

# Latproc Tools User Guide

January 24, 2013

## 1 Preface

Please note that this is a very early draft.

## 2 Introduction

This guide describes the tools that come with the Latproc project (<https://github.com/latproc>); the programs include:

**beckhoffd** a program to provide a zmq interface to an EtherCAT installation. *Not currently working*

**cw** a daemon that runs clockwork programs

**iod** a daemon that talks to EtherCAT to interact with io hardware using the clockwork language

**iosh** a shell to interact with iod

**modbusd** an interface between iod and modbus

**persistd** a basic persistence daemon to record state changes from iod

**zmq\_monitor** a program to monitor zmq messages published by iod

For the remainder of this guide, the above programs are split between those that run clockwork ('Language') and those that provide interfaces ('Tools').

### 2.1 Acknowledgements

The language and tools making up Latproc are built using a variety of open source tools and platforms, including;

- anet tcp wrappers (part of redis - <http://redis.io>)
- boost (<http://boost.org/>) - various c++ bits and pieces
- IgH EtherCAT Master for Linux (<http://www.etherlab.org/en/ethercat/index.php>)
- libmodbus (<http://libmodbus.org/>) - for communication with modbus/tcp terminals
- zeromq (<http://www.zeromq.org/>) - for inter-program messaging

We developed the software using open source development tools: GNU gcc, bison and flex and used Linux and MacOSX workstations.

## 3 Language

There are two language drivers at present, *cw* and *iod*. The difference between them is that *iod* includes facilities to talk to I/O hardware via the IgH EtherCAT Master for Linux and *cw* does not. We will discuss the use of *cw* first and discuss the extensions that *iod* provides afterwards.

### 3.1 Getting started

The file: README, that comes with the latproc distribution explains how to build the *cw* program:

1. pull the latproc project from git

```
git clone git://github.com/latproc/latproc.git latproc
```

2. change to the latproc directory and build the interpreter

```
cd latproc/iod
make -f Makefile.cw
```

The 'make' process should produce a file *latproc/iod/cw* that can be copied to a convenient location (eg */usr/local/bin*)

### 3.1.1 Hello World

Here is a program that will display a message on the terminal:

```
Hello MACHINE {
    ENTER INIT {
        LOG "Hello World";
        SHUTDOWN
    }
}
hello Hello;
```

The program defines a class of state machine called 'Hello' and provides instructions that the machine will execute when it enters a state called INIT.

In Clockwork, instructions can only be executed inside event handlers, so to log a simple message first requires that we define a class of state machine and then an event handler within that state machine. Every state machine enters a state called 'INIT' when it is enabled and our machine, 'Hello', thus, our LOG statement is executed when an instance of the 'Hello' class of machine is started.

The last line makes an instance of the Hello machine and calls it 'hello'. Note that case is important in Clockwork.

The above program can be executed by saving it into a file (eg hello.cw) and running cw:

```
cw hello.cw
```

The output should be displayed as:

```
----- hello: Hello World -----
```

Note that after the LOG statement, our program has a SHUTDOWN statement; Clockwork is intended to be used to monitor system states or provide ongoing control functions so it normally does not exit; the SHUTDOWN statement tells it to.

### 3.1.2 Light Sensor

Here is an example of how Clockwork can be used to define a monitor that turns a light on or off once there is no activity in a room. We start with a sensor and a switch, with the idea that when the sensor comes on there is activity so we turn the light on. When the sensor goes off, we turn off the light. For the time being, we use the builtin statemachine called 'FLAG' to simulate the sensor and the switch. A FLAG has two states, *on* and *off*.

We can generally define things in any order, so lets define our light controller first.

```
LightController MACHINE sensor, light_switch {
    active WHEN sensor IS on;
    inactive DEFAULT;
    ENTER active { SET light_switch TO on }
    ENTER inactive { SET light_switch TO off }
}
```

The definition simply says that when the sensor is on, the light controller is active and the light should be turned on. Otherwise, the light controller is inactive and the light should be turned off. The ENTER methods are executed each time the MACHINE enters a given state. Notice that we do not initialise the light, when the program starts, the LightController will determine what to do from the rules we have supplied.

The LightController needs two parameters; a sensor and a light switch. For the time being, we instantiate our Flags and our controller as follows:

```
sensor FLAG;
switch FLAG;
controller LightController sensor, switch;
```

Note that these entries can be given in any order.

## 3.2 Communicating with Clockwork servers

When a Clockwork program is being run, you can interact with it using a command interface or via a web page. The simplest method got get started with is the command interface, the web interface needs some extra configuration for the web server. Refer to section 4.1 for further information about iosh.

The latproc source includes some PHP code that provides a simple web view of the state machines in the executing program. The program requires PHP version 5.3 and has been tested with apache and with minihttpd. [further explanation of the setup to be done].

### 3.3 Source code conventions and file structure

When writing programs for Clockwork, program source can be split between any number of files within a user-nominated directory structure. Files and directories are provided to cw on the commandline and the program scans all files in the directories to build a consistent set of definitions from the fragments found within the files. There is no requirement to list the files in any particular order but it is an error if a definition is used but not provided or if a definition is provided more than once.

- Program text is freeform, where line breaks, tabs and spaces are all treated equally.
- Comments can be started with ‘#’ and continue to the end-of-line or can be started by ‘/\*’ and ended by ‘\*/’.
- Statements must be separated by semicolon (;) but the semicolon before the closing brace (‘}’) that ends a group of statements may still be given.

