OpenComRTOS a Runtime Environment for Interacting Entities

Bernhard H.C. Sputh, Oliver Faust, Eric Verhulst, and Vitaliy Mezhuyev

Altreonic N.V.
Linden Labs
Email: bernhard.sputh@altreonic.com

03.11.2009

Outline

History of Altreonic

- Eonic (Eric Verhulst): 1989 – 2001
  - Developed Virtuoso a Parallel RTOS (sold to Wind River Systems);
  - Communicating Sequential Processes as foundation of the "pragmatic superset of CSP";
- Open Licence Societey: 2004 – now
  - R&D on Systems and Software Engineering;
  - Unified Semantics & Interacting Entities;
  - Formally developed OpenComRTOS;
- Altreonic: 2008 – now
  - Commercialisation of Open Licence Society Results;
  - Based in Linden (near Leuven) Belgium;
  - "Push the button – high reliability": `http://www.altreonic.com`

History of Altreonic

- Eonic (Eric Verhulst): 1989 – 2001
  - Developed Virtuoso a Parallel RTOS (sold to Wind River Systems);
  - Communicating Sequential Processes as foundation of the "pragmatic superset of CSP";
- Open Licence Societey: 2004 – now
  - R&D on Systems and Software Engineering;
  - Unified Semantics & Interacting Entities;
  - Formally developed OpenComRTOS;
- Altreonic: 2008 – now
  - Commercialisation of Open Licence Society Results;
  - Based in Linden (near Leuven) Belgium;
  - "Push the button – high reliability": http://www.altreonic.com

History of Altreonic

- Eonic (Eric Verhulst): 1989 – 2001
  - Developed Virtuoso a Parallel RTOS (sold to Wind River Systems);
  - Communicating Sequential Processes as foundation of the "pragmatic superset of CSP";
- Open Licence Societey: 2004 – now
  - R&D on Systems and Software Engineering;
  - Unified Semantics & Interacting Entities;
  - Formally developed OpenComRTOS;
- Altreonic: 2008 – now
  - Commercialisation of Open Licence Society Results;
  - Based in Linden (near Leuven) Belgium;
  - "Push the button – high reliability": `http://www.altreonic.com`

Interesting Facts about OpenComRTOS

- CSP inspired Real-time Operating System;
- Formally designed and developed;
- Small code size (typically 5 – 10KiB);
- Support for Systems composed out of different CPU Architectures;
- Currently available Ports:
  - Posix32/64
  - Win32
  - ARM Cortex M3
  - Leon3
  - Microblaze
  - MLX-16
  - XMOS (experimental under development)

Altreonic

Interesting Facts about OpenComRTOS

- CSP inspired Real-time Operating System;
- Formally designed and developed;
- Small code size (typically 5 – 10KiB);
- Support for Systems composed out of different CPU Architectures;
- Currently available Ports:
  - Posix32/64
  - Win32
  - ARM Cortex M3
  - Leon3
  - Microblaze
  - MLX-16
  - XMOS (experimental under development)

Outline

OpenComRTOS Programming Model

Axioms:

- Every Node / Processor has its own private memory;
- Each OpenComRTOS Application is composed out of Interacting Entities: Tasks (Threads) and Hubs (Generic Synchronisation Primitive of OpenComRTOS).
- All interactions between Tasks is decoupled over a *Hub*.
- Nodes communicate over Links (unidirectional or bidirectional);
- Each Node executes an instance of OpenComRTOS;

Altreonic

Tasks

- Each Task in OpenComRTOS is prioritised. The Kernel-Task has the highest Priority (1), the Idle-Task the lowest Priority (255).
- Each Task has one Packet which it can use to request services.
- Tasks, like Processes do not share memory.

