OpenMRN hands-on introduction

Stuart W Baker, Balazs Racz

Overview

- Basic concepts of OpenMRN
- Status and features
- Hands-on session: build an IO board app
  - Building and running example applications
  - Creating an empty application
  - Setting up CAN-bus connections
  - Creating OpenLCB stack
  - Implementing the configuration definition
  - Adding producers and consumers

Positioning of OpenMRN

- Free
  - commercial-use-friendly BSD license
- Cross-platform & easily portable
  - OS abstraction layer allows shared code
  - desktops, microcomputers, MCUs
  - not the smallest MCUs ($2+, 32-bit required)
- Production-quality & highly modular
- C++ implementation of MR protocols
  - first one is OpenLCB / LCC

Ports

- Desktop OS
  - Linux, MacOS, Windows; both 32 and 64 bit
  - javascript (for running in browsers)
- Microcomputers (RPi, BeagleBone Black)
- ARM microcontrollers
  - Cortex-M3/M4 (TI Tiva, NXP LPC17xx, STM32, Freescale Kinetis)
  - Cortex-M0 (Freescale Kinetis, STM32)
- MIPS microcontrollers
  - MIPS-M4K (PIC32MX)

System features

- multi-threading, mutexes, semaphores, message queues
- fully posix-compatible driver model, select() support
- driver implementations for MCUs (CAN, Serial, USB for most ports)
- EEPROM emulation via FLASH writes
- native support for state machines
- cooperative multi-tasking scheduler including timeouts
- controlled memory usage, buffer allocation and freelists
- virtual CAN-bus
- GridConnect protocol parsing and rendering
- connecting CAN-bus via serial, USB, CAN, or TCP connections
- fully unit-tested, test coverage report generation
- compiles to 32 or 64-bit targets with GCC or LLVM (Clang) compilers

Model Railroading features

- CAN-USB, CAN-CAN bridge, CAN TCP hub
- OpenLCB
  - CAN-bus interface, alias handling, framing
  - multiple virtual node support, local loopback
  - node initialization, lookup
  - PIP and SNIP
  - events protocol, modular event handler concept
  - Datagram protocol, framing
  - Memory Configuration Protocol, CDI, xml generation

Features (cont)

- OpenLCB
    - Traction protocol (Jim’s): Train and Traction proxy
    - Bootloader using streaming protocol in 5kB flash
- DCC
    - Command station (packet stream generation)
    - DCC and Marklin-Motorola protocol
    - API for smart refresh plugin
    - RailCom cutout and feedback reception
    - POM (CV reads&writes using RailCom)
        - accessible via OpenLCB

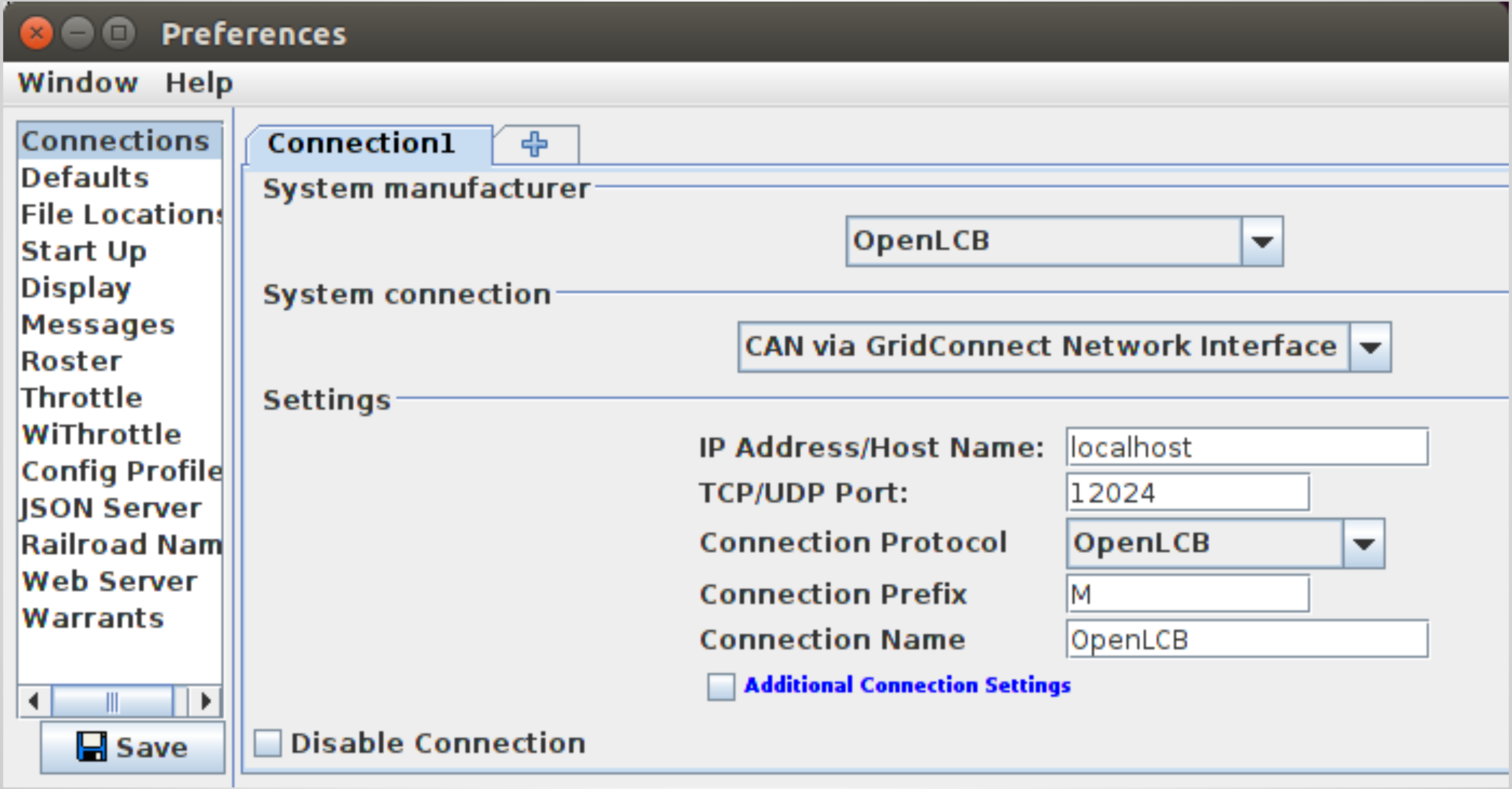
let’s get started

Build & run “hub” application

```
cd applications/hub/targets/linux.x86
make -j
./hub -p 12024
```

