Understanding the Real-Time Executive Daemon (rtxd) for Linux

Cost-Minimized, Performance-Maximized Industrial and Scientific Computing

What is the rtxd?

The real-time executive daemon, or the "rtxd", is a set of Linux processes and Linux process APIs providing a high-resolution computing time reference, a shared memory data management architecture, and a comprehensive set of computing services for real-time and accelerated industrial computing applications. The rtxd transforms the standard Linux computing environment into a microsecond resolution, highly instrumented, distributed computing, client-server platform.

[Keywords: *Computing and Data Handling, Smart Manufacturing, Smart Cities, Autonomous Vehicles, Artificial Intelligence, Robotics, Industrial Control, Digital Twin, Verification and Validation, Machine Learning, Multiphysics Modeling*]

Who uses the rtxd?

The rtxd began as a real-time server code in the aerospace and defense industry and evolved over more than two decades. The rtxd is used extensively in complex product cyberphysical development. Since the ADEPT/rtxd release in Q1-2018, the rtxd has seen adoption in applications spanning smart manufacturing, autonomy, industrial data and controls, industrial digital twin systems, etc.

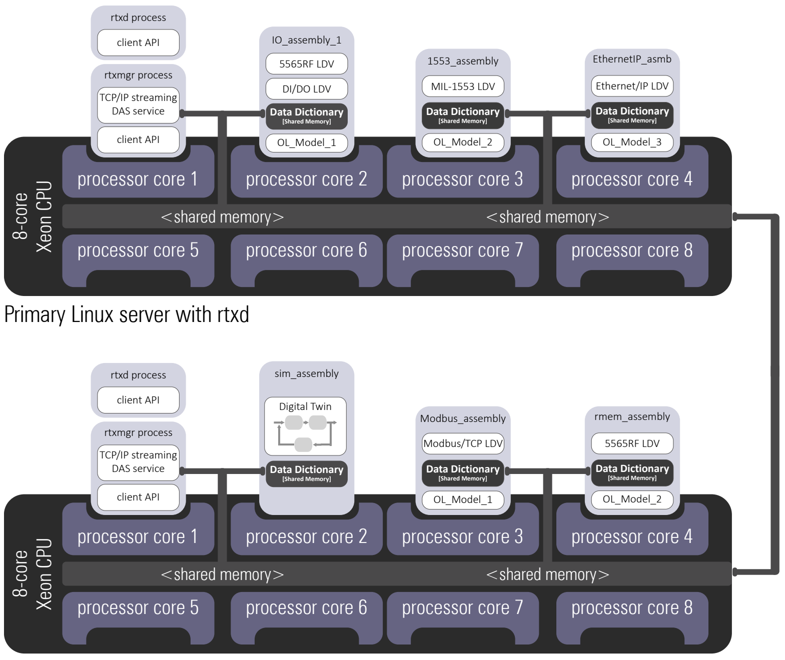


Figure 1 – rtxd Linux Real-Time Executive

Contents

The rtx daemon – Linux Real-Time Executive

The “rtx daemon” is a set of Linux services which runs on ARM and Intel-based computers and transforms them into full-featured real-time computers that are internally instrumented with performance, latency, jitter, and other precision time-based measurements for all aspects of the computer architecture. The rtx daemon (rtxd) software was developed, refined, and optimized by ADI over three decades and is now managed by the rtxd project. The goal of the rtxd project is to simplify the development and management of time-deterministic programming.

