uOSC: The Open Sound Control Reference Platform for Embedded Devices

Andy Schmeder
Center for New Music and Audio Technologies,
University of California, Berkeley
1750 Arch Street
Berkeley, CA 94720
+1 (510) 643-9990
andy@cnmat.berkeley.edu

Adrian Freed
Center for New Music and Audio Technologies,
University of California, Berkeley
1750 Arch Street
Berkeley, CA 94720
+1 (510) 643-9990
adrian@cnmat.berkeley.edu

ABSTRACT
A general-purpose firmware for a low cost microcontroller is described that employs the Open Sound Control protocol over USB. The firmware is designed with considerations for integration in new musical interfaces and embedded devices. Features of note include stateless design, efficient floating-point support, temporally correct data handling, and protocol completeness. A timing performance analysis is conducted.

Keywords
Open Sound Control, PIC microcontroller, USB, latency, jitter

1. INTRODUCTION
1.1 Motivation
The Open Sound Control (OSC) protocol [10] is widely adopted by the NIME community as a common means for the transmission of streaming musical gesture data. In the observation of the authors, the success of OSC arises not from its technical features but rather from its simplicity (i.e., low conceptual overhead and human-readability) and the promise of interoperability with a diverse array of applications. For example, the need for conceptual simplicity and generalized interoperability has led developers to create OSC "wrappers" that translate other hardware protocols into OSC message sources and sinks—such as HID (CUI-OSC, oscjoy), P5 Glove (GlovePie), Nintendo Wii, SpaceNavigator, and MIDI (OSCulator). While these efforts achieve usable results, indirection due to protocol translation introduces unnecessary latency, and such translations have no hope of achieving timestamp and atomicity semantics present in OSC. It is natural then to ask why OSC is not implemented directly in the hardware, thereby obviating the problem.

A common assumption is that the features of OSC (floating point support, high resolution timestamps, and a moderately verbose binary representation) are excessive for embedded targets. The work described in this paper demonstrates that this is no longer the case for contemporary microprocessors.

The freely available OSC-enabled firmware described here, *uOSC* (pronounced "micro-OSC"), improves on the performance, standards-conformance, and cost of OSC implementations for new controllers and retrofits. *uOSC* uses the OSC protocol at the level of the embedded device itself, obviating the need for intervening applications to provide protocol translation and making possible more direct (and thereby, higher performance) access to the data. In addition, because high-speed manipulation of microcontroller pin functions is provided, users can develop applications in any programming environment with OSC support without learning microcontroller programming or a new specialized language such as Wiring [http://wiring.org.co].

A key aim of the *uOSC* effort is to provide the developer community with a solid reference implementation of OSC to extend and port to other embedded devices. Developers of other OSC clients and servers are facilitated by an affordable source and sink of OSC data that can be integrated into tangible human-computer interfaces.

1.2 Implementation Challenges
Since its introduction in 1997, the OSC protocol has been successfully integrated in dozens of hardware and software products and used in thousands of performances and installations. Unfortunately, nearly all implementations fail to implement the complete OSC 1.0 Specification[1]. In spite of general consensus that temporal semantics are crucial for musical interfaces [9], meaningful support for OSC timestamps is extremely rare. Support for the full address pattern-matching syntax is often omitted perhaps because of unfounded concern over the performance of the pattern-matching algorithm.

The implementation challenges have been noted by the core OSC creators [2], and this new implementation is carefully written to demonstrate efficient solutions and best practices.

1.3 Overview
The focus in this paper is on novel features, design concepts and especially on how the implementation was optimized to achieve the performance and completeness that OSC applications require within the programming and performance constraints of small microcontrollers.

To promote wider use of OSC, we initially targeted a physically small, readily available, extremely low cost (USD $25) hardware platform, the PIC18F2455-based "bitwacker".

NIME08, June 5-7, 2008, Genova, Italy

[1] A database of OSC implementations and their features is online: http://opensoundcontrol.org/implementations.

