ORACLE

JavaOne™

ORACLE®

Java ME 8: Java that scales from Desktop to Tiny Embedded

CREATE THE FUTURE

CON6222

Terrence Barr
Senior Technologist and Principal Product Manager
Java Embedded & Internet of Things
Oracle

Sep, 2014

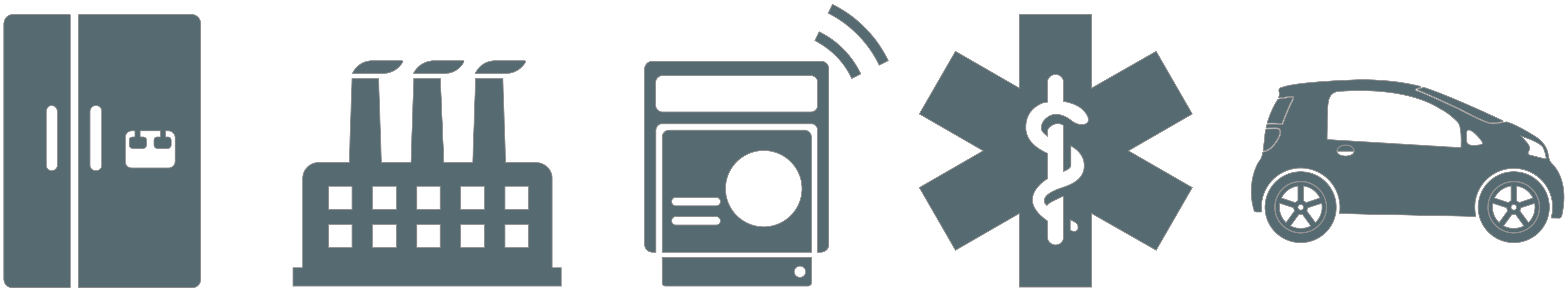
JavaOne™ ORACLE

Program Agenda

1. Why Java ME 8?
2. Building a Smart Sensor
3. Setting up and Developing
4. Where to go next & Resources

Why Java ME 8?

Java Embedded Enables New IoT Services



Home Automation

Industrial Automation

Smart Utilities

Healthcare

Automotive Telematics

Java ME 8: Top 10 Features

1. Alignment with Java SE platform
2. Designed for Embedded
3. Highly Portable and Scalable
4. Consistent Across Devices
5. Advanced Application Platform
6. Modularized Software Services
7. Multiple Client Domains (Device Partitioning)
8. Access to Peripheral Devices
9. Compatible to existing standard APIs
10. Dedicated Embedded Tooling

Java ME 8 Platform Overview

Use Case Software
(e.g. smart pen)

Use Case Software
(e.g. wireless module)

Use Case Software
(e.g. control unit)

Use Case Software
(e.g. smart meter)

Additional APIs (Examples)

- Messaging
- Wireless Communication
- Protocols and Data Conversion
- Security and Management
- Additional Optional APIs
- Location
- Web Services
- Sensors
- Additional Optional JSRs
- Vertical Specific APIs

On-Device I/O Access
- Device I/O API

Generic Connection Framework
- GCF 8

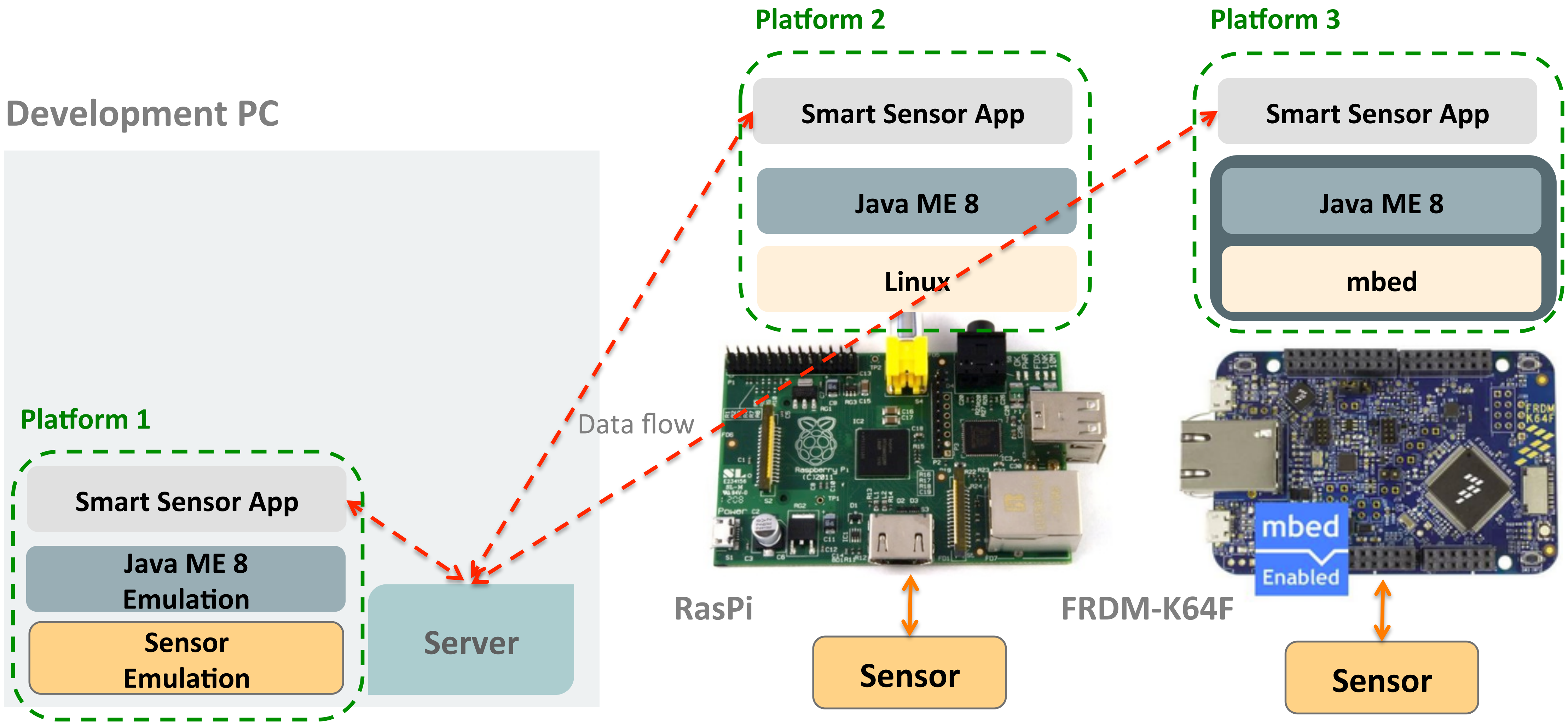
Application Platform
- Java ME Embedded Profile (MEEP) 8 (JSR 361)

