The Future of the Clockwork Programming Language

Martin Leadbeater

November 3, 2021

Abstract

Clockwork has been a work in progress since 2010, it was published in github around 2012. It has been able to be used to reliably control machinery since 2013 and has has demonstrated quite good code reuse and has been able to implement complex behaviours with small code components. This paper describes various language features that are being considered for future versions of Clockwork, based on our experiences with this type of programming.

Contents

1 Introduction

Clockwork is not strictly typed but once a value is assigned to a variable a type is inferred and used to affect the type of the result. Program control within clockwork is done by defining actions that take effect on state changes; within those actions, calculations can be performed and other entities can be affected in various ways.

1.1 Types

- Integers
- Strings
- Floating point calculations
- Binary values
- Sets and relationships
- Lists
- References
- Type conversions (x AS type)
- Key-value access to redis has been implemented via a plugin

1.1.1 Not implemented

- Streams (also generators)
- Replace the current symbol table and Value system with a more efficient and robust implementation
- Type extensions (except for channels)
- Records/Property Sets
- Database/Key-Value
- Queue

1.2 Data

- Sharing properties or data
- Events with data
- Copying properties en mass via variants of COPY PROPERTIES

1.3 Features

Not all of these are implemented directly, some of these can be implemented in library code or plugins and do not need to be language features per-se.

- Substates (implemented via nesting of machines)
- Exceptions [prerelease] (THROW, CATCH), no global CATCH at present
- Conditions
- Processing priorities - low priority/slow scan inputs
- Functions (passing parameters and getting a return result)
- External function calls
- Dynamic allocation (ie creating and destroying data items)
- Dynamic configuration (creating and destroying machines)
- Modular/dynamic components
- Support of parallel execution (calculation of stable states and execution of actions)
- Controlling safe access to data - stable and changing property states, dynamic data privacy
- Sequences and Cycles - shortcuts for transition tables? Transitions are limited to be within a list or a cycle of states.
- Executing a subexpression on the default state
- Persistence (via persistd)
- Property type and state enforcement via STRICT
- namespace scopes
- Macros
- State validation during handlers
- Syncronisation of read via shared clock

1.3.1 Potential other features

- Method extensions
- WAITing for a machine to enter a state currently only checks that the state name has changed. It should also wait for the enter_function to have been processed. A complex solution to this problems is discussed later; utilising CHANGING STATE operators but perhaps there is a simpler method also.

1.4 Bugs

- Passing SELF to an internal object
- Stable states that use timers of other machines (retest)
- State changes during stable state evaluation (details?)

1.5 Communication

- Communication between instances of clockwork drivers
- Global configuration (eg timer range in usec, msec, sec)
- Formal/scrict interfaces
- sending data with a message
- holding messages that arrive outside the state where they can be used
- limiting execution of commands apart from in certain states

1.6 Modules

- PID controls - ramps, constant speed
- Counter inputs
- Frequency inputs (calculated or read from advanced devices)
- Low level module initialisation
- Bus change events
- CAN objects for paramaterisation

1.7 Time

- Delayed messaging (sending messages after a delay and cancelling..)
- Wall time (long period timers)
- Polling - repeatedly executing a statement
- Retriggering EXECUTE / TAG actions

1.8 Macros

Macros provide a way to restructure the source into a form that improved readability. The macro facilty is simply a string replacement with the ability to substitute variables within the string as it is inserted.

1.9 Dynamic instantiation

Since machines are defined statically there is no way to create a new machine or destroy one that is not needed. The language uses properties to store values and although machines can be named, there is no way to make a property that refers to a machine although it is possible to have a reference to a machine. It is not even possible to save the state of a machine and restore that state later. [this is changing (see LIST), TBD]

2 Feature discussion

Clockwork is currently based around the notion of machines, states and properties. Machines cannot carry data per-se but since the property system provides for simple key-value objects, the effect can be simulated at a basic level. Machines also need the notion of a working-set of data that is being processed or held. Traditionally data is grouped into 'structure's or 'object's and these terms and their common usage seems fine. More generally, a 'working set', if we chose to use the term, would refer to a collection of various values, structures and objects.

2.1 Value system

The way property values are stored in machines desparately needs an overhaul; the current system is build on a C type structure and uses runtime type comparions. A new version will use C++ virtual methods that bind functions at compile time.

2.1.1 Properties

Properties of a machine can be declared using the OPTION keyword:
[data-type] OPTION (property-name [value]) | (property-name [',' property-name ...]_
the word 'PROPERTY' can be used as an alternative to 'OPTION'
If a value is not given to a property, a value must be provided at instantiation

2.1.2 Floating point values

Within clockwork, floating point values are used as in the following examples:

```
OPTION x 1.0;
OPTION y 1.;
OPTION z 1.3E3;
```

Usage

```
OPTION a 1;
OPTION x 1.3;
a := a + x; /* a converted to float with value 2.3 */
a := INT(a+x); /* INT is an alias for INTEGER */
a := a + x AS INTEGER;
a := (a + x) AS INTEGER;
OPTION x; /* (new) no default value, must be given on instantiation */
```

Equality tests for floating point values can given a tolerance:
x == y WITHIN 0.001;

2.1.3 Engineering values

We could explore the ability to refer to values with units, to help readability and clarify the calculations being performed. The IN keyword can be used to express values in other units.

```
x := 1 hr;
y := x IN min;
```

There are a small set of predefined endineering values but this set can be extended within user configurations. TBD work on this syntax

```
hr EXPRESSED IN min USES { min / 60 }
min EXPRESSED IN hr USES { hr * 60 }
```

Probably the term 'EXPRESSED' could be omitted here.

