

Introduction

This document describes the internal structure of Studio, explaining how data flows through the runtime module and how they are executed. A good understanding of the information covered in this document is important to avoid unexpected behavior when developing complex applications and to guarantee the best performance during the execution of the application.

This document presumes that the reader is familiar with the basic components of Studio and how to configure them.

Internal Structure and Data Flow

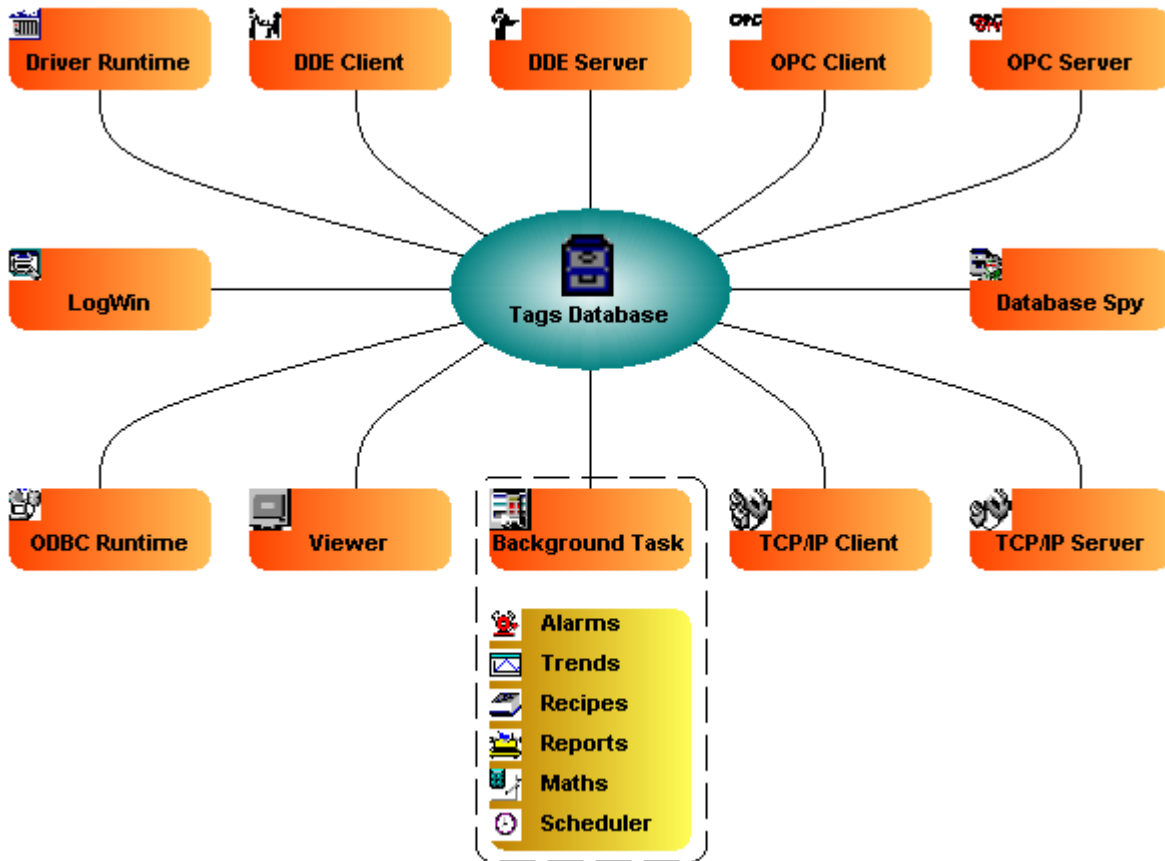
Studio is composed by the following runtime tasks (threads):