Connect to it with JMRI

- Start JMRI (say PanelPro)
- Click “new profile”, add name, then OK
- In the preferences dialog:
  - System manufacturer = “OpenLCB”
  - Connection = “CAN via GridConnect Network If”
  - Host Name = “localhost”
  - TCP/UDP port = 12024
- Save and allow restart


Preferences
Window
Help
Connections
Defaults
File Locations
Start Up
Display
Messages
Roster
Throttle
WiThrottle
Config Profile
JSON Server
Railroad Nam
Web Server
Warrants
Save
Connection1
System manufacturer
OpenLCB
System connection
CAN via GridConnect Network Interface
Settings
IP Address/Host Name:
localhost
TCP/UDP Port:
12024
Connection Protocol
OpenLCB
Connection Prefix
M
Connection Name
OpenLCB
Additional Connection Settings
Disable Connection

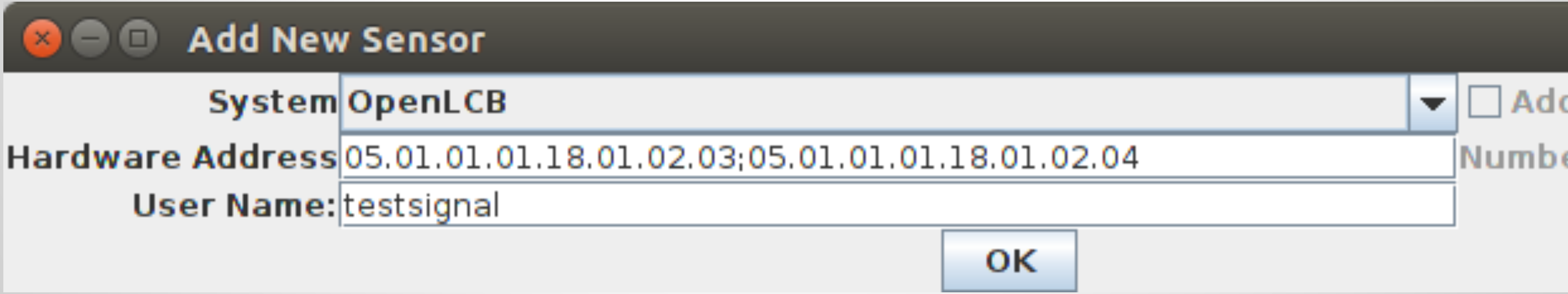
When you restart, watch the traffic on the hub:

```
$ ./hub -p 12024
Listening on port 12024, fd 3
Incoming connection from 127.0.0.1, fd 4.
:X17000847N;
:X16000847N;
:X15000847N;
:X14000847N;
:X10700847N;
:X19490847N;
```

This means JMRI connected to the bus and allocated an alias.

Generate traffic from JMRI

- Tools > Tables > Sensors
- Add… button on the bottom
- HW address (do not change now!)
  - 05.01.01.01.18.01.02.03;05.01.01.01.18.01.02.04



Test

- Hit OK, then File > Store > Configuration...
- keep clicking on the State button
  - Active/Inactive/Unknown
- watch the hub for traffic

```
:X195B4000N0501010118010204;
:X195B4000N0501010118010203;
:X195B4000N0501010118010204;
```

- alternatively: “OpenLCB > Traffic monitor”

Connect our CAN buses

We’ll create a CAN bus over the internet

`./hub -p 12024 -u 28k.ch -q 50007`

```
$ ./hub -p 12024 -u 28k.ch -q 50007
Listening on port 12024, fd 3
Connected to 28k.ch:50007. fd=4
```

Quit and restart JMRI

- Panels->Open Panels… and load the saved file

Open the sensors table, watch/click the button

Add an example node

```
cd applications/simple_client
make -j
targets/linux.x86/simple_client -p 12024 -n
  0x0501010118XX
```

make up a number for XX and tell us

or use your own node ID range

hit the sensor button in JRMI; watch the output

let’s make a node

Copy-paste application template

```
cd applications

cp -rf empty_app webinar_app

cd webinar_app/targets ; mkdir freertos.armv7m.ek-tm4c123gxl

cd freertos.armv7m.ek-tm4c123gxl

ln -s ../../../../boards/ti-ek-tm4c123gxl-launchpad/* .
```

edit webinar_app/targets/Makefile

```
SUBDIRS = \
        freertos.armv7m.ek-tm4c123gxl \

#       freertos.armv7m.ek-tm4c1294xl \
#       freertos.armv7m.lpc1768-mbed \
#       linux.x86 \

include $(OPENMRNPATH)/etc/recurse.mk
```

<- add target dir

Write a simple main.cxx

put it into webinar_app/ or

webinar_app/targets/freertos.armv7m.ek-tm4c123gxl

```
#include "os/os.h"
#include "utils/blinker.h"

int appl_main(int argc, char*argv[]) {
    resetblink(0xF280);
    while(true) {}
    return 0;
}
```

Build and run

```
cd webinar_app/targets/freertos.armv7m.ek-tm4c123gxl
make -j flash
```

Let’s add an OpenLCB stack

We’ll instantiate a C++ object ‘SimpleCanStack’

Give it the NodeID

Let it run infinite loop in appl_main().

