Proceedings of the ASME 2022
International Mechanical Engineering Congress and Exposition
IMECE2022
October 30–November 3, 2022, Columbus, OH

IMECE2022-95438

FLEXIBLE LOW-LEVEL CONTROL SOFTWARE FRAMEWORK FOR ACHIEVING CRITICAL REAL-TIME DEADLINES

Nicholas J Tremaroli[1,†], Maxwell A Stelmack[1,†], Connor W Herron[1,†,*], Bhaben Kalita[1], Alexander Leonessa[1]

[1]Terrestrial Robotics Engineering and Controls (TREC) Laboratory, Virginia Tech, Blacksburg, VA 24060, USA

ABSTRACT

In this work, a low-level software framework is proposed to simplify software development for Hardware Abstract Layered (HAL) control systems, identify networking methods for accurate real-time communication between devices, and verify task completion. The framework is implemented on a distributed microcontroller system composed of Texas Instruments TM4C123GXL Tivas for a multi-joint robot. The robot's high-level controller executes dynamic motion control algorithms, with low-level controllers responsible for each individual joint. All microcontroller software is unified into one program and uses initialization files from the high-level controller to configure each individual Tiva depending on its location on the robot. The EtherCAT communication protocol is utilized to avoid unnecessary overhead from traditional networking protocols. A real-time operating system, TI-RTOS, enforces crucial deadlines and provides powerful diagnostic tools for the designer to optimize task completion. Overall, our proposed framework overcomes the major challenges of writing low-level control software so that development is less time-consuming, simpler to manage, and easier to validate. Further, this work can be used for many kinds of robotic systems and applications that use microcontrollers within a multi-layered control architecture.

Keywords: Software Framework, Networking, Microcontroller, Control System, Communication, Operating System

1. INTRODUCTION

Complex robotic systems are becoming more flexible by utilizing distributed control approaches which separate high and low-level controllers into different pieces of hardware, thus allowing faster execution rates for high-speed control. For multi-joint robotic systems such as manipulators, exoskeletons, and humanoid robots, microcontrollers are often paired with robot joints to handle sensor collection and actuation, allowing for persistent software at the low-level [1–4]. These distributed control systems are scalable and allow for increased controller speeds and microcontroller software unification, but require timing guarantees from i) low-level processor capability and ii) networking for high-performing, robust robot control. Xi et al. [5] discussed the importance of using a Real-Time Operating System (RTOS) in the low-level system, emphasizing that distributed control systems must properly evaluate task completion for effective performance. Cui and Park [6] argued that achieving complex dynamic behavior for robots is throttled by the lack of in-robot network (IRN) architectures to ensure reliable data transmission across all sensors and actuators, where the most popular networking methods for robotic systems are the CAN and EtherCAT protocols. These networking methods have been utilized in applications such as an autonomous underwater robot [7], a wheeled soccer robot [8], a humanoid robot [9], a multi-axis robot arm [10], and a collaborative manipulator [5].

In this work, a low-level software framework is proposed for multi-joint robotic systems utilizing RTOS and the EtherCAT networking protocol and is designed to meet the specific challenges of a robotic system. Low-level controllers are often implemented using single-threaded microcontrollers, only capable of executing a single task at one time. Microcontrollers are inherently opaque, and their internal states cannot be easily observed during runtime. Traditional networking strategies between microcontrollers and high-level processors are inefficient due to unnecessary overhead procedures [11]. Hardware Abstract Layered (HAL) programming differs between microcontrollers, making it tedious and time-consuming to transition between platforms. Multi-layer controlled robotic systems require a low-level framework with timing guarantees for network communication and task execution to ensure good performance and safe action [6]. The developed framework for this research, addresses each of these challenges and constraints.

The paper is organized as follows: Section II describes the code unification, networking, and operating system within the proposed Low-Level Software framework, Section III details the results which verify networking efficacy and low-level processor capability, and Section IV is the Conclusion.

