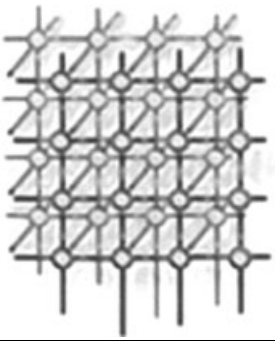
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:995–1006
Published online 10 June 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1428



Process-oriented device driver development‡

F. R. M. Barnes*, † and C. G. Ritson

Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, U.K.

SUMMARY

Operating systems (OSs) are the core software component of many modern computer systems, ranging from small specialized embedded systems through to large distributed OSs. The demands placed upon these systems are increasingly complex, in particular, the need to handle concurrency in order to exploit increasingly parallel (multi-core) hardware, to support increasing numbers of user and system processes and to take advantage of increasingly distributed and decentralized systems. The languages and designs that existing OSs employ provide little support for concurrency, leading to unmanageable programming complexities and ultimately errors in the resulting systems, which are hard to detect, hard to remove and hard to prove correct. This article presents the process-oriented design of a universal serial bus device driver infrastructure for the Raw Metal occam eXperiment (RMoX) OS and its implementation in the occam-pi multiprocessing language. We show how concurrency can be used for the benefit of such systems, simplifying design and implementation, providing freedom from race-hazard and aliasing errors and the potential for guarantees of operating system scalability, reliability and efficiency. Copyright © 2007 F.R.M. Barnes & C.G. Ritson. [Correction made here after initial online publication]

Received 4 February 2009; Accepted 10 February 2009

KEY WORDS: concurrency; operating systems; RMoX; embedded systems; occam-pi; CSP; USB

1. INTRODUCTION

The RMoX operating system (OS) [1], on which the work presented here is based, represents an interesting and well-founded approach to OS development. Concurrency is utilized at the lowest level, with the whole OS comprised of many interacting parallel processes. Compared with existing

*Correspondence to: F. R. M. Barnes, Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, U.K.
†E-mail: f.r.m.barnes@kent.ac.uk

‡Revised version of Ritson CG, Barnes FRM. A Process Oriented Approach to USB Driver Development. In McEwan AA, Schneider S, Ifill W, Welch PH (eds). Communicating Process Architectures 2007. IOS Press: Amsterdam. Published with permission from IOS Press.

This article was published online on 8 May 2009. An error was subsequently identified. This notice is included in the online and print versions to indicate that both have been corrected [18 June 2009].

Contract/grant sponsor: EPSRC; contract/grant number: EP/D061822/1

systems, which are typically sequential, RMoX offers an opportunity to easily utilize increasingly available multi-core hardware. The system is primarily developed in occam-pi [2,3], a language with Communicating Sequential Processes (CSP) [4] based semantics that incorporates ideas of mobility from the π-calculus [5], with guarantees of freedom from race-hazard and aliasing error.

The overall aim of the RMoX OS is to provide an OS that is *reliable*; in that we should have some guarantee about the correct operation of system components and the system overall, *scalable*, both in design and in response to available hardware and the demands of users and *efficient*, using the available resources effectively. Many commonly available operating systems fail to meet one or more of these goals, due in a large part to the nature of the programming languages used to build them—typically procedural low-level languages such as C. These languages offer little or no support for concurrency or formal verification, resulting in systems that can be difficult to design, hard to understand and maintain, and for which formal verification (in whole or in part) is hard to impossible. Concurrency is inescapable for OS designers, necessary at the simplest level for handling interrupt-driven operation and supporting multiple tasks, but increasingly often for exploiting multi-core processors and decentralized systems (e.g. Graphics Processing Units (GPUs) for general-purpose computation [6]).

Although there are a variety of tools available to assist in the verification of C programs, such as BLAST [7] and ACE [8], these require some amount of effort on the part of the system developer as well as an understanding of model checking, neither do such tools help the developer to build the system correctly in the first place. As a result, verifying systems may often require revising the design and implementation, substantially lengthening the time to delivery for a verified system. Systems that have undergone rigorous verification have proved themselves, however, with certified OSs such as Integrity [9] and LynxOS [10] used in real-time critical applications.

