Performance Analysis using NXP's i.MX RT1050 Crossover Processor and the Zephyr™ OS

A benchmark study to understand performance advantages as compared to Linux BSP on i.MX 6UL

FLORIN LEOTESCU
MARIUS CRISTIAN VLAD

Abstract

Software and hardware performance analysis is integral to the evaluation of system efficiency in embedded designs. Such analysis helps to understand system limitations, identify performance bottlenecks, and determine how well the system performs in comparison with other devices and operating systems. Specifically for the fast-growing Industrial & IoT applications, we need embedded systems with excellent real-time response to ensure deterministic, high performance behavior. In this work, we compare Zephyr OS running on high performance MCUs to Linux running on applications processor, and demonstrate the significant advantage of Zephyr OS for real-time embedded systems.

Traditional MCUs have not fared well against Applications Processors for overall system efficiency, primarily due to the big performance gap between the two. However, with the launch of NXP's i.MX RT Crossover processors that can run at 600MHz, the performance gap is no longer a limiting factor for MCUs. On the software side, Zephyr™ OS is fast gaining popularity due its open-source nature & comprehensive middleware support. Thus we now have an unique opportunity to run efficient RTOS on high performance MCU to create more of a level field against applications processors and therefore, truly compare the 'real-time system efficiency' between RTOS and Linux.

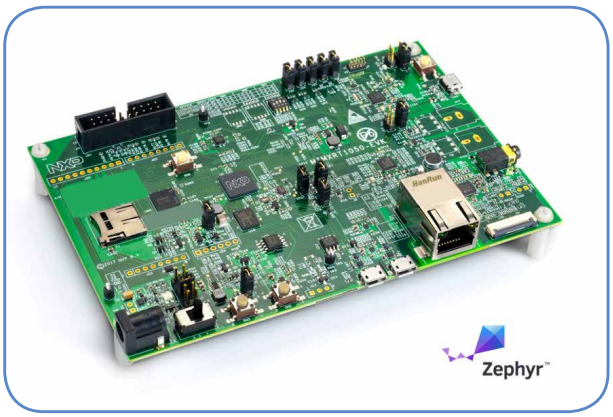
To assess the real-time efficiency of an embedded system, we identified four key software execution metrics, namely, dynamic memory allocation & deallocation, mutex lock & unlock, thread creation & joining, and context switching. We then developed synthetic microbenchmarks to measure these metrics for the systems under test. Given the inherent differences between a RTOS designed for IoT and Linux that is derived from Unix-like desktop OS kernel, this comparison is not fully 'apples to apples,' but is sufficient to provide embedded designers a benchmark to compare solutions when executing similar tasks. The analysis shows that NXP's i.MX RT1050 (Arm® Cortex®-M7) running Zephyr™ OS is 5-100x faster in executing the four performance metrics compared to an applications processor running Linux at the same speed (in this case, Cortex-A7 based i.MX 6UL), thus offering a more cost-effective real-time solution. The following chapters explain the metrics, measurement methodology, and comparative results.

Contents

Configuration Information

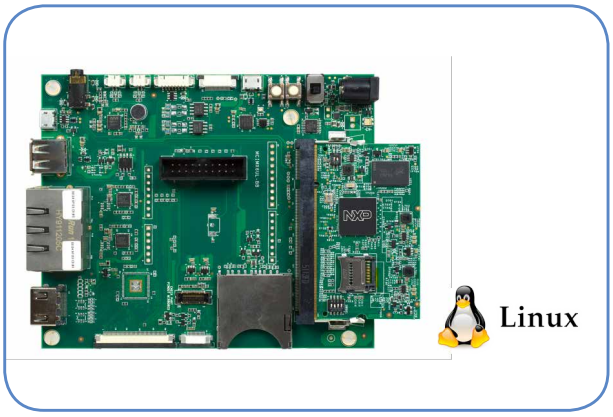
i.MX RT1050 configuration – Hardware + Software

- Development board: MIMXRT1050-EVK
  - Processor: MIMXRT1052DVL6A Arm® Cortex®-M7 core
  - Number of cores: 1
  - Core Frequency: 600 MHz
  - Board schematic: SCH-29538 REV A1
- OS name: Zephyr OS 1.11.99
  - OS type: Real Time OS
  - Zephyr OS web page



i.MX 6UL configuration

- Development board: [MCIMX6G2CVM05AB]
  - Processor: i.MX6UL: i.MX 6UltraLite Processor based on Arm Cortex-A7 core
  - Number of cores: 1
  - Core Frequency: 528 MHz
  - Board schematic: SCH-29163 REV A2
- OS name: Linux BSP - kernel 4.9.88-imx_4.9.88_2.0.0_ga
  - OS Type: Non- Real Time OS



