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 |

Parsing Methodology

For developers of the language, knowing how the interpreter analyzes and executes commands given is important when making modifications to the engine itself. There are several key steps in the code preparation and execution phase.

* Tokenization (Direct and hierarchal)
* Block buffering
* Lexical analysis
* Data and reference retrieval
* Error handling
* Process modification handling
* Command Execution
* Reconstitution

The first stage in the code interpretation cycle is tokenization. Tokenization is the process of converting plain text into a sequence of ‘tokens’, or recognizable chunks of data. For most cases, this is simply applying metadata to a small chunk of text to indicate what its value or function is. During tokenization, the interpreter may switch into one of several modes. These modes are:

*String Mode*: When an un-escaped ‘”’ character is detected, the interpreter begins ignoring all non-‘”’ characters, with the exception of the escape character ‘\’ itself. The escape character changes function from being an ‘ignore’ operator to special character identifier, in some cases. For example, in normal operation, \n will cause the scanner to skip over the following character. In this case the letter ‘n’. However, with *string mode* activated, the escaping of this character replaces it with the *newline* character. Escaping characters without special escape values, such as ‘.’, ‘”’, or ‘a’, for example, will simply cause them to skip as normal. In this way, it is possible to add special operators to an input without activating their effects.

*Pending Mode*: Pending mode is how block groups and lists are handled. When the operators ‘(‘, ‘\’, ‘{‘are used, pending mode is activated. All following code, be it on the same or an incoming new line of text, will not be immediately executed. Instead, they will simply begin the construction of a branch within the token tree. The token tree will be explained later, and is very important in understanding how signals are carried out in nested code blocks. Once the lowest existing set of these two grouping operators is closed, the code block will begin execution. Make note however that the escape character ‘\’, while activating pending mode, will not count towards any counter. This will not escape empty characters, and only serves to signal that the following line should be counted as a continuation of the current line, assuming not non-whitespace character immediately proceeds it.

*Comment Mode*: Activated with an un-escaped ‘#’ character, bounded until a newline character or a proceeding ‘#’ character, this mode simply ignores input completely.

Token Tree

The token tree is a sort of ternary tree used to control the execution path of block statements. Each token acts a node with a left and right child, and access to its parent. This is particularly useful for speeding up the execution of loop statements.

Each node in a tree (not to be confused with the previous definition of a node) has a left, right and center branch. The left branch consists of a *list*, or a sequence of nodes that are evaluated before being passed into the parent node as arguments. Then, assuming the parent node was successful in its operation activates its center child node, assuming it exists. If the operation fails then the right node is triggered. The right node allows for statements such as ‘else’ and ‘catch’ to occur. If the target trigger node does not exist, then the node branch ‘collapses’, or begins transferring control back up its parent path until a waiting parent node is found. This parent node will continue its standard operation once it has received control again.

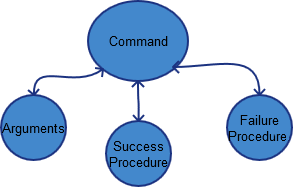


Figure 1 : Basic node Diagram

Arguments are executed in a sequential fashion, and may spawn their own branches. For a better visualization of how the node tree works, take the following script and its associated tree:

***bool(“cnd”, true)***

***while($cnd){***

***int(“val”, rand(0, 10))#val created in function scope***

***display($val+’\n’)***

***if ($val==0)***

***cnd=false***

***}***

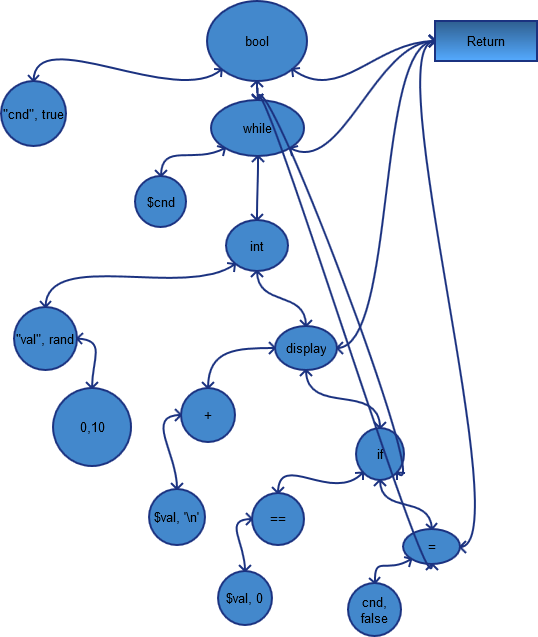
******

Figure 2 : Node tree of script