Industrial Automation

Tech Note 16 Crimson® 3.0 Runtime Overview

Abstract

This document explains how the Crimson® 3.0 runtime schedules the different activities that it has to perform, and hopefully gives some insight into how system performance will vary with load. It assumes a basic understanding of the Crimson environment, and some knowledge of real-time operating systems.

Product Families

G3 Series HMI / G3 Kadet HMI / Graphite® HMI / Modular Controller / Data Station Plus / ProducTVity Station

Use Case: Understanding the Crimson® 3.0 OS

How the operating system functions and how specific communications items can affect (perceived) overall system performance.

Required Software:

Crimson 3.0

Operating System

The operating system used by Crimson 3.0 was developed in-house by Red Lion Controls. Its basic unit of execution is referred to as a task. Each task is assigned a priority, which must be unique, and used to place the tasks in an ordered list. The basic scheduling rule is that the highest priority task wanting to run will be allowed access to the processor, with the lower priority tasks only being executed when the higher priority tasks have entered a “waiting” state. Tasks may either wait for a specific period of time, or may wait on one of a number of synchronization objects, such as semaphores, mutexes and events. Synchronization objects are used to control access to shared resources, and to indicate the result of I/O operations. A communications task may, for example, enter a waiting state until a character is received on a serial port. Or, the user interface task may enter a waiting state for a given number of milliseconds to allow other tasks to run.

Crimson Task Priorities

Crimson's runtime software has the following tasks:

Priority	Name	Operation
0	BASE	Does nothing except loop while system is otherwise idle
7	DISPATCH	Group of tasks, run user programs set to execute in background mode
12	PROM	Responsible accesses to Flash Memory
14	SYNC	Implements FTP client for transfer of files to and from remote devices
15	SQL	Implements SQL synchronization
17	FTPSERV	Implements FTP server for transfer of files to and from remote devices
18	STICK	Manages USB Stick access
19	PDF	Implements PDF file reading from the memory card
20	TIME	Implements Simple Network Time Protocol client and serve
22	MAIL	Implements the Simple Mail Transfer Protocol client
24	OPC	Implements the OPC server
26	W3SERV	Implements the web server.
28	SCANNER	Scans tag alarms, triggers and events
30	PERSIST	Manages retentive tags and events
32	LOGSAVE	Writes trending data from internal memory to CompactFlash
40	COMMS	Group of tasks, one for each comms port to implement each protocol
76	CONTROL	Crimson Control
80	USER	Responsible for Display Pages
82	QUICK	Manages Quick Plot Tags
84	LOGSCAN	Responsible for creating Data Logs
85	LOGPOLL	Adds logged data to a data log
86	LOGSAVE	Saves the log to memory card
88	MUSIC	Plays tunes on the sounder using RTTTL
90	COMMS	Responsible for MC Rack Comms driver
100	SYSTEM	Starts and stops other tasks during power-up and database updates
150	FAT16	Responsible for synchronizing FAT16 accesses
160	USBHOST	Responsible for USB Host port
200	XPSER	Accepts programming and processes requests over serial
250	XPNET	Accepts programming and processes requests over TCP/IP
300	TCPIP	Handles the transmission of IP packets to all interfaces
320	USB Scsi Driver	USB SCSI driver
325	XPUSB	Accepts programming and processes requests over USB
330	USBDEV	Responsible for USB Device port
340	ETHERNET	Handles the reception of IP packets from the primary Ethernet interface

Many of these tasks spend most of their time either idle or waiting for network requests, with the majority of the work in the system being performed by W3SERV, COMMS and USER. Of these tasks, you will note that USER has the highest priority, meaning that while it is executing, no application-level communications activity can take place[1]. Most screen updates take little time, so this is rarely an issue, but the operating system in any case provides a throttling mechanism whereby the USER task will yield for 5ms for every 50ms it executes. This allows COMMS tasks to continue to operate even when complex screens are being drawn.

COMMS Tasks

The COMMS tasks are responsible for transferring data between external communications devices and the data server, which is where the system stores the current data values.

COMMS Scheduling

Each COMMS task is responsible for a single comms port object. Such an object might be a physical serial port, or a logical connection on the IP network. The number of COMMS tasks per device will vary, but allow support for up to four serial ports, up to 10 IP protocols, and multiple comms expansion cards/modules. If a slave protocol is being used on the associated port, the task transfers control to the protocol driver, which typically sits there and responds to requests as they arrive. Master protocols use an object known as a driver proxy to schedule the comms requests which are actually carried out by the driver.

