Lingua Franca on NRF

Abhi Gundrala, Chinmoy Saayujya, Eduardo Diaz, Samuel Berkun, Zhihao Deng
gundralaa,chinmoy,eduardoidiaz,sberkun,zhdeng@berkeley.edu

Abstract

Lingua Franca (LF) is a coordination meta-language, based on an actor model. Code is written in a mix of LF syntax and the host language (in our case, C), compiled to the host language, and then run on the target architecture. Before this project, the C runtime of LF was only supported on Linux, MacOS, and Windows, with no support for embedded platforms. Our main challenge was to port and demonstrate usage of LF on the nRF52832 SoC.

1 Introduction

Lingua Franca (LF) is a polyglot coordination language that emphasizes determinism and explicit timing, displaying particular advantages for applications that require concurrent and time-sensitive execution. In the LF paradigm, components called reactors are defined and the interations between them specified by means of reactions. The logic within each reactor is written in one of many compatible languages including C, C++ and Python. The LF program is passed through a code generator, which in-turn produces a program in the target language that can subsequently be compiled and executed using standard workflows. The LF project aims to achieve platform independence, or the ability to reproduce deterministic behavior across multiple processor targets and architectures.

The nRF52832 (henceforth referred to as the NRF) is a system on a chip developed by Nordic Semiconductor. It includes an Arm Cortex-M4 CPU and a Bluetooth Low Energy radio [1]. Its considerable processing power combined with the extensive libraries provided by Nordic Semiconductor make the nRF52832 a useful platform for embedded projects and bluetooth applications in particular. The nRF52832 is used in conjunction with the Berkeley Buckler, an educational board with additional peripherals used as part of the EECS149/249A course at UC Berkeley. For the Buckler, Berkeley's Lab11 provides a repository with libraries and tools that may be used in programs written for the NRF [2].

In this project, we extended LF support to the nRF52832 by writing the necessary supporting functions, linkers and makescripts in order to further the goal of ensuring consistent LF performance across multiple platforms and architectures. The compilation, execution and output of multiple tests programs written in LF were compared between runs on the nRF52832 and an Intel target. We also provided a simple demonstration to showcase the functionality of LF on the nRF platform while displaying the advantages of LF over traditional bare-metal C code.

2 Platform Architecture

2.1 Repository Structure

In order to demonstrate both the features of the NRF and Lingua Franca as much as possible, work had to be done to consolidate their repository structures. Lingua Franca expects source `.lf` files to be stored in a `src/` folder, and then will generate `src-gen/` and `bin/` folders with compiled code. On the NRF side, the most straightforward build process is to compile and flash code from within the Buckler repository, which is set up to allow easy use of libraries and board feature (such as the Buckler's LCD screen). Source C files are stored in a `software/apps/projectname/` folder, which can reference libraries and board configurations in the `software/` folder. In order to combine these two structures, we based our repository structure off of the Lingua Franca repository and moved the supporting folders (namely, `libraries/`, `buckler_revC/`, and `nrf52x-base/`) to the top level. We then modified the Makefiles such that they would point to these folders.

2.2 Build Process

In order to compile a Lingua Franca project for the NRF, users put the `.lf` files in the `src/` folder and run our build script (For detailed instructions on using our build process, refer to the repository's `README.md`). The build script first calls the LF compiler, which generates C code. Then, the build script adds several files to the generated C code, needed to provide hardware support to the NRF. These are:

- `include_nrf.c`: This file defines the `CLOCK_REALTIME` and `CLOCK_MONOTONIC` constants (which are defined by the OS for POSIX systems, but have to be manually defined for the NRF), and includes `lf_nRF52832_support.c` (so that GCC knows to compile it).

- `lf_nRF52832_support.c` and `lf_nRF52832_support.h`: These files provide methods to initialize the clock, get the current time, and sleep. On the NRF, these are implemented using a 1MHz hardware timer.
- `platform.h`: This file includes `lf_nRF52832_support.h` so that the rest of the runtime can use it.
- `Makefile`: This Makefile points to makefiles from `nrf52x-base/` and `buckler_revC/`, which specify how to compile the code and flash the NRF.

At this point, the generated C code is ready to be compiled and flashed to the NRF. The build script provides an option to do this automatically, but this can also be done manually by navigating into the generated C code and running `make flash`.

A high level overview of the components in the process is summarized in Figure 1. Note that although the build script is labelled as a component in the process, the build script also invokes both the LF compiler and `make`. From a usage standpoint, the entire build process (except writing the original LF program) is handled by the build script.