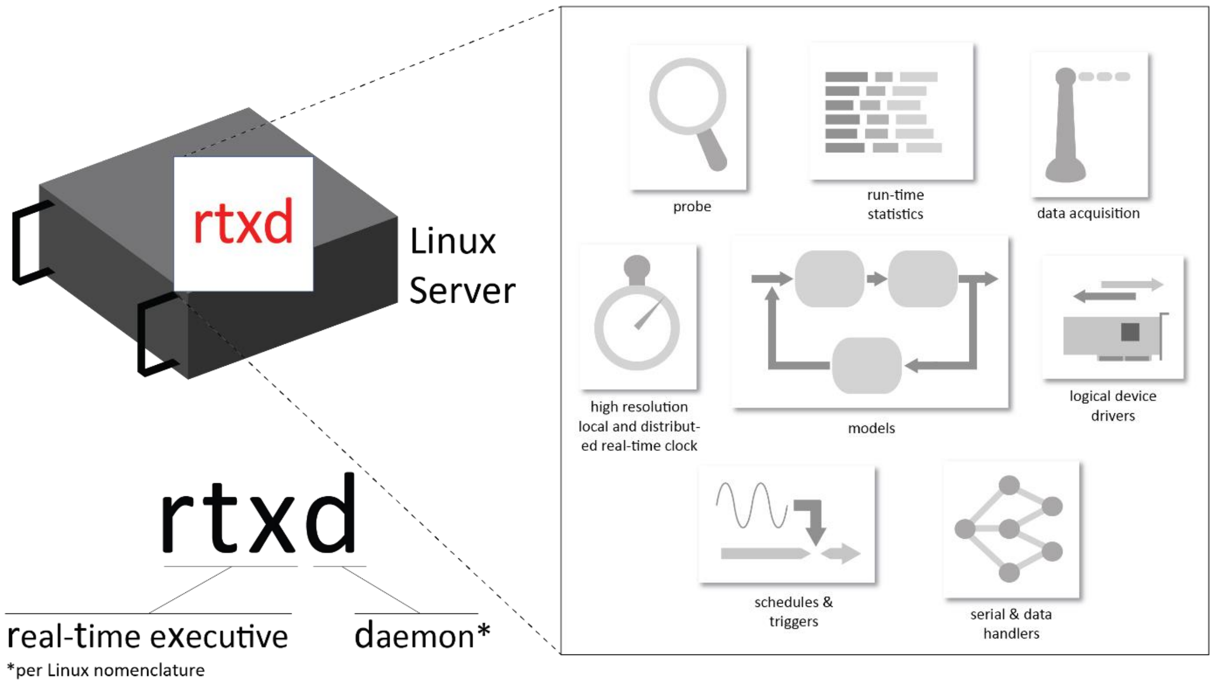


Figure 2 – rtxd Linux Real-Time Executive

By installing rtxd on your real-time Linux servers you get, right out of the box, a set of optimized real-time services with a multi-processor, multi-server distributed real-time computing and data handling architecture within a scalable client-server paradigm. The rtxd dramatically reduces the engineering and IT costs of modernizing your computing and data handling assets.

Real-Time Computing and Data Handling Frameworks

A real-time computing and data handling framework (real-time framework) is made up of one or more real-time Linux servers operating in coordination and with time synchronization. The real-time framework provides a flexible computational structure allowing these Linux servers to be interfaced with the real world, e.g. equipment, PLCs, sensors, actuators, data acquisition devices, etc. and operated as a time-synchronized cyberphysical system.

