KNX Runtime Language

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 | 2 |
| Node Hierarchy, Communication and Permissions | 4 |
| Memory Structure |  |
| Scope and Permissions |  |
| Object Orientation and Inheritance |  |
| Data Types |  |
| Operators and Logic |  |
| Keywords |  |
| Macros |  |
| Flags |  |
| Terminal Features |  |
| Optimizations |  |
| Thread Synchronicity and Safety |  |
| General Syntax |  |
| Program Design |  |
|  |  |
| 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 |
| -r [file] | Executes script |
| --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**

*Basic Design*

A key aspect of the engine is its ‘node-based’ architecture. Comprised of a tree of control structures called nodes, a child-parent hierarchy is formed. Starting at the root node, or *node0*, each subsequent node is generated with its own worker thread. In fact, the nodes are primarily threading wrappers, each acting as a standalone interpreting unit.

While any individual node may be, to an extent, independent of others, there are a few exceptions. First off, the life cycle of a node is dependent on the parent. When a parent node is terminated, the first step in its destruction is the collapse of its children sub-trees. This is a recursive process to ensure all memory and threads are successfully freed. In addition, cross communication between nodes is also allowed. Nodes are most closely linked to those in their family branch, but may also request and send data to other non-relative branches. Special permissions may be set to restrict communication access between nodes, similar to the *public* and *private* keywords of other languages. This will be further explained later on.

This brings about the topic of memory space. Each node contains a local memory space, whilst maintaining system-wide access to the global database. By default, all memory access and initialization occurs at the local level first before moving onto the global scope. Communication may exist between nodes, however. All memory, local and global, is protected by a Mutex queueing system. This prevents memory corruption or collisions from occurring during asynchronous operation across multiple threads and cores. This is explained in more detail in a later section.

*Node ID System*

There is a hard limit on the available number of nodes a system can have. By default, this is 500 active nodes at any one time. (Hopefully) this is by far more than enough capacity. The reason for the hard limit stems from the implementation of the ID system.

Each node has several forms of identifications. These are its thread handle, its name hash, its memory address and its ID number. Hard coded in the engine is a set of 500 prime numbers, starting at one and skipping two. This has two useful mathematical properties. The first is the use of memory and processing efficient access authorization. As briefly mentioned before, permissions may be set node-wide to allow or deny access to other nodes, according to the set privacy policy. But how are these permissions held by the system? A simplistic design might create a table to store the ID’s. However, by utilizing semi-prime numbers, that is the product of two or more prime numbers, many ID’s may be stored on a single *size\_t* instance. To determine if a number has been added to the permission, a simple remainder check is performed to see if the semi-prime contains the prime ID of the calling node.

**Node Hierarchy, Communication and Permissions**

Interaction between nodes depends on a small number of factors. The most important is the concept of permissions, or policies established by the node in question in order to regulate external read/write actions on its local memory space. The different policies are listed below:

|  |  |
| --- | --- |
| Public | Accessible to all except blacklisted id’s |
| Private | Accessible to none except the root node |
| Legacy | Accessible to parents only |
| Heritage | Accessible to parents and children only |
| Protected | Accessible only to root and whitelisted id’s |

Unlike common languages these assignments cannot be made to individual memory types as the policy is node-wide. By default the memory policy is set to public. This may be changed by specifying a policy during the call of the node factory *xnode(…)* with an appropriate overload. Permissions may be set by a node via a call to the *setPolicy* function.

Make note, however, that a policy list is NOT preserved when switching to another. This is due to how the rulesets each policy differ. For instance, black listing a set of ID’s within a public policy has the exact opposite effect of the list on a protected policy. The legacy and heritage policy automatically generate and maintain ID permissions.

A small amount of basic mathematics is applied in the identification system. Each node is assigned a prime number upon its registration to the system. The system has a hardcoded set of 500 prime numbers starting at 1 (reserved for the root node), skipping 2 and continuing on. A soft limit may also be assigned to limit the number of active nodes on the system.

Policy ‘lists’ are actually a single value of size *size\_t*. Initialized to 1, these values are used to hold semi-prime numbers. In this way, three operations are possible:

|  |  |
| --- | --- |
|  | Assign ID |
|  | Removal ID |
|  | Check ID |

Note that as the root node has an ID of 1, it has unrestricted access at all times.

Permissions may only be assigned by the host node. However, the node may be simply used as a channel to gain access to its internal data. An authorized node may send a message instructing a particular action. For instance, if a node **A** has a *protected* policy set but node **B** requires access to **A**’s resources, the *signal* operators (>> and <<) may be used. Essentially, the goal is to be able to force the target node to make a call to

**Memory Structure**

**Scope and Permissions**

**Object Orientation and Inheritence**

**Data Types**

Several data types are available as part of the core language. These are called the *primitive* types. Built-in keywords are used to initialize objects of these types. However, the language technically supports any number of custom data types. External libraries may define a type and its functionality, while a user may also do so at runtime, although with limitations.

For instance; in regards to the former there are several standard libraries that add in additional data types. One of which is the *network* library. This defines the data type *port*. After this library is imported, this memory type is registered with the KNX engine and is available within the scope where it is defined.

Otherwise, a user may create an object of type *struct*, a primitive, in order to establish a new type. While both effectively do the same thing, there are a few differences.

**Operators and Logic**

Listed in the table below are the operators of the language. While most are common and should seem familiar, there are a few with uncommon meanings.

Arithmetic

|  |  |  |
| --- | --- | --- |
| Symbol | Descriptions | Use |
| + | Addition | [lVal] + [rVal] |
| - | Subtraction | [lVal] - [rVal] |
| \* | Multiplication | [lVal] \* [rVal] |
| / | Division | [lVal] / [rVal] |
| % | Remainder | [lVal] % [rVal] |
| √ | Root | [root order] √ [value] or  √ [value] (square root) |
| ^ | Power | [val] + [order] |

Assignement

Logic

|  |  |  |
| --- | --- | --- |
| & | AND |  |
| | | OR |  |
| ! | NOT |  |
| !& | NAND |  |
| !| | NOR |  |
| |! | XNOR |  |
| || | XOR |  |
| < | LESS THAN |  |
| > | GREATER THAN |  |
| <= | LESS THAN OR EQUAL |  |
| >= | GREATER THAN OR EQUAL |  |
| != | NOT EQUAL |  |
| == | IS EQUAL |  |

Misc.

|  |  |  |
| --- | --- | --- |
| Symbol | Description | Use |
| << |  |  |
| >> | Send Message | [expression] >> [node] |
| . | Member Access | [object].[member] |
| @ |  |  |
| $ | Literal Access | $[object] |
| ~ | Flag | ~[identifier] |
| ? | Generic Template | function(? var) |
| , | List | [object],[object] |
| # | Comment | #This is a comment |
| {} | Expression Body | {…} |
| () | Parameter Body/ Order of Operation Body | Function(3+(2-1)) |
| [] |  |  |
| : | Index | List:index |
| ; |  |  |
| \ | Escape | ‘\n’ |
| \ | Continue Next Line | *Expression…\*  *…expression* |
| ‘’ | Character | *‘c’* |
| “” | String | *“string”* |

**Keywords**

**Macros**

**Flags**

Flags are hints to the engine on how to process data. Certain flags are processed directly by the parser, while others may be interpreted by an individual function.

**Terminal Features**

**Optimizations**

**Thread Synchronicity and Safety**

**General Syntax**

**Program Design**

**Versioning**

**Platform Dependencies**

**Compilation from Source**

**KNX SDK**

**Building Custom Libraries**