For the moment just headless (no I/O)

```
#include "nmranet/SimpleStack.hxx"

const uint64_t NODE_ID = 0x050101011801ULL;
nmranet::SimpleCanStack stack(NODE_ID);

extern const char* const nmranet::
SNIP_DYNAMIC_FILENAME = "/dev/eeprom";

int appl_main(int argc, char*argv[]) {
    stack.print_all_packets();
    stack.loop_executor();
    return 0;
}
```

Build and run

```
cd clinic_app/targets/freertos.armv7m.ek-tm4c123gxl
make -j flash
```

then observe the output on the serial port

```
stty -F /dev/ttyACM0 115200 raw ; cat /dev/ttyACM0
```

then hit the reset button on the device

Add CAN bus connection

Instead of print_all_packets()

```
int appl_main(int argc, char*argv[]) {
    // Add one of these three
    stack.add_can_port_select("/dev/can0");
    stack.add_gridconnect_port("/dev/serUSB0");
    stack.add_gridconnect_port("/dev/ser0");

    stack.loop_executor();
    return 0;
}
```

Build and run

```
cd webinar_app/targets/freertos.armv7m.ek-tm4c123gxl
make -j flash
```

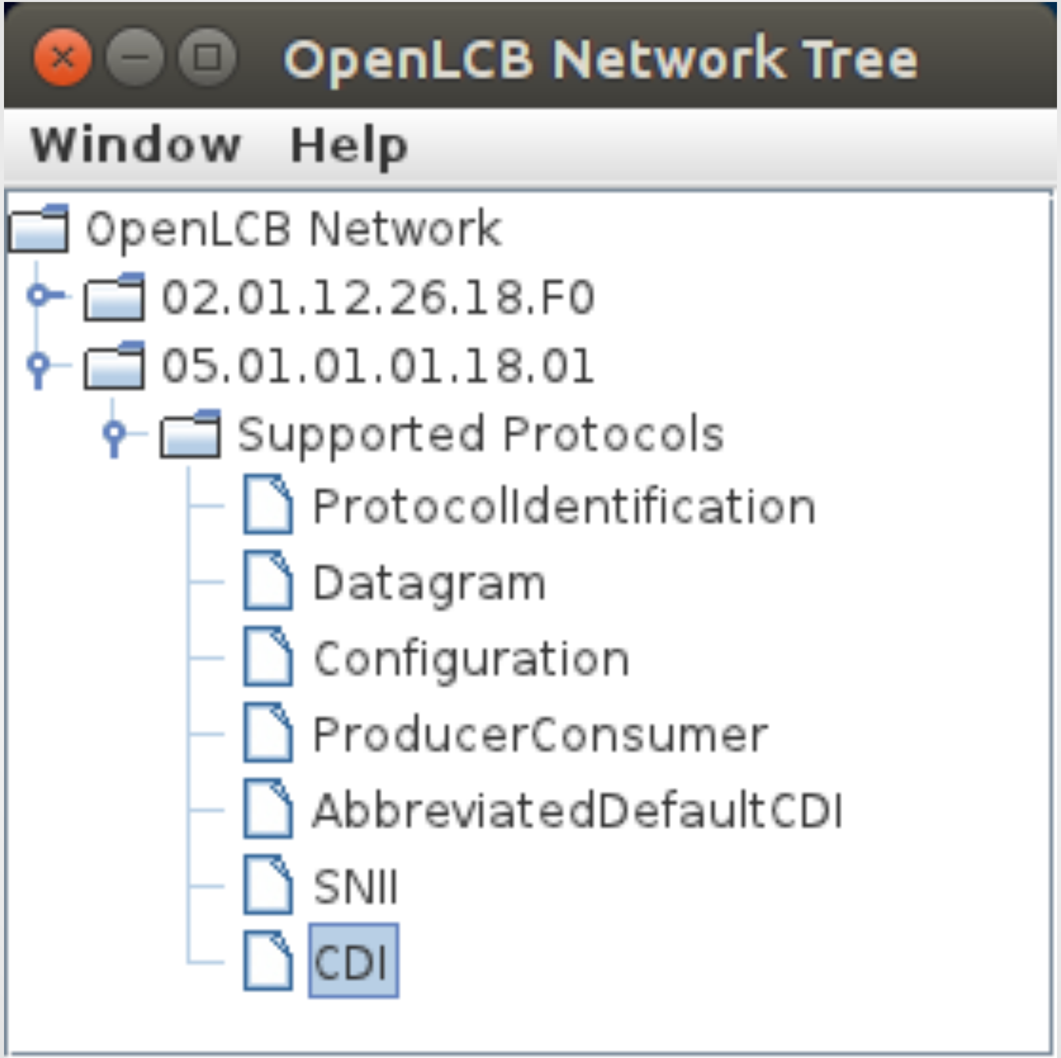
change the hub to connect to the device

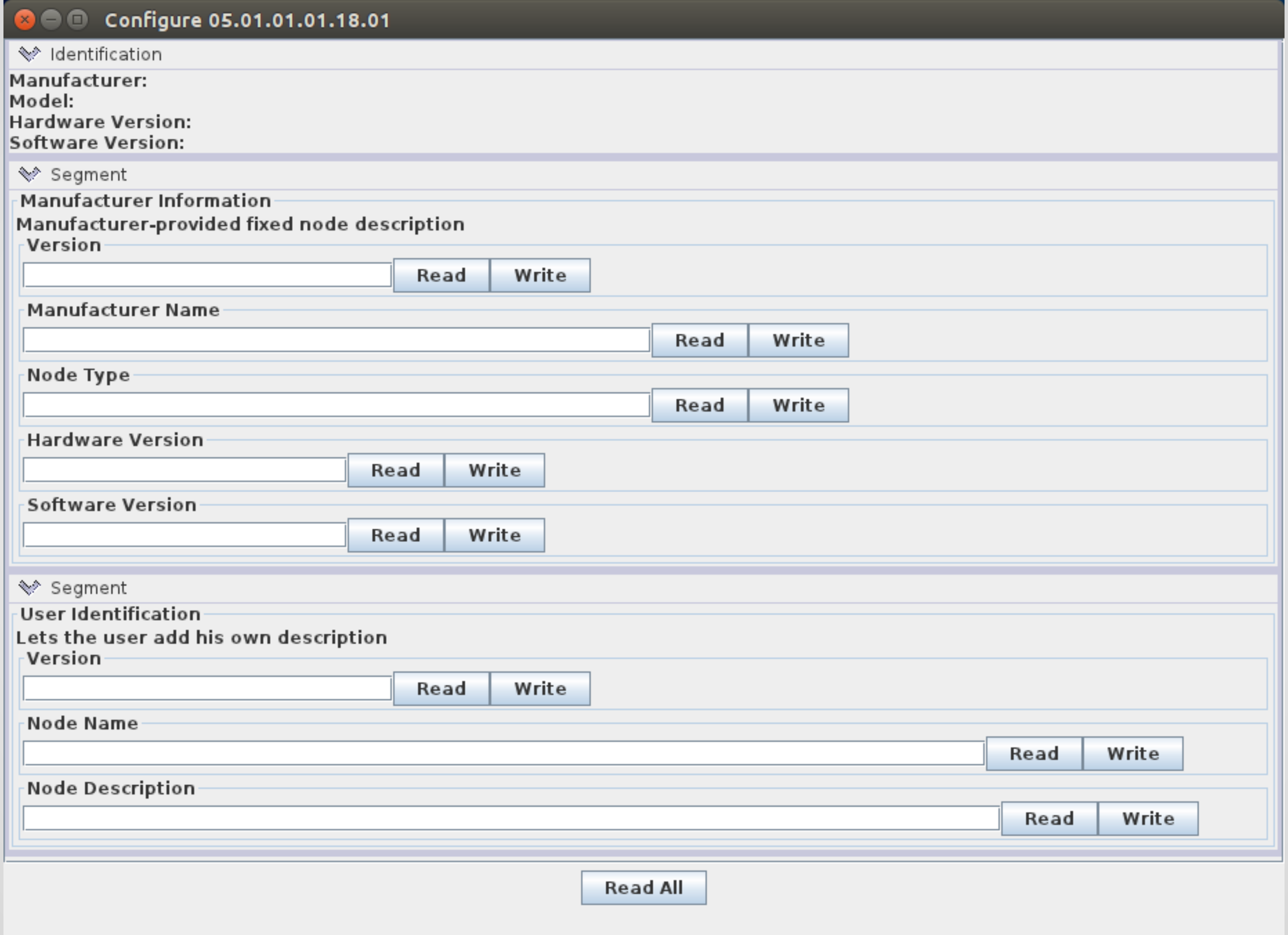