The master driver proxy implements a polling system in order to service outstanding comms requests. At the highest level, the proxy cycles around each remote device for which it is responsible. It then performs any read requests required for that device, interleaving queued write requests so as to give the latter priority without preventing the former from happening. Read requests are generated based on comms blocks allocated by the configuration software—the blocks laid out so as to group similar registers into sensible groupings to optimize comms on those devices that allow multiple contiguous registers to be read in a single request. For devices which work on “named” data items (ie. those without the orders implicit in most PLC tables) the blocks merely serve as somewhere to store the data, with each read being performed in a single request.

Each request issued by the proxy is sent to the protocol driver, which will typically transmit a frame and wait for a reply. This process may take anything from one to a few hundred milliseconds, during which time the associated COMMS task will be in a waiting state, allowing other tasks to run. Replies from expansion cards such as the Profibus or CAN card are typically processed first at the interrupt level, with the associated task then being signaled via an event object. The task then runs as soon as it is able, and processes the data. It can be seen that the exact timing will vary somewhat according to the activity of higher priority tasks, with most of the variation coming from the USER task.

After the proxy has completed its scan, Crimson tells the operating system to put the task into a waiting state for enough time to ensure that the task has yielded the processor for at least 50ms during the scan. For example, if the task has executed for 5ms, been preempted by higher priority tasks for 20ms, and slept for 30ms in the last set of updates, the operating system will put the task into a waiting state for a further 20ms to allow a total of 50ms for lower priority tasks. This means the comms scan for that task will take 5ms + 20ms + 50ms = 75ms, with the 20ms being variable according to USER task activity.

[1]Comms activity handled at the interrupt level, such as medium access control on DH-485 and similar networks can still occur, but application-layer requests will not be generated, nor will replies be processed.

The single largest factor that causes perceived system slowness is the amount of time it takes to gather all of the communications data. This document covers the variables affecting the speed at which the data can be gathered, as well as some ideas that can be used to maximize overall communications speed.

COMMS Blocks

COMMS blocks are internal lists of external data references which are created based on data required at the time. Below is a list of reasons data will be included in the current scan, which will create COMMS blocks containing the required data.

Tag Data

1. Used as the source (right-hand side) of a Gateway Block.
2. With Alarms or Triggers.
3. In the Contents of a Data Log.
4. In the Contents of a web page with Prefetch Data set to *Yes*.
5. On the currently active Display Page.
6. Used in a Program (taking into consideration the External Data setting of the program).
7. Used as the Source, scaling parameters, SP Value, Limits, Alarm, or Trigger parameters of another tag (which is in the current scan for one of the previously mentioned reasons).

Directly Referenced Data

1. Used as the source (right-hand side) of a Gateway Block.
2. On the currently active Display Page.
3. Used in a Program, taking into consideration the External Data setting of the program.
4. Used as the Source, scaling parameters, SP Value, Limits, Alarm, or Trigger parameters of a tag (which is in the current scan).

COMMS Scan (Single Device)

A common question is how often the data from a device updates. This is a variable and will depend on several factors.

1. The number of data items the unit needs from the device.
2. The number of requests required to gather all of the data. This may vary based on:
   a. The protocol being used.
   b. Settings within the protocol/device selection:
      1) Comms Delay
      2) Spanning Reads
      3) Frame Register Limits (for Modbus based drivers).
3. The response time of the device.
4. If a token passing protocol, such as DH485 or BACnet MS/TP, is in use; the time to attempt to pass the token to all of the other possible devices (if all of the network devices are not configured correctly for their network) will also incur a small delay.

COMMS Scan (Multiple Devices on One Serial Port)

When there is more than one device configured on a serial port, they are serviced sequentially. This means that all of the Single Device variables need to be considered for each device, as well as some additional variables.

1. Are any of the other devices offline? Offline devices will cause a delay; the amount of delay will vary based on the timeout of the driver. Some protocol specifications include a pre-defined timeout; others are adjustable at the driver or device level in Crimson.
   - **a.** If the driver uses a ping, there will be a 1x timeout delay per offline device if the device was offline during the previous scan.
   - **b.** If the driver does not use a ping register, or the device went offline since the last comms scan, there will be a 3x timeout delay. This is due to the system retrying 2x additional transactions if a timeout occurs.
2. Do any devices have Preempt Other Devices set to *Yes*? If so, any time there is a write queued for a device with this setting, that write will take precedence over all other communications on the port. It is recommended that this setting is left at *No*.

