suppose a high priority task sending a window creation message and blocking until it receives a confirmation. If a medium priority task preempts the VITRAL task, the confirmation will be delayed, making the high priority task block for an additional amount of time correspondent to the execution time of the medium priority task.

This is a complex synchronization problem since the coordinator task has to manage between events that arrive from an interrupt driven source and from a task, where priority inversions scenarios can occur. This algorithm can not use semaphores because block commands can not be issued inside an ISR. Instead, an elegant solution is provided by an upgrade of the VITRAL Task priority to a predefined ceiling [10] (see Figure 6).

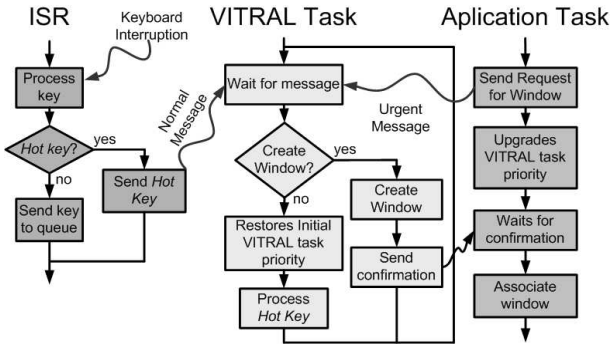

Figure 6. VITRAL functional description: processing of hot keys and creation of windows

To establish a maximum window creation time, the system must provide a limited interference due to hot key processing. Hence, VITRAL establishes hot key messages to have lower priority than window creation **messages**, as illustrated in Figure 6. Case these two types of

messages had the same priority, an overflow of hot key events would cause a high (unbounded) response time for window creation requests. Window creation requests take precedence of hot keys messages in the message queue which ensures that, in the worst case, a window creation request has to wait for one hot key processing time.

Since window creation is typically done at system initialization time, hot key processing is generally not affected by this differentiated service. Nevertheless, if the application desires higher priority processing for hot keys, the system can dynamically reverse the messages priority.

In regards to the flowchart in Figure 6, notice also that restoration of the VITRAL Task priority is not done by the application task, otherwise the system could not establish a maximum window creation time. Take for example a scenario where there are several hot keys in queue and a medium priority task sends a window creation message. Restoration of the priority by the application task would imply that all hot keys are processed (because VITRAL Task has the highest priority) and only then would the application task resume its execution.

After a window has been created, the application task will associate itself to that window. This can be done by defining a variable on the TCB (Task Control Block) for each task that points to the associated window.

Input Management

Normal keys (not hot keys) are multiplexed between several application windows. The active window is determined by the use of hot keys pressed by the user (CTRL-TAB), Figures 6,7. The VITRAL Task, during hot key processing, determines which window is the active window.

Inside the keyboard ISR, a message containing the pressed key is sent to the active window input queue. There may be several tasks waiting for a pressed key under the same window. VITRAL allows the programmer to specify the order in which the tasks receive the pressed keys: FIFO order; priority-based order. Under the FIFO order the first task that performs a reading operation (e. g. scanf) will receive the pressed key. Under the priority-based order, it is the task with higher priority.

Both the keyboard ISR and the VITRAL task access the active window pointer so an interrupt disable must be done when accessing this variable.

In order to minimize space and resources, the application can define, at window creation time, if the window allows keyboard input and the queue buffer dimension.

Temporal Protection Mechanisms

Another fundamental issue concerns the integration of temporal protection mechanisms regarding keyboard operation [1]. In order to detect an interrupt overload such as a stuck key and preventing it from interfering with the system timeliness, the keyboard interrupt service routine is extended with a component that controls the interrupt

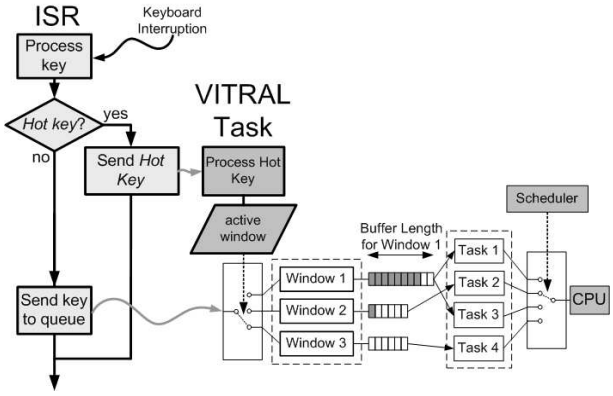

Figure 7. Keyboard input multiplexing through active window

rate before calling normal key processing functions. If an overload is detected, keyboard interrupts are disabled and the system switches to a polling mode. A polling task reads periodically the value from the keyboard, determining if a failure is still present. If not, the system returns to the normal state by enabling again keyboard interrupts.