Introduction

Analysis scope

This research is intended to evaluate the performance of the i.MX RT1050 EVK with Zephyr OS in comparison with i.MX 6UL EVK with Linux BSP. Our goal was to determine the performance gap between the IMXRT1050-EVK board equipped with an embedded ARM SoC and a similar board equipped with an application processor. Because the closest i.MX family CPU configuration to the i.MX RT1050 EVK is the i.MX 6UL EVK, we selected the i.MX 6UL development board for best comparison.

In addition, we selected the Zephyr OS as compared to other real time operating systems (RTOS) because it is free and very comprehensive (e.g. with complete network and security stacks), developed as a collaborative project, and supported by an active open source community. While both Zephyr and Linux OSes can exhibit real time characteristics, the Zephyr OS was originally designed to fully abide to traditional RTOS principles, whereas Linux has traditionally served larger workloads for the desktop and server spaces. Furthermore, Linux requires additional patches to abide to fundamental RTOS principles.

To obtain comparable results, despite the known operating system differences, we focused on using the same application peripheral interface (API) for the microbenchmarks. The microbenchmarks were developed in C language and made use of the Pthreads API library (POSIX API library). In the case of Zephyr OS, the available API version was POSIX PSE52, which according to Zephyr community documentation, implements only partial support for the full POSIX specifications.

The microbenchmarks perform memory allocation & deallocation, mutex lock & unlock, thread creation, thread joining, context switching, and records the time spent in each of these actions.

To determine the time spent performing the tasks, we used the POSIX clock_gettime () for the Linux + i.MX 6UL EVK solution, and for the Zephyr OS running on i.MX RT1050 EVK, we used the TIMING_INFO_PRE_READ () function instead of clock_gettime(). Due to the nature of the OS scheduler on Zephyr, the clock_gettime() function generates inconsistent timing values, and the fact that Zephyr source code also uses the TIMING_INFO_PRE_READ() function, we continued with TIMING_INFO_PRE_READ() function.

Dynamic memory (Heap) allocation & deallocation benchmark

In C programming language, dynamic memory allocation refers to performing manual memory management via a group of functions in the C standard library. The C programming language manages memory statically, automatically, or dynamically.

Static variables are allocated in main memory, along with the executable code, and persist for the lifetime of the program. The automatically managed variables or local variables are allocated on the stack and they come and go as functions are called and exited. Except for the variable-length variable arrays, the size of memory allocation for the static & local variables is defined at compile-time. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate. In this situation the dynamic memory allocation solves the problem, in which memory is more explicitly managed, typically by allocating it in large regions of free space called heap.

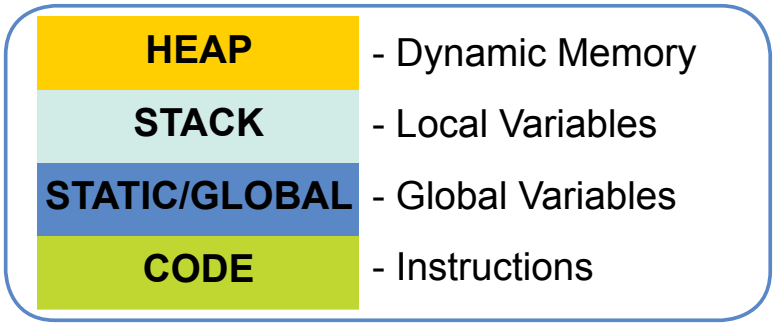


Figure 1 -Application Memory Organization

In other words, heap is a memory region of the processor which the programmer manages manually (in the case of C language). In other programming languages (e.g. Java), memory is managed automatically.