- **Background Tasks:** Execute the scripts configured in the *Math* and *Scheduler* worksheets and manage the settings configured in the *Alarm*, *Trend*, *Recipe* and *Report* worksheets.
- **Database Spy:** Debugging tool used to: Read data from the tags database (for example: tags values); Write data to the tags database (for example: tags values); Execute functions and/or expressions for testing purposes.
- **DDE Client:** Manage the DDE communication messages with any local/remote DDE Server, according to the settings configured in the DDE Client worksheets.
- **DDE Server:** Manage the DDE communication with any local/remote DDE Client.
- **Driver Runtime:** Manage the reading/writing commands configured in the *Driver* worksheets.
- **LogWin:** Debugging tool used to trace messages generated from the other tasks.
- **ODBC Runtime:** Manage the ODBC data communication with any SQL Relational database, according to the settings configured in the *ODBC* worksheets.
- **OPC Client:** Manage the OPC communication messages with any local/remote OPC Server, according to the settings configured in the *OPC Client* worksheets.
- **OPC Server:** Manage the OPC communication with any local/remote OPC Client.
- **TCP/IP Client:** Manage the TCP/IP communication messages with a remote *TCP/IP Server* module (from Studio), according to the settings configured in the *TCP/IP Client* worksheets.
- **TCP/IP Server:** Manage the TCP/IP communication messages with a remote *TCP/IP Client* module (from Studio).
- **Viewer:** Execute the scripts configured on the screen (On Open, On While, On Close, Command, Hyperlink, etc) and updates the objects on the screen.

All runtime tasks exchange messages directly with the **Tags Database**. The **Tags Database** is the “heart” of Studio and it keeps the current values and status of each tag configured in the application. The tasks never exchange data

each other directly. They always send/receive messages to/from the **Tags Database** and it manages the data flow between all the modules.

The following diagram shows the internal structure of Studio, where all the runtime tasks exchange data directly with the **Tags Database**:



For instance, if the **Driver** reads a new value from the PLC, it updates the value of the tag associated to this information in the **Tags Database**. If this information must be shown on the screen, the **Tags Database** will send a message to the **Viewer** with the new value of the tag, so the **Viewer** module will update this information on the screen.

Notice that the **Driver** didn't send the message directly to the **Viewer**. Also, there is not pooling between the tasks. As soon as any information is updated on the **Tags Database**, it will forward this message to all runtime modules which need this information. This behavior allows a high performance for the internal data flow. Also, a new task can easily be included to this architecture, since each internal task (thread) works independently each other but can access any information from any other task, by the **Tags Database**.

Note: The **Tags Database** stores not only the value of each tag, but also the status of all properties associated to each tag (alarm conditioning, timestamp, quality, etc.).

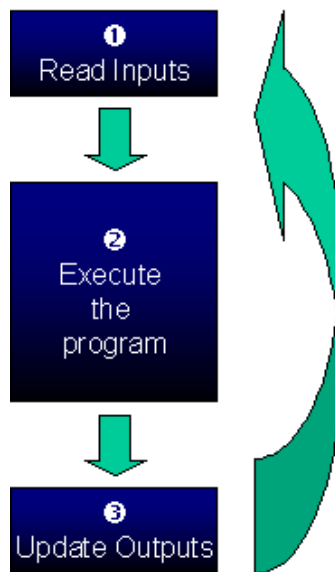
Each task keeps one virtual table with the tags which are relevant for them at the current time. The **Tags Database** uses this table to decide which information must be updated in each task. For instance, the *Viewer* module keeps one virtual table with the list of all tags configured in the screens which are open. If any of these tags change value in the **Tags Database**, it will send a message to the *Viewer*. Then, the *Viewer* will update all objects where this tag is configured.

Tips: It is important to keep in mind that the *Viewer* module updates each object only when at least one tag configured in the object changes value. If a dynamic (for example: *Text I/O*) is configured with a function which does not require any tag (for example: *NoInputTime()*), then the object will not be updated by the *Viewer* because there is no tag associated to the object.

Execution (Tasks switching)

Studio is a SCADA system composed of several modules and they must be executed simultaneously. Based on the multitasking concept, each runtime task (**Viewer**, **Driver**, etc) is a thread and the operating system switches automatically from one thread to another.


There is a common misunderstanding between the execution of a SCADA system with the execution of a PLC program. In a PLC program, there is a simple loop as shown in the following diagram:



For a SCADA system, there is not only one program to be scanned. There are several tasks running simultaneously and most of them can read and write data. The data (value of the tags) are modified continuously during the execution of the tasks. Therefore, the diagram above is NOT applied for a SCADA system.

Studio has only one process (**Studio Manager.exe**). When the runtime application is executed, this process starts the **Tags Database** and all the runtime modules configured in the application. The user can configure which modules should be started during the runtime (for example: **Viewer** and **Driver** runtime modules).

