XILINX®

WP502 (v1.0) June 15, 2018

The Value of Python Productivity: Extreme Edge Analytics on Xilinx Zynq Portfolio

By: Giulio Corradi, PhD

The Xilinx® PYNQ framework thoroughly enables and integrates the Python language and run time onto the Zynq® portfolio. Leveraging Python productivity directly on the Zynq SoC architecture, users can exploit the benefits of programmable logic and microprocessors to more easily build designs for artificial intelligence, machine learning, and information technology applications.

ABSTRACT

The Python open-source programming language has become a de facto standard in applications ranging from engineering, scientific, data science, machine learning, information technology, and artificial intelligence.

Modern systems-on-a-chip (SoCs), when used in embedded applications, make it possible to run Python to execute complex analytical algorithms with performance close to that of a desktop workstation—but with a much smaller form factor and significantly lower power requirements. By pre-processing the data read from sensors, the Xilinx® Zynq portfolio provides a much higher level of performance and determinism as well as lower latency.

This approach, referred to as the PYNQ framework, effectively offloads many critical but repetitive operations that consume processor bandwidth unnecessarily from the application processor. The offloading capability is critical to meeting the need for increased intelligence in IIoT edge applications.

New Paradigms in Embedded Computing

A recent IEEE survey reported the two most popular programming languages in 2017 were Python and C. Within the embedded computing sphere, C has long been a stalwart. Traditionally, Python has been used for web or desktop computing and not as a language used within embedded computing; however, this is changing.

Python and its associated frameworks enable the development of complex algorithms used across data analytics, machine learning (ML), and artificial intelligence (AI) applications. Of course, these applications are hot topics within embedded computing and are driving the adoption of Python, especially within Industrial IoT (IIoT) at the edge.

C, C++, and Python are closely interlinked, as Python itself relies upon both C and C++ for its innermost libraries. However, unlike C and C++, which are compiled languages and can be executed using a bare-metal approach, Python is an interpreted language. This difference provides its own challenges in embedded computing: for example, Python requires an operating system, typically Linux, along with volatile and nonvolatile memory resources.

When it comes to embedded computing at the IIoT edge, implementations of ML and AI are increasingly acting as digital twins(1) and controlling physical actuators. As such, the ability of the solution to react in real time with a low deterministic latency is paramount. In addition, IIoT solutions must also be able to support other industry trends—namely:

- Partitioning among real-time processors, application processors, and dedicated processing elements according to the impending problem
- Creating interfaces to specialized processor offload engines to accelerate performance-critical kernels
- Using standard operating systems like Linux
- Offering a solution that provides scheduling and determinism
- Providing a high-productivity framework for prototyping and production
- Covering standard and legacy network communication interfaces and protocols, including converged IT and OT networks
- Providing rich libraries for machine learning and analytics
- Functional safety
- Cybersecurity

Architecting an IIoT platform is not a simple endeavor. The entire chain from the edge of the physical world to the cloud, including AI and ML, is complex and requires several specializations. Developing an IIoT solution, therefore, requires the use of increasing levels of abstraction, often tied to different job functions involved with the project, so that development takes place within an acceptable timescale and cost profile. See Figure 1.

1. A **digital twin** is a digital replica of physical assets, processes, and systems, allowing real-world behavior to be modeled providing monitoring, prognostics, and diagnostics.