To manage heap memory location in C under Linux, the malloc () & free () functions are used (there are also new () and delete () functions on C++). The malloc function is used for allocating a space into this memory; free is used to deallocate it. In the case of Zephyr™ OS these functions are named k_malloc() and k_free(); these perform the same function as malloc() and free(). For this analysis, the microbenchmark, named heap_bench, was developed around these functions. Its purpose was to measure the time for allocating and deallocating heap memory. Behind the scene, heap_bench allocates 4 bytes (int size) 1000 times and after that it deallocates that space. For the Linux BSP, each loop of allocation and deallocation time is measured using a time_get_time (); this is a wrapper function on top of clock_get_time (). For Zephyr OS, we used the TIMING_INFO_PRE_READ () function.

```
...
for (i = 0; i < ITERATIONS; i++) {
                time_get_time(&start);
                pointer[i]= malloc(sizeof(int));
                *pointer[i] = 0xdeadbeef;
                time_get_time(&stop);
                diff = time_get_diff(&stop, &start);
                total += diff;
}
printf("Only call time function: %.1f ns\n", (total) / (double) ITERATIONS);
...
```

Algorithm 1 – Linux heap allocation

```
...
for (i = 0; i < ITERATIONS; i++) {
                time_get_time(&start);
                free(pointer[i]);
                time_get_time(&stop);
                diff = time_get_diff(&stop, &start);
                total += diff;
 }
printf("Average heap free time: %.1f ns\n", (total) / (double)ITERATIONS);
...
```

Algorithm 2 - Linux heap deallocation

```
...
for (i = 0; i < ITERATIONS; i++) {
                TIMING_INFO_PRE_READ();
                heap_malloc_start_time = TIMING_INFO_OS_GET_TIME();
                pointer[i]= k_malloc(sizeof(int));
                *pointer[i] = 0xdeadbeef;
                TIMING_INFO_PRE_READ();
                heap_malloc_end_time = TIMING_INFO_OS_GET_TIME();
...
```

Algorithm 3 - Zephyr heap allocation

```
...
for (i = 0; i < ITERATIONS; i++) {

                TIMING_INFO_PRE_READ();
                heap_free_start_time = TIMING_INFO_OS_GET_TIME();
                k_free(pointer[i]);
                TIMING_INFO_PRE_READ();
                heap_free_end_time = TIMING_INFO_OS_GET_TIME();
...
```

Algorithm 4 - Zephyr heap deallocation

At the end of the loop, the final time is calculated as an average of all iterations timing. The same thing is done for deallocating the heap memory (see code above).

Thread Creation & Joining Benchmark

In computer science, a thread of execution is the smallest sequence of programmed instructions that an operating system's scheduler can manage independently. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time. Figure 2 depicts two processes, each one having one or multiple threads.

In this comparison, the process is the running benchmark, named thread_bench, which measures the time to spawn 2000 threads using the pthread_create () POSIX function; this time is then divided by 2000 to determine the average time to create a thread.

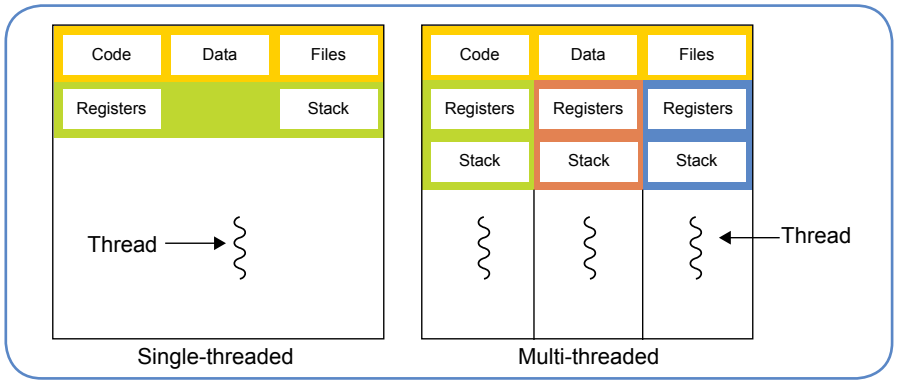


Figure 2- Single Threaded process model vs Multithreaded Process model

```
 ...
for (i = 0; i < ITERATIONS; i++) {
                if (pthread_attr_init(&attr[i]) != 0) {
                        fprintf(stderr, "pthread_attr_init!\n");
                        exit(EXIT_FAILURE);
                }
                if (posix_memalign(&stacks[i], sysconf(_SC_PAGESIZE), MAX_STACK_SIZE) != 0) {
                        fprintf(stderr, "Failed to allocate aligned memory\n");
                        exit(EXIT_FAILURE);
                }
                if (pthread_attr_setstack(&attr[i], stacks[i], MAX_STACK_SIZE) != 0) {
                        fprintf(stderr, "Failed pthread_attr_setstack!\n");
                        exit(EXIT_FAILURE);
                }
                time_get_time(&start);
                if (pthread_create(&threads[i], &attr[i], test_function, NULL) != 0) {
                        fprintf(stderr, "Failed to create thread!\n");
                        exit(EXIT_FAILURE);
                }
                time_get_time(&stop);
#ifdef DEBUG
                fprintf(stdout, "Created thread_id %d\n", i);
#endif
                diff = time_get_diff(&stop, &start);
                total += diff;
        }
fprintf(stdout, "Average pthread_create time: %.lf ns\n", (total/(double)ITERATIONS));
...
```