The view maintained by this work is that a concurrent, process-oriented approach to design and development can help to overcome the problems currently experienced by a range of systems. A communicating process model of concurrency, with the compositional formal semantics of CSP, allows for *scalable* designs that can be readily understood, and provides routes into formal verification as the system is developed, although these are not considered here. The occam-pi language provides compile-time guarantees of freedom from aliasing and race-hazard error, improving *reliability*, and at the same time permits high levels of *dynamics*, including the reconfiguration of process networks. An *efficient* runtime system permits the scheduling of thousands to millions of concurrent processes across multiple processing cores. Combined, these encourage the use of concurrency as a fundamental design tool for reliable, scalable and efficient systems, and not just as a necessary hurdle in systems development.

The work presented here concentrates on the design and development of a single aspect of the RMoX OS, specifically the *universal serial bus* (USB) [11] driver stack. Supporting this hardware presents some significant design challenges in existing operating systems, as it requires a dynamic approach that layers easily—USB devices may be plugged and unplugged arbitrarily in tree-like structures, and this should not break system operation. The lack of support for concurrency in existing systems can make USB development hard, particularly when it comes to guaranteeing that different third-party drivers interact correctly. RMoX's USB architecture shows how concurrency can be used to our benefit: breaking down the software architecture into simple, understandable, concurrent components, producing a design that is scalable, and an implementation that is reliable and efficient.

DOI: 10.1002/cpe

A high-level overview of the RMoX OS is given in Section 2, with some details regarding its operation and limitations, followed by a brief summary of the USB hardware standards in Section 3. In Section 4 we describe the design and implementation of the USB device driver components within RMoX. Initial conclusions and consideration of future and related work are given in Section 5. Examples of USB device programming within RMoX are not presented here. For these, the reader may refer to our earlier conference publication [12], which additionally contains further low-level details of our implementation, or to the freely available RMoX source-code and documentation (available from `http://rmox.net/`).

2. THE RMoX OS

The RMoX OS is designed and built using layered networks of communicating processes, as shown in Figure 1. Each of the individual components, such as the '`keyboard`' device driver (which could internally be a network of parallel processes), are isolated from one another and scheduled independently. Communication between components is by means of synchronized channel communication, using an efficient non-aliasing pointer passing mechanism for large data items. Communication of channel bundle ends over channels (including themselves) allows the dynamic restructuring of process networks, an idea taken from the π-calculus. Processes in RMoX are organized into client–server networks, whereby clients initiate communication, and in the absence of cycles in the communication graph, this gives guarantees of deadlock freedom [13].

There are three core services provided by the RMoX system: device drivers, file systems and networking. These simply provide management for the sub-processes (or sub-process networks) that they are responsible for (some processes inside the 'driver-core' are shown in Figure 1). When a request for a resource is made, typically via the 'kernel' process, the relevant 'core' process routes that request to the correct underlying device. Using mobile channels [3], this allows *direct* links to be established between low-level components providing a particular functionality and the high-level components using them. Protocols for the various types of resource (e.g. file, network socket, block device driver) are straight forward and well understood in the context of OSs—e.g. a file-system driver (inside 'fs.core') can use any driver that provides a block–device

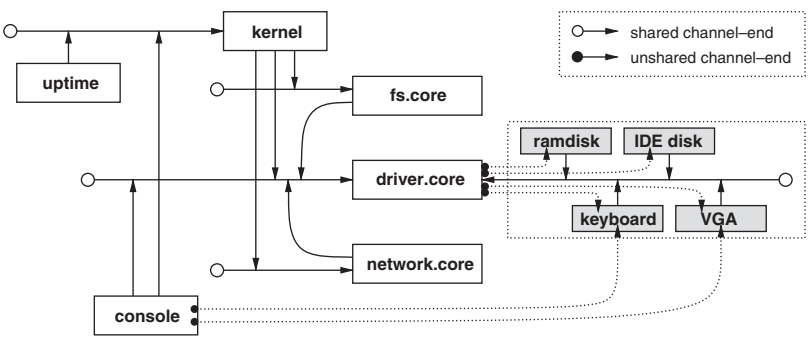


Figure 1. RMoX top-level process network.

DOI: 10.1002/cpe

interface. Since such protocols are well defined, in terms of interactions between processes, building pipelines of processes that layer functionality is no problem. Some consideration must be given to shutting these down correctly (i.e. without inducing deadlock); fortunately that process is well understood [14].

