Github Repo: https://github.com/NaomiGonz/RoboArm/tree/main

Youtube Video: https://youtu.be/PB8VNwyJbrM

RoboArm: Robotic Path-Planning and Optimization on an Embedded Processor

1. Abstract

Robotic path planning and control algorithms can be easily run on desktop computers in “unconstrained” settings. However, deploying these methods in embedded systems with real-world resource constraints presents unique challenges. This project set out to design and implement a full embedded robotic control pipeline using the Rapidly-exploring Random Tree Star (RRT*) algorithm on a BeagleBone MCU. The system integrated path planning, forward kinematics, and collision detection to control a 5-degrees-of-freedom (DOF) robotic arm navigating a simulated cluttered environment. Joint positions were computed in C++, transmitted using serial communications in JSON format, and actuated in hardware. Performance was profiled using gprof, and revealed computational bottlenecks. This further led to an exploratory study of fixed-point conversion and hardware acceleration via a novel DAISY toolchain developed by researchers at Boston University. Although fixed-point optimization showed minimal gains on the BeagleBone (due to its FPU), the approach has potential for MCU platforms lacking floating-point hardware, and for acceleration using FPGAs. This work highlights the feasibility and limitations of deploying sampling-based motion planning algorithms in constrained, real-time embedded systems.

2. Introduction

Robotic motion planning is an important capability across a broad range of industrial sectors, including automated manufacturing, space robotics, surgical robotics, etc. For self-contained systems, and systems operating in austere environments in particular, real-world embedded constraints apply, including limited compute, power, and real time demand. Implementing complex motion planning algorithms on an embedded CPU can pose challenges, given limited CPU performance, potential lack of floating point hardware, strict latency and timing constraints, and the need for robust communications and actuation interfaces.

With this background in mind, this project set three core goals, relating to applied robotics:

1. Implement a complete embedded path-planning pipeline using RRT*, integrating forward kinematics and obstacle avoidance
2. Demonstrate real-time control of a robotic arm
3. Profile and evaluate algorithm performance on embedded hardware
4. Explore optimization and acceleration options for performance bottlenecks

These are all important steps in moving towards deployable, low-cost and effective autonomous systems. In a nutshell, this project sought to develop and implement commonly used robotics algorithms under realistic hardware constraints, and explore options to optimize for effective operations.

RRT* (Rapidly-exploring Random Tree Star) is a sampling-based path planning algorithm that builds a tree from a starting point, and expands randomly by selecting new nodes. It differs from standard RRT in that it rewires the tree dynamically to find optimal paths. The algorithm is widely used in several applications, including robotic path planning. Forward kinematics, used here to translate robotic joint angles to cartesian coordinates, is widely used, and critical for determining whether a candidate arm configuration collides with any obstacles.

Both algorithms are computationally expensive, especially in systems with many degrees of freedom. To this end, their performance can be optimized using a variety of methods, including use of optimization flags to maximize code performance, use of fixed-point arithmetic to make use of cheaper hardware, and hardware acceleration of computationally expensive elements. This project assessed algorithmic performance using optimization flags as well as a novel high-level synthesis toolchain being developed by researchers at Boston University's CS Robomorphic Computing Lab. For this project we also received guidance from BU researcher Alp Eren Yilmaz, who kindly provided us with advice regarding RRT implementation, a robotic arm to test our implementation on, as well as access to a previously developed forward kinematics algorithm, and the associated optimized toolchain outputs.

The remainder of this report is organized as follows: Section 3 describes the methodology, including RRT*, forward kinematics, obstacle detection, and hardware acceleration workflow. Section 4 presents the study's results. Section 5 outlines limitations and potential future work.

3. Method

3.1. Contribution Overview

Project Element	**Contributions**
RRT* Implementation	Equal contributions by Naomi Gonzalez and Gidon Gautel, with guidance from Alp Eren Yilmaz
RRT Visualisations	Naomi Gonzalez
Hardware Set-up	Robotic arm provided by Alp Eren Yilmaz. Serial interface implemented by Naomi Gonzalez and Gidon Gautel.
Forward Kinematics	Provided by Alp Eren Yilmaz.
Fixed-point Optimized Forward Kinematics	Provided by Alp Eren Yilmaz.
Profiling and Data Collection	Gidon Gautel