Hub

- One of the results of formal modelling of OpenComRTOS;
- Can be specialised to represent: Events, Semaphores, FIFOs, Ports, Resources, Mailbox, Memory-pools, etc;
- A Hub has 4 functional parts:
  - Synchronisation point between Tasks
  - Stores task's waiting state if needed
  - Predicate function: defines synchronisation conditions and lifts waiting state of tasks
  - Synchronisation function: functional behaviour after synchronisation: can be anything, including passing data
- All HUBs operate system-wide, but transparently: Virtual Single Processor programming model
- Possibility to create application specific hubs & services! $\Longrightarrow$ a new concurrent programming model

Hub

- One of the results of formal modelling of OpenComRTOS;
- Can be specialised to represent: Events, Semaphores, FIFOs, Ports, Resources, Mailbox, Memory-pools, etc;
- A Hub has 4 functional parts:
  - Synchronisation point between Tasks
  - Stores task's waiting state if needed
  - Predicate function: defines synchronisation conditions and lifts waiting state of tasks
  - Synchronisation function: functional behaviour after synchronisation: can be anything, including passing data
- All HUBs operate system-wide, but transparently: Virtual Single Processor programming model
- Possibility to create application specific hubs & services! $\Longrightarrow$ a new concurrent programming model

Hub

- One of the results of formal modelling of OpenComRTOS;
- Can be specialised to represent: Events, Semaphores, FIFOs, Ports, Resources, Mailbox, Memory-pools, etc;
- A Hub has 4 functional parts:
  - Synchronisation point between Tasks
  - Stores task's waiting state if needed
  - Predicate function: defines synchronisation conditions and lifts waiting state of tasks
  - Synchronisation function: functional behaviour after synchronisation: can be anything, including passing data
- All HUBs operate system-wide, but transparently: Virtual Single Processor programming model
- Possibility to create application specific hubs & services! $\Longrightarrow$ a new concurrent programming model

Available Task - Hub Interactions

- _W — Waiting, blocking behaviour, the Task will not be scheduled unless the synchronisation occured.
- _NW: — Non waiting, if the other side is not ready to synchronise, the operation is aborted and the Task rescheduled.
- _WT: — Waiting with Timeout, blocking until a certain time has expired, then behaving like _NW.

Altreonic

Available Task - Hub Interactions

- _W — Waiting, blocking behaviour, the Task will not be scheduled unless the synchronisation occured.
- _NW: — Non waiting, if the other side is not ready to synchronise, the operation is aborted and the Task rescheduled.
- _WT: — Waiting with Timeout, blocking until a certain time has expired, then behaving like _NW.

Available Task - Hub Interactions

- _W — Waiting, blocking behaviour, the Task will not be scheduled unless the synchronisation occured.
- _NW: — Non waiting, if the other side is not ready to synchronise, the operation is aborted and the Task rescheduled.
- _WT: — Waiting with Timeout, blocking until a certain time has expired, then behaving like _NW.

Available Task - Hub Interactions

- _W — Waiting, blocking behaviour, the Task will not be scheduled unless the synchronisation occured.
- _NW: — Non waiting, if the other side is not ready to synchronise, the operation is aborted and the Task rescheduled.
- _WT: — Waiting with Timeout, blocking until a certain time has expired, then behaving like _NW.

From Idea to Implementation

1. Define a topology;
2. Define the Tasks and Hubs;
3. Write the code for the Tasks;
4. Compile the project.
    1. Generate the code representing the topology;
    2. Generate the corresponding build system;
    3. Compile the code for the individual nodes.

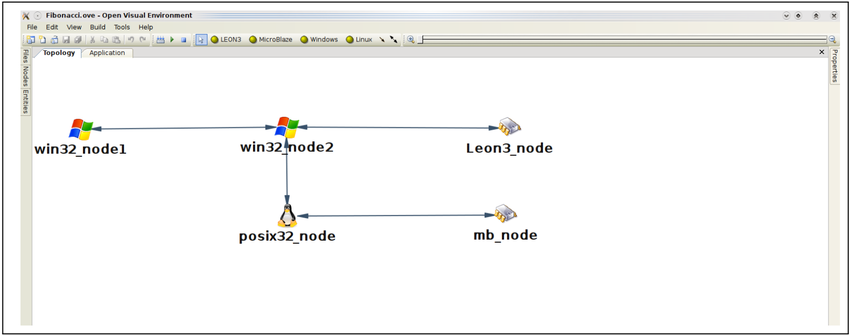
From Idea to Implementation

1. Define a topology;
2. Define the Tasks and Hubs;
3. Write the code for the Tasks;
4. Compile the project.
    1. Generate the code representing the topology;
    2. Generate the corresponding build system;
    3. Compile the code for the individual nodes.