2.1.4 Type translation

Values are typed internally and can be forcibly converted from one type to another via the AS operator:

```
a := b AS INTEGER;
b := a AS FLOAT
```

Numeric values can be combined in expressions without specific type casting and the rules for determining resultant types are:

- string expressions cause numeric values to be converted to strings in the result
- when a floating point value is used in an subexpression, the subexpression type is converted to floating point
- floating point values will be displayed in fixed-point notation from clockwork (hard-coded at 3 decimal places at present

2.2 State Rules

The primary state rule is the 'WHEN' clause. Expressions linked to the WHEN clause define the rule for entering the state.

2.3 Predicates

Predicates are expressions that return boolean values. Common operators are:

- IS (state test)
- numeric comparison operators
- consider also adding ENTERS or LEAVES tests/triggers

2.4 Sets and relationships

We need to decide whether bits that are allocated to symbols should be allocated in least-significant or most-significant order. Perhaps a mapping function should be used (like a C union), with this in mind, defining three symbols, x,y and z to a block of bits would map bit 2 to x, bit 1 to y and bit 0 (least significant) to z. This gives a nice mapping for those who visualise the bits in a block and want to map the symbols over the block.

```
cutter1 Cutter;
cutter2 Cutter;
B BITSET [a,b,c,d]; # defines symbols a..d for bits 0..3
C BITSET [r:8,g:8,b:8]; # defines symbols r,g and b for bits 0..23 of C
INCLUDE a,d IN B
x := B AS INTEGER;
cutters SET[cutter1, cutter2];
s INCLUDES? [a,b,c];
SELECT state_name/value FROM set_name;
```

Sets could be implemented as a machine where each combination of members is a distinct state there doesn't seem to be much value in this. It may be useful to use three states: empty, nonempty and full.

Statements involving sets may be classified by whether they refer to the set itself, for example 'INCLUDE 3 IN s', or the members.

Sets can be used in expressions in the following ways

- ANY IN set_name ARE | IS state_name # returns true if any of the items in the set are in the state given
- ANY PROPERTY property IN set_name ARE | IS value
- ALL IN set_name ARE | IS state_name/value
- ALL PROPERTY property_name IN set_name ARE | IS state_name/value
- COUNT state_name FROM set_name # counts the number of items in the set that are in the named state
- COUNT WHERE expression FROM set_name # counts the number of items in the set where the expression returns true
- BITSET FROM set_name state_name # returns a bitset that represents the an on/off value for the given state name. Further examples use variable values..
- COPY n FROM set_name # picks n elements at random from a set
- TAKE n FROM set_name # takes n elements at random from a set

2.4.1 Special Sets

ALL [MACHINES]

ENABLED [MACHINES]

CHANNELS a list of all channels

2.4.2 Operators

CLASS

2.4.3 Enumeration

There should be a way to trigger events for each item in a set:

```
EACH item IN set DO command
```

at present, a message can be sent to all members of a set by sending the message to the set but there is no simple way to know when the message has been handled by all machines.

2.4.4 Events

Sets should send messages on state change between empty, nonempty and full. It may be useful to send events as each item enters/leaves but at this point that seems excessive and of little value.

2.4.5 Dynamic values

Some expressions include a calculation that needs to be reevaluated each time the expression is evaluated, these are calculations, for example of the size of a list, the class of a machine, the number of items on a list that are in a given state. These 'dynamic values' are:

- ANY IN
- ALL IN
- COUNT .. FROM ..
- SIZE OF
- INCLUDES
- ITEM AT
- BITSET FROM
- ENABLED
- DISABLED
- EXISTS
- CLASS OF
- SUM OF list [not implemented yet]
- VALUE OF symbol [when is this needed]
- INVALID statename

2.4.6 Automatic properties

Properties can be defined to be equivalent to the result of a calculation by using the LET command:

```
LET property_name BE expression;
```

When defind this way, the expression will be evaluated when necessary, for example when one of its members changes. The guarantee applies (at least initially) only for state calculations and upon entry to a command or state-change method. Automatic properties are private.

2.4.7 Conditional Assignment

The conditional assignment operator provides a way to make an assignment based on evaluation of conditions. A synopsis is:

```
property =: expression1 WHEN condition1 [, expression2 WHEN condition2] ... [ expressionN OTHERWISE]
```

the first assignments is made based on the first condition to evaluate to true when evaluated in the order written. This is equivalent to:

```
IF condition1 property := expression1
ELSIF condition2 property := expression2
...
[ELSE property := expressionN]
```

2.4.8 Invariants

Methods can contain declarations of invariant conditions. If an invariant is evaluated as false the program cannot continue to run and will enter its failsafe mode

2.4.9 Strict

When applied to a machine, STRICT causes the machine to:

- refuse to change the type of a property once set
- mark its properties as STRICT
- cause a system FAIL if a machine is asked to transition to an unknown state
- fail to start if a STRICT property of an incorrect type is used on instantiation

STRICT can be included outside of a machine in which case all machines in the current file are regarded as strict.

