Clockwork Language Reference

Martin Leadbeater

August 10, 2020

Contents

Chapter 1

Overview

1.1 Preface

This desribes the Clockwork language, the User Guide and Programming Guide provide additional information that will help newcomers learn to program in Clockwork.

1.2 Introduction

Clockwork is a language intended to be used to describe systems of interacting components using state machines. The purpose of the language is to provide a platform for monitoring and control and it has currently been implemented as a control system using Beckhoff EtherCAT® and The Internet of Things (MQTT).

The language provides the programmer with a means to:

- define the states of a machine in terms of conditions
- define commands that can be invoked on the machine
- define transition rules that determine what actions can be performed and what state transtions are allowable
- define state entry functions that execute when a machine enters a state
- monitor the operation of a machine by sampling events as they occur and interactively inspecting the state and properties of internal components
- view the state and properties of the machine in a web browser
- view the operation of a machine in a 3D view
- interface to serial devices, network devices and modbus devices
- program embedded devices and microcontrollers (not yet available)

1.2.1 Elements of a clockwork system

A clockwork system consists of

state-machines which are the components that contain the properties and behaviour of the system

EtherCAT which is the main hardware IO platform for clockwork

channels which connect separate instances of clockwork together or connect clockwork to other tools

hmi whether it is a web application, humid user interface or command interface

monitors implemented via the external sampler/scope tools or via tools within humid

connections to non-clockwork systems such as TCP/IP connections, MQTT or serial devices

1.3 Review of the language

For a more detailed discussion of programming in clockwork, please refer to the Programming Guide or User Guide.

The main components in a Clockwork program are called Machines. Machines are the processing component of the latproc suite. They are defined as finite state machines that are interconnected by message passing. A machine will automatically detect its state by polling input conditions that are defined to match certain configurations. By issuing commands, a machine can change the state of other machines and these changes can trigger further state changes.

1.3.1 Machines

A Machine is similar to a Class in other languages in that it encapsulates state and provides methods that implement procedural logic. Machines are different to classes in that they execute as independent entities, continuously evaluating the state of their environment and reacting to it. Machines are models of processing components or real-world devices that can be interconnected. Each Machine has a current state and (normally) several other states that the machine may automatically move to or that can be manually set by processing steps in the program. Normally the state of a system includes all of its properties and their current values. We make a distinction, however, between the state of a machine and the properties of the machine, and let the rules determine how the value of a machines properties determine its state.

State-machine based modeling systems, clockwork included, use transition tables that describe state changes in response to events. The key difference between Clockwork and other state machine systems is that in Clockwork the software components can also automatically move between states by monitoring a set of rules. When a state includes a condition, we tend to refer to that as a *stable* state but these could also be called *automatic* states.

A machine is defined by a block of code that defines the states and rules

```
<machine-class-name> ‘‘MACHINE’’ <parameter> [, parameter] ...
‘‘{’’

    <state-name> ‘‘WHEN’’ <rule> ‘‘;’’
    [...]
    [ <state-name> DEFAULT ‘‘;’’ ]

‘‘}’’
```

A rule is simply a predicate expression that compares various properties and values; it evaluates to eithe *true* or *false* For example, here is a definition for a Machine that behaves like an AND gate in an electronic circuit; an AND gate is on only when both inputs are on:

```
AndGate MACHINE input1, input2
{

    on WHEN input1 IS on AND input2 IS on;
    off DEFAULT

}
```

1.3.2 Instances of machines

Clockwork deals primarily with definitions of finite state machines (referred to as MACHINES in the language) and instance of those definitions. There is a common pattern for declaring an instance of a machine; providing the name and then the machine class. The same pattern of name then element is used in most parts of the language.

Other things, such as properties and parameters or action blocks follow after the name/element pair; for example:

name machine-class [’(’ property-name ’:’ property-value ... ’)’] parameters

For example, given machine classes called ‘MODULE’ and ‘POINT’, an instance can be declared in the following way.

```
Beckhoff_2008 MODULE 2
NG_Output POINT Beckhoff_2008 5
```

This defines a Module and a point within that module. In the example, the number '2' indicates the position of the module on the bus and the number '5' defines the particular output id of the point that we want to call 'NG_Output'. After the object-class, a list of property key,value pairs may be given., for example

```
NG_Output POINT (tab:Outputs) Beckhoff_2008 5
```

describes the same point but sets a property called 'tab'[1] to the value 'Outputs'.

