A Wireless Sensor-based Mobile Music Environment Compiled from a Graphical Language

by

Robert N. Jacobs

S.B. EECS, M.I.T., 2003

Submitted to the Department of Electrical

Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 2007

Author .....
Department of Electrical Engineering and Computer Science
August 24, 2007

Certified by .....
Joseph Paradiso
Associate Professor, M.I.T. Media Lab
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

A Wireless Sensor-based Mobile Music Environment Compiled from a Graphical Language

by

Robert N. Jacobs

Submitted to the
Department of Electrical Engineering and Computer Science

August 24, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis I demonstrate a framework for a wireless sensor-based mobile music environment. Most prior work has not been truly portable. Those that were have focused on external data as opposed to properties of the listener. In this project I built a short-range wireless sensor network (using the ZigBee protocol and an accelerometer) and a compiler for PureData, a graphical music processing language. With these parts, I realized a synchronized music experience that generates a soundtrack based on the listener's movement. By synchronizing the music to the user's natural rhythms, it encourages the user to maintain a given pace for a longer period of time. I describe extensions to this example that point to a future of portable interactive music tied to exercise and physical activity.

Thesis Supervisor: Joseph Paradiso
Title: Associate Professor, M.I.T. Media Lab

Acknowledgments

I would like to thank the following people whose assistance has been invaluable:

Joseph Paradiso, for giving me the opportunity to work with him and his lab on an enjoyable thesis, and for the knowledge he imparted in MAS.836 (Sensor Technologies for Interactive Environments)

Mark Feldmeier, for extensive advice and support, the original idea, and soldering big wires to little tiny cramped test pads

Manas Mittal, for pointers on ZigBee

John Anckorn, of Nokia, for his help adapting the N800 to my needs

Mats Pettersson, of IAR Systems Software, for his helpful advice on IAR's compiler systems.

My family, for their support over the years

Kelsey Byers, for all she's done and helped me with

Contents

Chapter 1

Introduction

1.1 Application

Many people with portable music players use them to generate soundtracks to their lives[6]. What would be more appropriate than being able to generate these soundtracks based on the activity being done at the time? If the user went jogging, a system could synchronize the music to the rate at which they were going. Alternately, it could encourage the user to speed up or slow down as part of a exercise program[3]. This responsive environment would be able to syncronize to any periodic motion, such as their warm up or cool down period[16]. By allowing the user to set the pace in an intuitive and straightforward manner, they would not be subconsciously pushed to go through their exercise regimen faster than would be healthy. Many similar applications can be found for the combination of a wireless sensor network and a small handheld computer running a compiled version of a signal analysis and music generation program.

ZigBee is a recent wireless standard used for low data rate communication between arbitrary devices. By using a ZigBee-enabled microcontroller with an integrated analog-digital converter, arbitrary remote data can be measured and sent to a central computer to do the sound generation. Having written a compiler to preprocess the musical analyzer, this computer can be a small portable device that is easily carried around, distinguishing this from previous work which used a central server[9][8].

Beat extraction, or the process of automatically finding the beats in a piece of music, is a subject into which considerable research has gone with varying levels of success for varying applications[2]. Fortunately, the application outlined in this thesis does not require

much: we only need to know when the user is stepping, and the acceleration data from such an action is appreciably more periodic than music[4]. Later versions may want to take extra data from multiple accelerometers and calculate additional features (to enable more complex mappings) or other clinical sensors (such as a heart rate monitor).

1.2 Compiler

PureData[1] is a versatile language which was primarily designed for audio processing, although various extensions allow it to handle OpenGL-based graphics[13] or random-access full-motion video[12]. However, the fundamental design of PureData's runtime allows certain actions to be calculated in bulk (so called "signals", typically carrying audio data) while others are much more computationally intensive due to the amount of overhead from PureData's interpreter ("messages" – short lists of words and numbers).

Even though PureData's system requirements are much cheaper than those of its parent language Max/MSP (PureData can run on Linux while Max/MSP is only available for MacOS and Microsoft Windows), one may desire a smaller or less expensive system. Unfortunately, many inexpensive or smaller systems have tremendously lesser computational ability, and (in the case of an algorithmically complicated patch) PureData's interpreter may be the deciding factor as to whether the patch can run real-time.

Previous work in this direction has included PureData Anywhere (PDa)[7] – a port to embedded systems. Due to the common absense of floating point hardware on such processors, PDa made a compromise: floating point math would continue to be used for message logic, but 13.19 fixed point math would be used for the audio signals. This requires a dramatic difference in the function of two PD objects (`tabread~` and `tabwrite~`, which read and write audio data to look-up tables), and doesn't help for control-heavy applications because the use of floating point emulation for control signals is very slow.

There are many applications for which these compromises are very inconvenient. Control-heavy patches, such as ones that use a MIDI interface with frequent continuous control, will run very slowly due to the floating point emulation. Patches heavy on presampled data may run into difficulty with the 8 second (2^{13} ms) limitation for `tabread~` and `tabwrite~`.

To address these performance issues, I decided to write a compiler for PureData. Max / MSP, which PureData is conceptually derivatived from, natively supports multiple different

data types. Max also dates to an era when compilation was necessary for every computer. This versatility in data types in Max frequently causes more confusion as to why the math is not working as desired than benefit in faster execution. My compiler PuDaC (PUre DAta Compiler) divides data into the (approximately) same two types that PureData does: high bandwidth signals (audio) and low bandwidth signals (control).

For superior performance on embedded systems, a model more like MSP's dynamic compiler proves beneficial – by being able to do compile-time optimization, dramatically superior performance should be available. Very brief experimentation comparing PD's speed of multiplying floating point numbers to C's showed PD being slower by a factor of 1000 on otherwise identical hardware.

Chapter 2

Concept for Use

The system might be used as follows:

Alyssa, a musician, has been producing music using PureData for years, and is a well-known interactive composer in the genre. She is approached by CellCo, a mobile telephone provider, to produce an exercise mapping to be used in their newest phone, which includes support for ZigBee, allowing wireless sensors to be easily integrated.

A few months later, Alyssa has produced a mapping she is happy with, and sends it to CellCo. CellCo then uses my compiler (PuDaC) to condense her patch into C source, and any optimizing C compiler to compile the resulting C program into an application that will run on their phone.

A few months after that, Ben goes to the CellCo store to get a new phone, and sees their new phone with a novel feature: ZigBee, which (as he's read) can be used to communicate with sensor networks. Wireless temperature sensing networks have been appearing recently, and allow him to know the temperature immediately outdoors of his overly air conditioned office, so Ben is already familiar with this concept. This cell phone comes with a additional part: a wrist-watch sized device with ZigBee and a 3-axis accelerometer that Ben can wear. Ben buys this phone, and brings it home. Once home, he digs out the miscellaneous peripherals and explores the bundled features of his new ZigBee-equipped phone.

In one menu of the phone, he find's Alyssa's patch, described as "SynchronousMusic", and, curious, he starts it. He is told to take the accelerometer, put it on his wrist or ankle, and turn it on. Moments later the accelerometer has found the cell phone and starts feeding it acceleration data. The cell phone then instructs Ben to start walking, and once he does

so it starts playing music that is synchronized to his movement - whether swinging his arms or stepping with his feet. Ben finds this sufficiently entertaining that he starts wearing it to the gym, since he finds it helps him keep a constant pace in an intuitive manner without pushing him too hard on his exercise routine. Ben's doctor hears about this new system from Ben, and begins using it with his older patients to encourage them to exercise with music that plays at a healthy speed for their more gentle exercise requirements. But he has a concern: there's no direct feedback from the user's heartrate, so the patch can't react accordingly. He suggests this to CellCo, and they discuss this with Alyssa.

Later on, Alyssa produces another patch for CellCo. When she made the original patch, they only provided her with a single accelerometer, but she's since had ideas for what might be done with multiple sensors. This patch includes integration with a heart rate monitor (persuant to Ben's doctor's concerns) and multiple accelerometers, allowing a much wider gamut of sounds. The advice Ben's doctor gave has been passed on, and she adds the ability for the program to encourage the user through an entire exercise regimen, from warmup to a variety of paces and exercises to cool-down. One needs to move to experience this new art form – not just sit and listen.

In another situation we might find my system used in a different but analagous way. Miller is a member of a street performing dance group that often appears in parades. Along with his dancing role, Miller also helps choreograph music for his group to dance to. Miller has seen stationary dance performances with wireless sensors on the dancers, and would really like to adopt this system for the dance group, but finds the mobility issues difficult to deal with.

Doing some research with his friends in the music world, Miller learns about a new option for PureData, a program he's been testing for writing music for his group. Miller finds that he can use a new compiler (PuDaC) to compile the PureData patches into a small program that can be run on a handheld or phone instead of running PureData natively on a desktop or laptop machine.

Combining his knowledge of wireless sensor networks for dancers and his new discovery about PureData, Miller is able to set up a system for his dance group to bring responsively choreographed music along with them on their parade routes. This makes the dancers' time easier, as they no longer have to choreograph everything precisely, and it also means Miller doesn't have to haul a heavy laptop along while he's dancing.

Chapter 3

Rationale and Background

3.1 Background

While each individual part of this project is not original, I believe the combination of an inexpensive platform with a wireless sensor system will allow for many new applications, particularly when combined with my compiler for PureData. This puts the functionality of a powerful music processing language in a small package at the user's disposal.

3.1.1 Sensor systems

In the past, many sensor systems – either fully wired (like some of the very first systems), wired networks transmitting via wireless[8] or fully wireless[9] have fed information to a central computer that analyzes the data and generates audio. This is fine for performance or other applications with small active areas, but does not allow for an experience that follows the user, particularly in cases where the user cannot accomodate heavy equipment such as a laptop.

To interact with the program, the user needs a set of wireless sensors that are light, have reasonable battery life, and are non-intrusive.[17] In this implementation I only have accelerometers, but many other sensors should be straightforward additions.

One of the earlier systems to achieve this set of requirements was Paradiso et al.'s CyberShoe[15], which measured myriad properties of a dancer's feet (acceleration, pressure, etc.). However, it was somewhat bulky, didn't accomodate a scalable, channel-shared network, and many of the sensors required extensive modifications to the shoe, so a less

intrusive solution would be useful. One follow-up project that the group explored with instrumented shoes was a medical application in real-time gait analysis, where music would dynamically change while walking to encourage patients with injuries or illness to maintain a healthy gait[19]. The sensor packages in this example were complex and expensive, and the demo required a laptop. Finally, the recent popularity of Nintendo's Wii interface[21] points to the potential market for this system. While the Wii is tethered to a television, our environment is truly mobile.

Another of the group's subsequent projects, the Sensemble[20] demonstrated a channel-based inertial measurement unit (IMU) array for interactive dance, but the system accomodated over 25 IMU packages, was much more expansive than that described here, and needed a powerful desktop or laptop computer for sensor interpretation and music generation. Other previous projects explored switching tracks or changing tempo with exercise pace[3][4] but didn't explore generative music.

3.1.2 Immersive environments

Many people currently own portable music players, whether Apple's extremely popular iPod or another manufacturer's. Frequently, these people use the music on their music players to form a soundtrack for their life – during their commute, their work, even sometimes during their vacation.[6]

Jones and Jones[10] have built a system to appropriate a listener's music such that they interact with their environment. Using GPS data, they change the balance of the audio to guide the user to various targets. However, this system is much more oriented around taking fully external data (a path for someone to follow) and queuing the user in a non-intrusive manner.

Gaye, Mazé, and Holmquist[5] invent a world very similar to what is envisioned for my project, involving deciding on a context from sensor data, and emphasing natural rhythms heard by the wearer. However, their emphasis is more on integrating into the listener's city experience, and less on exercise.

The system I have designed takes the concepts of the personal soundtrack and how it can be used to guide the user to another level. By allowing the user to self-direct, they can control the level to which the personal soundtrack affects their actions - do they speed up or slow down based on the audio feedback, or do they remain at a steady pace? The user

and interactive experience composer dictate how the user's activity (present and past) will map into music.

3.2 Rationale

As there are two major halves to my thesis, there are two sets of reasons as to how it is important.

The compiler presents a middle ground between PureData and a lower-level language like C, with much of the ease of use of PD and much of the speed of C. By using a highly optimizing C compiler, many of the inefficiencies due to mechanical translation are further eliminated. For example, many objects in PD patches have exactly one incoming connection. A good C compiler, if told to optimize sufficiently, will take these objects and put them inside their callers. Further optimizations would then go and find redundant checks on the data type of the incoming message and discard the second set of checks.

As the resultant application retains little of the original patch's appearance (and, once optimized, little of the original patch's file format), the compiled form could additionally be used as a layer of tamper resistance, since the recipient has no easy way in which to add or rewire objects or examine the contents of data. This has its drawbacks, however: since the runtime little resembles the original, any bugs in the patch would have to be retriggered in the original copy running under PureData. Additionally, the recipient cannot learn anything from the patch, unlike Max/MSP's standalone version.

Technology has now progressed to the point where it is possible to put a radio in a very small area, and as a result, small wireless applications are smaller and easier than ever before. Furthermore, the ZigBee protocol is versatile enough (if more than is necessary for some tasks) that many wireless sensor applications are now feasible for widespread deployment.

Since music is well-known to help people get through tedious tasks, such as exercise, it seems obvious to use music as an incentive for them. However, unlike an album or the radio, a dynamic system can generate music to synchronize with or give feedback to the user. This is important, because the tempo of the music on an album or the radio can vary, which can be annoying to the exerciser, particularly someone in serious training. If it's too fast, it is potentionally unsafe, as the user might push themselves harder than they

should. If it's too slow, they may not get a good workout. Enabling this extra layer of feedback that can dynamically push or pull the user's pace and not just blindly synchronize should allow people to get into a groove and enjoy the most of their workout. Many people who exercise at a certain speed have a specific set of music they listen to for its synchronicity with their movements; allowing them to create their own equivalent provides a flexible alternative. Exploring timbral as well as rhythmic feedback adds many new layers of information, perception, and immersion, ideally making the musical accompaniment more immersive and the union of motion and sound much deeper.

With the new ways this system provides to interact with the users' phyiscal activity, and the ease of embeddability portable from having a compiler, we enable a whole new creative medium for people to experience and musicians to compose in.

Chapter 4

Design and Implementation

4.1 Preexisting Components

4.1.1 PureData

PureData is very good at providing a system for rapid development of audio generating programs. By providing high level abstractions of audio and data processing functions, it is easy to build a variety of complicated programs that make a variety of interesting and useful sounds.

Since my platforms were made to be adaptable but lack much in the way of a user interface, I needed something to develop the mappings from the sensor data to an audio environment. PureData's saved file format is sufficiently cryptic that writing patches by hand would be intractable, and since the design was specifically for a program that would use PureData's pre-existing paradigm, PureData is rather necessary.

4.1.2 Nokia N800

Nokia's N800 is a small portable computer (measuring 2.95 x 5.66 x 0.51 inches, weighing 7.26 ounces). It includes bluetooth, 802.11g, an ARM11 processor, a dedicated DSP, and runs the Linux-based Maemo internet tablet software suit. It shares general features with similar handheld computing devices, so the exact choice of platform is flexible.

There are a variety of small single board computers that are capable of running Linux – for example, Gumstix makes a suite of Intel XScale based boards that range from slightly less powerful than the N800 to dramatically faster[18]. However, the Gumstix computers

Figure 4-1: Front and back of N800, with serial port exposed in open back showing our custom adapter

do not have their own battery, and the N800 has a floating point unit, making the amount of initial investment to get a useful system from an N800 much lower. Furthermore, the N800 was much more easily obtained.

However, my requirements were sufficiently nongraphical that I could have used any battery-powered Linux machine with an FPU and a serial port (the N800's is in figure 4-1). That said, future incarnations of the system will benefit fron the N800's interface in customizing the user's experience, much in the way parameters are set and adjusted on conventional music players. Additionally, I've minimized platform dependencies (beyond network in and audio out), and so it should be straightforward to adapt my work to any other platform regardless of the overall requirements.

4.1.3 Texas Instruments (TI) CC2431

The sensor system looks very straightforward. Each sensor should use a microcontroller, a simple radio, and an accelerometer, preferably three-axis. When I was introduced to the CC2431, it looked like an ideal solution, because it had almost everything I would need already in it. It's tolerant of a wide range of voltages and so could be run directly off a battery, contains an ADC, and contains its own integrated ZigBee radio. All I would need to add would be the accelerometer.

TI provides their ZigBee implementation free of charge, as a method to sell their ZigBee-capable microcontrollers. Furthermore, unlike a simpler off-the-shelf design (such as a 9600 bps wireless modem), this is much more extensible to a large number of sensors and has better guarantees of data transmission.

I use one CC2431 as a coordinator to initialize and gather the sensor data from all the others.

4.1.4 TI's Zstack ZigBee implementation

The wireless protocol has a few constraints: many transmitters, one receiver, no real need for collision detection, low bandwidth per transmitter (less than two kilobits per second), moderate-to-low aggregate bandwidth (less than a megabit per second), low latency, and permissibility of dropped data. Each sensor needs a coin cell, a low-power microcontroller, a transmitter and the sensor for that system. Texas Instrument's ZigBee system was chosen.

Texas Instruments provides, without charge (although with rather restrictive licensing terms) a software suite they call the Z-Stack. This system provides a complete implementation of the ZigBee 2006 protocol, versatile enough for any use. However, the system is sufficiently large (since the standard is so complex) that it easily fills the vast majority of the 128kB of program space available to the program.

ZigBee is a complicated standard designed for a variety of low-bandwidth applications. Multihop mesh routing is a primary feature, although completely useless in this specific application since it imposes a significant latency cost. General applications include home automation, smart objects, and any other application where dynamic sharing of information between devices is useful.

I used ZigBee to transfer data from the sensors to the coordinator.

4.1.5 SparkFun IMU

The 5-axis accelerometer/gyroscope ("IMU", Inertial Measurement Unit) board from SparkFun Electronics contains an Analog Devices ADXL330M and an InverSense IDG300.

The ADXL330M measures $\pm 3.6\ g$ on all 3 axes, and is configured to give a 50Hz bandwidth. The IDG300 measures $\pm 500°/s$ about the two axes not normal to the chip, and has a 140Hz bandwidth.

Accelerometers provide approximately the only way to determine how the user is moving without an external fixed reference.

I use this IMU to measure the user's footsteps, since landing on the ground is a large change in acceleration; however, the system should also work relatively well on other parts of the user's body, such as the wrists.

4.2 Modifications

4.2.1 Wireless firmware

The Zstack examples usually assume the user is running the example on a development kit which contains 6 buttons and 4 LEDs. The GenericApp example provides a simple interface: the ZigBee node will start, find an appropriate peer upon one button being pressed, bind to that peer upon another button being pressed, and will then send a packet containing "Hello World" every 15 seconds. Upon receiving a packet, it would display the contents on the LCD in the evaluation.

It was easy to modify this code for my purposes: rather than having initialization tied to two buttons, I instead have it automatically initialize over 3 seconds – 1 second after the network stack starts, it finds a peer, 1 second after that it binds to that peer, and 1 second after that it starts sending data. This could certainly be sped up, but I chose the timing to be cautious after my initial problems using the Simple API. After it's bound, it starts sampling at 100 Hz (a reasonable rate for the application, though it still causes aliasing from the gyroscope) at its highest accuracy. Unfortunately, their implementation, involving a sigma-delta ADC, produces significant amounts of noise in the output reading even on just battery power, and the time to take one (highest accuracy) sample (250μs) is slow enough that it would be difficult to take multiple samples for filtering to achieve good noise blocking, good latency, and high accuracy on all 5 channels from the IMU while ever returning from the poll subroutine (to allow the radio to work!). Furthermore, there is a maximum aggregate report rate across all remote sensors due to the fixed total amount of bandwidth available – something less than 1000 reports / second. Problems with ?? and the need for a very high data rate can be alleviated by having the embedded controllers sample the sensors faster and report aggregated featuresa to the N800 rather than raw data streams.

The sensors report to a central microcontroller that is directly attached to the N800. This central CC2431 then encapsulates the data using the SLIP[11] (Serial Line Internet Protocol) format, even though what is being encapsulated is not IP. This was done to support arbitrary and variable length data without any particular worry about framing, even in a noisy environment.