3.2. Hardware

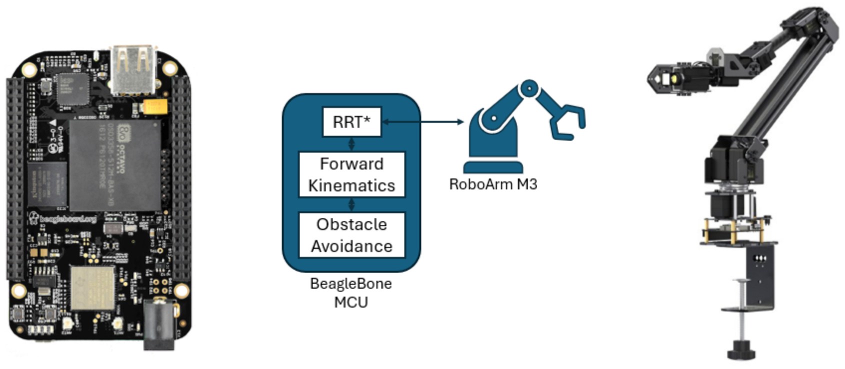

Figure 1. Hardware set-up

The hardware set-up used consisted of a BeagleBone Black and a Waveshare RoboArm M3[1], a 5 DOF robotic arm for research and development applications. The RoboArm M3 has two USB-C ports, and serial communications were used to send joint angles from the BeagleBone to the arm to execute.

3.3. Software

3.3.1. Embedded Operating System

Our project required a linux kernel image that would allow us to configure the BeagleBone's IO ports, as well as profile our C++ code, and compile both C++ and python. To this end, we used Balena Etcher and the provided SD card to flash the BeagleBone with an AM335x Debian 11.7 IoT image, found on BeagleBoard.[2] This provided the required functionality, and was appropriate for these applications, as a lightweight build optimized for the AM335x, designed for headless and minimal GUI operation. Debian 11 is also stable and widely supported.

3.3.2. RRT*

3.3.2.1. Theory

RRT* is a sampling-based motion planning algorithm that is commonly used in robotics for pathfinding. It builds on RRT, which is fast but often produces suboptimal paths.

RRT grows a tree from a starting configuration by sampling random points in an n-dimensional space. For each sampled point, the nearest node in the tree is found via a distance metric (in our case square distance). A new node is then steered towards the sampled point, limited by a preset step size. If the new node is collision free, it is added to the tree. The tree is grown in this manner until a path to the desired goal is found, with the goal considered "reached" when a point is sufficiently close. A goal bias is used, which biases sampling of new points towards the goal with a probability p (in our case 0.5).

RRT* builds on this approach by adding a rewiring step. This checks nearby nodes within a given radius, to test whether connecting through a new node yields a lower-cost path. In our

[1] Vendor site: https://www.waveshare.com/roarm-m3.htm

[2] Debian Image site: https://www.beagleboard.org/distros/am335x-debian-11-7-2023-08-05-4gb-emmc-iot-flasher

application, the cost is modelled as Euclidean distance. If so, parent connections are rewired in order to take the more optimal path.

3.3.2.2. Implementation

RRT* was implemented in C++. We began first by implementing RRT Base classes, which defined the generic data structures and methods needed. These were implemented to be dimension-agnostic, so that dimension-specific implementations could be built on top of it. Specifically, the base code comprised three elements:

1. **Configurations** - a simple vector to store point/node coordinates
2. A **Node** class and associated methods - to represent single points in the RRT tree, where connections between nodes are defined by parents children
3. An **RRT* Base** class with methods to build the RRT tree, including methods to sample configurations, as well as methods to implement the 'star' approach and rewire the tree to an optimized path.

With the base class implemented, we proceeded first by testing it in two dimensions by implementing an rrt_2d class. This implemented dimension-specific methods that performed functions such as distance measurement, steering, sampling, and collision detection in two dimensions.

Once it was confirmed that our codebase performed well in two dimensions, we proceeded with developing a 3D implementation that could be used to control our robotic arm. This rrt_3d class built on our previous work, however differed in that it was implemented primarily in joint space. Specifically, the tree in this case was grown in five-dimensional joint space. Forward kinematics was then used to translate joint configurations into cartesian space. Collision detection was then conducted in cartesian space, and the results fed back to the algorithm in order to determine whether a sampled point was valid or not, and could be added to the tree.