1.3.3 Properties and parameters

As shown, the declaration of a machine may have parameters and machines may also have properties. Properties and parameters are distinguished as follows:

- properties have default values and do not have to be predeclared unless they are used in stable state conditions
- parameters generally refer to objects that the machine manipulates and serve to provide an internal alias for a globally defined object. Use of parameters provides for the reuse of machines for different parts of the system.
- when a parameter changes state, the machine receives an event that it may act on to perform an action.
- when a property of a machine or one of its parameters or local variables is changed, the machine reevaluates its stable states.

The above machine *definition* for this AndGate requires two *parameters* for the two inputs and defines a rule on state 'on' that will activate when both inputs are on. The statement '`off DEFAULT`' defines a default condition on the `off` state that always returns true. There can be only one DEFAULT state in each machine but it is not required; the presence of a default state ensures that machines of this type will be in the on state when the condition is true and only when the condition is true.

A machine definition can be instantiated by providing a name, the name of the machine definition, also called the machine 'machine class', and a list or parameters that are required by the definition.

```
<instantiated-name> <machine-class-name> [ <parameter> [ , <parameter> ... ] ] ‘‘;’’
```

For example, following is a machine named 'and_gate' that used the definition 'AndGate' and takes two other machines as parameters:

```
and_gate AndGate a, b;
a FLAG;
b FLAG;
```

Note that the FLAG machine type is defined internally in the language and at this stage it can be regarded as a machine defined like this:

```
FLAG MACHINE
{
    on STATE;
    off INITIAL;
}
```

Thus a FLAG can be either on or off and is initially off. There are no rules to define how the machine might automatically move between those states so it is left to other parts of the program to change the state of the FLAG as required.

1.3.4 State Transitions

When a machine changes state, several stages occur:

1. if the machine has a leave action it is started
2. when the leave action is completed the current state of the machine is changed
3. once the machine state has changed the enter action for the state is started if there is one
4. if the machine has dependencies those machines are told the machine is entering a state

[1] The web interface to cw and iod happen to use this property to group various items into tabs on the web page

Symbol	Name	Type	Description		
+	Plus	binary	adds the lhs and rhs		
-	Minus	binary	subtracts rhs from lhs		
*	Times	binary	multiplies lhs and rhs		
/	Integer Divide	binary	divides lhs by rhs and truncates the result		
%	modulus	binary	returns the remainder after dividing lhs by rhs		
^	XOR	binary	returns the exclusive or of the lhs and rhs		
|	OR	binary	returns the result of a bitwise or operation on the lhs and rhs		
&	AND	binary	returns the result of a bitwise and operation on the lhs and rhs		
&& or AND	Boolean AND	binary (bool)	performs a boolean and of the expressions on the lhs and rhs		
|| or OR	Boolean OR	binary (bool)	performs the result of an or operation on the expressions on the lhs and rhs		
!	Negate	unary	returns the bitwise inverse of the rhs		

Table 1.1: C-based Expression operators

1.3.5 Expressions

There are two uses of expressions within Clockwork: definition of states and calculation of values; conditions are boolean expressions whereas calculations may produce any kind of result that a property can hold. Expressions are modelled on expressions in the C language; the operators and precedence can be found in Table 1.1.

1.3.6 Passive states

States can be defined by providing a name along with the 'STATE' keyword as follows: *state_name* STATE. When the machine enters a state, a script can be invoked to cause behaviour, often resulting in other state changes.

Such states can be used within transitions to define how the state of the machine changes in response to events. For example, the following states flip from one to the other continuously:

```
off STATE;
on STATE;
ENTER off { SET SELF TO on }
ENTER on { SET SELF TO off }
```

In the above example, the keyword 'SELF' refers to the current state of the machine executing the script. the SET..TO.. statement causes the state to change to the named state.

1.3.7 Stable States

The definition of machines with a set of states and actions is a common approach and works well for describing event driven processes. In clockwork, machines are repeatedly monitoring their inputs to ensure that the machine's state is consistent with the definition of that state. When the machine's input change to a configuration that implies the external system has changed, clockwork machines automatically shift to the first state they can find that matches the measured conditions.

Clockwork can automatically check a set of rules to determine what state a machine is in. This is used by applying a WHEN clause to a state name:

```
busy WHEN customers > 0;
idle WHEN customers == 0;
```

The above states, 'busy' and 'idle' switch automatically depending on the property 'customers'. We call theses 'stable' states based on the idea that the machine finds. Stable state tests are performed in order and stop evaluating as soon as a match is found.