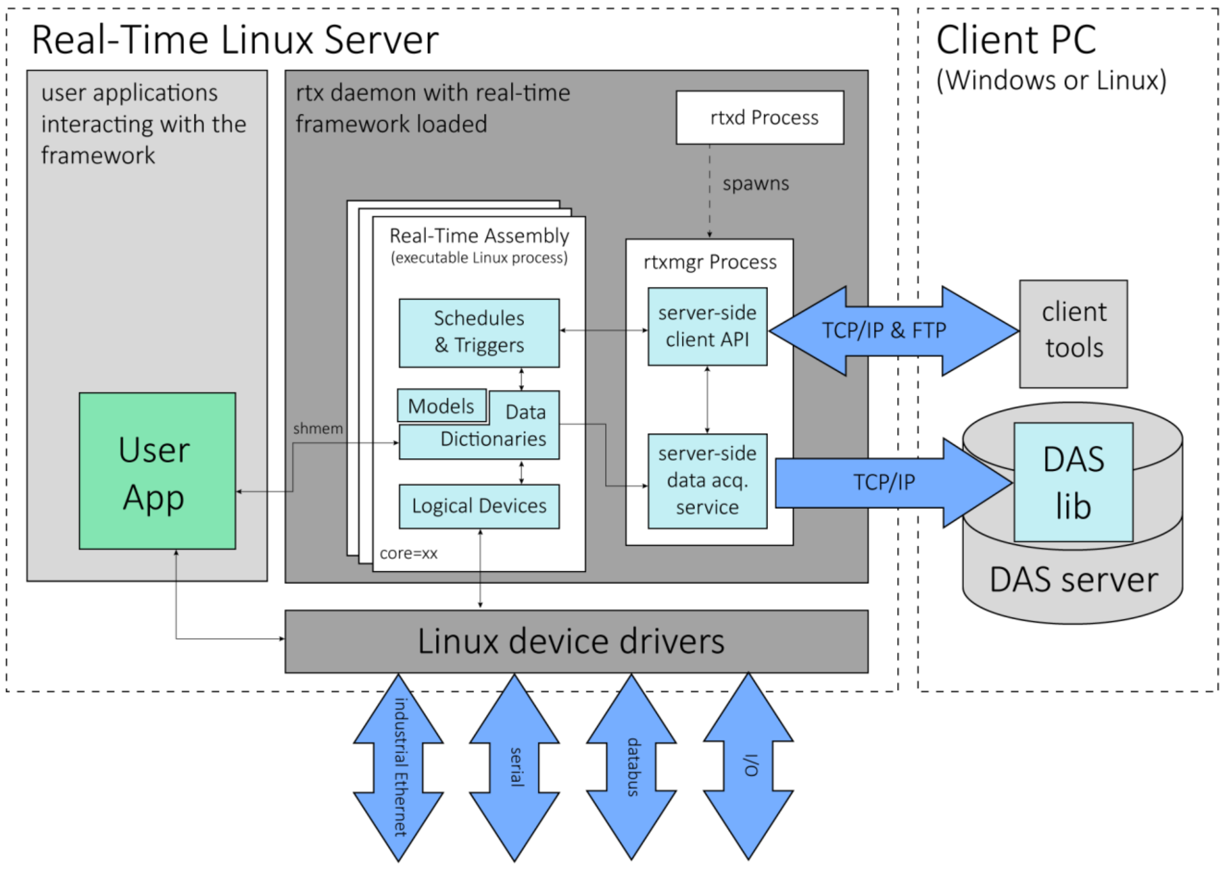


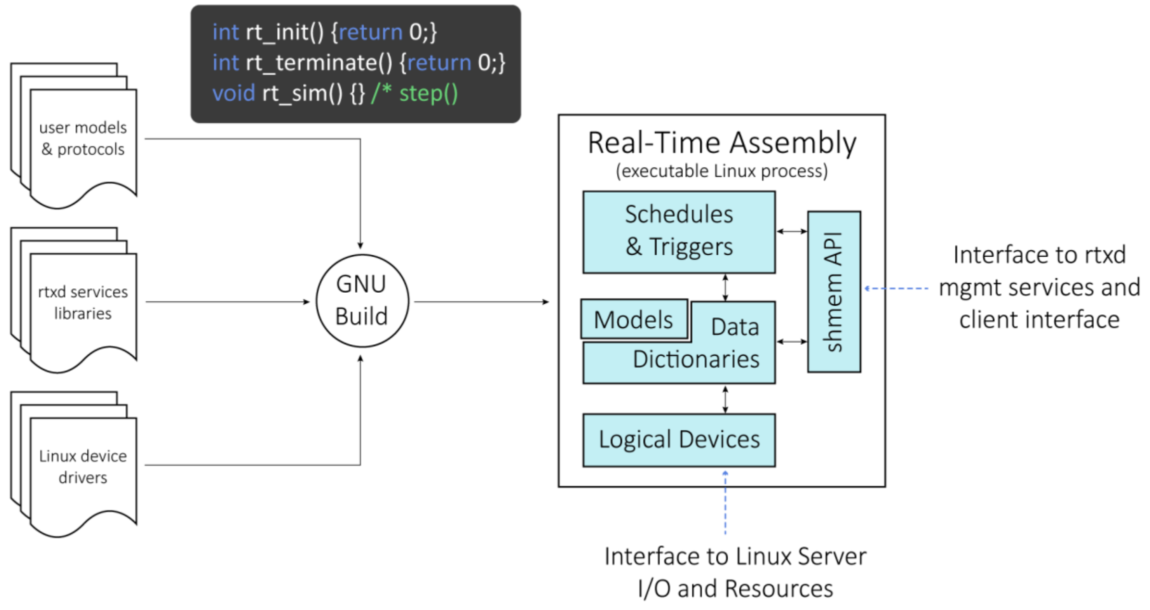
Figure 3 – Real-Time Assemblies Executing in an rtxd-Installed Linux Server

When a real-time Linux server boots, the rtxd process is spawned as part of the boot process. A real-time framework is loaded on one or more Linux servers by FTPing the set of assemblies to the participating servers along with configuration, data connections, and data dictionary XML files. After FTPing the framework files to the server, a framework load is commanded through the rtxd process API. This load command spawns the rtxmgr process which proceeds to execute and control each real-time assembly executable. Client tools interface with the framework through its TCP/IP API to get and put values, setup live streaming data, configure schedules & triggers, and interact through a wide set of functions.