Algorithm 5 - Linux thread creation

The thread_bench microbenchmark also measures the join time using the pthread_join () function; this function synchronizes the parent thread by pausing its execution, until the child thread ends.

```
...
for (i = 0; i < ITERATIONS; i++) {
                if (threads[i]) {
                        time_get_time(&start);
                        if (pthread_join(threads[i], NULL) < 0) {
                                fprintf(stdout, "Failed to join thread\n");
                                exit(EXIT_FAILURE);
                        }
                        time_get_time(&stop);
                        diff = time_get_diff(&stop, &start);
                        total += diff;
#ifdef DEBUG
                        fprintf(stdout, "thread %d, joined\n", i);
#endif
                }
        }
        fprintf(stdout, "Average pthread_join time: %.1f ns\n",(total/(double)ITERATIONS));
...
```

Algorithm 6 - Linux thread joining

Mutex Lock & Unlock Benchmark

Mutual exclusion is a concurrency control method dedicated to prevent race conditions between two or more threads. Race condition is a system behavior where two independent work flows attempt to modify a shared resource being used to generate the system's output. Making a real-world analogy, consider two mechanics (two threads) which jointly assemble a car engine. They assemble the engine in parallel, however some of the subcomponents must be assembled in a specific order to ensure that the engine works properly. Each mechanic has its own parts of the engine to assemble. To be sure that components are mounted in the correct order, each mechanic should have exclusive ownership of the engine during the critical assembly sections. This exclusive ownership is analogous to the mutex lock, where the mutex is the car's engine. Freeing the engine could be associated with mutex unlock.

The mutex lock & unlock benchmark, named mutex_bench, measures the time of these two combined actions 1000 times (then divides by 1000 to determine the average time). To execute mutex lock and unlock, we used pthread_mutex_lock (), pthread_mutex_unlock () functions of the Pthread API library (see code samples below).

```
...
for (i = 0; i < nr_iterations; i++) {
                time_get_time(&start);
                pthread_mutex_lock(&lock);
                time_get_time(&stop);
                delta = time_get_diff(&stop, &start);
                total_lock += delta;
                time_get_time(&start);
                pthread_mutex_unlock(&lock);
                time_get_time(&stop);
                delta = time_get_diff(&stop, &start);
                total_unlock += delta;
}
        fprintf(stdout, "Average time for locking a mutex: %.8f ns\n",
                        (double) total_lock/ (double) nr_iterations);
        fprintf(stdout, "Average time for unlocking a mutex: %.8f ns\n",
                        (double) total_unlock/ (double) nr_iterations);
...
```

Algorithm 7 - Linux mutex lock & unlock

```
 ...
for (i = 0; i < nr_iterations; i++) {
                  TIMING_INFO_PRE_READ();
                  mutex_lock_start_time = TIMING_INFO_OS_GET_TIME();
                  pthread_mutex_lock(&lock);
                  TIMING_INFO_PRE_READ();
                  mutex_lock_end_time = TIMING_INFO_OS_GET_TIME();
   total_lock +=  (mutex_lock_end_time -mutex_lock_start_time);
                  TIMING_INFO_PRE_READ();
                  mutex_unlock_start_time = TIMING_INFO_OS_GET_TIME();
                  pthread_mutex_unlock(&lock);
                  TIMING_INFO_PRE_READ();
                  mutex_unlock_end_time = TIMING_INFO_OS_GET_TIME();
   total_unlock +=  (mutex_unlock_end_time -
                        mutex_unlock_start_time);
...
```

Algorithm 8 - Zephyr mutex lock & unlock

Context Switching Benchmark

A context switch is the process of storing the state of a thread's process, so that it can be restored and then resume the execution from the same point. This allows multiple processes to share a single CPU and is an essential feature of a multi-tasking operating system.