Unfortunately, I did not have the chance to figure out how to make each sensor only bind

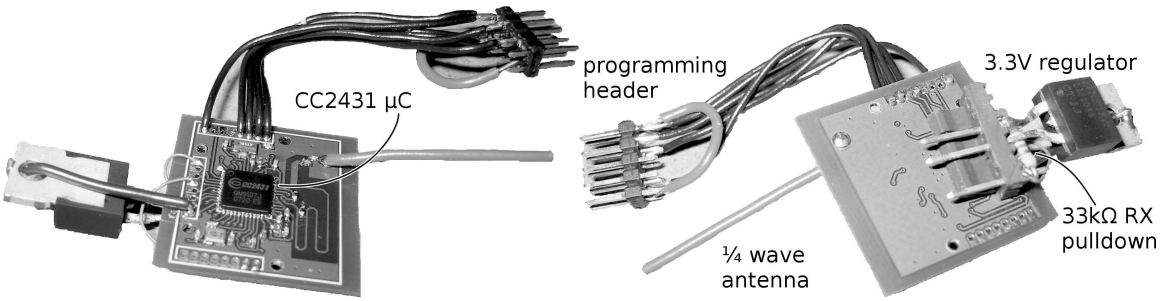

Figure 4-2: Front and back of the RF microcontroller to be attached to N800

with the intended receiver – the current code has a node associate with whichever other node it finds first. This is, again, fine for demonstration purposes, but limits its utility in application.

4.2.2 Circuit to attach one CC2431 to the N800

The bridge between the N800 and our RF network (pictured in figure 4-2) contains a low-power microcontroller and a ZigBee transceiver. It would be possible to add additional sensors onto the bridge.

More interesting mechanically than electrically, this involved attaching wires to the test pads inside the N800, using a LM1086 to regulate the LiIon battery voltage to 3.3V for the CC2431, and bending things into a nice flat shape to fit nicely behind the N800.

The N800 contains a serial port, apparently included for debugging the system in the factory. By using the flasher tool enable the serial port, and disabling the getty normally running on the serial port, I can use the serial port to receive arbitrary low-bitrate data (115200 bps, 8 bits per byte, no parity, one stop bit – this is restricted, and appears to be an intentional crippling of the driver in the Linux source tree). Transmission is also possible, although the kernel and applications on the system send large amounts of debugging messages to the serial port and would have to be silenced first. The pictured 33kΩ resistor is needed because the N800's serial port would pick up ambient noise on the receive line, in turn causing it to reset.

The hardware serial port is very convenient since both it and the CC2431 support 3.3V logic-level serial. The N800 also supports switching its USB port to run in host mode, but using the internal serial port is simpler and smaller.

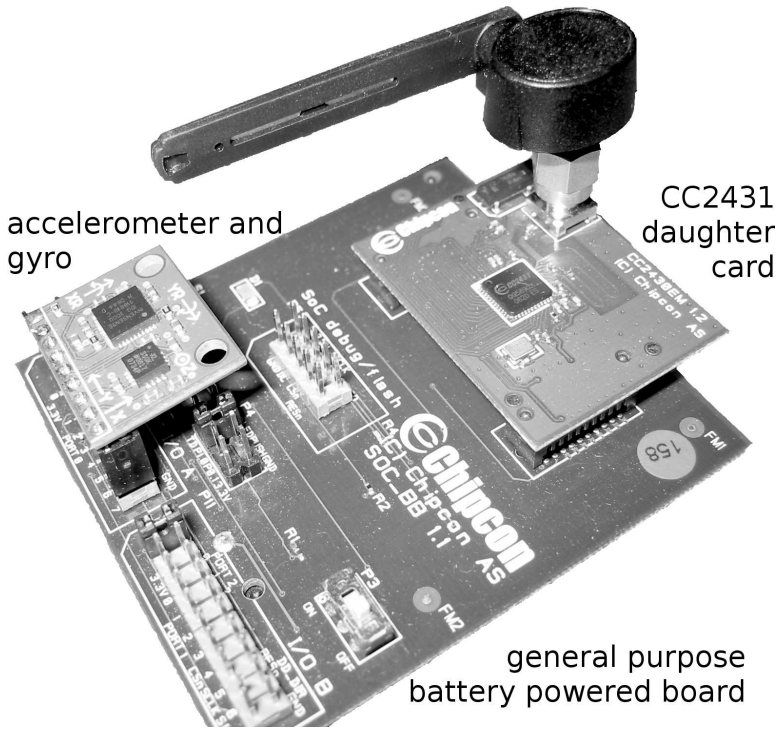

Figure 4-3: Prototyping board attached to accelerometer board

4.2.3 Protocol translation program

The reports from the microcontroller are sent as a 16-bit sensor address followed by a series of 16-bit words. The program converts this into the PureData-style `[addresssymbol] [space separated array of decimal numbers];`. It also handles the logical connection to the serial port and sends the resulting PureData-encoded datagram to a UDP destination of the user's choice, presumably to PureData or the compiled patch.

PureData does not easily support binary data, so an external converter simplifies things tremendously. If I were just attempting to use PD (or PDa), it might have made sense to have just made an external module, however since my goal was to compile a PD patch into a program, it made sense to only use built-ins – i.e., `netreceive`, which receives data from the network into PD.

4.2.4 Circuit to attach IMU to development kit battery board

Purely of mechanical interest, I removed half of the pins out of a 10x2 dual-inline female header to attach the IMU board to the male header on the demonstration battery-powered board.

The battery board (shown in figure 4-3) provides access to the CC2431's ports 0, 1, and 2, but in a somewhat strange manner; the left set of headers provides access to the 8 pins on port 0 (which are the 8 ADC channels), two pins to the positive terminal of the battery, and one ground. A two-row connector had to be used because the top row contains the only ground pin, while the bottom row contains all the port 0 pins. The two-row connector is also useful because it provides greater vibration resistance – if the accelerometer were attached to the battery board via a set of wires, the stiffness of these wires would produce a damped-mass vibratory system that would add unwanted noise to the desired signal. Of course, any real commercial implementation of this system would use a much smaller and streamlined custom circuit board.

4.3 PuDaC

PureData's performance at signal processing is hard to beat, but control signals in it come with a heavy overhead, due to the way the interpreter works. My compiler converts tha vast majority of data into native C constructs with a marked improvement in processing efficiency. I demonstrate this improvement in the next chapter.

Since I am converting between one format to another of text, I have written my compiler in Perl. Perl excels at parsing text, especially rigidly defined text like PD's save format. After parsing the entire file into a structure in memory, I execute the generators for all the objects that produce the C code for each.

The compiler takes in a plain text PureData patch file and produces C output. This allows me to take advantage of the large amount of work that other people have put into optimizing compilers without having to implement it myself.

PuDaC replaces each object with a uniquely-named subroutine (and possibly some uniquely-named globals). Each "wire" connecting objects is replaced with directly calling the connected object.

I made a few design decisions to simplify parsing: for example, there are no Anythings. I thought Anythings were messages for which parsing is deferred – since in my design I parse almost all messages at compile time, there is a disadvantage to deferring said parsing. Every data type in PureData actually has a word at the beginning (such as "list" or "symbol"), and a specific few words mean specific things to the PureData runtime. However, any

message can contain an arbitrary first word, and thus an arbitrary type. Unfortunately, this simplification posed a number of problems, and will need to be undone in a later revision.

Chapter 5

Evaluation

In doing this thesis, I learned a number of things. I learned how conventional compilers work, and that there is very little published research into making compilers that target things other than machine code. As a result, much of my background research into the compiler was not applicable, although I was able to find plenty of information on wireless sensor systems.

In writing the compiler, I needed to acquire an extensive comprehension of how PureData worked. I now have a strong grasp of how PD is put together. Specifically, I learned how the object model of PD is built in plain C, and how the limited amount of inheritance works in PD's object-oriented system. For example, large numbers of objects in PureData are classified as binary operations, and all behave similarly. They typically have two inlets, one outlet, are frequently based on operations easily expressed in C, remember the last number in each inlet, and, if given a list of two numbers in the left inlet, distribute the second number to the right inlet.

The sensors were a pointed reminder that there is a tradeoff between measurement range and sensitivity due to constant magnitude noise sources. The large amount of noise in the implementation of the ADC in the microcontroller makes a gain adjustment necessary for the accelerometers. A more accurate ADC would increase sensitivity but might negate many of the cost advantages of the system.

Several issues came up while working with the PureData compiler. For example, PureData message boxes provide a simple list manipulation tool: they can reorder, insert, and select arbitrary elements of an incoming list. However, PD messages frequently include an

explicit cast word at the beginning of the list, like `float` (floating point number) or `symbol` (arbitrary non-number), and this word instructs the parser how to interpret the message. If a message starts with a number, the message is treated as a `float` or `list` (array of arbitrary things) without a cast word, depending on whether there is more than one element. This implicit casting produces unexpected results when using "dolsyms" – dollarsign symbols.

Dolsyms in PD are like those in Max or any UNIX shell. For example, $1 in a message box extracts the contents of the first element of the incoming message. However, the implicit types that are added in front produce confusing results. In figure 5-1, there are 3 message boxes and a print object. If you click on the message box containing "1 2 3" (an implicit list), you get the outputs "1 2 3" and "1 2". However, if you click on the message box containing "a b c" (an anything), you get the outputs "a b c" and "b c". Why "b c"? Because the message box "$1 $2" looked at its input, saw the first element, treated it as the type, and returned the 1st and 2nd elements of the message. Miller Puckette says:

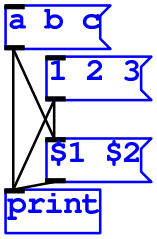

Figure 5-1: type troubles

> "Messages in Pd are somewhat artificially divided into two classes. First are data-holding messages (bang, float, symbol, list) which are the primary way of communicating between objects. Second is 'everything else' (you could call them out-of-band messages or metamessages) that describe changes in configuration, read and write files, quit Pd, etc. These are provided so that complex objects don't need to have 100 separate inlets for every possible functionality. It's not clear whether this was a good design choice, but it's entrenched.
>
> The distinction becomes visible, and ugly, when the leading item in a data-holding message is a symbol (i.e. word, fbar). In this case, to disambiguate it from the other sort, the printed form of the message has a selector, 'list' or 'symbol' prepended to it. Underneath, there is always a selector in front of data messages, but it is implied if the first data item is a number."[14]

In any case, since the compiler automatically promotes all Anythings to symbols or lists, it can't know whether it should skip the first element or not.

`route` objects stumble into the same problem – if the patch uses a `route` as a type router

(e.g. `[route float symbol list robert]`), due to my discarding Anythings, there's no way for me to determine between the anything `[robert jacobs]` and the list `[list robert jacobs]`. Currently, I work around the problem by making `symbol` and `list` match with lower priority than any other option. This certainly doesn't help in the case of using `route` to catch Anythings, since they no longer exist.

In the end, however, the desired system was created. There are several parts that still need polish, but overall the system performed as desired - faster actuation of the accelerometer generated music with a faster beat, slower acutation generated slower music, and changes in actuation speed were successfully detected and integrated into the musical feedback.

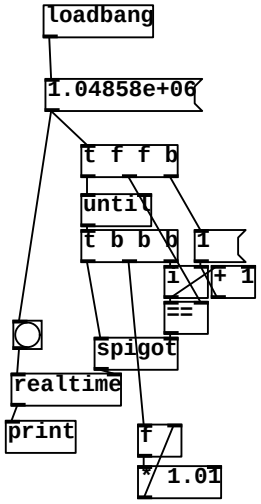

Figure 5-2: Compiler performance test patch

The compiler runs in two passes (three if you include the additional stage of running gcc). First it parses the input file, loading all the objects into an associative array with a UID for the object as the key. The value of each entry is an another associative array with several predefined entries specifying the object's arguments, the C representation of the objects attached to the inlets and outlets of the object, the C-generating perl code for this object, and a redirection specifying the object has another name (like `sel` / `select`). Then it prints a prologue, executes the C-generating perl code for all objects, and prints the main function. This model makes debugging tremendously easier, although it is probably significantly less efficient than is possible.

Even so, a simple test patch (shown in figure 5-2, it does one million floating point multiplies) shows a significant performance increase over the plain interpreter: on an Athlon (Thunderbird core) running at 1066 MHz, PureData takes an average of 533ms, compared to as little as 158ms for the optimized version of the compiler output, about 30%. (Histograms of trials are in figure 5-3. The variance was consistently $\frac{1}{20}$th of the mean, probably due to the way the Linux scheduler works)

The final mapping patch is shown in figure 5-4. This patch generates a simple drum track

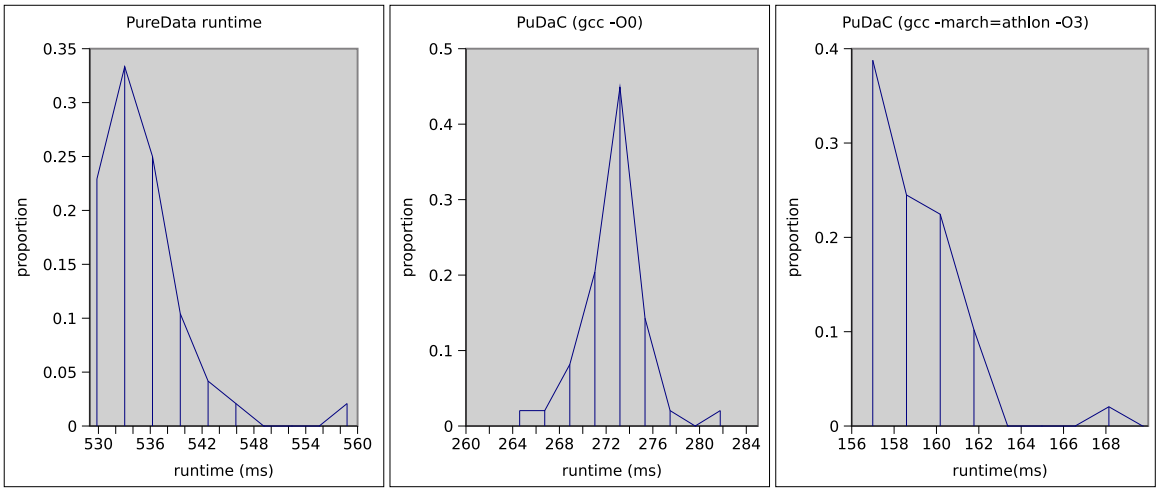

Figure 5-3: Histogram of time to multiply 2^{20} floating point numbers, left to right: Time to run the patch in just PureData, time to run the compiled program when compiled in GCC using the "no optimizations" flag (-O0), time to run the compiled program when compiled in GCC using the maximal optimizations (-O3) and generating code targeting my specific computer

that synchronizes to the user's motion. The data is accepted, routed, and the acceleration and angular velocity from the IMU are converted to jerk and angular acceleration. We then compute the ratio of local mean to local average deviation of the magnitude of both, look for when that exceeds a given threshold, and feed this output into a delay with holdoff. The delay with holdoff feeds into a phase-locked loop (PLL) that attempts to match the phase and frequency of the input, which is attached to simple logic to run the audio loop.

An example showing how well it works can be seen in figure 5-5. The Z axis of the accelometer (labeled a_z) was oriented normal to the leg, in the direction of stride (because no rotation was expected about this axis). The X axis (a_x) was parallel to the leg. |Jerk| and |Angular Acceleration| were calculated as the magnitude of the first backwards differences on available axes. The detected and predicted footfalls plots indicate a detection or prediction when they change from low to high or vice versa. The period ranges from 0 to 2 seconds. Both plots are approximately 40 seconds (4000 centiseconds) long. The gap seen from 16-17 seconds in the bottom graph is because the data dumping program momentarily paused.

For jogging data, the both the gyroscope and accelerometer data provided a good impulse source, and the detected footfalls plot shows this. Unfortunately, as can be seen in both plots, the predicted period oscillates with a period of 10 steps, never successfully really

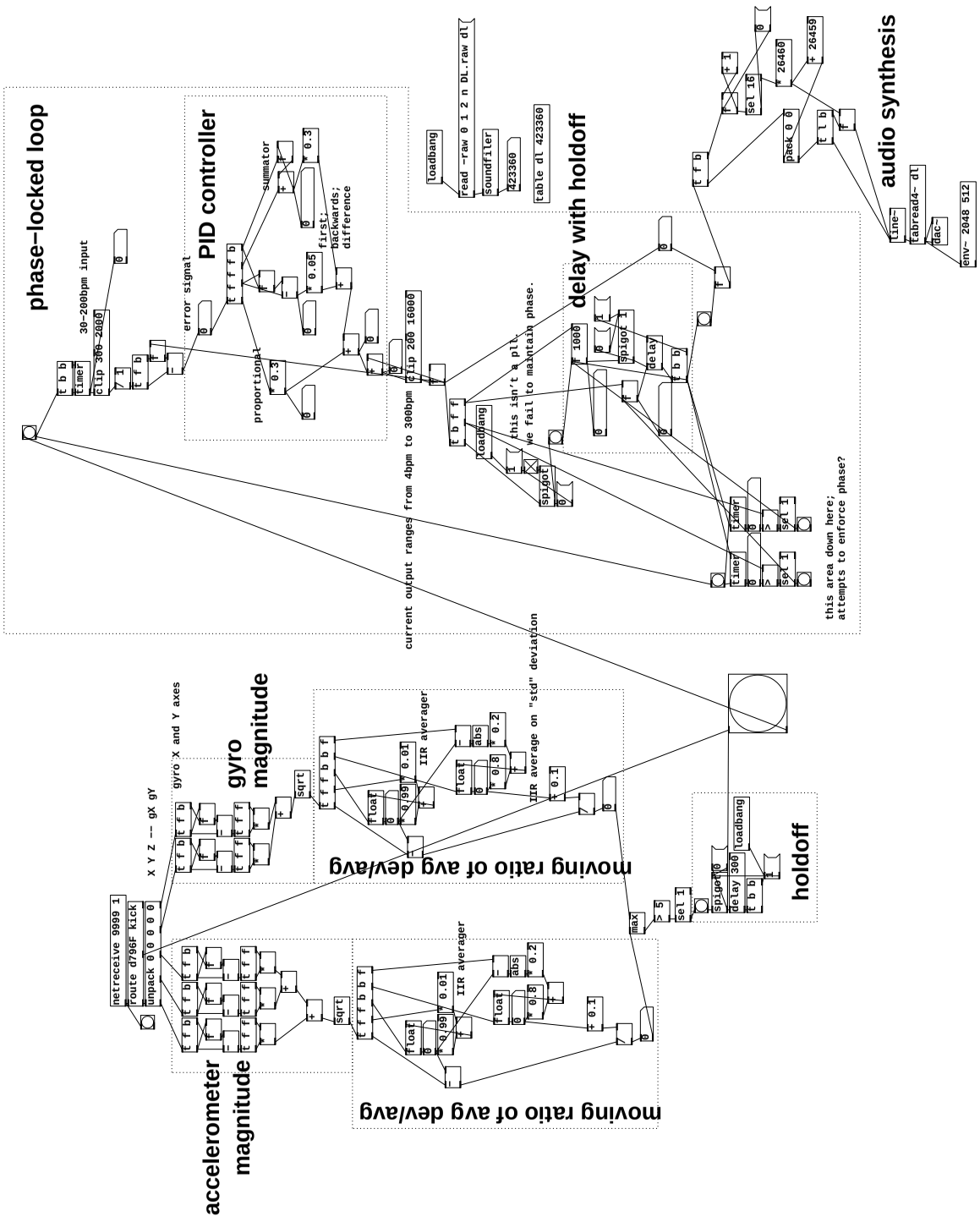

Figure 5-4: The final mapping used – a drum track is synchronized to the user's motion