As the system evolves, links established between different parts of the system can lead to a complex process network. However, with guarantees that individual processes interact with their environments in 'safe' ways (with a per-process analysis performed automatically by the compiler), we can guarantee the overall 'safe' behaviour of the system—a feature of the compositional semantics of CSP. This type of formalism is already exploited in the overall system design—specifically that a client–server network is deadlock free; all we have to do is ensure that *individual* processes conform to this. In cases where a communication cycle might occur at runtime, dynamic process creation can be used to create a short-lived 'client' process that breaks this cycle (effectively by providing asynchronous communication).

Although the majority of RMoX is written in occam-pi, and as such is a concurrent system that we can reason about, there remains an amount of C and assembler code, for which we cannot yet make the same claims of correctness. This includes the occam-pi runtime process scheduler (CCSP), a small amount of additional low-level code (providing the memory allocator and other low-level routines), and the occam-pi compiler and native-code translator, which form part of the toolchain. These tools are essentially unchanging with respect to the rest of the system, however, and a large part has been tried and tested within the KRoC occam-pi system [15]. We also have the opportunity to verify these using existing model-checkers such as BLAST and ACE [7,8], although have not yet done so.

3. THE USB

The USB [11,16] first appeared in 1996 and has undergone many revisions since. In recent years it has become the interface of choice for peripherals, replacing many legacy interfaces, e.g. RS232, PS/2 and IEEE1284. The range of USB devices available is vast, from keyboards and mice, through flash and other storage devices, to sound cards and video capture systems. Many classes of device are standardized in documents associated with the USB. These include human-interface devices, mass-storage devices, audio input/output devices and printers. For these reasons adding USB support to the RMoX OS increases its potential for device support significantly. It also provides an opportunity to explore modelling of dynamic hardware configurations within RMoX.

3.1. USB hardware

The USB itself is a 4-wire (2 signal, 2 power) half-duplex interface, supporting devices at three speeds: 1.5 Mbps (low), 12 Mbps (full) and 480 Mbps (high). There is a single bus master, the host controller (HC), which manages all bus communication. Communication is strictly controlled—a device cannot initiate data transfer until it has been offered the appropriate bandwidth by the HC. The topology of a USB bus is a tree, with the HC at the root, providing a root hub to which devices can be connected. Additional ports can be added to the bus by connecting a hub device to one of the existing bus ports. Connected hubs are managed by the USB driver infrastructure, which

DOI: 10.1002/cpe

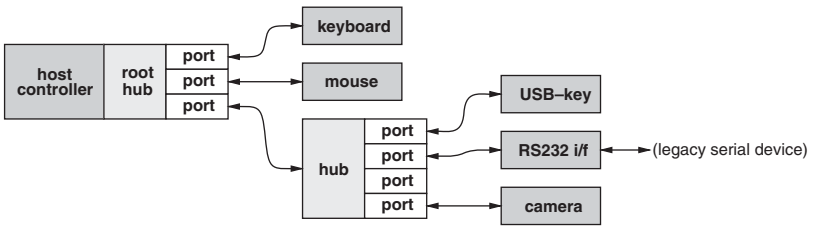


Figure 2. Example USB hardware tree.

maintains a consistent view of the topology at all times. Figure 2 shows a typical arrangement of USB hardware.

Unlike more traditional system buses, such as Peripheral Component Interconnect (PCI) [17], the topology of the USB is expected to change at runtime. For this and the reasons above, access to bus devices is via communication primitives provided by the USB driver infrastructure, rather than directly using CPU I/O commands or memory-mapped registers. Although it should be noted that this difference does not preclude the use of direct memory access (DMA) data transfers to and from bus devices.

3.2. USB interfaces and endpoints

Each device attached to the bus is divided into interfaces, which have zero or more endpoints, used to transfer data to and from the device. Interfaces model device functions, for example a keyboard with built-in track-pad would typically have one interface for the keyboard and one for the track-pad. Interfaces are grouped into configurations, of which only one may be active at a time. Configurations exist to allow the fundamental functionality of the device to change. For example, an Integrated Services Digital Network (ISDN) adapter with two channels may provide two configurations: one configuration with two interfaces, allowing the ISDN channels to be used independently, and another with a single interface controlling both channels bound together.

