The Future of the Clockwork Programming Language

Martin Leadbeater

November 22, 2013

Abstract

This paper describes various language features that will be added to Clockwork in the future

1 Introduction

1.1 Types

- Floating point calculations
- Sets and relationships
- Lists
- Streams (also generators)
- Replace the current symbol table and Value system with a more efficient and robust implementation
- Type conversions (x AS type)
- Type extensions

1.2 Data

- Sharing properties or data
- Events with data
- Copying properties en mass

1.3 Features

- Substates
- Exceptions
- Conditions
- Processing priorities - low priority/slow scan inputs
- External function calls
- Dynamic allocation (ie creating and destroying data items)
- Dynamic configuration (creating and destroying machines)
- Modular/dynamic components
- Support of parallel execution
- Controlling safe access to data - stable and changing property states, dynamic data privacy
- Method extensions
- Sequences and Cycles - shortcuts for transition tables?
- Threaded implementation
- Executing a subexpression on the default state

1.4 Bugs

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

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

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

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. It is not even possible to save the state of a machine and restore that state later.

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.

2.2 Floating point calculations

Clockwork is very weak when manipulating floating point data. Currently all calculations are done using integer arithmetic and the type system needs to be updated to fix this.

2.3 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

2.3.1 Enumeration

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

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

2.3.2 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 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 READER "example.dat":1; # read one bit at a time
done WHEN s IS invalid; # terminating condition
start WHEN s IS closed;
one WHEN s.curr == 1;
zero WHEN s.curr == 0;
ENTER start { SEND open TO s }
ENTER one { LOG "1"; SEND forward TO s }
ENTER zero { LOG "0"; SEND forward TO s }

2.5 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
```

Note that this statement does not copy the reserved properties NAME and STATE.

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:

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

2.5.1 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

2.6 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.

2.7 Substates

2.8 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.

2.9 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.

2.10 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

2.11 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.

2.12 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.

2.13 Date and Time functions

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)

2.14 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.

2.15 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

2.16 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.

2.17 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.

2.17.1 Command Polling

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

DO command EVERY time_period;

2.18 Dynamic machine