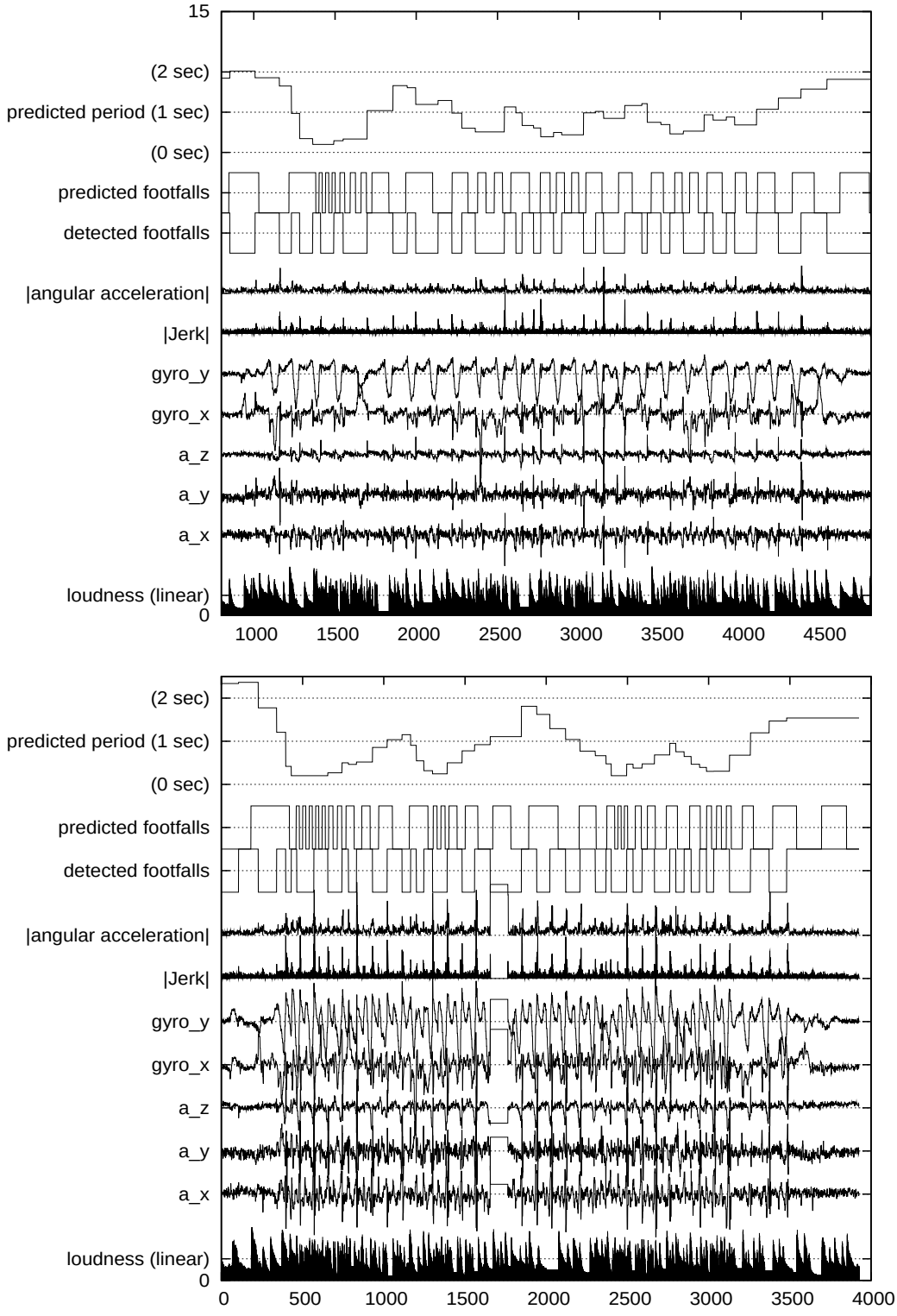

Figure 5-5: Graphs of results for walking (top) and jogging (bottom)

getting to the true pace (about 46/min when walking, 67/min when jogging). This is solely the fault of the PID (proportional integral derivative) controller inside the phase-locked loop. Later tuning should make this react better, but it is hard to adjust well.

The predicted footfalls are used to trigger the beats of a 4-measure long drum loop. Later patches would easily add other voices and more control.

Chapter 6

Conclusion

6.1 Future work

6.1.1 PuDaC

Currently, there is no support for external patches or subpatches. Since PureData starts counting from 0 in each subpatch, and my implementation doesn't recurse or handle multiple instances, the patch currently has to be flattened by hand before it can be compiled. Fortunately, flattening is a relatively easy, if tedious, task, only needs to be done once for each patch, and (if the patch is known to be destined for PuDaC) can be avoided altogether by not using subpatches. Furthermore, implementing subpatches in PuDaC is not difficult but was tremendously further down on the author's list of priorities.

For efficiency and putting similar code together, I should aggregate similar objects. In PD, all the so-called "bin(ary)op(eration)s" inherit from a common class, so that each has only small fragments of C to specify the minor differences between them.

I have implemented a small subset (approximately 60) of the total number of built-in objects in PD (These are listed in Appendix A). There are at least another 80 to go to just implement the core functionality available in PD, ignoring various very useful externals like Zexy (which provides boolean-result functions for audio streams) or Cyclone (a Max/MSP compatability layer). I have, however, implemented enough objects that many control patches will not require further work, and adding additional built-ins onto the compiler should be easy and can be done on an as-needed basis.

Because the N800 has a floating point unit, I did not look into implementing automatic

fixed point casting of the various numbers. As such, my compiler is not yet very useful on small hardware such as cell phones or similar devices, but the small size of the N800 and functional equivalents compares favorably with current portable audio players.

Currently, several PD functions (such as `cos~` and `osc~`) are implemented using the FPU; for machines that lack a FPU, a lookup-table based solution will be necessary.

6.1.2 ZigBee

ZigBee is a complex standard that, although it supports low-latency networks, has some confusing features. Zstack provides an interface to adjust the protocol, but initial tweaking of the system created serious failures with no positive results.

ZigBee itself is very usable, but the extra physical parts tremendously reduce the utility of such a device – recall that the user has to wear not only a computational device but also separate wireless sensors requiring an accelerometer, microcontroller, and antenna. Miniaturizing this system as much as possible as described below would help somewhat, but the system still has multiple parts. However, in comparison to other exercise sensors such as heart-rate bands, the accelerometer is quite unintrusive. Furthermore, Bluetooth peripherals seem to be quite a success, so this concern may be wholly unwarranted.

6.1.3 Analog circuitry

Due to a combination of lack of time and desiring the elegance of a tiny solution, I just directly connected the accelerometers and gyros to the ADC inputs on the CC2431. This is fine for the gyroscope, since its full-scale reading is relatively slow (500°/sec). However, the accelerometers are sensitive to a much larger range, and need amplification and biasing to increase the signal-to-noise ratio and enable the use of the full range of the ADC. Although I used both accelerometers and gyros in my demonstration, simple wireless sensors using only an accelerometer chip may be adequate for an interactive exercise application. Note that, with increased integration and mature production, costs of both accelerometers and gyros are steadily dropping.

6.1.4 Demonstration patch

The phase-locked loop rhythm tracker (figure 5-4) is not implemented particularly well – it doesn't match phase very well and it is far too sensitive to missed beats or other momentary

errors. The PID (proportional integral derivative) controller also needs significant tweaking to get good response. To make it worse, the PID controller is sufficiently hard to characterize that it is difficult to determine the correct parameters to use. Lastly, the rather naïve approach I use for synchronizing to footfalls tends to miss steps, which the phase-locked loop does not take kindly to. Certainly, a bit more work here can generate a much more engaging demonstration, although the patch of figure 5-4 sufficed to show the system in basic operation.

6.2 End Result

The compiler is distinctly to the point where it produces useful results, but it needs a lot more effort to bring it to the point where it could be used in a commercial setting. It has a number of rough edges (mostly due to lack of time), but can now be used on an experimental basis.

It is now the case that we could make the sensor peripheral small enough to be a feasible device: the microcontroller is a mere $7mm^2$ and the accelerometer is $5mm^2$. The largest part is now the battery, and a CR2032 lithium coin cell, containing 700mWHrs, could run the system for multiple hours. Furthermore, the single-item costs of this radio and accelerometer are $10 each, and the accelerometer already has lower cost versions. It seems likely that the microcontroller's bulk cost will drop to a even better rate within the next few years. This will result in a system that is both tiny and affordable for the end user.

Appendix A

PuDaC source

The following PureData objects have been implemented fully:

`* + - / < == > abs b bang bng clip dac~ f float i int line~ loadbang max min osc~ pack print r realtime receive route s sel select send spigot sqrt table tabread~ tgl timer t trigger unpack until` and [FloatAtom⟩

The following PureData objects have been implemented partially:

`netreceive soundfiler` and [MessageBox(

The following PureData objects have been implemented in a manner that is inaccurate, but will probably work anyway:

`timer cputime metro delay tabread4~`

The following are not PureData objects but are there for reference:

`\dspstub~`

Electronic copies of the source may be obtained by emailing the author at `rnjacobs@alum.mit.edu`

A.1 pudac.pl

```
#!/usr/bin/perl -- -*- cperl -*-

use Carp;
use Data::Dumper;

use strict;
use warnings;

# interpolation of $1 &c should be done by perl in the parser
# ... well, in objects. in messages it's messy. i'll have to figure that out.
# also, need to figure out exactly how I'm doing send/receive.
# and metro.

our %objectlist;
our ($globalinit, $loadbangs, $periteration, $dsptrace)=("","","","");
our $objectcount = 0;

our %objecttranslations;
our %xtypes;
```

```
our %parser;
do "pudac.objects.pl" or croak "Object translations broken";
do "pudac.xtypes.pl" or croak "Xtypes broken";
do "pudac.parser.pl" or croak "Parser broken";

# Lexer time!

# (i.e. build data structures in memory corresponding to input .pd file)
my $FH;
open $FH, '<', $ARGV[0];

{
  while (my $line = <$FH>) {
    # need to fix this: it will not work with embedded escaped semicolons (\;)
    $line .= <$FH>
      until (not $line or $line =~ m.[^\\];.); # not escaped semicolons
    $line =~ s/;$//;

    # escape all " %
    # (don't need to escape \, they can't exist)

    $line =~ s/"/\"/g; # escape " (C)
    $line =~ s/%/\%/g; # escape % (printf)
    $line =~ s'\\\$'$'g; # unescape $
    $line =~ s/\\;/;/g; # unescape ;
    $line =~ s/\\,/,/g; # unescape ,

    my @parse = split /\s/,$line;
    # Should escape any quotation marks here!

    if ($#parse > 0) {
      #print STDERR join(":", @parse)."\n";
      croak "Don't understand data type $parse[0]\n"
unless (exists $parser{$parse[0]});

      $parser{$parse[0]}->(@parse[1..$#parse]);
    }
  }
}

#print STDERR join "=", keys %objecttranslations;

# Make header:
# (should perhaps roll pudac.h contents in?)
print qq[
#include <search.h>
#define _GNU_SOURCE
/* for getopt_long_only */
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <alsa/asoundlib.h>
#include "pudac.h"

unsigned int sample_rate;
unsigned int block_size;

];

# Make prototypes:
print "void object$_ (int inletnumber, const struct PDdata * message);\n"
 for (sort {$a <=> $b } keys %objectlist);

# sort connects for each object (very important!)
# puredata sends things out of objects from rightmost outlet first
#  (which i store as "highest")
for (keys %objectlist) {
  if (exists $objectlist{$_}->{'AOCS'}) {
    my @t = split /\n/,$objectlist{$_}->{'AOCS'};
    @t = map { $_->[0] }
      sort { $b->[1] cmp $a->[1] }
map { [$_, /,(.+)$/] }
  @t;
    $objectlist{$_}->{'AOCS'} = join "\n",@t;
  }
}

# Generate code for objects:
for (sort {$b <=> $a } keys %objectlist) { # because send/receive go by *reverse object order*
  print "/* @{$objectlist{$_}->{'ARGS'}} */\n";
  if (exists $objectlist{$_}->{'CODE'}) {
    $objectlist{$_}->{'CODE'}
      ($_,
       %{$objectlist{$_}});
  } else {
    croak "Didn't have a code routine for $_: Only had ".join(", ",keys %{$objectlist{$_}});
  }
}

print qq[
int main ( void ) {
  struct PDdata DSPtracer={PDC_DSP_REQUEST,0,NULL,0,0,NULL,NULL};
  struct timeval now;
  int16_t * pcm_nonint;
  snd_pcm_t * handle;
  snd_pcm_sframes_t frames;
  void *indir[2];
  int err;
  int i;

  sample_rate = 48000;
  block_size = 64;
  DSPtracer.dsp_stream = calloc(block_size * 2,sizeof(float));
  pcm_nonint = calloc(block_size * 2,sizeof(int16_t));
  if (DSPtracer.dsp_stream == NULL || pcm_nonint == NULL) {
      error("Unable to allocate memory for root dac structure");
      exit(1);
  }
```

```
  indir[0] = pcm_nonint;
  indir[1] = &pcm_nonint[block_size];

  hcreate(1000); // Maximum number of listening names.
  // It runs out of space if we use it up.

  // open libasound stuff
  // DSP:
  err=snd_pcm_open(&handle, "default", SND_PCM_STREAM_PLAYBACK, 0);
  if (err < 0) {
    error("playback open failure: \%s",snd_strerror(err));
    exit(1);
  }
  err=snd_pcm_set_params(handle, SND_PCM_FORMAT_S16, SND_PCM_ACCESS_RW_NONINTERLEAVED,
    /*channels:*/2, sample_rate, /*allow resampling:*/0, /*latency:*/100000);
  if (err < 0) {
    error("playback params failure: \%s ",snd_strerror(err));
    exit(1);
  }

  // MIDI:
  //(maybe we won't do this)

  $globalinit

  $loadbangs

  while ( 1 ) {
    gettimeofday(&now,NULL); /* we should tie this to the sample clock */
    $periteration
    memset(DSPtracer.dsp_stream,0,block_size*sizeof(float)*2);
    $dsptrace
    for (i=0;i<block_size*2;i++) {
      if (DSPtracer.dsp_stream[i]>=1) pcm_nonint[i] = 32767;
      else if (DSPtracer.dsp_stream[i]<-1) pcm_nonint[i] = -32768;
      else pcm_nonint[i] = DSPtracer.dsp_stream[i]*32768;
    }
    frames = snd_pcm_writen(handle,indir,block_size);
    if (frames < 0)
      frames = snd_pcm_recover(handle,frames,0);
    else if (frames < block_size)
      error("short write? (\%d < \%d",frames,block_size);
    if (frames < 0) {
      error("playback data failure: \%s",snd_strerror(err));
      exit(1);
    }
    DSPtracer.sample_count += frames;
  }
}

];
```

A.2 pudac.parser.pl

```
# -*- cperl -*-

%parser =
  ('#A' => sub {
     local *__ANON__ = 'PuDaC::Parser::A';
     $previousarray->{'contents'} = \@_;
     # "#A" contains initial data for the preceeding "#X array" object
     # for this, i need to keep a pointer to it lying around

     # The presence of a #A seems to ignore whether the previous object
     # nominally saved its contents.
   },
   '#N' => sub {
     local *__ANON__ = 'PuDaC::Parser::N';
     # currently there are only "#N canvas"es
     croak "Do not understand new $_[0]"
       unless (lc $_[0] eq "canvas");
     # this creates a canvas, used for patches, subpatches and graphs
     #  I don't want to handle it yet.
     # As a workaround for not having graphs, use [table arrayname]
     # Because they can be nested, each time we find one we need to
     #  ignore another depth until we find another "#X restore"

     # three kinds of #N canvas:
     #   #N canvas [X] [Y] [W] [H] [PtSz]   = First line only, patcher
     #   #N canvas [X] [Y] [W] [H] [name] [OpenOnLoad] = subpatcher or graph

     #  name is something like "(subpatch)" in the case of unnamed subpatchers
     #   or graphs
     #  name does not need to be unique

     # possible working suggestion: each subpatcher's objects start counting at
     #  1k higher than the last one; PD deals sufficiently poorly with 1k objects
     #  that I doubt anyone will have a patch with 1k objects in it.

     # If I'm wrong, I can just increase it to 10k at not worry about it.

     # The real problem comes from keeping track of multiple subpatches --
     #  i guess we need a stack

     $number_of_canvases++;
     if ($number_of_canvases > 1) {
       croak "We don't yet deal with subpatches, sorry.\n".
 "If you need arrays, try using the [table] object";
     }
   },
   '#X' => sub {
     local *__ANON__ = 'PuDaC::Parser::X';
     my ($name, @rem) = @_;
     if (exists $xtypes{$name}) {
       $xtypes{$name}->(@rem)
     } else {
       croak "Don't undertstand #X $name";
     }
     # All X types increase the object count, even comments.
     #    EXCEPT NOT CONNECTs
```

```
    #    (coords probably don't either)
    #    but these both tend to be the last items in the list
    #    and so don't matter for numbering
    # So the patcher containing
    #   [X] 1
    #  4 |    comment 2
    #   [X] 3
    # has the connect going from object 0 to object 2
    }
  );

1;
```

A.3 pudac.xtypes.pl

```
# -*- cperl -*-

%xtypes =
  ('array' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::array';
    # #X array [name] [sz] float [Properties];
    # #X obj [X] [Y] table [name] [sz];
    my ($name,$size,$check,$properties,@verify)=@_;
    carp "Array (object #$objectcount) is not float? (claims $check)"
      unless (lc $check eq "float");
    carp "Array (object #$objectcount) is unknown visual representation"
      unless (0 <= $properties and $properties <= 5);
    carp "Array (object #$objectcount) has extra arguments?"
      if ($#verify > 0);

    # properties:
    #  0- do not save, polyline
    #  1- save contents, polyline
    #  2- do not save, points
    #  3- save contents, points
    #  4- do not save, bezier
    #  5- save contents, bezier

    # Note that array is a kind of sugar for a "#X obj table"
    # they just always plot on parent (in fact, _all_ of the parent)
    # which we don't care about, because we're not doing graphics
    $objectlist{$objectcount}->{'CODE'} =
      $objecttranslations{'table'}->{'CODE'};
    $objectlist{$objectcount}->{'ARGS'} = [$name,$size];
    $previousarray = $objectlist{$objectcount};
    $objectcount++;
    # this is probably incomplete, i need to figure out how tabread/&c work
    },

  'connect' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::connect';
    # #X connect object-out outlet# object-in inlet#

    # we build the $attachedobjectcalls strings here
    my ($objfrom, $outletnum, $objto, $inletnum,@verify) = @_;
    carp "Connect has extra arguments?"
      if ($#verify > 0);

    # Note: Connecting to non-existant outlets causes C compile time errors,
    #  while connecting to non-existant inlets causes runtime errors
    # (for comparison, invalid arguments should cause PUDAC compile tile errors)

    $objectlist{$objfrom}->{'AOCS'} .=
      "object${objto}($inletnum,&outvalue".
(($outletnum!=0)?($outletnum+1):"").
  ");\n";

    # actually, other direction for DSP objects.

    # (This is the reverse-linked for use by DSP objects:)

    $objectlist{$objto}->{'DSPCS'} .=
      "object${objfrom}(".-($outletnum+1).",&DSPtrace".
($inletnum+1).");\n";
    },

  'coords' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::coords';
    # Coords seems to be purely visual?
    # I think we can ignore it.
    # It also doesn't count towards the object count.
    # (It also is written by PD as the last thing,
    #  so it oughtn't have mattered anyway)
  },

  'msg' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::msg';
    my ($x,$y,@rest) = @_;
    # How do we do this? We probably make a special kind of object
    # that parses $0 and friends.

    # Messages by default send lists.
    # So    [ foo (    sends "foo"->NULL

    # Unlike many other objects, they keep no state
    # So    [ foo $1 (    sends "foo"->0->NULL
    #  unless it was sent something better than a bang

    # Msgs show one of the problems with the representation I've
    #  chosen for lists:
    # [ bob ( -- [ foo $1 (    errors and sends "foo"->0->NULL
    #  because it tries to interpret bob (an anything) as a float (why?)
    # However,
    # [ symbol bob ( -- [ foo $1 (   sends "foo"->"bob"->NULL
    #  because it's told to interpret bob as a symbol
    # Similarly,
    # [ list bob ( -- [ foo $1 (   sends "foo"->"bob"->NULL
    #  because ... the list ... you know, i have no idea.

    # Anyway, I'm just going to gloss over that, since I hope my compiler
```

```
    #  won't get me in trouble by being too tolerant.

    # The object name is named \MSG because Tk can't deal with backslashes, and
    #  and so PureData forbids them.
    $objectlist{$objectcount}->{'CODE'} =
      $objecttranslations{'\msg'}->{'CODE'};
    $objectlist{$objectcount}->{'ARGS'} = \@rest;
    $objectcount++;
  },

  'restore' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::restore';
    my ($x, $y, $type, $name, @verify) = @_;

    carp "Restore has extra arguments?"
      if ($#verify > 0);

    croak "Don't yet deal with subpatches, sorry.\n";
  },
  # Atoms are interesting in that they're like a couple of objects together
  # (ignoring their graphic representation) :
  # * send, receive, [symbol/float]
  'floatatom' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::floatatom';
    my ($x, $y, $width, $min, $max, $labelpos, $label, $rcv, $snd, @verify) = @_;

    carp "Floatatom has extra arguments?"
      if ($#verify > 0);

    # of the arguments, we don't care about any but rcv and snd,
    # because PureData doesn't clip inputs to floatatoms
    # this basically acts like a [float] object.

    push @{$receivers{$rcv}},$objectcount;

    $objectlist{$objectcount}->{'CODE'} =
      $objecttranslations{'\floatatom'}->{'CODE'};
    $objectlist{$objectcount}->{'ARGS'} = [$rcv,$snd];
    # we don't care about the other arguments because:
    # $x, $y, $width: only affect graphical appearance (or, when width=1, user input)
    # $min, $max: only affect user input (a floatatom can have an arbitrary # sent to it)
    # $labelpos, $label: only graphical

    # Sends tied to are processed after propagation
    #  there is a reason [send] objects don't have outlets

    $objectcount++;
  },
  # As far as I know, floatatom and symbolatom only accept the
  'symbolatom' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::symbolatom';
    my ($x, $y, $width, $min, $max, $labelpos, $label, $rcv, $snd, @verify) = @_;

    carp "Symbolatom has extra arguments?"
      if ($#verify > 0);

    $objectcount++;

    croak "Symbol boxes unimplemented, sorry.\n";
  },
  # The obj's that have graphical representations have many of the same
  # properties as the atoms
  'obj' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::obj';
    my ($x, $y, $name, @args) = @_;

    croak "Object $name unimplemented.\n"
      unless (exists $objecttranslations{$name});

    $name = $objecttranslations{$name}->{'SYNONYM'}
      while (exists $objecttranslations{$name}->{'SYNONYM'});

    $objectlist{$objectcount}->{'CODE'} =
      $objecttranslations{$name}->{'CODE'};
    $objectlist{$objectcount}->{'ARGS'} = \@args;
    $objectcount++;
  },
  'text' => sub {
    local *__ANON__ = 'PuDaC::Xtypes::text';
    # These are comments. They get ignored.
    $objectcount++;
    # except they count as objects.
  },
 );

1;
```