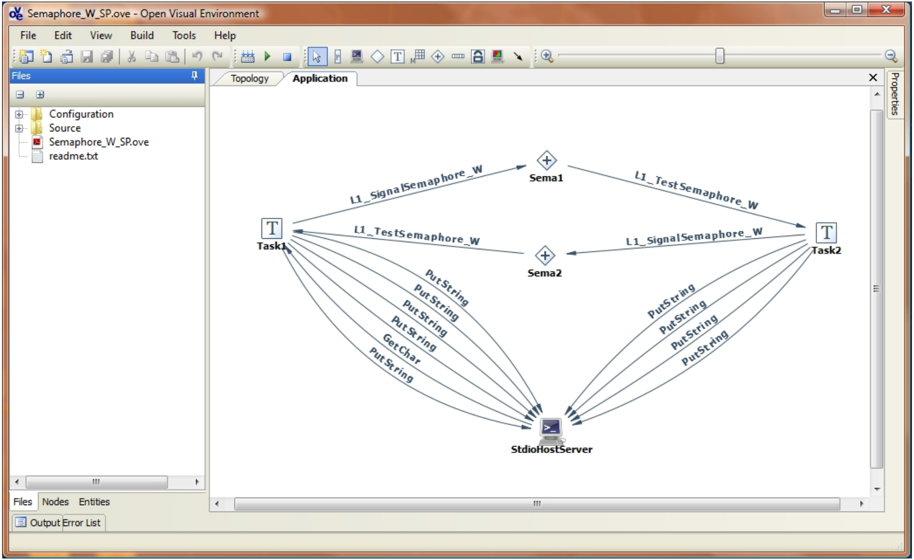
OpenVE

- Separates Topology and Functionality;
- Generates of Source Code as much as possible;
- Supports heterogeneous systems, i.e. systems composed of nodes with different CPU architectures and link technologies.
- Generates configuration files for the OpenComRTOS code generators.

Defining a Topology with OpenVE



Defining the Entities and the Interactions



The user now adds the desired functionality to the generated code.

Defining the Program Logic

Topology | Application | APP_1_Task.c*

```c
#include <L1_api.h>
#include "L1_node_config.h"
#include <StdioHostService/StdioHostClient.h>

void APP_1_Task (L1_TaskArguments Arguments)
{
  PutString(StdioHostServer, "This example demonstrates the Tasks synchronization mechani
    Task1 and Task2 signal Sema1 and Sema2 by L1_SignalSemaphore_W\n\
    Task1 and Task2 test Sema1 and Sema2 by L1_TestSemaphore_W\n\n\
    Used waiting versions of SignalSemaphore and TestSemaphore functions(_W)\n\
    Both tasks are placed on the same node, so its a single processor example\n\n\
    Press Enter to start the example\n");

  GetChar(StdioHostServer);

  while(1)
  {
    PutString(StdioHostServer, "Task 1 Signal Sema 1\n");
    if (RC_OK != L1_SignalSemaphore_W (Sema1))
    {
      PutString(StdioHostServer, "Not Ok\n");
    }
    PutString(StdioHostServer, "Task 1 Test Sema 2\n");
    if (RC_OK != L1_TestSemaphore_W (Sema2))
    {
      PutString(StdioHostServer,"Not Ok\n");
    }
  }
}
```

Building the Application

From the system description provided by OpenVE, the complete Application is built in two steps:

