NeurOn Neural Network Design Language and Compiler

1.0.0 Alpha



User Manual

Contents

* About neurOn
* Command Line Arguments
* Compilation Chain
* Neuron and Connectome files
* CNS files
* Operators and Keywords
* Variables
* Functions
* Inline API
* Runtime Interfacing
* Makefile compilation
* Structure of a connectome file
* Encoding (internal and connectome)
* mRNA Ciphers

About neurOn

Biological neural systems are extraordinarily complex. In a SAPA system, the intention is to create a sort of hybrid logical system that behaves similarly to a biological model but built to function on modern hardware. The intent is to create programs capable of complex learning.

With rapid development in mind, it can be extremely tedious and time consuming to constantly write, re-write and edit lower level code. Thus, neurOn offers a higher level way of designing systems without worry about the underlying overhead.

The principle is similar to a hardware design language such as VHDL or Verilog. Circuits may be designed via written expressions and compiled into a final product. However, there are several major differences.

First of all, SAPA systems are dynamic. The program written and compiled will only yield an optimized program and a starting point. From there, a system is free to evolve and learn as time progresses. Secondly, the systems intended to be created are significantly more complex than an average electrical circuit. Thus, expressions are designed with extreme complexity in mind and make heavy use of loops and regional generations.

The intent of this part of the SAPA toolset makes the development of learning machines quick and (relatively) simple. The remainder of this manual contains definitions, explanations and examples to begin creating unique neural systems.

Command Line Arguments

The table below lists all accepted command line arguments. Note that options must be set in the form of -[n] or --[setting]=*value* with no spaces in an expression.

|  |  |
| --- | --- |
| CMD Option | Description |
| -d | Enable Debugging |
| -b | Enable Build (Automatic if no makefile specified) |
| -m | Searches source directory for makefile (implicitly enables build) |
| -c | Enables connectome state file compilation |
| -r | Enables runtime engine compilation |
| -v | Prints version |
| -s | Prints symbolic signature |
| -h | Prints help information |
| -i | Saves a copy of the bytecode file |

Compilation Chain

The compilation process in neurOn is different from many other languages in the regard that it actually produces two conjugate outputs. The first is a runtime engine, and the other a ‘connectome file’, or simulation file.

The runtime engine is an executable binary with pre-built types and functionality. On startup, a built runtime engine acting a simulation environment loads a set bytecode file called a connectome, or \*.ctm, file. From here, all saved-state information from when the simulation was last interrupted can be resumed. In addition, this allows an optimized environment to load compatible network designs.

At the first stage is the pre-compiler. The pre-compilation phase is responsible for resolving pre-compilation directives, such as macro replacements and importing additional source files. In addition, the pre-compiler will combine all imported files into a single large source code file, as well as simplify it by removing unnecessary whitespace, replacing macros and performing certain logical statements.

Pre-compilation statements may be used in both source files and the project file. Implemented pre-compiler statements as of version 1.0.0 are:

* import
* pseudo
* define
* if
* else
* elseif
* end
* isdef
* equ
* nequ
* lss
* gtr
* and
* or
* not
* inline
* symb
* depend
* \_OS
* \_BIT
* \_VERSION

Immediately following per-compilation begins the lexical analysis. At this stage, a stream of tokens containing meta data for each identifiable term is generated. These tokens identify terms by keywords, primitive type (such as string, char, int and float...) and special operators such as logical and arithmetic expressions.

This token stream is then passed to the semantic analysis. Here, the token stream is scanned and a dependency profile is generated, detailing existing objects in scope-form. This is the final stage in the code analysis. The result is a bytecode file capable of being converted into the desired language, or generating neural design files.

Neuron and Connectome Files

Connectome files project files that detail information on a particular build in a single location. But what are neurOn and connectome files? While they both contain information pertaining to the design of a system, they are very different.

A neurOn files is a human-readable source code file. The language outlined in this manual explains its syntax and use. However, it would be very resource consuming for the simulation enviornment to attempt to execute this code continuously. Therefore, the connectome file is generated from source neurOn files.

The connectome file is a sort of save-state file. It doesn not detail WHAT a system is comprised of, only HOW it is comprised. For instance, the connectome file has no idea what methods a particular object may have. What is does know, however, is what those objects' values were at the end of the last simulation, and how the paths between structures looked like.