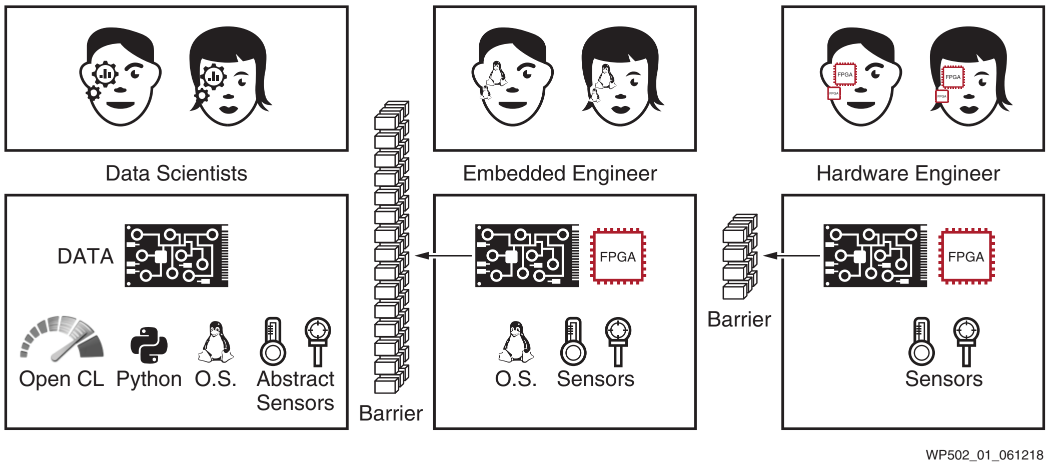

Figure 1: **Different Levels of Abstraction for Same Platform**

Exploring Machine Learning in IIoT

Increasingly, IIoT solutions incorporate embedded intelligence at the edge. For many applications, this means the implementation of machine learning inference. When implemented, ML algorithms reach a conclusion based on a set of input data using its experience. In ML, experience is provided by a learning process called training. Training of an ML application is performed using one of two methods: (1) human supervision or (2) fulfillment of a judgment function. Both require the application of large data sets of positive and negative cases to the ML network. After the ML algorithm is sufficiently trained, it can be deployed at the edge to perform inference from new and unknown inputs.

IIoT solution applications benefit from the use of ML, especially those applications where traditional methods do not result in acceptable performance or require significant human intervention, like re-callibration, maintenance, diagnostic, and fail-safe operation. These applications include:

- Sensors and measurement systems
- System identification
- Machine learning system control
- Advanced signal processing
- Reinforcement learning for autonomous systems
- Image processing

Developing ML solutions therefore requires both a platform and ecosystem with enough versatility to support a full development model, not just isolated elements of the solution.

The PYNQ Framework

The Xilinx Zynq-7000 SoC contains a dual-core Arm® Cortex®-A9 processor system (PS) and programmable logic (PL) designed to provide de facto standard features for a modern SoC, while also providing a unique and differentiated level of flexibility for offloading critical tasks to the PL. The Zynq UltraScale+ MPSoC and the Zynq UltraScale+ RFSoC extend this model with a quad-core Arm Cortex-A53 PS, PL, and other processing blocks depending on the specific part number.

This tight coupling of PS and PL allows for the creation of a system that is more responsive, reconfigurable, and power efficient when compared to a traditional approach. Traditional CPU-based approaches require the use of external memory to share large data structures, e.g., images. This reduces determinism and increases both power dissipation and latency, as arbitration and communication off-chip are required. Heterogeneous system-on-chip devices like those in the Zynq portfolio allow the designer to accelerate functionality within the PL of the device. This provides a deterministic response time with reduced latency and an optimal power solution. See Figure 2.