COMMS Scan (Multiple Devices on One Ethernet Protocol)

When there is more than one device configured for an Ethernet Protocol they are serviced sequentially, one at a time. This means that all of the Multiple Devices variables need to be considered for each device, as well as some additional variables.

1. Link Type:
   - **a.** Dedicated Socket (limited to 4 devices per Protocol): this setting creates a socket connection that is not closed unless there is a communications failure, or requested by the other device.
   - **b.** Shared Socket: this setting opens a TCP socket connect, transfers data, and then closes the connection. If more than 4 devices are configured as Dedicated Socket, the first 3 devices the unit establishes connections with will be dedicated connections; the remaining devices will share a Shared Socket connection.
     - Connection Backoff: the (minimum) amount of time between closing the connection with a device and opening another connection to the same device.
2. Is ICMP Ping enabled? If so the system will issue an ICMP ping to determine if the device is on the network before attempting to open a socket. This will avoid waiting the Connection Timeout before moving on to the next device.
3. Connection Timeout: the amount of time that the system will wait when attempting to establish a socket connection.
4. Transaction Timeout: the amount of time the system will wait for a response to a transmission. If there was previous communications, 3 attempts of the same transmission will be attempted before determining a communications error and moving on to the next device.

Write Processing

Devices running Crimson 3.0 can handle writes to external devices in two distinct ways. The non-transactional technique involves looking for changes in data items, and then writing those changes as they are identified. This technique has a downside, in that if two sequential changes occurred between comms scans, only the latter value would be written to the external device. For most data items, this will not cause an issue, but it can produce difficulties in several distinct situations...

1. A panel would sometimes be configured to write a bit in the PLC to an “on” state when a key was pressed, and to then write it to an “off” state when the key was released. If the release occurred too quickly, only the “off” state would be written, and the required effect would not be achieved.
2. Some devices—typically servo or variable-speed drives—have registers that are used to issue commands. Distinct values must be written to these registers in a specified sequence in order to produce the desired behavior. Unless the writes were separated by enough time to allow each individual value to be written, the command sequence would not be transferred correctly.
3. Some devices have write-only registers that can be changed by the remote device, but which the HMI needs to repeatedly set to the same value. For example, a command register may need to be set to ‘1’ to clear an alarm state, and then set to ‘1’ again to clear a subsequent alarm. Paradigm HMIs would not perform the second write unless the data was first set to another value.

Using Transactional Writes in Crimson solves all of these problems. This means that writes to remote data items within a single device are performed in exactly the order that they were requested by the controlling database. It also means that, if the data item in question is set to Write Only, repeated writes of the same value can be performed.

This process is managed by a queue that can hold up to 512 pending writes per devices. Writes will be added to the queue, and then processed in sequence by the comms drivers. If the queue grows beyond 100 items, Crimson will invoke a technique called write throttling, adding a small delay after each write request until the queue is reduced to a manageable level. The system is designed to maintain the queue at 32 writes, and to turn off throttling if the queue empties below 16 entries.

If a device goes off-line, the write queue is emptied, with the written values being transferred into the data items within the terminal. This means that when the device comes back online, the last written values will be transferred, but any sequences contained in the queue will be lost. This compromise is required to ensure that the queue does not get out-of-control in off-line situations. If a specific sequence needs to be sent to a device when it comes back online, use a trigger based on IsDeviceOnLine to invoke the writes as required.

One artifact of the queued approach is sometimes seen in demo databases, where a register might be incremented on every display update. Crimson will write the changes to the remote device, but will ensure that each value is sent individually. If the comms driver is configured at a low Baud rate, the queue may grow if the driver is not able to keep up with the requests. The queue will stabilize as write throttling kicks-in, but the remote device will typically be a couple of dozen values behind those being displayed by the HMI

The good news is that this is rarely as issue with real applications, as they do not typically perform writes at such a frequency. In fact, Crimson’s write processing makes nearly all databases behave more intuitively, with the required data being written to the selected devices in exactly the required order. Push button replacement is made much easier, and operations that require data items to be written and then strored by a command register can now be performed without the need for Sleep statements.