```
cd applications/hub/targets/linux.x86
./hub -p 12024 -d /dev/ttyACM0
```

then restart JMRI (since it lost the hub)

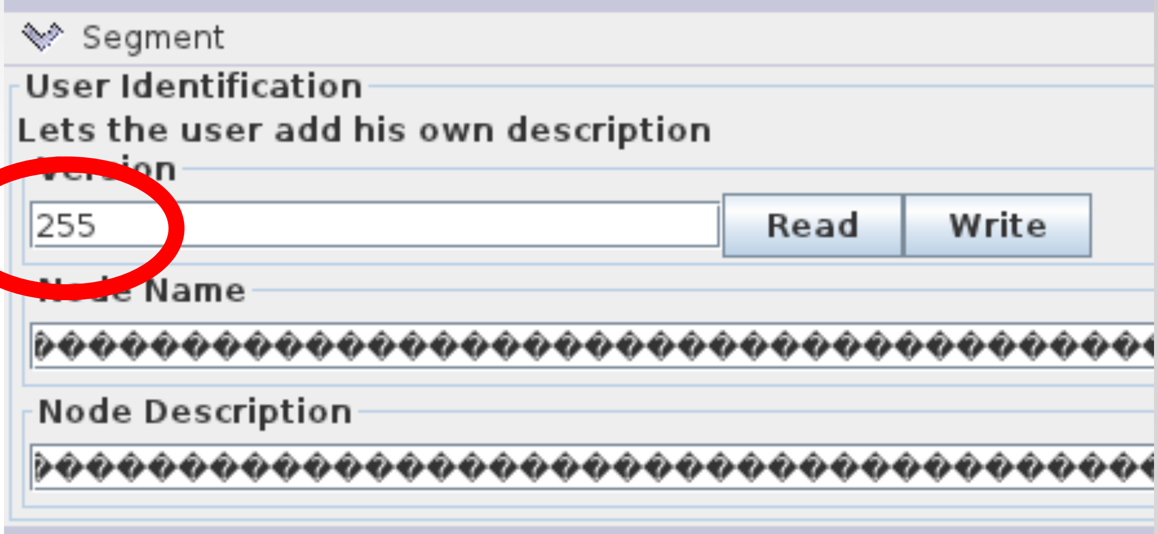
Let’s go configure

- In JMRI, go OpenLCB > Configure Nodes
- will crash your node (“. . . - - -”)
- hit reset
- expand your node ID, “supported protocols”
- click CDI


OpenLCB Network Tree
Window
Help
OpenLCB Network
02.01.12.26.18.F0
05.01.01.01.18.01
Supported Protocols
ProtocolIdentification
Datagram
Configuration
ProducerConsumer
AbbreviatedDefaultCDI
SNII
CDI


Configure 05.01.01.01.18.01
Identification
Manufacturer:
Model:
Hardware Version:
Software Version:
Segment
Manufacturer Information
Manufacturer-provided fixed node description
Version
Read
Write
Manufacturer Name
Read
Write
Node Type
Read
Write
Hardware Version
Read
Write
Software Version
Read
Write
Segment
User Identification
Lets the user add his own description
Version
Read
Write
Node Name
Read
Write
Node Description
Read
Write
Read All

- Hit “Read All”
- EEPROM is empty (>crash)
- set version = 2
- hit Write



We didn’t factory-initialize the EEPROM.