The *uOSC* project source code, new developments, benchmarks and details beyond the scope of this paper are documented online at http://cnmat.berkeley.edu/research/uosc.

2. HARDWARE PLATFORM

2.1 Microchip PIC USB Full-Speed

uOSC runs on the popular and compact Microchip PIC18F USB-Full-Speed family of microcontrollers. The product line spans chips from 20-80 pins, 10+ analog inputs, hardware modules for TTL, PWM, etc, 2-4Kb RAM, 8-128Kb ROM, and CPU speeds of 12 MIPS. Many prototyping boards for these devices are available for less than USD $100. The initial release of *uOSC* specifically supports the Sparkfun Bitwacker, the CREATE USB Interface (CUI) [4]; and Olimex PIC-USB-455x. (Pictured in Figure 1, ordered bottom-to-top). Microchip provides a free C compiler (C18), an implementation of C standard library and a comprehensive IDE.

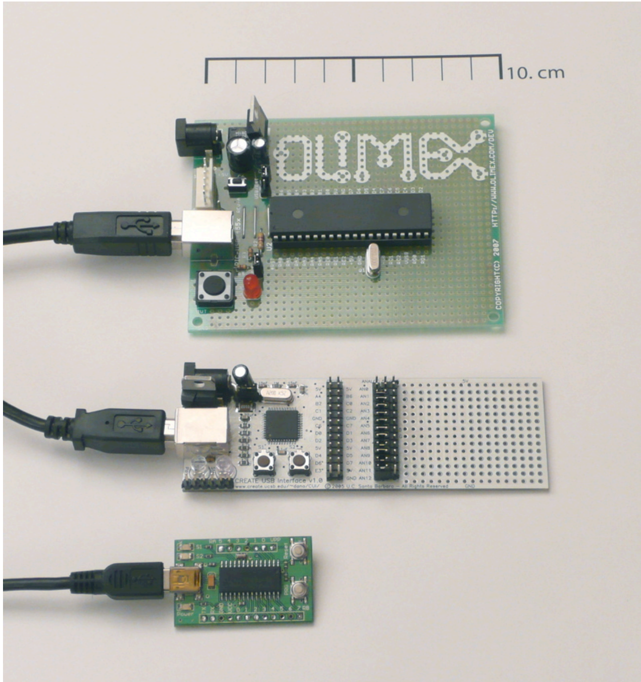

Figure 1: A collection of devices supported by *uOSC*

2.2 USB vs. Ethernet

Many OSC users are only familiar with OSC data transported in TCP/IP packets. Even though OSC is transport independent, overall application performance does depend on the particular transport used, so it is worth examining the advantages of various transports. The key advantage to the NIME community of USB is that power is provided over the cable. Although there are now standards for sending power over Ethernet cables these are not employed in current desktop or laptop computers. Another advantage of USB concerns timing aspects of OSC. USB provides a timing beacon (the Start-of-frame packet), and supports the timing guarantees and bandwidth reservation of isochronous data streams for appropriate device classes [12].

USB provides point-to-point connections in a shared bus arrangement whereas Ethernet has network-wide addressing, electrical capacity for long cable runs and can leverage the performance benefits of a switched communication fabric. Currently 10-gigabit Ethernet is winning a throughput-performance race over Firewire, USB and SATA, but at a cost point that is irrelevant to the affordable applications we have in mind.

3. FIRMWARE OVERVIEW

uOSC builds on the MCHPFSUSB firmware [13], an open-source implementation of the USB control endpoint and a USB class-compliant serial port. The *uOSC* core program is triggered by activity on the USB interface: receipt of the USB start-of-frame (SOF) packet from the host controller serves as an isochronous 1000Hz timing beacon to which the firmware operations are synchronized.

3.1 Device Clock

The current time, relative to device initialization, is tracked with a precision of 1 msec. The clock is incremented by the SOF interrupt. Because this signal comes from the host controller, the clock is not subject to any thermal drift or resonator imprecision caused by the hardware. The clock is used for bundle timestamping and scheduling.