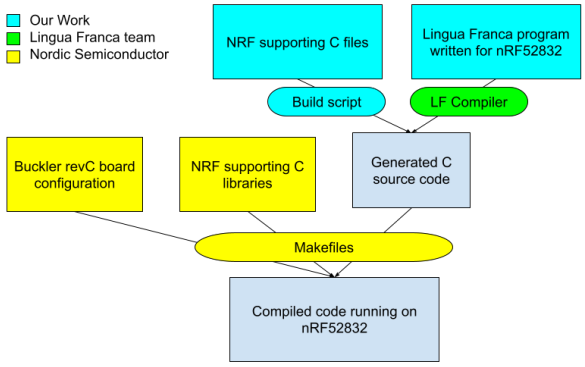

Figure 1: Top-down system architecture required for LF compatibility on nRF52832

2.3 Testing Methodology and Results

Once the nRF support files were written and the build script and linkers completed, we tested the nRF implementation of LF on the provided source test files in the LF repository. The LF linker script was used to compile a C object which was flashed to the nRF52832 board, from which the runtime output was received and displayed using the Real-Time Text (RTT) protocol. The tests were also compiled and executed on an Intel target and the outputs of the nRF and Intel platforms compared. Of the 98 total test files, we observed no runtime errors. 84 tests ran successfully an showed identical results on both the Intel and nRF targets. Ten (10) tests displayed some output differences but no runtime errors, an example of which is shown below:

`DelayedAction.lf` Intel Target Output:

```
Nanoseconds since start: 100000000.
Physical nanoseconds since start: 806000.
Nanoseconds since start: 1100000000.
Physical nanoseconds since start: 817000.
Nanoseconds since start: 2100000000.
Physical nanoseconds since start: 821000.
Nanoseconds since start: 3100000000.
Physical nanoseconds since start: 826000.
Nanoseconds since start: 4100000000.
Physical nanoseconds since start: 830000.
---- Elapsed logical time (in nsec): 5,000,000,000
---- Elapsed physical time (in nsec): 846,000
```

`DelayedAction.lf` nRF52832 Output:

```
Nanoseconds since start: 100000000.
Physical nanoseconds since start: 221000.
Nanoseconds since start: 1100000000.
Physical nanoseconds since start: 415000.
Nanoseconds since start: 2100000000.
Physical nanoseconds since start: 613000.
Nanoseconds since start: 3100000000.
Physical nanoseconds since start: 810000.
Nanoseconds since start: 4100000000.
Physical nanoseconds since start: 1008000.
```

Tests in which these differences arose usually had multiple reactors with multiple reactions that resulted in RTT outputs, all of which were bound by a total runtime constraints. We attribute these runtime differences to differences in the clock speed between the nRF board and the Intel target, due to which some LF programs were not able to complete execution before the execution deadline was reached. In the above example (`DelayedAction.lf`), execution of the LF program was made to time out after 1 ms of physical time. As such, while the logical timepoints in both executions matched exactly, the slower nRF processor timed out during execution and failed to print the final two statements. Four (4) test cases (`CountTest.lf`, `HelloWorld.lf`, `ManualDelayedReaction.lf` and `SlowingClockPhysical.lf`) threw compile-time errors and thus failed to produce a C object that could be flashed to the nRF. Upon further investigation, we noted that each of these cases required some form of threading, which is not supported by the nRF52832. Other versions of the nRF52x series of embedded platforms do support multithreading, and may be investigated in future.

3 Concurrency Demo: Haptic Depth Sensing

3.1 Motivation

About 12 million people in the US suffer from severe vision impairment, and a quarter of those are legally blind [3]. The most common assistive device used by the visually impaired, the white cane, does not work very well in detecting objects at chest level and above. We propose haptic depth sensing to remedy this problem, which utilizes motors that vibrate according to the depth detected by ultrasonic sensors. For best effect, we utilize multiple depth sensors, which raises problems of concurrency, since the pings sent by one sensor might be wrongly read by another.

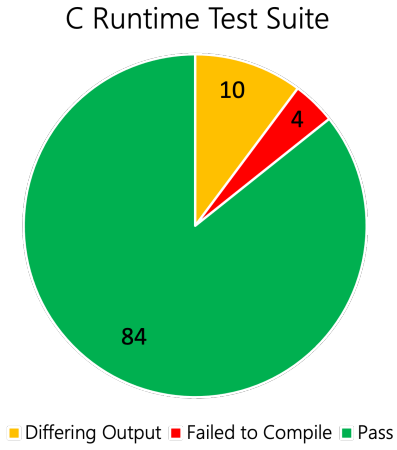

Figure 2: Pie chart showing proportion of total tests (98) that passed (84), differed in output (10) and failed to compile (4).