†Joint first authors
*Corresponding author: cwh@vt.edu

Copyright © 2022 by ASME

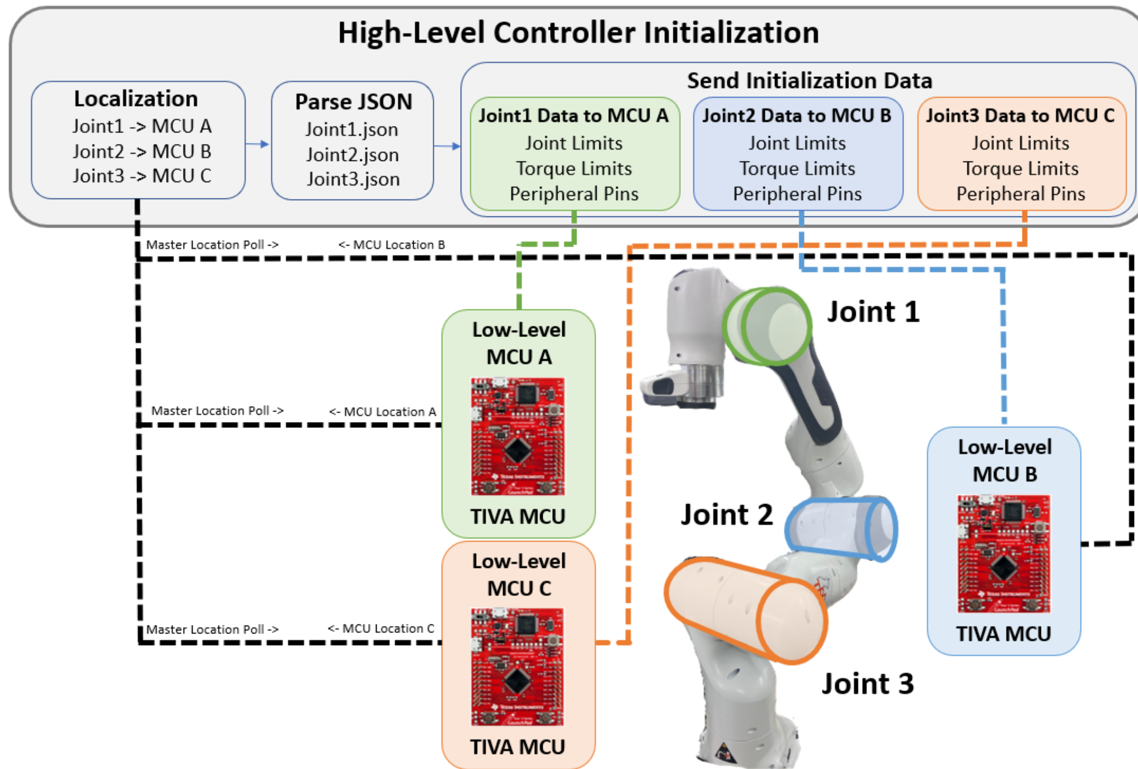


FIGURE 1: INITIALIZATION PROCESS

2. SOFTWARE FRAMEWORK

The low-level software framework is developed for the TM4C123GXL TIVA Launchpad as shown in Fig. 1. The TIVA microcontroller is an 80 MHz ARM Processor with embedded communication interfaces such as I2C, UART, CAN, and SSI, along with I/O modules such as GPIO, Timers, PWM, QEI, and ADCs [12]. These modules are utilized for collecting sensors feedback, directing actuator commands, and networking with the high-level controller. The proposed framework leverages several techniques to solve issues related to low-level software development on microcontrollers. Subsection I provides a low-level initialization procedure to unify behavior across distributed microcontrollers. Subsection II explains the EtherCAT networking implementations developed for each mode of microcontroller operation. Subsection III describes a real-time operating system implemented for the code of the low-level controller to provide better timing guarantees. Subsection IV lists the steps for implementing low-level controller software within the proposed framework.

2.1 Low-Level Initialization

In a distributed robotic control system, each low-level microcontroller is responsible for a particular joint as shown in Fig. 1. A microcontroller (MCU) requires different software depending on the assigned joint's sensor configuration and desired functionality. To avoid writing multiple programs with each joint's parameters hard-coded, an initialization process is used to unify the code of the distributed controllers.

Identical code is flashed to MCUs A, B, and C, with the high-level computer sending an initialization file to each microcontroller with its joint-specific parameters. This file includes enabled peripherals, sensor calibration values, and joint and torque limits. Simply from editing the initialization file, the software behavior of individual MCU-joint pairs can be altered without re-deploying code to the low-level device.