This approach allows the use of discrete signal processing methods, which we apply to the case under analysis, i.e. the keyboard interrupts. In particular, we use a simple IIR (Infinite Impulse Response) filter [1].

3.2 VITRAL API

As it is represented in Figures 3 and 4, the application interfaces with the keyboard/screen through the standard library (stdio) and an extended set of functions that provide window management capabilities (VITRAL API). In particular, the VITRAL API provides window creation/destruction/association and attributes set/restore methods:

```
Window createWindow(Coordinate ul, br; Color bg, fg;
                    String title; Byte flags)
Status associateWindow(Window w)
Status setAttributes(Color bg,fg)
Status restoreAttributes()
```

The *createWindow* method creates a window with the definition of an upper left and bottom right corners, background and foreground colors, title (optional) and some flags representing the window input and scroll functionality.

The *destroyWindow* method destroys the window passed as the argument and returns the status of the operation. The *associateWindow* method associates the current task to the window passed as the argument. The *setColor* and *restoreColor* methods allow changing/restoring dynamically the color attributes of the output data.

In turn, the application has to provide some variables that characterize the VITRAL manager operating mode:

1. VITRAL Task base priority
2. Input queue parameters
3. Priority ceiling of all tasks that require window creation
4. Parameters of the interrupt overload module

The first item refers to the base priority of the VITRAL task. If set too high, hot key processing has low response time but blocks lower priority tasks. On the other hand, **if** it is set to a low level, hot key processing has a high response time.

The second parameter refers to the way in which two (or more) tasks associated to the same window catch pressed keys (either based on their priority or by a FIFO order).

The third item represents the priority ceiling of all tasks that may perform a window creation request. This is part of the hot key and window creation management synchronization algorithm, described before.

The last item regards the interrupt overload module (optional). This module requires the definition of some variables, such as sampling rate, upper and lower threshold, polling period and polling task priority [1].

3.3 VITRAL Integration

VITRAL was originally made for the RTEMS (Real Time Executive for Multiprocessor Systems) Operating System [6], but it was soon concluded that it possesses a high portability level, being possible to port to other real-time multitasking kernels such as eCos (embedded Configurable operating system) [9]. There are currently well-defined modules that separate the integration of VITRAL with the OS and the devices used (keyboard and video card). However, there are some basic requirements that the OS must be able to provide. For example, VITRAL requires: means of communication and synchronization - as message queues and semaphores; dynamic task priority assignment; differentiate urgent from non-urgent messages.

If an OS does not directly provide such mechanisms, the portability layer should be extended with additional support for the missing components (built from the OS basic functionality).

4. Timing Properties and Resource Usage

Integrating VITRAL within a real-time system requires that maximum waiting times can be established. We tested our VITRAL version in a PC Intel 486 16 MHz with 4 MiB [3] of RAM memory under the real-time operating system RTEMS (Real Time Executive Multiprocessor System). The time intervals observed depend obviously on the hardware used but also on the Operating System since system calls are made inside VITRAL. During these tests no higher priority tasks are assumed to be ready.

The maximum interrupt disable time interval observed inside VITRAL was $3\mu s$. This is due to the multiple access of the active window pointer by both the keyboard ISR and the VITRAL task. In a similar platform (a faster one) RTEMS results show that it has a maximum critical section of $13\mu s$ [7]. Thus VITRAL does not increase the critical section.

The interrupt overload protection mechanism (which is optional) increases the maximum interrupt latency by $3\mu s$.

For a window creation time we observed a maximum of $10.4ms$. It was concluded that the display initialization (background color establishment) is responsible for most of the consumed time. This test corresponds to the full-screen window creation (maximum size).

The worst time for a key to be ready for processing by the application is the sum of the original OS interrupt latency (that can be be increased if the interrupt overload protection mechanism is used), some keyboard ISR processing, a message_send and a message_recv calls ($25 + (3) + 7 + 144 + 127 = 303(306)\mu s$). This is the time needed for an application task (or the VITRAL server task) to receive a key. The time needed by the VITRAL task to process a hot key varies between a hide/show window (where it can reach $9.4ms$ for a full-screen window) and other special keys (maximum of $1.2ms$).