A.4 pudac.objects.pl

```
# -*- cperl -*-
# This file contains the %objectstranslations hash of object-to-C translations

## Notes:
## event objects-
##  data flows down (by calls)
##  each call contains payload
##  certain objects source events (metro, delay, midiin, netreceive, &c)
## DSP objects-
##  data flows up (by return)
##  objects cache with the sample # include
##  certain objects sink data (dac~, tabsend~, tabwrite~, print~, &c)

## All objects: inletnumber >=0 -- message intlets (number N = (N+1)th inlet)
##  inletnumber < 0 -- dsp outlet (number -N = Nth outlet)

my $MSGwarning = 0;
my $ROUTEwarning = 0;


%objecttranslations =
```

```
    ('b' => { 'SYNONYM' => 'bang' },
     'f' => { 'SYNONYM' => 'float' },
     'i' => { 'SYNONYM' => 'int' },
#    'l' => { 'SYNONYM' => 'lister' }, # from zexy, not implementing, but here for completeness.
     'r' => { 'SYNONYM' => 'receive' },
     'r~' =>{ 'SYNONYM' => 'receive~' },
     's' => { 'SYNONYM' => 'send' },
     's~' =>{ 'SYNONYM' => 'send~' },
     't' => { 'SYNONYM' => 'trigger' },
     'v' => { 'SYNONYM' => 'value' },
     'sel'=>{ 'SYNONYM' => 'select' },
     'fswap'=>{'SYNONYM'=> 'swap' },
     'hslider'=>{'SYNONYM'=>'hsl' },
     'radiobut'=>{'SYNONYM'=>'rdb' },
     'my_canvas'=>{'SYNONYM'=>'cnv' },
     'my_numbox'=>{'SYNONYM'=>'nbx' },
     'radiobutton'=>{'SYNONYM'=>'rdb' },

     # these are lies:
     'tabread4~' => { 'SYNONYM' => 'tabread~' },
     'timer' => { 'SYNONYM' => 'realtime' },
     'cputime' => { 'SYNONYM' => 'realtime' },

     '+' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::+';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is an ADD object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("+ takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft + lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft + lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("+ doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("+ takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("+ only has two inlets");
            return;
    }
    return;
}
];
}
  },
     'print' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::print';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 my $initializer = "print";
 if (length $arguments[0] > 0) {
     $initializer = $arguments[0];
 }

 print qq[
/* object $number is a PRINT object */
void object$number (int inletnumber, const struct PDdata * message) {
    const char printname[] = {"$initializer"};
    const struct PDdata * args;

    if (inletnumber != 0) {
        error("print only has one inlet");
        return;
    }

    args = message;
    printf("\%s: ",printname);
    while (args != NULL) {
        switch (args->type) {
        case PDC_TYPE_FLOAT:
            printf("\%f",args->val_float);
```

```
            break;
        case PDC_TYPE_BANG:
            printf("bang");
            break;
        case PDC_TYPE_SYMBOL:
            printf("\%s",args->val_symbol);
            /* Technically, we're supposed to clip at 78 characters. I think I can not care. */
            break;
        default:
            error("unknown type \%d", args->type);
            return;
        }
        if (args->linked != NULL) {
            printf(" ");
        }
        args = args->linked;
        /* We assign this so that when our linked list runs out,
           the while up there is what terminates */
    }
    printf("\\n");
    return;
}
];
 # print never has attached objects
     }
      },
     'float' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::float';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 my $initializer = 0;
 $initializer = $arguments[0]
   if (exists $arguments[0]);

 print qq[
/* object $number is a FLOAT object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float internal=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                internal = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("float for some reason cares that 2nd elt in list is nonfloat");
                        /* however this doesn't fail to propagate */
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = internal;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = internal;
                break;
            case PDC_TYPE_SYMBOL:
                error("float doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("float takes only takes float in right inlet");
        return;
            }
    internal = message->val_float;
    break;
default: /* not an inlet */
            return;
    }
    return;
}
];
}
  },
     'int'  => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::int';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 my $initializer = 0;
 $initializer = $arguments[0]
   if (exists $arguments[0]);

 print qq[
/* object $number is an INT object */
void object$number (int inletnumber, const struct PDdata * message) {
    static int internal=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                internal = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("int for some reason cares that 2nd elt in list is nonfloat");
                        /* however this doesn't fail to propagate */
```

```
                }
            }
            outvalue.type = PDC_TYPE_FLOAT;
            outvalue.val_float = internal;
            break;
        case PDC_TYPE_BANG:
            outvalue.type = PDC_TYPE_FLOAT;
            outvalue.val_float = internal;
            break;
        case PDC_TYPE_SYMBOL:
            error("float doesn't understand symbols");
            return;
        default:
            error("unknown type \%d",message->type);
            return;
        }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
        if (message->type != PDC_TYPE_FLOAT) {
      error("float takes only takes float in right inlet");
      return;
        }
    internal = message->val_float;
    break;
default: /* not an inlet */
        return;
    }
    return;
}
];
}
  },
     'bang' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::bang';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 print qq[
/* object $number is a BANG object */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the inlet */
        outvalue.type = PDC_TYPE_BANG;
        $attachedobjectcalls
        break;
default: /* not an inlet */
    error("bang only has one inlet");
        return;
    }
    return;
}
];
}
  },
     # metro loadbang, and so on are interesting, because they don't just involve
     # walking the trees.
     # loadbang is fairly straightforward, i think?
     # metronome, midiin, &c are much more interesting.
     'loadbang' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::loadbang';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 # basically we generate a bang object, but it only gets called by the main loop, once.
 # i believe the order is ambiguous, PD 0.40 seems to do them in creation order.

 print qq[
/* object $number is a LOADBANG object */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue;

    /* loadbang should never be compiled in, except to the initial LB evaluation */

    outvalue.linked = NULL;

    outvalue.type = PDC_TYPE_BANG;
    $attachedobjectcalls
    return;
}
];

 $loadbangs .= qq[
/* now call loadbang obj $number: */
object$number(PDC_CALLBACK,NULL); /* loadbang doesn't care about inlet # */
];
    }
 },
    'metro' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::metro';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};
 # here's where i'd like to have classes: i want both the object, and an accessor to
 # its internal state so that i can find out if i should call it.

 # options:
 # 1) make inlet # -1 special
 # 2) make the internal state not internal, but given its own unique global name

 # i've done some of each here

 my $initializer = 0;
 $initializer = $arguments[0]
   if (exists $arguments[0]);

#  print STDERR "making metro $number w/ period $initializer ms\n";

 print qq[
```

```
/* object $number is a METRO object */

struct timeval object${number}_time;
int object${number}_running;

void object$number (int inletnumber, const struct PDdata * message) {
    static float period_in_milliseconds=$initializer;
    struct PDdata outvalue;
    struct timeval TV_internal;

    outvalue.linked = NULL;
    outvalue.type = PDC_TYPE_BANG;

    switch (inletnumber) {
        case PDC_CALLBACK: /* a timeout */
            $attachedobjectcalls
            object${number}_time.tv_usec += period_in_milliseconds*1000;
            while (object${number}_time.tv_usec > 1000000 ) {
                object${number}_time.tv_sec++;
                object${number}_time.tv_usec-=1000000;
            }
            break;
case 0: /* the toggly inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT: /* 0 means stop, non-zero means run */
                object${number}_running = (message->val_float != 0);
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("metro takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        /* schedule the next one */
                        period_in_milliseconds = message->linked->val_float;
                    }
                }
                break;
            case PDC_TYPE_BANG: /* bang means run */
                object${number}_running = 1;
                break;
            case PDC_TYPE_SYMBOL: /* "stop\s?" means stop, i'm being a little more tolerant */
                if (strncasecmp("stop",message->val_symbol,4)==0) {
                    object${number}_running = 0;
                }
                error("metro doesn't understand that symbol");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    if (object${number}_running) {
                $attachedobjectcalls
                gettimeofday(&TV_internal,NULL);
                object${number}_time.tv_usec = TV_internal.tv_usec
                                             + period_in_milliseconds*1000;
                object${number}_time.tv_sec = TV_internal.tv_sec;
                while (object${number}_time.tv_usec > 1000000) {
                    object${number}_time.tv_sec++;
                    object${number}_time.tv_usec-=1000000;
                }
            }
    break;
case 1: /* the timer inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("metro takes only takes float in right inlet");
        return;
            } else {
        period_in_milliseconds = message->val_float;
                object${number}_running = 0;
            }
    break;
default: /* not an inlet */
    error("metro only has two inlets");
            return;
    }
    return;
}
/* }} cperl has trouble */
];

 $globalinit .= qq[
  /* now initializing metro $number: */
  object${number}_running = 0;
];
 $periteration .= qq[
/* Now checking timeout for obj $number: */
if (object${number}_running &&
    (now.tv_sec > object${number}_time.tv_sec ||
     (now.tv_sec == object${number}_time.tv_sec &&
      now.tv_usec > object${number}_time.tv_usec))) {
       object${number}(PDC_CALLBACK,NULL);
}
];
      }
},
     'dac~' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::dac~';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $reverseDSPcalls = $myself{DSPCS};

 croak "Non-stereo audio generation unimplemented!"
   if ($#arguments > 0);

 print qq[
/* object $number is a DAC~ object */
float *object${number}_upstr1, *object${number}_upstr2,
  *object${number}_cache1, *object${number}_cache2;

void object$number (int inletnumber, const struct PDdata * message) {
    static struct PDdata DSPtrace1={PDC_DSP_REQUEST,0,NULL,-1,0,NULL,NULL},
                         DSPtrace2={PDC_DSP_REQUEST,0,NULL,-1,0,NULL,NULL};
    static float * cache1 = NULL, * cache2 = NULL;
    float * writer;
    int i;
```

```
    if (NULL == DSPtrace1.dsp_stream) DSPtrace1.dsp_stream = object${number}_upstr1;
    if (NULL == DSPtrace2.dsp_stream) DSPtrace2.dsp_stream = object${number}_upstr2;
    if (NULL == cache1) cache1 = object${number}_cache1;
    if (NULL == cache2) cache2 = object${number}_cache2;

    writer = message->dsp_stream; /* break const-ness of message */

    if (inletnumber == PDC_DSP_DAC &&
        message->type == PDC_DSP_REQUEST) {
        if (message->sample_count != DSPtrace1.sample_count ||
            message->sample_count != DSPtrace2.sample_count) {
            memset(DSPtrace1.dsp_stream,0,block_size * sizeof(float));
            memset(DSPtrace2.dsp_stream,0,block_size * sizeof(float));
            DSPtrace1.sample_count = DSPtrace2.sample_count = message->sample_count;
            $reverseDSPcalls
            memcpy(cache1,DSPtrace1.dsp_stream,sizeof(float)*block_size);
            memcpy(cache2,DSPtrace2.dsp_stream,sizeof(float)*block_size);
        }
        for (i=0;i<block_size;i++) {
            writer[i]+=cache1[i];
            writer[i+block_size]+=cache2[i]; /* This is ONLY OK in dac~s */
        }
    } else
        error("dac~ got funny input? inlet:\%d type:\%d",inletnumber,message->type);
}
];#]

 $globalinit .= qq[
  /* allocate "private" memory for dac~ $number: (dac~) */
  object${number}_upstr1 = malloc(block_size*sizeof(float));
  object${number}_upstr2 = malloc(block_size*sizeof(float));
  object${number}_cache1 = malloc(block_size*sizeof(float));
  object${number}_cache2 = malloc(block_size*sizeof(float));

  if (object${number}_upstr1 == NULL || object${number}_upstr2 == NULL ||
      object${number}_cache1 == NULL || object${number}_cache2 == NULL) {
      error("Unable to allocate memory for obj $number (dac~)!");
      exit(1);
  }
];

 $dsptrace .= qq[
    /* Checking on obj $number: (dac~) */
    object$number(PDC_DSP_DAC,&DSPtracer);
];
       }
       },
     '\dspstub~' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::\dspstub~';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $reverseDSPcalls = $myself{DSPCS};

 print qq[
/* object $number is a dsp stub, should never be rendered */
];

 $globalinit .= qq[
  /* allocate "private" memory for obj $number */
];
 $dsptrace .= qq[
  /* checking on dsp terminus obj $number */
];
       }
       },
     'osc~' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::osc~';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $reverseDSPcalls = $myself{DSPCS};

 print qq[
/* object $number is an OSC~ object */
float *object${number}_upstr, *object${number}_cache;

void object$number (int inletnumber, const struct PDdata * message) {
    static struct PDdata DSPtrace1={PDC_DSP_REQUEST,0,NULL,-1,0,NULL,NULL};
    static float * cache = NULL;
    float * writer;
    static float cosph, freq;
    char * write_is_dsp;
    int i;

    if (NULL == DSPtrace1.dsp_stream) DSPtrace1.dsp_stream = object${number}_upstr;
    if (NULL == cache) cache = object${number}_cache;

    writer = message->dsp_stream; /* break const-ness of message */

    switch (inletnumber) {
        case -1: /* outlet 1 */
            if (message->type != PDC_DSP_REQUEST)
                return;
            write_is_dsp = &message->is_dsp;
            *write_is_dsp=1;
            /* find out if we have a dsp rate input */
            if (message->sample_count != DSPtrace1.sample_count) {
                memset(DSPtrace1.dsp_stream,0,block_size * sizeof(float));
                DSPtrace1.sample_count = message->sample_count;
                $reverseDSPcalls
                if (DSPtrace1.is_dsp) {
                    for (i=0;i<block_size;i++) {
                        cache[i]=cos(cosph);
                        cosph+=DSPtrace1.dsp_stream[i] * M_PI*2.0/sample_rate;
                    }
                } else {
                    for (i=0;i<block_size;i++) {
                        cache[i]=cos(cosph);
                        cosph+=freq * M_PI*2.0/sample_rate;
                    }
                }
                while (cosph>7) cosph -= M_PI*2;
                while (cosph<-7) cosph += M_PI*2;
            }
            for (i=0;i<block_size;i++) {
```

```
                writer[i]+=cache[i];
            }
            break;
        case 0: /* inlet 1 = freq */
            if (message->type == PDC_TYPE_FLOAT)
                freq = message->val_float;
            else
                error("osc~ takes only floats or dsp chains in freq inlet (got \%d)",message->type);

            if (message->linked)
                if (message->linked->type == PDC_TYPE_FLOAT)
                    cosph = 0;
                else
                    error("osc~ takes only float as 2nd elt of list in 1st inlet (got \%d)",message->type);

            break;
        case 1: /* inlet 2 = reset phase */
            /* only if numeric */
            if (message->type == PDC_TYPE_FLOAT)
                cosph = 0;
            else
                error("osc~ takes only floats in reset_phase inlet (got \%d)",message->type);
            break;
        default:
            error("osc~ only has two inlets (tried \%d)",inletnumber);
            return;
    }
}
];#]]]]

 $globalinit .= qq[
  /* allocate "private" memory for osc~ $number */
  object${number}_upstr = malloc(block_size*sizeof(float));
  object${number}_cache = malloc(block_size*sizeof(float));

  if (object${number}_upstr == NULL || object${number}_cache == NULL) {
    error("Unable to allocate memory for obj $number (osc~)!");
    exit(1);
  }
];
       }
       },
     'table' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::table';
 my ($number, %myself, @error) = @_;
 my ($name,$size) = @{$myself{ARGS}};

 # table without any arguments produces an incremententally named
 # table (table1, table2, &c) of length 100

 # table with one argument produces a table with name that argument,
 # of length 100

 $name = "table".(++$number_of_tables) if (length $name < 1);
 $size = 100 if ($size < 1);

 # Technically, tables are *intimately* tied into sends/receives.
 # I'll care about these eventually:
 #  CONST n (set to constant value)
 #  RESIZE n (set array size)
 #  PRINT (print size of table, only useful for debugging)
 #  SINESUM/COSINESUM size N1 N2 N3 .. Nn (Nth harmonic synthesis, also resizes)
 #  NORMALIZE n (sets to maximum volume times n)
 #  READ filename (doesn't resize) (space-separated ascii floats)
 #  WRITE filename (ditto)
 #  N1 N2 N3 .. Nn (set contents)

 # The following don't make sense to implement:
 #  BOUNDS
 #  XTICKS
 #  YTICKS

 # The following is a pain to implement correctly so I'm not.
 #  RENAME
 #  (plus, i can't think of a legitimate use for it.)

 print qq[
/* object $number is a TABLE object */
float *object${number}_table;
int object${number}_size;

void object$number (int inletnumber, const struct PDdata * message) {
  int i;
  float t;
  const struct PDdata * args;
  float ** tablecontents;
  int * tablelen;

  if (inletnumber != PDC_SENT) {
    error("table should have no connections (got event via inlet \%d)",inletnumber);
    return;
  }

  switch (message->type) {
    case PDC_TYPE_FLOAT:
      args = message;
      for (args=message,i=0;
           i<object${number}_size && args != NULL;
           i++,args=message->linked)
        object${number}_table[i]=message->val_float;
      return;
    case PDC_TYPE_SYMBOL:
      if (strcasecmp("const",message->val_symbol)==0) {
        if (message->linked == NULL) t=0;
        else if (message->linked->type != PDC_TYPE_FLOAT) {
          error("table received 'const nonfloat', don't deal");
          return;
        } else t=message->linked->val_float;
        for (i=0;i<object${number}_size;i++)
          object${number}_table[i]=t;
        return;
      } else if (strcasecmp("resize",message->val_symbol)==0) {
        if (message->linked == NULL ||
            message->linked->type != PDC_TYPE_FLOAT) {
          error("table received 'resize nonfloat', don't deal");
```

```
          return;
        }
        i = message->linked->val_float;
        if ( i < 1 ) i = 1;
        object${number}_table = realloc(object${number}_table,i);
        if (object${number}_table) {
          error("TABLE REALLOC FAILED");
          exit(1);
        }
      } else if (strcasecmp("print",message->val_symbol)==0) {
        printf("table $name: \%d elts\\n",object${number}_size);
        return;
      } else if (strcasecmp("\\\\getguts",message->val_symbol)==0) {
        tablecontents = &(message->dsp_stream); /* needed for efficiency in all table routines */
        *tablecontents = object${number}_table;
        tablelen = &(message->sample_count);
        *tablelen = object${number}_size;
        return;
      } else if (strcasecmp("normalize",message->val_symbol)==0) {
      } else if (strcasecmp("read",message->val_symbol)==0) {
      } else if (strcasecmp("write",message->val_symbol)==0) {
      }
  }
}
];

 $globalinit .= qq[
  /* Allocate shared memory for table $number */
  object${number}_size = $size;
  object${number}_table = calloc(object${number}_size,sizeof(float));
  if (object${number}_table == NULL) {
    error("Unable to allocate memory for obj $number (table)!");
    exit(1);
  }
  listens(\& object${number},"$name");
];
       }
},
# to implement soundfiler, i must implement \msg
# to implement soundfiler, i need to deal with runtime names of objects
# so i may as well do send / receive next. sigh.
     'send' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::send';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};

 my $name = $arguments[0];

 # two kinds of sends:
 # no creation arguments
 #  have two inlets
 #  (left inlet is what to send,
 #   right inlet takes ONLY SYMBOLS (or lists) and is the recipient name
 #   (recipient name initialized to "") )
 if (length $name == 0) {
 print qq[
/* object $number is a SEND object with no args */
void object$number (int inletnumber, const struct PDdata * message) {
  char * name = NULL;

  if (message->type == PDC_DSP_REQUEST)
    return;

  switch(inletnumber) {
    case 0:
      sendbyname((name)?name:"",message);
      return;
    case 1:
      if (message->type != PDC_TYPE_SYMBOL) {
        error("obj $number: send can only be given a symbol as a recipient (not \%d)",message->type);
        return;
      }
      name = realloc(name,strlen(message->val_symbol));
      if (name == NULL) {
        error("obj $number: Memory realloc failed for name (probably fatal, trying anyway)");
        return;
      }
      memcpy(name,message->val_symbol,strlen(message->val_symbol));
      return;
  }
}
];
        } else {
 # one creation argument
 #  has one inlet
 #  (is what to send, recipient is set by creation arg)
 print qq[
/* object $number is a SEND object with name */
void object$number (int inletnumber, const struct PDdata * message) {

  if (message->type == PDC_DSP_REQUEST || inletnumber != 0)
    return;

  sendbyname("$name",message);
  return;
}
];
       }
       }
       },
     'receive' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::receive';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 my $name = $arguments[0];

 # receive, on the other hand, receives from the zero-length name if given no arguments
 # and can't be changed.

 # hack to avoid copying
 $attachedobjectcalls =~ s/&outvalue/message/g;

 print qq[
```

```
/* object $number is a RECEIVE object */
void object$number (int inletnumber, const struct PDdata * message) {
  if (inletnumber != PDC_SENT ||
      message->type == PDC_DSP_REQUEST) {
    return;
  }

  $attachedobjectcalls
}
];

 $globalinit .= qq[
  /* receive $number*/
  listens(\& object${number},"$name");
];
       }
  },
     'soundfiler' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::soundfiler';
 my ($number, %myself, @error) = @_;
 my $attachedobjectcalls = $myself{AOCS};

 print qq[
/* object $number is a SOUNDFILER object */
void object$number (int inletnumber, const struct PDdata * message) {
  struct PDdata totable, outvalue;
  int idx, count, bufidx;
  char * filename, * tablename;
  int16_t buf[1024];
  FILE * FH;

  if (inletnumber != 0 || message->type != PDC_TYPE_SYMBOL)
    return;

  if (strcasecmp(message->val_symbol,"read")==0) {
    /* syntax:  read [flags] [filename] [tablename 1] [tablename 2] ... [tablename n]
       i may want to use getopt
       documented flags:
        -skip n (skips first n frames)
        -nframes n (reads not more than n frames)
        -resize (resizes tables to size of file, unless:)
        -maxsize n (do not resize beyond n frames)
        -raw skip chn bps endian (skip skip bytes, read chn channels, each sample
           is bps bytes, endian is one of "l","b","n" to mean "little" "big" or "native" endian

       For my desperately close deadline, I'm going to require that you use "-raw ### 1 2 n"
    */
    if (linkedlength(message)!=8 ||
        strcasecmp(linkedelt(message,1)->val_symbol,"-raw") != 0 ||
                   linkedelt(message,3)->val_float != 1 ||
                   linkedelt(message,4)->val_float != 2 ||
        strcasecmp(linkedelt(message,5)->val_symbol,"n") != 0) {
      error("Crippled version of soundfiler; must be invoked as 'read -raw number 1 2 n file table'");
      return;
    }
    filename = linkedelt(message,6)->val_symbol;
    tablename = linkedelt(message,7)->val_symbol;
    totable.val_symbol = "\\\\getguts";
    totable.type = PDC_TYPE_SYMBOL;
    sendbyname(tablename,&totable);
    FH = fopen(filename,"r");
    if (FH == NULL) {
      error("Failed to open file: \%s for soundfiler read",filename);
      return;
    }
    fseek(FH,linkedelt(message,2)->val_float,SEEK_SET); idx=0;
    while (!feof(FH) && idx<totable.sample_count) {
      count=fread(buf,sizeof(int16_t),1024,FH);
      for (bufidx=0;bufidx<count && idx<totable.sample_count; bufidx++,idx++)
        totable.dsp_stream[idx]=buf[bufidx] / 32768.0;
    }
    fclose(FH);
  } else if (strcasecmp(message->val_symbol,"write")==0) {
    error("soundfiler write unimplemented, sorry.");
  } else error("soundfiler only understands symbols 'read' and 'write', not '\%s'",message->val_symbol);
}
];
       }
     },
 '\msg' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::\msg';
 my ($number, %myself, @error) = @_;
 my $attachedobjectcalls = $myself{AOCS};
 my @args = @{$myself{ARGS}};

 # msg syntax looks like:
 # #X msg x y args

 if (not $MSGwarning) {
   print STDERR "Please be careful; PuDaC's message implementation is more tolerant than PureData's!\n";
   $MSGwarning =1;
 }

 # args may optionally have a symbol as first elt specifying the type of the 2nd elt.
 # (known types: bang float symbol pointer list)
 #  BANG: takes no args. resultant message is just a bang.
 #  FLOAT: takes one arg. resultant message is just the following fp#. If it's not a fpn, error, don't propagate
 #  SYMBOL: takes one arg. resultant message is just the following symbol. If the symbol is a number, it's truncated to null.
 #  POINTER: don't exist in my magical little world.
 #  LIST: takes all args. Casts them all. Used in PD to make things not anythings, but I don't support anythings.

 # Also: Must interpolate $1 .. $n (note: $0 always is 0 in messages!)

 # commas separate a single msg box into multiple messages to be sent (in order)
 # semicolons make the remainder of the message a send
 # source implies dollars are special? how?

 # One: See what the first arg is. If one of the above types, cast the remainder appropriately.
 # Two: For each arg, determine its type (float or symbol)
 # Three: split on all semicolons and commas. Each comma means restart parsing
 #  (each semicolon means "instead of propagating, send to the following listener")
 #  commas immediately after semicolons are ignored -- there's no way to say "now that i've sent
 #   that message to that listener, i'd like to propagate."
 #  there's also no way to specify the null listner
```

