ZOINKS and Z.759 - The Unfinished Computer Experimental Control System

Raymond Meng Gao
Quantum Degenerate Gas Laboratory
Prof. Kirk W. Madison
Chem/Phys A015/023 UBC

Last Modified Date: August 31, 2005

Contents

Chapter 1

About This Report

This document tries to glue together the bits and pieces of the Computer Control Project that I have been assigned to during my 8 month co-op workterm here at the QDG lab. At this point, there are so many components of the project unfinished, untested, and undocumented, it is almost impossible to make this paper perfectly coherent. This is my fair warning to the reader.

`[What is ZOINKS?]`

Zee Open Interface Networked Kontrol Standard was proposed by Dr.Kirk Madison and Dr.Dan Steck as a series of open source, reusable, and scalable experimental control/data acquisition systems for the Atom Optics research community around the world.

`[Does it have a webpagge?]`

Yes. Although this might be a temporary location.
http://atomoptics.uoregon.edu/`zoinks/`

`[What is Z.759?]`

Z.759 is the name dubbed for the software suite being developed to control ZOINKS devices. Since I am leaving this project incomplete at the end of my workterm, I decided to name it after Schubert's Unfinished Symphony #8 D.759.

`[Doesn't naming the project after Schubert's famous composition make you sound pretentious and arrogant?]`

No, I find it to be in good humour.

Disambiguation of Terminology:

There are 2 words that are widely abused in this document: **Bus** and **Driver**. I will try to clarify them here.

In this document, whenever I refer to **UT Bus Driver**, I am referring to a piece of hardware that can write bit patterns onto the 50-pin UT bus, either **NiDAQ card** or the **Fast Bus Driver**.

Whenever I refer to **UT Device Drivers**, I am referring to the set of hardware can communicate on the UT bus, ie: the Analog Out, Digital Out, or the DDS.

When I refer to **UT Bus Devices**, I am referring to any device that can be driven by the **UT Device Drivers.**

Also a hardware **driver** is different from a software **driver**.

Acknowledgements:

My best regards to everyone in the QDG lab during my workterm: Paul, Swati, Tao, Peter, Aviv, Janelle.

Special thanks to Dr Kirk Madison and Dr Bruce Klappauf who not only offered me this great job but helped me time and time again with all my projects.

More thanks to all people in the UBC Electronic Shop Richard, Tom, Gar, Marcel, and Stan.

And Dr. Daniel Steck's group in Eugene, as well as Dr. Todd Meyrath and Dr. Florian Schrek in Austin who designed and built the UT Bus system and the original \textbf{Control.}

All the note icons are Microsoft ClipArt

Chapter 2

A Brief History of ZOINKS

If you believe that the universe began 13 billion years ago, earth formed 5 billion years ago, the dinosaurs became extinct 65 million years ago, the great pyramids built 2500 years ago, then on the same scale, project ZOINKS started about 2 years ago.

Taking the idea of connecting all devices to LAN, project ZOINKS revolved around a multipurpose, open source, microcontroller board, the Ethernut. An overwhelming number of ZOINKS devices were then proposed and currently being developed in a collaborate effort by Kirk Madison's group at UBC, Dan Steck's group at U of Oregon, and Mark Raizen's group at U of Texas, Austin.

Below is a list of some important projects and their status as of now. Not all of these are Ethernut devices, and the development of some of these predate ZOINKS.

Legend:

🟥	The project is in need of a lot of work (>6 months)
🟨	The project is in need of some work (<6 months)
🟩	The project is complete.

Ethernut Projects

- 🟩 **Ethernut-GPIB** - Allows GPIB instruments to be controlled from LAN.
- 🟨 **Ethernut-Serial** - Allows serial instruments to be controlled from LAN.
- 🟩 **Web Thermal Couple** - Take temperature readings on the web.
- 🟩 **Ethernut Analog Input Board** - Based on the web thermocouple design.
- 🟥 **Fast Bus Driver** - The replacement for the NiDAQ cards
- 🟥 **Rubidium Clock Controller** - Twiddle with the clock from LAN.

UT I/O Projects

🟨 **DDS for QDG** - the version that the UBC E-Shop is building

🟩 **DDS Version 2.0** - the version that Todd Meyrath built.

🟨 **RF Amplifier** - Amplifier for the DDS.