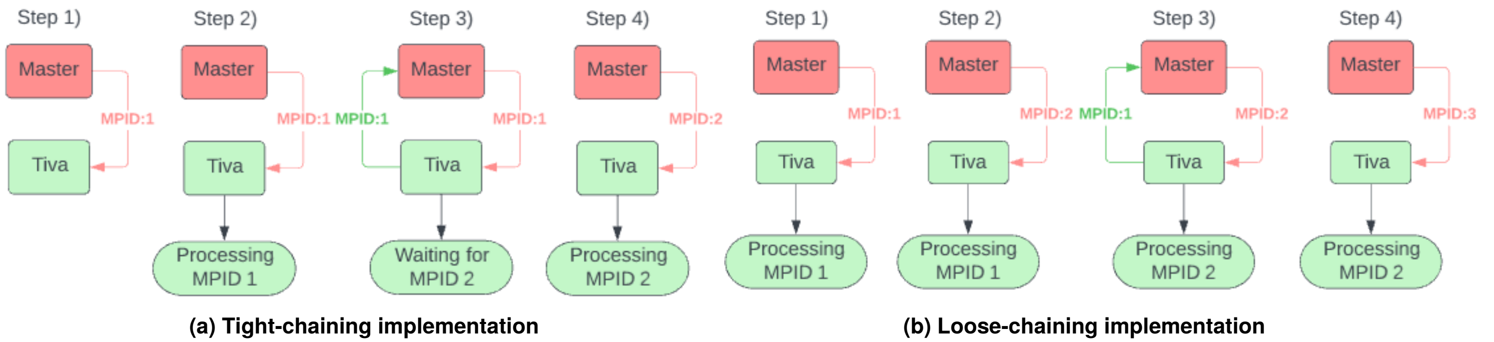
The high-level computer stores all of these initialization files in a JSON format. Before the sending an initialization file to a microcontroller, the high-level computer must know the associated joint. Before initialization, the high-level computer queries all low-level device locations. Once each microcontroller's joint is identified, the respective JSON initialization file is parsed and sent by the high-level computer.

A microcontroller stores each initialization frame from the high-level in a dynamically allocated array which is only used for this purpose. Once all initialization data from the high-level is received by the low-level, the dynamically allocated data structure is parsed and the peripherals on the microcontroller are made ready for operation. After initialization completes, the dynamically allocated initialization array is deallocated from memory to save space.

2.2 Networking

Traditional Ethernet-based networking protocols for a distributed microcontroller system tend to cause inefficiencies and unnecessary delays. These delays are caused by the nature of Ethernet-based routing protocols such as IP and ARP. These traditional protocols are much better suited for a dynamic network in which the state of the devices on the network can change. However, control-automated technologies such as robotics have

Copyright © 2022 by ASME



(a) Tight-chaining implementation **(b) Loose-chaining implementation**

FIGURE 2: NETWORKING METHODS UTILIZED FOR HIGH AND LOW-LEVEL COMMUNICATION.

static networks, where connected devices are not changing during runtime. Therefore, network flexibility protocols are not needed and cause unnecessary slowdown.

The EtherCAT protocol is designed for this very reason. EtherCAT uses a much more minimalistic approach to networking which does not have as much overhead. By using EtherCAT, the unnecessary protocols which stem from traditional networking are avoided and thus the communication loop is able to run faster [11]. This speed gained from using the EtherCAT protocol can be crucial when designing systems which require fast update rates. The EtherCAT network topology is significantly different from that of Ethernet. For Ethernet, all of the data for each device is routed through a central hub such as a switch, with the switch connected to each of the individual slaves. EtherCAT by default uses a ring topology, in which all of the slaves are connected to each other and then back to the master. From this configuration, the master can send out a single data frame to send and receive updated values from each of the low-level controllers instead of sending out multiple requests, as in a traditional Ethernet network. The authors' implementation uses the EasyCAT Pro as an EtherCAT shield. Each microcontroller has an attached EasyCAT Pro to send data back to the master as part of the ring topology chain. The EasyCAT Pro uses the LAN9252 EtherCAT slave controller chip [13]. A custom driver is written to communicate with this chip over SPI to the TIVA. The EasyCATs are configured to receive and send 32 bytes of data per communication loop. These 32 bytes of data are programmed to be serialized differently depending on which command signal the master is sending, which are specified below:

- **HALT:** Triggers the low-level controller's emergency-stop and turns motors off.
- **LOCATION DEBUG:** Queries the Low-Level Controller's location on the robot.
- **CONTROL:** Commands the Low-Level Controllers to move to a desired location. The Low-Level Controllers then sends the master the most updated sensor values.
- **IDLE:** Puts the Low-Level Controllers in standby.
- **INITIALIZATION:** Sends initialization data the microcontroller needs before it can start.