3.2 Pin Initialization

By default all pins are configured as inputs on power up. The user may change the direction of any pin by sending the appropriate OSC message, and if this direction state is committed to the flash memory, it will be restored on power-up.

3.3 Extensible Hardware Modules

Special features provided by hardware modules such as PWM control and TTL serial can be enabled on user request and, if desired, re-enabled on initialization.

3.4 Unique Identification

A non-volatile writable memory section is provided for the storage of a unique 64-bit identifier. On first startup when this identifier is undefined it is populated with a pseudo-random number derived from the non-deterministic USB host enumeration time.

3.5 Pin I/O

Double-buffering is used on digital and analog pin I/O to minimize possible skew in timing of pin read and write operations. Skew for digital I/O is less than one microsecond, and approximately 30 microseconds between analog inputs. Double-buffering also ensures that I/O operations always occur at regular and known time slices.

3.6 USB-Serial Interface

uOSC implements the descriptor and endpoint logic to appear to the host controller as a CDC-ACM device. Of the many possible USB device classes this one has the advantage of being the most “plug and play” as modern operating systems are shipped with drivers that support it. It also offers higher performance and more flexibility than HID classes.

CDC-ACM uses the USB bulk transfer type that has a theoretical maximum bandwidth of 12Mbit/sec on a USB Full-Speed bus. We have measured rates of up to 3Mbit/sec of fully-formatted OSC data.

3.6.1 SLIP for Serial Transport Framing

SLIP [11] is a simple and lightweight protocol popular for microcontroller applications that provides the framing necessary to mark the boundaries between OSC bundles on a serial transport. The double-ENDed variant of SLIP is recommended because it provides a robust state-free detection of the start of a packet.

SLIP is the recommended framing method for OSC encoding over stream-oriented transport such as TCP and has already been used for this purpose in the popular Make Controller Kit by Making Things [http://makingthings.com].

4. ULTRA-LIGHT OSC PROGRAMMING

The small memory model, limited type support, and low clock rate of the microcontroller imposes challenging limitations on the implementation of an OSC library that is both full-featured and easy to create and understand.

4.1.1 OSC as Binary Data Structure

OSC implementations typically translate from the OSC binary message structure to/from an appropriately typed data structure in the native format of the language along with encoding metadata. With only a few thousand bytes of memory to work with, *uOSC* cannot accommodate this style, and so the programmer works directly with C pointers to a statically allocated buffer. Only one incoming message and one outgoing message are simultaneously processed. This style was anticipated in the OSC specification with the mod-4 byte-alignment rule and conservative native type support.

4.1.2 Open-Ended Bundles

An important feature of an OSC bundle is that the total length of the frame is not encoded in the bundle header. This allows *uOSC* to format bundles with multiple messages while only retaining a single outgoing message in memory. In addition, the number of responses generated by an OSC pattern dispatch does not need to be known in advance.

4.1.3 Type Considerations

The PIC18 is an 8-bit processor, so, for efficiency the use of 8-bit and 16-bit numbers is preferred. OSC uses minimum 32-bit numbers, so *uOSC* provides efficient routines to pack 8-bit and 16-bit numbers. *uOSC* also provides routines to pack low-bit depth integers as normalized floating point fractions, and to pack automatically padded strings from ROM or RAM data. *uOSC* packs boolean data types using the 'T' and 'F' typetags, which do not consume any space in the data section of an OSC message.

4.1.4 Push-down of the SLIP encoder

The SLIP reserved characters have the two highest bits set (ASCII characters >= 192). The bulk of the output data stream does not require SLIP encoding. For example, the SLIP encoder can remain inactive in cases such as OSC address patterns that are printable ASCII, bundle sub-message lengths, NULL-padding bytes, and other bytes known to be strictly less than 192.