Security and Trust Services
- SATSA (JSR 177)

Java VM
- Java ME Connected Limited Device Configuration (CLDC) 8 (JSR 360)

Build a Smart Sensor

Smart Sensor Use Case Diagram



Smart Sensor Use Case

Connected Smart Sensor Use Case on IoT Device

- Application Flow
  1. Connect to I2C sensor and other peripherals (e.g. LED)
  2. Connect to server via network
  3. Periodially read sensor and blink LED as “heartbeat”
  4. Process sensor values
  5. Send sensor values to server
  6. Repeat 3-5
  7. After n seconds, exit app

Platform Comparison

	PC	Raspberry Pi	High-end MCU Device
CPU	X86 @ GHz	ARM11 @ 700 MHz	ARM Cortex-M4 @ 120 MHz
Approx. MIPS	100,000	900	150
OS	Windows 7	Linux	ARM mbed (or multiple others)
Java ME runtime	Emulation	Native ARM/Linux application	Binary image with Java runtime + native support
I/O	Limited I/O • LED and Sensor emulation	Some embedded I/O • LED on GPIO • Sensor on I2C	Lots of embedded I/O • LED on GPIO • Sensor on I2C
RAM **Persistent store**	• >= 1 **GB** • Large disk	• 256 **MB** (or 512 **MB**) • multi-**GB** flash disk	• 256 **KB** (on-chip) • 1 **MB** flash (on-chip) • Maybe external flash space
Power	>= 50 W	~3.5 W	Typ. few 100 mW
Ruggedization	Extra cost	Fragile design	Easy (single-chip SOC)
Cost in volume	>= ~$300	~$35	$5 or less

JavaOne™ ORACLE

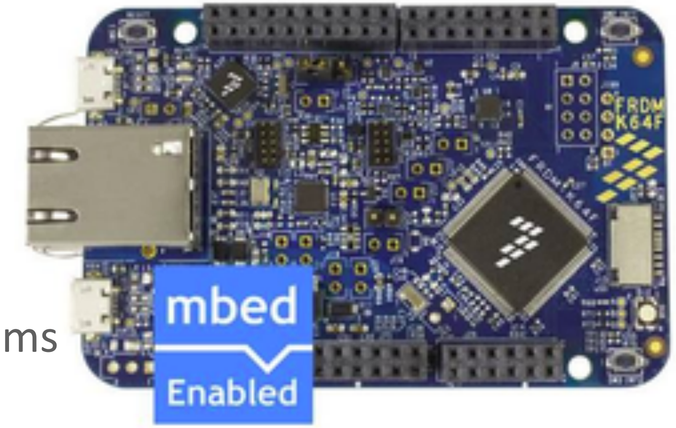
Development Options

- On PC with Java runtime emulation
  - Pros: All on one machine, no extra hardware, flexibility
  - Cons: Not the real thing, limited I/O emulation, memory/timing not accurate
- On Raspberry Pi or other desktop-class embedded device
  - Pros: Real I/O, functionally rich (Linux, storage, networking, etc)
  - Cons: Memory and timing not accurate, different to deployment device
- On Deployment Device, e.g. Micro-Controller
  - Pros: Target hardware, built for use case (cost, power, size, physical, etc)
  - Cons: May have limited flexibility, limited connectivity, limited/no debugging, may be slow, may not be available until late in project