The EtherCAT communication loop, in general, can run as fast as the master computer (high-level controller) would like. However, if the master runs the communication loop faster than the microcontrollers can process, frames will be skipped. Thus, two different EtherCAT implementations were developed as shown in Fig. 2, where each implementation has its own respective benefit for certain master commands. For initialization-based command signals, the "tight-chaining" implementation is utilized to guarantee two-way communication without loss of data. For processing control signals, the "loose-chaining" implementation is utilized to achieve faster speeds at the expense of occasional frame drops.

2.2.1 Tight-Chaining Implementation For Initialization. As for initialization, since every single frame is important, skipping a frame is catastrophic. Each frame the master computer sends out has an ID which the low-level device must echo back before the master sends the next frame. This ID number is called the master process ID (MPID).

From the microcontroller's perspective, if the MPID is an updated value, the EtherCAT frame contains new, unprocessed data which must be interpreted. If the MPID is the same as one in a previously processed frame, then the microcontroller ignores the frame and waits for an updated MPID. From the master's perspective, the master computer sends out initialization data with a particular MPID. The master computer then waits for the microcontroller to echo back the same MPID before it can send out another initialization frame.

This implementation of "tight-chaining" forces the master computer to wait for the microcontroller to complete processing before sending out another initialization frame. A demonstration of tight-chaining can be seen in Fig. 2a, in which the master sends out MPID:1 and waits for the microcontroller to process and echo back before sending MPID:2.

2.2.2 Loose-Chaining Implementation For Control. While a tight-chaining implementation may be good for initialization purposes by preventing missed frames, it is not the ideal implementation for sending control signals. Tight-chaining enforces that the master wait for the microcontroller's message before another frame is sent out. However, the microcontroller immediately accepts a new message after echoing the previous MPID. In a tight-chaining implementation, this new message will have the same MPID because the master has not sent out

Copyright © 2022 by ASME

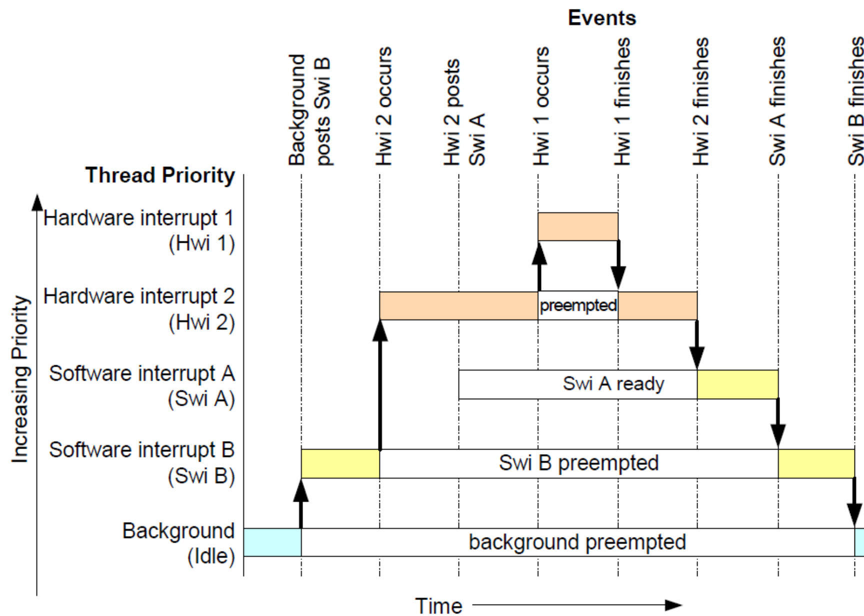


FIGURE 3: TASK PREEMPTION TI-RTOS [14]

a new frame, as the echo it is waiting on is still traveling to it. The end result in a tight-chaining implementation is that the microcontroller and master both wait on each other, which increases latency.

The alternate approach is loose-chaining. In this implementation, the master sends the first frame with an MPID and then immediately follows with a second with a different MPID. The microcontroller processes the first message from the master. When the Tiva echos to the master, it is able to pull the second frame with an updated MPID and begin processing that request. Once the master receives the first MPID from the Tiva, a third frame is sent out from the master with an updated MPID and the cycle continues. In this implementation, the master is always one step ahead of the Tiva and frames can be processed faster as the Tiva is not waiting on the master.

Once MPID:1 is received by the master, the Tiva begins MPID:2 and the master sends out MPID:3. It is important to note the key differences between the tight-chaining and loose-chaining implementations. Tight-chaining enforces that each frame sent by the master is read by the Tiva, making communication slower. During initialization, this approach ensures that each Tiva is fully set up prior to run time. Loose-chaining is faster but is subject to rare frame drops from either the master or the Tiva depending on their update rates. These frame drops happen infrequently and have negligible impact on the system when sending control signal frames.