Individual *interfaces* may also be independently configured with different functionality by use of an 'alternate' setting. This is typically used to change the transfer characteristics of the interface's endpoints. For example, a packet-based USB audio device may have alternate interfaces for different packet sizes, possibly selected by the device driver depending on bus load or other conditions.

Endpoints are the sinks and sources for communications on the bus. Bus transactions are addressed first to a device, then to an endpoint within it. A software structure known as a *pipe* is used to model the connection between the host and an endpoint, maintaining the state information (not entirely dissimilar to the structure and state maintained in a TCP/IP network connection).

There are four different endpoint types defined by the USB standards, which specify how communication on the relevant 'pipe' should be handled. *Control endpoints* use a structured message protocol, sending messages in either direction. These are used generically for enumerating devices not only on the bus, but also for device-specific control. *Bulk endpoints* communicate data on demand, with no particular structure imposed, conceptually similar to the Unix 'pipe'. *Interrupt endpoints* are similar to 'bulk', but exchange data according to a defined schedule. At a set

DOI: 10.1002/cpe

interval, the host offers bus time to the device, which may then respond, possibly with a negative acknowledgement, causing the host to retry at the next interval. *Isochronous endpoints* are similar to 'interrupt', but allow larger packets and do not support the retry mechanism, typically used by devices that can tolerate data loss, such as audio and video.

3.3. Implementation challenges

There are a variety of implementation considerations when building a USB device-driver stack. First, the dynamic nature of the hardware topology must be reflected in the software. Traditional operating systems use a series of linked data structures to achieve this, with embedded or global locks to control concurrent access. The implementation must also be fault-tolerant to some degree—if a user unplugs a device when in use, the software using that device should fail gracefully, not deadlock or livelock.

As USB is being increasingly used to support legacy devices (e.g. PS/2 keyboard adaptors, serial and parallel-port adapters), the device-driver infrastructure developed needs to be able to present suitable interfaces to higher-level OS components. These interfaces will typically lie *underneath* existing high-level device drivers. For instance, the 'keyboard' driver (primarily responsible for mapping scan codes into characters and control codes, and maintaining the shift state) will provide access to any keyboard device on the system, be it connected via the on board PS/2 port or attached to a USB bus. Such low-level connectivity details are generally uninteresting to applications-these expect to get keystrokes from a 'keyboard' device, regardless of how it is connected (PS/2, USB or an on-screen virtual keyboard).

4. SOFTWARE ARCHITECTURE

All device-driver functionality in RMoX is accessed through the central 'driver.core' process (Figure 1), which directs incoming requests (internal and external) to the appropriate driver within. The 'dnotify' device driver, although quite separate from the USB infrastructure, is used to notify processes when new USB devices become available or when existing ones are removed.

The USB driver infrastructure is built from several parts. At the lowest level is a *host controller driver* (HCD) that provides access to the USB controller hardware (via I/O ports and/or memory mapping). The implementation of one particular HCD is covered in Section 4.3. At the next

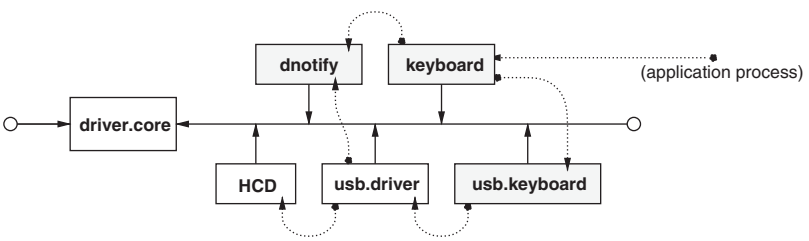


Figure 3. USB device-driver top-level components.

DOI: 10.1002/cpe

level is the 'usb.driver' (USBD) itself. This process maintains a view of the hardware topology using networks of processes representing the different USB buses. The USBD acts as a client to HCD drivers and as a server to higher-level drivers (e.g. 'usb.keyboard'). Figure 3 shows a typical process network setup, using USB to provide an application process with access to the keyboard.

The 'usb.keyboard' process uses the USBD to access the particular keyboard device and provides an interface for upstream 'keyboard' processes. The 'keyboard' process actively listens for newly arriving keyboards (using the 'dnotify' driver) and manages them all together by default—as many existing systems do (e.g. pressing 'num-lock' on *one* of the keyboards causes *all* num-lock states and indicators to toggle).