Each process keeps a list of *active* threads for the operating system. Actually, each process can activate and inactivate each thread during the runtime, according to the algorithm of each process. Also, each thread has a priority value, configured when each thread is created. The operating system keeps scanning all threads active at the current time. The threads with higher priority value are executed first. While threads with higher priority value are active, the threads with lower priority value are not executed at all. If there is more than one thread with the same priority number and there is not any other thread with higher priority, the operating system keeps switching through the threads with the same priority.

 **Note:** All threads of Studio are set with priority number 7 (THREAD_PRIORITY_NORMAL). Most of programs have this priority number. Real-time programs (for example: *SoftPLCs*) and *Device Drivers* have higher priority numbers (THREAD_PRIORITY_HIGHEST). However, they must provide a mechanism to keep them inactive for some time; otherwise, the threads with normal priority will not be executed. Studio uses the UNICOMM.DLL library for serial drivers. This library creates a thread with THREAD_PRIORITY_HIGHEST that keeps it inactive (sleeping) until data arrives in the serial channel. When new data is detected in the serial channel, this thread wakes up and transfers the data from the operating system buffer to the thread buffer, in order to be treated by the Driver. This is the only thread with highest priority created by Studio.

Each thread cannot be kept active all the time, otherwise the CPU usage would consistently be 100% – a situation that must be avoided. Each program provides its own mechanism to avoid keeping each thread active all the time. The following text describes some parameters which are used to explain the mechanism used by Studio to avoid having all threads active all the time.

- **TimeSlice** (from the operating system): The operating system switches automatically between all the active threads. By default, the operating system executes each thread for about 20ms and then switches to the next active thread. Therefore, the operating system does not keep executing the same thread for more than 20ms if there are other active threads with the same priority number waiting to be executed.
- **TimeSlice** (from Studio): In addition to the TimeSlice from the operating system, Studio sets a TimeSlice time for each thread. The TimeSlice time can be configured for each thread of Studio (except for **Background Task**) and it sets the amount of time that each thread remains continuously active. When a thread becomes active, the operating system can switch to it.
- **Period** (from Studio): This parameter can be configured for each thread of Studio (except for **Background Task**) and it sets the maximum time that each thread will keep inactive.

The **TimeSlice** and **Period** parameters from Studio can be set in the **Program Files.INI** file stored in the **IBIN** subfolder of Studio. The default values are listed below (the values are set in milliseconds):

```
[Period]
DBSpy=1000
UniDDEClient=200
UniDDE=200
Driver=20
LogWin=100
UniODBCRT=100
OPCClient=20
OPCServer=20
TCPClient=100
TCPServer=100
Viewer=50
```

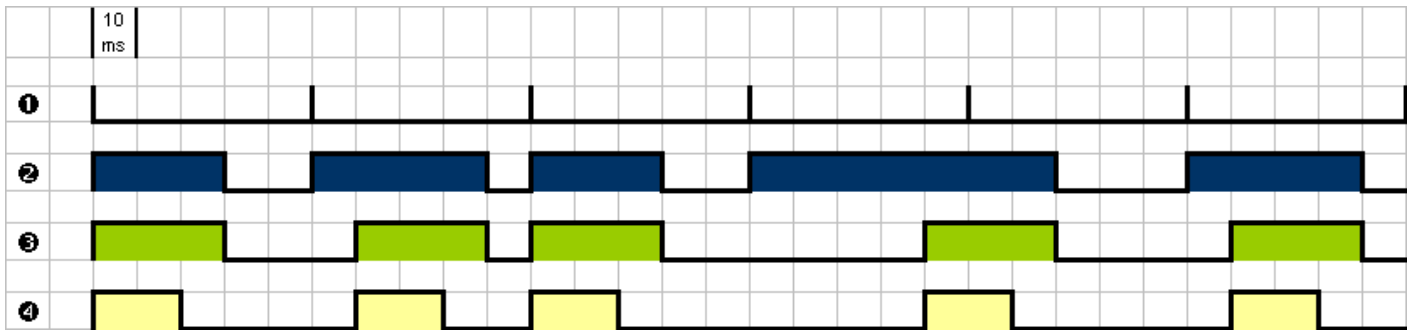
```
[TimeSlice]
UniDDEClient=100
Driver=10
OPCClient=10
OPCServer=10
TCPClient=200
TCPServer=200
Viewer=200
```

➔ **Caution:** The default settings should not be modified, unless strictly necessary. The wrong configuration of these parameters can result in malfunctioning of the whole system (for example: CPU usage at 100%) and/or bad performance of some tasks.