Java 8 for ARM Cortex-M3/M4 Micro-Controllers

- Java ME Embedded 8.1 Developer Preview
  - Supports Freescale FRDM-K64F
    - Kinetis K64F, 120 MHz, 256 KB RAM/1 MB Flash, running ARM mbed OS
    - Arduino form-factor and pin-out. Approx. $25 street price
  - Java ME 8 functionality on small embedded & IoT devices
    - Feature-rich, optimized Java ME 8 runtime in 190 KB RAM, enabling highly functional Java Embedded applications on single-chip micro-controller systems
    - Simple installation
    - Support for Java 8 language, core APIs, networking, device I/O, storage, and more
    - Rich development and tooling via Java ME SDK 8.1 and NetBeans 8 IDE
    - Complements existing Java ME 8 platforms such as Raspberry Pi, scaling Java ME 8 from large to small
    - Ideal for evaluation and prototyping of small embedded & IoT solutions
  - FREE download available now via Oracle Technology Network (OTN)

ARM



freescale

JavaOne ORACLE

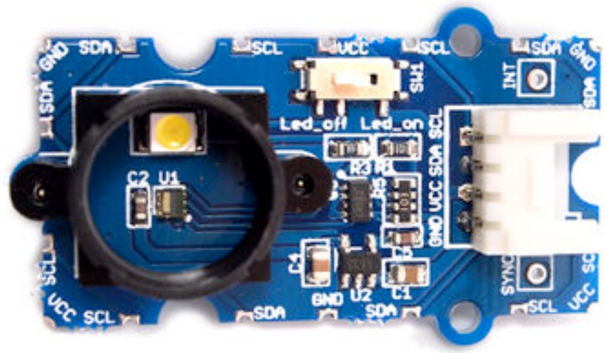
Features: Developer Preview on FRDM-K64F

CLDC 8 “Full Profile”	Full CLDC 8 VM, language, API support
MEEP 8 “Minimal Profile Set”	MEEP 8 application model (single application execution)
Application provisioning and control	• Remote installation (onto SD card) • Remote application execution and life-cycle control
GCF 8 API	Supported protocols: • Socket, secure socket, HTTP, HTTPS, TLS 1.0
NIO File API	Access to SD file system for storage of applications, data, and configuration files
Device I/O API	Supported interfaces/devices: GPIO, I2C, UART, ADC/DAC, SPI, PWM, Pulse Counter, including on-board LEDs, buttons, and accelerometer, magnetometer
Optional APIs	JSON, OAuth 2.0, Async HTTP (as application libraries, memory permitting)
Networking	Ethernet IPv4, DHCP or static addressing
USB/serial	Console output and logging
Tooling via ME SDK & NetBeans IDE	Edit, build, deploy, control (no on-device debugging due to memory limits)
Ready-to-run, flashable binary	Complete Java runtime (includes mbed kernel, native modules, Java libs)
Free heap space for applications	Approx. 60 KB

Setting Up and Developing

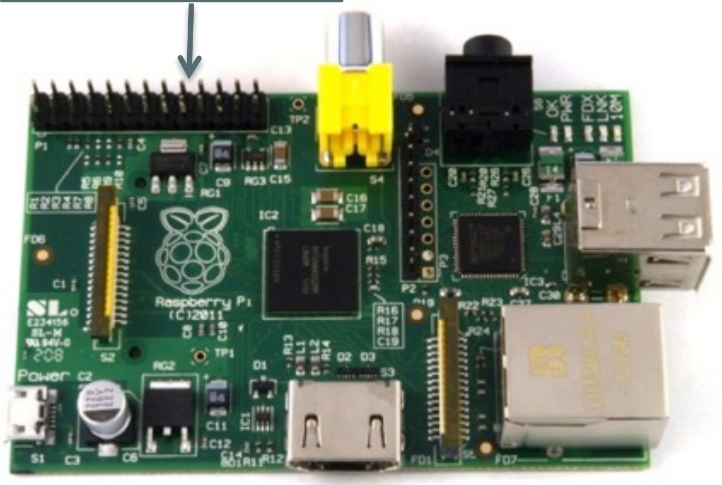
I2C Color Sensor

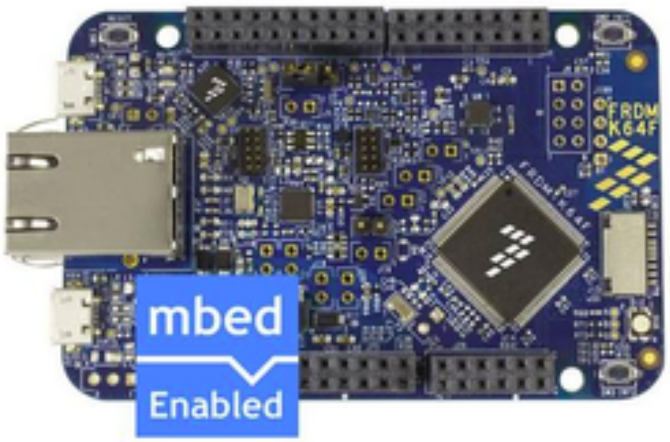
- I2C
  - Inter-Integrated-Circuit bus
  - Low-cost, low-bandwidth, 2 wire serial bus
  - Multitude of devices available
- SeedStudio Grove I2C Color Sensor
  - Built around TC3414CS
  - Includes I2C pull-up resistors
  - Measures red, green, blue, and clear (white)
  - 16 bit digital out on I2C up to 400 KHz
  - I2C Device Bus Address: 57 (x39)