The real power of industrial real-time Linux servers comes when you make use of multi-server distributed frameworks. The rtx daemon makes distributed computation and data handling a greatly simplified task by including both inter-process communication and inter-server communication. A primary rtxd server is assigned to act as synchronization and scheduling master for all Linux servers participating in the framework. Connections between data dictionary items are defined and if connected assemblies are allocated to different servers then the rtxd services automatically transfer data at each frame or selected frame multiple.

Real-Time Assemblies

A central element within rtxd's approach to scalable industrial real-time Linux servers is the "real-time assembly". Assemblies are used to package your computation and data handling tasks into one or more executable Linux processes which get uploaded to your real-time Linux server. Assemblies are allocated to specific processor cores on a multi-core server (Intel or ARM) and to specific Linux servers in a distributed, multi-server deployment. Figure 4 illustrates the components of a real-time assembly and build process.



```
int rt_init() {return 0;}
int rt_terminate() {return 0;}
void rt_sim() {} /* step()
```

Figure 4 – Real-Time Assemblies

Each assembly includes a C/C++ model that exposes the basic functions of initialize(), step(), and terminate(). ADI's ADEPT Framework client tools include functions for importing Simulink and FMI models into your real-time assembly. The simplest model provides only a set of variables in memory to read and write device driver calls in order to pass data in and out of data dictionary items. Data dictionaries are made up of a block of shared memory and an XML metadata specification for the dictionary, e.g. variable names, data types, dimensions, read/write access, data range, etc. Connections between data dictionary items and logical device ports provide the mechanism for performing real-time I/O device reads/writes.

The process of compiling a real-time assembly for ADEPT/rtxd links in user model code and protocol libraries, rtxd services libraries, and Linux device drivers to build an executable Linux process.

Assemblies are able to access data from interfaces such as computer boards, e.g. PXI, PCIe, and local I/O resources using Logical Devices, or LDVs. LDVs are a device driver layer that converts from procedural low-level Linux drivers to data "ports". Ports get connected to data dictionary items to read and write data from I/O channels, e.g. TCP/IP and UDP Ethernet, industrial Ethernet, e.g. Ethernet/IP, Powerlink, discrete inputs. LDVs can have features such as calibration, scaling, and protocol logic.

The rtx daemon services libraries also include dynamic, real-time scripting capability called "Schedules and Triggers" or S&T. S&T is implemented as a logic driven real-time perturbation engine acting off data dictionary items. S&T are commonly used to detect conditions based on a trigger expression and trigger an action, e.g. if TEMP>125 then send alert message and set several data dictionary values to TRUE.

Framework Client-Server Architecture

The main driver for the rtxd's client-server architecture is the desire to separate asynchronous user interaction activities, HMI refresh, disk access, and other computationally disruptive tasks away from the industrial computing and data handling workhorse Linux servers. The client-server architecture also happens to bring along a great deal of other benefits including efficient and flexible control of frameworks distributed across multiple servers, access for multiple framework users through multi-client access, and enhanced data acquisition stream scalability.

The rtxd TCP/IP APIs allow multiple clients to connect and interact with the framework for management and operation. Figure 5 illustrates an industrial computing and data handling framework implemented using multiple real-time Linux servers and accessed by multiple clients.