Quoted strings and numeric values are regarded as 'STRICT' properties.tea

2.4.10 Failsafe Mode

A program may be configured with a failsafe mode that includes a set of states and properties that can be activated whenever the program detects a non-recoverable error. In the case of EtherCAT there needs to be a way to move to SAFEOP. Failsafe mode can be activated by:

- throwing an exception that nothing catches,
- causing an invariant to evaluate to false
- executing a FAIL statement.
- fail-safe mode can be entered by executing FAIL SAFE. This mode will set EtherCAT into SAFEOP mode

2.4.11 Structs

Structs are a way to structure or group property names that can be used to initialise machines or used to change the value of several properties at the same time.

Initialising a machine will look something like:

```
Sample MACHINE WITH (a:1, b:''test'') { ... }
```

or

```
my_properties STRUCT {
    OPTION a 1;
    OPTION b ``test'';
    OPTION c REFER TO a.b; ## the when update is used the target machine will read its c value from its
}
Sample MACHINE WITH my_properties { ... }
```

Property sets can be updated from a database:

```
COLLECT record FROM data_source INTO my_properties ON ENDOFDATA done;
UPDATE my_machine WITH my_properties [EXCLUDING name_list];
UPDATE my_machine WITH [name_list FROM] my_properties
```

When a Property Set is being applied it is the changes are made to an invisible data structure linked to the property set. Once all of the invisible data values are updated the values are applied into the machine and a property_change event is triggered.

A normal property change that is executed while a property set is being update will be handled as normal. If that change updates a property that is changing within the Property Set the changed value will be lost on completion of the change. The effect is that the property assignments might occur out of order.

Property Set changes may be performed against shadowed data and the update will not occur until all properties are synchronised at both ends of the channel.

Partial updates from Property Sets should also be possible

```
COPY PROPERTIES a,b,c FROM machine_a TO machine_b;
```

See also comments about structured data, below.

2.4.12 Data Sources

Data sources are objects that provide data, either on a polled or automatic basis.

```
my_data DATASOURCE {
    COMMANDS {
        CONNECT, # connect to the external source
        DISCONNECT, # disconnect from the external source
        NEXT # attempt to get a record from the external source and update linked properties
    }
    STATES {
      idle,
      connected,
      error,
      invalid,
      ready
    }
    PROVIDES (custid, custname)
    USING SQL { select id as custid,name as custname from customer; }
    # or FROM JSON { [{''custid'': 1, ''custname'': ''fred''}] }
}
```

Data forms can be SQL, XML or JSON. Data is explicitly copied into a clockwork object via a 'COLLECT' statement:

```
COLLECT [NEXT] my_properties FROM data_source USING my_iterator; #
```

Exceptions possible from COLLECT are ENDOFDATA or ERROR.

There is an implication that data has multiple records and that iterators on that data can step through it.

2.4.13 Records

Records provide the data fields that map to database records.

Given a data source and customer record.

```
sample_db DATASOURCE {
  OPTION database ''sample.db'';
}
Customer RECORD sample_db {
  TYPE customer;
  OPTION name;
  OPTION age;
}
```

Records have states:

empty no fields have been set

new some fields have been set but the record hasn't been collected (i.e., doesn't have a link to a database record)

ready fields have been collected and no changes have been made to the db record or the cw record

modified field hav been changed since the record was collected

Records will always have an 'id' field that is a unique identified in the database, they can be bound to a clockwork object; this prepares a mapping from the 'current' row in the data source to the properties of a clockwork object and enables the use of a state indicator in the database for a clockwork object state.

```
BIND my_data TO my_machine WITH STATE [PROPERTY/FIELD] status;
```

Thus, my_machine can have states determined by the status field within my_data. Also my_machine has properties sharing the fields in my_data.

example 1: get details for a customer

```
fred MACHINE ds {
  cust Customer ds; # cust is invalid (could be 'empty')
  done WHEN cust IS ready;
  waiting default;
  COMMAND a {
    cust.name := ``fred''; # cust becomes new (could be invalid since not all fields are set)
    COLLECT cust FROM ds WHERE name = 'fred';
    # cust becomes invalid and eventually becomes ready
    # request: { type: customer, action: request, keys: { name: 'fred' }, fields: [name, age ] }
    # or request: { type: customer, action: request, keys: { name: 'fred' } }   # (all fields)
  }
  ENTER done { LOG ``fred is `` + cust.age; }
}
```

example 2: get all customers

```
all MACHINE ds {
  curr Customer ds;
  cust_i ITERATOR(order: ``name desc''); # iterators sequencing of data
  done WHEN cust_i IS done;
  ready WHEN curr IS ready;
  waiting WHEN cust_i IS busy;
  ENTER INIT {
    COLLECT curr FROM ds USING cust_i; # cust_i becomes busy
    # request: { type: customer, action: request, fields: [name, age ], order: [{name: desc}] }
  }
  ENTER ready {
    LOG ``customer: `` + curr.name + ``, `` + curr.age;
    SEND next TO cust_i; # cust_i becomes busy
  }
}
```

example 3: create a customer

```
new_cust MACHINE ds, cust {
  COMMAND save REQUIRES cust IS invalid OR cust IS new { STORE cust TO ds; }
  CATCH InvalidRecord {
    LOG ``customer could not be saved''
  }
}
#example 4: remove a customer
del_cust MACHINE ds, cust {
  COMMAND delete { REMOVE cust FROM ds; }
```

2.5 Functions

Actions are executed when a message is received by a machine. These actions may return a result that can be used in a property assignment.