As another example, the following machine tries to stay in an inverted state compared to its input:

```
Inverted MACHINE input {

    on WHEN input == off;
    off DEFAULT

}
```

At first glance, it seems that all instances of 'Inverted' will always move to the opposite state of their inputs. However, in clockwork, we do not formally require that machines conform to any particular interface, for example, the only requirement in the above 'Inverted' machine is that the input might at some time enter the 'off' state and when this happens, the Inverted machine will be 'on'.

Both the input and the inverted machine may be in the same state for short times, for example, for a short time after the machines are first enabled, they will both be in the 'INIT' state. Also using the above definition we cannot guarantee that the inverted machine is actually the inverse of the input since the input may have pass through other states and during all of these states, the inverted machine will stay 'off'.

The above definition may be made more strict by tightening the definition of the off state:

```
Inverted MACHINE input {

    on WHEN input == off;
    off WHEN input == on;
    unknown DEFAULT;

}
```

This definition enables us to be more confident in claiming the machine to be an inversion but if the input machine has many other states we may have to deal with the 'unknown' state in some way. In practice, the previous definition is quite practical and these subtle distinctions are not helpful to the modelling process.

1.3.8 Transition statement

Transitions define how a system changes state based on the receipt of an event or command. Transitions can:

- be used to prevent arbitrary state changes
- can be guarded by requirements that must be met for the transtion to occur
- can define a command to execute when a machine is set into a state by another machine

The following Machine changes between a 'ready' and 'started' state when a start or stop command is received.

```
StartStopTimer MACHINE {

    ready INITIAL; started STATE;
    COMMAND stop { timer := TIMER; }
    COMMAND start { }
    TRANSITION started TO ready ON stop;
    TRANSITION ready TO started ON start;

}
```

1.3.9 Event handlers (methods)

Event handlers are commands that execute when a machine changes state. Currently handlers are executed upon entry and exit of a state, they are defined using:

```
ENTER state-name { actions }
```

and

```
LEAVE state-name { actions }
```

1.3.10 Commands

Commands are lists of actions that are executed in response to receipt of a message. While commands are being executed, the stable state evaluation for a machine are not executed. Commands may have an associated state that the machine moves to while executing the actions.

The mechanism for this will be to add clauses to identify timeouts or other conditions:

```
COMMAND example { statements } ON ERROR message
```

or

```
COMMAND example { statements } ON TIMEOUT message
```

1.3.11 Exporting properties

A machine can export its properties for use by Modbus applications through use of the EXPORT statement...

1.3.12 Lists

Machines can be grouped together into lists. Lists have some features and restrictions:

- machines may be placed on a list but every entry in the list must be unique, attempting to include an item a second time leads to undefined bahaviour
- messages sent to a list are also sent to all entries of the list
- entries of a list remain in the order they are added (ie., the INCLUDE statement adds an elment to the end of the list) unless the list order is changed by a SORT statement
- lists can be reordered by using a property of the entries as the sort key, by default the VALUE property is used
- lists entry states or properties can be packed into an binary value property using the BITSET FROM statement and can be restored using the SET LIST ENTRIES statement
- items can be tested to see if a list contains a reference to them (eg., my_list INCLUDES my_machine)
- set operations are available to load lists based on operations involving other lists

1.3.13 Monitoring globals

Generally a Machine will only receive events from machines that are passed as parameters or that are included within the Machine. By using the GLOBAL keyword, a Machine can monitor state changes on other machines, not passed as parameters. By this technique, it is possible to avoid passing parameters to machines and to simply refer to all machines globally. This is probably fine for small systems but less practical for larger systems.

1.3.14 References

The builtin machine, REFERENCE provides a way to link machines together. When an object is assigned to a reference, the reference enters the ASSIGNED state, when the reference is cleared, it enters the UNASSIGNED state. Other machines can detect these changes by listening for ASSIGNED_enter or UNASSIGNED_enter mesages. If an assignment is attempted to a non-existent machine any preexisting assignment is cleared and the new assignment fails.

1.3.15 Conditions

Conditions are lists of boolean expressions that are generally used to determine the current state of various parts of the machine. Conditions are implemented by the use of separate state machines. that contain a state (normally called 'true') that indicates the condition is true and a state (normally called 'false') that is the default.