4.1.5 Input decoding state machine

The SLIP decoder must be active at all times. To avoid the necessity to reexamine input bytes, the OSC parser is embedded inside the SLIP decoder. The SLIP decoder, in turn, is embedded in the USB serial input handler, resulting in a third-order nested state machine. The OSC parser consists of bundle start detection, basic sanity checks on the packet format, and pointer retention to the location of address, typetags, start and end of the data section. Any SLIP decoding error causes the entire bundle to be discarded.

4.2 Code Example

The following example illustrates the programming style on the microcontroller using the ultra-light OSC implementation to create a port report with 8 data values of variable type:

```
oscBundleOpenTimestamped(); // sends SLIP_END and packs time
oscMessageOpen(); // reserves 4-bytes at start for length
oscPackROMString("/rb");
p_osc_tt = p_osc_message + 1; // pointer to typetags
oscPackROMString(",NNNNNNNN"); // final types unknown
for(i = 0; i < 8; i++) {
        // invokes oscPackInt16ToFractionalFloat
        // returns 'T' or 'F' for digital pins
         *p_osc_tt++ = oscReportPin(i);
}
oscMessageClose(); // prepends length, invokes CDCTxRAM
// other messages are packed in here…
oscBundleClose(); // sends SLIP_END and finalizes CDCTxRAM
```

5. LOW-COST FLOATING POINT

A widely adopted OSC convention also used by audio plug-ins is to scale control parameters to floating point values using a conventional representation such as the unit interval. The benefit of this abstraction became obvious for PIC18 family of microcontrollers as Microchip recently upgraded the ADC on some new variants from 10-bit to 12-bit—an integer encoding would require target-specific logic on the client side to accommodate both ranges.

Even though the PIC18 processor has no hardware FPU, Microchip provides an implementation of <*math.h*>, the C *float* type, and IEEE-754 compliant operations by software emulation. Profiling of this code revealed that the cost of int-to-float conversion (90 microseconds per conversion) was too great for use at the desired reporting rates.

We therefore created novel special-purpose code for floating point conversion that is exact for integers up to 23-bits and is approximately 3 times faster than the general-purpose library.

5.1 Theory

We take the normalized target range to be the closed interval [0.0, 1.0]. This results in the conversion formula:

$$y = x / (2^n - 1)$$

For simplicity, suppose that n = 8. x is given in binary digits as:

$$x = x_8x_7x_6x_5x_4x_3x_2x_1$$

where x_8 is the most significant bit. Then, as a binary repeating decimal:

$$y = 0.x_8x_7x_6x_5x_4x_3x_2x_1(x_8x_7x_6x_5x_4x_3x_2x_1)\ldots$$

The conversion to y attains the sufficient precision as x when the decimal expanded to the first repetition of the most-significant-bit of x (the 9th fractional digit above). This bit equals 1 when y >= 0.5, and 0 otherwise. Furthermore, a special case applies when $x = 2^n - 1$, y = 1.0 since by definition of a real number, the repeating binary decimal 0.11111111(111111111)… = 1.0.

5.2 Conversion Algorithm

The calculation of y as IEEE-754 single-precision floating point proceeds as follows:

1. If x = 0, return 0.0. If $x = 2^n - 1$, return 1.0.
2. Scan digits of x to find the index, i, of the most significant non-zero bit. Requires O(log n) comparisons. If $x >= 2^{(n-1)} - 1$ then least significant bit of y (first repetition of most significant bit of x) is 1, else it is 0.
3. Compute the exponent as e = 127 – (n – i).
4. Left-shift x by (n – i) + 1 places to obtain mantissa.
5. Composite the exponent, mantissa and least significant bit together to realize IEEE-754 format. Requires O(n/8) *shift* and *or* operations.

The first byte is never SLIP encoded (sign bit is always zero). The last byte is SLIP encoded for y >= 0.5, otherwise the last byte is zero.

The inverse conversion similar algorithm is similar but also requires detection of denormal numbers and a rounding operation.