The precise meaning of the phrase "context switch" varies significantly in usage. In a multitasking context, it refers to the process of storing the system state for one task, allowing that task to be paused and another task resumed. A context switch can also occur as the result of an interrupt, such as when a task must access disk storage, freeing up CPU time for other tasks. Some operating systems also require a context switch to move between user mode and kernel mode tasks. The process of context switching can have a negative impact on system performance, although the size of this effect depends on the nature of the switch being performed.

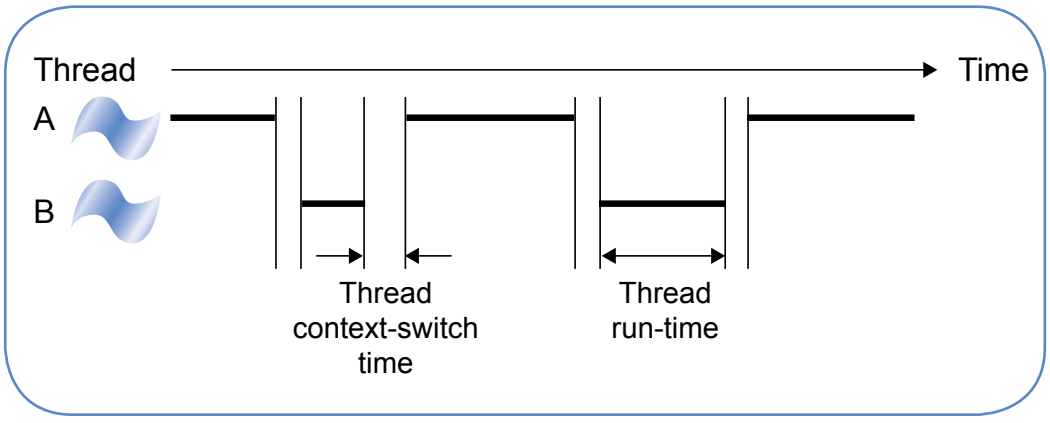


Figure 3 Thread context switch

The context_switch benchmark measures the context switch time by creating two threads, which are continuously context switched for 500,000 times. During the context switch time, [benchmark name] records the elapsed time, then divides by 500,000 to determine the average time.

Analysis results

Collected data

The charts below show the scores reported by the aforementioned benchmarks. Three iterations were done for each benchmark. The results using Zephyr OS are deterministic; each benchmark execution on Zephyr OS with i.MX RT1050 EVK produces the exact same results.

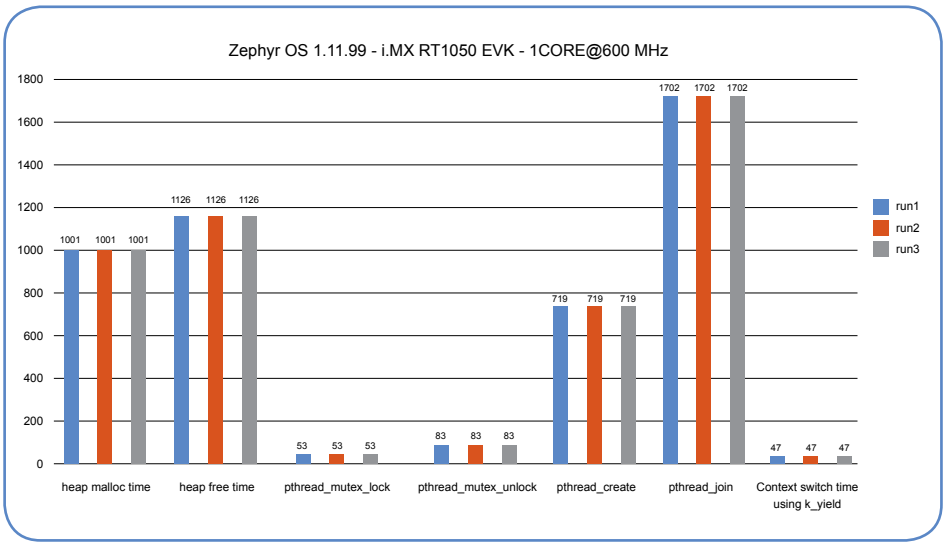


Figure 4 Benchmarks results on i.MX RT1050 EVK with Zephyr OS

With the Linux BSP running on i.MX 6UL, the results varied from run to run by up to 9% deviation from the average.