1.3.16 Notes

An action is a list of steps, such that each step requests that a machine enters a state or waits for a message. Regardless of the action, there are two components: sending a message and waiting for a response. On receipt of the response, the action transitions to the next step in the list.

When a state change request is received, the transitions on the target machine are searched for an action that will satisfy the requested state change. That action causes a further sequence of actions to be queued by having the machine enter a substate with each action sending a message and waiting for a response.

1.4 Machines

1.4.1 Parts of a machine

Machines are the core of clockwork programming; they define a virtual representation of the operation of real machines or processes and in so doing provide a correspondence between the state of the program and the state of the machinery in the real world.

The definition of a machine includes several sections:

- declaration and initialisation of properties (also called options)
- declaration of states
- definition of stable states
- definition of transitions
- definition of event handlers
- definition of commands
- export declarations

These sections are only for conceptual convenience; declarations and definitions can be freely intermixed but the order that stable states are declared is important.

To define a machine use: *class_ name* MACHINE *parameter1, parameter2, ...* { } and place the definition between the braces.

To create an instance of a machine, use: *name class_ name param1, param2, ...* ;. There can be any number of instances for a given machine definition. The trick is mostly to do with what goes between the braces.

1.4.2 Components and scope

The language deals with Finite State Machines, which the language simply calls 'Machines' and their states. The program defines a machine as having certain states and provides conditions and scripts that can identify the current state and change states.

some predefined objects; Modules, Points, Machines, Values, Conditions and States. These objects may have parameters and may also have properties. Properties and parameters are distinguished as follows:

- properties have default values and do not have to be provided
- parameters generally refer to objects that the machine manipulates and serve to provide an internal alias for a globally defined object. Use of parameters provides for the reuse of machines for different parts of the system.

1.4.3 Parameters

When a machine is instantiated, parameters are resolved by providing either a symbol name or a symbol value. The Machine Instance retains a list of ParameterReferences and for parameters passed by value, a local symbol table allocates a local name

1.4.4 States

A state is defined by a particular configuration of inputs or execution of an action.

1.4.5 Actions

An action is a sequence of steps that are executed in response to the receipt of a message, including steps taken dugin an entry function.

1.4.6 Transitions

Transitions occur automatically when a machine is idle, when it detects a change of state. When a transition occurs:

- the timer is reset,
- the state variable 'CURRENT' is updated
- a state change message is queued for delivery to interested parties
- the entry function for the state is executed

1.4.7 Events

An event corresponds to the sending of a message. Examples of events include timers and changes in input levels or analogue or counter values. Timer events are intented to trigger reasonably precisely, based on a fixed time after the message was sent.

1.4.8 Message Passing

Machines can send and receive messages directly, using the commands SEND and CALL and the handler RECEIVE or it can send messages indirectly by attempting to change the state of the target machine.

1.4.8.1 RECEIVE and RECEIVE..FROM

Each machine can listen for messages from any entity it knows of. Messages are sent when entities change state or when a script deliberately sends one. To listen for a message, use:

RECEIVE objname.message '{' actions '}'

or

RECEIVE message FROM objname '{' actions '}'

This identifies the sending entity from its local name and registers a listener for the *entiyname_statename_enter* message. In a future release, the current machine will also registered as a dependant of the sender. Currently, messages are only received from parameters, local instances and machine instances that are specifically listed in GLOBALS.

1.4.8.2 SEND and SEND..TO

1.4.8.3 CALL

The CALL statement acts like theSEND except that the statement hangs and waits for a response from the target. There are a few forms:
CALL command
CALL command ON ERROR CALL command
CALL command, EXECUTE command WHEN predicate

1.4.9 Exceptions

[Note: not implemented]An exception is a message sent using the 'THROW' command. It is similar to a message sent with 'SEND' except that there is no requirement that there is a machine listening for the message. Messages that are thrown are caught using the 'CATCH' command.