2.3 Real-Time Operating System (RTOS)

The computational function of a microcontroller can often be broken down into discrete and repeatable segments. In a robotic system, this may take the form of acquiring sensor data, computing mathematical formulas, commanding actuators, and transferring data. A common practice in microcontroller development is to denote each routine functionality as a "task". Many different properties of a task can be described, but the most important are worst-case execution time, period, and deadline. Many microcontrollers do not make use of multiple processors, restricting execution to one task at a time. Choosing which task to run at a particular moment can have an immense effect on system stability (imagine a processor so backlogged by tasks that it cannot command actuators). Efficiently completing tasks before their deadline is the purpose of using a real-time operating system (RTOS).

A microcontroller usually does not feature an operating system. An operating system induces additional computing overhead without much benefit for a typical microcontroller application. Traditionally, routine tasks of a microcontroller are handled through an Interrupt Service Routine (ISR). The approach is to set up a timer for each task aligned with its period, and when that timer expires the task is considered "ready-to-run". When the currently running task completes, the next ready-to-run task with the highest priority executes. Tasks of the same priority run on a FIFO (First-In-First-Out) basis.

There are several issues with an ISR. Once a task begins, it cannot be interrupted. This means a lengthy low-priority task can delay a critical high-priority task. An ISR is not aware of task deadlines and cannot guarantee their satisfaction. On the other hand, a real-time operating system implements a scheduler that manages execution of tasks. A scheduler can use a variety of algorithms to decide which tasks run. A scheduler can force preemption, where a high priority tasks temporarily takes execution time from a lower priority tasks (Fig. 3), often to meet the higher-priority deadline. This allows deadlines to be better enforced for high priority tasks. Tasks of greater importance to robot functioning can be given higher priority to better guarantee their timely completion and a stable system.

Since our implementation of the framework resides on a

Copyright © 2022 by ASME

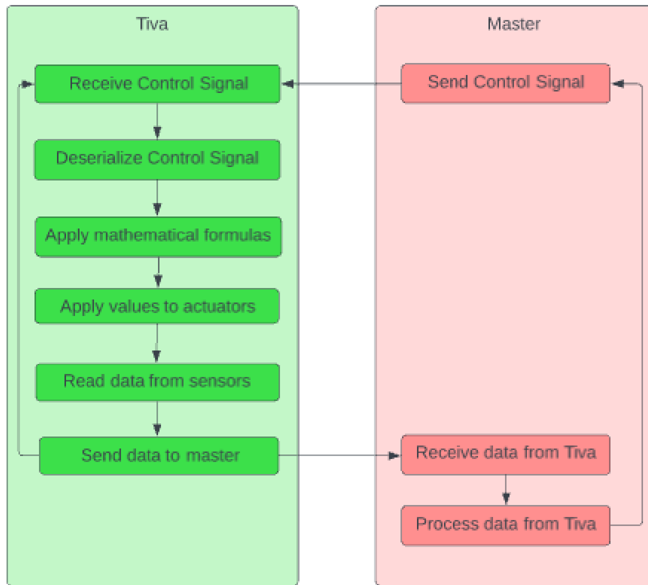


FIGURE 4: CONTROL SIGNAL STEPS

Texas Instruments microcontroller, TI-RTOS is used. This has the additional benefit of providing logging tools for in-depth performance analysis. Through tools like Utilization Analysis and Execution Analysis, the exact processor load and task execution are made visible for validation. TI-RTOS also supplies a HAL for various low-level functionalities, as well as its own configuration. Since a microcontroller has a limited memory size, the TI-RTOS can be configured to disable unneeded features to save space. To govern what aspects of the RTOS are included, a configuration script is added to the software project that builds the operation system to specification. While the configuration file can be edited directly, a GUI can be used to customize most features. The GUI also guides the user through configuration of each module's parameters (logging buffer size, task instance creation, etc).