3.2 Software Design

The Finite State Machine (FSM) that depicts the concurrent operation of two ultrasonic distance sensors (and two connected haptic motors) is shown in Figure 3. Each ultrasonic sensor switches between 3 states: $\mathbf{u_{OFF}}$ (off state), $\mathbf{u_{TX}}$ (the transmit state) and $\mathbf{u_{RX}}$ (the receive state). When one ultrasonic sensors is in either the transmit or receive states, all other ultrasonic sensors must be in the off state. An example workflow of the FSM in the midst of continuous operation is as follows:

1. Sensor 2 completes a full TX/RX cycle and outputs `done1`. Sensor 1 receives `done1` as input, sets $y(t) := 0$ and transitions from $\mathbf{u1_{OFF}}$ to $\mathbf{u_{TX}}$.

2. Sensor 1 transmits an ultrasound ping (8 pulses at 40 kHz lasting a total of $t_{pulse} = 200\ \mu$s). The internal variable $y(t)$ reaches t_{pulse} at the end of this cycle, after which sensor 1 transitions to state $\mathbf{u1_{RX}}$.

3. While in this state, two scenarios are possible:

    - Sensor 1 receives an echo from the environment, and input `echo` is present. The timer variable $x_1(t)$ records the value of the total to-and-fro time of flight of the ultrasound ping, corresponding to twice the distance to the object.
    - Sensor 1 does not receive an echo from the environment before $x_1(t) \geq t_{TO} = 42$ ms, which is the timeout time constant corresponding to an object approximately 7m away. Since the maximum range of the ultrasonic sensor as stipulated by the datasheet is 4m, we assume that an object distance of 7m or greater will result in too weak of an echo to mistakenly trigger any other active ultrasonic sensor in the event of a return ping.

4. Sensor 1 returns to the $\mathbf{u1_{OFF}}$ state outputs the `done1` signal, which results in sensor 2 being activated. Simultaneously, haptic motor 1 (**m1**) receives the distance $x_1(t)$ detected by sensor 1. It performs the necessary calculates to set the level of motor activation and updates this value internally.

The above cycle repeats, resulting in sensors 1 and 2 activating alternately.

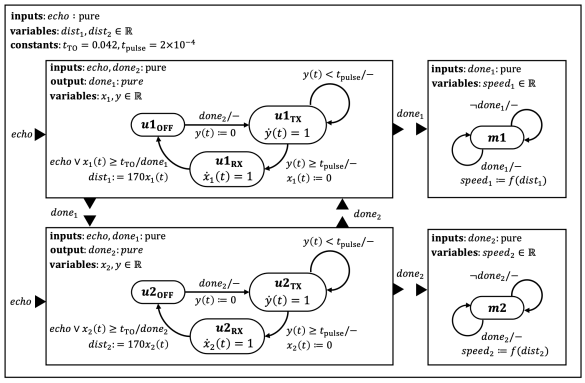

Figure 3: FSM depicting a system that uses alternating ultrasonic sensors to concurrently drive 2 sets of haptic motors.

3.3 Advantages of Lingua Franca

Lingua Franca was a particularly useful framework for this demo because concurrent state machines, such as the one in Figure 3, translate very naturally to Lingua Franca reactors. Lingua Franca also allows reuse of components (such as motors and ultrasonic sensors), which helps reduce the amount of code and simplify the problem. Our equivalent C version of the demo maintained all state and input/output with global variables; not only is this more prone to error, but it scales horribly. Our demo only had two ultrasonic sensors, but could easily scale to 6-10 of them. The raw C version, with its current design, would have a few dozen local variables, and would probably have to be redesigned from scratch to avoid becoming a ball of spaghetti. In contrast, the Lingua Franca version scales up easily, by simple creating more instances of the reactors and connecting their inputs and outputs appropriately.

3.4 Hardware Design

The hardware implementation of a single sensor-motor pair makes use of 3 GPIO pins as well as the `VCC` and `GND` pins of the nRF52832 as seen in Figure 4. We utilized the SainSmart HC-SR04 ultrasonic sensor in order to measure distances and a button-size DZS piezoelectric haptic motor to relay the measured distance via tactile sensation. 2 board GPIO pins were connected to the `TRIG` (trigger) and `ECHO` (echo) pins of the ultrasonic sensor. A $\geq$10 μs Transistor-Transistor Logic (TTL) high signal is sent to the trigger pin in order to begin the transmit/receive sequence of the sensor.