4.1. USB driver structure

Processes outside the USB driver can gain access to the USB at three levels: bus-level, device-level and interface-level. The 'usb.driver' contains within it separate process networks for each individual bus—typically identified by a single HC. These process networks are highly dynamic, reflecting the current hardware topology. When an HCD instance starts, it connects to the USB driver and requests that a new bus be created. Mobile channel bundles are returned, on which the HC implements the low-level bus access protocol and the root hub. Through this mechanism the bus access hardware is abstracted. Figure 4 shows the process network for a newly created bus, with three connected USB devices, one of which is a hub. Some of the internal connections have been omitted for clarity.

Without concentrating too much on the internal detail of the individual components, it is possible to see in Figure 4 that a hierarchy of process connections exists, starting at the high level with 'usb.keyboard', through 'usb.device' and 'hub.manager', down to the HCD. The 'usb.hub' process converts the abstract hub protocol used by 'hub.manager' into accesses on the hub device's end-points. The root hub, not being an actual USB device, is implemented directly by the HCD, so no 'usb.hub' process is necessary.

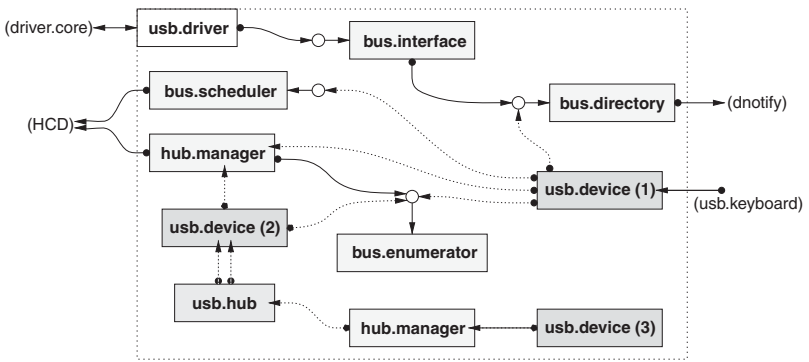


Figure 4. USB device-driver bus-level components.

DOI: 10.1002/cpe

When a new device is connected to the bus, reported by either the root hub or a 'usb.hub' process, it is first enumerated by the appropriate 'hub.manager', which *forks* (dynamically creates) a new 'usb.device' process to manage it. The two processes 'bus.enumerator' and 'bus.directory' maintain the shared state on a bus, i.e. what devices are attached and their unique identifiers. When a 'hub.manager' is notified that an attached device has been disconnected, it shuts down gracefully, notifying the associated 'usb.device' and processing any pending requests.

4.2. USB device structure

Figure 5 shows the internal structure of two 'usb.device' processes, which internally maintain a network of interconnected 'interface' and 'endpoint' processes. With the exception of the default control endpoint, these form the structure described in Section 3.2 and model the hierarchy defined in the USB specification directly as processes. When a device is configured, it dynamically creates interface processes to match those defined in the configuration read from the device. The interfaces in turn dynamically create endpoints to match their current alternate setting. Changing an interface's alternate setting causes the existing endpoint processes to be shut down and new ones created; changing the configuration of the device shuts down and re-creates all interface and endpoint processes.

Device, interface and endpoint processes can each act as servers, giving out the *unshared* client-end of a channel bundle when they are 'opened'. If the device is disconnected, or interface or endpoint processes shut down, the relevant server process continues to respond to client requests (with errors), until that client disconnects. As the USB topology is expected to change during normal system operation, the process network must not only safely grow, but also safely shrink. Using unshared channel-bundles makes this simpler, but devices, interfaces and endpoints can still be shared at a higher-level if necessary; although, this would be unusual given the nature of typical USB devices. Requests to open a particular USB device, interface or endpoint originate within the 'bus.directory' process and are routed to the relevant internal process.