2.4 Low-Level Controller Process

The control loop begins running when the high-level sends a CONTROL signal. Once a microcontroller enters a control state, its functionality can be broken down in repetitive discrete steps which are essential for normal operation. Reading data from master is the first step of the control loop. For this step, the Tivas must extract the raw EtherCAT frame data from the EasyCAT board and then deserialize the data so that it can be interpreted further. After the data is deserialized, the Tiva must apply mathematical formulas relating to calibration and orientation. Once the data is processed, the Tiva applies the values to its assigned actuator to achieve a desired force. The final step is to read the values from each of the sensors, serialize the data, and send it back to the master. This allows the master to process the changes on the robot as they are happening in real-time. Each of the individual responsibilities of the microcontroller is implemented as a task within the real-time operating system. These tasks are assigned a priority based on their criticality to the system.

```
Number of frames dropped in a loose-chaining implementation
Frame Missed at: 0.02s
Frame Missed at: 1.64s
Frame Missed at: 3.02s
Frame Missed at: 4.45s
Frame Missed at: 6.02s
Frame Missed at: 7.65s
Frame Missed at: 9.04s
A total of 7 frames got dropped out of 10142 frames in 10 seconds
```

FIGURE 5: LOOSE-CHAINING RESULTS

```
Number of frames dropped in a tight-chaining implementation
A total of 0 frames got dropped out of 5012 frames in 10 seconds
```

FIGURE 6: TIGHT-CHAINING RESULTS

3. RESULTS AND DISCUSSION

This section provides analyses of the authors' implementation of the framework components outlined in Section II. Results are shown which validate the networking methods of loose and tight-chaining and further demonstrate their efficacy for different operation scenarios. The RTOS Execution Analysis tool is utilized to present for managing microcontroller tasks. An example with encoder and controller tasks outlines the functionality of RTOS for validation of timing constraints.

3.1 Networking Implementation

The networking implementations used throughout this framework maximized efficiency when communicating with the low-level controllers. The use of EtherCAT and the various EtherCAT implementations reveal idle methods of communication per each command the master wanted to send to the low-level controllers. With the incorporation of EtherCAT and the various implementations described, the control loop of the system is able to be run successfully at 1KHz with minimal frame drops during a control signal. From the tight-chaining implementation, the Tivas have never had a fault related to initialization and are always able to fully processes every frame the master sends out. The loose-chaining implementation increases the update rate of the EtherCAT loop for control signals. These implementations are tested using a program designed specifically to identify frame drops in the EtherCAT communication network. The output of this program can be seen in Fig. 5 and Fig. 6 for loose-chaining and tight-chaining, respectively.

By preemptively sending one frame ahead of a Tiva's processing capabilities, we are able to maximize the processing output of the microcontroller and achieve faster update rates. It is also observed that the frame drops on the EtherCAT network which occurs from the loose-chaining implementation only once, every 1.5 seconds. This occurrence is deemed to be negligible for our purposes, as the update rate is significantly higher to compensate for occasional frame drops.

3.2 TI-RTOS Timing Evaluation

To evaluate the effects of an RTOS implementation on the timing of critical tasks, the Execution Analysis Tool inside of TI-RTOS's debugging mode is used to collect data and generate graphs. A bar is drawn in the row of the executed process for a particular time and is color-coded. The execution graph seen in Fig. 9 is zoomed in on a single tick of the kernel clock.

