KNX Runtime Engine

2015

Node/Object Orientated Programming Language and Interpreter

V 1.0.0 A

**Abstract**

This document describes the KNX runtime language. The defining features of this language primarily revolve around its node-based architecture. That is, its utilization of asynchronous threads which behave as individual active interpreters. This gives the language value in multi-processing applications as commonly found in scientific modeling and game design, although advanced graphical support will be a distant addition, if ever.

The following document will first cover the technical features of the language so that modifications to the original source code may be authored more quickly and effectively. Following this will be the language references, including data types, calling conventions, examples and all other information required to fully understand the language and its uses.

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| Section | Description | Page No. |
| System Requirements | Minimum system specifications | 2 |
| Definitions | Terminology definitions | 2 |
| Architectures | Technical data architecture | 2 |
| Node | Node details | 3 |
| Internode Communication | Data transfer between nodes | 4 |
| Command-line Arguments | CMD options |  |
| General Syntax | Basic syntax familiarity |  |
| Operators | Logical, Comparative and Assignment Operators |  |
| Std Libraries | Standard Packaged Libraries |  |
| Std Functions | Hardcoded functionality |  |
| Data Types | Standard memory types |  |
| Flags | Modifier flags |  |
| Scope | Memory scope resolution |  |
| Arithmetic | Using mathematics |  |
|  |  |  |

**System Requirements**

At the moment, the only operating system officially supported is Windows 7 for 64-bit machines. Although support for Linux distributions is planned, this support will likely not be added until development of the core engine and standard libraries is complete. At that time, the source code will modified to support Linux compilation via the GCC compiler. 32 bit support may be added some time after this for both Linux and Windows machines.

Although not necessarily required, multicore processors are strongly advised due to the nature of how this interpreter handles multiple points of simultaneous code execution.

**Definitions**

The following terminology will appear throughout this document. These definitions are intended to clarify the meaning of these terms ahead of time.

***Node orientated language***: In a similar vain to object orientated programming (OOP), Node orientated programming (NOP) utilizes *nodes*, or scoped thread wrappers which allow controlled multi-threaded execution of commands and scripts. These nodes are very much a part of the language, and not simply an effect of a multi-threading capability.

***Node***: The central concept of the KNX language is the node. A node is wrapper for a particular thread of execution. Each node contains a local and global memory space, and allows for the execution of commands. The master node of the system is defined as ‘node0’.

**Master Record**: The actual location of all nodes loaded in memory. The master record is responsible for generating a new node, connecting it to its parent and assigning it an id. It then returns a pointer to the node for use in the node’s command loop. This allows for all nodes in memory to be accessible in a constant runtime O(k) if required, as well as simplifying procedures required to maintain the system. The master record also manages resource streams, such as the *stdin*, or which node, if any, is capable of accepting direct user input. The root node *node0* is given this by default.

**Flag:** A character following the ‘~’ identifier that modifies an attribute or function of its associated token.

**Architecture**

While each node technically exists in a hierarchical tree, referencing children and parent by pointer, the actual memory allocation exists within an array called the *master record*. This ultimately maintains all threads and objects running, and allows for complex data management to occur.

**Node**

The node is the primary feature of the KNX language. In conventional languages, the system will execute along a single path of execution. Multiple threads may be defined later, but this can often be a tedious and unsafe task. For applications that wish to make strong use of multithreading, a node may be a good solution.

Each node behaves almost as if it were a stand-alone interpreter. Each one has a local default memory space for storing functions and variables. In addition, a reference to a global memory space shared by all nodes is available. Each node branches into child nodes to form a multiply linked node tree.

Nodes will remain active whilst they await further commands. If a node is certified by the *master record* to accept the stdin stream, and the node is currently set to accept said stream, the node will first process all existing commands in its stack, and once empty, will halt all command stack processing and will not resume until a user has entered data. The current node must release its hold of the resource in order to allow another node to gain access to it. By default, the right to the resource will return to the root node. However, the default priority may be changed to any currently existing node. If the current holder of any resource stream is deleted, all associated resource rights return to the root node.

**Inter-nodal Communications**

**Command Line Arguments**

Arguments passed from the command line are treated as ordinary commands. Because of the overhead associated with loading the interpreter, it is unwise to invoke simple, single commands via the command line. However, it may be beneficial for use in invoking scripts which will be loaded and run from the original command.

Some option flags may be passed, however. These must be passed first before any language commands may be executed. These option flags must present in the form of *–c [*option*] [option]…~*, and may be chained. Each term, including the terminating ‘~’, must be separated via a blank space. An example may be:

*-c dbg\_off echo\_off ~*

**Table of Options**

|  |  |
| --- | --- |
| Option | Description |
| dbg\_*on/off* | Switches debug printing on or off. Off by default in release, on in dev builds. |
| echo­­\_*on/off* | Switches printing off on or off by default for nodes. On by default. |
| wrn\_*on/off* | Switches warning printouts. |
| err\_*on/off* | Switches error printouts. |
| werr\_*on/off* | Switches error and warning printouts. Overrides wrn and err. |
| maxmem\_*num* | Sets maximum memory allocation to ‘num’ megabytes. |
| maxcores\_num | Sets maximum cores to allow to ‘num’ cores |

**Genral Syntax Familiarity**

The syntax of KNX is reminiscent of python and the c languages, although has several noticeable differences. For instance, variable declarations are not symbols defined proceeding a data type; instead, these keywords are replaced with built-in functions. The following example shows variable declarations in both C and Python for the creation of an integer *iVal*, followed by an equivalent declaration in KNX.

C: int iVal = 10;

Python: iVal = 10

KNX: int(“iVal”,10)

By default, this will create a standard 32-bit signed integer and store it in the most local memory space available. Modifications can be made to this via the use of *flags*, discussed later in this documentation. For instance, while a global variable in c must be declared outside a function body, the KNX example above can be saved to the global memory space by adding the ‘global flag’, as shown below:

int (“ival”, 10) ~g

There is a large table of such flags listed later on. However, care must be taken when using flags. Flags are processed by the passed-to function. A function may interpret a flag differently than another, or may even ignore it altogether. It is important then to read the supported flags and their functions of an unfamiliar function.