Steps within a function are executed until a step cannot be resolved immediately, for example, when a SET statement is used to set the state of another machine. The SET does not complete until the other machine has changed state. In the current implementation, the other machine may have only started to invoke its ENTER function (TBD).

Steps may be artifically stopped by using the YIELD statement and in the future functions may be interrupted because of the use of a regular I/O poll.

It should be assumed that there is no guarantee that non-blocking steps of a method are always executed at once but this happens to be a property of the current implementation.

A function may be invoked by a CALL statement in which case the calling machine blocks until the function is complete. On completion, a result property 'RESULT' will contain the result of the function. RESULT is actually a REFERENCE (see section 2.6) so it can return multiple properties and a reference to a machine. For example:

```
CALL method ON machine;
property := RESULT.val;
SEND go TO RESULT.ITEM;
```

TBD add details about parameters.

2.6 References

The reference type REFERENCE is provided to provide a way to make a reference to another object. It has operations to assign and to clear the reference and has two states: EMPTY and ASSIGNED. Once an object is assigned to a reference, the object can be obtained using reference.ITEM. Referring to the ITEM on an empty reference produces undefined behaviour.

The implementation of a reference is similar to

```
REFERENCE MACHINE {
    ITEM ANY_MACHINE;
    ASSIGNED WHEN ITEM != NULL;
    EMPTY DEFAULT;

    COMMAND clear { } # clear the current ITEM
    COMMAND assign WITH new_item {  } # clear the current item then set it to new_item
}
```

Calling clear on the reference will unassign the object that was previously assigned and calling assign will perform an assignment.

Note: Initially, the language does not provide a way to return an object from a function.

A reference can be bound to an internal object:

```
Sample MACHINE customers {
  current REFERENCE;
  wf MyWorkflow; # a user defined machine
  ENTER INIT { BIND wf TO current; }
...
}
```

When a machine is bound to a reference, the fields in the referenced item appear within the bound machine. If the reference is not assigned, it cannot know what fields are to be provided to the bound machine and thus no fields are bound. To simplify handling of this case, a bound machine returns the special value NULL to all property requests except those that are defined within the machine. TBD, should the state of the machine be a set value when it is bound to an unassigned reference?

2.6.1 Properties that refer to other properties

This feature is only partially implemented

```
PropertyReferenceExample MACHINE {

    # The VALUE property is linked to the 'answer' property of
    # the local machine 'calculator'
    OPTION VALUE REFER TO calculator.answer;
    LOCAL calculator;

}
```

2.7 Iterator

An ITERATOR is like a REFERENCE except that it is linked to an item within a list. Iterators provide additional methods, 'prev' and 'next'. When an ITERATOR points to a member of its list, it is 'ASSIGNED' if it is moved before the first value or after the last value, it is 'EMPTY'.

```
iter ITERATOR list; # automatically assigns FIRST OF LIST to iter
working WHEN iter IS ASSIGNED;
idle DEFAULT;
ENTER working { LOG iter.ITEM.NAME; SEND next TO iter; }
ENTER idle { LOG ‘‘------’’; RESET iter; }
```

The FIND statement can be used to search a list for a match and return an iterator at the found position.

Similarly, the SEEK command can be used to obtain an iterator that points to a specific index within a list.

ITERATORs may be assigned to REFERENCEs so the following code is ok except that the reference has no ability to move through the list.

```
ref REFERENCE;
iter ITERATOR;
a OBJECT(val:1); b OBJECT(val:3);
list LIST a, b;
COMMAND example {

    x := FIND ITEM.val == 3 IN list;
    ASSIGN x TO iter; IF iter ASSIGNED { SEND prev TO iter; }
    ASSIGN x TO ref; # ref still works as a REFERENCE

}
```

2.8 Streams

A Stream is an ordered collection of data that can be grouped into fixed size units (bits, bytes, chars etc); it has a current value and position. the position can be moved forward or backward and the value can be read and changed. Streams may have a beginning and end and has a state of closed, valid or invalid. In the closed and invalid state, the current value cannot be read; if the stream is moved past the end or before its beginning, it becomes invalid. A Reader is a special case of a Stream in which the data cannot be changed, similarly, a Writer is a Stream that can only be written to.

```
s FILESTREAM ‘‘example.dat’’;
r READER s(block_size:1); # read one char at a time
demo Demo r;
Demo MACHINE r {
    done WHEN SELF IS done OR r IS invalid; # terminating condition
    start WHEN r IS closed;
    one WHEN r.curr == ’1’;
    zero WHEN r.curr == ’0’;
    other DEFAULT;
    ENTER done { SEND close TO r }
    ENTER start { SEND open TO r }
    ENTER one { LOG ‘‘1’’; SEND forward TO r }
    ENTER zero { LOG ‘‘0’’; SEND forward TO r }
    ENTER other { SEND forward TO r }
    COMMAND run WITHIN done { SET SELF TO start; }
}
```

FILESTREAM options

- mode: (read, write, readwrite, append)

READER options

- block_size: (positive integer)
- block_terminated: (character)

2.9 Data

Machines currently have an unstructured set of properties as the only method of handling data. In the short term this won't change but we are starting to add some tools to help manupulate data. For example

```
COPY PROPERTIES FROM machine1 TO machine2
COPY PROPERTIES property_name,... FROM machine1 TO machine2
```

Note that this statement does not copy the reserved properties NAME and STATE, other pseudo-properties like TIMER or properties marked as LOCAL.