For both 2D and 3D implementations, methods were added to extract the final path by backtracking through the tree from the goal node using parent pointers.

3.3.3. Forward Kinematics

Forward kinematics is a well established process for computing the position (and orientation) of a robot's end effector, and any link. It takes as inputs the joint angles of a particular robot, and produces the cartesian coordinates and orientation of the end-effector and intermediate points. In our case, only the cartesian coordinates were used. In essence, forward kinematics applies a chain of transformations between adjacent coordinate frames. Each link has a frame, and each joint moves a particular frame with respect to another. These transforms take the form of homogenous transformation matrices that combine 3D rotation and 3D translation in a 4x4 matrix.

For our project, the primary goal was to implement and use RRT*, and explore optimization options of the overall system. This in mind, forward kinematics was required as a tool to do this in 3D, and we therefore did not implement forward kinematics ourselves, but utilized a C++ module provided to us by our partner in the Robomorphic Computing lab. The forward kinematics algorithm was used to translate a given joint space configuration into cartesian coordinates, in order to perform collision detection, and determine whether a given node was valid for the RRT tree.

3.3.4. Collision Detection

Three forms of collision detection were performed for our application: Collision with the table, collision with obstacles, and self-collision of the robotic arm.

Of these, collision with the table was the simplest to implement, and was simply a matter of assigning any point with a negative z value (within a particular margin) as invalid.

Obstacles were modelled as spheres, and were declared as part of the RRT* set-up for each run. Collisions with these obstacles were detected using a two-stage process within the check_robot_collision function. First, the Cartesian coordinates of each robot joint point, obtained via forward kinematics, were checked for containment within any defined spherical obstacle. This involved calculating the squared distance from the point (x,y,z) to the sphere's center (obs.x,obs.y,obs.z) and comparing it to the sphere's squared radius (obs.radius^2). If this calculated distance squared was less than the radius squared for any point and any obstacle, a collision was immediately flagged, indicating the point lies inside the sphere.

Second, if no joint points were found inside any obstacles, the line segments connecting consecutive joint points (let's call them p1 and p2) were checked for intersection with the spherical obstacles using the segment_shepre_intersection function. This function employs a standard geometric approach based on vector algebra. It determines the vector representing the line segment (V=p2−p1) and the vector from the sphere center C to the segment's start point (W=p1−C). It then solves for the parameter t along the line P(t)=p1+tV that satisfies the sphere equation ||P(t)−C||^2=r^2. This substitution leads to a quadratic equation in t: (V·V)t^2+2(V·W)t+(W·W−r2)=0. A collision is flagged if this quadratic equation has real roots (i.e., its discriminant is non-negative) and at least one of the roots for t falls within the range [0,1], indicating the intersection point lies on the segment itself. An additional check confirms collision if both endpoints (p1 and p2) are found to be inside the sphere, meaning the entire segment is contained within it. Although the additional check should have already previously been caught in the first check it is added for redundancy.

Self-collision detection was achieved by modelling each arm segment as a series of small spheres, placed evenly along the line between each pair of adjacent joint positions (obtained from the FK output). Spheres were generated by interpolating the number of spheres per link between joints using a linear blend. All pairs of spheres were checked for overlap, excluding nearby spheres in order to avoid false positives from adjacent spheres (skipping 5 indices ahead). If the square distance between sphere centers was less than the square of twice the sphere radius, a collision was flagged, discarding that particular joint configuration. This approach balanced accuracy, speed and complexity. For our purposes it was sufficient for preventing self-collision, was less complicated than other modelling approaches, and faster than approaches such as full mesh collision.

3.4. Profiling and Optimization Methods

The raw, unoptimized RRT* stack took over 2 minutes to run on the BeagleBone Black. For embedded robotic systems, execution speed is critical, and therefore, various options were explored for optimizing the system and improving execution time. While writing our code, we had already taken steps to try and write in an optimized manner, for example using square distance rather than root square distance (with sqrt() being a computationally expensive operation). This in mind, our optimization approach explored two avenues:

1. Using optimization flags at compilation to produce more efficient executables

2. Using a novel HLS toolchain produced by our Robomorphic Computing Lab partner

3.4.1. Optimization Flags

We explored the use of two primary G++ optimization flags:

- O2, which focuses on improving performance without increasing code size (too much) or breaking standards. For example, O2 performs optimizations such as eliminating dead code, inlining of small functions, branch prediction hints, loop-invariant code motion (moving calculations outside of loops if they don't change) etc.
- O3, which pursues a more aggressive approach, even if this means increasing the binary size or compile time, or violating some standards. On top of O2, O3 performs functions such as loop unrolling, vectorization, even more aggressive inlining, and function cloning.

