SpaceWire Working group 15th Sept 2004

SpaceCommRTOS

From a formal RTOS concept to
a universal communication mechanism
for distributed real-time systems

Eric.Verhulst@OpenLicenseSociety.org

www.OpenLicenseSociety.org

Background

• Virtuoso distributed RTOS (Eonic Systems)
  • Transputer background (CSP concepts)
  • from ADSL modem chips to > 1000 DSP sonar
  • ported and adapted to SpaceWire targets (e.g. Mosaic20)

• Features
  • Virtual Single Processor programming model
    • Distributed semantics
    • Transparant distributed/parallel processing
    • Low latency for high throughput parallel DSP
    • Small code size (5K code …)

• Issues
  • Acquired by Wind River Systems (and then slowly died)
  • Source access, long term support (COTS)
  • Got features-bloat in the end
  • Static model

• => Time to use the experience to reinvent it
The forgotten system level support

- software was invented to provide more abstraction
- software was invented to hide the hardware from the application
- benefits:
  - master complexity, larger systems, higher productivity
- problem1: single process view (processor support for scheduling is very weak - von Neumann syndrome)
- problem2: single processor view, but most systems are now distributed, MP, heterogenous, ...
  - communication aspects are often not part of the software development paradigm (von Neumann syndrome)
  - (asynchronous) communication aspects are often not part of the system level support provided by the hardware (synchronous logic syndrome)
**CSP: the forgotten paradigm or being reborn?**

- **CSP:** Communicating Sequential Processes (Hoare)
- **For HW design:**
  - In order to safeguard abstract equivalence:
    - Communication backbone needed
    - Automatic routing needed (but deadlock free)
    - Process scheduler if on same processor
  - In order to safeguard real-time behavior in distributed targets:
    - Prioritization of communication for dynamic applications
    - Allocate time-slots beforehand for stationary applications
  - In order to handle asynchronous multi-byte communication:
    - Buffering at communication layer (unless handshake and throttling possible)
    - Packetisation
    - Packet size determines throughput + blocking time
    - DMA in background
  - **Result:**
    - prioritized packet switching: header, priority, payload (cargoload)
    - hardware supports needs to be adequate (better a link than a shared bus)
    - Communication not fundamentally different from data I/O
  - => SpaceWire

---

**Distributed real-time applications**

- Distributed real-time can be done quasi-transparently
- **Preconditions:**
  - distributed semantics
  - communication is system level
  - use prioritised packet switching
  - communication latency is the new issue (set-up times)
  - addressing scheme
  - hardware must support it:
    - communication/computation ratio >> 1
    - preemption when using buses
    - better are dedicated links
    - DMA with separate busses
  - hardware and software should be co-designed
  - run-time programming model best developed using the CSP paradigm
    - likely the only formal model that can prove a large system with multiple processes and communication channels is correct
    - first well with model checkers using timed automata a.o.
Rationale: what’s wrong with COTS and Open source?

- Needs of users:
  - quality: certified trusted components
  - needs correct IP model (-> interfaces !)
  - verification needs proof of correctness of the design
  - certification needs design document
  - control:
    - long term availability and support (up to 25 yrs)
    - non-restrictive licensing terms
  - flexibility:
    - modularity, adaptability, scalability

- COTS:
  - often overly protects vendors
  - no guarantee for quality

- Open Source:
  - source, but GPL and no documentation
  - you’re often ‘on your own’
  - quality? innovative? Yes, but often not

- The real issue:
  - engineering vs. crafting

Quality = added value

- Quality of the process: this means (software) development should be predictable in terms of the specifications to be met (size, cost, performance, deadlines, ...).

- The 3 qualities:
  - Correctness:
    - Trustworthy in the sense that the software/system should operate as specified [correct by design, verified by proof, checked by certification]
  - Safety:
    - Trustworthy in the sense that there should be no harmful effects when system components fail (non-intentional) [support fault tolerance]
  - Security:
    - Trustworthy in the sense that there should be no harmful effects when systems components are tampered with (intentional) [support human generated faults]

- Conclusion: community needs
  - In other words, we should be able to develop and use software as trusted components to produce trustworthy products in a predictable way
OpenComRTOS

- Scalable distributed RTOS based on message passing
  - actually: **scalable communication** layer with **scheduling** support
- distributed semantics (transparant parallel programming)
- to be formally analysed and validated
  - extensive system-wide message passing protocols
  - using formal model checkers (e.g. based on CSP)
- safety and security by compatible plug-ins/extensions
  - same external behavior, but blocking 'faults' and 'intrusions'
- 3+ layers:
  - (NULL-OS)-L: testing and local I/O
  - L0
    - very small (1 K), core primitives, core system packet
    - typical use: MP-SoC, DSPs
    - includes scheduler, low latency router and drivers
  - L1
    - sema, queue, mailbox, resources, ...: traditional RTOS services
    - emulate RTOS (but often only SP), cabinet level
  - L2
    - supports widely distributed operation, RT-CORBA

OpenComRTOS: Level L2
- variable size packets
- widely distributed addressing
- dynamic protocol packets
- extensible API

OpenComRTOS: Level L1
- fixed size packets
- cluster addressing
- dynamic protocol packets
- API emulation

OpenComRTOS: Level L0
- fixed size packets
- tightly clustered addressing
- static protocol packets
- system packets
- scheduler
- runtime monitor

Packet structure

- **HeaderL0**
- **DataL0/HeaderL1**
- **DataL1/HeaderL2**
- **DataL2**

- **SpW**
  - public connections
  - target: 100-500K

- **std RTOS**
  - target: 10-20K

- **semaMW (system-wide)**
- **peek-poke (system-wide)**
- target: 1K code

Hardware support or tunneling protocol
Open licensing model

- Cross between COTS and Open Source
  - affordable licensing fees
  - with source code
  - with full documentation
  - if possible validated, certified => trusted component
  - no limits on application development
  - OLS remains owner and master of versions
- Licensee = membership of OLS (/year)
  - can vote for representation in Scientific Committee
  - can contribute
- Third parties can contribute
  - royalties

The systems engineerings approach (semantics!)
Conclusion

- SpaceWire addresses a fundamental need of reliable and scalable communication at the hardware level
- A higher level protocol standard is needed but is not enough
  - But a formal development is needed to provide a trustworthy result
- System level issues:
  - scalability
  - portability
  - communication is tightly integrated with scheduling and drivers
  - correctness, safety, security (-> formal development)
  - integration with 3rd party systems and standards
  - similar needs in other domains but less real-time critical
    - EU security
    - C3SI
    - Virtual corporations
  - ...
- Hence: OpenComRTOS adopted to needs of SpaceWire and Space projects

=> SpaceCommOS