Linkwave
Wireless Connectivity

RENFELL
WWW.RENFELL.COM

Practical Legato

A quick start guide to Legato development

Sierra Wireless Innovation Summit | June 13, 2017

Leveraging existing Linux knowledge in the Legato Environment

Use the FX30 and Renfell serial card to build a simple chat application (the Case Inverter)

Serial ports in Linux

- **Serial port in Linux is technically a device**
- **A device in Linux looks like a file**
- **The ‘filename’ is found under the /dev directory and has read/write security permissions like any other linux file**
- **Named depending on what hardware is providing the communications interface**
  - /dev/ttyS0 – first hardware UART (16550 equivalent on x86 motherboard)
  - /dev/ttyUSB0 – first enumerated USB based serial port
  - /dev/ttyUSB1 – second enumerated USB serial port etc
  - /dev/ttyACM0 – another type of USB serial port
- **Devices can come and go – especially USB based serial ports**
- **USB serial ports may not always have the same device name depending on the order that the devices were enumerated – use mdev (embedded version of udev) rules to assign devices to fixed names if required**

Mixing Legato and Linux

- **Legato framework APIs sit over the top of standard Linux APIs**
- **If it's not possible to do in Legato, try to do it using standard Linux API**
- **There may be caveats when mixing APIs – check the documentation first!**
- **It is possible to use existing linux libraries in legato – will need to be cross-compiled before application can run on the WP – source required – no binaries!**

Dealing with a serial port

There is a difference between a fd and FILE*

- **A fd (file descriptor) is a low level integer 'handle' used to identify an opened file/socket/device at the kernel level**
- **A FILE* (FILE pointer) is a higher level library construct used to identify a file and wraps a file descriptor and adds buffering and other features to make I/O easier**

- **fd is used for low level functions such as open(), read(), write() and close()**
- **FILE* is used for buffered I/O functions such fopen(), fgets(), fclose() and similar**

Do not use a FILE* to access a device – devices needs fd to access low level functions such as termios() interface management and ioctl() hardware drivers

Open and set up a Serial port

- Open a serial port using open() – save the fd that is returned and always check for errors!
- A serial port has a number of characteristics that need to be set (BAUD, Stop bits, Parity are the common characteristics, but there lots of others such as flow control, input and output buffering etc)
- Use the termios() family of commands to set the serial port configuration.

```
int32_t SerialFd  = -1;
SerialFd = open(SERIAL_PORT,(O_RDWR | O_NOCTTY);
if (SerialFd < 0 )
{
    LE_FATAL("Error opening %s err[%d]:%s",
    SERIAL_PORT, errno, strerror(errno));
}
int32_t initSerialPort( int pFd )
{
    struct termios options;

    LE_INFO( "Configuring Fd [%d]", pFd );

    /* Get current options for the fd */
    tcgetattr( pFd, &options );

    /* make the device 'raw' */
    cfmakeraw( &options );

    /* set the baud rate */
    cfsetspeed( &options, SERIAL_BAUD );

    /* and set the new options to the fd */
    tcsetattr( pFd, TCSANOW, &options );

    return pFd;
}
```

Sidebar: The Legato Sandbox

- **Legato application runs inside a ‘sandbox’ by default**
- **Sandbox enforced at Operating System level using a chroot() jail**
- **Sandbox prevents application accessing anything outside chroot() jail**
- **This includes accessing devices**

Sidebar: The Legato sandbox (continued)

There are two ways around this:

- Resources outside the application can be added to the ‘requires’ stanza in the Component.cdef. Files, devices, directories and libraries can all be ‘required’ by an app and will either be copied or symlink’d into the application sandbox by the legato framework*

Or:

- The application can be configured to run ‘unsandboxed’ using the sandboxed:false command in the application .adef file.

Running un-sandboxed may have reliability and security implications.

*If a device doesn’t exist when the application is started, the application will fail to start because the ‘resource is unavailable’. This is an issue when expecting to use a USB or other removable device and the device is not connected at start-up.

Waiting for input

Obvious: loop waiting for read() to return data

Smarter: use select() or poll() to monitor the fd, then read when data available

Both methods are bad in the Legato environment as will block execution inside COMPONENT_INIT{}

***proper* legato method:** use the File Descriptor Monitor API to set up event handler to listen for FD events.

Sample application: The CaSe InVeRtEr

Requirements:

- **Open serial port (/dev/ttyUSB0)**
- **Configure serial port for 115200 Baud, 8 data, 1 stop, no parity**
- **Set up a fd event monitor for the ‘data available to read()’ (POLLIN) event**
- **Listen and read() incoming characters**
- **For each character, if it’s an alpha, invert the case, else do nothing**
- **Write() the modified character back to the serial port**
- **Run un-sandboxed so that the application will always start, even if the serial device is not present**

Remember, the COMPONENT_INIT{} function HAS to complete, or other components in the application will not run.