```
# XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX
# THIS IMPLEMENTATION IS NOT BUG-COMPATIBLE WITH PUREDATA's.
# I lump float and symbol together. I pretend anythings are lists, always.
# (e.g. [symbol 5< would result in receivers getting "symbol (null-length-symbol)"
#  while [float notafloat< would result in an error.
# We eat the descriptors at the beginning of messages, so we
#  will successfully propagate when PD wouldn't.
# I DO NOT SUPPORT SET messages TO MSG BOXES
# XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX

# So: build original args, resplit on semicolons
my @rcvrs = split /;/, (join " ", @args);
# lose any initial or terminal commas
s/(^\s*,|,\s*$)//g for @rcvrs;

# (rcvrs right now contains something like
#  "this message gets sent via a wire"
#  "this message gets sent to listeners for 'this', as does this one"
#  "while this message gets sent to listeners for 'while', and this one" )

# separate these into messages to be sent, and messages to propagate

# (these are easy)
# separate into multiple messages (formerly comma separated)
@propagaters = split /,/, (shift @rcvrs);
# For each space-separated thing, split it, and put it back
for (@propagaters) {
  s/(^\s*|\s*$)//g;
  my @t = split / /,$_;
  $_ = \@t;
}
# now propagaters should be something like (["this","message"],["that","one"])
# now parse type specifiers
# (start with "symbol" "float" "list" "bang" (ignore others))

for (@propagaters) {
  if (lc $_->[0] eq "symbol" or
      lc $_->[0] eq "float") {
    $_ = [ $_->[1] ];
  } elsif (lc $_->[0] eq "bang") {
    $_ = [ $_->[0] ];
  } elsif (lc $_->[0] eq "list") {
    shift @$_;
    # empty lists are bangers.
    $_ = [ "bang" ]
      unless (scalar @{$_});
  }
}

# generate the dollar-sign parsing code as necessary
# also, eat any initial type specifiers

# rcvrs looks like:
# now it looks like ("one message, two message","red message, blue message")
# Next step: split on commas, add the first word of the preceeding message to each.
for (my $i=scalar @rcvrs - 1;$i>=0;$i--) {
  my @expanded = split /,/, $rcvrs[$i];
  #this should look like ("one message","two message")
  for (@expanded) {
    s/(^\s*|\s*$)//g;
    my @t = split / /,$_;
    $_ = \@t;
  }
  # now this should look like (["one","message"],["two","message"])
  for (my $j=1;$j<scalar @expanded;$j++) {
    unshift @{$expanded[$j]}, $expanded[0][0];
  }
  # this should now look like (["one","message"],["one","two","message"])
  $rcvrs[$i] = \@expanded;
}
# and now (["one","message"],["one","two","message"],["red","message"],["red","blue","message"])

# pop off the outer layer of array indirection.
@rcvrs = map { @{$_} } @rcvrs;

# Also, get rid of any which are just a name: we can't send null messages.
@rcvrs = map { $_ if (scalar @$_ >= 2) } @rcvrs;
# (technically, PD errors if a receiver doesn't exist, but doesn't send
#  a null message if they do)

for (@rcvrs) {
  if (lc $_->[1] eq "symbol" or
      lc $_->[1] eq "float") {
    $_ = [ $_->[0] , $_->[2] ];
  } elsif (lc $_->[0] eq "bang") {
    $_ = [ $_->[0] , $_->[1] ];
  } elsif (lc $_->[0] eq "list") {
    splice @$_,1,1; # get rid of the word "list"
  }
}

# TO BE IMPLEMENTED:
# More stupid dolsym crap:
#  If we're sent a list, and the first elt is not a number
#  this may have been an Anything. then we're off by one... augh.
# XXXX THIS IS A BUG XXXX

# now we have two arrays of arrays (propagaters and rcvrs)
# now we need to start cooking these down into native C structures:
# { TYPE , floatval , symbolval , 0 , 0 , NULL, { repeat } }
my $declstructures="";
my $dollarsigncode="";
for (my $i=0;$i < scalar @propagaters; $i++) {
  for (my $j =scalar @{$propagaters[$i]} - 1; $j >= 0; $j--) {
    $declstructures .= "struct PDdata prop$i".(($j==0)?"":"_$j")." = ";
    if (lc $propagaters[$i][$j] eq "bang") {
      # hey, a bang:
      $declstructures .= "{PDC_TYPE_BANG, 0,NULL,0,0,NULL,".
(($j == scalar @{$propagaters[$i]}-1)?"NULL":("&prop${i}_".($j+1))).");\n";
    } elsif ($propagaters[$i][$j] =~ /^-?(\d*\.)?\d+([eE][-+]?\d+)?$/) {
      # looks like a float:
      $declstructures .= "{PDC_TYPE_FLOAT, $propagaters[$i][$j], NULL,0,0,NULL,".
(($j == scalar @{$propagaters[$i]}-1)?"NULL":("&prop${i}_".($j+1))).");\n";
    } else {
      if ($propagaters[$i][$j] =~ /^\$(\d+)$/) {
```

```
 # dolsym - the easy kind: (copy the referrer)
 $dollarsigncode .= qq[
if (linkedlength(message) < $1) {
  error("Too few args for $\%d (had \%d)",$1,linkedlength(message));
} else {
  t = linkedelt(&prop$i,].($j  ).qq[);
  u = linkedelt(message,].($1-1).qq[);
  t->type = u->type;
  if (t->type == PDC_TYPE_FLOAT) t->val_float = u->val_float;
  else if (t->type == PDC_TYPE_SYMBOL) t->val_symbol = u->val_symbol;
}
];
       } elsif ($propagaters[$i][$j] =~ /\$(\d+)/) {
 # pain in the neck dolsym: requires interpolation
 # we can have a$1b
 # which, if sent "13", should produce a13b
 # and if sent "symbol bsor" would produce absorb
 # ... this requires dynamic allocation of memory, EW.
 croak "Don't yet deal with interpolating messages (specifically".
   " $propagaters[$i][$j])\n[".(join " ", @args)."<";
       }
       # must be a symbol, do symbol (dolsyms are a subset)
       $declstructures .= "{PDC_TYPE_SYMBOL, 0, \"$propagaters[$i][$j]\",0,0,NULL,".
 (($j == scalar @{$propagaters[$i]}-1)?"NULL":("&prop${i}_".($j+1))."};\n";
     }
   }
 }

 # Now do the same for sent messages
 for (my $i=0;$i < scalar @rcvrs; $i++) {
   for (my $j= scalar @{$rcvrs[$i]}-1; $j >= 1; $j--) {
     $declstructures .= "struct PDdata send$i".(($j==1)?"":"_$j")." = ";
     if (lc $rcvrs[$i][$j] eq "bang") {
       # hey, a bang:
       $declstructures .= "{PDC_TYPE_BANG, 0,NULL,0,0,NULL,".
 (($j == scalar @{$propagaters[$i]}-1)?"NULL":("&send${i}_".($j+1)))."};\n";
     } elsif ($rcvrs[$i][$j] =~ /^-?(\d*\.)?\d+([eE][-+]?\d+)?$/) {
       # looks like a float:
       $declstructures .= "{PDC_TYPE_FLOAT, $rcvrs[$i][$j], NULL,0,0,NULL,".
 (($j == scalar @{$propagaters[$i]}-1)?"NULL":("&send${i}_".($j+1)))."};\n";
     } else {
       if ($rcvrs[$i][$j] =~ /^\$(\d+)$/) {
 # dolsym - the easy kind: (copy the referrer)

 # yes, the $j-1 is supposed to be different here.
 $dollarsigncode .= qq[
if (linkedlength(message) < $1) {
  error("Too few args for $\%d (had \%d)",$1,linkedlength(message));
} else {
  t = linkedelt(&send$i,].($j-1).qq[);
  u = linkedelt(message,].($1-1).qq[);
  t->type = u->type;
  if (t->type == PDC_TYPE_FLOAT) t->val_float = u->val_float;
  else if (t->type == PDC_TYPE_SYMBOL) t->val_symbol = u->val_symbol;
}
];
       } elsif ($rcvrs[$i][$j] =~ /\$(\d+)/) {
 # pain in the neck dolsym: requires interpolation
 # we can have a$1b
 # which, if sent "13", should produce a13b
 # and if sent "symbol bsor" would produce absorb
 # ... this requires dynamic allocation of memory, EW.
 croak "Don't yet deal with interpolating messages (specifically $rcrs[$i][$j])\n[".
   (join " ", @args)."<";
       }
       # must be a symbol:
       $declstructures .= "{PDC_TYPE_SYMBOL, 0, \"$rcvrs[$i][$j]\",0,0,NULL,".
 (($j == scalar @{$propagaters[$i]}-1)?"NULL":("&send${i}_".($j+1)))."};\n";
     }
   }
 }

 print qq[
/* object $number is a MESSAGE */
void object$number (int inletnumber, const struct PDdata * message) {
  struct PDdata *t, *u;
$declstructures

  /* messages only have one inlet, and are never in DSP chains */
  if (inletnumber != 0 || message->type == PDC_DSP_REQUEST) return;

$dollarsigncode

];
 # now fix calldowns:
 for (my $i=0; $i < scalar @propagaters; $i++) {
   my $t = $attachedobjectcalls;
   $t  =~ s/outvalue/prop$i/g;
   print $t;
 }
 # now fix sent messages:
 for (my $i=0; $i < scalar @rcvrs; $i++) {
   # this is not quite right, as we can interpolate receiver names. Fix it later.
   print "sendbyname(\"$rcvrs[$i][0]\",&send$i);\n";
 }

 # close the subroutine:

 print "\n}\n";

 # i guess that's it for \msg.
       }
    },
     '-' => { 'CODE' => sub {
         local *__ANON__ = 'PuDaC::Objects::-';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a MINUS object */
```

```
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("- takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft - lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft - lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("- doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("- takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("- only has two inlets");
            return;
    }
    return;
}
];
}
  },
      '/' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::/';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a DIVIDE object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("/ takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft / lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft / lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("/ doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("/ takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("/ only has two inlets");
            return;
    }
    return;
}
];
}
```

```
  },
    '*' => { 'CODE' => sub {
       local *__ANON__ = 'PuDaC::Objects::*';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a MULTIPLY object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("* takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft * lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft * lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("* doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("* takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("* only has two inlets");
            return;
    }
    return;
}
];
}
  },
    '>' => { 'CODE' => sub {
       local *__ANON__ = 'PuDaC::Objects::>';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a GREATER-THAN object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("> takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft > lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft > lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("> doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
```

```
        error("> takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("> only has two inlets");
            return;
    }
    return;
}
];
}
  },
    '<' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::<';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a LESS-THAN object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("< takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft < lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft < lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("< doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("< takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("< only has two inlets");
            return;
    }
    return;
}
];
}
  },
    'realtime' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::realtime';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# unlike PD, our timer measures real time.

print qq[
/* object $number is a REALTIME object */
struct timeval object${number}_time;

void object$number (int inletnumber, const struct PDdata * message) {
    struct timeval t;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the COLD inlet (resets count) */
            if (message->type != PDC_TYPE_BANG) {
                error("realtime only understands bangs (got \%d)",message->type);
                return;
            }
            gettimeofday(&object${number}_time,NULL);
            return;
case 1: /* the HOT inlet (sends current time delta) */
            if (message->type != PDC_TYPE_BANG) {
        error("timer only understands bangs");
        return;
            }
            gettimeofday(&t,NULL);
            outvalue.type = PDC_TYPE_FLOAT;
            outvalue.val_float = (t.tv_sec - object${number}_time.tv_sec)*1000.0;
            outvalue.val_float += (t.tv_usec - object${number}_time.tv_usec)/1000.0;
    $attachedobjectcalls
    return;
```

```
default: /* not an inlet */
            return;
    }
}
];

$globalinit .= qq[
  /* now initializing realtime $number: */
  gettimeofday(&object${number}_time,NULL);
];
}
  },
     'spigot' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::spigot';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

$attachedobjectcalls =~ s/&outvalue/message/g;

print qq[
/* object $number is a SPIGOT object */
void object$number (int inletnumber, const struct PDdata * message) {
    static int open = $initializer;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            if (open) {
                $attachedobjectcalls
            }
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("spigot takes only takes float in right inlet");
        return;
            }
    open = (message->val_float!=0);
    break;
default: /* not an inlet */
            return;
    }
    return;
}
];
}
  },
     'clip' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::clip';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my ($initmin, $initmax) = (0,0);
$initmin = $arguments[0]
  if (exists $arguments[0]);
$initmax = $arguments[1]
  if (exists $arguments[1]);

print qq[
/* object $number is a CLIP object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float min = $initmin, max = $initmax, lastout;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch (message->type) {
                case PDC_TYPE_BANG:
                    outvalue.type = PDC_TYPE_FLOAT;
                    outvalue.val_float = lastout;
                    $attachedobjectcalls
                    return;
                case PDC_TYPE_FLOAT:
                    lastout = message->val_float;
                    if (message->linked && message->linked->type == PDC_TYPE_FLOAT) {
                        min = message->linked->val_float;
                        if (message->linked->linked && message->linked->linked->type == PDC_TYPE_FLOAT)
                            max = message->linked->linked->val_float;
                    }
                    if (lastout < min) lastout=min;
                    else if (lastout > max) lastout=max;
                    outvalue.type = PDC_TYPE_FLOAT;
                    outvalue.val_float = lastout;
                    $attachedobjectcalls;
                    return;
                default:
                    error("clip takes only floats");
                    return;
            }
    break;
case 1: /* the cold min inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("spigot takes only takes floats");
        return;
            }
    min = message->val_float;
    break;
case 2: /* the cold max inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("spigot takes only takes floats");
        return;
            }
    max = message->val_float;
    break;
default: /* not an inlet */
            return;
    }
```

```
    return;
}
];
}
  },
     'trigger' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::trigger';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# parse ARGS to some subset of f\S* b\S* s\S* l\S* a\S*
# i'll just generate all of them and make aliases. (especially since lists and anythings will just be copies)

# 1- static bang elt
# 2- semistatic float elt
# 3- semistatic symbol elt

for (reverse 0 .. $#arguments) {
  # have to do the last one first; i was stupid and defined the 1st
  # output as "outvalue" while subsequent ones are labeled outvalue2, 3, &c.
  my $curname = "outvalue".(($_!=0)?(1+$_):"");
  if ($arguments[$_] =~ /^b/i) {
    $attachedobjectcalls =~ s/$curname/outbang/g;
  } elsif ($arguments[$_] =~ /^s/i) {
    $attachedobjectcalls =~ s/$curname/outsym/g;
  } elsif ($arguments[$_] =~ /^[la]/i) {
    $attachedobjectcalls =~ s/\&$curname/message/g;
  } else { # apparently it's otherwise treated as a float
    $attachedobjectcalls =~ s/$curname/outfloat/g;
  }
}

print qq[
/* object $number is a TRIGGER object */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outbang={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};
    struct PDdata outfloat={PDC_TYPE_FLOAT,0,NULL,0,0,NULL,NULL};
    struct PDdata outsym={PDC_TYPE_SYMBOL,0,NULL,0,0,NULL,NULL};
    const char flstr[] = {"float"}, symstr[]={"symbol"};

    if (inletnumber != 0 || message->type == PDC_DSP_REQUEST)
        return;

    switch(message->type) { /* This is right! */
        case PDC_TYPE_FLOAT:
            outfloat.val_float = message->val_float;
            outsym.val_symbol = flstr;
            break;
        case PDC_TYPE_BANG:
            outsym.val_symbol = symstr;
            break;
    }

    /* PuDaC should have renamed everything in the following calls */
    $attachedobjectcalls
    return;
}
];
}
  },
     'bng' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::bng';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# lots of stupid arguments
# relevant bit:
#  if $arguments[3], register for a loadbang
#  if $arguments[4] != "empty", add a "sendbyname"
#  if $arguments[5] != "empty", listens for $arguments[5]
# plus, bng (unlike tgl or friends) doesn't care about [set X< messages

# our implementation is quite incomplete; there are a wide variety
# of settings that bangers and toggles are supposed to support (via
# symbol-value messages)
# the important missing part is that WE DO NOT CHANGE THE SENDER
# NAME and CANNOT CHANGE THE RECEIVER NAME. (My architecture was
# perhaps misdesigned...)

print qq[
/* object $number is a BANGER */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (!(inletnumber == 0 || inletnumber == PDC_CALLBACK)) return;
    if (message && message->type == PDC_DSP_REQUEST) return;

    $attachedobjectcalls
];
if ($arguments[4] and lc $arguments[4] ne "empty") {
  print qq[sendbyname("$arguments[4]",message);\n];
        }
        print qq[
}
];
if ($arguments[5] and lc $arguments[5] ne "empty") {
  $globalinit .= qq[
  /* object $number is a banger */
  listens(\& object${number},"$arguments[5]");
];
        }
        if ($arguments[3]) {
    $loadbangs .= qq[
/* now call banger obj $number: */
object$number(PDC_CALLBACK,NULL);
];
        }

 }
          },
     'tgl' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::tgl';
my ($number, %myself, @error) = @_;
```

```
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# lots of stupid arguments
# relevant bit:
#  if $arguments[1], set on creation
#  if $arguments[2] != "empty", add a "sendbyname"
#  if $arguments[3] != "empty", listens for $arguments[5]
#  $arguments[12] contains the non-zero value if $arguments[1]
#  $arguments[13] contains the non-zero value if not $arguments[1]

my ($nzval,$curval)=(1,0);
        if ($arguments[1] and $arguments[12]) {
  $nzval = $curval = $arguments[12];
} else {
  $nzval = $arguments[13];
        }

# our implementation is quite incomplete; there are a wide variety
# of settings that bangers and toggles are supposed to support (via
# symbol-value messages)

# the important missing part is that WE DO NOT CHANGE THE SENDER
# NAME and CANNOT CHANGE THE RECEIVER NAME. (My architecture was
# perhaps misdesigned...)

print qq[
/* object $number is a TOGGLE */
void object$number (int inletnumber, const struct PDdata * message) {
    static float nonzerovalue=$nzval, curvalue=$curval;
    struct PDdata outvalue={PDC_TYPE_FLOAT,0,NULL,0,0,NULL,NULL};

    if (message->type == PDC_DSP_REQUEST) return;
    if (!(inletnumber == 0 || inletnumber == PDC_CALLBACK || inletnumber == PDC_SENT)) return;

    switch (message->type) {
        case PDC_TYPE_FLOAT:
            if (message->val_float == 0)
                curvalue =0;
            else
                nonzerovalue = curvalue = message->val_float;
            break;
        case PDC_TYPE_BANG:
            if (curvalue == 0) curvalue = nonzerovalue;
            else curvalue = 0 ;
            break;
        case PDC_TYPE_SYMBOL:
            if (strcasecmp("set",message->val_symbol)==0) {
                if (message->linked && message->linked->type == PDC_TYPE_FLOAT) {
                    if (message->linked->val_float == 0) curvalue = 0;
                    else nonzerovalue = curvalue = message->linked->val_float;
                } else
                    error("toggle box doesn't understand message 'set [nonfloat]'");
            } else
                error("toggle box doesn't understand message starting with \%s",message->val_symbol);
            return;
    }

    outvalue.val_float = curvalue;
    $attachedobjectcalls
];
if ($arguments[2] and lc $arguments[2] ne "empty") {
  print qq[sendbyname("$arguments[2]",message);\n];
        }
        print qq[
}
];
if ($arguments[3] and lc $arguments[3] ne "empty") {
  $globalinit .= qq[
  /* object $number is a toggle */
  listens(\& object${number},"$arguments[5]");
];
        }
 }
          },
     '\floatatom' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::\floatatom';
my ($number, %myself, @error) = @_;
my ($rcv,$snd) = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# lots of stupid arguments (already stripped in xtypes)
# relevant bit:
#  if $snd != "-", add a "sendbyname"
#  if $rcv != "-", listens for $rcv

print qq[
/* object $number is a FLOATATOM */
void object$number (int inletnumber, const struct PDdata * message) {
    static float curvalue=0;
    struct PDdata outvalue={PDC_TYPE_FLOAT,0,NULL,0,0,NULL,NULL};

    if (message->type == PDC_DSP_REQUEST) return;
    if (!(inletnumber == 0 || inletnumber == PDC_CALLBACK || inletnumber == PDC_SENT)) return;

    switch (message->type) {
        case PDC_TYPE_FLOAT:
            curvalue = message->val_float;
            break;
        case PDC_TYPE_SYMBOL:
            if (strcasecmp("set",message->val_symbol)==0) {
                if (message->linked && message->linked->type == PDC_TYPE_FLOAT) {
                    curvalue = message->linked->val_float;
                } else
                    error("floatatom doesn't understand message 'set [nonfloat]'");
            } else
                error("floatatom doesn't understand message starting with \%s",message->val_symbol);
            return;
    }

    outvalue.val_float = curvalue;
    $attachedobjectcalls
/*}*/]; #]
if ($arguments[2] and lc $arguments[2] ne "-") {
  print qq[sendbyname("$arguments[2]",message);\n];
```

```
        }
        print "}\n";

if ($arguments[3] and lc $arguments[3] ne "-") {
  $globalinit .= qq[
  /* object $number is a floatatom */
  listens(\& object${number},"$arguments[5]");
]; #]
        }
 }
          },
     'unpack' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::unpack';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# unpacks have # args = # outlets (except 0 args = "f f")
# all args are s\S* f\S* \d+ p\S*
# it's either explicitly a Symbol, a Pointer, or else it's a float.

# so handle default:
@arguments=("f","f") if (scalar @arguments == 0);

# unbundle attachedobjectcalls:
my @AOCsByOutlet;
for (split "\n",$attachedobjectcalls) {
  if (/outvalue(\d*)/) {
    my $t = $1;
    if ($t>0) { $t--; }
    else { $t=0; }
    if ($arguments[$t] =~ /^s/i) { # symbol
      s/outvalue\d*/outsymb/;
    } else {
      s/outvalue\d*/outfloat/;
    }
    $AOCsByOutlet[$t] .= "$_\n";
  } else {
    carp "Didn't understand a call i generated?  [$_]\n";
  }
}

print qq[
/* object $number is an UNPACK object */
void object$number (int inletnumber, const struct PDdata * message) {
    int i;
    struct PDdata outfloat={PDC_TYPE_FLOAT,0,NULL,0,0,NULL,NULL};
    struct PDdata outsymb={PDC_TYPE_SYMBOL,0,NULL,0,0,NULL,NULL};

    if (message->type == PDC_DSP_REQUEST || !(inletnumber == 0)) return;
/*} cperl */
];#]
for (my $j=$#arguments;$j>=0;$j--) {
        next if (length $AOCsByOutlet[$j]==0);
  if ($arguments[$j] =~ /^s/i) {
    # a symbol:
    print qq[
    if (linkedlength(message) > $j && linkedelt(message,$j)->type == PDC_TYPE_SYMBOL) {
        outsymb.val_symbol = linkedelt(message,$j)->val_symbol;
        $AOCsByOutlet[$j];
    } else
        error("Unpack got non-symbol for a symbol slot");
];#]
         } elsif ($arguments[$i] =~ /^p/i) {
    carp "PuDaC doesn't deal with pointers";
  } else { # a float
    print qq[
    if (linkedlength(message) > $j && linkedelt(message,$j)->type == PDC_TYPE_FLOAT) {
        outfloat.val_float = linkedelt(message,$j)->val_float;
        $AOCsByOutlet[$j];
    } else
        error("Unpack got non-symbol for a symbol slot");
]; #]
  }
}

print "}\n";
}
          },
     'select' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::select';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# select comes in four flavors: float selector and symbol selector; zero-or-one args or more
# augh!

# select with no args compares things to zero.

# if the first arg is a float, we cast everything else to float (symbols become 0)

@arguments = ( 0 )
  if (scalar @arguments == 0);

# unbundle attachedobjectcalls:
my @AOCsByOutlet;
for (split "\n",$attachedobjectcalls) {
  if (/outvalue(\d*)/) {
    my $t = $1;
    if ($t>0) { $t--; }
    else { $t=0; }
    if ($t != scalar @arguments) { # symbol
      s/outvalue\d*/outvalue/;
    } else {
      s/\&outvalue\d*/message/;
    }
    $AOCsByOutlet[$t] .= "$_\n";
  } else {
    carp "Didn't understand a call i generated?  [$_]\n";
  }
}

if (scalar @arguments <= 1) {
  my $init = $arguments[0];
```

```
  if ($arguments[0] =~ /^-?(\d*\.)?\d+([eE][-+]?\d+)?$/ or not $arguments[0]) {
    # is no args or a single numeric arg:
    print qq[
/* object $number is a SELECT object with 0/1 numeric args */
void object$number (int inletnumber, const struct PDdata * message) {
    static float comvalue=$init;
    struct PDdata outvalue={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (!(message->type == PDC_TYPE_FLOAT || message->type == PDC_TYPE_SYMBOL))
      return;

    switch (inletnumber) {
        case 0: /* HOT */
            if (message->type == PDC_TYPE_FLOAT && comvalue == message->val_float) {
                $AOCsByOutlet[0]
            } else {
                $AOCsByOutlet[1]
            }
            break;
        case 1: /* COLD */
            if (message->type != PDC_TYPE_FLOAT) {
                error("This selector ($number) only knows how to compare floats");
                return;
    }
            comvalue = message->val_float;
            break;
    }
}
];
  } else {
    # is 1 non-numeric arg:
    print qq[
/* object $number is a SELECT object with 1 symbol arg */
void object$number (int inletnumber, const struct PDdata * message) {
    static char * comvalue = NULL;
    struct PDdata outvalue={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (comvalue == NULL) {
        comvalue = malloc(1+strlen("$init"));
        if (comvalue == NULL) { error("MEMORY ALLOC FAIL"); exit(1); }
        strcpy(comvalue,"$init");
    }

    if (!(message->type == PDC_TYPE_FLOAT || message->type == PDC_TYPE_SYMBOL))
      return;

    if (message->type == PDC_DSP_REQUEST) return;
    switch (inletnumber) {
        case 0: /* HOT */
            if (message->type == PDC_TYPE_SYMBOL && strcmp(comvalue,message->val_symbol)==0) {
                $AOCsByOutlet[0]
            } else {
                $AOCsByOutlet[1]
            }
            break;
        case 1: /* COLD */
            if (message->type != PDC_TYPE_SYMBOL) {
                error("This selector ($number) only knows how to compare symbols");
                return;
    }
            comvalue = realloc(comvalue,1+strlen(message->val_symbol));
            if (comvalue == NULL) { error("MEMORY ALLOC FAIL"); exit(1); }
            strcpy(comvalue,message->val_symbol);
            break;
      }
}
];
  }
        } else { # has more than one arg, ugh
  if ($arguments[0] =~ /^-?(\d*\.)?\d+([eE][-+]?\d+)?$/ or not $arguments[0]) {