3.4.2. Daisy Toolchain

Daisy is an analytical framework and tool developed by researchers at MPI-SWS, TUM and Saarland University, designed for the analysis and optimization of finite-precision computations.[3] Daisy can perform a number of functions. Particularly relevant to our project, however, the tool can analyze round-off error of a real-valued function, and then generate fixed-point code that approximates it within a user-defined error bound. Daisy can automatically generate C or Scala code using fixed point arithmetic, including correct bit-shifting and scaling logic. This tool has been augmented by Alp Eren Yilmaz and co-authors at the Robomorphic computing lab to make it more versatile and user-friendly, allowing a user to provide Rust code and pass this through a Daisy-enabled toolchain and produce fixed-precision c-code or even a Verilog implementation of the given functions that can be run on reconfigurable hardware.[4]

This is relevant for our project, because, under certain conditions, conversion from float to fixed-point precision can offer performance improvements, given that floating point operations are often computationally expensive. For our purposes, we used an alternative FK module that had been run through the Daisy toolchain and performed all calculations in fixed-32 format, instead of floats. Our theory was that this may offer further performance improvements in addition to the use of optimization flags.

4. Results & Discussion

4.1. Results

4.1.1. 2D Results

We tested our RRT* by performing initial runs in a 2D setting, where the algorithm was tasked with finding a path around circular obstacles. The algorithm performed well, and we were able to verify that it was able to produce both the standard RRT path, and an optimized path via RRT*. A python script was written to take as inputs the RRT outputs, and produce a 2D visualization of the RRT edges and final goal path and simplified path.

[3] Darulova et al. 2018. Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). Available online: https://people.mpi-sws.org/~eva/papers/tacas18_daisy_toolpaper.pdf

[4] Eren Yilmaz et al. 2025 (pre-publication). Towards Embedded Robotics with Guarantees: Enabling Rigid Body Dynamics Across Fixed-Point Precisions with Bounded Accuracy.

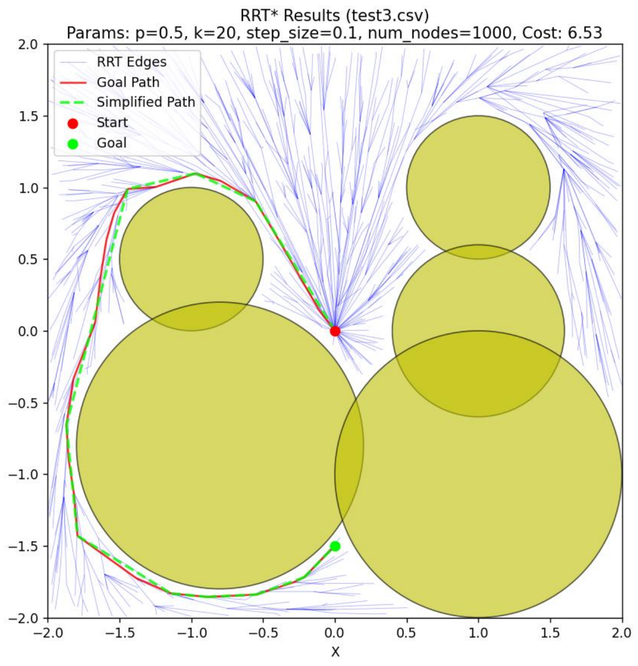

Figure 2. Demonstration of successful RRT* run in 2 dimensions

4.1.2.3D & Physical Results

As with our 2D implementation, our 3D implementation performed well, and was successfully able to determine paths around a variety of obstacles. Plotting all RRT paths would be visually cluttered, so we opted to only visualize the start position, end position, and intermediate positions on the goal path, as well as any obstacles. Figure 3 shows the obstacle modelled for our physical test case, where we tasked the robotic arm to navigate around a cardboard box. As can be seen, the RRT* algorithm translates well to higher dimensions successfully finding a path. A video of the physical test using the BeagleBone can be seen in our final demo video.