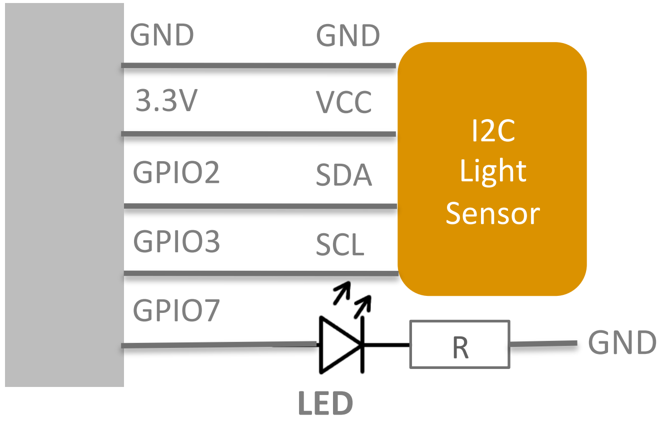
I/O Connections

GPIO, UART, I2C, SPI

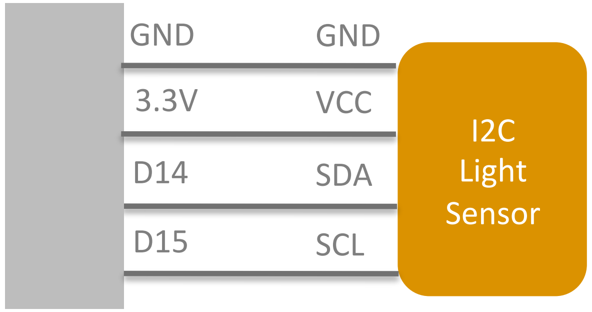




RasPi I/O Connector

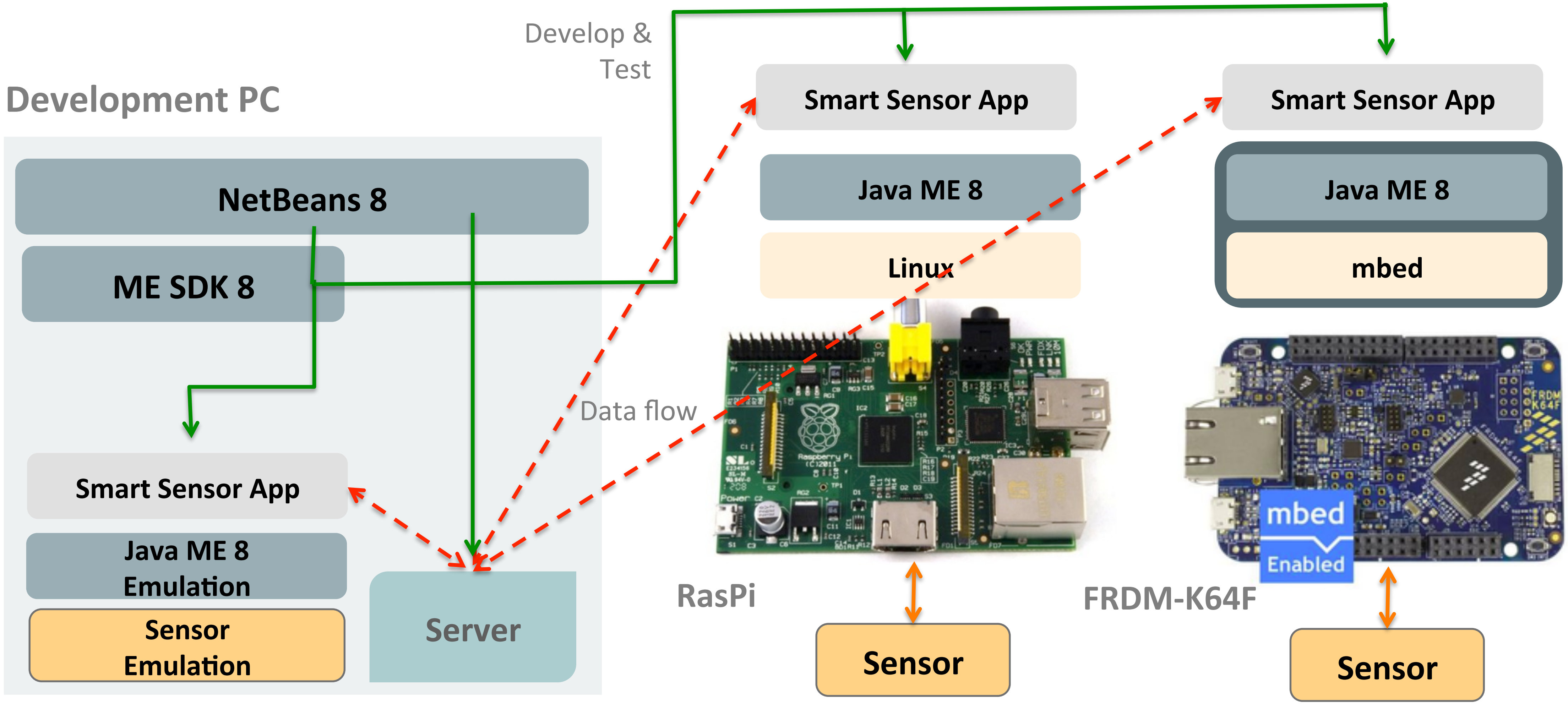


FRDM-K64F I/O Connector



LED: Onboard via LED_PIN_1

Development Setup



Developing with the Runtime Emulation

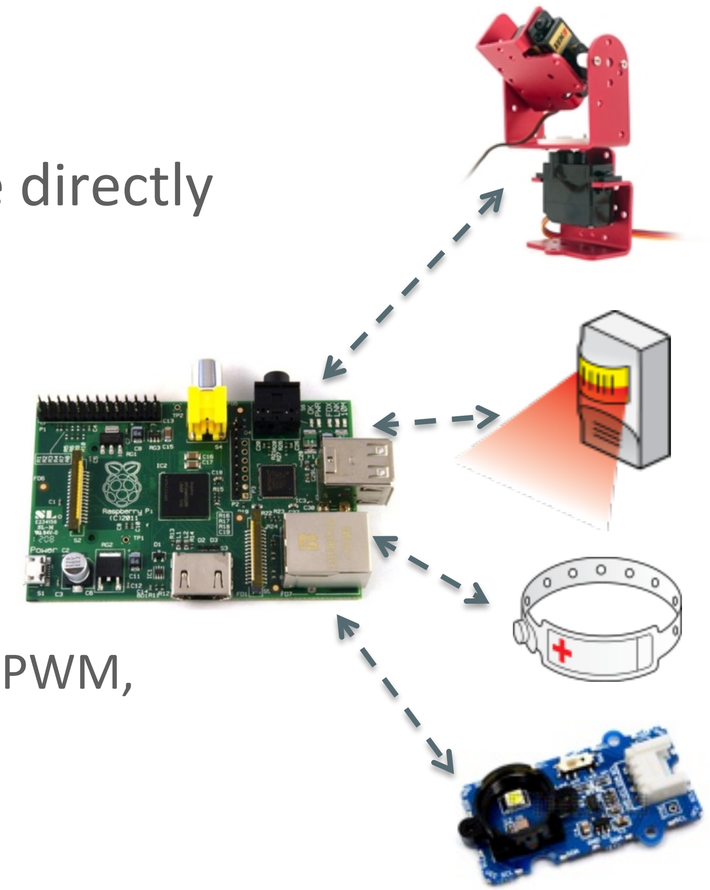
1. Install Java ME SDK & NetBeans
2. Install NetBeans Mobility plugin
3. Install NetBeans Java ME SDK plugins
4. Set up the Java ME platform in NetBeans
5. Create project
6. Use default device emulation or create custom device emulation
   – See https://blogs.oracle.com/javatraining/entry/emulating_i2c_devices_with_java
7. Write and run the app