Allowing existent applications to use VITRAL (without any changes to their functioning) is made by the use of extensions, defined by RTEMS, installed at key points of the preemption. During task creation, a VITRAL handler is automatically called, which associates the task to a full-screen window. This increases the task creation time by $3\mu s$.

In regards to memory consumption, VITRAL demonstrated an increase of 67 KiB for data storage and approximately 5 KiB of source code. As expected, the VITRAL requirements are far less than those of graphical environments.

5. Related work

There are many graphical window managers available. The most popular environments are: NanoX (formerly MicroWindows) [3] [5], openGUI [8], Qt/Embedded [11] and MiniGUI [4]. These technologies are elaborate window managers. They possess several features which are not available in VITRAL, such as graphical environment support, overlapped windows and several other enhanced features.

However, in embedded control systems, two key characteristics concern the timeliness properties and memory/resource consumption. Of the presented technologies only MiniGUI claims to be designed for real-time systems. It has, nevertheless, undesirable characteristics: a central coordinator task that writes to the screen messages delivered by the application tasks. As described before,

[3] **According to SI standards - International System of Units**

under this approach, messages are treated after a FIFO order and so relative urgencies are ignored. VITRAL has a coordinator task that controls only window creation management and hot key processing, whereas output processing is done in the context of the calling task.

Graphical environments are also a high source of memory and resource consumption. In embedded systems, where resource management is fundamental, VITRAL lowers these requirements due to its text mode characteristic.

6. Conclusions

In summary, the functionality provided by VITRAL enhances the support for the development of real-time embedded applications. Making sure that the kernel intrinsic real-time characteristics are not compromised, and are even extended through the use of specific input/output event handling timeliness protection mechanisms, is extremely important for dependable real-time applications.

The VITRAL window manager provides a driver supporting simple text mode windows, compatible with the standard I/O library functions, where application tasks can dynamically associate themselves to a window, separating their I/O operations from other tasks.

Operating System and device portability is also addressed with the definition and implementation of a module that makes the VITRAL Core transparent to the target platform.

There are currently several other powerful window managers but none offers similar timeliness properties or resource usage as VITRAL. In particular, only MiniGUI claims to support the development of real-time applications but it has some undesirable characteristics. Even though VITRAL does not support graphical environments, the application can still provide a friendly interface to the user.

VITRAL also includes temporal protection mechanisms bounding the interference in the time domain of window management and console handling. This is a significant enhancement of OS kernel functionality concerning a user friendly interface that nevertheless preserves timeliness guarantees.

Acknowledgments

The authors wish to thank Pedro Romano for his contribution to the engineering of an earlier version of VITRAL.

References

[1] M. Coutinho, J. Rufino, and C. Almeida. Control of event handling timeliness in RTEMS. In *Proceedings of the 17th IASTED International Conference on Paralel and Distributed Computing Systems - PDCS 2005*, Phoenix, Arizona, USA, November 2005. IASTED.

[2] M. Coutinho, J. Rufino, and C. Almeida. VITRAL: A text mode windows manager for RTEMS. In *Terceiras Jornadas de Engenharia de Electrónica e Telecomunicações e de Computadores*, Lisboa, Portugal, Novembro 2005.

[3] G. Haerr. The microwindows project: Enabling graphical applications on embedded linux systems. http://www.microwindows.org/MicrowindowsPaper.html.

[4] Minigui white paper. http://www.minigui.org, 2004.

[5] Nano-x programming tutorial. http://www.microwindows.org/Nano-XTutorial.html, 2000.

[6] On-Line Applications Research Corporation (OAR). *RTEMS C User's Guide*, edition 4.6.2, for rtems 4.6.2 edition, August 2003. (The RTEMS Project is hosted at http://www.rtems.com.).

[7] On-Line Applications Research Corporation (OAR). *RTEMS Intel i386 Applications Supplement*, edition 4.6.2, for rtems 4.6.2 edition, August 2003.

[8] Opengui home page. http://www.tutok.sk/fastgl/.

[9] Red Hat, Inc. *eCos Reference Manual*, 2003.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[11] Trolltech. Qt 3.3 whitepaper. http://www.trolltech.com.