Restart JMRI, go to configure nodes -> no crash.

You may also set Node Name, Description.

but JMRI caches SNIP info until restart.

Adding a Consumer

```
[...]
extern const char *const nmranet::SNIP_DYNAMIC_FILENAME = "/dev/eeprom";

#include "BlinkerGPIO.hxx"
#include "nmranet/EventHandlerTemplates.hxx"


const uint64_t EVENT_ID = 0x0501010118010203ULL;
nmranet::GPIOBit blinker_bit(
    stack.node(), EVENT_ID, EVENT_ID + 1, BLINKER_Pin());
nmranet::BitEventConsumer consumer(&blinker_bit);

int appl_main(int argc, char *argv[])
[...]
```

Adding a Consumer

Two step process:

- dynamic parts - BitEventInterface
  - which virtual node
  - event ID for ON, and OFF
  - how to reach the hardware (GPIO bit here)
- shared part - BitEventConsumer
  - two consumers, for on and off
  - registration, callbacks, event identification & handling
- will be simpler later

Build and run

```
cd webinar_app/targets/freertos.armv7m.ek-tm4c123gxl
make -j flash
```

the hub will reconnect, no need for restart

go to JMRI, open sensor table

- hit the sensor active/inactive button
- see the LED go on and off

SNIP: Simple Node Info Protocol

- Instantiated in SimpleCanStack

```
/// General flow for simple info requests.
SimpleInfoFlow infoFlow_{&ifCan_};
/// Handles SNIP requests.
SNIPHandler snipHandler_{&ifCan_, &infoFlow_};
```

- Currently with default values for manufacturer
- and /dev/eeprom for the user part
- also exported on ACDI memory spaces

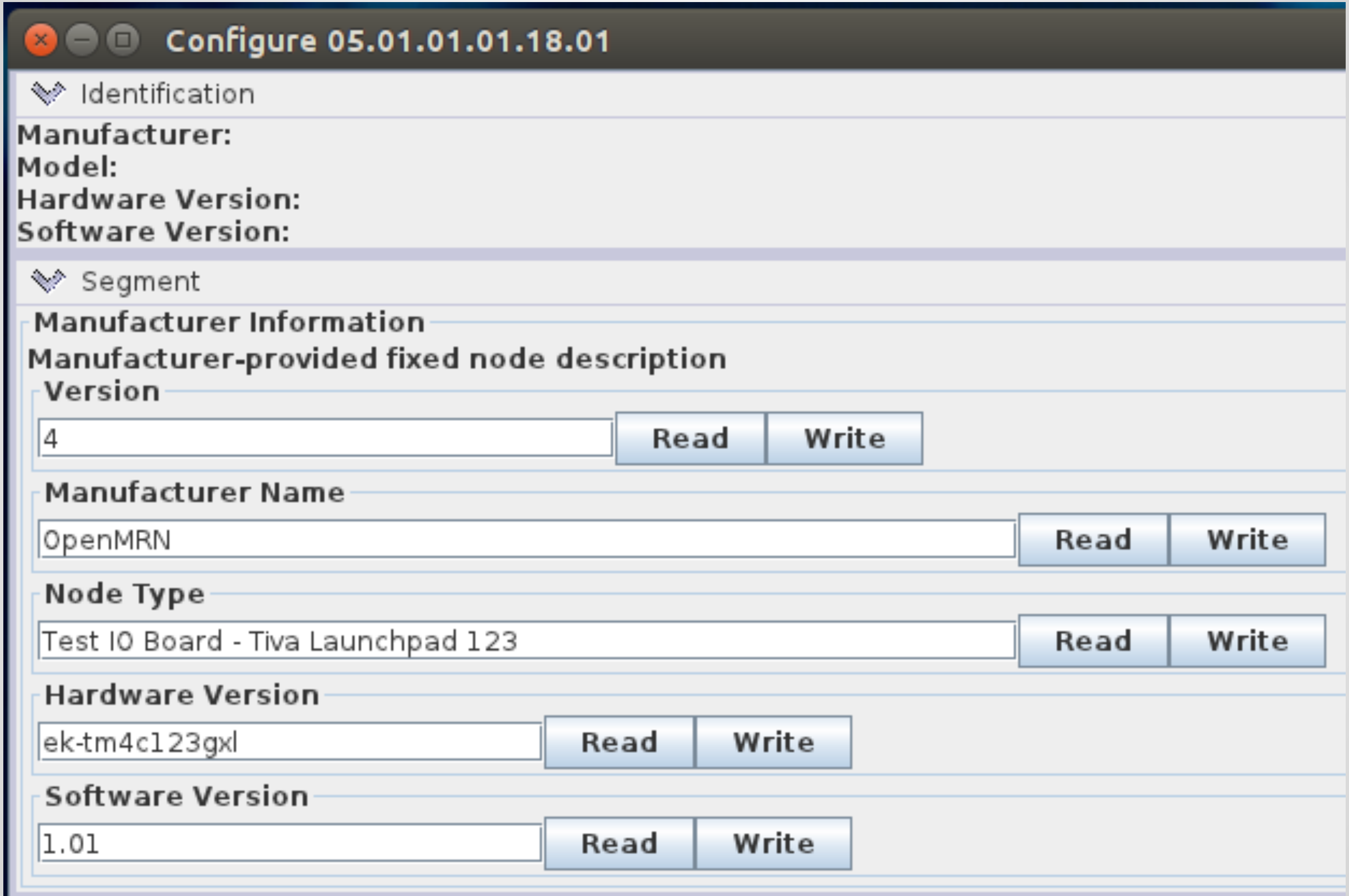
Specify manufacturer info

Create “config.hxx”

```
#include "nmranet/SimpleNodeInfo.hxx"

namespace nmranet {

extern const SimpleNodeStaticValues SNIP_STATIC_DATA =
{
    4,  "OpenMRN", "Test IO Board - Tiva Launchpad 123",
    "ek-tm4c123gxl", "1.01"};

} // namespace nmranet
```