JavaOne™ ORACLE

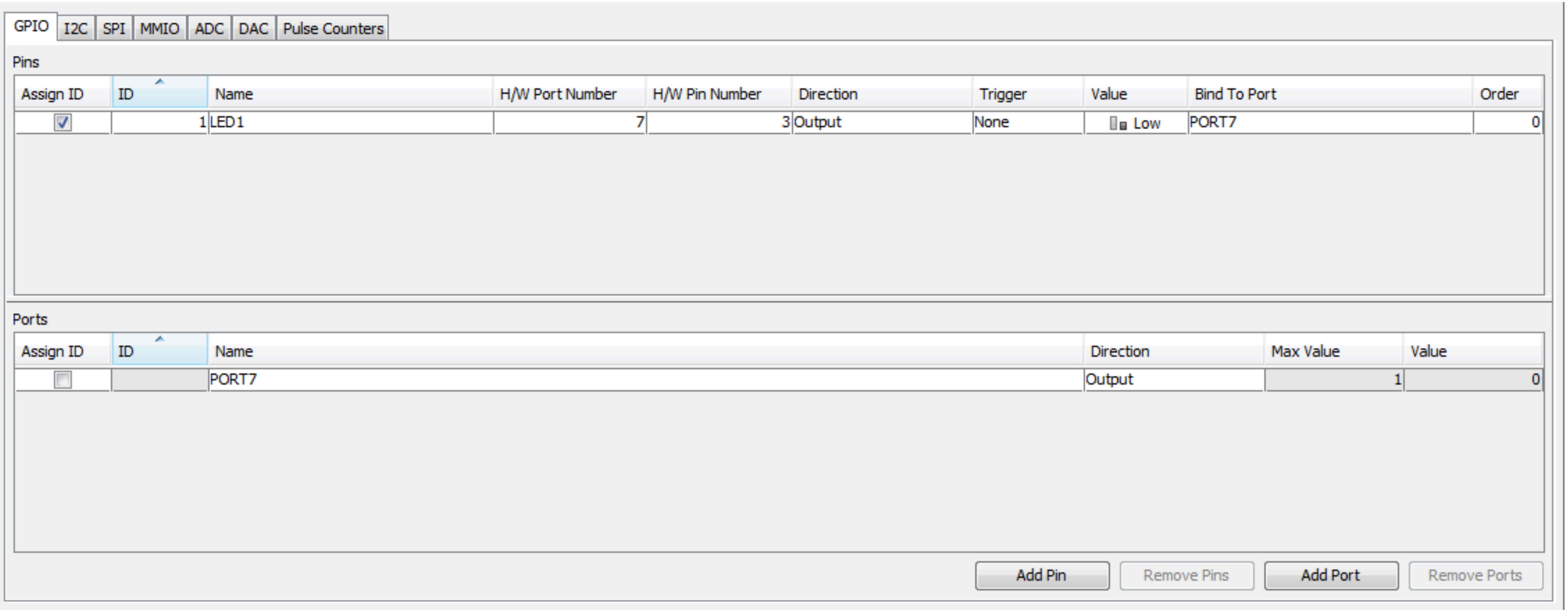
Access to Peripheral Devices

Extensible I/O directly from Java applications

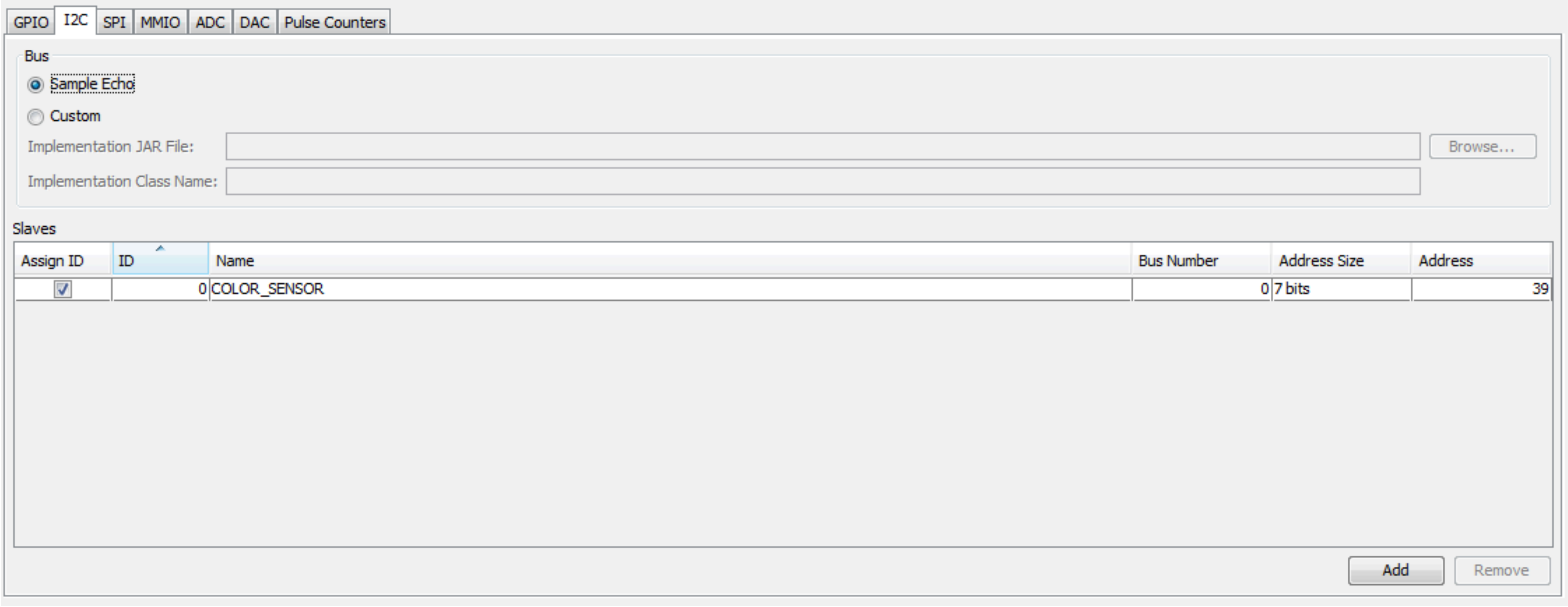
- Device I/O API
  - Platform-neutral access to peripheral device hardware directly from Java, no native coding involved
  - Allows easy support of use-case specific peripherals, such as sensors, actuators, converters, etc
  - Extensible for specialized devices
  - Supports a range of common I/O
    - GPIO, I2C, SPI, ADC, DAC, UART, AT Commands, Pulse counter, PWM, memory-mapped I/O, and more
  - Also planned for Java SE



Configuring custom GPIO in the Emulator



Configuring custom I2C in the Emulator



Java ME Application Model

Application lifecycle control

```
Public class SensorDemo extents MIDlet {
  public void startApp() { // called by AMS to start the app
    // initialization code here …
    new MainController.start(); // then, run main thread
  }
  public void pauseApp() { // called by AMS to start the app
  }
  public void destroyApp(boolean unconditional) { // called by AMS to terminate the app
  }
}

Public class MainController implements Runnable {
  public void run() {
    // main work happens here
  }
}
```

JavaOne ORACLE

Accessing the LED

```
if (RasPi) // RasPi
  ledPin = LED_PIN_7; // well-known platform value
else // Emulator or FRDM-K64F
  ledPin = LED_PIN_1; // well-known platform value
led = (GPIOPin)PeripheralManager.open(ledPin); // open connection to predefined pin
led.setValue(true); // LED on
led.setValue(flase); // LED off
```