It is convenient to be able to save and restore the state of an entire list of objects. One mechanism to support this is the bit functions; a list of objects can be packed into a bitmap, represented in the language by a property value. For example, a LIST may contain a list of FLAG machines, each of which have a state of on or off. This list can be mapped to a an array of bits where each bit records the state of a machine. In theory, a list can be of arbitrary length and so can a BITMAP, in practice, we expect that many algorithms only need a small list and so such a list may be able to be packed into a single integer property value. BITSET values are packed structures that can pack lists with a limited number of items (currently 32) [and possibly with the advantage that the items of a bitset are still addressible].

Further, a BITSET can be generated from properties as well as states in either case, a notation is required that provides a way to describe the way that bitsets are packed.

The state of list entries can be converted to a bitset and back using statements like the following

```
my_flags := BITSET FROM ENTRIES OF list WITH STATES off,on
SET ENTRIES OF list FROM BITSET my_flags WITH STATES [on=1,off=0]
```

Equivalent forms exist that can map a set or properties rather than states

```
val := BITSET FROM ENTRIES OF list WITH PROPERTY a;
SET PROPERTIES OF ENTRIES OF list FROM BITSET val WITH PROPERTY [a:1];
```

Both the state and property forms of these statements can be abbreviated by removal of the 'ENTRIES OF' clauses:

```
my_flags := BITSET FROM list WITH STATES off,on
SET list FROM BITSET my_flags WITH STATES [on=1,off=0]
```

In both cases, the WITH STATES clauses can also be removed if the specification is using a list of flags or a list of integers:

```
my_flags := BITSET FROM list;
SET list FROM BITSET my_flags;
```

2.9.1 Bit packing notation

The programmer can control how states or properties are mapped into a bitmap; a subset of states may be used and multiple bits per state or property may also be used. The notation is

```
specification = specifier | '[' specifier [ ',' specifier ] ']'
specifier = name [ '=' value ] [ ':' size ]
```

The size indicate sets the number of bits to be used for each value, if it is not given, the size will be the smallest number of bits needs to hold the largest value in the specification. The name is the name of the state or the name of the property, depending on the statement in which the specification is being used. The value is a non-negative integer (0,1,2,..) giving the bit pattern to be used to represent each state. Note that in this specification, there is currently no way to nominate whether a big-endian or little endian packing of the values.

For example,

on=1,off=0 indicates that state or property 'on' will be represented by one and state or property 'off' will be represented by zero.

off,on has the same effect; since no value is given, it defaults to 0, 1 etc and all states or properties not represented in the list map to zero

on=1:1,off=0:1 is the complete specification, avoiding the use of defaults

Often several control signals or status signals share a single 8- or 16-bit memory location. In clockwork, each of the status bits is represented by a MACHINE object and the setup is managed by use of the OVERLAPS syntax. Given a control word defined like this:

```
Control: xxxEMMMR
Bits:    76543210
```

where x bits aren't used, E is the 'Enable' control bit and R is a 'Reset' command and MMM is a 3-bit 'Mode' value, clockwork can be configured as follows:

```
# access the 1st component (an 8 bit value) of module1:
control DIGITALOUTPUT(size:8) module1, 0;
enable FLAG OVERLAPS control.VALUE(offset:4);
reset FLAG OVERLAPS control.VALUE(offset:0);
mode DIGITALOUTPUT(size:3) OVERLAPS control.VALUE(offset:1);
```

With this configuration, the program can set mode.VALUE to any 3-bit value, i.e., a value from 0 (0x0) to 7 (0b111).

2.9.2 Structured Data

TBD add details about structures, JSON and XML and transactions/atomic changes.

```
OPTION value { x:0, y:0 }
LOCK value;
value.x := 3;
value.y := 4;
RELEASE value or perhaps COMMIT value (either would release the lock)
```

Should we use the 'WITHIN' syntax:

```
WITHIN value x := 3, y:= 7;
```

2.9.3 RECORDS

RECORDs in clockwork define ways to interpret blocks of memory and group properties togetherFor example, clockwork can load an XML file describing EtherCAT hardware into a structure that includes named elements and has a contiguous block of memory that maps to hardware I/O.

Records:

- have an offset
- have a size
- can contain metadata (properties)
- can contain other records
- fully contain child records (it is an error to declare a child that extends past the bounds of the parent)

- may contain named or anonymous chunks of BIT, BYTE, WORD16, WORD32, WORD64

```
Colour RECORD {
    r BYTE;
    g BYTE;
    b BYTE;
};
Inputs RECORD (size: 8) {
    BIT[4]; # skip 4 bits
    in1 BIT;
    in2 BIT;
    in3 BIT;
    in4 BIT;
};
Outputs RECORD (size: 8) {
    out1 BIT;
    out2 BIT;
    out3 BIT;
    out4 BIT;
    BIT[4]; # skip four bits
};
```

When records include other records it is possible to have them start at the same place using the '|' operator

```
IO RECORD (size: 8) {
    Inputs | Outputs; # Inputs and Outputs start at the same location
    data BYTE[10]; # a 10 byte block following the above bits
}
```

Records may overlap the same piece of memory. The name of a field may be unique across all records mapped over a memory location. In this case, it may be reasonable to allow use of the field name without qualification by record id. When a record is loaded, a symbol table that provides direct access to the offset and size of each field is built, for example the above might be initially loaded as:

```
{IO: {Inputs: {in1:{offset:4,size:1},in2:{offset:5,size:1},...},Outputs:{out1:{offset:0,size:1},{offset:
```

This may then be converted to:

```
{IO.Inputs.in1:{offset:4,size:1},in1:{offset:4,size:1},...
```

Thus, the bit at offset 4 can be found using 'IO.Inputs.in1' or simply 'in1'. Note that IO.Inputs doesn't really need to resolve since it isn't a single field.