Chapter 2

Statements

This chapter provides examples of common statements and clauses used in clockwork. (TODO)

Statement	*Comment*
`y := ABS x`	if x is negative, y becomes -x otherwise, y becomes x. If x is a string, y becomes zero
`LOG "x:  " + x + " y:" + y`	the values of x and y as strings are concatenated with the other strings to produce a single string that is then logged to standard out
`myOnTime := myOnTime + TIMER`	the current state time is added to the property: myOnTime
`start_time := NOW`	the current time in milliseconds
`IF (start_time == 0) { a := b }`	example simple if
`WAIT 1000;`	the machine will halt execution for 1 second and will not process messages received or check for state changes during this time. Execution of the current block continues after 1 second.
`SEND hello TO SELF`	the message is queued for execution and will start once all of the actions in the current block has completed.
`raw := TAKE FIRST FROM queue;`	the first item in the list named queue is removed from the list and placed into raw
`tmp := COPY ‘[0-9][0-9]+‘ FROM raw;`	the first sequence of digits in raw is copied into tmp
`n_tmp := "1" + tmp;`	tmp is appended as a string to the string “1” and the result placed into n_tmp
`n_tmp := COPY ALL ‘[0-9]‘ FROM tmp;`	all of the digits in tmp are copied into n_tmp
`Weight := 0 + n_tmp`	n_tmp as an integer is added to zero and the result placed into Weight
`CLEAR test;`	all entries are removed from the list named test
`COPY ALL FROM source TO dest;`	all items in the list named source to the list named dest
`ADD f1 BEFORE FIRST OF test;`	the property or machine named f1 is placed before the first item in the list named test
`ADD f1 AFTER LAST OF test`	the property or machine named f1 is placed after the last item in the list named test

Statement	*Comment*
`MOVE ITEMS 2 TO 3 OF test TO work`	the second and third items of the list named test are removed and added to the end of the list named work
`x := TAKE LAST FROM test`	the last element of the list test is removed and placed into the property x. It is possible in this case for x to be holding a machine reference but the only thing that can be done with this value it to place it onto another list
`WAITFOR test IS off`	the current machine pauses execution and waits until the machine named test enters the off state. During this time, the current machine does not process messages or check for automatic state changes
`DISABLE script`	the machine named script becomes disabled
`x.a := (Zone + -1) % 9 - (Location - 1) * 3 + 1`	the property a in machine x is set to the result of the given calculation (% is the modulus operator)
`LOG "a:  " + FORMAT a WITH "%04d"`	the property a is converted to a string with 4 digits and leading zeros and appended to “a: “ before being logged to standard out
`TRANSITION on TO off ON next`	when the message ’next’ is received, the machine will leave the on state and go to the off state. The message will be ignored if the machine is not in the on state
`RECEIVE in.on_enter { SEND next TO SELF; }`	when the machine named in enters the on state, send the message “next” to the current machine.
`val := one | six`	the property one is bitwised or’ed with the property six and the result placed in val
`one WHEN BITSET FROM a == 1`	a must be a list in which case all items in a are interpreted as boolean flags where the state ’on’ represents one and all other states represent zero. These flags are collected together in order to form an integer value from a set of bits
`on WHEN SELF IS off AND TIMER >= 1000 || SELF IS on AND TIMER < 1000;`	the cuttent machine will move to the on state if it is off and has been off for at least 1 second or stay in the on state if it is on and hasn’t been on for a second yet
`RECEIVE calc { a := NOW; }`	executes the action to assign NOW to a when the message ’calc’ is recieved
`SEND calc TO worker`	sends the message ’calc’ to a machine names worker
`a FLAG(val:1);` `b FLAG(val:2);`	instantiates a machine named ’a’ that uses the behaviour defined for FLAG and addes a property ’val’ within initial value 1 and instantiates a machine named ’b’ the same way ...
`abc LIST a,b;`	instantiates a list called ’abc’ with two machines, a and b initially

Statement	*Comment*
`COPY ALL FROM list TO copied WHERE list.ITEM.val == 1`	copies all machines in the list named list to the list named copied where the condition is true that the machine to be copied has a property 'val' with value 1
`COPY ALL FROM data TO result SELECT USING sel WHERE data.ITEM.val == sel.ITEM.x`	joins two lists, 'data' and 'sel' where the val property of items in data match the x property of items in sel
`off WHEN af IS on;`	sets the current machine to off when the machine named 'af' is on

`stable WHEN x.VALUE == last.VALUE;`
`recalculate DEFAULT;`
`ENTER recalculate { last := x }`
	a pair or rules cause a machine to recalculate whenever a value in x changes. Inside the handler for recalculate, the value in x is copied into last

Chapter 3

Subtleties

3.0.10 Enabling and disabling machines

Machines in clockwork may be prevented from operating by disabling them. Messages sent to disabled machines are silently ignored and attempts to set the state of a disabled machine triggers an exception. TBD more information about interactions with disabled machines.