6. OSC REPORTING

uOSC sends OSC packets reporting the current state of all pins isochronously at intervals of two milliseconds. The reporting itself consumes only approximately one millisecond of processor time. The remaining time is used to handle other device functions such as processing of incoming OSC messages. Note that this doesn't mean that there is 2msec of jitter in the measurements themselves -- their timing relationship to the 1000Hz USB-SOF beacon is precisely known. An appropriately implemented host driver could achieve sub-millisecond timing precision.

6.1 Bundle Timestamps

The bundle timestamp conforms to the NTP fixed-point format described in the OSC specification. The fractional part is computed to a precision of 1 msec This is approximately 2^{-10}, so a 16-bit integer is sufficient for the calculation. The fractional part is exactly zero at intervals of 1000 SOF interrupts, i.e., there is no roundoff error accumulation. The integer part is a long integer, which is unbounded for all practical purposes. Since the host and microcontroller have a point-to-point connection the timestamp can theoretically be conformed to the host computer's best UTC approximation [2].

6.1.1 Use of IMMEDIATE

Informational messages such as device firmware version, pin capability reports, profiling and debugging information are not time-sensitive and are encapsulated in bundles that use the IMMEDIATE timestamp (value: 0x0.0x1).

6.2 Efficient encoding of Port Reports

To save space in the data stream, sequentially numbered pins are grouped together in a single message called a *port report*. Each analog input pin is reported as a normalized floating-point number, OSC typetag 'f', requiring approximately 5 bytes. A pin configured as a digital input or output is reported as boolean using OSC typetag 'T' or 'F', requiring 1 byte of data space. A pin that is not connected or in a reserved state (e.g., in use by a hardware module) is reported as NULL using OSC typetag 'N', consuming 1 byte data space. The CNMAT OpenSoundControl object for MaxMSP[2] supports these types sensibly.

For a port of 8 pins, the total size of the OSC message is 12-60 bytes, depending on current pin configuration. The same number of pins encoded as separately addressed messages would require 96-160 bytes.

7. OSC DISPATCH

An incoming OSC message is *dispatched* by matching its address pattern against a nested structure of path components and invoking the appropriate callback for each match. Full support for OSC address pattern matching is implemented in *uOSC*.

[2] This and other OSC-related software is available online at: http://cnmat.berkeley.edu/downloads.

7.1 Dispatch Table Structure

The dispatch structure is a statically allocated tree structure using the following data structure:

```
typedef struct _oscSchemaNode {
     oscCallback target;
     byte num_children;
     rom char* child_name[OSC_MAX_CHILDREN];
     rom struct _oscSchemaNode* child[OSC_MAX_CHILDREN];
} oscSchemaNode;
```

Adding new method calls is simply a matter of inserting new nodes into the root node.

7.2 Efficiency of Pattern Matching

The purpose of the OSC pattern syntax is primarily to enable the client to compactly describe certain bulk and atomic operations, not to provide a sophisticated search mechanism. The OSC address pattern syntax is significantly less complex than typical general-purpose regular expression languages. Specifically: 1. Patterns may not cross '/' boundaries in the address, 2. List matches do not support nesting or containment of other pattern operators, and 3. Character-class matches and wildcard operators '?' and '*' are always greedy, obviating the need for backtracking. Therefore a pattern match is O(1) in memory.

The set of possible matching addresses is finite, and for patterns up to a set length, the total execution time to match is bounded. Furthermore, the dispatch process can leverage the nested structure since child addresses cannot match if the parent fails to match.

Our profiling shows that the cost of matching in *uOSC* is not a cause for concern and in particular is not more expensive than a standard string comparison for the most common case of addresses that contain no wildcards.

7.3 Scheduled Dispatch

When a received bundle has a timestamp in the future relative to the device internal clock, the action of the packet can be delayed until the requested time. A bundle with a timestamp in the past is discarded. This mechanism makes possible the forward synchronization method for jitter compensation [5].

The embedded processor has insufficient RAM to retain entire packets for future processing so scheduling is limited to digital pin writes, which are stored in a fixed length, insertion-sorted list.