Care must be taken when implementing the main-loop of the 'endpoint' processes, such that the channel from the interface is serviced at a reasonable interval. This is mainly a concern for

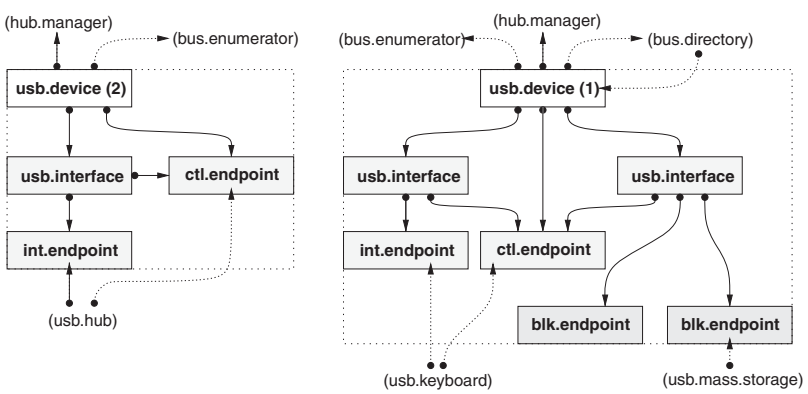


Figure 5. USB device-driver device-level components.

DOI: 10.1002/cpe

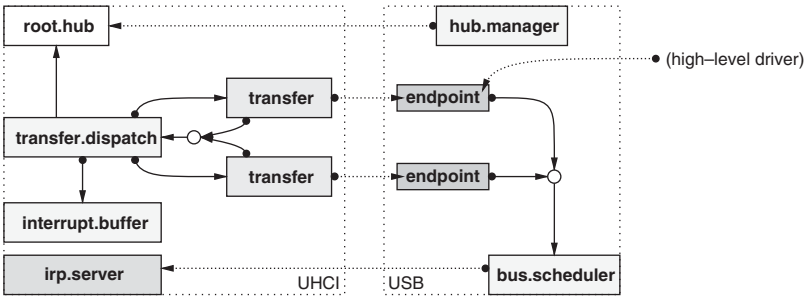


Figure 6. Overview of the 'uhci.driver' host controller driver.

interrupt endpoints, where requests to the underlying bus could wait for a long period of time before completing (due to the lower-level retry mechanism). For all other endpoint types, bus transactions are guaranteed to finish within a short period of time. The consequence of ignoring this detail would be that the system could appear to freeze until some external event (e.g. key press or device removal) occurs, causing a pending interrupt request to complete and the associated endpoint process to resume.

4.3. USB HCs

A number of HC standards exist, implemented by a wide range of hardware (e.g. different PC and PC/104 mainboards). RMoX has drivers for the *Universal Host Controller Interface* (UHCI), Open Host Controller Interface (OHCI) and Enhanced Host Controller Interface (EHCI) standards. The UHCI [18] standard, released by Intel in 1996, is the simplest of these. Figure 6 shows the RMoX implementation of this, alongside the USB driver processes to which it is connected.

The UHCI hardware registers are partitioned between the different processes inside the 'uhci.driver', guaranteeing no shared resource race hazards. To further reinforce this, there are no shared memory buffers; all memory used is *mobile* and is *moved* between processes as appropriate. Memory buffers from high-level clients such as the 'usb.keyboard' driver process are passed directly through the endpoint process into the UHCI 'transfer' process, where they are used for DMA with the underlying hardware. With some small modifications to the occam-pi runtime allocator, to make it DMA aware, an efficient *zero-copy* architecture is created.

5. CONCLUSIONS AND FUTURE WORK

In this article we have presented the design and development of a robust and efficient process-oriented USB driver. Significantly, the software process networks bear an almost *picture perfect* resemblance to the hierarchy presented in the USB standards and the network that exists between physical devices. Furthermore, as a feature of the development language and process-orientated approach, our driver components are scheduled independently. This allows us, as developers, freedom from almost all scheduling concerns. Developers can instead concentrate on the correct functionality

DOI: 10.1002/cpe

of their particular driver, without needing to worry about how the surrounding USB infrastructure operates. If the USB infrastructure does change in the future, all we need do is to ensure backwards compatibility—easily done with concurrent 'adapter' processes.