Copyright © 2022 by ASME

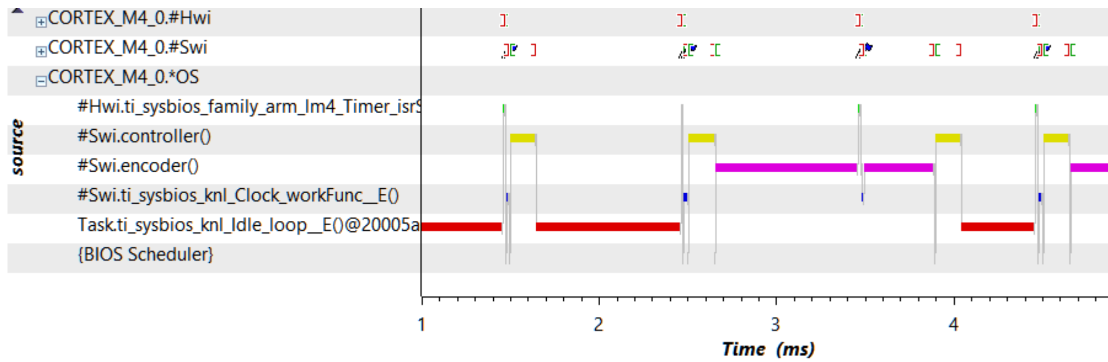


FIGURE 7: PSEUDO-ISR EXECUTION

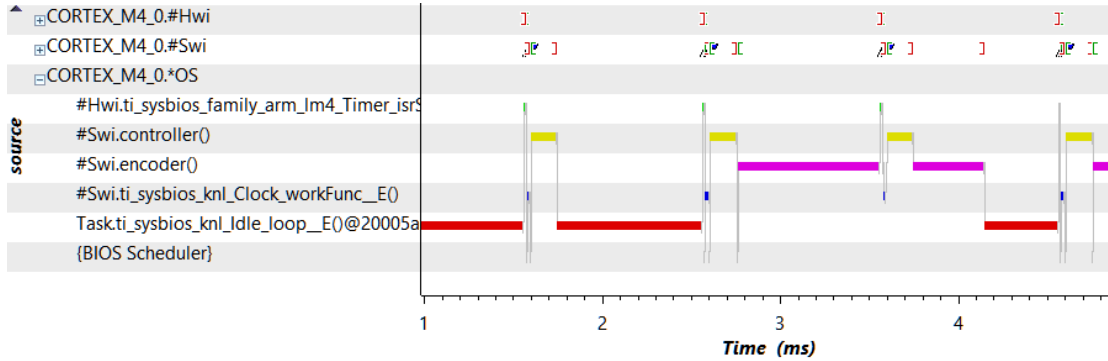


FIGURE 8: TI-RTOS PRIORITIZATION OF MULTIPLE TASKS

The "clock tick" is a periodic software function (in this case the period is 500 µs) that is the base unit for all timing. Every task's period is expressed in ticks rather than continuous time units. Figure 9 depicts the system being in the idle function (red) leading up to the tick, the tick counter incrementing (green), the scheduler dictating which process runs next (lower blue), the callback functions for expired timers executing (upper blue), and a return to the idle state after callbacks are resolved (red). This routine set of functionality occurs quickly in comparison to other tasks that it will be scaled to a single vertical bar in other graphs.

An example task set demonstrates the effects of RTOS are summarized in Table 1. These tasks are not representative of our system performance and are exaggerated to make effects on timing more apparent. The controller task represents a low-level controller algorithm implemented on the system. The encoder represents the processing required to obtain a value from a single sensor where there would be several sensors to be measured for a robot joint.

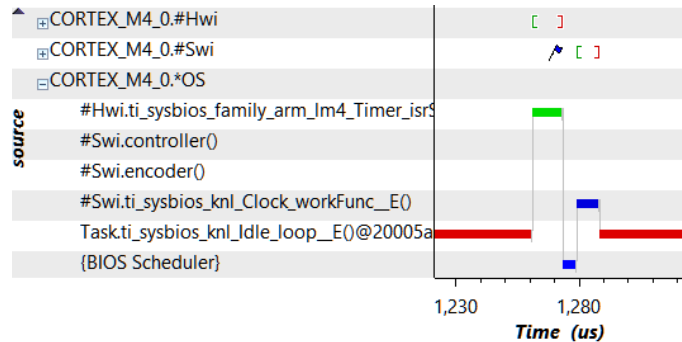


FIGURE 9: TI-RTOS PRIORITIZATION AND PREEMPTION

In Fig. 7, the controller task is color-coded yellow, and the encoder task is shown as magenta. The two tasks are set to the same priority, replicating ISR execution where tasks cannot preempt each other. At the first tick (1.4 ms), the controller is run successfully to completion. At the second tick, the controller and encoder are both posted. The controller runs to completion and the scheduler calls the encoder task next. Due to this task's long execution time, it has not finished before the third tick. The third tick makes the controller ready to run, but the scheduler waits for the encoder task to run to completion before executing the controller. This pattern repeats itself, with the controller running at an inconsistent timestep.

On the other hand, Fig. 8 depicts a controller task with a higher priority than the encoder task. This allows the controller to preempt the encoder task after the third tick to maintain a far more consistent timestep. Thus, consistent timing can occur with far greater surety in a real-time environment where tasks

TABLE 1: EXAMPLE TASK SET

Task	Avg. Execution Time [µs]	Period [ticks]
Controller	143	1
Encoder	1223	2

Copyright © 2022 by ASME

can be prioritized by criticality. Furthermore, advanced logging tools provide greater insight into actual system performance and enhance empirical verification of timing constraints.

4. CONCLUSION

This research proposed a software framework to surmount difficulties commonly faced by developers of low-level controllers. Combining robust initialization procedures for distributed microcontrollers, efficient protocols for static networks, and a real-time operating system for superior timing guarantees, this approach overcomes various challenges unique to microcontroller programming. The low-level software framework can be utilized for many kinds of multi-joint robotic systems. Furthermore, this work can be extended to multi-agent systems containing several robots with more complicated leadership hierarchies where there may be many high-level controllers.

