GNU KNX

* 1. 2016

KNX Runtime Language: Technical documentation and users’ guide.

The KNX runtime language is an object-orientated language built on a node-based architecture. The primary feature of this language is its use of multithreading to create ‘sub-interpreters’ for individual, yet connected points of execution. The target application base is primarily aimed at scientific and monitoring programs that require simultaneous computation.

This project is licensed under the GPL version 3.0. In addition, the relevant source code is designed to work with both the Windows and Linux line of operating systems on 64 bit and 32 bit platforms. Developed in C, the GCC compiler is the officially supported for building the project.

*Definitions*

Node: A thread wrapper of execution. Starting from the root node, or *node0*, nodes are threads that execute scripts or commands. These can spawn children that allow for multiple command sets to be executed in parallel. However, each node acts as its own ‘mini interpreter’, meaning that, by default, a node thread will remain active even after its workload is complete. This allows for nodes to be assigned duties to be executed as work becomes available. In addition, each node has its own memory space separate from the system as a whole, but still allows communication towards other nodes in the system. This idea allows for powerful yet organized programs to run in a very organic fashion. An instance of KNX with many nodes can be thought of as an office, with lots of workers cooperating and acting autonomously to ensure a smooth workflow.

Object: An object is defined in this document to be a general term for a variable instance. In the technical sense of the interpreter, an object is acts as a sort of super class for all memory types.

Library: An external binary library in the form of a Dynamic Link Library (DLL) that acts modules for the language. Several of these are present as standard libraries that ship with the interpreter. However, additional libraries may also be included. However, these non-standard dependencies must be present whenever a script calling for their import is launched.

Standard Libraries

All standard libraries may be imported via the *import* command.

|  |  |
| --- | --- |
| io | Basic and non-basic input/output functions |
| file | Disk file operations |
| cgraphic | Console or terminal graphics |
| wgraphic | Graphical interfaces |
| math | Non-trivial mathematical functionality |
| network | Networking functionality (useful for *port* objects) |
| pipe | Inter-processes functionality (useful for *pipe* objects) |
| util | General utility operations |
| string | Advanced string operations |
| collection | Collections and containers |

Non-standard libraries are called similarly to the standard libraries. For instance, importing the standard library *io* is simply calling the import command as follows:

import(“io”)

Calling a hypothetical library *mylib* is simply:

import(“mylib”)

Library lists are first scanned from the standard library, and then from the external library directories. In order to slightly speed up the scanning processes, or if a non-standard library for some reason assumes a name already occupied by a standard library, the *import* command accepts the *~s* and *~e* option flags. These signal the import command to start at the standard directory (default) and external path, respectively.

Built-in functions

display : Print input to screen

Arguments: Miscellaneous

Return: void

Flags: none

getln : Get input from console

Arguments: none

Return: string

Flags: none

terminate : Exit current node or end targeted node

Arguments: none (terminates current node)

Child node (terminates target)

Return: bool (true if successful)

Flags: ~c

xnode : Create new child node

Arguments: string (command to send to next node)

Return: Node handle

Flags: none

object : Create new variable object (do not call object, call name of type)

Arguments: (depends on type: refer to data type sheet)

Return: none

Flags: ~l ~g

import : Import specified library

Arguments: string (library)

Return: bool (true if successful)

Flags: ~s ~e ~f (~f overwrites existing instance, if any)

delete : Deletes specified object

Arguments: string (variable name)

Return: none

Flags: ~l ~g ~f

Flags

Flags modify the operation of the affected method or declaration, or add attributes to particular argument passed to a function. Certain flags will be processed during the tokenization process or during certain processing steps by the engine. Those that are assigned as attributes will be processed independently by the affected method in question. These attribute flags do not necessarily hold a constant effect, as it is up to the particular implementation to decide how to handle certain flags, if at all, and so it is important to check which flags are reserved for the interpreter and which will be handled by an operation itself, and how it interprets those attributes. Below are the standard definitions of flags used by the interpreter.

Flags are declared by the convention ~*f*, where *f* is a single letter immediately following the flag operator ‘~’. A declared flag will affect its immediately left-hand group. Make note that flags are case-sensitive.

|  |  |  |
| --- | --- | --- |
| Flag | Letter | Description |
| Force | f | Forces an operation, regardless of warning or error |
| Local | l | Search locally first (default) |
| Global | g | Skip local search and start at the global scope |
| External | e | Search an external database |
| Standard | s | Search a standard or internal database |
| Debug | d | Apply debug printouts, even if the system is set to ignore debug printouts |
| Precompile | p | Compile operation as bytecode before execution to speed up future calls |
| Confirm | c | Hang operation until fully resolved |

Data Types

|  |  |  |
| --- | --- | --- |
| Type | constructor | Description |
| integer | int(“name”)  int (“name”, value) | Signed integer |
| short integer | short(“name”)  short (“name”, value) | Signed short integer |
| long integer | long(“name”)  long (“name”, value) | Long long signed integer |
| double | float(“name”)  float(“name”, value.x) | Double precision float |
| character | char(“name”, value) | Signed character |
| string | string(“name”)  string(“name”, “value”) | Character string |
| structure | struct(“name”) | Aggregate collection |
| array | array(“name”, type) | Single type array |