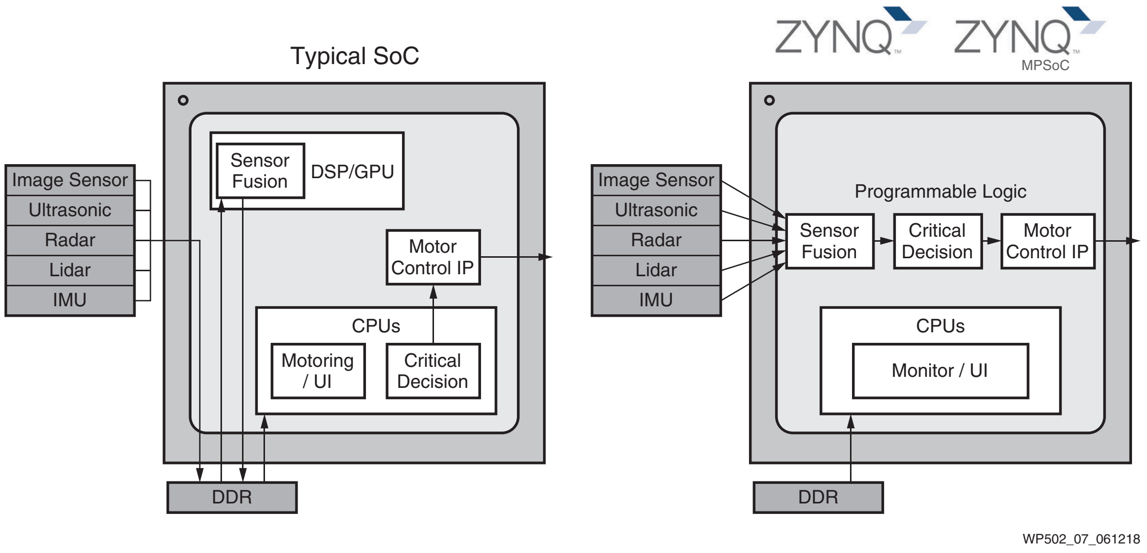

Figure 2: **Zynq Portfolio Benefits Compared to a Typical SoC**

The use of the PL features a wider interfacing capability than traditional CPU approaches, which come with fixed interfaces. The flexible nature of PL I/O structures allows for any-to-any connectivity, enabling industry standard, proprietary, or legacy interfaces to be implemented.

Thanks to the high-level synthesis tools SDSoC™ and Vivado® HLS, developers are able to turn C and C++ compiled code directly in hardware.

Along with acceleration of functions in the programmable logic, it is also possible to instantiate one or more MicroBlaze™ 32-bit soft-core processors. The MicroBlaze core allows execution of real-time critical applications. These features allow the Zynq portfolio to be used to meet all the requirements of a Python-enabled embedded platform.

The PYNQ framework enables Python to work with the Zynq portfolio, taking advantage of the acceleration provided by programmable logic. To do this, PYNQ employs PL overlays packaged as hybrid libraries. A hybrid library is a new form of library that includes an overlay bitstream, along with its associated hardware-dependent C code, and Python APIs. Hybrid libraries are a key mechanism for enabling reuse and can be installed easily using PIP Install (package management system used to install and manage software packages written in Python).

The PYNQ framework therefore provides the necessary abstraction levels required for all developers: data scientists, embedded engineers, hardware engineers, and system engineers. See Figure 3.

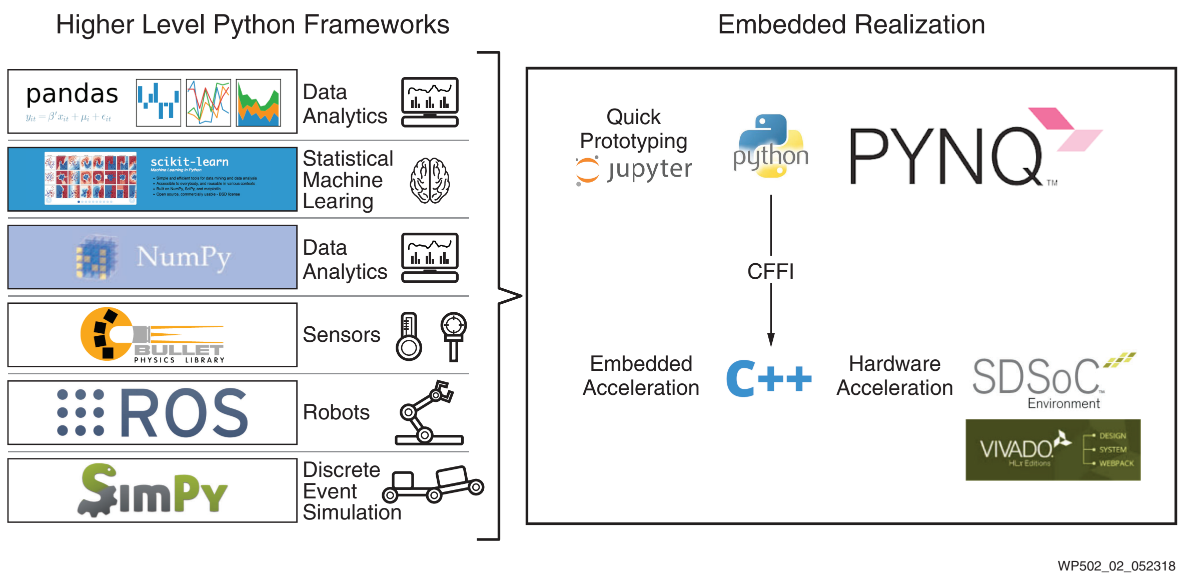

Figure 3: **Increasing Levels of Abstraction in the PYNQ framework**

PYNQ is the first framework to combine the following elements, which simplify and improve the Zynq portfolio based designs:

- A high-level productivity language (**Python**)
- **Hybrid libraries**, which enable acceleration in programmable logic
- A **web-based architecture** served from the embedded processors
- **Jupyter Notebook** framework

With these elements, the PYNQ framework offers significant advantages over the traditional SoC approach, which does not have the ability to leverage programmable logic.