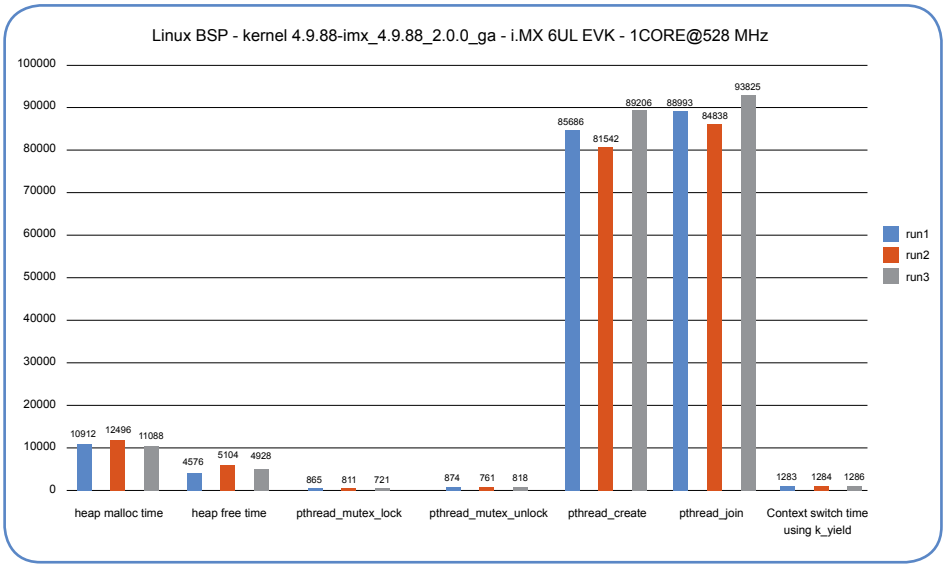


Figure 5 - Benchmarks results on i.MX 6UL EVK with Linux BSP

The table below highlights the average time calculated from these benchmark iterations.

OS	Zephyr OS 1.11.99	Linux BSP 4.9.88-imx_4.9.88_2.0.0_ga	Difference (x times)
Board Name	i.MX RT1050 EVK	i.MX 6UL EVK	-
CPU Cores	1	1	-
Core Frequency (MHz)	600	528	-
Average heap malloc time (cycles)	1001	11499	11x
Average heap free time (cycles)	1126	4870	4x
Average pthread_mutex_lock time(cycles)	53	799	15x
Average pthread_mutex_unlock time(cycles)	83	818	10x
Average pthread_create time (cycles)	719	85478	118x
Average pthread_join time (cycles)	1702	89219	52x
Average Context switch time (cycles)	47	1284	27x

The average time is calculated in cycles (lower is better).

According to this data, the Zephyr OS running on the i.MX RT1050 presented a significant improvement in all time cycles as compared to the Linux BSP + i.MX 6UL EVK.

Conclusions

1. The performance analysis was done by running custom microbenchmarks on two different hardware & software platforms.
2. Benchmarks were developed around a common API to have comparable results.
3. Different functions were used for collecting the elapsed time: clock_get_time() on Linux and TIMING_INFO_PRE_READ on Zephyr OS.
4. As compared to the Linux BSP and i.MX 6UL solution, the Zephyr OS and i.MX RT1050 EVK solution is:
   a. 27x times faster in context switching
   b. Up to 11 x faster in allocating, deallocating memory
   c. Up to 15 x faster in locking & unlocking mutexes using pthread library
   d. Faster in creating, joining and canceling threads
   e. Provides more performance at a lower cost
5. The Zephyr OS with i.MX RT1050 EVK board presents a predictable execution time, offering the possibility to be used in applications that require various time constrains.

How to Reach Us:

Home Page: **www.nxp.com**
Web Support: **www.nxp.com/support**

USA/Europe or Locations Not Listed:
NXP Semiconductors USA, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.nxp.com/support

Europe, Middle East, and Africa:
NXP Semiconductors Germany GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.nxp.com/support

Japan:
NXP Japan Ltd.
Yebisu Garden Place Tower 24F,
4-20-3, Ebisu, Shibuya-ku,
Tokyo 150-6024, Japan
0120 950 032 (Domestic Toll Free)
www.nxp.com/jp/support/

Asia/Pacific:
NXP Semiconductors Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@nxp.com

Document Number: ZEPHYRIMXRTWP REV 0