Build and run

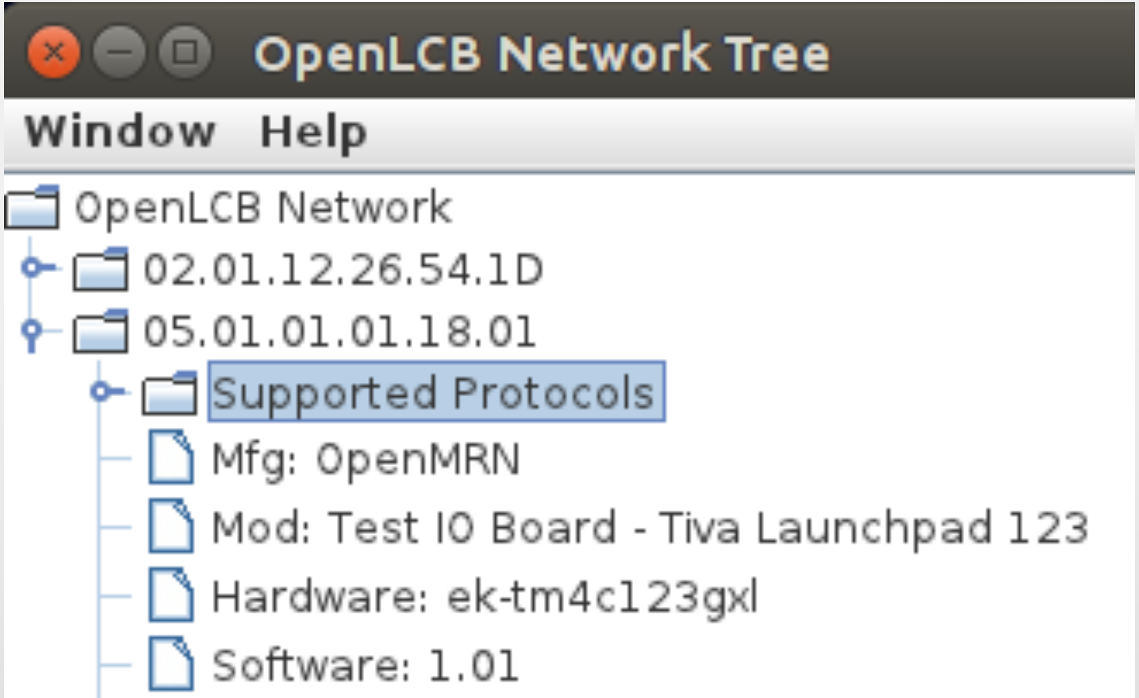
- Link into binary
  - #include “config.hxx”   <- add into main.cxx
- Build and run (make -j flash)
  - No need to restart anything
- Go to JMRI
  - open the configuration/CDI window
  - Hit “Read All”



Configure 05.01.01.01.18.01
Identification
Manufacturer:
Model:
Hardware Version:
Software Version:
Segment
Manufacturer Information
Manufacturer-provided fixed node description
Version
4
Read
Write
Manufacturer Name
OpenMRN
Read
Write
Node Type
Test IO Board - Tiva Launchpad 123
Read
Write
Hardware Version
ek-tm4c123gxl
Read
Write
Software Version
1.01
Read
Write

If you restart JMRI...

...the SNIP info will be refreshed too



Writing Configuration Definition Info

- CDI is an xml file
  - specifies the configuration options
  - flashed into your firmware, delivered with the HW
  - so far we've been using the default
  - which is at src/nmranet/DefaultCdi.cxx
  - pretty much only the ACDI example for setting user data
- We have a way to generate it
  - let's go back into config.hxx

```
#include "nmranet/ConfigRepresentation.hxx"
namespace nmranet {
[...]
  "ek-tm4c123gxl", "1.01"};

/// The main structure of the CDI. ConfigDef is the symbol we use in main.cxx
/// to refer to the configuration defined here.
CDI_GROUP(ConfigDef, MainCdi());
/// Adds the <identification> tag with the values from SNIP_STATIC_DATA above.
CDI_GROUP_ENTRY(ident, Identification);
/// Adds an <acdi> tag.
CDI_GROUP_ENTRY(acdi, Acdi);
/// Adds a segment for changing the values in the ACDI user-defined
/// space. UserInfoSegment is defined in the system header.
CDI_GROUP_ENTRY(userinfo, UserInfoSegment);
/// Closes the CDI declaration.
CDI_GROUP_END();
}  // namespace nmranet
```