In the case of EtherCAT, we can use the RECORD syntax to define the structures we need.

```
Module RECORD {
    OPTION position 1;
    OPTION alias 0;
    pdo1 RECORD {
        OPTION SM 3;
        OPTION index 0x1a00;
        OPTION name ‘‘Inputs’’;
        RECORD {
            INITIAL 0; # this can also be done via a prepare statement
            RECORD { # unnamed map, this is the default interpretation for this region
              BIT[3];
              valve BIT;
              motor BIT;
              BIT[3]; # avoid warnings by labelling all 8 bits
            }
            byte BYTE; // a named map, pdo1.byte interprets the region as a single byte
        }
    }
}
```

2.9.4 Constrained Data

See LET.

CONSTRAIN x,y : y := x+ 5; # this would iteratively attempt to solve for x.
Should we add EQUIVALENT (better name?) clauses:
CONSTRAIN x,y : y := x + 5 EQUIVALENT x := y - 5;
If there is no solution for a constraint, an error is raised.

2.9.5 Persistence

Currently persistence is a hack using a property 'PERSISTENT' that causes a machine to announce statue or property changes on a persitence channel whenever a change occurs. This is to be upgraded to become a feature of the language and the following clauses will be added:

PERSISTENT means that the machine is persistent and all properties within the machine are saved in the persistent store if enabled at runtime. For backward compatibility the PERSISTENT property is still supported for this purpose.

VOLATILE means that the given property is not included in the persistent properties of a machine so these values are not written to the persistence store

PRIVATE means that the named state is not visible outside of the machine (even to sub-machines) and it is not published on any channel unless the channel includes a specific option to include private objects (persistence, samplers and debuggers may include private properties for example)

LOCAL means that the named option is not published on any channel unless the channel includes a specific option to include local properties. Local properties are still accessible to other machines.

2.10 Clocking

Sometimes it is useful to perform actions are regular intervals and for several actions to be synchronised. To provide this facility, clockwork uses the notion of a CLOCK that triggers a timer event at regular intervals.

```
CLOCK MACHINE {

    OPTION rate 1000; # milliseconds
    started FLAG;
    tick WHEN SELF IS wait && TIMER >= rate;
    wait WHEN started IS on;
    idle DEFAULT;
    COMMAND start { SET started TO on; }
    COMMAND stop { SET started TO off; }

}
```

Machines that are synchronised to a clock receive tick events:

```
Reader MACHINE input, clock {

    OPTION value 0.0;
    RECEIVE clock.tick_enter { value = input.VALUE; }

}
```

Should we add syntactic sugar:

```
read_clock CLICK(rate: 100);
Reader MACHINE input, clock {

    OPTION value 0;
    ON clock.tick { valuie := input.VALUE; }

}
```

Define an interface for CLOCK and link it to an analogue input

```
ClockInterface INTERFACE {

    STATES { idle, tick, wait }
    COMMANDS {start, stop }

}
ain ANALOGUEINPUT (clock: read_clock = SYSTEM.READ_CLOCK) {

    clock IMPLEMENTS ClockInterface;

}
```

2.11 Lists

Procedural or functional style? A functional language tends to use functions such as car and cdr. Similar to sets, it should be possible to enumerate objects and also to receive events as items are added or removed from the list.

2.12 Data hiding

Add: check PRIVATE STATE WITHIN visible_alias or perhaps just PRIVATE STATE that simply doesn't update the external view of what state the machine is in. Also consder aliasing perhaps SHOW state_name1 FOR state_name2.

2.13 Type checking

Consider adding: sample MACHINE p1 class1, p2 class2

2.14 Substates

2.15 Exceptions

Exceptions are messages that are broadcast to several receivers, based on whether they CATCH the event or not. An exception is sent by the THROW command and this command causes a handler to abort at point the exception is sent. I would like to consider the option of temporarily listening for events, including exceptions and this is likely to be done via a LISTEN command to enable listening and an IGNORE command to disable listening. By default, if a machine implements an event handler for an event it is assumed to be listening for the event.

2.16 Conditions

Conditions are not currently implemented but they are intended to operate as shortcuts or optimisations that reduce the need to have a large set of expressions on WHEN clauses. Effectively, a CONDITION is a machine that is true when the expression is true and false otherwise. Conditions are evaluated before stable state tests and are guaranteed to be valid throughout the test process.

A condition is instantiated just like a machine except that the final parameter is a boolean expression within braces. For example:

```
door_open CONDITION prox { prox IS on }
```

2.17 Error handling

Clockwork 1.0 introduces exception and error handling

```
TRY {
     statement-list
}
WHEN TIMER >= timeout {
     ABORT; # unsuccessful completion
     RETURN; # successful completion
}
CATCH message {
     LOG ‘‘Error: ‘‘ + message + ‘‘ detected’’
}
```

TBD how should this be used to enable validation when done?