    # is comparing any number of floats
    # first force all args to float type
    @arguments = map { (/^-?(\d*\.)?\d+([eE][-+]?\d+)?$/) ? $_ : 0 } @arguments;
    # fortunately we can't change anything after creation.
    print qq[
/* object $number is a SELECT object with >= 2 numeric args */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (!(message->type == PDC_TYPE_FLOAT || message->type == PDC_TYPE_SYMBOL) || inletnumber != 0 )
      return;
];
    if (length $AOCsByOutlet[scalar @arguments]) {
      print qq[
    if (message->type == PDC_TYPE_SYMBOL) {
        ].$AOCsByOutlet[scalar @arguments].qq[
        return;
    }
];
    }

    for (my $i=0;$i<=$#arguments;$i++) {
      # apparently PD itself just does == on the floating point numbers?
      print qq[
    if (message->val_float == $arguments[$i]) {
        $AOCsByOutlet[$i]
        return;
    }
];
    }
    print "\n".$AOCsByOutlet[scalar @arguments]."\n}\n";
  } else {
    # is comparing any number of symbols
    # first convert all fp numbers to null strings. (augh, stupid puredata)
    @arguments = map { (/^-?(\d*\.)?\d+([eE][-+]?\d+)?$/) ? "" : $_ } @arguments;

    print qq[
/* object $number is a SELECT object with >= 2 symbolic args */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (!(message->type == PDC_TYPE_FLOAT || message->type == PDC_TYPE_SYMBOL) || inletnumber != 0 )
      return;
];
    if (length $AOCsByOutlet[scalar @arguments]) {
```

```
      print qq[
    if (message->type == PDC_TYPE_FLOAT) {
        ].$AOCsByOutlet[scalar @arguments].qq[
        return;
    }
];
    }

    for (my $i=0;$i<=$#arguments;$i++) {
      # apparently PD itself just does == on the floating point numbers?
      print qq[
    if (strcmp(message->val_symbol,"$arguments[$i]")==0) {
        $AOCsByOutlet[$i]
        return;
    }
];
    }
    print "\n".$AOCsByOutlet[scalar @arguments]."\n}\n";
  }
}
}
          },
     'route' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::route';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# route comes in two flavors: float router, and symbol/type router

# symbol/type router is technically a side affect of how puredata
# stores types (with a word at the beginning specifying
# type. oops.)

# route without any args compares things to zero

# if the first arg is a float, we cast everything else to float (symbols become 0)

@arguments = ( 0 )
  if (scalar @arguments == 0);

if (not $ROUTEwarning) {
  print STDERR "Please be careful; PuDaC doesn't have Anythings, so routing by type is Very different\n";
  $ROUTEwarning =1;
}

# unbundle attachedobjectcalls:
my @AOCsByOutlet;
for (split "\n",$attachedobjectcalls) {
  if (/outvalue(\d*)/) {
    my $t = $1;
    if ($t>0) { $t--; }
    else { $t=0; }
    if ($t != scalar @arguments) { # symbol
      s/\&outvalue\d*/message->linked/g;
    } else {
      s/\&outvalue\d*/message/g;
    }
    $AOCsByOutlet[$t] .= "$_\n";
  } else {
    carp "Didn't understand a call i generated?  [$_]\n";
  }
}

if ($arguments[0] =~ /^-?(\d*\.)?\d+([eE][-+]?\d+)?$/ or not $arguments[0]) {
  # is comparing any number of floats
  # first force all args to float type
  @arguments = map { (/^-?(\d*\.)?\d+([eE][-+]?\d+)?$/) ? $_ : 0 } @arguments;
  # fortunately we can't change anything after creation.
  print qq[
/* object $number is a ROUTE object with numeric args */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata banger={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (!(message->type == PDC_TYPE_FLOAT || message->type == PDC_TYPE_SYMBOL) || inletnumber != 0 )
      return;
];
  if (length $AOCsByOutlet[scalar @arguments]) {
    print qq[
    if (message->type == PDC_TYPE_SYMBOL) {
        ].$AOCsByOutlet[scalar @arguments].qq[
        return;
    }
];
  }

  for (my $i=0;$i<=$#arguments;$i++) {
    # apparently PD itself just does == on the floating point numbers?
    print qq[
    if (message->val_float == $arguments[$i]) {
        if (message->linked) {
            $AOCsByOutlet[$i]
        } else { ];
    my $t = $AOCsByOutlet[$i]; $t =~ s/message->linked/&banger/g;
    print qq[
            $t
        }
        return;
    }
];
  }
  print "\n".$AOCsByOutlet[scalar @arguments]."\n}\n";
} else {
  # is comparing any number of symbols
  # first convert all fp numbers to null strings. (augh, stupid puredata)
  @arguments = map { (/^-?(\d*\.)?\d+([eE][-+]?\d+)?$/) ? "" : $_ } @arguments;
  print qq[
/* object $number is a ROUTE object with symbolic args / type router */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata banger={PDC_TYPE_BANG,0,NULL,0,0,NULL,NULL};

    if (message->type == PDC_DSP_REQUEST || inletnumber != 0 )
      return;
];
  # XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX
```

```
  # here's another place where my abstraction's gotten me into trouble:
  # I can't tell the difference between symbols and anythings.
  # To work around this, we make symbol and list bind more loosely than anything but failure.

  # maybe I should make a hash?
  for (my $i=$#arguments;$i>=0;$i--) {
    $codefor{$arguments[$i]} = $AOCsByOutlet[$i]; # the assignment occludes later matches
  }

  if (exists $codefor{'bang'}) {
    $codefor{'bang'} =~ s/message->linked/message/g;
    print qq[
    if (message->type == PDC_TYPE_BANG) {
        $codefor{bang}
        return;
    }
];
  }
  if (exists $codefor{'float'}) {
    $codefor{'float'} =~ s/message->linked/message/g;
    print qq[
    if (message->type == PDC_TYPE_FLOAT && message->linked == NULL) {
        $codefor{float}
        return;
    }
];
  }
  # now we try everything /except/ bang float symbol and list
  for (keys %codefor) {
    # (bang and float we've already tested, so we skip them for non-stupid code. )
    #  list and symbol should bind more loosely
    next if (/^(bang|float|list|symbol)$/);
    #everything else should be the first elt of an anything
    print qq[
    if (message->type == PDC_TYPE_SYMBOL && strcmp(message->val_symbol,"$_")==0) {
        $codefor{$_}
        return;
    }
];
        }
  if (exists $codefor{'symbol'}) {
    print qq[
    if (message->type == PDC_TYPE_SYMBOL && message->linked == NULL) {
        $codefor{symbol}
        return;
    }
];
  }
  if (exists $codefor{'list'}) {
    print qq[
    if (message->linked != NULL) {
        $codefor{list}
        return;
    }
];
  }
  # default:
  print "\n".$AOCsByOutlet[scalar @arguments]."\n}\n";
}
}
        },
     'netreceive' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::netreceive';
my ($number, %myself, @error) = @_;
my ($port,$mode) = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# port: zero means any; non-zero means that port
# mode: zero means TCP; non-zero means UDP

if ($mode) { # UDP
  $attachedobjectcalls =~ s/\&outvalue/outarray/g;

  print qq[
/* object $number is a NETRECEIVE (UDP) object */
int object${number}_fd;
struct sockaddr_in object${number}_saddr;
struct pollfd object${number}_poll;

void object$number (int inletnumber, const struct PDdata * message) {
    int n, i, from, to;
    char buffer[4097];
    char *p, *tester, *restart, *nextcomma;
    char * pointers[2048];
    struct PDdata outarray[2048];
    float t;

    n = recv(object${number}_fd,buffer+1,4096,0); // this version blocks. need to fix that.
    // recv doesn't add a null terminator.
    buffer[n+1]=0;
    buffer[0]=',';
    // PD's netreceive:
    //  --UDP version--
    //  Doesn't parse anything after the first semicolon on a line
    //  Each datagram is parsed as a message, regardless of whether it contains a semicolon
    //  Commas divide a message into multiple messages
    //  --TCP version--
    //  Each semicolon separates messages.
    //  Commas divide a message into multiple messages
    //  Packets are accumulated until a semicolon is found

    // Either way, it does the stupid named casting thing that's getting me in trouble everywhere.

    memset(pointers,0,2048*sizeof(char *));
    memset(outarray,0,2048*sizeof(struct PDdata));

    // With UDP datagrams, if it contains a semicolon, we forget everything after.
    if (p = strchr(buffer,';')) {
      *p = 0;
      n=p-buffer;
    }

    restart=buffer;
    while (restart-buffer < n) {
      restart++;
```

```
      nextcomma = strchr(restart,',');
      if (nextcomma) *nextcomma = 0;
      else nextcomma = buffer+n;

      // Parse things out.
      p=restart; i=0;
      do {
p--; //stupid fidgeting
while (isspace(*++p)) { if (! *p) break; }
//T should point to next non-space, so make note of it
pointers[i++]=p;

while (!isspace(*++p)) { if (! *p) break; }
// T should point to next space, so null-terminate
*p++=0; // more stupid fidgeting

      } while (p < nextcomma && i < 2047);

      if (strcmp(pointers[0],"bang")==0) {
outarray[0].type = PDC_TYPE_BANG;
from=to=i; // do not parse
      } else if (strcmp(pointers[0],"float")==0) {
outarray[0].type = PDC_TYPE_FLOAT;
outarray[0].val_float = strtod(pointers[1],NULL);
from=to=i;
      } else if (strcmp(pointers[0],"symbol")==0) {
outarray[0].type = PDC_TYPE_SYMBOL;
outarray[0].val_symbol = pointers[1];
from=to=i;
      } else if (strcmp(pointers[0],"list")==0) {
from=1; to=0;
      } else {
from=to=0;
      }

      for (;from<i;from++,to++) {
t = strtod(pointers[from],&tester); // I'd like to use strtof and avoid the cast, but ...
if (tester-pointers[from] == strlen(pointers[from])) {
  outarray[to].type = PDC_TYPE_FLOAT;
  outarray[to].val_float = t;
} else { // must be a symbol
  outarray[to].type = PDC_TYPE_SYMBOL;
  outarray[to].val_symbol = pointers[from];
}
outarray[to].linked=&outarray[to+1];
      }
      outarray[to-1].linked=NULL; // undo that.

      $attachedobjectcalls

      restart = nextcomma;
    }
}
];

 $globalinit .= qq[
  /* now initializing netreceive $number: */
  if((object${number}_fd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP))<0) {
    error("can't create udp socket, aborting");
    exit(2);
  }

  object${number}_saddr.sin_family      = AF_INET;
  object${number}_saddr.sin_port        = htons($port);
  object${number}_saddr.sin_addr.s_addr = htonl(INADDR_ANY);

  if(bind(object${number}_fd,(struct sockaddr *)&object${number}_saddr,sizeof(struct sockaddr_in))<0) {
    error("can't bind udp socket to port $port");
    perror("");
    exit(2);
  }

  object${number}_poll.fd = object${number}_fd;
  object${number}_poll.events = POLLIN;
];
 $periteration .= qq[
/* Now checking if traffic has arrived for netreceive $number: */
/* This implementation may yield a DoS? Maybe s/while/if/ */
   while (poll(&object${number}_poll,1,0)) {
      object$number(PDC_CALLBACK,NULL);
   }
];
} else {
  croak "Haven't implemented TCP netreceive yet, sorry.";
}
}
  },
      'tabread~' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::tabread~';
 my ($number, %myself, @error) = @_;
 my ($name) = @{$myself{ARGS}};
 my $reverseDSPcalls = $myself{DSPCS};

 print qq[
/* object $number is a TABREAD~ object */
float *object${number}_upstr, *object${number}_cache;

void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata totable={PDC_TYPE_SYMBOL,0,"\\\\getguts",0,0,NULL,NULL};
    static struct PDdata DSPtrace1={PDC_DSP_REQUEST,0,NULL,-1,0,NULL,NULL};
    static float * cache = NULL;
    static float * table = NULL;
    static int tablesize;
    float * writer;
    static unsigned int pos;
    int i;
    char * write_is_dsp;

    if (NULL == DSPtrace1.dsp_stream) DSPtrace1.dsp_stream = object${number}_upstr;
    if (NULL == cache) cache = object${number}_cache;
    if (NULL == table) {
      sendbyname("$name",&totable);
      table = totable.dsp_stream;
      tablesize = totable.sample_count; /* This is bad. We can't dynamically resize things */
    }
```

```
    writer = message->dsp_stream; /* break const-ness of message */

    switch (inletnumber) {
        case -1: /* outlet 1 */
            if (message->type != PDC_DSP_REQUEST)
                return;
            write_is_dsp = &message->is_dsp;
            *write_is_dsp=1;
            /* find out if we have a dsp rate input */
            if (message->sample_count != DSPtrace1.sample_count) {
                memset(DSPtrace1.dsp_stream,0,block_size * sizeof(float));
                DSPtrace1.sample_count = message->sample_count;
                $reverseDSPcalls
                if (DSPtrace1.is_dsp) {
                    for (i=0;i<block_size;i++) {
                        if (DSPtrace1.dsp_stream[i] < 0) cache[i] = table[0];
                        else if (DSPtrace1.dsp_stream[i] >= tablesize) cache[i] = table[tablesize-1];
                        else cache[i] = table[(unsigned int)DSPtrace1.dsp_stream[i]];
                    }
                } else {
                    for (i=0;i<block_size;i++) {
                        cache[i]=table[pos];
                    }
                }
            }
            for (i=0;i<block_size;i++) {
                writer[i]+=cache[i];
            }
            break;
        case 0: /* inlet 1 = position */
            if (message->type == PDC_TYPE_FLOAT) {
                if (message->val_float < 0) pos = 0;
                else if (message->val_float >= tablesize) pos=tablesize-1;
                else pos = message->val_float;
            } else
                error("tabread~ takes only floats or dsp chains in idx inlet (got \%d)",message->type);

            break;
        default:
            error("tabread~ only has one inlet (tried \%d)",inletnumber);
            return;
    }
}
];#]]]]

 $globalinit .= qq[
  /* allocate "private" memory for tabread~ $number */
  object${number}_upstr = malloc(block_size*sizeof(float));
  object${number}_cache = malloc(block_size*sizeof(float));

  if (object${number}_upstr == NULL || object${number}_cache == NULL) {
    error("Unable to allocate memory for obj $number (tabread~)!");
    exit(1);
  }
];
       }
       },
     'line~' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::line~';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};

 print qq[
/* object $number is an LINE~ object */
float *object${number}_cache;

void object$number (int inletnumber, const struct PDdata * message) {
    static int cachepos;
    static float * cache = NULL;
    float * writer;
    static float curpos, dest;
    static int blockstogo;
    int i;
    char * write_is_dsp;

    if (NULL == cache) cache = object${number}_cache;

    writer = message->dsp_stream; /* break const-ness of message */

    switch (inletnumber) {
        case -1: /* outlet 1 */
            if (message->type != PDC_DSP_REQUEST)
                return;
            write_is_dsp = &message->is_dsp;
            *write_is_dsp=1;
            /* find out if we have a dsp rate input */
            if (message->sample_count != cachepos) {
                cachepos = message->sample_count;
                if (blockstogo) {
                    /* (dest-curpos)/blockstogo */
                    for (i=0;i<block_size;i++)
                        cache[i]=i*((dest-curpos)/blockstogo)/block_size+curpos;
                    curpos += (dest-curpos)/blockstogo;
                    blockstogo--;
                } else
                    for (i=0;i<block_size;i++)
                        cache[i] = dest;
            }
            for (i=0;i<block_size;i++) {
                writer[i]+=cache[i];
            }
            break;
        case 0: /* inlet 1 = numbers or lists */
            if (message->type == PDC_TYPE_FLOAT)
                dest = message->val_float;
            else {
                error("line~ takes only floats or dsp chains in left inlet (got \%d)",message->type);
                return;
            }

            if (message->linked)
                if (message->linked->type == PDC_TYPE_FLOAT) {
                    blockstogo = message->linked->val_float * sample_rate / block_size / 1000;
                    if (blockstogo < 0) blockstogo = 0;
```

```
                } else {
                    error("line~ takes only float as 2nd elt of list in 1st inlet (got \%d)",message->type);
                    return;
                }
            else curpos=dest; // Act immediately if we got a single number

            break;
        case 1: /* inlet 2 = time for next line~ to take */
            /* only if numeric */
            error("line~ incomplete; does not accept input in right inlet.");
            break;
        default:
            error("line~ only has two inlets (tried \%d)",inletnumber);
            return;
    }
}
];#]]]]

 $globalinit .= qq[
  /* allocate "private" memory for line~ $number */
  object${number}_cache = malloc(block_size*sizeof(float));

  if (object${number}_cache == NULL) {
    error("Unable to allocate memory for obj $number (line~)!");
    exit(1);
  }
];
       }
       },
     'max' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::max';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a MAX object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                if (message->linked != NULL) {
                    error("max doesn't understand lists");
                    return;
                }
                lastleft = message->val_float;
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = (lastleft>lastright)?lastleft:lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = (lastleft>lastright)?lastleft:lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("+ doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("max takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
            return;
    }
    return;
}
];
}
  },
     'min' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::min';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a MIN object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                if (message->linked != NULL) {
                    error("min doesn't understand lists");
                    return;
                }
```

```
                lastleft = message->val_float;
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = (lastleft<lastright)?lastleft:lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = (lastleft<lastright)?lastleft:lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("+ doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("min takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
            return;
    }
    return;
}
];
}
  },
    'sqrt' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::sqrt';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is a SQRT object */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue;

    outvalue.linked = NULL;
    if (message->type != PDC_TYPE_FLOAT || inletnumber != 0)
        return;

    outvalue.type = PDC_TYPE_FLOAT;
    if (message->val_float > 0)
        outvalue.val_float = sqrt(message->val_float);
    else
        outvalue.val_float = 0;

    $attachedobjectcalls
}
];
}
  },
    'abs' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::abs';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is an ABS object */
void object$number (int inletnumber, const struct PDdata * message) {
    struct PDdata outvalue;

    outvalue.linked = NULL;
    if (message->type != PDC_TYPE_FLOAT || inletnumber != 0)
        return;

    outvalue.type = PDC_TYPE_FLOAT;
    outvalue.val_float = fabs(message->val_float);

    $attachedobjectcalls
}
];
}
  },
    'delay' => { 'CODE' => sub {
 local *__ANON__ = 'PuDaC::Objects::delay';
 my ($number, %myself, @error) = @_;
 my @arguments = @{$myself{ARGS}};
 my $attachedobjectcalls = $myself{AOCS};

 # i've evidently decided to implement delay in terms of metro.

 my $initializer = 0;
 $initializer = $arguments[0]
   if (exists $arguments[0]);

 print qq[
/* object $number is a DELAY object */

struct timeval object${number}_time;
int object${number}_running;

void object$number (int inletnumber, const struct PDdata * message) {
    static float delay_in_milliseconds=$initializer;
    struct PDdata outvalue;
    struct timeval TV_internal;

    outvalue.linked = NULL;
    outvalue.type = PDC_TYPE_BANG;
```

```
    switch (inletnumber) {
        case PDC_CALLBACK: /* a timeout */
            object${number}_running=0;
            /* this order is vital: otherwise we can't trigger ourselves to restart*/
            $attachedobjectcalls
            break;
case 0: /* the hot inlet */
            switch(message->type) {
            case PDC_TYPE_FLOAT: /* N means schedule a timeout for N milliseconds */
                delay_in_milliseconds = message->val_float;
            case PDC_TYPE_BANG: /* bang means run */
                object${number}_running = 1;
                break;
            case PDC_TYPE_SYMBOL: /* "stop" means stop */
                if (strcmp("stop",message->val_symbol)==0) {
                    object${number}_running = 0;
                }
                error("delay doesn't understand that symbol");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    if (object${number}_running) {
                gettimeofday(&TV_internal,NULL);
                object${number}_time.tv_usec = TV_internal.tv_usec
                                             + delay_in_milliseconds*1000;
                object${number}_time.tv_sec = TV_internal.tv_sec;
                while (object${number}_time.tv_usec > 1000000) {
                    object${number}_time.tv_sec++;
                    object${number}_time.tv_usec-=1000000;
                }
            }
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
        error("delay takes only takes float in right inlet");
        return;
            }
    delay_in_milliseconds = message->val_float;
    break;
default: /* not an inlet */
            return;
    }
    return;
}
];

 $globalinit .= qq[
  /* now initializing delay $number: */
  object${number}_running = 0;
];
 $periteration .= qq[
/* Now checking timeout for obj $number: */
if (object${number}_running &&
    (now.tv_sec > object${number}_time.tv_sec ||
     (now.tv_sec == object${number}_time.tv_sec &&
      now.tv_usec > object${number}_time.tv_usec))) {
       object${number}(PDC_CALLBACK,NULL);
}
];
      }
},
     'pack' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::pack';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

# packs have # args = # inlets (except 0 args = "f f")
# all args are s\S* f\S* \d+ p\S*
# it's either explicitly a Symbol, a Pointer, or else it's a float.

# so handle default:
@arguments=("f","f") if (scalar @arguments == 0);

s/^f$/0/ig for (@arguments);

print qq[
/* object $number is an PACK object */
void object$number (int inletnumber, const struct PDdata * message) {
    int i;
    static struct PDdata outvalue[]={];
for (my $j=0;$j<=$#arguments;$j++) {
  if ($arguments[$j] =~ /^s/i) { # a symbol
    print "{PDC_TYPE_SYMBOL,0,\"symbol\",0,0,0,";
    if ($j==$#arguments) { print "NULL}};\n" }
    else { print "&outvalue[".($j+1)."]},"; }
  } elsif ($arguments[$i] =~ /^p/i) {
    croak "PuDaC doesn't deal with pointers";
  } else { # a float
    print "{PDC_TYPE_FLOAT,$arguments[$j],0,0,0,0,";
    if ($j==$#arguments) { print "NULL}};\n" }
    else { print "&outvalue[".($j+1)."]},"; }
  }
}

print qq[

    if (message->type == PDC_DSP_REQUEST || inletnumber < 0) return;

    switch(inletnumber) {
];
for (my $j=0;$j<=$#arguments;$j++) {
  if ($arguments[$j] =~ /^s/i) { # a symbol:
    print qq[
    case $j:
        if (message->type == PDC_TYPE_SYMBOL) {
            outvalue[$j].val_symbol = message->val_symbol; // Static-ness of this will probably break!
            error("Likely bug in pack. Check your symbols."); ].
            (($j==0)?$attachedobjectcalls:"").qq[
        } else
            error("pack got non-symbol for a symbol slot");
        break;
];
```

```
          } elsif ($arguments[$i] =~ /^p/i) {
    carp "PuDaC doesn't deal with pointers";
  } else { # a float
    print qq[
    case $j:
        if (message->type == PDC_TYPE_FLOAT) {
            outvalue[$j].val_float = message->val_float; ].
            (($j==0)?$attachedobjectcalls:"").qq[
        } else
            error("pack got non-float for a float slot");
        break;
];
  }
}

print "}\n}\n";
}
          },
     'until' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::+';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

print qq[
/* object $number is an UNTIL object */
void object$number (int inletnumber, const struct PDdata * message) {
    static unsigned int counts=-1;
    struct PDdata outvalue={PDC_TYPE_BANG,0,0,0,0,0,0};
    static int counter;

    switch (inletnumber) {
case 0: /* the burning hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                counts = message->val_float;
                break;
            case PDC_TYPE_BANG:
                break;
            case PDC_TYPE_SYMBOL:
                error("until doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
            counter = counts;
            while (counter--) {
                $attachedobjectcalls
            }
    break;
case 1: /* the stopping cold inlet */
            if (message->type != PDC_TYPE_BANG) {
        error("+ takes only takes float in right inlet");
        return;
            }
    counter=0;
    break;
    }
    return;
}
];
}
  },
     '==' => { 'CODE' => sub {
        local *__ANON__ = 'PuDaC::Objects::==';
my ($number, %myself, @error) = @_;
my @arguments = @{$myself{ARGS}};
my $attachedobjectcalls = $myself{AOCS};

my $initializer = 0;
$initializer = $arguments[0]
  if (exists $arguments[0]);

print qq[
/* object $number is an EQUALS object */
void object$number (int inletnumber, const struct PDdata * message) {
    static float lastleft=0, lastright=$initializer;
    struct PDdata outvalue;

    outvalue.linked = NULL;

    switch (inletnumber) {
case 0: /* the hot inlet */
    /* here the compiler needs to insert the calls to all attached objects*/
            switch(message->type) {
            case PDC_TYPE_FLOAT:
                lastleft = message->val_float;
                if (message->linked != NULL) {
                    if (message->linked->type != PDC_TYPE_FLOAT) {
                        error("+ takes only float as 2nd elt in list");
                        /* however this doesn't fail to propagate */
                    } else {
                        lastright = message->linked->val_float;
                    }
                }
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft == lastright;
                break;
            case PDC_TYPE_BANG:
                outvalue.type = PDC_TYPE_FLOAT;
                outvalue.val_float = lastleft == lastright;
                break;
            case PDC_TYPE_SYMBOL:
                error("== doesn't understand symbols");
                return;
            default:
                error("unknown type \%d",message->type);
                return;
            }
    $attachedobjectcalls
    break;
case 1: /* the cold inlet */
            if (message->type != PDC_TYPE_FLOAT) {
```

```
        error("== takes only takes float in right inlet");
        return;
            }
    lastright = message->val_float;
    break;
default: /* not an inlet */
    error("== only has two inlets");
            return;
    }
    return;
}
];
}
  },
    );

1;

# objects done:
#  == + - * / > < int/i float/f floatatom send/s receive/r max min sqrt abs
#  loadbang table print metro delay timer until bang/b \msg
#  spigot clip trigger unpack pack select route toggle/tgl bng
#  osc~ dac~ tabread~ line~
#  netreceive

# route and select have two forms each (annoyingly):
#  one that deals with symbols, and one that deals with floats.
#  as far as i can tell, there's no good reason for this.
```