RMoX itself still has far to go. The hardware platform for which we are developing currently is a PC104+ *embedded PC*—a standardized way of building embedded PC systems, with stackable PCI and Industry Standard Architecture (ISA) bus interconnects [19]. This makes a good initial target for several reasons. First, the requirements placed on embedded systems are substantially less than what might be expected for a more general-purpose (desktop) OS—typically acting as hardware management platforms for a specific application (e.g. industrial control systems, automotive applications). There is, however, a strong requirement for reliability in such systems. Second, the nature of the PC104+ target makes the developed components immediately reusable when targeting desktop PCs in the future. Additionally, USB is being increasingly used for device connectivity within embedded PC104 systems, due to its versatility. The builds are routinely tested on desktop PCs and in emulators as standard, exercising aspects of RMoX's *scalability*.

In addition to USB, RMoX has support for the PCI bus and several PCI drivers, including the RTL8139-based network interfaces present on our PC/104+ boards. We aim to experiment with distributed RMoX systems in the near future, using nodes in a standard PC cluster or a distributed collection of PC/104+ boards, to further exercise scalability.

To further guarantee the *reliability* of RMoX, as well as other process-oriented systems programmed in occam-pi, we are looking at ways of formally specifying process behaviours that the compiler can check against the actual implementation. In the first instance, this will be to check communications on mobile channel bundles, of which RMoX makes extensive use, particularly within the USB infrastructure presented here. The ideas here are similar to 'contracts' [20], which have been introduced into other languages such as Erlang [21] and Sing# [22]. Importantly, these checks will require no additional work on the part of the driver developer.

5.1. Related work

The most significant piece of related research is Microsoft Research's Singularity OS [23], which takes a similar concurrent approach to OS design. Their system is programmed in a variant of the *object-orientated* C# language, Sing#, which has extensions for efficient communication between processes—very similar in principle and practice to occam-pi's mobilespace [24]. Within the embedded systems market are long-established OSs such as Integrity [9], LynxOS [10], VxWorks [25] and QNX Neutrino [26], certified at various levels and used in a range of mission-critical applications. The established *integrity* and *reliability* of these systems represent some of our long-term aims for RMoX.

More generally, there is a wide range of related research on novel approaches to OS design. Most of these, even if indirectly, give some focus to the language and programming paradigm used for implementation—something other than the *threads-and-locks* procedural approach of C. For example, the Haskell OS [27] uses a functional paradigm. The Plan9 OS [28] uses a concurrent variant of C ('Alef'). However, we take the view that the *concurrent process-oriented* approach of occam-pi *seems* to be more suitable. Future work will seek to investigate if this view can be supported both analytically and experimentally.

DOI: 10.1002/cpe

A large amount of ongoing research elsewhere aims to make the existing languages and paradigms more efficient and concrete in their handling of concurrency. With RMoX, we are starting with something that is already highly concurrent with extremely low overheads for managing that concurrency—due in part to years of experience and maturity from CSP, occam and the Transputer [29].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers who provided valuable and insightful feedback on earlier versions of this article. This work was funded by EPSRC, grant EP/D061822/1.