A connectome file is used to not only begin a simulation, but also changes as the simulation changes, so that it may be resumed at a later date. However, they can become enourmously complex as they explicitly record every single object and data instance, and is almost entirely composed of hexadecimal values.

CNS files

A CNS file is the project file used by the compiler to generate connectomes and runtime engines. Here, compilation settings can be set, and the required neurOn source files are identified.

There are several system settings that may be set from this file. The CNS file is interpreted separately from the \*.nrn file, and so does not follow the neurOn syntax.

Below is an example CNS files:

**#test CNS file**

**#SAPA project file**

**# = SET**

**# ^ IMPORT**

**^ src**

**#compilation settings**

**= evolution false**

**= optimize false**

**= debug true**

**= title dev**

**= version 1.0.0**

**= language c**

**= makeRTE false**

**= makeCTM false**

**= prsvSymb false**

**#sim settings**

**= RSP -70#resting potential**

**= TDF 1.0#time dilation factor**

**= NGEN true#neuro genesis**

**= NMAX 500**

**= NTRAN 2**

*example CNS file*

Noteably, there are several settings to be modified, each following the '=' operator. The first argument defined is the setting name (case sensitive). The next is the assignment value.

Here's a quick explanation on each of the current CNS settings.

Evolution: An experimental option for self modifying binaries. This is not currently implemented, and may be removed/re-thought. A future implementation may be for the simulation to pause occasionally and self modify its neural pathways via genetic algorithms.

Optimize: Runs additional scans to make a more complex, but more efficient compiled system.

Debug: If enabled, the compiled simulation environment will print diagnostic information.

Title: Will be compiled into retrievable project information, and will be set as the heading of a terminal when executed.

Version: Will be compiled into retrievable project information, and will be appended to title in header.

Language: This defines the language that the simulation enviornment source code is compiled to. The currently supported languages are C and Java.

MakeRTE: When this is true, the compiler will generate a simulation environment source file.

MakeCTE: When this is true, the compiler will generate a connectome file.

PrsvSymb: This option is not currently implemented

RSP: This is simulation resting potential.

TDF: The time dilation factor. This controls the rate of time as perceived by the simulation. Adjusting this may result in unstable performance.

NGEN: When true will allow for the simulation to prune unneeded neurons, and also spawn new cells.

NMAX: The maximum of active cells at one time.

NTRAN: Number of bytes allocated for neurotransmitter models. By default, this is one byte, allowing for 8 neurotransmitters to be defined. This number is limited to 4 on a 32 bit system, and 8 on a 64 bit system. This number must be at least 1.

Operators and Keywords

|  |  |
| --- | --- |
| Operator | Description |
| # | Comment line, or until next ‘#’ on same line |
| #\* / \*# | Multiline comment |
| ! & | !& !| || !| | Logical operators: not, and, or, nand, nor, xor, xnor |
| + - / % \* √ ^ | Addition, Subtract, Divide, modulo, root, power |
| ? | Precompilation command |
| <> | Outside of |
| >< | Inside of |
| = | Set |
| == < <= > >= != | Equal to, less than, less or equal, greater than, greater or equal, not equal |
| : | Index / Bitwise selector |
| += -= \*= /= | Add / Subtract / Multiply / Divide then assign |
| =| =^ =& =~ | Bitwise or/xor/and/not and assign |
| . | Member reference |
| >> << | Left/ Right bitwise shift |
| ~ | Ternary evaluator |
| (type) | Type Cast |
| ; | End of statement |

|  |  |
| --- | --- |
| Keyword | Description |
| new | Creates a new Sapa object (cell or group). Implicit |
| delete | Deletes a Sapa object |
| public | Allows external access to members |
| while | Executes next code block or statement until a condition becomes false |
| break | Exits a loop |
| switch | Executes a code block with a ‘case’ value that matches the given argument |
| case | Declares a switch case value |
| dowhile | Executes next statement once, and then continues normally |
| continue | Skips remaining body of loop and restarts |
| if | Executes next statement if condition evaluates to true |
| else | Executes if previous statement is false |
| print | Prints to console |
| log | Appends a line to an internal log buffer to dump periodically and upon exit |
| shutdown | Begins shutdown and state save of system |
| default | Default case statement |

Variables

Variables in a Sapa project tend to be very limited, with the few directly available data types designed to optimize processing, and minimize memory usage. Certain data types may be different from one environment to the next, due to project configurations or bit modes.

