KNX Runtime Engine

2015

Node/Object Orientated Programming Language and Interpreter

V 1.0.0 A

Definitions

1. Node Orientated Programming (NOP): The use of thread wrappers with hierarchy trees and dependencies with local memory spaces.
2. Object Orientated Programming (OOP): The use of data wrappers to simplify data management by use of encapsulating symbols.

Language description:

The primary objective of the KNX runtime language is to allow for cross-platform execution of modular libraries, and to make full use of multi-core processors. The target applications of this language run from server applications that handle variable threads designed to coordinate activity among multiple sources of input. In addition, with regards to common applications in such artificial intelligence (AI) or simultaneous sub-procedures are vital to operation; the architecture of this language shall be designed with the intent of allowing systematically distributed resources of memory and processing affinity. This language is to be developed in the C++ language for all targeted platforms.

Architecture and design concepts:

Use of OOP: Objects, while not a major focus of the language, do play a useful role in data management. Similarly to languages such as C, C++, and python, KNX employs a *struct* variable class in which all manner of data may be collected, including other declarations of type *struct*, functions, but not of type *node*. The accessing of member objects is performed by the classical indirection operator ‘.’. Internally, variables are initialized via a wrapper which includes a code denoting the variable type, its value, a hash representation of its name, and any functionality that is specific to its type. These declarations are stores within *struct* types via a vector array of the generic base class.

Use of libraries: Libraries come in two varieties; *system libraries*, or DLL’s, contain and expand functionality that is to be used by the interpreter. Alternatively *code libraries*, or pre-written *KNX* files, are simply included files in which declarations and procedures may be imported. *System libraries* are loaded during runtime. The standard libraries, such as StdIO.dll and StdMath.dll, come packaged with the compiler and are loaded immediately. However, custom libraries may also be installed by placing the respective .DLL and .DEF files within the *libs* directory. These libraries can be manually imported by the *import* command. By default, a loaded library with be saved to the global memory space, becoming available for every node. However, use of the local flag (*~l*) can be used to constrain the library input to only a selected node. Code libraries are loaded as any regular *.knx* file.

Use of logic operators: Logical operators are divided into two categories: *comparative* and *logical* operators. Comparative operators determine the relationship between left and right hand operands, and then return a Boolean result. Slightly differing, logical operators compare the Boolean values of operands. Being such, comparative operators are first evaluated, followed by the logical operators, which evaluate multiple Boolean results into a single true or false statement. Below is a complete list of acceptable logic gates:

*Logic Operators*

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Operator | Name | Description |
| Comparative | > | Greater Than | True when left is greater than right |
| Comparative | < | Less Than | True when left is less than right |
| Comparative | == | Equal To | True when left and right are equal |
| Comparative | >= | Greater Than or Equal To | True when left is greater or equal to right |
| Comparative | <= | Less Than or Equal TO | True when left is less than or equal to right |
| Comparative | != | Not Equal To | True when left and right are not equal |
| Logical | & | And | True when left and right sides are true |
| Logical | | | Or | True when if either side is true |
| Logical | ! | Not | Inverts right value. Does not evaluate left |
| Logical | !& | Nand | True when at least one value is false |
| Logical | !| | Nor | True when left and right are false |
| Logical | || | Xor | True when only one side is true |
| Logical | |! | XNor | True when left and right sides are similar |

Comparative operators are slightly flexible in operation. All numeric values can be evaluated by these operators. When evaluating a character to a numeric value, the character is assumed to be of an integer representing its ASCII designation. Strings may only be evaluated by the ‘==’ and ‘!=’ operators, which will compare their contents. All other operations will fail. These operations are possible between a character and a string, but will always fail if the string is of a length longer than 1. As logical operations compare only Boolean values, and assume all non-zero values to be true, logical operators are unrestricted, although careful use of operands is advised.

Use of arithmetic operators: Arithmetic operators are primarily focused on numeric operations, but also hold some use for character and string type operations. These operations are carried out from left to right, and do not head any rules of operations. As of the initial build, operations cannot be nested, and all operations must be explicit. Below is a table of recognized operators and their respective functions.