## 4 Tools

### 4.1 iosh

Clockwork and iod both provide support for a simple shell, called iosh via the Ømq (zeromq) network library. To connect to the clockwork server, simply run iosh:

```
$ iosh
Connecting to tcp://127.0.0.1:5555
Enter HELP; for help. Note that ';' is required at the end of each command    use exit; or ctrl-D to exit
>
```

at the prompt, enter any supported command, as follows:

**DEBUG machine on|off** start/stops debug messages for the device

**DEBUG debug\_group on|off** starts/stops debug messages for all the devices in the given group

**DISABLE machine** disables a machine; in the case of a POINT, it is turned off, other machines simply sit in the current state and do not process events or monitor states

**EC command** send a command to the ethercat tool (iod only)

**ENABLE machine\_name** enable a machine; set the machine state to its initial state and have it begin processing events and monitoring states

**GET machine\_name** display the state of the names machine

**LIST** show a list of all machines

**LIST [group\_name]** show a list of machines and their current state and properties in JSON format, optionally limit the list to the named group.

**MASTER** display the ethercat master state (iod only)

**MODBUS export** write the modbus export configuration to the export file (the file name is configured on the commandline when cw or iod is started

**MODBUS group address new\_value** simulate a modbus event to change the given element to the new value

**PROPERTY machine\_name property\_name new\_value** set the value of the given property

**QUIT** exit the program

**RESUME machine\_name** enable a machine by reentering the state it was in when it was disabled.

**SEND command** send the event, given in target\_machine\_name '.' event\_name form.

**SET machine\_name TO state\_name** attempt to set the named machine to the given state

**SLAVES** display information about the known EtherCAT slaves

**TOGGLE machine\_name** changes from the on state to off or vice-versa, only usable on machines with both an on and off state.

## 4.2 persistd

## 4.3 beekhoffd

## 4.4 modbusd

# 5 Other Features

## 5.1 Connecting other devices

Currently external devices can be connected to cw and iod by use of the EXTERNAL machine class. To define a connection:

- instantiate an EXTERNAL machine
- set parameters on that machine:

**HOST** a string with the name or ip address of the host. The sepecial host name “\*” indicates that the program should use a publisher/subscriber messaging model and not expect any replies.

**PORT** a number with the port to connect to on the remote machine

When a message is sent to the machine defined in this way, a connection is made via ØMQ and the message is sent. The connection is help open, ready for more messages.

# 6 Examples

## 6.1 Patterns

The following demonstrates how to detect a message and extract data from it. In particualr, this machine looks for a four character message beginning with ‘c’ and ending with ‘l’. When a match is detected, it will set a variable called ‘single’ to the first character of the message and the variable all to the first \$n\$ alphabetic letters.

Note that another machine might send the message using:

```
pattern_test.message := 'curl'
```

```
PatternTest MACHINE {
    OPTION message "";
    found WHEN message MATCHES 'c..l';
    not_found DEFAULT;
    ENTER found {
        single := COPY '[A-Za-z]' FROM message;
        all := COPY ALL '[A-Za-z]' FROM message;
    }
}
pattern_test PatternTest;
```

## 6.2 Dining philosopers

The following simulates several philosophers dining at a circular table. There are seven philosophers and only seven chopsticks so chopsticks must be shared as two chopsticks are required to eat. The program uses resource locking to guarantee that chopsticks are not being used by different philosplers simultaneously.

```
Chopstick MACHINE {
    OPTION tab Test;
    OPTION owner "noone";
    free STATE;
    busy STATE;
}
Philosopher MACHINE left , right {
    OPTION tab Test;
    OPTION eat_time 20;
    OPTION timer 20;
    full FLAG;
    okToStart FLAG;
    okToStop FLAG;
    finished WHEN SELF IS finishing || SELF IS finished && TIMER < timer;
```

```

    finishing WHEN left.owner == SELF.NAME && right.owner == SELF.NAME
        && TIMER >= eat_time;
    eating WHEN left.owner == SELF.NAME && right.owner == SELF.NAME,
TAG full WHEN TIMER > 10;
    starting WHEN left.owner == "noone" && right.owner == "noone";
    waiting DEFAULT;
ENTER INIT {
    eat_time := (NOW % 10) * 10;
    SET okToStart TO on;
    SET okToStop TO on;
}

ENTER starting {
    LOCK left;
    IF (left.owner == "noone") {
        LOG "got left";
        left.owner := SELF.NAME;
        LOCK right;
        IF (right.owner == "noone") {
            right.owner := SELF.NAME;
            LOG "got right";
        }
        ELSE {
            UNLOCK left;
            UNLOCK right;
        }
    }
    ELSE {
        UNLOCK left;
    }
}

ENTER finished {
    LOG "finished";
    left.owner := "noone";
    right.owner := "noone";
    UNLOCK right;
    UNLOCK left;
    timer := 10 * (TIMER + 1);
}

TRANSITION INIT TO starting REQUIRES okToStart IS on;
TRANSITION eating TO finished REQUIRES okToStop IS on; }

```

```

c01 Chopstick; c02 Chopstick; c03 Chopstick;
c04 Chopstick; c05 Chopstick; c06 Chopstick; c07 Chopstick;
phil1 Philosopher c01, c02;
phil2 Philosopher c02, c03;
phil3 Philosopher c03, c04;
phil4 Philosopher c04, c05;
phil5 Philosopher c05, c06;
phil6 Philosopher c06, c07;
phil7 Philosopher c07, c01;

```