Accessing the Color Sensor

```
// For TC3414CS
if (RasPi) // RasPi
  config = new I2CDeviceConfig(1, 57, 7, 100000); // bus 1, address 57 (dec), 100 KHz clock
else // Emulator or FRDM-K64F
  config = new I2CDeviceConfig(0, 57, 7, 100000); // bus 0, address 57 (dec), 100 KHz clock
device = (I2CDevice)DeviceManager.open(config); // open connection to sensor
// Access registers of TC3414CS
tx[0] = (byte)0x80; // control register offset
tx[1] = (byte)0x03; // set ADC_EN to start measurement
device.write(tx, 0, 2); // write to device
tx[0] = (byte)(reg | 0x80); // reg: green = 0x10, red = 0x12, blue = 0x14, white = 0x16
device.write(tx, 0, 1); // write register offset
device.read(rx, 0, 1); // read low byte
tx[0]++; // register offset high byte
device.write(tx, 0, 1); // write register offset
device.read(rx, 1, 1); // read high byte
// compose 16-bit value from lower 8 bits in rx[0] and upper 8 bits in rx[1]
colorVal = (0xFF & rx[0]) + (0xFF & rx[1]) * 256;
```

Connecting to the Server

```
// Using Java ME Generic Connection Framework (GCF)
serverUrl = “socket://” + serverIP + “:” + serverPort; // server address + port
socketConnection = (SocketConnection) Connector.open(url); // open connection
outputStream = new OutputStreamWriter(socketConnection.openOutputStream());
outputStream.write(“Hello”); // write to server
outputStream.flush(); // flush to make sure data is sent right away
```

Server-side Code (Java SE)

```java
// Standard Java SE code
while (true) { // repeat for each client that connects

  // Wait for incoming client connection
  clientSocket = serverSocket.accept();
  // Connection made, get buffered input stream to client
  input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

  String inputLine;
  while ((inputLine = input.readLine()) != null) { // read lines until empty
    System.out.println("read: " + inputLine); // print on console
  }

  input.close(); // close input stream
  clientSocket.close(); // close socket
}
```

JavaOne ORACLE

Running the Code on the Emulator

1. Run DemoServer (only need to start once, runs forever)
2. Select device (“EmbeddedSensorDevice”)
3. Run
   – Start emulator in Java ME SDK
   – Deploy application into emulator
   – Run application, under control of AMS dialog
4. Observe I2C echo
5. Observe server output
6. Optional: Run in “Debug” mode

Developing on the RasPi or FRDM-K64F

1. Install Java ME runtime on device
2. Configure in `jwc_properties.ini` as needed
   - Configure networking, enable debug agent, etc
3. Configure in `_policy.txt`
   - Add `device_access` permissions for untrusted domains
4. On Raspberry Pi: Start Java ME Embedded runtime
5. Connect to board via ME SDK DeviceManager
6. When running application from NetBeans, select desired device

Running the Code on the Raspberry Pi

1. Select device (“EmbeddedExternalDevice1”)
2. Run
   – Deploy application into device (in NetBeans, via network)
   – Run application, under control of AMS dialog
3. See sensor and LED in action
4. Observe server output
5. Optional: Run in “Debug” mode

Running the Code on the FRDM-K64

Scaling down is simple: Java ME does all the hard work for us!

1. Select device ("EmbeddedExternalDevice2")
2. Run
   – Deploy application into FRDM-K64F board via network
   – Run application, under control of AMS dialog

- … done!
  – **Nothing to do**, only switch the deployment target
  – Deploy **exact same Java application binary** to the IoT device(s)
  – **No** specialized expertise, porting, cross-compliation, platform dependencies, specialized languages and tools, etc
  – **Life is good:** Move from large systems to small IoT platforms **in seconds**