|  |  |
| --- | --- |
| int | A 32-bit signed integer |
| uint | A 32-bit unsigned integer |
| float | A 32-bit floating point decimal |
| state | A one bit boolean value. These will consume one byte per group of eight declared in an object or scope. |
| char | An unsigned one-byte integer used to hold an ASCII text character. |
| string | A dynamically allocated string of characters. |
| ulong | An unsigned integer. 64 or 32 bit, depending on bit-mode of project. |
| cell | The basic logical node. Contains core functions and data sets required for the designed a system. The default cell is completely overrideable, but does come with default implementations. |
| group | The basic processing module. This container provides a standard interface for designing networks, grouping networks together, and controlling their local simulation environments and regulating cell procedures. |
| signal | A bitwise container for storing inter-neural signals during action or graded potential. This gives context to an input, allows signal propagation, and additional situation-dependent reactions to certain stimuli. By default, the signal type is a single byte, allowing for 8 unique messages to be sent in one pulse. The limit is 4 bytes for a 32-bit system, and 8 for a 64-bit system. |
| generic | A typeless object that may refer to any object at any time, or nothing at all. A generic type is generally unsafe to use, but may provide additional flexibility if used appropriately. |

Variables may be defined within an object or function, or globally outside of a containing body. Variables are resolved from the most local scope down to global, so a global variable may be ignored if another exists closer to the code reference.

Modifiers may be used to change the visibility or behavior of a variable, as show in the table below.

|  |  |
| --- | --- |
| const | A variable that cannot be modified after declaration. Unlike in other languages, a constant variable in neurOn also may behave as a static member. |
| static | This variable will persist after the scope in which it is declared exits. |
| persistent | This may only be applied cell objects. In the event that a cell no longer has outgoing connections, it will not be pruned, and instead will continue to exist so as to be able to continue to make and accept connections. This is useful for ensuring portal nodes are not destroyed. |

Macros

Certain macros are pre-registered with the compiler. These include:

* True : 1
* False : 0
* SgSz : Signal Size (Defined in project file)
* OS : LINUX or WINDOWS
* BM : Bitmode (32 or 64)
* LANG : Language (Only available through simulation environment)
* VERSION : Project defined version
* ENCODEVRS : mRNA encoder version
* SIGNATURE : Data structure signature
* CMPVERS : Compiler version

Functions

There are a small handful of built-in functions available. These keywords consist of declarations, IO and debugging methods, as well as common operations such as loops.

Certain modifiers may be applied to functions to change their behavior. Modifiers are declared before the function identifier, before or after the type declaration.

|  |  |
| --- | --- |
| virtual | A virtual function may be overridden at some point in time by another object. Virtual methods may only be defined within an object, and overridden by an inheriting object. This is a characteristic of object polymorphism. If a function prototype is marked as virtual, it behaves as an interface. The object in which it is declared may not utilize it, and inheriting objects must override the method in order to use it. |
| override | A function modified with override must be in reference to a virtual method. A method may only be overridden if it is a virtual function in an inherited object. The type and parameters must stay the same. |
| deprecate | When marked deprecate, the function will still be compiled, but a compiler warning will displayed alerting the user that the function should be avoided. |
| async | This function will be launched in an asynchronous thread. |
| sync | This function will be launched in a synchronous thread. |
| immutable | Disables further overrides in inheriting objects. |

Inline API

In some cases it may be useful to directly program in your language of choice, as opposed to finding native solutions which may not provide the full range of control needed in a particular situation. As such, the inline API allows for code to be written directly without compilation.

To inline a code segment, you must use the *?inline* pre-processor operator followed by the *inline* keyword. The first term following this is the dependency pattern. This tells the pre-compiler how to properly format the import statement in the language. A second *?* character followed by the identifier *symb* will denote the place to insert the libraries. Each following term, separated by space, will be added to the library import list. Redundant statements will be ignored. Nothing needs to be placed if the appropriate library is placed elsewhere

**cell exampleCell [**

**override void update()**

**{**

**?inline “C” #include<?symb> stdio.h**

**printf(“This is an inline C statement\n”);**

**printf(“Running on %dbit %s\n”, \_BIT, \_OS);**

**?inline “Java”**

**System.out.println(“This is an inline Java statement”);**

**?end**

**}**

**]**

**Or**

**?if \_OS equ “C”**

**?depend #include <?symb> stdio.h math.h stdlib.h**

**?end**

**…**

Something to take care to note: inline statements will be dropped directly where they are declared. Therefore it is important to make sure you know what the surrounding neurOn code will compile to in order to prevent stray code.