3.0.11 Controlling startup

When the clockwork environment starts, all machines initially disabled and are enabled depending on the following rules:

if the application program implements a machine with the class 'STARTUP', only that machine is enabled and the entire startup process is left to application control.

if there is no instance of a 'STARTUP' machine machines that have a property: 'startup_enabled' with a value of false are not enabled and all other machines are enabled.

The system attempts to start machines in order such that machines passed as paremeters to other machines are enabled first.

Machines that include a 'DEPENDS ON' clause can only enabled when the named machine has been enabled. If the named machine does not exist it is not possible to enable the dependent machines. In the case where the application does not provide a STARTUP machine, when the machines that are the object of a DEPENDS ON clause are enabled or disabled those machines that depend on them are automatically enabled or disabled.

3.1 Details

Chapter 4

Vocabulary

Within the program text, reserved words are presented in all capitals to distinguish them from user defined values. The language is case sensitive.

4.0.1 Glossary of Reserved Words

16BIT defines an exported modbus property as a 16bit integer

32BIT defines an exported modbus property as a 32bit integer

ABORT aborts the current action with an error status

ABS returns the absolute value of a number

ADD adds an item to a list in a specific position

AFTER a qualified used when adding an item to a list indicates the item will be after the given item

ALL used with COPY and EXTRACT to collect all matches from a property

AND (also &&) used in predicates (state rules) and boolean expressions to join two boolean clauses

ANY used when checking the state of all items in LISTs

ARE used when checking the state of all items in LISTs

AS used for type convertion to STRING, FLOAT or INTEGER

ASC used to define an ascending sort order (alternate form of ASCENDING)

ASCENDING used to define an ascending sort order

ASSIGN used to set a reference value

AT used within a RESUME clause to resume a machine at a named state

BECOMES name of the assignment operator (:=)

BEFORE used when insterting items into a list

BETWEEN used to check whether a value is in a given range

BITSET used when converting between a LIST and a number representation of a set of boolean flags

BY used as a conjunction in SORT..BY and INC/DEC BY statements

CALL sends a message to a given machine and waits for completion of handlers attached to that message.

CATCH defines an exception handling action

CHANGING used to test whether a machine is changing state

CHANNEL defines a connection between one clockwork program and another or between a clockwork program and external tool

CLASS used to get the name of the definition of a machine

CLEAR resets a LIST or REFERENCE

COLLECT used for collecting data from an external source

COMBINATION used to form the union of two sets

COMMAND defines an action that handles a message arriving at a machine from a SEND or CALL action

COMMANDS used in defining an interface for a CHANNEL

COMMON used to collect common elements in two lists

CONDITION defines a condition object that has a value of true or false depending on the state of an expression

CONSTANT a predefined machine class that acts like a constant from a syntactic viewpoint

COPY used to collect items from a list and put them into another list or to copy substrings matching a pattern from a given property

COUNT used to find how many items in a list

CREATE creates a list with some elements of another list

DEC decrement a property by an amount

DEFAULT defined an always-true rule that is evaluated last in a machines automatic states list

DESC used to specify a descending sort order (alternate name for DESCENDING)

DESCENDING used to specify a descending sort ordeer

DIFFERENCE used when copying members from a list that are unique to that list compared to another

DISABLE used to make a machine unable to respond to requests to change state or process commands. Properties of a disabled machine can still be changed. The machine stays in the state it was is when it was disabled

DISABLE used to stop a machine from further processing until enabled or resumed

DISABLED used to test whether a machine is disabled

DURING used to define a state that should be used while a given action is being excuted in response to a specific message, after the completion of the action, the machines automatic state changes or transitions will execute. If there are no automatic state rules or transitions, the machine will stay in the given state.

ELSE marks the beginning of the code that is executed when the IF condition evaluates to false

ENABLE makes a machine able to respond to requests and changes state. The machine enters its initial state and executes the initial state entry method. The leave event to the previous state is also executed and the message is sent to dependent machines.

ENABLED used to test whether a machine is enabled

ENTER used to define an action that executes when a machine starts to enter a state

ENTRIES used in the conversion between lists and numeric representations of a set of booleans

EQ the name of the '==' operator

ERROR used to define what happens if an action experiences an error or timeout

ERRORS used when indicating that errors should be ignored for a given action

EXECUTE extends the definition of a stable state to specify a command that should be executed when a special sub-condition becomes active.