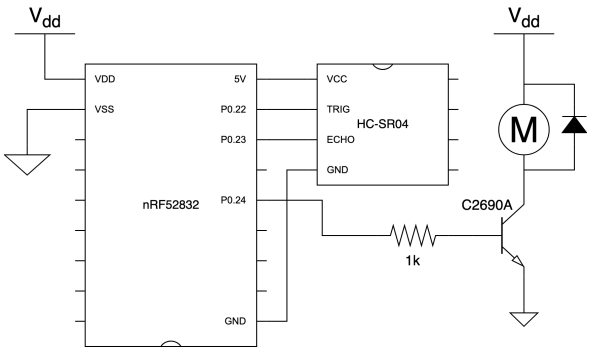

Figure 4: Schematic of nRF pinout alongside a single HC-SR04 ultrasonic sensor and transistor-diode motor driver pair. The actual device has two sensor-motor pairs.

After this TRIG pin is pulled low, the sensor emits 8 pulses of ultrasound at 40 kHz. The `ECHO` pin is then pulled TTL high by the HC-SR04, indicating that the pulses have been emitted. When the echo is received, the `ECHO` pin is pulled low. The distance to the object is calculated from the time for which the pin remained high. The haptic motor is driven by the third GPIO pin in the pulse-width modulated (PWM) operating mode. The motor sinks more current than the nRF can provide and thus required a separate power supply. The motor was switched on and off using an NEC C2690A N-type power transistor through a 1 kΩ series base resistance. A free-wheeling diode is placed in shunt with the motor in order to prevent wind-down motor currents from being forced through and damaging the transistor when it is off.

4 Course Concepts

4.1 Composition of State Machine

The fundamental challenge behind our demo is composing the state machines for the ultrasonic sensors and the motors, which all run concurrently. Our final state machine, depicted in Figure 3, uses synchronous composition, which translates well into LF code.

4.2 Input and Output

In order to get input from the ultrasonic sensors, we used interrupts that triggered on the falling edge of each echo pin.

4.3 Sensors and Actuators

Our hardware design fundamentally depended on our sensors (the ultrasonic sensors) and the actuators (vibration motors, and the pulses from the ultrasonic sensors). The relationship between the echo time and the distance measured from the ultrasonic is a linear model, with relationship $d = 0.034t$, where d is the distance in cm and t is the echo time in microseconds. We assumed that the motors also have a roughly linear model between average voltage and vibration strength, although we don't have an accurate way to measure this.

5 Difficulties and Future Work

The nRF52832 board does not support threading and is unable to naively handle physical actions. We attempted to use interrupts to handle physical actions in LF but this proved difficult due to the incompatibility between the LF task list and bare-metal interrupts. Future versions of the nRF SDK may support threading, which will allow for the multi-threaded and physical action support in LF.

However, creating full-featured support for single-threaded execution, while more complex, may be more valuable. At a high level, one of the major milestones to accomplish this would be to support scheduling actions/events from within interrupts. This presents two immediate difficulties.

The first main difficulty is that the event queue may not be modified from within the interrupt. If the main thread is in the middle of processing an event, modifying the event queue will leave it in an undefined state. The obvious solution is to protect it with a mutex, but then what should the interrupt do if the mutex is locked? There is a possible solution if we can assume that "scheduling interrupts" (interrupts that schedule LF actions) can't preempt each other. In that case, the problem can be solved by keeping 2 event queues, protected by an atomic boolean. Since the main thread can only be processing one event queue at a time, the interrupt will always be able to access and modify the other one. However, special care has to be taken to preserve determinism; In particular, both event queues have to be checked for events, and their actions processed in order. This may hurt performance, and make the internal design of the runtime significantly more complex.

The second main difficulty is with sleeping. If the main thread is sleeping when the interrupt adds an event to the event queue, the event won't be processed until the main thread wakes from sleep. This is an issue if the interrupt wants to schedule an immediate event, since the event will necessarily be processed late. In order to circumvent this, one solution could be to use busy waiting instead of sleeping (as suggested to us by Professor Edward A. Lee during project exposition). This would hurt efficiency and performance (since the processor is always running), but would solve this issue, as well as make the runtime more platform-independent.

Acknowledgements

We would like to thank Shaokai for his help during office ours and in brainstorming our LF demo.

References

[1] Nordic semiconductor nrf52832 system on chip infocenter. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fstruct_

nrf52%2Fstruct%2Fnrf52832.html. Accessed: 2021-11-30.

[2] CDC.gov. . nrf52x base repository: Lab11 on github. https://github.com/lab11/nrf52x-base. Accessed: 2021-11-30.

[3] CDC.gov. . United states centers for disease control and prevention: Fast facts of common eye disorders. https://www.cdc.gov/visionhealth/basics/ced/fastfacts.htm. Accessed: 2021-11-30.

A Repository Links

Lingua Franca on NRF:
https://github.com/sberkun/buckler
Berkeley Buckler:
https://github.com/lab11/buckler
Lingua Franca Compiler:
https://github.com/lf-lang/lingua-franca
Lingua Franca C Runtime:
https://github.com/lf-lang/reactor-c