The data scientist can immediately use the system Xilinx created in the Python framework with familiar packages. Figure 3 demonstrates a possible stack where some of the standard packages can be used. Panda, Scikit-learn, and NumPy are at the top, with other packages providing more specialized functions like ROS for robots and SimPy for simulation.

Python can import one or more ready-to-use pre-configured hardware modules, enabling the acceleration of the application and removing bottlenecks. All the packages then can be explored and used within the Jupyter environment and interfaced to the programmable logic using hardware libraries based on FPGA overlays.

The PYNQ framework is designed around an open-source community where overlays are created and shared between developers. Using this method, developers are not required to build overlay(s) until they see the need for a new one—that is, when an overlay performing the desired function does not already exist. Each new overlay should ideally follow an established "design pattern" for IIoT. These design patterns include (but are not limited to) the following:

- **Accelerators** expedite computation-sharing and/or exchanging data with the main processors. Exchange can happen in PL memory (block RAM), in on-chip memory (OCM), in the L2 cache, and in the DDR memory as function of data size and performance.
- **Loggers** acquire data—in general, raw data—that is shared with the main processors. Logging can happen in PL memory (block RAM), in on-chip memory (OCM), in the L2 cache, and in DDR memory as a function of data size and performance. The acquisition process is started by explicit triggering of an event from the main processors or by capturing an external event.
- **Sequencers** generate automatic sequences of logical values; depending upon the complexity, these can include instances of programmable generators to implement Boolean functions, FSMs and/or arbitrary digital patterns.
- **Workers** provide real-time control based on the MicroBlaze 32-bit soft-core processor, programmable in C/C++ with the purpose of offloading repetitive tasks from the main processors to ensure determinism.
- It is possible to use workers as **Safety Modules**. MicroBlaze processors can be configured for lock-step operation. While the full Zynq portfolio has been designed for functional safety, this capability is important when the product is going to move from the initial prototype into something that is production-ready.

Sensors and Measurement Systems

Sensors are a crucial part of any industrial system and especially IIoT solutions. IIoT solutions deploy several different sensor modalities, from simple thermocouples for temperature measurement, to complex sensor fusion that combine multiple heterogeneous sensors for measuring certain physical quantities. Implementing ML within an IIoT solution enables the developers to get the best performance from a given sensor, enabling greater efficiency of:

- Sensor data acquisition (example: vibration analysis)
- Sensor data normalization
- Sensor linearization
- Sensor diagnosis
- High-level sensors
- Sensor fusion
- Calibration and self-calibration

Sensor Diagnosis

Sensor performance changes throughout the operating lifetime due to aging. This is especially true if sensors are used in harsh environments, where aging affects reliability and introduces drift and deviation. The ability to diagnose a sensor is also extremely useful if the sensor is used in safety applications; in this case, proper diagnostic processes are also part of the safety system.

Using ML, the designer can create a sensor model—or digital twin, as it is more commonly known—to predict sensor output while also continuously monitoring actual sensor output. At nominal conditions, sensor signals follow some known patterns with a specific degree of uncertainty due to system and measurement noise. However, when sensor failures occur, the observable outputs deviate from the predicted values, and sensor failure can be declared when the deviation exceeds specified timing or value thresholds. Analytical redundancy techniques allow information from the sensors at nominal conditions to be used to create models for the dynamic system, so the “digital twin” does not age or fail, and lives forever.

Predictive Maintenance Use Case: Ball Bearing Fault Detection for Diagnostic and Safety

The packaging materials industry has recognized the importance of *total productive maintenance* as a system of proactive techniques for improving equipment reliability. Bearing faults, which often occur gradually, represent one of the foremost causes of failures in the industry. Therefore, detection of these faults in an early stage is quite important to ensure reliable and efficient operation. Individual packaging machines often employ more than eight motors and many spindles, providing several causes of failure that can result in a *line stop* condition.