REFERENCES

1. Barnes F, Jacobsen C, Vinter B. RMoX: A raw metal occam experiment. *Proceedings of Communicating Process Architectures 2003*, Broenink J, Hilderink G (eds.). IOS Press: Amsterdam, The Netherlands, 2003.
2. Barnes FR. Dynamics and pragmatics for high performance concurrency. *PhD Thesis*, University of Kent, June 2003.
3. Welch P, Barnes F. Communicating mobile processes: Introducing occam-pi. *25 Years of CSP* (*Lecture Notes in Computer Science*, vol. 3525), Abdallah A, Jones C, Sanders J (eds.). Springer: Berlin, 2005; 175–210.
4. Hoare C. *Communicating Sequential Processes*. Prentice-Hall: London, 1985. ISBN: 0-13-153271-5.
5. Milner R. *Communicating and Mobile Systems*: *The Pi-calculus*. Cambridge University Press: Cambridge, 1999. ISBN: 0-52165-869-1.
6. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream computing on graphics hardware. *SIGGRAPH '04*: *ACM SIGGRAPH 2004 Papers*. ACM: New York, NY, U.S.A., 2004; 777–786. DOI: 10.1145/1186562.1015800.
7. Beyer D, Henzinger TA, Jhala R, Majumdar R. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer* 2007; **9**:505–525. DOI: 10.1007/s10009-007-0044-z.
8. Sharma B, Dhodapkar SD, Ramesh S. Assertion checking environment (ace) for formal verification of C programs. *Proceedings of SAFECOMP 2002* (*Lecture Notes in Computer Science*, vol. 2434/2002). Springer: Berlin, 2002.
9. Green Hills Software Inc. Integrity RTOS. URL: http://www.ghs.com/ [22 March 2009].
10. Lynux Works. LynxOS. URL: http://www.lynuxworks.com/ [22 March 2009].
11. Compaq, Intel, Microsoft, NEC. Universal Serial Bus Specification—Revision 1.1, September 1998.
12. Ritson CG, Barnes FR. A process oriented approach to USB driver development. *Proceedings of Communicating Process Architectures 2007*, McEwan AA, Schneider S, Ifill W, Welch P (eds.). IOS Press: Amsterdam, The Netherlands, 2007.
13. Welch P, Justo G, Willcock C. Higher-level paradigms for deadlock-free high-performance systems. *Proceedings of the 1993 World Transputer Congress*. IOS Press: Netherlands, 1993.
14. Welch P. Graceful Termination—Graceful Resetting. *Proceedings of OUG 10*. IOS Press: The Netherlands, 1989.
15. Welch P, Moores J, Barnes F, Wood D. The KRoC Home Page, 2008. URL: http://www.cs.kent.ac.uk/projects/ofa/kroc/ [22 March 2009].
16. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. Universal Serial Bus Specification—Revision 2.0, April 2000. URL: http://www.usb.org/developers/docs/usb_20_05122006.zip [22 March 2009].
17. PCI Special Interests Group. PCI Local Bus Specification—Revision 2, 2 December 1998.
18. Intel. Universal Host Controller Interface (UHCI) design guide, March 1996. URL: http://download.intel.com/technology/usb/UHCI11D.pdf [22 March 2009].
19. PC/104 Embedded Consortium. PC/104-Plus Specification 2001. URL: http://pc104.org/ [22 March 2009].
20. Boosten M. Formal contracts: Enabling component composition. *Proceedings of Communicating Process Architectures 2003*, Broenink J, Hilderink G (eds.). IOS Press: Amsterdam, The Netherlands, 2003.
21. Jimenez M, Lindahl T, Sagonas K. A language for specifying type contracts in Erlang and its interaction with success typings. *Erlang '07*: *Proceedings of the 2007 SIGPLAN Erlang Workshop*. ACM: New York, NY, U.S.A., 2007; 11–17. DOI: 10.1145/1292520.1292523.
22. Spear MF, Roeder T, Hodson O, Hunt GC, Levi S. Solving the starting problem: Device drivers as self-describing artifacts. *EuroSys '06*: *Proceedings of the First ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. ACM: New York, NY, U.S.A., 2006; 45–57. DOI: 10.1145/1217935.1217941.

23. Fahndrich M, Aiken M, Hawblitzel C, Hodson O, Hunt G, Larus J, Levi S. Language support for fast and reliable message-based communication in singularity OS. *Proceedings of EuroSys 2006*, Leuven, Belgium, 2006.
24. Barnes F, Welch P. Mobile data, dynamic allocation and zero aliasing: An occam experiment. *Proceedings of Communicating Process Architectures 2001*, Chalmers A, Mirmehdi M, Muller H (eds.). IOS Press: Amsterdam, The Netherlands, 2001.
25. Wind River. VxWorks. URL: http://www.windriver.com/ [22 March 2009].
26. QNX Software Systems. QNX Neutrino RTOS. URL: http://www.qnx.com/ [22 March 2009].
27. Hallgren T, Jones MP, Leslie R, Tolmach A. A principled approach to operating system construction in Haskell. *ICFP '05*: *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*. ACM Press: New York, NY, U.S.A., 2005; 116–128. DOI: 10.1145/1086365.1086380.
28. Pike R, Presotto D, Dorward S, Flandrena B, Thompson K, Trickey H, Winterbottom P. *Plan 9 from Bell Labs*, Murray Hill, New Jersey, U.S.A., 1995. Available from: http://www.cs.bell-labs.com/plan9dist/ [22 March 2009].
29. May M, Thompson P, Welch P. *Networks*, *Routers and Transputers*. Transputer and occam Engineering Series, vol. 32. IOS Press: Amsterdam, The Netherlands, 1993.