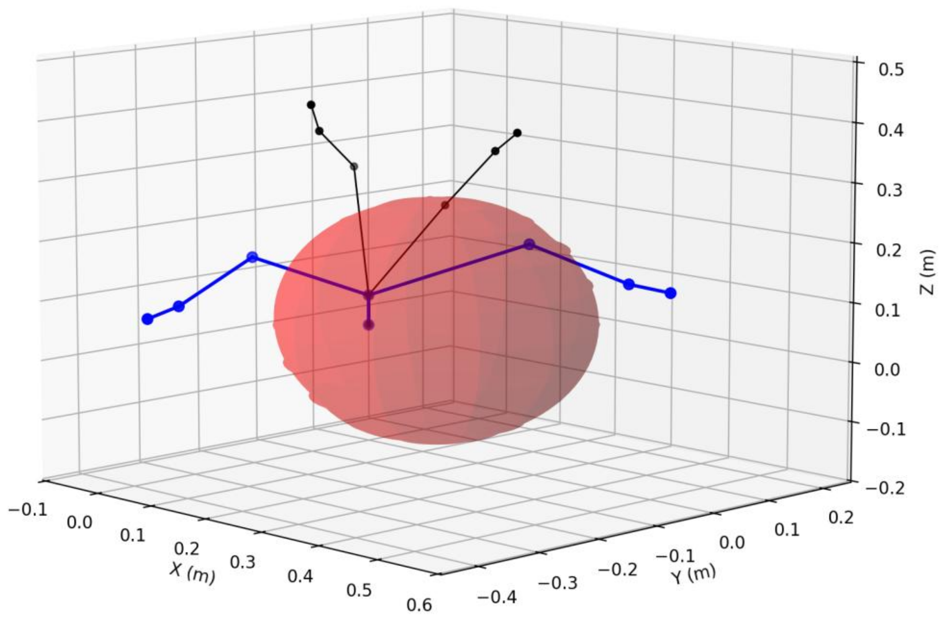

Figure 3. Demonstration of successful RRT run in 3 dimensions

4.1.3. Computation & Optimization Results

The RRT* stack was profiled on the BeagleBone black using gprof and the results logged, in particular we logged the most computationally expensive functions, and tracked gprof overhead to distinguish this from “relevant” results. Multiple runs were conducted, using purely optimization flags, as well as optimization flags as well as the fixed-point version of the forward kinematics algorithm. The results of these runs can be seen in Table 2 and Figure 4.

	O0	O2	O3	O0 + Fixed-Point	O2 + Fixed-Point	O3 + Fixed-Point
gprof overhead	31.86	0.18	0.09	31.9	0.28	0.14
check_robot_collision()	15.17	5.19	5.53	15.47	5.49	5.15
ForwardKinematics()	10.32	0.98	0.82	10.39	0.98	0.94
forward_kinematics()	0.72	0.63	0.71	0.95	0.64	0.8
distance()	1.22	0.24	0.39	0.94	0.38	0.32
other	88.87	7.37	7.04	90.4	7.19	7.47
Total	148.16	14.59	14.58	150.05	14.96	14.82

Table 2. Summary of optimization results (runtime, seconds)