In addition, certain statements add ON ERROR and ON TIMEOUT clauses that can be used separately ot together.

```
SET machine TO state ON ERROR { ... }
CALL handler ON machine ON ERROR { ... }
SET machine TO state ON TIMEOUT n { ... }
CALL handler ON machine ON TIMEOUT n { ... }
```

Errors may be generic, as above, where the one error handler will handle all errors from the statement or my be specific where the error is generated with the handler and caught in the ON ERROR. In the following example, the handler may return different application specific errors and the CALL of the handler can deal with each error with a specific block.

```
COMMAND handler { IF ... ERROR error1; ... ELSE... ERROR error2; ... }
CALL handler ON machine ON ERROR error1 { .... } ON ERROR error2 { ... }
```

If necessary a handler can THROW a message and another part of the machine can CATCH it

```
COMMAND handler { ...; THROW error_name; ... }
...
CATCH error_name { ... }
```

Rather than having a single CATCH within the machine, an error handler may be specified on the CALL:

```
CALL handler ON machine ON error1 {...} ON error2 {...}
COMMAND handler { ... THROW error1; ... }
```

Note that statements after a THROW are not executed and the handler is deemed to have failed. An ON ERROR will catch a THROWn error but it will also be caught by matching CATCHers. The special name ALL can be used with a CATCH to catch every THROWn message:

```
CATCH ALL { num_errors := num_errors + 1 }
```

2.18 State changes

We have clarified what is meant by "machine IS state" to mean that the conditions for that state are true and the machine has completed the transition to the state. Once a machine has started to change state, it enters a private "leaving" state for the previous state and then a private "entering" state for the new state before finally becoming on. Since they are private states, the machine will appear to the outside world to still be in the original state.

Two new verbs are added: LEAVING and ENTERING, these can be used to get access to the private parts of the transition.

Is a transitional state equal to a normal state with the same name?

is a private state equal to a normal state with the same name?

PRIVATE states can be used to simplify the external interface of a machine.

```
turning_on PRIVATE STATE;
on STATE;
PRIVATE STATES a,b,c;
```

To indicate that a stable state is private the state needs to be declared as a PRIVATE STATE as well as being defined in a WHEN clause in the normal way.

Consider simply setting 'transitional' on a state during the enter/leave process.

Consider adding a new test to see if a machine is currently transitioning: wait WHEN other IS CHANGING STATE;

As part of this cleanup, some work needs to be done on the current implementation of USING. At present, given

TRANSITION a TO b USING next;

will provide a command called 'next' that causes a transition from 'a' to 'b'. This will also cause an effect whereby if the machine moves from 'a' to 'b' by a direct 'SET', the next command will still be executed. A new verb is required to specifically enable this behaviour:

TRANSITION a TO b USING next; # retains the legacy behaviour

TRANSITION a TO b AFTER next; # the 'next' command causes the transition once the 'next' method completes

TRANSITION a TO b INVOKES starting; # the 'starting' handler is invoked after the machine transitions from a to b

The two clauses can be combined.

2.19 Support of parallel execution

The most direct method to support parallel execution is to identify separate chains of dependency within the machine graph. Thus, for example, changes that occur in one input may be able to be completely processed in parallel with another chain if the steps are independent of changes that occur in another input. [yuk reword]

More explanation required

2.20 Type Extensions

Type extensions provide for adding parameters, statements to the handlers within a state machine, fields and substates.

```
A MACHINE { ... };
B MACHINE EXTENDS A <additional parameters> {

    <event handler extensions>
    <additional fields>
    <state extensions>

}
```

More thought is required here, what I am planning is that clauses within B can be defined as separate or as extensions of those in A, for example INIT { ... } simply defines a handler for the INIT state in B. Whereas INIT EXTENDS A.INIT { ... } defines extra steps that are added to the handler already defined in A.

When a MACHINE is extended, all instances of the object share default properties of the extended type unless they are explicitly added for the derived machine.

2.21 Protocols

Protocols can be used to ensure that machines passed as parameters actually provide the correct facilities. A machine can conform to multiple protocols.

```
switch PROTOCOL { STATES { on, off; } COMMANDS {turnOn, turnOff} }
controller MACHINE on_off<switch> { ... }
```

In the above, the machine passed to the controller will be required to have only a state 'on' and a state 'off' and support the turnOn and turnOff commands.

If the switch is to be also allowed to have other states or commands we would describe it:

```
switch PROTOCOL { STATES { on, off, ...; } COMMANDS {turnOn, turnOff, ...} }
```

TBD

2.22 Data access

Data is often modified during a process but it is not appropriate for external monitors to see data until it is in a stable state. The intention is to add a facility that can be used to perform data calculations using internal, private instances and to only make the data available externally in certain states. See LOCAL, PRIVATE.

2.23 Date and Time functions

Note that these properties cannot be used in WHEN clauses of automatic states since they do not generate events as they change.

NOW the current time in milliseconds

DAY the number of the current day (1..31)

MONTH the number of the current month (1..12)

YR the current year (two digits)

YEAR the curren tyear (four digits)

HOUR the current hour (0..23)

MIN the current minute within the hour (0..59)

SEC the current second within the current minute (0..59)

Retriggering of subcondition; syntax:

```
state WHEN condition,
EXECUTE method WHENEVER condition,
TAG machine WHENEVER condition,
EXECUTE/DO method EVERY duration;
```

2.24 Frequency Calculation

All single bit inputs have the option of having their cycle frequency calculated. The simplest method for this is to record the number of cycles of the input state over a period of time. There is no need to perform the calculation of $\frac{cycle}{time}$ until the frequency data is actually needed. A clever algorithm will be used to ensure that the counter and time period values use sensible ranges (eg per sec, per msec etc).