REFERENCES

[1] Hopkins, Michael A, Hong, Dennis W and Leonessa, Alexander. "Humanoid locomotion on uneven terrain using the time-varying divergent component of motion." *2014 ieee-ras international conference on humanoid robots*: pp. 266–272. 2014. IEEE.

[2] Sakagami, Yoshiaki, Watanabe, Ryujin, Aoyama, Chiaki, Matsunaga, Shinichi, Higaki, Nobuo and Fujimura, Kikuo. "The intelligent ASIMO: System overview and integration." *IEEE/RSJ international conference on intelligent robots and systems*, Vol. 3: pp. 2478–2483. 2002. IEEE.

[3] Infante, Y. Y., Blanco, A. E. and Jonguitud, A. E. "The exoskeleton: Operation with electrostimulators for rehabilitation." *2019 IEEE International Conference on Engineering Veracruz (ICEV)*, Vol. I: pp. 1–4. 2019. DOI 10.1109/ICEV.2019.8920521.

[4] Yang, Yong, Ma, Lei and Huang, Deqing. "Development and Repetitive Learning Control of Lower Limb Exoskeleton Driven by Electrohydraulic Actuators." *IEEE Transactions on Industrial Electronics* Vol. 64 No. 5 (2017): pp. 4169–4178. DOI 10.1109/TIE.2016.2622665.

[5] Xi, Qiang., Zheng, Chang W., Yao, Mao Y., Kou, Wei. and Kuang, Shao L. "Design of a Real-time Robot Control System oriented for Human-Robot Cooperation." *2021 International Conference on Artificial Intelligence and Electromechanical Automation (AIEA)*: pp. 23–29. 2021. DOI 10.1109/AIEA53260.2021.00013.

[6] Cui, Chengyu and Park, Sungkwon. "In-Robot Network Architectures for Humanoid Robots With Human Sensor and Motor Functions." *IEEE Access* Vol. 9 (2021): pp. 89325–89335. DOI 10.1109/ACCESS.2021.3082143.

[7] Li, He, Yin, Bo, Wang, Shanshan and Yang, Qingshu. "Design of underwater robot controller based on CAN bus." *Proceedings of 2011 International Conference on Electronic Mechanical Engineering and Information Technology*, Vol. 9: pp. 4906–4909. 2011. DOI 10.1109/EMEIT.2011.6024063.

[8] Darmaji, Darmaji, Wibowo, Iwan Kurnianto, Ardilla, Fernando and Ibadurrohman, Dliyauddin. "Hardware Architecture For Robot Soccer ERSOW Using Control Area Network Bus." *2019 International Electronics Symposium (IES)*: pp. 464–468. 2019. DOI 10.1109/ELECSYM.2019.8901662.

[9] Ressler, Stephen. "Design and Implementation of a Dual Axis Motor Controller for Parallel and Serial Series Elastic Actuators." BS Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA. 2014.

[10] Li, Xiang, Ma, Xudong and Song, Wenbin. "Multitasking Syetem Design for Multi-axis Synchronous Control of Robot Based on RTOS." *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*: pp. 356–360. 2020. DOI 10.1109/ICIEA48937.2020.9248319.

[11] Beckhoff. "EtherCAT - the Ethernet Fieldbus." Beckhoff Automation, Verl, Germany (1980). Accessed April 22, 2022, URL https://www.beckhoff.com/en-us/products/i-o/ethercat/.

[12] Texas Instruments, Austin, TX. *Tiva TM4C123GH6PM Microcontroller Data Sheet*, Release Rev V (2014). URL https://www.ti.com/lit/ds/symlink/tm4c123gh6pm.pdf.

[13] "EasyCAT PRO." AB&T, Ivrea, Italy (2014). Accessed April 22, 2022, URL https://www.bausano.net/en/hardware/easycat-pro.html.

[14] Texas Instruments, Cary, NC. *TI-RTOS Kernel (SYS/BIOS) User's Guide*, Release Rev V (2020). URL https://www.ti.com/lit/spruex3.

[15] Voss, Wilfried. "A Brief Introduction to Controller Area Network." Copperhill Technologies (2022). Accessed April 22, 2022, URL https://copperhilltech.com/a-brief-introduction-to-controller-area-network/.

[16] "CANopen or EtherCAT: Discover All Features of ESA EDWU Drive." Esa Automation (2019). Accessed April 22, 2022, URL https://www.esa-automation.com/en/canopen-or-ethercat-discover-all-features-of-esa-edwu-drive/.

Copyright © 2022 by ASME