EXISTS used to test if a machine is defined

EXPORT used when exporting properties or options to modbus or humid

EXPORTS used when defining a channel interface to indicate whether exported options are passed to the channel

EXTENDS used for type extension for channels

EXTRACT used with regular expressions to extract subexpressions into options

FAILURE used when defining what to do if a transition fails

FIND returns a list of objects that are in the given state, options may be added to restrict the range of items found [not implemented]

FIRST indicates the first item in a list

FLAG a builtin machine definition with an on and off state, commonly used as a boolean indicator

FLOAT declares a variable to have a floating point value

FLOAT32 declares the floating point format used when a float value is exported

FOREACH performs an operation for each object within the given list that are in a particular state [not implemented]

FORMAT used when converting a value to a string

FROM used in several expressions where a value is bing generated or copied from another

GE the name of the '>=' symbol

GLOBAL used to introduce a dependency with an instance of a machine that is not directly connected

GLOBALS lists machines, external to the current class that the current machine is monitoring. State changes on the machines listed as GLOBAL cause a reevaluation of stable states and generate events that can be received.

GT the name of the '>' operator

IDENTIFIER provides a unique id for a channel

IF used within operations (Commands and Enter functions) to provide for conditional behaviour

IGNORES used in a channel interface definition to desribe machines that the channel ignores

IN used when searching for an item in a list, replacing a pattern within a string, copying items from lists and similar actions

INC increments an option by an amount

INCLUDE used to add a machine to a list

INCLUDES used in testing whether a machine is in a list

INDEX identified the position (numbered from zero) of an item in a list,

INIT a predefined state that statemachines enter as soon as they are enabled.

INITIAL defines a state as the state to use for the initial state of the machine.

INTEGER used in converting a value to an integer

INTERFACE introduces a new interface definition, used by a CHANNEL

ITEM used as a proxy name for the item that a reference is pointing to or the current item within a list search

ITEMS used when adding or removing machines to/from a list

KEY used to define a unique secret for a channel

LAST refers to the last element of a list

LE the name of the '<=' symbol

LEAVE declares an action to be executed when a machine leaves a given state

LENGTH defines an action to be executed when a machine leaves a given state

LINE

LINKED used in an INTERFACE to indicate that machines dependent on another are to be passed through a channel

LOCAL defines an option that does not generate property change messages when its value changes, used for temporary variables in a calculation

LOCK prevents a machine from being changed by other external machines until the machine that locked it unlocks it again, used as a synchronisation mechanism to prevent, for example, multiple machines simultaneously changing the state of a machine.

LOCK provides a way to mark a protected section to prevent, for example, multiple machines simultaneously changing the state of a machine.

LOG places a message in the application output and in the message log accessible from iosh with the MESSAGES command

LT the name of the '<' symbol

MACHINE defines a programmed component within the system. All control functionality is implemented within these objects. Machines are based on finite statemachine concept and include both monitoring (automatic state switching) and control (forced state changes to cause actions).

MACHINES used when defining the machines that are passed through a channel

MATCHES indicates whether a string mathes a pattern

MATCHES used in an expression to test for a property matching a given pattern

MATCHING used to define selectors for machines that are passed through a channel

MAX the largest value in a list

MEAN the arithmetic mean value of items in a list

MIN the arithmetic minimum value of items in a list

MODBUS used in defining selectors for machines that are passed through a channel

MODULE declares use of an EtherCAT IO Module

MODULES begins a section for declaration of EtherCAT IO Modules

MONITORS used in defining selectors for machines that are passed through a channel

MOVE takes an item from a list and puts it into another

NAME used when defining selectors for machines that are passed through a channel

NE the name of the '!=' operator

NOT a prefix operator that inverts a boolean result, such as in the predicate of automatic state rules

NUMBER

OF

ON

OPTION defines an option (also called 'property') name and value

OR (also ||) used to join expressions within a condition

PERSISTENT a property that indicates that propertye changes on this machine will be published. Machines marked as persistent are no longer automatically enabled on startup.

PLUGIN

POINT refers to an addressable port within a module or a builtin machine with that name (see Section [subsec:Builtin-classes])

PROPERTIES part of the COPY PROPERTIES ... statement

PROPERTY indicates that the following symbol is used as the name of a property containing a value, rather than being a value

PROPERTY_CHANGES indicates that a channel emits property changes

PUBLISHER indicates in a channel interface that the channel is a publisher (sends state and value changes via publish/subscribe)

PUSH adds an item to a list

QUOTE

RAISE send a message that any machine may catch. It is an error if a message is raised but not caught. [not implemented]

READONLY used in an EXPORT specification to mark the property as read only

READWRITE used in an EXPORT specification to mark the property as read/write

RECEIVE indicates that a machine should be informed when a specific object sends a particular message. The RECEIVE statement has an associated set of actions that are acted when the event is collected.