Ball bearings ensure free rotation of the shaft or the axle with minimum friction and hold a shaft or axle in its correct position. If a ball bearing of a rotating system fails, its consequences can be catastrophic. There are many causes of bearing damage, including:

- Bearing misalignment
- Fretting wear, a type of corrosive damage of the bearings’ contact surfaces
- Variable-frequency drive, generating shaft currents that cause pitting, fluting, and fusion craters
- Improper bearing lubrication
- Bearing corrosion
- Bearing fatigue
- High temperature and other factors

Detection of such faults before a failure occurs is of paramount importance to reduce operational and maintenance costs. Thankfully, the availability of low-cost accelerometers that provide high bandwidth measurements enable very powerful vibration data acquisition systems. When these are deployed with advanced motor control, it is possible to deliver a system that can detect possible failures in real time, based upon the PYNQ framework.

For this application, the PYNQ framework is deployed on an ARTY-Z7 board. This board supports the PYNQ base overlay, but it is just one of the possible incarnations of the PYNQ framework. In this example, the system setup contains a high-performance motor control system and an acquisition system formed by five Kionix[1] three-axis accelerometers.

There are several possible techniques for vibration monitoring, including vibration measurement, acoustic measurement, temperature measurement, and wear analysis. The use of silicon accelerometers provides an economical way to sense vibration in all three axes, making it possible to get a full picture of all possible vibration modes happening in the system. This is illustrated in Figure 4.

To ensure accurate measurement, accelerometers are positioned close to the ball bearings in the motor, gearbox, and main pulley.

System parameters, such as motor currents, stator voltages, and the position of the shaft angle, are monitored. A motor model is used to estimate angle and internal voltages.

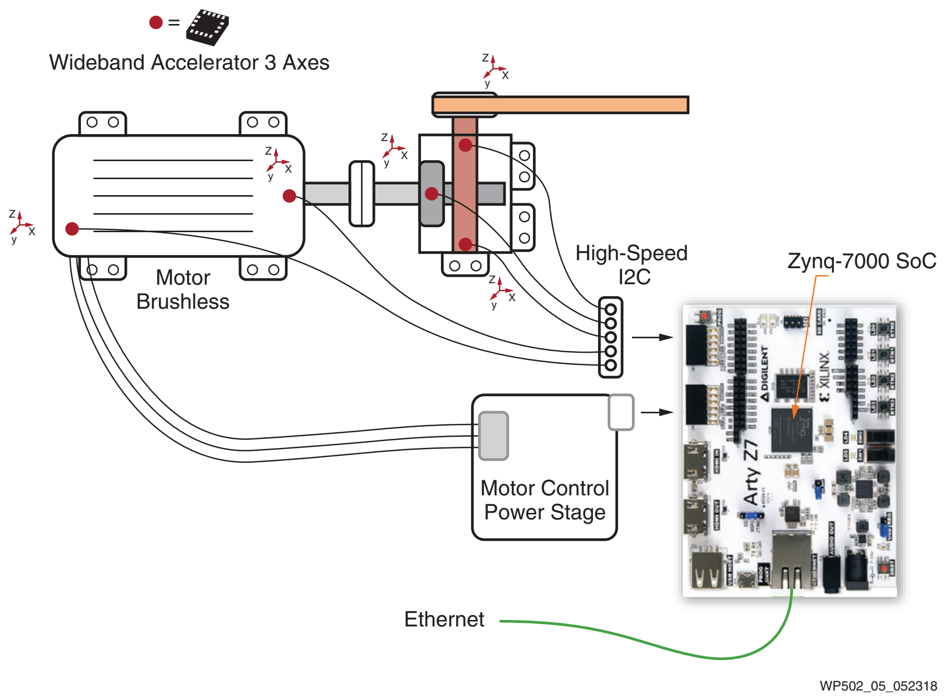

Figure 4: **Fault Detection and Motor Control**

1. Kionix Inc., www.kionix.com, manufacturer of microelectromechanical (MEMS) systems, including accelerometers, gyroscopes, and sensor fusion inertial sensor technology.