Runtime Interfacing

When a Sapa system launches, a terminal is produced. A new thread for handling user input is spawned, and reserves a spot on the terminal that allows a user to enter commands without sprawling text from interrupting input.

Through this interface, a basic set of commands may be sent. These allow a user to view information about the system, pause operation, shutdown and save a system, and a variety of other functionality.

Makefile Compilation

Makefiles are a loosely supported feature. Template makefiles are supplied for supported languages in the **support/<language>/** directory. These templates may be modified if desired, but otherwise the compiler will fill in empty fields automatically when binary compilation is specified.

However, template makefiles will only be used if an existing makefile does not exist. If a makefile already exists inside the target project directory, this will be used instead. The compiler will not call a compiler directly, and thus requires that the appropriate compilers are installed on the working system, as well as a *make* program.

By default, GNU compilers will be used.

Structure of a Connectome File

The header of the connectome file details metadata about the save state, including a timestamp, data structure signature, and soft-coded configuration options. In addition, details on the state information are described in terms of container density. From each base container up, the number of unique object types (both cells and groups) are given. Each container up from that is the same. In this way, the system can described and populated at once, so as to remove the need for dynamic memory allocation on startup. From there, the data types need only to be initialized.

In the body of the file, all initialized sets for each container and object are described in the order in which they were allocated.

Encoding (Internal and Connectome)

In order to simply the way in which information is processed and stored, encoded strings are used to explain the way information is to be structured, and how information is to be processed. This bytecode format uses hexadecimal command codes and chunk size references. Listed at the end of this segment is the particular command code set used by the internal compilation units as well as the simulation environments themselves. In this way, data may be easily serialized and searched,

The reason this method is used in the internal compilation chain is to simplify the process of extending the language, although the actual information stored may be too complex to read manually. By keeping a particular grammar, new additions and renovations to the language is simple while the engine is in a volatile development stage.

The downside of this, of course, is a significant hit to efficiency. To remedy this to some degree, objects searchable in the information registry are still separate entities, although their content is in a single encoded format, sans particular attributes used by the system.

Grammatical rules used by this system are actually fairly simple. At first, a command code ranging from 0x00 to 0xFF is followed by a value specifying the number of parameters. These parameters consist of three values. First is a value identifying the type of the following data structure. Next is the size in number of characters of the information set. And lastly is the information itself. For instance, the line of code

**float x = 28;**

would have the bytecode sequence

**[def] [float] [1] [x] [end] [assign] [float] [1] [x] [int] [2] [1C] [end]**

Of course, with the plain text replaced with hex values. In the case of '1C', this would be stored as plain-text hex as to allow values greater than 0xFF.

Each set of opcodes and identifiers will modify the way proceeding sets of data are interpreted. Components that request a set of arguments behave as singular objects, included with said arguments. Thus nested methods and encapsulations do not need to factor into the size information of lower-level bodies.