Appendix B

Serial converter source

B.1 sliprecv.c

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <netdb.h>

/* Use:
  ./sliprecv-adv /dev/ttyS0 9999

  This should probably support specifying baud rate and host to send to.
*/

#define END     0xC0 /* indicates end of packet */
#define ESC     0xDB /* indicates byte stuffing */
#define ESC_END 0xDC /* ESC ESC_END means END data byte */
#define ESC_ESC 0xDD /* ESC ESC_ESC means ESC data byte */

FILE * serialdev;


/* Recv_packet basically copied from the RFC */
int recv_packet(char * p, int maxlen) {
  unsigned char c;
  int received = 0;

  /* sit in a loop reading bytes until we put together a whole packet.
   * Make sure not to copy them into the packet if we run out of room. */
  while(1) {
    /* get a character to process */
    c = fgetc(serialdev);
    /* handle bytestuffing if necessary */
    switch(c) {
    case END:
      /* if it's an END character then we're done with the packet */
      if (received)
        return received;
      break;
      /* if it's an ESC character, wait and get another character */
    case ESC:
      c = fgetc(serialdev);

      /* if "c" is not one of these two, then we have a protocol violation.  The
         best bet seems to be to leave the byte alone and just stuff it into the
         packet */
      switch(c) {
      case ESC_END: c = END; break;
      case ESC_ESC: c = ESC; break;
      }