🟩 **Digital Out boards** - digital

🟩 **Analog Out boards** - analog

🟨 **New Control Software** - for the UT Bus devices

Chapter 3

The Texas Box

This anonymous box, built by Todd Meyrath and Florian Schreck at the University of Texas, Austin, contains arrays of digital and analog line drivers for general purpose experimental controls. All these devices in the box complies with a simple standard, the 50-pin parallel UT bus. The line drivers come in 3 flavours, the analog outputs, digital outputs and the DDS (Direct Digital Synthesizer). It was originally designed to be driven by the National Instruments DAQ cards for PCI, but adapted as a ZOINKS project and eventually will be driven by the Ethernut [Figure 3.1] All references to the UT Box, UT bus, and the UT devices are available on the UT control system's homepage. (Refer to Appendix) The following sections will make brief introductions to the UT 50-Pin Bus, and the Analog/Digital Out boards. The DDS will be discussed in the next chapter.

3.1 50-Pin Bus Interface

All devices on the bus must comply with the bus requirements, 16 data lines, 8 address lines and 1 strobe line. The data and address lines are synchronized to the strobe signal. That is, on every pulse of the strobe, data and address will be loaded onto the bus. Only the device whose address matches the address on the bus will latch the data from the data lines.

To represent the address on a UT bus device, a dip switch and a comparator is setup on board so that if and only if **the address pattern from the bus matches the dip switch pattern**, will the comparator produce a signal to allow data to be latched. [Figure 3.2] (Refer to UT bus manual for more details)

Figure 3.1: UT Box for ZOINKS

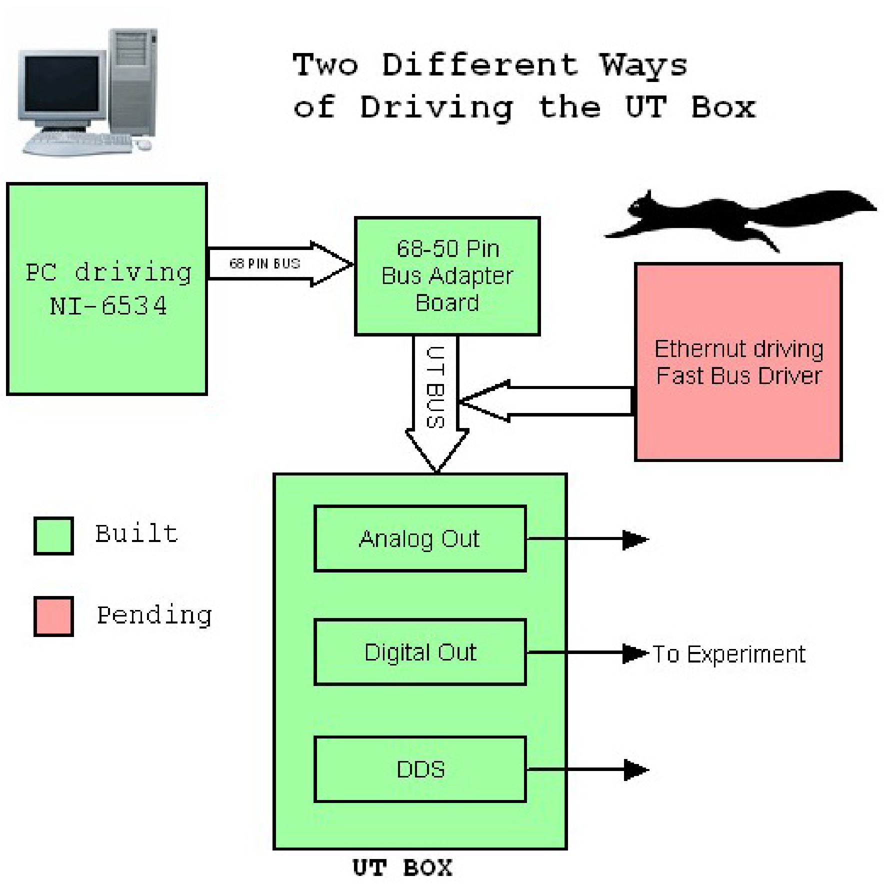
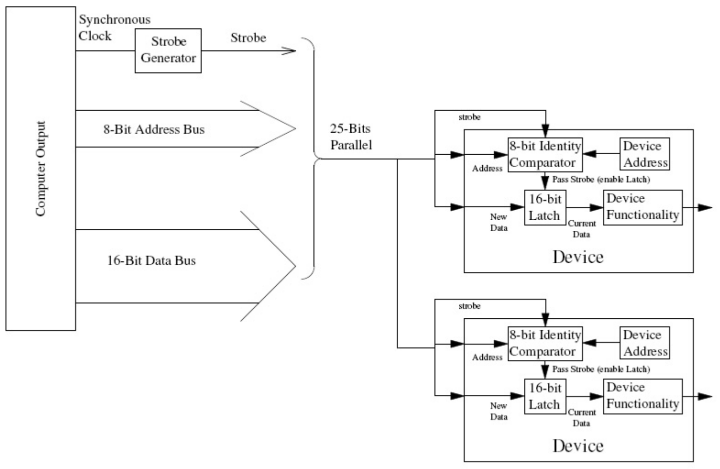