*Arithmetic Operators*

|  |  |  |
| --- | --- | --- |
| Type compatibility | Operator | Function |
| Numeric, String | + | Sum two numbers; Concatenate two strings |
| Numeric | - | Return difference of two numbers |
| Numeric | \* | Multiply two numbers |
| Numeric | / | Divide two numbers |
| Numeric | √ | Root number by power of left operand, default of 2 |
| Numeric | ^ | Raise left operand by power of right |
| Numeric | % | Return the remainder of the division between operands |

As seen above, the ‘+’ operator can be overloaded to concatenate two strings, or to append a character to a string. Adding a number and a string will result in the numeric value being cast to a character and appended. Automatically casted numbers will wrap in value. Likewise, a character being added to a number will first convert the character to an integral value; two characters being summed will result in a character with the integral value of the two sums.

Use of nodes: Nodes are central to the internal architecture of KNX. By the definition described at the header of this document, a node is a wrapper that encapsulates a memory space, input/output operations, and code execution. To further explain, each node is bound by a parent/child tree that allows for better management of sub-processes and task handling. All nodes stem from a master node, or the level 0 node, which holds priority over all other nodes below it. This node runs from the moment the interpreter is executed. Once a node is running, the global memory space is passed down from the parent node. In this respect, all nodes of all lineages are able to share data by saving and pulling from the global memory space. Local memory spaces are, by contrast, bound to a particular node. This data is invisible to all other nodes, including its parent node by default. However, this does not mean that local memory can never be accessed. Please read the following section on memory protection levels. Just note that nodes can request information from each other, which may be granted or denied based on the securities set on the memory in question. Another core feature of a node is the *command stack*. A command stack is a set of instructions that hold priority over direct user input. When a node loads a script, or a command is sent from one node to another, these commands are saved to a stack. These commands will be pulled said stack and executed until the stack is empty. At which point, input is returned to the user, assuming input has not be disabled. Nodes are defined by name, and can be operated on as if they were structures or classes, by invoking the member indirection operator. Nodes are always global.

Use of memory permissions: There are four tiers of memory permission. Each tier grants a freer or more restricted level of access, requiring special permissions or criteria to be met. Memory restrictions are ignored within the global memory space, and so only hold meaning within local declarations. Below is a table of memory permission tiers.

*Memory permissions*

|  |  |
| --- | --- |
| Public | Openly accessible to all requests |
| Private | Only accessible to friend nodes |
| Heritage | Accessible to all parental/child and friend nodes (Excludes parental branches) |
| Secure | Never accessible (Even to friends) |

Restrictions are placed on a declaration either at time of initialization, or by the *setPermission* function. In either case, a flag is used to set the permission state.

Use of flags: A flag is a character or number following the ‘~’ token. There are no spaces between this token and the flag identifier. Flags are case-sensitive. Only the first character following the flag token will be identified as a flag. Flags are used to modify the function of a declaration or procedure by placing it within the parameter slot desired. Flags are not always applicable depending on the usage, and can very between functions. Below is a table of flags, followed by some simple example code detailing the syntax of flag use.

*Flags*

|  |  |  |
| --- | --- | --- |
| Flag | Meaning | Common interpretation |
| ~l | Local | Ignore global, Prefer local (default) |
| ~g | Global | Skip local |
| ~P | Public (uppercase) | Use public permission |
| ~p | Private (lowercase) | Use private permission |
| ~h | Heritage | Use heritage permission |
| ~s | Secure | Use secure permission |
| ~f | Force | Continue through errors |
| ~e | Error | Suppress errors |
| ~w | Warning | Suppress warnings |
| ~d | Debug | Show debug information |
| ~c | Echo | Suppress all output |
| ~a | Abort | Abort operation at error or warning |
| ~n | Newline | Append newline to output (default) |
| ~C | Conserve (uppercase) | Automatically keep arrays at smallest volume |

*Example code*

#Flag1.knx

#Using flags to modify code

int i1, 3, ~p ~l#initialize integer as local protected

int i2, 5, ~l#initialize integeras local public

int i1, 7, ~g#initialize integer as global public with same name as first object

display i1, ‘ ‘, i2#display i1 local and i2 local

display i1 ~g, ‘ ‘, i2#display i1 global and i2 local

delete i1#delete local i1

display i1, ‘ ‘, i2#display global i1 without global flag

//------------------output-----------------//

3 5

7 5

7 5

As show above, memory space is, by default, searched at the local level, and then proceeds to the global level. Therefore, there may be a local and global declaration in existence; in which case, the local is retrieved first, unless it does not exist or if ~g hints to use the global variation in its place. Note, if the global flag is used and the variable does not exist, the local memory space will NOT be searched. And error will be returned instead.