The diagram below illustrates the execution of a generic thread (for example: **Viewer**). In the example, the **Period** time was set in Studio with the value 50ms (signal ❶) and the **TimeSlice** time was set in Studio with the value 30ms (signal ❸). The signal ❷ shows when the thread is active for the operating system and the signal ❹ shows the execution of the thread itself.

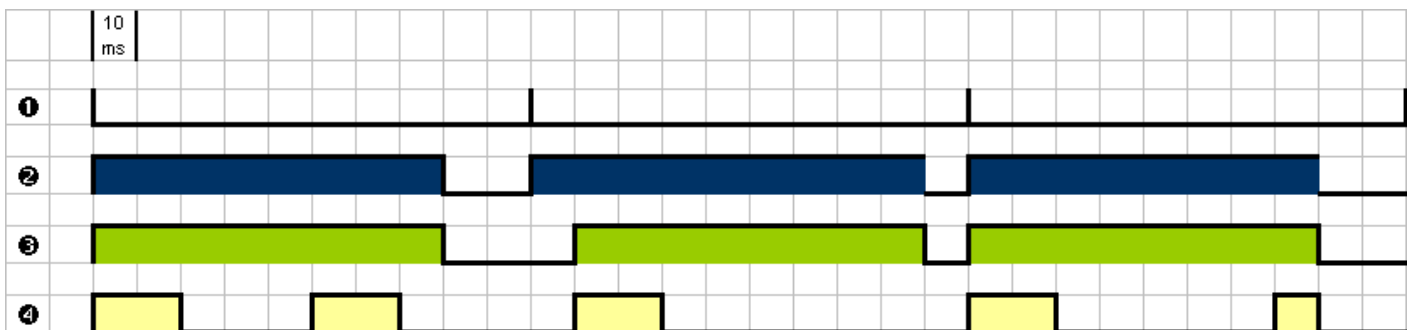
Studio generates a **Period** message each 50 milliseconds (signal ❶). Whenever this message is generated, its thread turns to the active state and **becomes this state until the TimeSlice time (from Studio) is over**. Then, the thread will remain inactive until the next **Period** message is generated by Studio (signal ❶).

While the thread is active, the operating system is in charge of executing it. The fact that the thread is active does not mean that the operating system will start executing it immediately – it may be executing other threads when this thread became active. For example, when the thread is executed by the operating system, the **TimeSlice** timer will start counting. The thread is executed for 20ms (**TimeSlice** from the operating system). Then, the operating system switches automatically to the next active thread (for example: **Driver**) and so on.



In the previous example, the **TimeSlice** time from Studio was set with the value 30ms which means that the thread is not supposed to be executed more than once in each **TimeSlice** of Studio. However, if the Studio **TimeSlice** is set with higher values, the same thread is likely to be executed more than once in the same **TimeSlice** time.

In the next example, the **Period** was set to 100ms and the Studio **TimeSlice** was set to 80ms. Note that the thread can be executed more than once during the same **TimeSlice** time. When the Studio **TimeSlice** time is over, the thread execution is interrupted. Regardless of the Studio **Period** and **TimeSlice** settings, the thread is not executed contiguously for more than 20ms, due to the **TimeSlice** time from the operating system.



In the previous example, while the **Viewer** thread is not being executed, the CPU may be executing any other thread or may be idle (if there is not any other active thread to be executed). It is important to remember that the **Period** and **TimeSlice** settings from Studio were created to avoid having all threads active all the time. It would require 100% of the CPU usage – a condition that must be avoided.

While each thread is executed, it must treat its pendent messages. For instance, the **Viewer** module must update the objects on the screen(s) which must be updated. When there is no message to be treated, the thread becomes inactivate and gives the control back to the operating system, which will then immediately switch to the next active thread. Therefore, the thread can interrupt its own execution even before the **TimeSlice** time from the operating system is over. It happens often in real-world applications.

Note: The threads **Database Spy**, **LogWin**, **DDE Server** and **ODBC Runtime** do not have a **TimeSlice** setting from Studio. When each one of these threads treats all pendent messages, they became inactivate until the next **Period** message.

The mechanism above does not apply to the **Background Task** thread. Its mechanism is described in the following paragraphs.