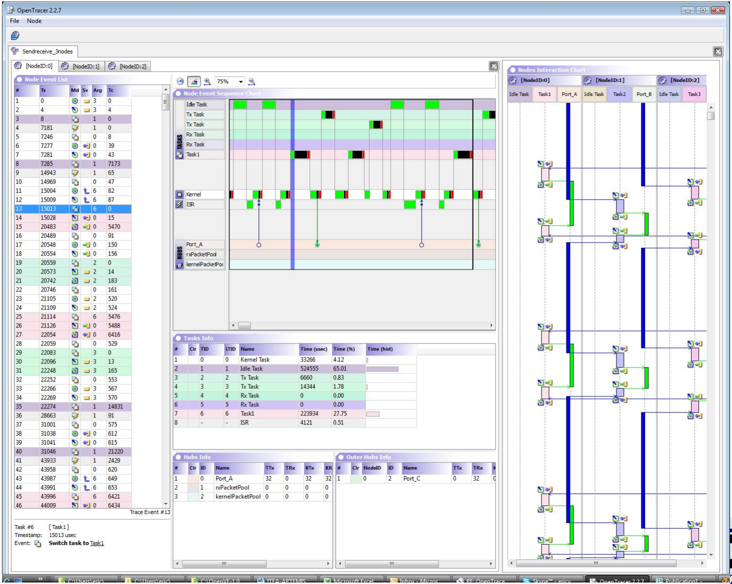
1. Project Generation Phase:
   - Routing Table for the individual Nodes;
   - Makefile based build system to build the complete System
   - Derives individual Node descriptions, from the system description.
2. Node Generation Phase:
   - Generates the source code which creates the Tasks and Hubs of a Node
   - Generates the IDs for the entities of a Node.
   - Node specific build system (at present CMake based).

Building the Application

From the system description provided by OpenVE, the complete Application is built in two steps:

1. Project Generation Phase:
   - Routing Table for the individual Nodes;
   - Makefile based build system to build the complete System
   - Derives individual Node descriptions, from the system description.
2. Node Generation Phase:
   - Generates the source code which creates the Tasks and Hubs of a Node
   - Generates the IDs for the entities of a Node.
   - Node specific build system (at present CMake based).

Diagnosis using the Tracing Capability



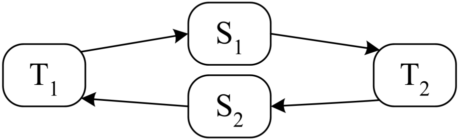
Outline

Code Figures (SP) in Byte

Service	MLX16	MB[1]	Leon3	ARM	XMOS
L1 Hub shared	400	4756	4904	2192	4854
L1 Port	4	8	8	4	4
L1 Event	70	88	72	36	54
L1 Semaphore	54	92	96	40	64
L1 Resource	104	96	76	40	50
L1 FIFO	232	356	332	140	222
L1 PacketPool	NA	296	268	120	166
Total	1048	5692	5756	2572	5414

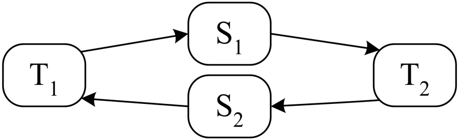
[1] Xilinx MicroBlaze

Measurement Setup I



- Every Interaction a Task performs with a Service requires two context switches:
  1. From the User-Task to the Kernel-Task;
  2. From the Kernel-Task to the User-Task;
- In this loop there are 4 Interactions, thus a total of 8 context swiches are performed by it.

Measurement Setup I



- Every Interaction a Task performs with a Service requires two context switches:
  1. From the User-Task to the Kernel-Task;
  2. From the Kernel-Task to the User-Task;
- In this loop there are 4 Interactions, thus a total of 8 context swiches are performed by it.

Measurement Setup I

```c
void T1 (L1_TaskArguments Arguments){
    L1_UINT32 i=0, start=0, stop=0;
    while(1) {
        start = L1_getElapsedCycles();
        for (i = 0; i < 1000; i++){
            L1_SignalSemaphore_W(S1);
            L1_TestSemaphore_W(S2);
        }
        stop = L1_getElapsedCycles();
    }
}

void T2 (L1_TaskArguments Arguments){
    while(1) {
        L1_TestSemaphore_W(S1);
        L1_SignalSemaphore_W(S2);
    }
}
```