Where to go next & Resources

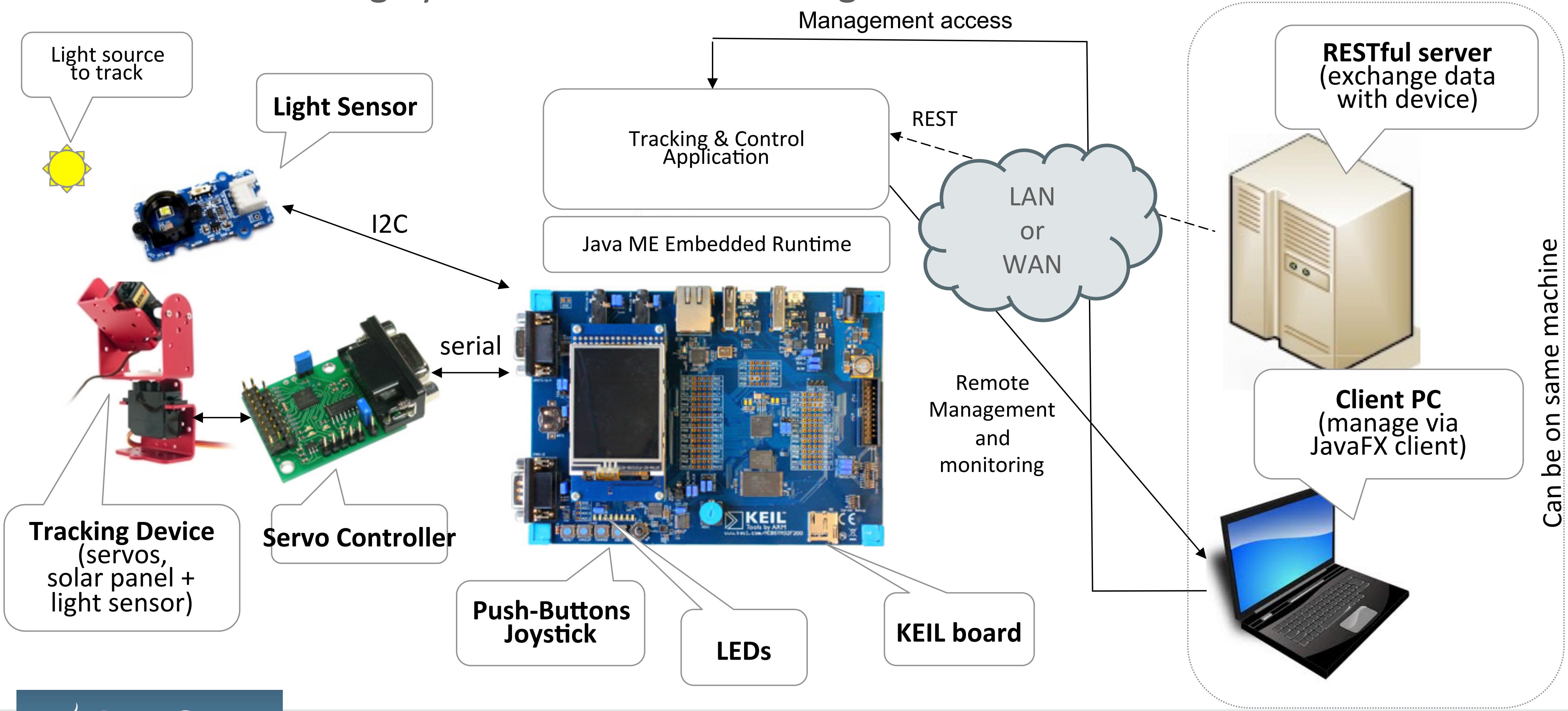
Obviously, just a start!

Lots of Possibilities ….

- Interface with more I/O
  - Different sensors, actuators, other devices
- Add more software functionality
  - Data proprocessing & filtering, notifications, alarms
  - Enhanced networking (p2p, gateway, server, protocols, optimization, etc)
  - Security (authentication, encryption, etc)
- Modularize and partition your software
  - Seperation of concerns, easier development and update
  - Add system management application to manage and control device and software

Example: Industrial Control Demo

Smart Solar Tracking System with Remote Integration



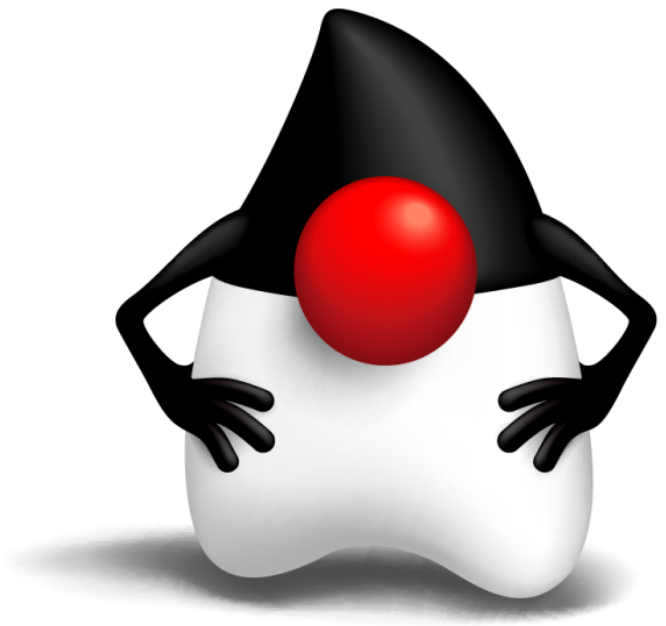
Hints: Optimizing for Resource-Constrained Devices

- Design for the target
  - Be aware of memory and processing limitations
  - Partition your problem and keep local processing small & efficient, push heavyweight operations to next tier (e.g. gateway or server)
  - Leverage built-in Java ME 8 platform functionality
    - Java 8 language features, application framework, security model, built-in libraries and APIs, communication protocols, I/O access, and more
- Conserve footprint
  - Especially important on low-RAM devices (below ~300 to 400 KB RAM)
  - Watch for number/size of classes & number/size of runtime objects
  - Reduce jar size by building with debug info off and enabling obfuscation

Java ME 8 Resources

- Java ME 8 Oracle Technology Network (OTN) downloads
  Free for development and evaluation purposes
  - Oracle Java ME Embedded 8.1 Developer Preview
  - Oracle Java ME SDK 8.1 Early Access #3
  - http://www.oracle.com/technetwork/java/embedded/javame/embed-me/downloads/index.html
- Java ME 8 Documentation
  - Developer Preview on FRDM-K64F: *Release Notes*, *Getting Started Guide*
  - *Java ME 8 Developer Guide*, plus new chapter: *Java ME Optimization Techniques*
  - Full Java ME 8 API doc set
  - http://docs.oracle.com/javame/8.0/
- Terrence Barr's blog
  - http://terrencebarr.wordpress.com/

Questions?



Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Hardware and Software

Engineered to Work Together

JavaOne™

ORACLE®

ORACLE®