As can be seen in Figure 5, a new overlay is created for the PYNQ environment using the following elements:

- A **motor control system** is instantiated as an HDL block using QDESYS (www.qdesys.com) high-performance motor control IP set.
- A **logger** is instantiated as a Vivado HLS module and is connected to high-performance AXI bus 0.
- The **on-the-fly Fast Fourier Transform (FFT)** is instantiated as a Vivado HLS module and is also connected to high-performance AXI bus 0.
- The **MicroBlaze soft-core processor** implements a worker that decouples and manages the accelerometer data, making it available as a shared segment in DDR memory. The MicroBlaze processor offloads management of the accelerometers from the Arm processors. The I2C protocol, addressing them individually, compensates for bias, sensitivity, temperature, and ratiometric errors.

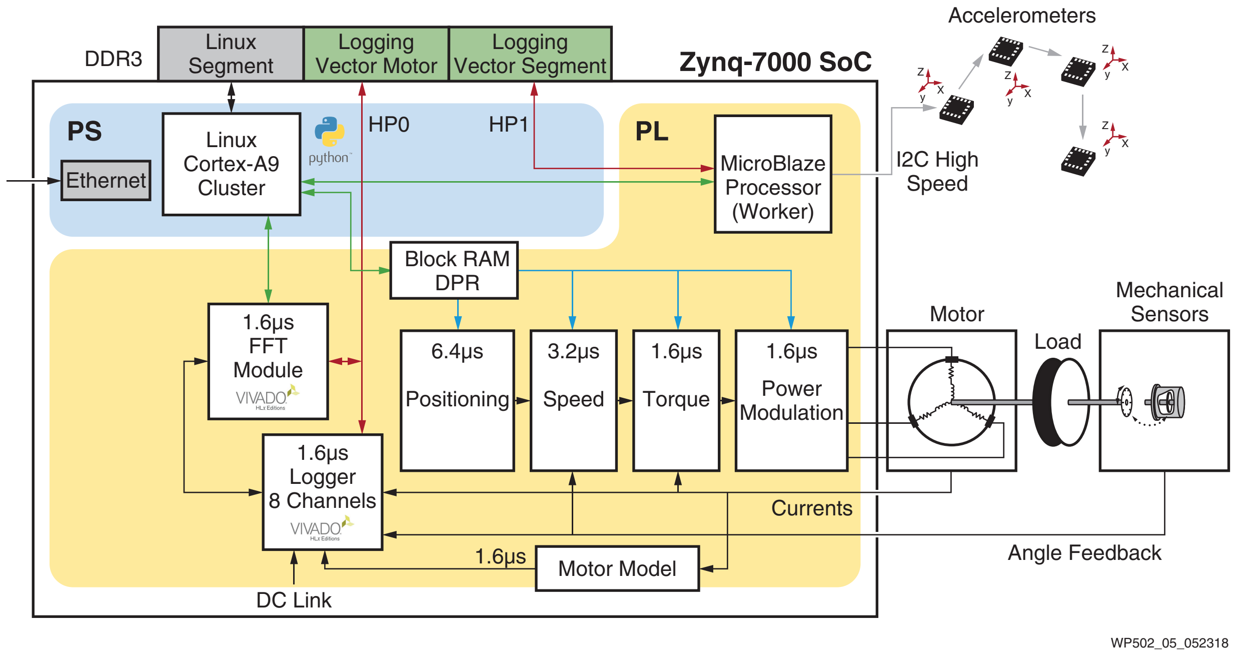

Figure 5: **Acquisition and Motor Control Internal Blocks**

To enable the new overlay to work with the PYNQ framework, the Python C Foreign Function Interface (CFFI) is used. This interface allows interaction with almost any C code when using Python. Within the PYNQ framework are programmable logic acceleration functions, and the framework's C/C++ based software drivers are accessed. The PYNQ framework also provides additional functions that are helpful when developing applications. Two examples of PYNQ core functions are (a) **mmio**, which allows memory-mapped I/O, and (b) **xlnk**, which allows allocation of DDR memory as buffer space visible to NumPy (the fundamental package for scientific computing with Python). **xlnk** is responsible for getting the virtual and physical addresses for the PL where the logged vectors are mapped.

As a small example, the vibration signals from the accelerometer can be correlated with the vibration signals detected by looking at the currents through the motor, so that correlation is possible by selecting the traditional vibration analysis or the more sophisticated signature analysis.