Maximizing Communications Speed

There are a few methods that may be employed to improve the communications speed.

1. Optimize Comms blocks by consolidating the memory locations in the external device.
    - Many enumerated (non-tag name) protocols have the ability to read blocks of data starting at a particular register. If all of the required data is consolidated into a contiguous chunk of memory, then fewer transactions will be required to gather all of the data.
2. Minimize ‘clutter’.
    a. Do not log tags that are not required.
    b. Do not create alarms/events/triggers that are not required.
    c. Do not include items in gateway blocks that are not required.

3. Set programs’ External Data option based on the needs of the program.
    a. *Read When Referenced* if execution speed of the program supersedes overall system speed.
    b. *Read When Executed* if overall comms speed supersedes program execution speed.

4. Use separate devices under separate Protocols referencing the same device. *Ethernet only, requires that the external device accepts multiple connections from the same master.
    a. Use one device for critical data (alarms, etc…).
    b. Use the other for the less critical data.

5. Disable Transactional Writes.
6. Minimize amount of data on the page; remove excess data points, potentially adding a page that can be shown as a popup with further details.

Alarms, Events, and Triggers

Alarms, events, and triggers are scanned every 100ms. This means that the alarm/event/trigger “condition” must exist for at least 100ms in order to be recognized, and this condition must be non-existent for 100ms prior to being recognized another time.

Delays

When a delay is added to an alarm, event, or trigger; it applies to “both sides”’ of the condition. When adding a 5000ms delay to an alarm, the condition must first not exist for 5000ms, and then exist for 5000ms, prior to being considered in the alarm state. Likewise, the condition must be non-existent for 5000ms before it will be recognized another time.

User Programs

User programs, found in the Programming section of Crimson 3.0, are event driven. Events such as the global actions, button presses, tag events, or other programs are used to begin program execution.

Program Properties

External Data

Read When Referenced: External data used by the program will be added to the comms scan whenever the program is referenced. If the program is referenced by a display page, the data will be read when that page is displayed; if the program is referenced by a global action or a trigger, the data will be read at all times. This is the default mode, and is acceptable for all programs, except those that use very large amounts of external data.

Read Always: External data used by the program will be read at all times, whether or not the program is referenced. This means that the program will always be ready to run, and that the operator will not see the “NOT READY” message that might otherwise occur when the program is first referenced. The downside of this mode is that comms performance may be reduced if large amounts of data are referenced by the program.

Read When Executed: External data used within the program will be read only when the program is invoked. The program will wait for the period defined in the timeout property for such data to be available. If the data cannot be read—perhaps because a device is offline—the program will not execute. This mode is typically used with globally-referenced programs that consume large amounts of data that would otherwise slow down the communications scan.

Run Anyway

This setting allows a program to execute even if all of the external data is not available. If this is disabled a program will not execute if any of the required external data is missing. If data integrity is essential, but the program relies on external data from other devices, offline device detection (IsDeviceOnline or GetDeviceStatus functions) along with the use of Run Anyway is recommended.

Program Options

There are 4 Execution Task options available for user programs: Same as Caller, Background 1, Background 2, and Background 3. This can be set on the Properties tab of the program, along with the External Data and Run Anyway settings.

Same as Caller

This selection executes the program in the same task that it was called from. It could be the UI task in the case of a button press or in a background task if it was called from a program executing in the background.

Background Execution

Crimson 3.0 provides the ability to execute program in the background. While this concept is simple enough in principle, the various ways in which background and foreground programs can be combined can cause confusion in the mind of the database author. This section explains how the background execution process operates, and attempts to dispel some of the confusion.

Background tasks

Crimson implements background execution by means of three background tasks. Each task has a queue, which is used to hold a list of programs to be executed. When a program marked for background execution is called, it is simply added to the queue, and then the calling program continues as if the background program had already completed. Meanwhile, the background task will pull the program from the queue and execute it. A program cannot be in the queue more than once, so a second attempt to execute a background program before it has begun execution will be ignored. (The phrase “begun execution” is important: The program is removed from the queue before being executed, so a program can add itself back to the queue without worrying about this limitation blocking the operation.)