Initially no attempt will be made to identify input frequency for analogue inputs.

2.25 Priorities

A machine will be able to be given a priority or a polling rate to indicate that events for this machine are to be handled more or less frequently than other machines

2.26 Cycle Time

The program must be able to be configured to use a faster or slower cycle time in order to save power on small devices and to give better performance when dealing with demanding tasks.

2.27 Input polling

Currently the machine follows a cycle that is input-process-output on a fixed clock. The new design pushes work to be done onto a processing queue and that processing queue operates until the cycle time arrives and then performs a read/update cycle to the IO.

This approach must buffer incoming data in order to prevent invalid state processing on machine with automatic states. This will be done by marking some tasks as high priority such that these tasks but be performed every cycle. Further thought is required.

2.27.1 Command Polling

A subcondition can be used to repeat an action only while a machine is in a particular state:

```
EXECUTE / DO command EVERY time_period;
```

Consider adding this with a standalone syntax also:

```
EXECUTE / DO command EVERY time_period [WITHIN state];
```

2.28 Property calculations

OPTION x expession

2.29 Dynamic machines

New machines can be constructed with the CREATE command and removed with the DESTROY command. An object cannot be destroyed if it is being used within another object (ie it has dependencies) or is passed to another object, it needs to be either removed from that object or that object must first be destroyed. Automatic reference counting is used so that when objects leave scope they are automatically destroyed.

Objects that are allocated statically can never be destroyed and reference counting is not necessary for these items (it may still be applied, however..TBD)

2.30 Linked states

States in cooperating machines can be linked to ensure that one machine passes through a gateway state before another machine may pass through a linked state.

2.31 Patterns

Access parts of a pattern via COPY pattern FROM source TO list

2.32 Extensions

New machine types can be defined by loadable modules. A loadable module is required to provide a particular interface that is used when instances of objects from that module are processed. It may be possible to provide loadable objects using a messaging or rpc interface. (TBD)

2.32.1 Defining PDOs

2.32.2 The EtherCAT extension

This extension provides an object of type EtherCAT_Entry when instantiated, this machine registers a pdo entry with ethercat, using the details provided. The extension also adds an SDO object for performing SDO functions. Note that currently this is hard-coded into the interpreter.

Using SDO within Clockwork

To add an SDO entry, add an instance of a machine with class SDOENTRY in the clockwork configuration. The parameters of an SDOENTRY are:

module the clockwork name for the EtherCAT module that the entry is in

index the index value of the entry

subindex the subindex value of the entry

size the size (in bytes) of the entry (TBD fix this and implement support for single bit SDO entries)

The object has a VALUE property that can be set and read within the code and an IOTIME that indicates the last time the entry was read from the io hardware. A 'default' property can also be set to indicate that this entry is to be initialised before the control logic is given access to the hardware.

2.33 Type Extensions

Machines should be able to be defined as extensions of existing machines. Initially these extensions will only be able to add actions and commands. Some thought will be needed as to whether actions will be executed before or after existing actions.

2.34 Introspection

Clockwork already has facilities to enable and disable parts of the executing machine. To make it easier to address groups of machines some facilities will be added tselect machines within the current instance:

```
points LIST FROM SELECT MACHINES OF TYPE POINT;
outputs LIST FROM SELECT MACHINES WITH NAME MATCHING '^O_';
```

or

```
points LIST;
SELECT MACHINES OF TYPE POINT INTO points;
```

2.35 Macros

Macros replace a tagged section of the program with a user-defined scrap of text, optionally substituting text in the process. All characters are copied literally, including line breaks, spaced and tab characters. [TBD should we instead trim leading and trailing whitespace and include a single space between entries when updating a macro definition?]

A macro has the form:

```
DEFINE name [ '(' name1 [ ',' name1 ]... ')' ] '{' text '}'
```

Within the text, name1, name2 etc are replaced with the values provided where the macro is to be expanded.

To introduce a macro use:

```
DEFINE sum(a,b) { a + b }
```

To expand the macro, use the name and parameters part of the above:

```
three := sum(1,2);
```

A macro name may include spaces and if desired, the expansion may include expansion markers '@<' and '@>' to make it clear that a macro is being used. For example:

```
DEFINE initialise the variables { x := 0; }
...
ENTER INIT { @<initialise the variables@> }
```

Note that it is perfectly fine to write the following

```
DEFINE initialise the variables { x := 0; }
...
ENTER INIT { initialise the variables }
```

Unlike other macro systems, in clockwork a macro can be reintroduced and this simply adds the extra text to the macro expansion. All macros are collected throughout the source before expansion begins so this feature can be used to reorder the program to suit the reader if desired. For example, the above example might have been:

```
DEFINE initialise the variables { x := 0; }
...
DEFINE initialise the variables { y := 0; }
...
ENTER INIT { @<initialise the variables@> }
```

In which case, the INIT method would have expanded to

```
ENTER INIT { x:=0;  y:=0; }
```

3 For consideration

- gecode for constraints (see LET)
- graphql for application state and api access