7.4 Port Writes and Pin Aliasing

The client can write to groups of pins organized in ports using the same format described in section 6.2. Individual pins can also be addressed using their specific addresses.

8. DEBUGGING AND PROFILING

Profiling is essential for code optimization. However, use of in-circuit serial debuggers is known to be problematic for USB devices because associated interrupts are time-sensitive. Timing issues can also arise when using *printf*-style debugging over the TTL serial port.

uOSC includes a microsecond-accuracy profiling system, and when enabled by compile-time switch, timing of various operations are measured and reported periodically in supplemental OSC messages. This solution has negligible impact on the timing performance of the system.

9. DEVICE ADDRESS SPACE

The following lists the messages that *uOSC* generates and accepts for the case of the Sparkfun Bitwacker board. Minor variations apply for other boards because of the "user friendly" design choice that parameters are named according to the silk screens on each development board.

9.1 Port and Pin Messages

```
/ra ffffff : generates/accepts port-report format
    /0 : individual pin control for /ra/0
    /info : returns "dio", "adc", "pwm", "ttl"
    /state : returns "input", "output", "reserved"
    /set : accepts "low", 0, 0.0, or "high", 1, 1.0
    /get : see /set
/1-5 (same as /ra/0)
/rb ffffffff : /rb port report (8 pins)
/0-7 (same as /ra/0)
/tx (same as /ra/0)
/rx (same as /ra/0)
/status
    /0 : controls the yellow status LED
    /set : accepts "off", 0, 0.0, "on", 1, 1.0
    /get : return LED state
/1 : controls the red status LED
```

9.2 Device Messages

```
/device
     /platform:returns "Bitwacker", "CUI", etc.
     /firmware : returns "uOSC 1.0"
     /processor : returns "PIC18F2455 Rev. B4"
     /ports : returns a list of port addresses
     /pins : returns a list of pin addresses
     /id : user-writeable 64-bit hex string
     /save 1 : commits pin and module state to flash
     /reset 1 : restore default state
/modules
     /list : list available modules
     /enable s : enable a module
     /disable s : disable module
/pwm
     /0 : control the first hardware PWM
        /rate f : rate in Hz
        /duty f : duty cycle in [0.0-1.0]
/ttl
     /0 : control first hardware TTL
         /open [baud, bits, stopbit]
         /read : return string data
         /write : write string data
         /close
/usb
    /stall : stall detected
    /error : error detected
```

10. HOST TIMING PERFORMANCE

In this section we analyze the timing performance (latency and jitter) for data received by the host transmitted from an input pin of a *uOSC* device attached to a simple switch circuit. The method described in Wright et al. [11] is used, whereby the acoustic signal of the switch activation is recorded with a known latency simultaneously with a signal generated by the method under test. Conditions are repeated with and without background system load.

The data presented is intended to show typical performance attainable with the current configuration and should not be interpreted as the final target goal of this project.

10.1.1 CoreAudio

The CoreAudio path copies the sensor data into a dedicated audio channel, available directly as audio in Mac applications in particular as a signal in MaxMSP. Since primary interrupts and core audio threads are the highest priority operations in OS/X no priority inversion occurs. This represents the highest reliability operating system path for musical applications, and has a consistent input latency of 4 msec and peak jitter of 0.7 msec corresponding to the gesture-input scan rate.

10.1.2 /dev/osc

The /dev/osc path writes the sensor data into a UNIX-style character-device file which is read using standard file I/O operations via the devosc object for MaxMSP (see Footnote 2). Although preemption can delay packet delivery to Max, only a single context switch is required to read data.

10.2 uOSC via Serial Connectivity

The serial driver is known to contain some input buffering so it is expected that this data pathway will not be as fast as the reference platform. Two variations on accessing the serial port data were tested:

10.2.1 MaxMSP serial -> slipOSC

The built-in MaxMSP serial object is used to perform high-rate non-blocking reads on the corresponding serial port. A custom object, slipOSC, decodes the SLIP framing into OSC "fullpacket" messages compatible with the CNMAT OpenSoundControl object.