Figure 3.2: UT Bus Block Diagram from the UT Bus Manual



3.2 Analog Out

The analog out board uses two 16 bit Quad DAC (DAC7744) digital to analog converter chips to convert a 16-bit integer from the UT bus into 8 multiplexed analog outputs.

The upper 5 bits of address bus must match the value on the 5 onboard dip switches, and the lower 3 bits refer to which of the 8 DAC channels to drive. The DAC7744 is capable of 16 bit resolution with +/-1bit LSB (least significant bit) error. The board is designed to drive maximum of 50 ohm loads at max current of 250mA. (Refer to Analog Out board manual for more details)

3.3 Digital Out

The digital out board simply consists of 16 BNC outputs that latches and buffers the 16-bits of data coming from the data bus. Each line is capable of driving 50 ohm loads. (Refer to Digital Out board manual for more details)

3.4 UT Bus Drivers: Past and Future

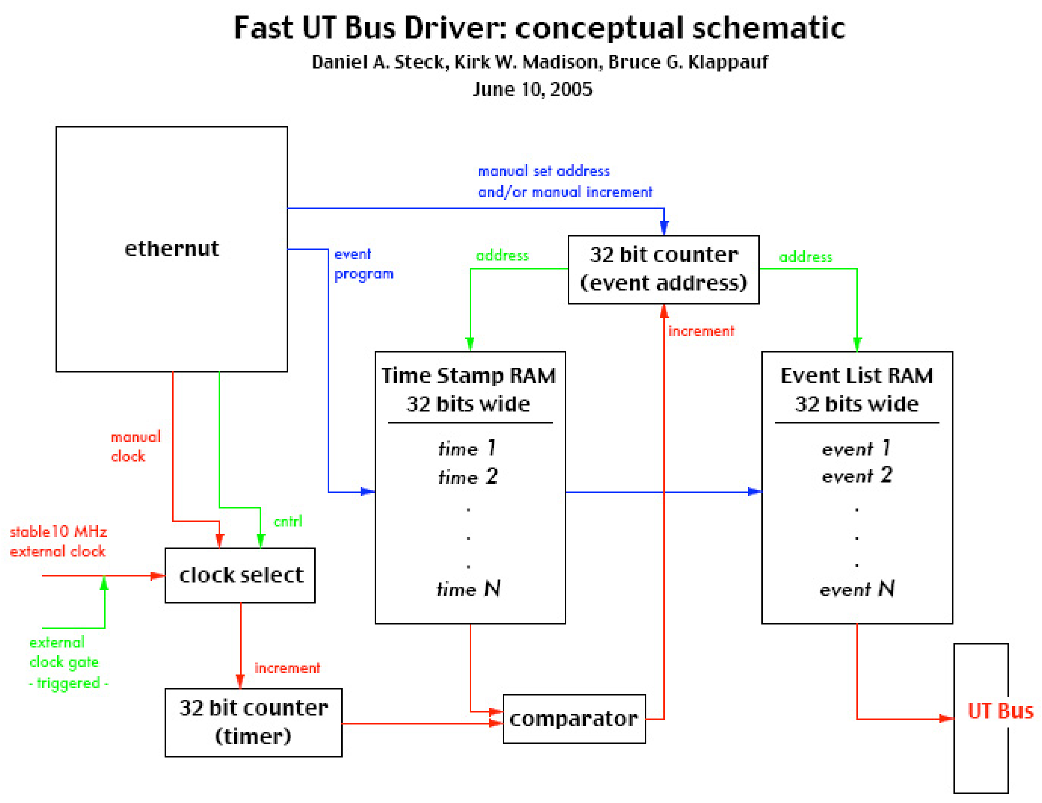
This section will describe the two bus drivers, the Ni653X card and the new Fast Bus Driver.