Measurement Results

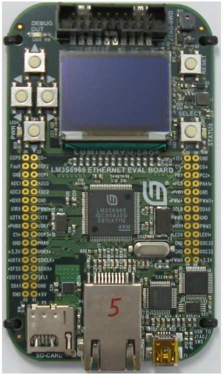
	MLX16	MB	Leon3
Clock speed	6MHz	100MHz	40MHz
Context size	4×16bit	32×32bit	32×32bit
Memory location	internal	internal	external
Loop time	$100.8\mu s$	$33.6\mu s$	$136.1\mu s$

	ARM	XMOS
Clock speed	50MHz	100MHz
Context size	16×32bit	14×32bit
Memory location	internal	internal
Loop time	$52.7\mu s$	$26.8\mu s$

Interrupt Latency of an ARM Cortex M3

Properties of the Development board used to measure the Interrupt Latency:

- CPU: LM3s6965 ARM Cortex-MX 50MHz
- 64kiBi RAM
- 256 kiBi Flash
- Timer / Counter counting clock ticks;



Altreonic

What did we measure

1. IRQ to ISR Latency: The measured time indicates, how long it took until the first useful instruction inside the ISR can be executed. This means all OS specific work has been carried out already.
2. IRQ to Task Latency: The measured time indicates, how long it took after an IRQ to occur until the the task designated to handle the IRQ could execute the first useful instruction.

Latency Demo

Demo Time

How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

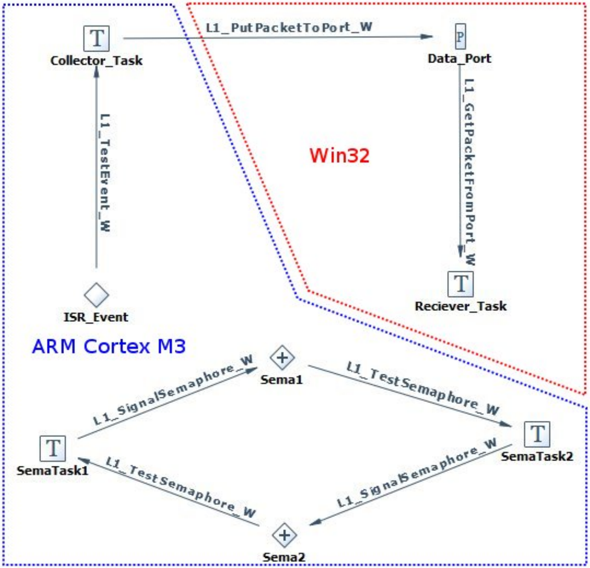
How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

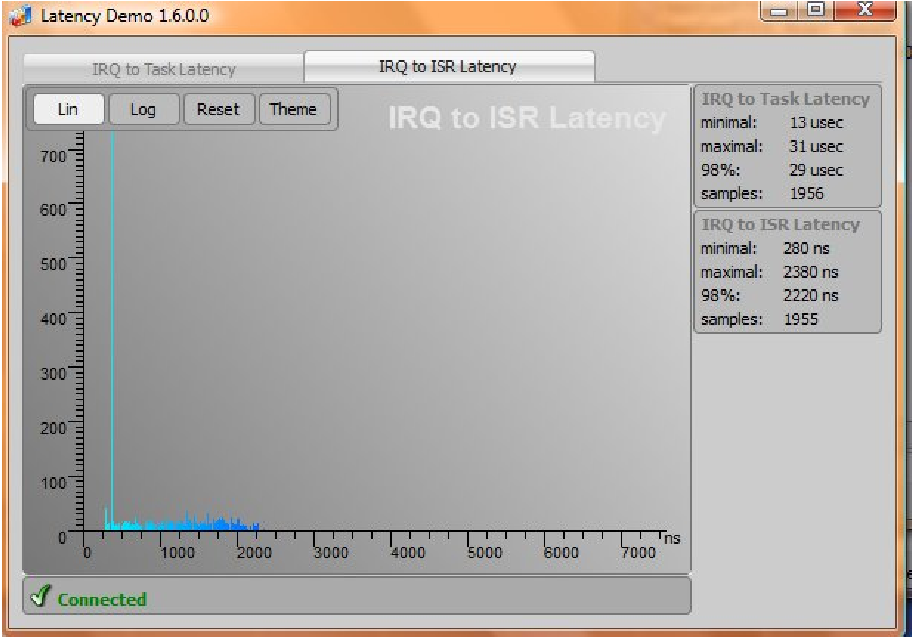
How do we measure it?