10.2.2 py-serial to UDP

In this configuration, a Python program reads the serial port, decodes the SLIP framing, and relays the resulting datagram to MaxMSP via the network stack as a UDP datagram.

10.3 Discussion

The py-Serial method is clearly the worst performer (Figures 2, 3), as is expected due to the extra layer of indirection.

It is clearly possible to attain timing performance within the desired latency bounds for musical performance (~ 6-8 msec), however the jitter observed requires consideration. The use of OSC bundle timestamping can be used to compensate for this jitter, and will be an interesting topic for future work.

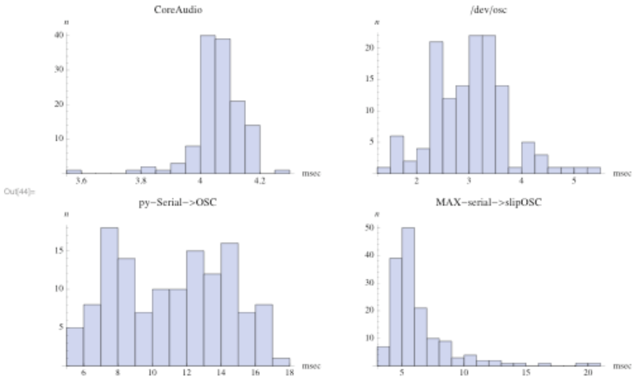

Figure 2: Latency histogram on idle system

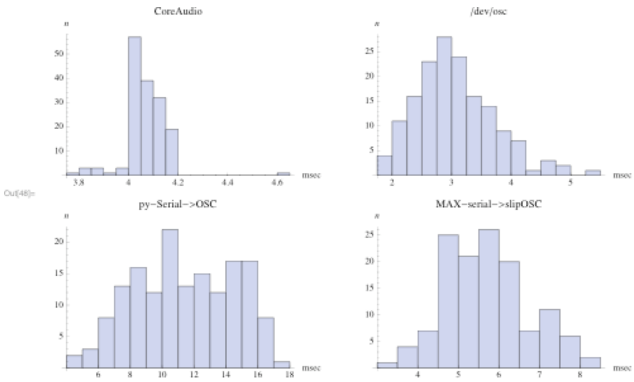

Figure 3: Latency histogram under system load

11. Sample Applications

The *uOSC* platform has been successfully integrated into several new music controllers developed at CNMAT [3]. The compact size of various hardware platforms have also allowed us to retrofit older MIDI and analog devices such as the Max Mathew's radio drum and various foot pedals.

A more sophisticated sensor platform was constructed using custom hardware module extension added to *uOSC* that makes use of the SPI port and other pins to communicate with a 3-axis magnetometer having a digital communication interface. Combined with standard analog input capabilities of *uOSC*, a compact, high-speed inertial measurement module was constructed for research into spatial gesture tracking (Figure 4).

12. CONCLUSION

This paper describes the implementation of OSC, including its advanced timing features and type support, for an embedded microprocessor.

By including deadline scheduling and timestamping, *uOSC* contributes to a large project now underway to implement solid deadline scheduling in future multi-core desktop and handheld device operating systems. [5].

The inclusion of end-to-end latency and jitter performance benchmarking demonstrates the current results with USB-serial in relation to a best-case reference platform, an analysis that the authors consider to be essential for the discussion and careful analysis of any similar implementation.

13. FUTURE WORK

Measurements and tuning on a wide range of host platforms is ongoing. We are exploring other USB device classes such as the CDC-ECM (Ethernet Control Model) and USB-Audio classes, both of which can use isochronous endpoints having improved reliability for real-time applications.

As we release the source code we will support the community's applications and participate in ports to other processors with initial focus on PIC controllers with integrated Ethernet.