Build and run

- Magic to generate the xml

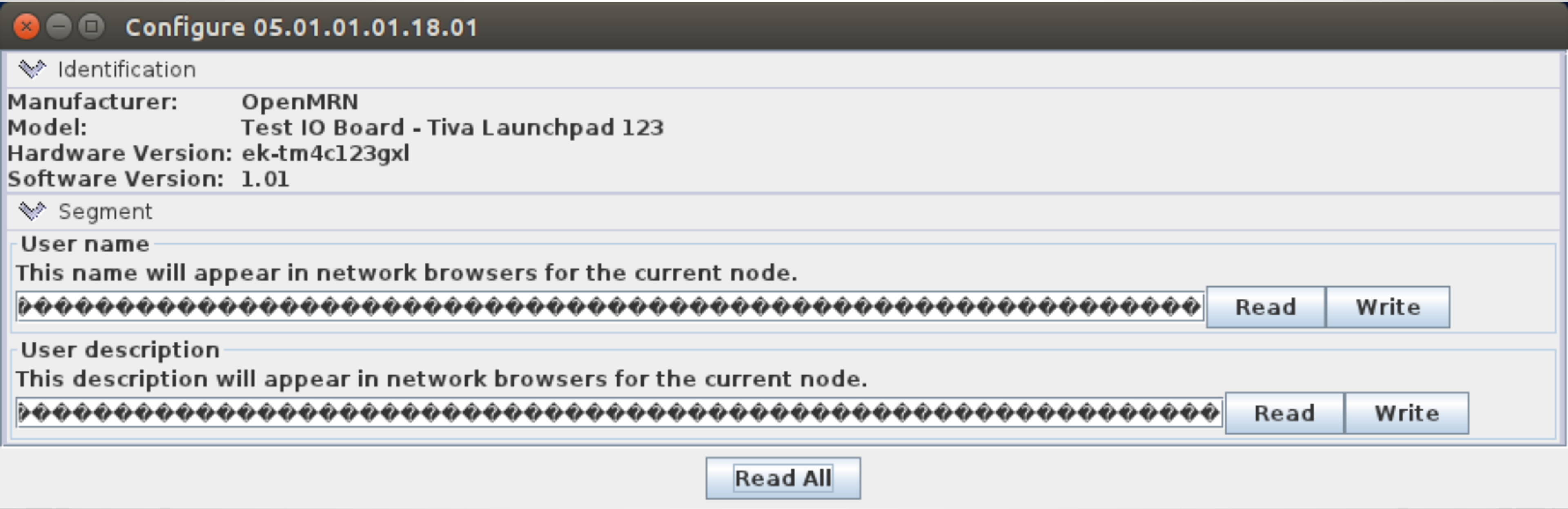
```
cd targets/freertos.armv7m.ek-tm4c123gxl
mv Makefile hardware.mk
cp ../../../io_board/targets/freertos.armv7m.ek-tm4c123gxl/Makefile .
```

- Build & run

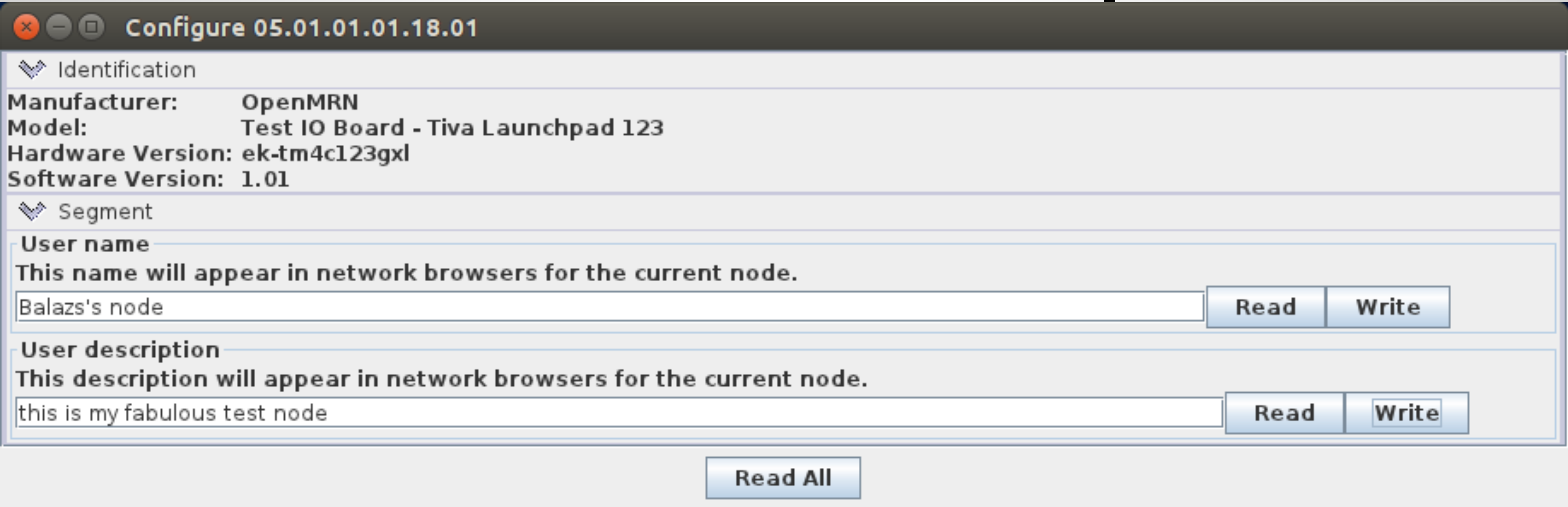
```
make -j flash
```

Test in JMRI

No restart needed, just click on “CDI” again



Set the user name & description



Don't forget to hit "Write" on each line.
Close, re-open, hit "read all" to see again.

Making events configurable

- Update config.hxx
  - We add a segment to the CDI
  - A ConsumerConfig inside that segment
  - Add the segment into the ConfigDef
- Update main.cxx
  - specify the config device filename
  - Instantiate the ConfigDef object
  - Replace GPIOBit + BitEventConsumer with a wrapper: ConfiguredConsumer

```
#include "nmranet/ConfiguredConsumer.hxx"
#include "nmranet/MemoryConfig.hxx"
[...] namespace nmranet { [...]
/// Defines the main segment in the configuration CDI. This is laid out at
/// origin 128 to give space for the ACDI user data at the beginning.
CDI_GROUP(IoBoardSegment, Segment(MemoryConfigDefs::
SPACE_CONFIG), Offset(128));
CDI_GROUP_ENTRY(blinker, ConsumerConfig, Name("Blinker LED"));
CDI_GROUP_END();
[...]
/// space. UserInfoSegment is defined in the system header.
CDI_GROUP_ENTRY(userinfo, UserInfoSegment);
CDI_GROUP_ENTRY(seg, IoBoardSegment);
/// Closes the CDI declaration.
CDI_GROUP_END();
```