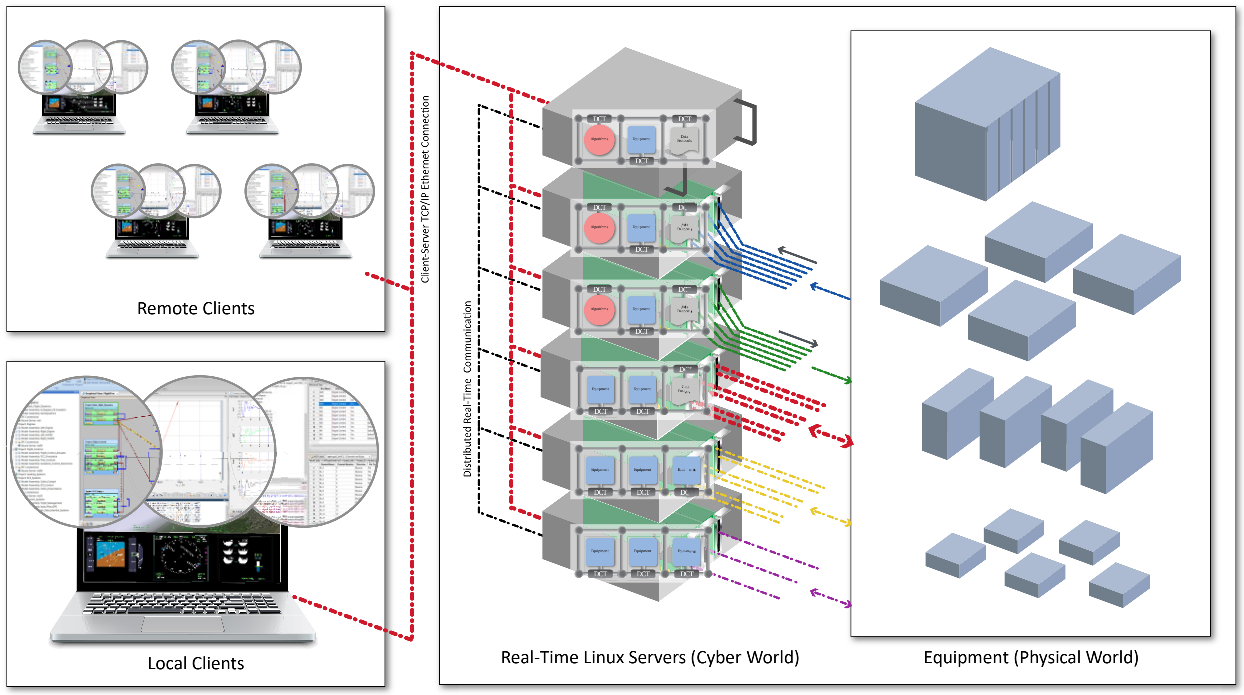

Remote Clients
Local Clients
Client-Server TCP/IP Ethernet Connection
Distributed Real-Time Communication
DCT
Algorithm
Equipment
Data Recorder
Real-Time Linux Servers (Cyber World)
Equipment (Physical World)

Figure 5 – Distributed Real-Time Framework Executing on Multiple Real-Time Linux Servers

Framework Control

The rtxd includes a comprehensive set of functions for loading and operating a framework. Control functions made available through the rtxd TCP/IP client API include:

- Framework load
- Framework Start/Stop/Reset/Halt
- Data Acquisition Service configuration
- Data Dictionary item value get/put
- Data Dictionary item value override
- Schedules & Triggers configuration

ADI's ADEPT client tools provide a user-friendly environment for working with rtxd frameworks.

Performance-Optimized Real-Time Computing – Method to the rtxd Madness

Real-time computing was once considered to be a bit of a black art. There are many reasons and opinions as to why it is viewed as unusual computing. A popular view suggests real-time computing tends to be foreign due to the way we typically learn to write software. We start with our “hello world” and we interact with the keyboard and the screen. Next we learn to allocate some memory for our program using a malloc(). Then we learn to read from and write to our hard disk drive for long-term data storage. We often are introduced to programming through the concept of a main() entry point and a procedural view of computing and task design.

In real-time computing, we switch to a frequency-based set of entry points, usually associated with a step() function in your code module. And there are a set of rules for implementing quality real-time code, including avoiding use of malloc(), avoiding disk reads and writes, and other best practices. In general, you try to avoid iterative numerical methods, e.g. solvers, integration algorithms, but sometimes they are unavoidable.

The rtxd software was designed and written with great effort invested in optimizing the services and architecture for real-time performance. A fundamental aspect of the rtxd high-performance real-time capability is its data-centric “Data Dictionary” concept that makes use of shared memory structures and APIs for accessing data from all assemblies and connecting data from one assembly process to another with ultra-low-latency and computational overhead. This shared memory structure lends itself naturally to distributed, multi-server real-time framework deployments.