Task Priorities

Each background task has a priority. Task 1 is the lowest priority, followed by task 2 and then task 3. Tasks on a higher priority task will block tasks on a lower priority thread, although a mechanism is provided to ensure that even long-running programs yield the processor for 5ms out of every 50ms of execution. It is generally not necessary to use multiple background tasks, unless your application really needs several tasks to be running at once. Excessive background processing typically indicates that you should be using techniques like gateway blocks and triggers to perform the equivalent operation.

Nested Execution

Most misunderstandings related to background tasks come from situations when one program calls another, and when either or both of those programs are set for background execution. The most important thing to note is that the opposite of background execution is not really foreground execution: It is the execution of a program by the same task as that of the calling program. For example, most programs are invoked by the user interface task in response to some user interaction. Those programs run on the user interface tasks, and other non-background programs invoked by those programs will run on the same task. If such a program then invokes a background program, it will be queued and executed in the background. If that background program then runs a non-background program, it will be run by the background task, but it will be called directly from the calling program and not via the queue mechanism. The table below explains the various scenarios that can occur when Program1 calls Program2…

Program1	Program2	Result
Foreground	Same as Caller	Program1 will wait for Program2 to execute.
Foreground	Background	Program1 will continue without waiting for Program2, which will be added to the appropriate queue. When it gets to the front of the queue, the associated background task will execute it. Program1 may or may not have completed by this point.
Background	Same as Caller	Program1 is running in the background, and will wait for Program2 to execute. The queuing mechanism will not be used. The associated background task will not be available to run another program from its queue until Program1 (and thus Program2) has completed.
Background	Background (Same Task)	Progam1 will continue without waiting for Program2, which will be added to the queue of the same task executing Program1. Program2 cannot execute yet as the background task is still busy. When Program1 has finished, and when Program2 moves to the front of the queue, it will be executed.
Background	Background (Different Task)	Progam1 will continue without waiting for Program2, which will be added to the queue of a different background task. Program2 may execute right away or later, depending on the state of the second task's queue.

Watchdogs

The Crimson runtime is protected by two watchdog timers, either of which can cause a reset of the system if the database appears to be "hung". The first watchdog will reset the system with a code of 0x2F if the lowest priority task in the system cannot run for 60 seconds. This task runs at a priority lower than even the background tasks, and therefore could be blocked by background programs, were it not for the 5ms yield process described above[1]. The second watchdog will reset the system with a code of 0x2E if the user interface tasks cannot update the display for 60 seconds. This will occur if a long program is invoked in response to a user action. In general, 0x2E indicates a programming error in the database, and more specifically, a situation where background execution should be used to keep the display updating. A trap of 0x2F should not occur, and may indicate a problem in the Crimson software which should be reported to Red Lion.

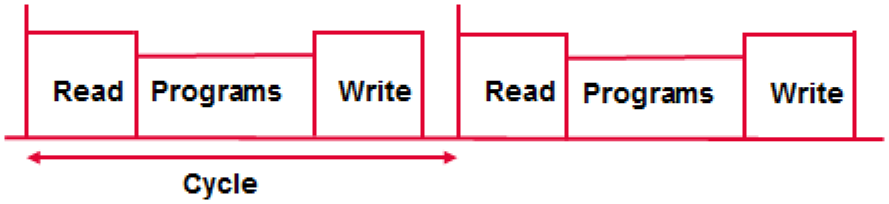
[1]There is a potential problem if multiple background tasks are in use: The 5ms yields may not "line up" and may trigger a watchdog failure when under heavy load.

Crimson Control

Unlike the event driven User Programs, the Crimson Control project is cyclically driven. All external data required for the project is permanently added to the comms scan. Lack of external data due to lack of communication does not affect the project's execution, it continues to run with the last know values of the external data items. The cycle is not transactional, as shown below the external data (I/O) is read at the beginning of the cycle and written at the end.

Cycle Execution

- A project is a list of programs executed sequentially according to the following model:
  - Begin cycle
    - *Read I/O from data server*
    - *Execute first program*
    - *…*
    - *Execute last program*
    - *Write I/O to data server*
  - Wait for cycle time to elapse
  - End cycle



- To change the execution order click on Project (Execution tab), then rearrange the programs
- **While the Cycle Time is adjustable to as low as 100ms, the reaction time of the I/O is still very much dependent on the overall system load.**

For more information: http://www.redlion.net/support/policies-statements/warranty-statement