The **Background Task** executes the scripts from the **Math** and **Scheduler** sheets, such as the messages from the **Alarm** and **Trends** sheets. In addition, it executes the **Recipe** and **Report** commands when the **Recipe()** or **Report()** functions are executed during the runtime.

Although **Math**, **Scheduler**, **Alarm** and **Trend** are not threads, it is possible to set their **Period** time in the **Program Settings.INI** file from the **\BIN** subfolder of Studio. The default values are listed below (the values are set in milliseconds):

```
[Period]
Math=100
Sched=50
Alarm=100
Trend= 1000
```

This means that in each 100ms, Studio generates a **Period** message for the **Math** task. In each 50ms, it generates a **Period** message to the **Scheduler** task and so on. These settings should not be modified, unless strictly necessary.

It is necessary to keep in mind that the **Background Task** is a thread with the same priority as the other threads in Studio (**Viewer**, **Driver**, etc.). This means that it will not be executed by the operating system contiguously for more than 20ms.

The **Recipe** and **Report** commands are executed by the **Background Task** when the **Recipe()** or **Report()** functions are executed. These functions are synchronous and once the **Background Task** start executing them, it will not switch to other tasks (**Math**, **Scheduler**, **Alarm** or **Trend**) until the function has been completely executed. **Recipe()** or **Report()** functions usually take a few milliseconds to execute.

Background Task must switch between the **Math**, **Scheduler**, **Alarm** and **Trend** tasks. When **Background Task** switches to the **Scheduler** task, it will not switch to other task (**Math**, **Alarm** or **Trend**) until all **Scheduler** worksheets are executed. After executing all **Scheduler** sheets, the **Scheduler** will not be executed again until the next **Period** message for the **Scheduler** task. The same behavior is applied for the **Alarm** and **Trend** tasks: When **Background Task** switches to each one of these tasks, it does not switch to other tasks managed by **Background Task** until all pendent messages are treated. They will not be executed again, until the next **Period** message for each of these tasks is generated by Studio.

The **Scheduler**, **Alarm** and **Trend** tasks are typically executed in a few milliseconds. However, the **Math** sheets can take a longer time to be executed, due to loops and complex scripts. Therefore, the same mechanism applied to the **Scheduler**, **Alarm** and **Trend** cannot be applied to the **Math** task.

The **Background Task** executes the **Math** sheet for no more than 10ms continuously and switches to other tasks (for example: **Scheduler**). **Background Task** cannot execute the **Math** task again during for the next 50ms. During these 50ms, **Background Task** can execute other tasks (**Scheduler**, **Alarm**, **Trend**, **Recipe** or **Report**). When all **Math** worksheets are fully executed, a new scan of the **Math** worksheets will not begin until a new **Period** message for the **Math** task is generated by Studio.

It is important to emphasize that this mechanism was created to avoid having the CPU usage at 100% all the time.

➔ **Caution:** Special caution must be taken when using the *Math()* function. If this function is configured in a **Scheduler** worksheet, it will be executed by the **Scheduler** task. This means that as soon as it is triggered in the **Scheduler** worksheet, no other task will be executed by the **Background Task** until the entire *Math* worksheet called by the *Math()* function is completely executed. It can take several milliseconds or even seconds, according to the script configured in the *Math* worksheet (especially for loops). If the *Math()* function is configured on any screen (for example: a **Command** dynamic), the **Viewer** thread will stop updating the screen until the *Math* worksheet called by the *Math()* function is completely executed. To avoid this situation, when the **Scheduler** or an object on the screen must enable the execution of a *Math* worksheet, the following procedure is recommended: Set one auxiliary tag with the value 1 (the **Scheduler** or the **Viewer** task will send a message to the **Tags Database** to update the value of this tag). Configure this tag in the **Execution** field of the *Math* worksheet that must be executed. When the **Background Task** scans this *Math* worksheet, it will be executed. Finally, reset this tag in the last line of the *Math* worksheet (writes the value 0 to this tag). Therefore, this *Math* worksheet will not be executed in the next scan, unless the auxiliary tag is set with the value 1 again.

Map of Revision

| Revision | Author | Date | Comments |
|----------|------------------|-------------|--|
| 0 | Fabio Terezhinho | Jan/30/2002 | Initial revision |
| A | Fabio Terezhinho | Feb/04/2002 | Updated the default values for the parameters Period and TimeStamp |
| B | Fabio Terezhinho | Oct/3/2003 | Layout revision |