The CaSe InVeRtEr: main.c

- Required legato header files
- ‘C’ library standard header files
- Local defines
- Global variables
- Initialize the serial device as a ‘raw’ device
- Use the linux standard termios() family of functions

```
#include "legato.h"
#include "interfaces.h"

#include <ctype.h>
#include <stdio.h>
#include <stdint.h>
#include <termios.h>

#define SERIAL_PORT "/dev/ttyUSB0"
#define SERIAL_BAUD B115200

/* Global Variables */
int32_t           SerialFd        = -1;
le_fdMonitor_Ref_t SerialFdMonitor = NULL;


int32_t initSerialPort( int pFd )
{
    struct termios options;

    LE_INFO( "Configuring Fd [%d]", pFd );
    /* Get current options for the fd */
    tcgetattr( pFd, &options );
    /* make the device 'raw' */
    cfmakeraw( &options );
    /* set the baud rate */
    cfsetspeed( &options, SERIAL_BAUD );
    /* and set the new options to the fd */
    tcsetattr( pFd, TCSANOW, &options );

    return pFd;
}
```

The CaSe InVeRtEr: main.c

- Set up the file descriptor monitor event handler
- If there is data available (POLLIN event)
- Loop forever
- Read a character
- Unknown error read()ing, exit
- No data left, break out of loop
- Process read data and write back to device

```
static void serialFdMonitorHandler(int pFd, short
    pEvents)
{
    if (pEvents & POLLIN ) // data available for read
    {
        ssize_t len;
        // need to read characters until no more data
        for (;;)        // read all data
        {
            char ch;
            // read single byte at a time
            len = read(pFd, &ch, 1);

            // test for error
            if ((len == -1) && (errno != EAGAIN))
            {
                LE_FATAL("read read() error:[%d:%s]",
                          errno, strerror(errno));
            }
            // if nothing to read,
            // len==-1 && errno==EAGAIN. bail out
            if ( len <= 0 )
            {
                break;  // break out of read() loop
            }
            // process buffer here
            if ( isalpha(ch) ) // A-Z or a-z
            {
                ch = ch ^ 0x20; // invert by xor'ing
            }
            write(pFd, &ch, 1);
        }
    }
}
```

The CaSe InVeRtEr: main.c

- **Signal handler to catch when the application is terminated**
- **Lets us clean up nicely**

```
static void sigHandlerSigTerm( int pSigNum )
{
    LE_INFO("SIGTERM caught, closing app");
    if ( SerialFdMonitor != NULL )
    {
        le_fdMonitor_Delete( SerialFdMonitor );
        SerialFdMonitor = NULL;
    }
    if ( SerialFd > -1 )
    {
        close(SerialFd);
        SerialFd = -1;
    }
}
```

The CaSe InVeRtEr: main.c

COMPONENT_INIT{} is the legato equivalent of main()

- Register handler to catch termination signal
- Open the serial device
- Initialize the serial device as a ‘raw’ device
- Setup the fd monitor to call an event handler when a POLLIN event (data is ready) is received

```
COMPONENT_INIT
{
    LE_INFO("FX30 USB Serial test init");

    // setup to catch application termination and
    // shutdown cleanly
    le_sig_Block( SIGTERM );
    le_sig_SetEventHandler( SIGTERM, sigHandlerSigTerm );

    // open the serial port
    if ( (SerialFd = open(SERIAL_PORT, (O_RDWR | O_NOCTTY))
                        ) < 0 )
    {
        LE_FATAL("Error opening %s err[%d]:%s", SERIAL_PORT
                                  ,errno, strerror(errno));
    }
    LE_INFO( "Open %s OK. fd[%d]",SERIAL_PORT, SerialFd );
    // configure UART port using termios
    initSerialPort( SerialFd );

    // set up FD monitor
    // note: this doesn't return on error, so no need to
    // check for errors....
    SerialFdMonitor = le_fdMonitor_Create("SerialPort"
                        ,SerialFd
                        ,serialFdMonitorHandler
                        ,POLLIN
                      );
     /* Remember that COMPONENT_INIT() must exit or the
     rest of the application will never run */
}
```

The CaSe InVeRtEr: Component.cdef

The Component.cdef file is straightforward

- To run the application ‘sandboxed’, the component requires the device to be linked from the root file system into the sandbox.
- Note that the application also needs both Read and Write access to the device

```
requires:
{
/*
 * create external IPC APIs that this component needs
 * to operate
 * These are generated into interfaces.h - which
 * needs to be included in any
 * source files that use the interfaces
 *
 * NOTE: These instances may need to be bound to the
 * appropriate service in the adef:bindings stanza
 */
    api:
    {
    }
    device:
    {
        // read-write access to USB Serial device
        // mapped internally to component, so
        // shouldn't need to run un-sandboxed
        [rw] /dev/ttyUSB0 /dev/ttyUSB0

    }
}
```

The CaSe InVeRtEr: caseinverter.adef

The adef file is very simple – no bindings or requires are needed even though we're mixing Legato and 'standard' linux

- Option to run the application 'un-sandboxed'
- Manual start for debug purposes
- Set the fault action to ignore (default, but let's be explicit about it)

```
sandboxed: false

version: 16.07.0
maxFileSystemBytes: 512K

start: manual

executables:
{
    caseinverter = ( caseinverterComp )
}

processes:
{
    envVars:
    {
        LE_LOG_LEVEL = DEBUG
    }
    run:
    {
        ( caseinverter )
    }
    maxCoreDumpFileBytes: 512K
    maxFileBytes: 512K

    faultAction: ignore
}
```

Demonstration

Demonstration