1. Interrupt to measure is the Timer / Counter (auto reloading) Interrupt, which is clocked with 50MHz, and fires every 1ms.
2. The ISR stores a copy of the counter value when it can perform the first useful statement. I.e. the point in the execution of the ISR when normal ISR thing can happen to make the hardware happy.
3. The ISR signals an Event-Hub, using non-waiting semantics, upon which a Task waits.
4. ISR terminates and schedules the Kernel Loop.
5. Kernel Loop processes request to signal the Event, and then schedules the Task waiting for this event.
6. Once running the Task reads in the counter value. Then it sends both the IRQ to ISR Latency and the IRQ to Task Latency to the Port-Hub located on the Windows Node.
7. The Task on the Windows Node retrieves the data from the Port-Hub and then passes it on to a Display Application.

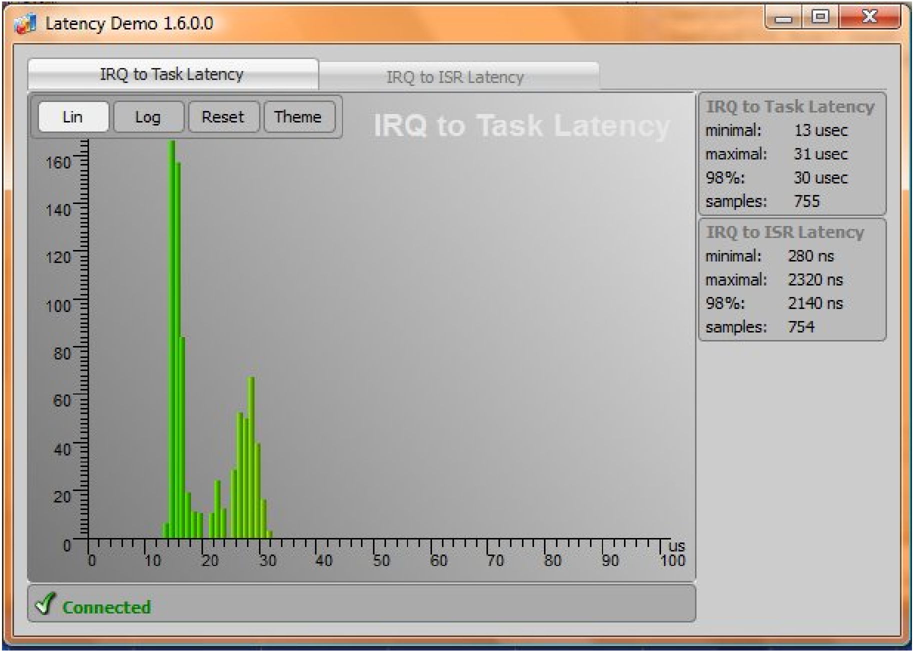
Application Diagram



Measurement Results 1: IRQ to ISR Latency



Measurement Results 2: IRQ to Task Latency



Outline

1. Introduction
   - History of Altreonic
   - OpenComRTOS Fact-sheet
2. OpenComRTOS Programming Model
   - Tasks
   - Hubs
   - From Idea to Implementation
3. Performance of OpenComRTOS
   - Code Size Figures
   - Context Switch Performance
   - Interrupt Latency of an ARM Cortex M3
4. Conclusions

Conclusions

- The formal development of OpenComRTOS resulted in:
  - small code size ($\sim 10\times$ smaller than Virtuoso from Eonic).
  - which results in higher performance, due to less code to be executed.
  - easily portable code base.
- OpenComRTOS Interacting Entities can be used to represent CSP style constructs.
- The separation of topology and application results in a Virtual Single Processor (VSP) programming model.

Questions?

Thank You!