The code structure of *uOSC* anticipates the desire to port to new microprocessor targets by isolating platform-independent code components. We are exploring the implementation of *uOSC* on the ATmega controllers employed on the Arduino and Wiring platforms. These implementations rely on a separate USB serial controller instead of using integrated USB. Therefore they cannot implement different USB protocols. They are also more expensive and have slower performance than PIC18F systems for time-critical applications. The Wiring platform, for example, has three different unconnected clock domains. Many Arduino-compatible systems such as the Lilypad use the cheap integrated clock that is neither accurate nor precise. We have achieved some success using forward and backward sychronization on the host side to obviate these problems [8] but we strongly encourage designers of future physical computing platforms to carefully study these timing and performance issues.

14. ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of Sennheiser, the pioneering implementations of Making Things by Liam Staskawicz, and to Dan Overholt who brought the integration advantages of the PIC processors to our attention with his CUI board.

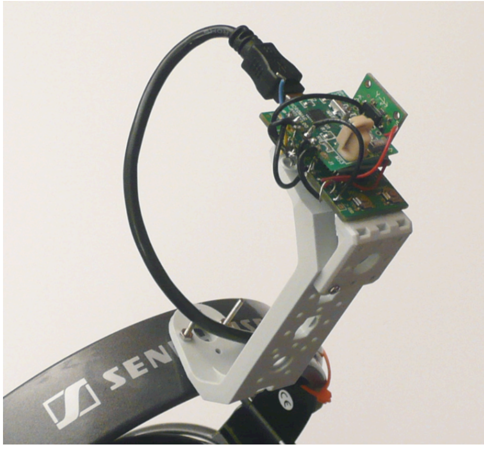

Figure 4: IMU+magnetometer hybrid sensor built on the Bitwacker running *uOSC*, mounted on Sennheiser HD650

15. REFERENCES

[1] Brandt, Eli; Dannenberg, Roger, Time in Distributed Real-Time Systems, in *Proceedings of the ICMC* (San Francisco, CA, USA, 1998) 523-526

[2] Freed, Adrian, Towards a More Effective OSC Time Tag Scheme, in *Proceedings of the OSC Conference* (Berkeley, CA, USA, June 30 2004)

[3] Freed, Adrian, Application of new Fiber and Malleable Materials for Agile Development of Augmented Instruments and Controllers, in *Proceedings of the NIME Conference*, (Genova, Italy, 2008)

[4] Freed, Adrian; Avizienis, Rimas and Wright, Matt, Beyond 0-5V: Expanding Sensor Integration Strategies, in *Proceedings of the NIME Conference* (Paris, France, 2006), 97-100

[5] Hayes, Brian, *Computing in a Parallel Universe*, American Scientist, Volume 95, Issue 6, 2007, 476-480

[6] Overholt, Dan, Musical Interaction Design with the CREATE USB Interface: Teaching HCI with CUIs instead of GUIs,, in *Proceedings of the ICMC* (New Orleans, LA, USA Juny 11 2006)

[7] Romkey, J., *A Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP*, RFC 1055, http://rfc.net/rfc1055.html, 1988

[8] Schmeder, Andy and Freed, Adrian, Implementation and Applications of Open Sound Control Timestamps, in *Proceedings of the ICMC* (Belfast, Ireland, 2008)

[9] Wessel, David and Wright, Matthew, *Problems and Prospects for Intimate Musical Control of Computers*, Computer Music Journal, Volume 26, Issue 3, 2002, 11-22

[10] Wright, Matthew, *The Open Sound Control 1.0 Specification*, http://opensoundcontrol.org/spec-1_0

[11] Wright, Matthew; Cassidy, Ryan J. and Zbyszynski, Michael F., Audio and Gesture Latency Measurements on Linux and OSX, in *Proceedings of the ICMC* (Miami FL, USA, 2004) 423-429

[12] *The Universal Serial Bus Specification Revision 2.0*, http://www.usb.org, April 27, 2000.

[13] *MCHPFSUSB User's Guide*, DS51679A, Microchip Technology Inc., 2007