Use of general operators: Many operators allow for data manipulation, such as for arrays and lists. These purposes vary, and so hold little common ground. Below is a table of the remaining operators and their purposes.

*General operators*

|  |  |  |
| --- | --- | --- |
| Name | Operator | Description |
| Bracket | [] |  |
| Encapsulation | () | Interprets highest to lowest level |
| Brace | {} | Ignored, only for readability |
| Index | : | Right integral operand determines index of left hand array |
| List | , | Separates input fields |
| Semicolon | ; | End field inputs |
| Comment | # | Ignores all characters until another ‘#’, or until a newline |
| Reference | @ | Returns memory address (not a pointer) of right operand |
| Character/String | ‘ | Identifies a character, or a string if more than one letter |
| String | “ | Identifies a string |
| Escape | \ | Determines escape value of proceeding character |
| Flag | ~ | Identifies following character as flag token |
| Value | $ | Identifies following as variable |

General Syntax: KNX features a particularly simplistic syntax. Commands and declarations need not be bound within function bodies or classes, and all code is executed in the order in which it is received. All code is newline delimited, with whitespace playing little role in interpretation. Multiline declarations are bound between the calling function or declaration, and terminated with an *end* statement. The *end* statement will always close the last valid multiline statement, such as often employed by sub-procedures and logic blocks. Braces may be used to allow for more visually appealing code, but are ignored unless preceded by the escape character. Important to note, function calls are not required to end with a semicolon, unless nested as a parameter. In such a case, parenthesis can also be used, as per preference. Below are a few short samples outlining general syntax.

#SyntaxDemo.knx

display “1”+4+” is ”+14

display “hello ”

string str, “how are you today?”

display “world”~n, $str #use of flags within input field

int i, toInt “14”;#use of semicolon to terminate nested function. Same as: int i, (toInt 14)

//------------------output-----------------//

14 is 14

hello world

how are you today?

Data types: As stated earlier, all type declarations are stored within wrappers that contain name hash values and some metadata. Each type of wrapper initializes its own unique data type. Below is a table of all useable data types.

*Data types*

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Calling Function | Description | Declaration |
| Signed Integer | int | 32-bit signed integer | Name, value |
| Unsigned Integer | uint | 32-bit unsigned integer | Name, value |
| Float | float | 32-bit float | Name, value |
| Double | double | 64-bit float | Name, value |
| Bool | bool | 8-bit Boolean | Name, value |
| Char | char | 16-bit character | Name, value |
| string | string | Character array | Name, value |
| Array | array | Generic array | Name, type |
| Usb | Usb | Usb port handler | Name, Usb port |
| Port | port | Network port handler | Name, port number |
| Graphic | graphic | Image handler | Name: Name, file |
| Struct | struct | Generic object storage | Name |
| File | file | File handler | Name: Name, file |
| Node | node | Child node | Name |
| Function | function | Instruction set | Name, return type |

To further explain a few more complex data types, that is: Array, Usb, Port, Graphic, Struct, File and Node, the following descriptions elaborate on the use and function of the aforementioned data types.

*Array*: Declared with a name and data type, an array takes on the form of the declared variable and allows for easy manipulation of its contents. Member functions include *.add [object]* and *.remove [index]*. These functions allow for elements to be added or deleted, respectively. If an object at a particular index which is deleted was added via reference, the referenced object does not get deleted; only the reference within the array is removed. Objects may be added via the *.add* member function, or by the ‘+’ operator. Non-similar additions will be auto-casted to the declared type. Incompatible datatypes will not be added, and an error will be thrown.

*Usb*: A Usb handle watches a specified port and allows data streams to be read or written.

*Port*: Network ports are similar in function to the *Usb* type, with the difference simply being the type of stream being read. Port types may also communicate via Url, and come with a multitude of configurations that range from communication protocol to timeout periods, etc.

*Graphic*: Graphic images are stored in a generic RGB which can be created blank and modified, or loaded from file.