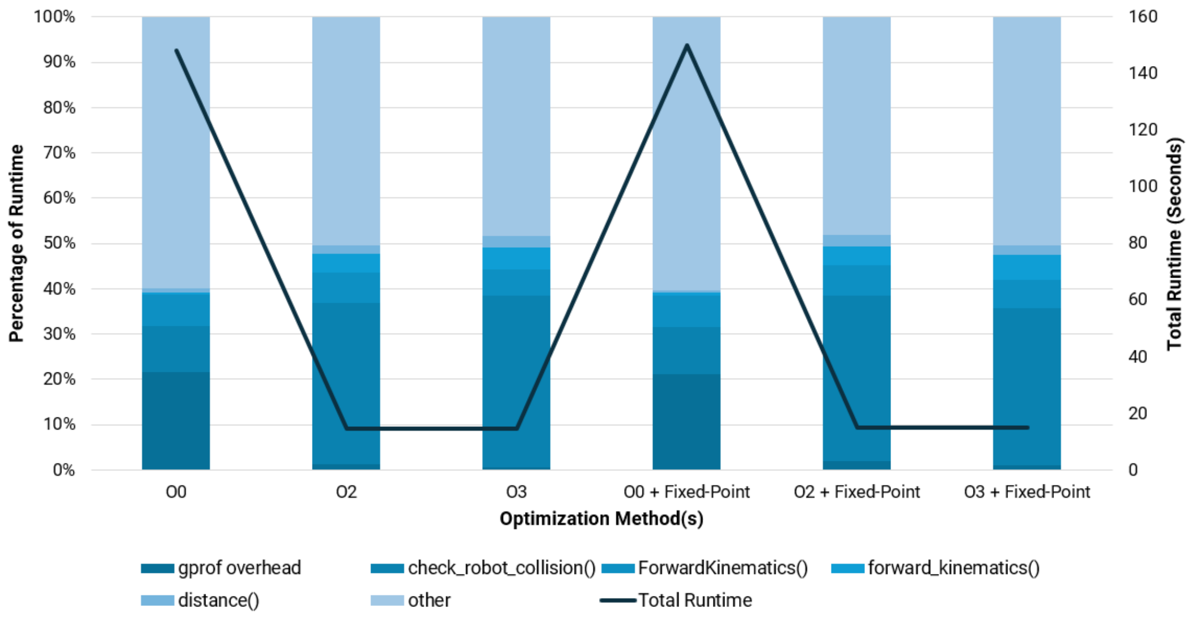

Figure 4. Visual summary of optimization results

4.2. Discussion

As discussed, we were successfully able to run a path-planning algorithm on an embedded processor and use this to control a real-world robotic arm to navigate around obstacles.

In terms of our optimization results, O2 shows a clear and marked improvement for runtime on the BeagleBone. However, O3 provided extremely marginal improvements over O2. We suspect this might be because the core loops in RRT are irregular and branch heavy, which does not lend itself well to further optimization via O3. Further, RRT* features heavy branch logic, which limits vectorization and loop unrolling. Operating on a Beaglebone also introduces cache and bandwidth limits, which may further mean O3 does not significantly improve execution time.

Interestingly, using fixed point format code did not seem to improve forward kinematics runtime significantly. We postulate that this is because the BeagleBone black does have a dedicated floating point unit, which may provide sufficient floating point computation performance to a degree where switching to fixed-point offers no tangible benefits. The fixed point implementation also requires bitshift operations, which may add computational overhead and erode some performance gain. It would be interesting in future study to conduct similar tests on an MCU without an FPU, to determine whether performance gains are realised in this scenario.

5. Limitations & Future Work

While we encountered challenges throughout our project, we were able to successfully execute on all of our project goals. If we had had more time, however, we would have liked to further explore methods to further optimize our algorithmic performance on the BeagleBone. Our best RRT* runtime was approximately 14 seconds, which is still fairly long for robotic applications that may need to run in near-real-time. One particularly intriguing path would be to make use of the Daisy toolchain's ability to produce hardware description language, and modify our codebase to offload some of the most computationally expensive functions, in particular elements of our collision detection, and forward kinematics, on to an FPGA, where they could be accelerated using custom logic, and in part parallelised. This may offer further improvement gains. During our project, we were successfully able to wire our BeagleBone up with a Basys3 FPGA development board, and modify open source UART modules by Nandland[5] to send dummy joint angles to the Basys3, and dummy cartesian coordinates back to the BeagleBone. With the hardware set-up already in place, this is a direction we intend to pursue in the near future.

6. Conclusion

This project set out to demonstrate that advanced motion planning algorithms like RRT* can be successfully deployed on real embedded hardware, not just simulated environments. We implemented a full robotic control pipeline on the BeagleBone Black, integrating path planning, forward kinematics, and actuation of a 5-DOF robotic arm. Along the way, we profiled and optimized our system using both compiler-level flags and an exploratory fixed-point conversion via the DAISY toolchain. While fixed-point acceleration didn't yield significant improvements on our specific hardware, the groundwork we have laid, including FPGA UART interfacing, positions us well to explore hardware acceleration in future iterations. Overall, this project validated that sampling-based planning can be made to work effectively in embedded, resource-constrained systems, and highlighted a clear path forward for building even faster, lower-power robotics systems.

[5] Nandland UART source code: https://nandland.com/uart-serial-port-module/