Mathematical functions are converted into post-fix notation during encoding, and are prefaced with a Math value identifier followed by a size parameter of the entire parsed section.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0x0 | End Of File | 0X14 | Mod | 0X28 | Outside <> |
| 0x1 | End Of Statement (;) | 0X15 | Pow | 0X29 |  |
| 0x2 | Start Body [ | 0X16 | Root | 0X2A |  |
| 0x3 | End Body ] | 0X17 |  | 0X2B |  |
| 0x4 | Start Group ( | 0X18 |  | 0X2C |  |
| 0x5 | End Group ) | 0X19 |  | 0X2D |  |
| 0x6 | Start Brace | 0X1A | And | 0X2E |  |
| 0x7 | End Brace | 0X1B | Or | 0X2F |  |
| 0x8 | Split Item (,) | 0X1C | Not | 0X30 | Cell (identifier) |
| 0x9 | Member (.) | 0X1D | Nand | 0X31 | Group(identifier) |
| 0XA | Index(:) | 0X1E | Nor | 0X32 | Float(identifier) |
| 0XB |  | 0X1F | Xor | 0X33 | Int(identifier) |
| 0XC |  | 0X20 | XNor | 0X34 | Bool(identifier) |
| 0XD |  | 0X21 | Equ | 0X35 | Char(identifier) |
| 0XE |  | 0X22 | NEqu | 0X36 | String(identifier) |
| 0XF |  | 0X23 | Gtr | 0X37 | Fiber(identifier) |
| 0X10 | Add | 0X24 | Lss | 0X38 | Expression (identifier) |
| 0X11 | Sub | 0X25 | GtrEqu | 0X39 | Synapse (identifier) |
| 0X12 | Div | 0X26 | LssEqu | 0X3A | Interface (identifier) |
| 0X13 | Mult | 0X27 | Within >< | 0X3B | Struct (identifier) |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0x3C | DeviceOut (Identifier) | 0x52 | Expression(keyword) | 0x68 |  |
| 0X3D | DeviceIn (Identifier) | 0x53 | Synapse(keyword) | 0x69 |  |
| 0X3E |  | 0x54 | Interface (keyword) | 0x6A | Rand |
| 0X3F |  | 0x55 | Struct (keyword) | 0x6B | Display |
| 0X40 | Constant | 0x56 | Device (keyword) | 0x6C | Startlog |
| 0X41 | Override | 0x57 |  | 0x6D | Wlog |
| 0X42 | New | 0x58 |  | 0x6E | Clog |
| 0X43 | Delete | 0x59 |  | 0x6F | Usin |
| 0X44 | Persistent | 0x5A | Set | 0x70 | Bind |
| 0X45 |  | 0x5B | AddSet | 0x71 | Copy |
| 0X46 |  | 0x5C | SubSet | 0x72 | Spawn |
| 0X47 |  | 0x5D | MultSet | 0x73 | Kill |
| 0X48 |  | 0x5E | DivSet | 0x74 | Unbind |
| 0X49 |  | 0x5F | Increment | 0x75 | Gauss |
| 0X4A | Cell (keyword) | 0x60 | Decrement | 0x76 |  |
| 0X4B | Group(keyword) | 0x61 | Ternary | 0x77 |  |
| 0X4C | Float(keyword) | 0x62 |  | 0x78 |  |
| 0X4D | Int(keyword) | 0x63 |  | 0x79 |  |
| 0X4E | Bool(keyword) | 0x64 |  | 0x7A | Stn (serial to neural) |
| 0X4F | Char(keyword) | 0x65 |  | 0x7B | Nts (Neural to serial) |
| 0x50 | String(keyword) | 0x66 |  | 0x7C | Usb |
| 0x51 | Fiber(keyword) | 0x67 |  | 0x7D | Fncgen |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0x7E |  | 0x94 |  | 0xAA |  |
| 0x7F |  | 0x95 |  | 0xAB |  |
| 0x80 | And (keyword) | 0x96 |  | 0xAC |  |
| 0x81 | Or(keyword) | 0x97 |  | 0xAD |  |
| 0x82 | Not(keyword) | 0x98 |  | 0xAE |  |
| 0x83 | Nand(keyword) | 0x99 |  | 0xAF |  |
| 0x84 | Nor(keyword) | 0x9A |  | 0xB0 | Math (internal) |
| 0x85 | Xor(keyword) | 0x9B |  | 0xB1 | Insert (internal) |
| 0x86 | Xnor(keyword) | 0x9C |  | 0xB2 |  |
| 0x87 |  | 0x9D |  | 0xB3 |  |
| 0x88 |  | 0x9E |  | 0xB4 |  |
| 0x89 |  | 0x9F |  | 0xB5 |  |
| 0x8A |  | 0xA0 | jump | 0xB6 |  |
| 0x8B |  | 0xA1 | break | 0xB7 |  |
| 0x8C |  | 0xA2 | goto | 0xB8 |  |
| 0x8D |  | 0xA3 |  | 0xB9 |  |
| 0x8E |  | 0xA4 |  | 0xBA |  |
| 0x8F |  | 0xA5 |  | 0xBB |  |
| 0x90 | Grid | 0xA6 |  | 0xBC |  |
| 0x91 | Matrix | 0xA7 |  | 0xBD |  |
| 0x92 | Ring | 0xA8 |  | 0xBE |  |
| 0x93 | Line | 0xA9 |  | 0xBF |  |

mRNA Ciphers

One of the most powerful features, granted also one of the most complex, is the mRNA cipher file. This is a file placed in the language support directory (**support/<language>/**) that provides information on how to convert the internal encoded language into the language of choice.

A cipher file uses the encoding table in the last section to map a language to a generated data stream. This can include block of text, formatting, and named replacements. However, creating a robust cipher text is more involved than simply comparing two tables together.