*Struct*: As a generic storage container, a *struct* object is capable of storing a variable number of any type, including other *structs*, with the exception of nodes. Members from within this wrapper may be accessed via the indirection operator. When invoking functions from a *struct*, the declaration is the same as if the function were called normally. However, the struct behaves as an additional low-level namespace. This means that a function called from a struct will first search the wrapper for relevant declarations. Calling a function with the ~l, or local, flag will skip this additional search. *Structs* contained within the calling object will also be searched in the order that they are stored.

*File*: A file is little more than a complex string, with the addition of storing type extensions and pathways as well as the ability to read and write to disk.

*Node*: All functionality occurs within nodes. Nodes are the basic execution mediums for code and input, as well as acting as data handlers. When a node is initialized, it automatically becomes a child of the calling node. Each node must have a unique name from any other registered within the system. It is also possible to limit the number of nodes active at any one time. A node cannot be stored within a wrapper, and are always global. Requesting access from a node is performed as any other object. When a node is terminated, all child nodes beneath is are also terminated. Only a parent node has authority to send commands to a child node, unless said node has granted access to another. Each node created generates a new memory space, but also shares the global memory space. All nodes are parent to the primary node, node0. Node0 is created on startup and will execute all initial code. Node0 has authority over all sub-nodes, but is not able to access protected memory of its children. A parent node can, however, force a child node to grant it access. If a node is busy, commands sent to it may not be processed. A call to *killChild* with the ~f flag will force the node to terminate, regardless of consequences.

Standard libraries: Be default, without having installed further libraries, all basic functionality is provided by external DLL’s, as stated earlier. Each library comes with a dynamic link library and a definition file, both of which are required to implement functionality. As libraries are loaded on runtime, new libraries may be installed without restarting the system. However, libraries must be unloaded and re-loaded in order to update existing versions. Care must be taken to avoid name collisions, as the system will be unable to import a library if one or more function or variable names already exist within the global scope. Below is a table of all current standard libraries.

*Standard libraries*

|  |  |
| --- | --- |
| Library | Description |
| IO | Basic input/output functions |
| Math | Mathematical functions, non-parse-time calculator |
| Graphic | Graphic processing |
| Network | Network functionality |
| Util | General utility functions |
| String | General string operations |
| Security | Encryption/Cipher methods |
| File | General file operations |
| Library | Handles library operations |
| Memory | Memory management |
| Debug | Added debugging functionality |
| Data | General data type processing, conversions (such as explicit casting) |
|  |  |
|  |  |
|  |  |

*Standard Functions*

|  |  |  |  |
| --- | --- | --- | --- |
| Library | Function | Parameters | Description |
| IO | display | … | Output value to console |
| IO | getln |  | Retrieve input from console |
| IO | prompt | [string] | Output string, retrieve input from console |
| Math | sin | [numeric] | Return value for sine at input in radians |
| Math | cos | [numeric] | Return value for cosine at input in radians |
| Math | tan | [numeric] | Return value for tangent at input in radians |
| Math | asin | [numeric] | Return value for arcsine at input in |
| Math | acos | [numeric] | Return value for arccosine at input |
| Math | atan | [numeric] | Return value for arctangent at input |
| Math | calc | [string] | Return result of string input |
| Graphic | loadImg | [string, Graphic] | Load image from file |
| Graphic | saveImg | [string, Graphic] | Save image from file |
| Network | connectPort | [integer, Port] | Connect Port type to designated network port |
| Util |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Intended uses: Any language is tailored to meet specific needs. Lower level languages aim at general and efficient design, with the cost of complexity and development time. Higher level languages sacrifice scope and efficiency for quicker development. KNX aims to meet the most common needs of today’s software, while still being quick and efficient. As the engine is almost completely modular, any required functionality may be implemented by the user, or taken from a third party source. However, with this in mind, the KNX language is primarily aimed at scientific modelling, game development and artificial intelligence. For specific developments, higher efficiency libraries may be built that aim at narrowing the capabilities of the engine, whilst improving efficiency of the task at hand.

Future development: Projected goals for this engine are primarily focused on expanding functionality and increasing efficiency. More distant goals include script-compilation to binary executables, and real-time compilation of stored memory to avoid the need for parsing in the most used functions.