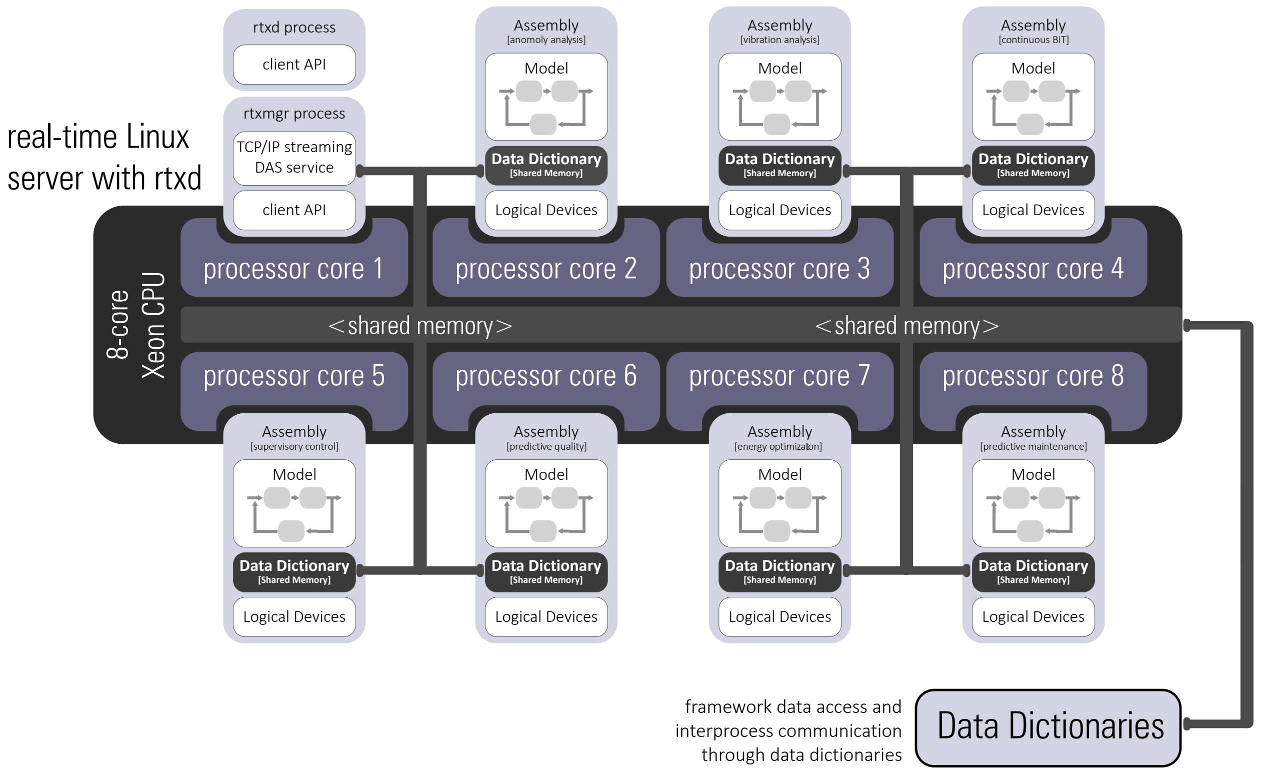

real-time Linux server with rtxd
rtxd process
client API
rtxmgr process
TCP/IP streaming DAS service
client API
Assembly [anomoly analysis]
Model
Data Dictionary [Shared Memory]
Logical Devices
Assembly [vibration analysis]
Model
Data Dictionary [Shared Memory]
Logical Devices
Assembly [continuous BIT]
Model
Data Dictionary [Shared Memory]
Logical Devices
8-core Xeon CPU
processor core 1
processor core 2
processor core 3
processor core 4
<shared memory>
<shared memory>
processor core 5
processor core 6
processor core 7
processor core 8
Assembly [supervisory control]
Model
Data Dictionary [Shared Memory]
Logical Devices
Assembly [predictive quality]
Model
Data Dictionary [Shared Memory]
Logical Devices
Assembly [energy optimizaton]
Model
Data Dictionary [Shared Memory]
Logical Devices
Assembly [predictive maintenance]
Model
Data Dictionary [Shared Memory]
Logical Devices
framework data access and interprocess communication through data dictionaries
Data Dictionaries

Figure 6 –Assemblies and Shared Memory Data Dictionaries

The decision to make use of compiled binary assembly executables as the basic container for computing and data handling tasks was made to get the very best real-time performance. Every computer has a fixed computational budget. As it says in its name, rtxd is a daemon, a set of services running in the background to implement a real-time computing framework. You get to choose how much of those services you want to make use of. You may write your Linux applications any way you like, and you can stitch them into an assembly as much or as little as you like. You may just pass some basic operational and/or management data into an assembly's data dictionary to give you client access to your application. At the other extreme, you may use rtxd as the main entry points for all your industrial computing and data handling capability.

