KNX Runtime Engine

2016

V 1.0.0 A

**Abstract**

This runtime language platform is a tool designed to make advantage of multiprocessing for applications such as scientific modeling and for automating tasks.

In general, this language is designed with high-control applications in mind. That is, this engine is designed for the creation and application of medium to large autonomous systems that facilitates a human operator.

Other viable applications may be in quick-and-dirty graphical applications and as a simple multi-purpose language.

Features include Object Orientated Programming (OOP), a multithreaded node based design as well as a module construction to allow for third party expansion.

The following sections of this document will explain further into the workings of the engine, as well as providing examples and tutorials of some of the more fundamental points of using this tool. For further reference on specific functionality, please refer to the API guide.

*Table of Contents*

|  |  |
| --- | --- |
| Description | Pg# |
| Command Line Options and System Configurations | 1 |
| Architecture |  |
| Node Communication, Hierarchy and Permissions |  |
| Memory Structure |  |
| Scope and Permissions |  |
| Object Orientation and Inheritance |  |
| Data Types |  |
| Keywords |  |
| Flags |  |
| Terminal Features |  |
| Optimizations |  |
| Thread Synchronicity and Safety |  |
| General Syntax |  |
| Program Design |  |
| Operators and Logic |  |
| Versioning |  |
| Platform Dependencies |  |
| Compilation from Source |  |
| KNX SDK |  |
| Building Custom Libraries |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Tutorials |  |

**Command Line Options and System Configurations**

The interpreter initializes with a set of default configurations listed in the table below. These settings may be modified with the passing of **command line arguments**, or special flags set when calling the executable. These flags are prefixed by the ‘-‘symbol to designate a single character modifier, or ‘--’ for a field modifier. The former is used primarily to toggle a setting from its default state, or to activate some operation before the interpreter begins normal operations. The latter maybe followed by an assignment to change a value used by the interpreter.

|  |  |
| --- | --- |
| -v | Displays current interpreter and SDK version |
| -d | Activates debug mode |
| -s | Deactivates system printout |
| -w | Suppresses warning |
| -e | Suppresses errors |
| -x | Prevents interpreter from launching |
| -t | Deactivates tab assist |
| -h | Prints “help” page |
| --tab=[num] | Sets the “tab assist” tab size |
| --maxnode=[num] | Sets soft limit for simultaneous nodes in system |
| --maxmemory=[num] | Sets limit for memory consumption by program |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Architecture**

*What is a node*: A node is a thread wrapper containing information pertaining to its identity, as well as keeping track of its own information structure and interpreter instance. A node will remain alive until terminated by another, or until its parent is terminated. Each node is responsible for its own set of instructions to carry out, and carries its own memory space.

*How do nodes fit into the design*: The first node to be generated by the system is the *root node*, otherwise known as *root0*. This node exists on the main thread of the program. By default, it is also granted access to the *stdio* stream, thereby giving it access to terminal input and output. In addition, it also has extended permission towards all other child nodes in the system.

*Child nodes* may be spawned from parents. These form the branches and leaves of the hierarchal tree. Each child node contains an instance of the interpreter structure in order to keep separate its operations and states from other nodes.

*What is the interpreter structure*: The interpreter structure is a collection of states that track how the execution of code proceeds. This includes the level of encapsulation by braces, brackets or other containment operators. In addition, it also carries flags tracking special states that regulate how code may proceed. For instance, if when using a logical expression a segment of code processed following an asserted *if* executes, the interpreter will know to ignore an immediately preceding *else* statement, and vice-versa.