RECEIVES

REPLACE

REPORTS

REQUIRES indicates that the associated condition must evaluate to True in order that the transition can occur or that a command can be executed [not implemented]

RESULT the returned value after an expression has been evaluated [not implemented]

RESUME resumes execution of the disabled machine from the beginning of its current state

RETURN performs a normal exit from an action without setting an error (compare to ABORT)

ROUTE redirects an incoming message

SELECT

SELF used in expressions to refer

SEND is used to send a message to a machine, this message must be captured by the machine, using a RECEIVE statement

SENDS

SEPARATOR

SET causes a machine to move to a given state after executing the optional command associated with the transition

SHARES defines a list of machines that shared on a channel. Shared machines can be updated by both ends of a channel

SHUTDOWN cause the clockwork daemon to exit

SIZE provides the number of items in a list

SORT used to sort a list of machines based on the value of a property or a list of value by the value

SOURCE is a variable that holds a reference to the object that send the current message or issued the current command [not implemented]

STATE defines a state name so that this state can be used in a transition or can be set by other machines.

STATE_CHANGES indicates in an interface definition whether state changes will be sent through the channel

SUM calculates the arithmetic sum of the items in a list

TAG links a FLAG to a sub-condition on a stable state so that the FLAG is automatically turned on and off to trak whether the subcondition is true or not.

TAKE removes one or more items from a list

THEN marks the beginning of the code that is executed when the IF condition evaluates to true

THROTTLE indicates that a channel should limit the rate at which it propagates state transitions or value changes

THROW stops execution of the current action and sends a given message to be caught by a related CATCH action

TIMEOUT defines an expected duration for the given statement to complete. If the statement doesn't complete within the given time an error action is executed in the same was as if a THROW had occurred

TIMER a numeric value in milliseconds giving the amount of time a given machine has been in its current state

TO used within a transition to indicate the destination state, used within a message operation to indicate the target of the message

TRANSITION describes the command that can be used to move the machine from one state to another

TRIM not implemented

UNLOCK reverses the effect of LOCK, to permit other instances execute code in a critical section.

UPDATES used in a channel interface, to indicate machines that are updated by the channel

USING as part of the transition statement, the Using clause indicates the command that is used for the transition

VARIABLE a predefined machine class that is persistent and acts like a variable from a syntactic viewpoint

VERSION used in a channel interface to identify its version

WAIT pauses execution of the current method for the set amount of time

WAITFOR pauses execution of the current method until the given machine enters the given state

WHEN defines a set of conditions that indicate a machine is in a particular, stable state. These conditions are not evaluated when the machine is in a transitional state.

WHERE used for bulk enable/disable/resume operations as well as in selectors when copying items from list and also used in interface definitions

WITH used in a variety of clauses such as channel interface definitions or when building a value from another or from a list. Also

WITHIN used to indicate a command that can only be executed when the machine is in a given state

4.1 Properties

Machine instances share some standard properties

current the current state of the instance

referers a list of machines that refer to this machine

references a list of machines this one refers to

sends a list of messages this machine sends

receives a list of messages this machine receives

4.1.1 Builtin classes

Builtin classes such as FLAG and BOOLEAN are readily implemented using the language itself. The POINT class, however is special as it provides an interface to hardware. Integers are currnetly implemented through properties.

VARIABLE

MODULE

CONSTANT

FLAG two state machine with states 'on' and 'off'

POINT a machine that links to a IO Point or a digital value accessible through the Internet of Things protocol.

PUBLISHER a machine that publishes changes to its 'message' property to Internet Of Things brokers using its 'topic' property for the IoT topic.

SUBSCRIBER a machine that subscribes to messages from Internet Of Things brokers to update its 'message' property when changes occur on the topic named in its 'topic' property.

(...TODO)

4.2 Syntax

(TODO)

$$
\begin{aligned}
program &= definition\, program \mid \emptyset \\
definition &= name\, object\, parameterlist \\
parameters &= parameter \mid parameter\,',\,'\, parameter \\
parameter &= value\, parameter \mid \emptyset
\end{aligned}
$$