Data Acquisition Services

The rtxd includes a Data Acquisition Service (DAS) providing time-deterministic data logging capability allowing time-stamped data dictionary entity values to be streamed to specified TCP/IP sockets. Using the DAS requires no user application code and does not interfere with real-time performance. The DAS enables the logging frequency to be configured for each data dictionary item on the fly.

Why a "Real-Time" Linux Server?

Industrial computing and data handling is an inherently frequency-based operation. At some frequency, often involving multiple rates, input channels are read, algorithms are executed, and outputs are sent. This frequency-based computing approach is fundamental to high-performance real-time control and is illustrated in Figure 7. At the beginning of each time step, the computer generates an interrupt or other entry point to start computational tasks, e.g. read inputs, run algorithms, write outputs. This set of computation repeats at every step so long as the system is operational.

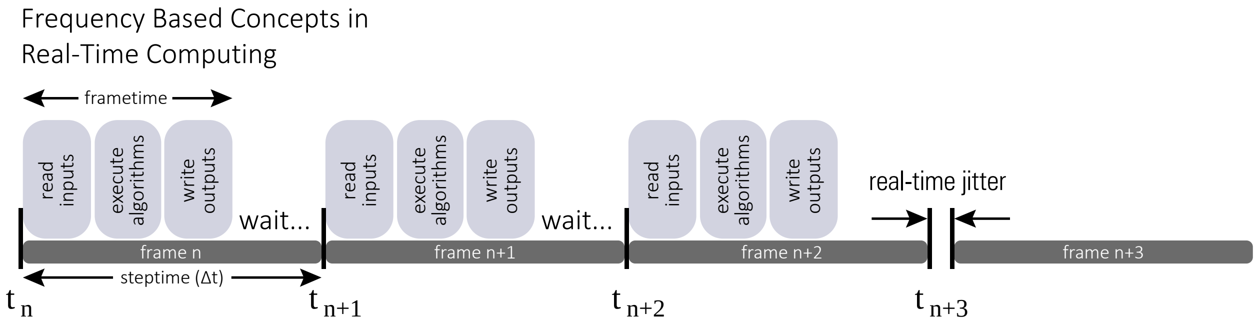


Figure 7 – Traditional Data Interfaces and Modern Industrial Ethernet Technologies

Most Industry 4.0 computing and data handing deployments won't be used for high-performance real-time control so why use a real-time Linux server?

An industrial computing and data handling deployment may have many functions, analysis, and data handling tasks that won't operate in a real-time, once-per-step type of operation. Many tasks and functions will be asynchronous and execute on command or in response to a condition or alert.

Using the real-time Linux operating system with a real-time executive, such as the rtx daemon, provides a set of powerful computing services that, amongst many other capabilities, sets a high-resolution time reference for your computing environment. Accurate, microsecond-resolution time measurement of events such as time of task initiation, time to complete task execution, and statistical variance of task behavior can be very useful for identifying and solving anomalous, difficult to find bugs in your operational systems. Real-time computation and analysis is crucial for effectively monitoring data throughput of mixed Industrial Ethernet and legacy industrial serial/network interfaces, e.g. Modbus, CANOpen, etc.

Beyond precision timing resolution for events and computation, real-time computing methods also bring another powerful benefit that ties specifically to computing costs. The world of real-time computing has maintained a built-in incentive to squeeze more computation out of a fixed computing budget. This may involve getting more control function out of a fixed microcontroller or getting higher fidelity dynamics within a real-time simulation. The result is that the real-time computing world has been focused on finding methods, algorithms, and technologies that get more out of the same CPU. A good example of this can be seen with the use of shared memory structures for communication between processes (applications). Using shared memory structures allows data to be moved from one location to another with minimal latency. This means your CPU is tied up for a shorter duration and can return to executing your algorithms. Using files and disk writes to share data between applications can slow down computation by 10x to 1000x and directly increases the computational cost for your industrial computing capability.