Link in the new segment

```
[...]
#include "BlinkerGPIO.hxx"
nmranet::ConfigDef cfg(0);
extern const char *const nmranet::CONFIG_FILENAME =
    "/dev/eeprom";
// The size of the memory space to export over the above device.
extern const size_t nmranet::CONFIG_FILE_SIZE =
    cfg.seg().size() + cfg.seg().offset();

nmranet::ConfiguredConsumer consumer(
    stack.node(), cfg.seg().blinker(), BLINKER_Pin());

int appl_main(int argc, char *argv[])
[...]
```

```
[...]
#include "BlinkerGPIO.hxx"
nmranet::ConfigDef cfg(0);
extern const char *const nmranet::CONFIG_FILENAME =
    "/dev/eeprom";
// The size of the m                                   above device.
extern const size_                                     =
    cfg.seg().size() +

nmranet::ConfiguredConsumer consumer(
    stack.node(), cfg.seg().blinker(), BLINKER_Pin());

int appl_main(int argc, char *argv[])
[...]
```

Tells the offset in the EEPROM

Build and run

- make -j flash
    - No need to restart anything
- Go to JMRI
    - open the configuration/CDI window
    - if already open, close it, click on a different protocol, click CDI again
    - Write 05.01.01.01.18.01.02.03 and .04 to the events
    - (hit the Write buttons), then reset your node
- now hit the sensor Active/Inactive button

Adding more GPIO pins

this goes to main.cxx

```
#include "TivaGPIO.hxx"
// The first LED is driven by the blinker device from BlinkerGPIO.
// We just create an alias for symmetry.
typedef BLINKER_Pin LED_RED_Pin;
// These are GPIO output pins from TivaGPIO.hxx
GPIO_PIN(LED_GREEN, LedPin, F, 3);
GPIO_PIN(LED_BLUE, LedPin, F, 2);
```

Pins are typedefs.

Name of pin

C++ symbol=NAME_Pin

Type

Defines input vs output, drive strength etc.

HW-specific

port name, pin number

Other hardware

#include "Lpc17xx40xxGPIO.hxx"

arguments: port number (int), pin number (int)

ex. GPIO_PIN(SECOND, LedPin, 0, 22);

TODO: same for the stm32

Defining more consumers

this goes to main.cxx

```
nmranet::ConfiguredConsumer consumer_red(
    stack.node(), cfg.seg().blinker(), LED_RED_Pin());
nmranet::ConfiguredConsumer consumer_green(
    stack.node(), cfg.seg().blinker(), LED_GREEN_Pin());
nmranet::ConfiguredConsumer consumer_blue(
    stack.node(), cfg.seg().blinker(), LED_BLUE_Pin());
```

These will all turn on/off together (same config).

Build and run

- When you click the sensor in JMRI, all LEDS will go on and off at the same time.
- In the Tiva example the RGB LED will go dark to white.

Adding repeated CDI groups

this goes to config.hxx

Before

```
CDI_GROUP_ENTRY(blinker, ConsumerConfig, Name("Blinker LED"));
```

After

```
/// Declares a repeated group of a given base group and number of repeats.
typedef RepeatedGroup<ConsumerConfig, 3> AllConsumers;
CDI_GROUP_ENTRY(consumers, AllConsumers, Name("Outputs"));
```

Attach consumers

this goes to main.cxx

```
nmranet::ConfiguredConsumer consumer_red(stack.node(),
    cfg.seg().consumers().entry<0>(), LED_RED_Pin());
nmranet::ConfiguredConsumer consumer_green(stack.node(),
    cfg.seg().consumers().entry<1>(), LED_GREEN_Pin());
nmranet::ConfiguredConsumer consumer_blue(stack.node(),
    cfg.seg().consumers().entry<2>(), LED_BLUE_Pin());
```

The entry indexes will be validated against the specified repeat count.

Build and run

- Reflash and go to JMRI.
- Clicking the sensor should light the first LED.
- Your CDI has changed
  - close and reopen the CDI window -> see the repeats
  - enter new event IDs, hit Write
  - hit Reset on the node
- Add new sensors to the JMRI table
  - click them to watch your LEDs go on and off

Adding producers

- Create GPIO pin definition
  - LedPin -> GpioInputPU
- Create configuration entries
  - ConsumerConfig -> ProducerConfig
- Create producer objects
  - ConfiguredConsumer -> ConfiguredProducer
- But we need to poll the IO pins
  - There is a component for that

Input pins

Tiva tm4c123

```
GPIO_PIN(SW1, GpioInputPU, F, 4);
GPIO_PIN(SW2, GpioInputPU, F, 0);
```

LPCXpresso

```
GPIO_PIN(SW1, GpioInputPU, ???
GPIO_PIN(SW2, GpioInputPU, ???
```

Tiva tm4c129

```
GPIO_PIN(SW1, GpioInputPU, J, 0);
GPIO_PIN(SW2, GpioInputPU, J, 1);
```

stm32

```
GPIO_PIN(SW1, GpioInputPU, ???
GPIO_PIN(SW2, GpioInputPU, ???
```

Polling syntax

this goes to main.cxx

```
// Similar syntax for the producers.
nmranet::ConfiguredProducer producer_sw1(
    stack.node(), cfg.seg().producers().entry<0>(), SW1_Pin());
nmranet::ConfiguredProducer producer_sw2(
    stack.node(), cfg.seg().producers().entry<1>(), SW2_Pin());

// The producers need to be polled repeatedly for changes and to execute the
// debouncing algorithm. This line instantiates a refreshloop and adds the
// two producers to it.
nmranet::RefreshLoop loop(
    stack.node(), {producer_sw1.polling(), producer_sw2.polling()});
```

Build and run

- Reflash and go to JMRI
  - close and reopen the CDI window
- Configure producer
  - Add events to the producer pins (Write)
  - Set debouncing param to 2 or 3 (Write)
  - Reboot your node
- Add another sensor with the new eventids
- Hit the button and watch the status in the table

Fun

- Reconfigure producer
  - Set events to be the same as your LEDs
- Hit the buttons and watch the LEDs
  - this was done by OpenLCB traffic
- Now try to change the LED from JMRI
  - the polling input will change it back

Lots of room for improvement

- Strong or weak producers? Consumers?
- Better debouncing algorithms
- Tap vs press
- Additional features
  - blinking
  - dimming
  - output pulse
  - event proxies
- Manufacturer freedom -- product excellence

thank you for your attention

questions, feedback?