Clockwork Language Reference (draft)

Martin Leadbeater

October 1, 2013

1 Preface

Please note that this is *still* a very early draft.

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

2.1 Machines

The main components in a Clockwork program are called Machines. 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.

The key difference between Clockwork and other state machine systems is that in Clockwork the software components automatically move between states by monitoring a set of conditions that define the states. As with other systems, it is also possible to define transition tables that describe how a system moves between states based on events. Conditions on a machine are defined in terms of the state and properties of itself and other machines, Machines continuously evaluate these conditions and automatically switch to the first state found that matches the condition, executing its state entry function as it does so. When a state includes a condition, we tend to refer to that as a *stable* state but these could also be called *automatic* states.

Each type of machine is defined by a block of code that provides the name, parameters, properties, conditions and processing of the machine. For example, here is a definition for a Machine that behaves like an and gate in an electronic circuit.

```
AndGate MACHINE input1, input2
{

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

}
```

The machine definition for this AndGate requires two parameters for the two inputs and defines a condition 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 'class', and a list or parameters that are required by the definition. For example, following is a machine that monitors two inputs and turns on when both inputs are on:

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

Note that the FLAG machine type is defined internally in the language but it is equivalent to 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.

2.2 Controlling outputs

The above AndGate machine does not directly control any other part of the system, it simply sets its own state based on its inputs. After detecting the condition where both of the inputs are on, we may want to do something about it. The AndGate can be adjusted to, for example, turn on a light when the two inputs are on, to do this, we add an ENTER method for the on state:

```
AndGate MACHINE input1, input2, output
{
    on WHEN input1 IS on AND input2 IS on;
    off DEFAULT;
    ENTER on { SET output TO on; }
    ENTER off { SET output TO off; }
}
light FLAG;
a FLAG;
b FLAG;
and_gate AndGate a, b, light;
```

This approach, where we pass the object to be controlled makes the linkage explicit but there is alternative, where another machine monitors the state of the AndGate and then performs the control function:

```
LightController MACHINE input, output {
      active WHEN 1==1, TAG output WHEN input IS on;[1]
}
```

in this case, AndGate does not manipulate or even know about the output:

```
AndGate MACHINE input1, input2
{
    on WHEN input1 IS on AND input2 IS on;
    off DEFAULT;
}
```

It is not clear yet whether this approach is better or worse in general however, if multiple machines need to be driven by the AndGate, the separate controller pattern is much more reusable.

So far we have been using FLAG machines as placeholders for machines that have an interface to hardware. For the sake of describing the language, this is sufficient and has the advantage that a command interface or web page can be used to alter the flags for test purposes. Please refer to the user manual for information about the command interface and the web interface.

[1] Note the test 1==1 is a functional placeholder; the indention is to define the value TRUE for this kind of situation.

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: Expression operators

2.3 Parts of a machine

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. Please refer to section 3 for more information about the definition of machines.

2.4 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.

3 Machines

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.

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.

3.1 State definitions

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.

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.

3.2 Transitions

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;

}
```

The following machine steps through states upon receipt of the 'next' command but may not transtion if the appropriate 'finished' flag is not set. This example uses an advanced feature that automatically sets a FLAG value as part of the stable state evaluation. While the 'next' command is executing, the Machine enters the 'changing' state. In this example this is important because a transtion to a stable state can not occur if the condition for the stable state is not true. In this example, if the 'next' command had any actions, those actions would be executed each time the event occurs but the automatic transtion will not occur if the requirement is not met.

```
Cycle MACHINE {

    a_finished FLAG;
    b_finished FLAG;
    c_finished FLAG;
    a WHEN SELF IS a OR SELF IS INIT OR SELF IS changing,
        TAG a_finished WHEN TIMER > 5000;
    b WHEN SELF IS b OR SELF IS changing,
        TAG b_finished WHEN TIMER > 5000;
    c WHEN SELF IS c OR SELF IS changing,
        TAG c_finished WHEN TIMER > 5000;
    ENTER a { SET a_finished TO off }
    ENTER b { SET b_finished TO off }
    ENTER c { SET c_finished TO off }
    changing DURING next { }
    TRANSITION a TO b ON next REQUIRES a_finished IS on;
    TRANSITION b TO c ON next REQUIRES b_finished IS on;
    TRANSITION c TO a ON next REQUIRES c_finished IS on;


}
```

3.3 Event handlers (methods)

Event handlers are commands that execute when a machine changes state. Currently handlers are executed upon entry to a state but no handler is called when the system leaves a state (this is a topic for future).

3.4 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.

Note: there is currently no way to abort the exectuion of a command, this feature will be added in future.

3.5 Exporting properties

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

3.6 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.

4 Details

4.1 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.

Names can be given for new instances of objects, by first providing the name, then the object and its properties and parameters, i.e.,

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
```

define 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'[2] to the value 'Outputs'.

4.2 Properties and states

We make a distinction between the state of a machine and the properties of the machine, in fact, we regard the state of the machine as one of its properties although more strictly, we should actually include the value of a machine's properties as part of its state. The definitions are

4.3 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.

5 Vocabulary

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

5.1 Glossary of Reserved Words

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

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

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

AND (also &&) used to join expressions within a condition

AT used with RESUME to cause execution to resume at a nominated state

BECOMES (also :=) used to assign values to properties

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

BY used with INC and DEC to change the step size

CATCH marks the beginning of an operation to be performed if any sends a particular message. [not implemented]

COMMAND defined a method that does not change the state of its machine while it is executing.

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

COPY copies substrings matching a pattern from a given property

DEC decrement a property

DEFAULT a stable state that is evaluated last and that always matches

DISABLE

DURING used to define a transitional state that occurs during the exectuion of a command

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

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

ENABLE

ENTER defines an operation that must be performed during entry to a state. The ENTER function may be restricted to performing functions that are atomic, such as sending message. Currently, these operations are not restricted but cause the machine to enter an invisible substate of the target state.

EXPORT indicates that a property should be exported to modbus

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]

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

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.

INC increment a property

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

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.

LOG emit a text string in the log

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

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

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).

MODULE defines an addressable module that sits on the EtherCAT cabling

NOT (also !)used to invert expressions within a condition

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

PERSISTENT a property that indicates that state changes on this machine will be published and at startup, this machine will be automatically enabled and initialised from its last known state.

POINT refers to an addressable port within a module or a builtin machine with that name (see Section 6.12)

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

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

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.

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]

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

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

SHUTDOWN cause the clockwork daemon to exit

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.

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.

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

TIMER

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

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

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

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.

5.2 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.

5.3 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.

6 Machines

Machines are the processing component of the latproc programming system. 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.

6.1 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.

6.2 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

6.3 States

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

6.4 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.

6.5 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

6.6 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.

6.7 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.

6.7.1 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.

6.7.2 SEND and SEND..TO

6.8 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.

6.9 Statements

6.10 Properties

Machine instances share some standard properties

current the current state of the instance

referers a list of machinesthat 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

6.11 Expressions

Expressions are based on C language expressions, standard arithmetic and bitwise operations are supported (with the exception at the moment of bitwise shift operators).

6.12 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.

7 Other features

7.1 Expressions

Recently added

- XOR (‘^’)
- Bitwise AND (’&’), OR (’|’) and negate (’!’)

8 Syntax

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