Network, Databus, and Serial Communications

The rtxd includes software support for a wide range of industrial, automotive, and aero/defense serial, databus, and network protocols. Supported communications include:

- UDP Ethernet (custom datagram)
- Modbus TCP
- Modbus RTU
- Ethernet/IP
- CANopen
- RS-232/422/485 (custom datagram)
- MIL-STD-1553
- MIL-AERO 1394
- ARINC-664
- ARINC-429
- ARINC-717
- SpaceWire
- More added each quarter

Configuration for each protocol is handled using CSV configuration files and data dictionary shared memory structures to contain the streaming network data. Configuration includes packing and unpacking of data items into messages, message scheduling, and other protocol-specific properties.

Compatible Modeling Tools

The rtxd supports a wide range of popular modeling tools including Matlab/Simulink, ADSIM, most tools that support the FMI/FMU model exchange standard, and any other Linux compatible software that can be linked and called via the rtxd main entry point functions, e.g. initialize(), step(), terminate().

Real-Time Linux Comes of Age

The key difference between a real-time operating system, e.g. VxWorks, QNX, RT Linux and a non-real-time operating system, e.g. Windows, Linux, Pharlap is the level of real-time jitter. As illustrated in Figure 7, real-time jitter is the time-based error between when the operating system begins a computation “frame” and the absolute start time.

The commercial Real-Time Operating System (RTOS) market is big business with serious industry players, e.g. Intel. Ever since its release in 1991, Linux has been hailed by many thought leaders in the computer science community as an existential threat to the RTOS market. In the early 2000's, Linux saw increased application for consumer embedded electronics devices. The commercial RTOS industry exposed the poor real-time performance exhibited by Linux and was able to draw a line in the market for where Linux would begin stealing market share and where commercial RTOS would continue to dominate and generate significant annual software licensing and support revenue.

In 2004, the Linux community released the "RealTime Preempt" patch, or RT Preempt. The Linux operating system, with its generic kernel, exhibits poor real-time performance as measured by real-time jitter. A key function of any real-time operating system is to allow frequency-based computing where a set of algorithms and/or data handling tasks are performed at every step, and executed at some frequency. Real-time jitter represents the time-based error associated with the operating system's ability to begin computation exactly at the beginning of every step. Today's commercial RTOS offerings typically perform with 3µs to 5µs of real-time jitter at three standard deviations. Traditional Linux, depending on the kernel configuration, could see real-time jitter as high as 500,000µs. RT Preempt dramatically improved the performance of open source Linux. Today, well-configured open source Linux with RT Preempt patch installed performs within the 3µs to 5µs jitter at three stddev range of commercial RTOS offerings. Figure 4X compares the real-time jitter of open source RT Preempt Linux to a high-performance commercial RTOS.

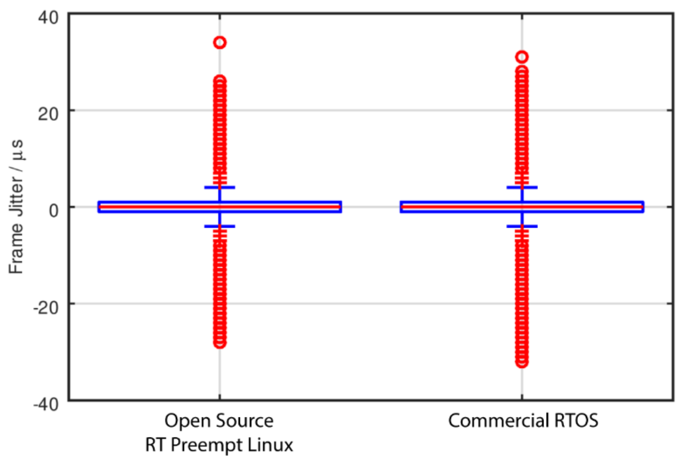


Figure 4X – Box Plots of Real-Time Jitter for Open Source Linux Versus Commercial RTOS

As a result of the improved Linux real-time capability, you no longer need to bear increased OS software costs to utilize a high-performance real-time operating system and the dynamics of industrial software begin to steadily change.

Go here to learn more about the ADEPT Framework: https://www.adi.com/products/adept-framework/

Go here to learn more about the rtx daemon: https://www.adi.com/resources/technical-library/