      /* here we fall into the default handler and let it store the character
         for us */
    default:
      if(received < maxlen)
        p[received++] = c;
    }
  }
}

void main (int argc, char * argv[]) {
  unsigned short ibuffer[512];
  unsigned char obuffer[2048];
  int len,i,p;
  int sockfd;
  struct sockaddr_in saddr;

  if (argc!=3) {
    fprintf(stderr,"Use: sliprecv-adc [serial port dev] [port]\n");
    return;
  }

  serialdev = fopen(argv[1],"r"); // open?
  if (serialdev == NULL) {
    fprintf(stderr,"Failed to open serial port %s",argv[1]);
    return;
  }

  if((sockfd=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP))<0) {
    fprintf(stderr,"can't create udp socket, aborting\n");
    return;
  }
```

```
  saddr.sin_family = AF_INET;
/*   if (!getnameinfo((struct sockaddr *)&saddr,sizeof(saddr), */
/*                    argv[2],strlen(argv[2]), */
/*                    argv[3],strlen(argv[3]),NI_NAMEREQD|NI_NUMERICHOST|NI_NUMERICSERV)) { */
/*     fprintf(stderr,"Failed to lookup that host/port.\n"); */
/*     return; */
/*   } */
  saddr.sin_port = htons(atoi(argv[2]));
  saddr.sin_addr.s_addr = htonl(0x7F000001); // Ugh.

  while (1) {
    len=recv_packet((char *)ibuffer,1024);
    // ibuffer[0] contains the sender address; ibuffer[1..] contain the payload.
    p=snprintf(obuffer,2048,"d%04X",ibuffer[0]);
    for (i=1;i<len/2;i++)
      p +=snprintf(&obuffer[p],2048-p," %d",ibuffer[i]);
    snprintf(&obuffer[p],2048-p,";\n");
    sendto(sockfd,obuffer,strlen(obuffer),MSG_DONTWAIT,(struct sockaddr *)&saddr,sizeof(saddr));
  }
}
```

Appendix C

Wireless firmware description

Due to licensing terms, the modified source cannot be provided here. However, the modifications were as follows:

C.1 GA-AS.c

The acceleration sensor firmware:

1. Added the include file for the hardware abstraction layer for the ADC
2. Removed support for the LCD, UART, and keys
3. Created a new subroutine I called "noHalAdcRead" which acts just like "HalAdcRead" except it does not enable an input pin, and allows measurement on ADC channels 8-15.
4. Changed the 15 second timeout started when the network start finished initializing to a 1 second timeout
5. Changed the timeout to not send a message but to instead, in order: Find a destination (AutoFindDestination), Bind to that device (EndDeviceBindReq), and start a faster timer
6. Had this faster timer measure all 8 channels of the ADC and temperature and voltage at 100Hz and send them as plain 20-byte packets
7. Had a failure to transmit cause a restart of the Find/Bind/Send sequence.
8. Modifications to elsewhere in the network stack to enable keyless operation

C.2 GA-SB.c

The serial bridge firmware:

1. Removed support for the keys
2. Added a new subroutine I called "SLIPify" which encoded a sequence of bytes with known length into a SLIP[11] encapsulated packet.
3. Initialized the serial port to run at 115200 baud instead of 38400
4. Moved serial port 0 to its alternative location on Port1 (instead of the ZStack default of port 0)
5. Replaced the subroutine called on data receipt with one that would, if given an LCD, dump the first 6 16-bit numbers to the LCD, and would otherwise toggle LED 1, encapsulate the received data, and send it via the serial port.

Appendix D

PuDaC test suite

Here are the patches I used to determine whether my compiler was producing the correct output.

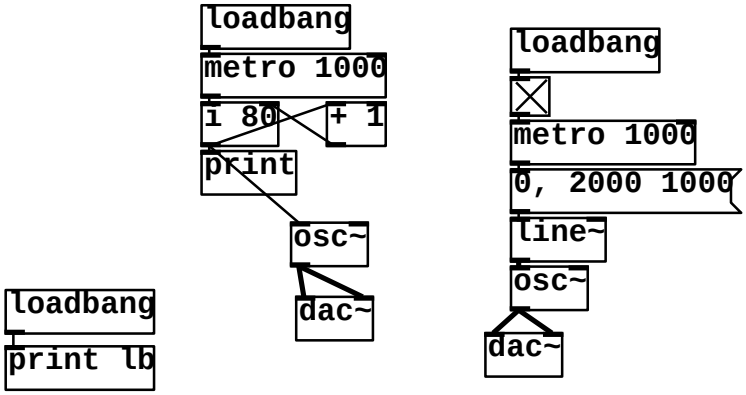

Figure D-1: New tests: `loadbang print metro + i osc~ dac~ line~`

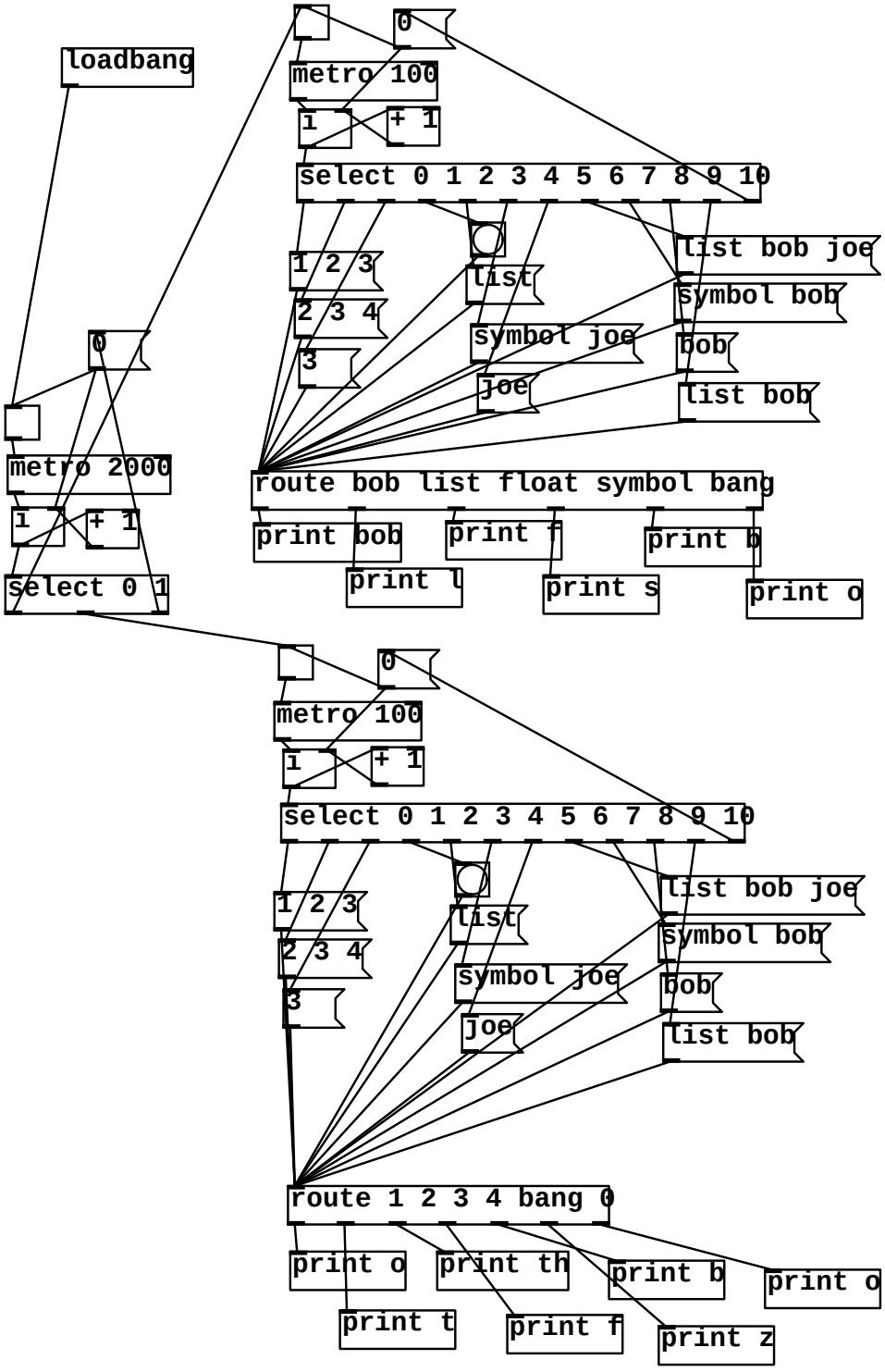

Figure D-2: New tests: `route`

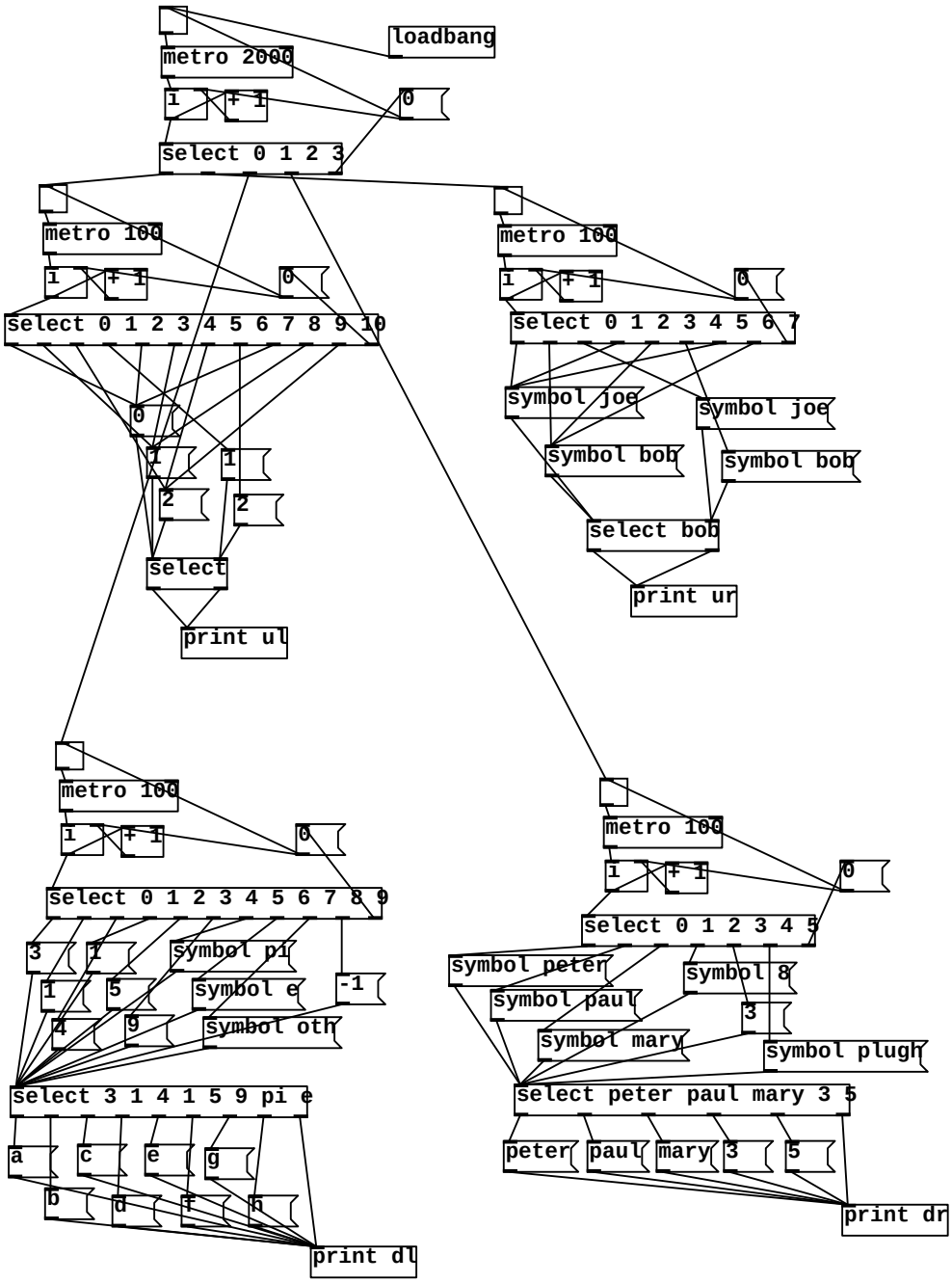

Figure D-3: New tests: `select` messagebox toggle

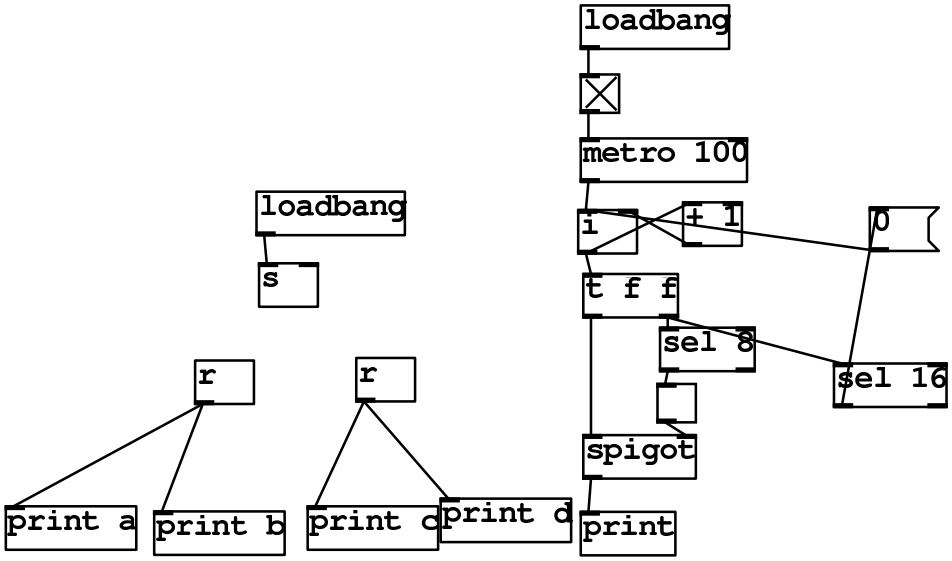

Figure D-4: New tests: `send receive spigot`

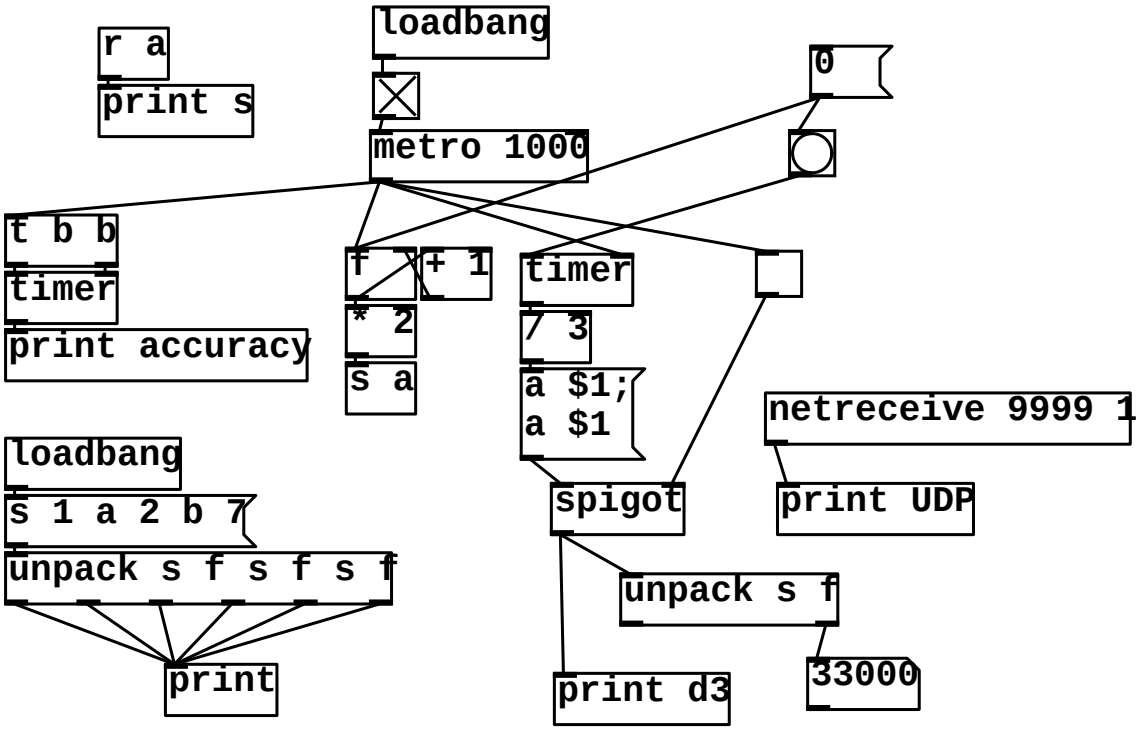

Figure D-5: New tests: `unpack timer trigger netreceive f`

Bibliography

[1] Miller Puckette. Pure data. In *Proceedings of the International Computer Music Conference*, page 269272, San Francisco, 1996. International Computer Music Association.

[2] Jean Laroche. Efficient tempo and beat tracking in audio recordings. *Journal of the Audio Engineering Society*, 51(4):226–233, 2003.

[3] Kent Larson Jason Nawyn, Stephen Intille. Embedding behavior modification strategies into consumer electronic devices: A case study. In *Proceedings of the Ubiquitous Computing 8th International Conference*, pages 297–314, New York, 2006. Ubiquitous Computing Conference Series.

[4] Steven Oliver Paul Gaudet, Thomas Blackadar. Measuring foot contact time and foot loft time of a person in locomotion. United States Patent 6,018,705.

[5] Lars Erik Holmquist Lalya Gaye, Ramia Maz. Sonic city: The urban environment as a musical interface. In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03), Montreal, Canada*, pages 109–115, Montral, Canada, 2003. New Interfaces for Musical Expression.

[6] Michael Bull. No dead air! the ipod and the culture of mobile listening. *Leisure Studies*, 24(4):343–355, 2005.

[7] Günter Geiger. PDa: Real time signal processing and sound generation on handheld devices. In *Proceedings on the International Computer Music Conference*, 2003.

[8] David Topper and Peter Swendsen. Wireless dance control: Pair and wisear. In *Proceedings of the 2005 International Conference on New Interfaces for Musical Expression*, pages 76–79, 2005.

[9] Wayne Siegel and Jens Jacobsen. The challenges of interactive dance: An overview and case study. *Computer Music Journal*, 22(4):29–43, 1998.

[10] Matt Jones and Steve Jones. The music is the message. *interactions*, 13(4), 2006.

[11] J. L. Romkey. RFC 1055: Nonstandard for transmission of IP datagrams over serial lines: SLIP, June 1988. Status: STANDARD.

[12] `http://zwizwa.fartit.com/pd/pdp/`. Retrieved on 19 August 2007.

[13] `http://gem.iem.at/`. Retrieved on 19 August 2007.

[14] Miller Puckette. `http://puredata.info/dev/PdMessages`. Retrieved on 17 August 2007
Also available in the PureData online help (on unix) as /usr/lib/pd/doc/5.reference/list-help.pd.

[15] J. A. Paradiso, K. Hsiao, A. Y. Benbasat, and Z. Teegarden. Design and implementation of expressive footwear. *IBM Syst. J.*, 39(3-4):511–529, 2000.

[16] Gertjan Wijnalda, Steffen Pauws, Fabio Vignoli, and Heiner Stuckenschmidt. A personalized music system for motivation in sport performance. *IEEE Pervasive Computing*, 4(3):26–32, 2005.

[17] Ryan Aylward and Joseph A. Paradiso. A compact, high-speed, wearable sensor network for biomotion capture and interactive media. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 380–389, New York, NY, USA, 2007. ACM Press.

[18] `http://gumstix.com`. Retrieved on 24 May 2007.

[19] Joseph A. Paradiso, Stacy J. Morris, Ari Y. Benbasat, and Erik Asmussen. Interactive therapy with instrumented footwear. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1341–1343, New York, NY, USA, 2004. ACM Press.

[20] Ryan Aylward and Joseph A. Paradiso. Sensemble: a wireless, compact, multi-user sensor system for interactive dance. In *NIME '06: Proceedings of the 2006 conference on New interfaces for musical expression*, pages 134–139, Paris, France, France, 2006. IRCAM — Centre Pompidou.

[21] Shinichiro Toyoda. Sensillum: an improvisational approach to composition. In *NIME '07: Proceedings of the 7th international conference on New interfaces for musical expression*, pages 254–255, New York, NY, USA, 2007. ACM Press.