The descriptions of the measurements and of the overall system are worthy of a separate white paper and are not included here. For illustrative purposes, Figure 6 shows the limit of a ball-bearing frequency exceeding the threshold. The signal is extracted from the apparently good current waveforms of Figure 7.

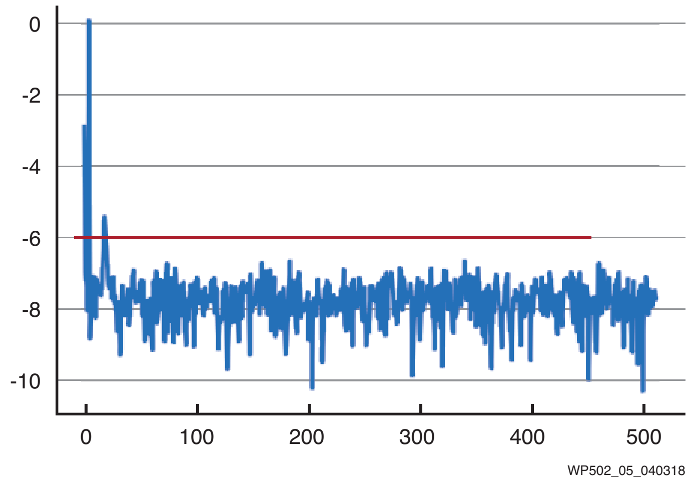

Figure 6: **Vibration Analysis with Current Signatures**

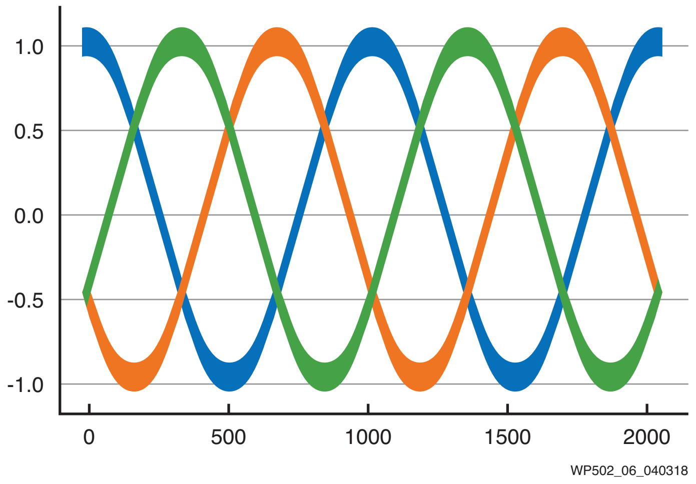

Figure 7: **Waveforms of the 3-phase Motor (the Ball Bearing Signature is Buried in the Noise)**

The above example demonstrates that with proper understanding of the problem, as well as of the Zynq portfolio, advanced diagnostics are possible and practical, opening the door to more intelligent approaches to machine learning-enabled IIoT systems.

Conclusion

This white paper covers only some of the basics of the PYNQ framework, which binds Python and the Zynq portfolio together. The PYNQ framework offers new and immediate possibilities for creating systems capable of exploiting the full capabilities of the IIoT and the emergent capabilities offered by Machine Learning.

Machine Learning at the edge is in its infancy, and Xilinx is committed to deliver best-in-class products and design frameworks to expedite and simplify the design, deployment, and maintenance of intelligent and adaptive assets. Being just at the start of a very exciting journey, many of the topics described in this white paper are to be detailed in further white papers, application notes, and examples. Some of them are already available in the PYNQ GitHub repository.

Getting Started

- The **PYNQ framework** is available at http://www.pynq.io.
- The **Arty Z7 board** described in this white paper is available from Digilent Inc. at: store.digilentinc.com/arty-z7-apsoc-zynq-7000-development-board-for-makers-and-hobbyists/
- A bundle featuring the Arty Z7 board, the power module for motor control, and a BLDC motor is available at: https://shop.trenz-electronic.de/en/TEC0053-04-K1-EDDP-Motor-Control-Kit-with-Motor-Power-Supplies

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
06/15/2018	1.0	Initial Xilinx release.

Disclaimer

Automotive Applications Disclaimer