3.4.1 The Ni635X card

Up to now, the only UT Bus Driver we have used is the Ni653X card. National Instruments supplies a variety of expensive digital and analog output cards known for their high sampling rate. The NI6534 card features 32 general purpose DIO (TTL/CMOS) lines capable of outputting at a maximum of 20MS/s. One of its main features not available in its sister model the 6533 is its onboard 32Mb memory bank. This onboard memory allows the card to perform processor independent operations such as storing an entire waveform in its internal memory and looping it. These cards currently only have drivers for Windows.

Our goal is to build a new bus driver with more memory, operating system independent, and possibly faster, ie: the **Fast Bus Driver.**

Figure 3.3: The Fast Bus Driver



3.4.2 The Fast Bus Driver

The **Fast Bus Driver** [Figure 3.3] was proposed by Dan Steck in collaboration with Kirk Madison and Bruce Klappauf. The ingenuity behind the Fast Bus Driver is in the design of the timing system. It uses one block of memory to store instructions and another block to store a 32-bit time constant associated with that instruction. A system counter will count endlessly from 1 to 2^{32}-1. The value of the counter will be compared to the value of the time constant and when they match, the associated instruction will be sent onto the bus. The advantage of this is that it allows events separated by different orders of magnitude in time units to be controlled in the same system with exactly the same precision. A simple calculation can show that using 100 megabytes of RAM for the timing chip and an atomic clock, one could retain nanosecond precision controlled events for as long as 3 years!. The FBD will be controlled through the Ethernut and has wandered onto ZOINKS' massive projects list.

The **Fast Bus Driver** is currently being developed by the UBC Electronic Shop. Unfortunately, it will most likely be 6 months to a year before the first model is built, tested, and programmed.

Chapter 4

DDS

The DDS board, designed by Todd Meyrath, is based on the AD9852 chip from **Analog Devices**. It is capable of generating sin waves from DC - 150Mhz, with programmable frequency,amplitude,and phase control. It also has built in operational modes for common experimental procedures such as frequency ramps, shape keying, and chirping. In Todd's design, the outputs pass through 135MHz low-pass ecliptic filters. The filter eliminates unwanted high frequencies above 135MHz.

There is an old board, version 1.0 which I will describe in more detail here, and **TWO** new boards, one being built by the UBC electronic shop and the other already built by Todd Meyrath's group at UT. Both are based on the testing and debug feedback done here in Kirk Madison's lab. Thus the future for the DDS at QDG is not certain at this point.

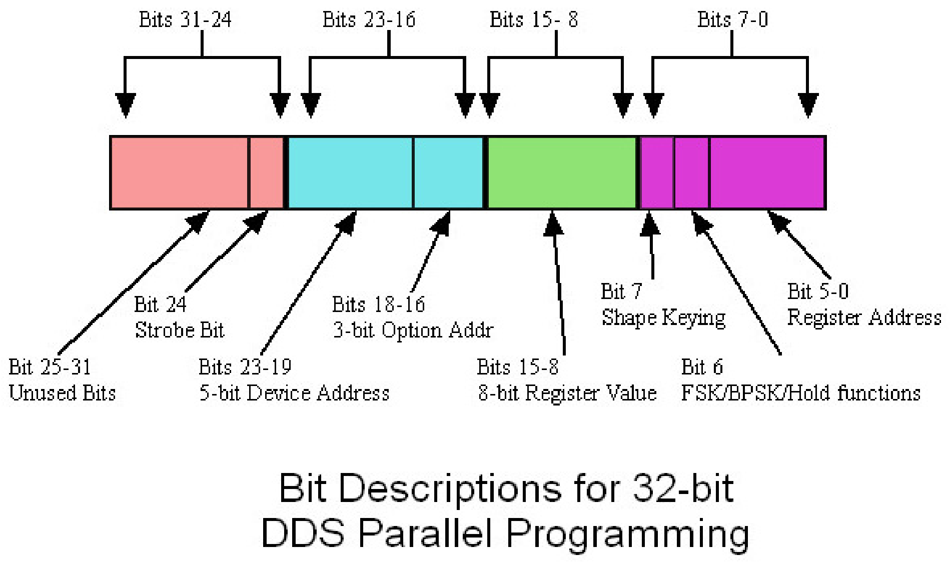
4.1 Programming the DDS

`References: AD9852 Manual, Meyrath's DDS Manual`

The programming information provided here refers to the old DDS board.

Since the DDS is built for the UT bus, programming it requires writing 25-bit instructions to the bus. I used the NI-DAQ card to drive the DDS so the easiest method was to use "Pattern I/O" mode (See the NI-653X DAQ manual) to write 32-bit data patterns onto the UT bus [Figure 4.1].

Figure 4.1: DDS Parallel Programming



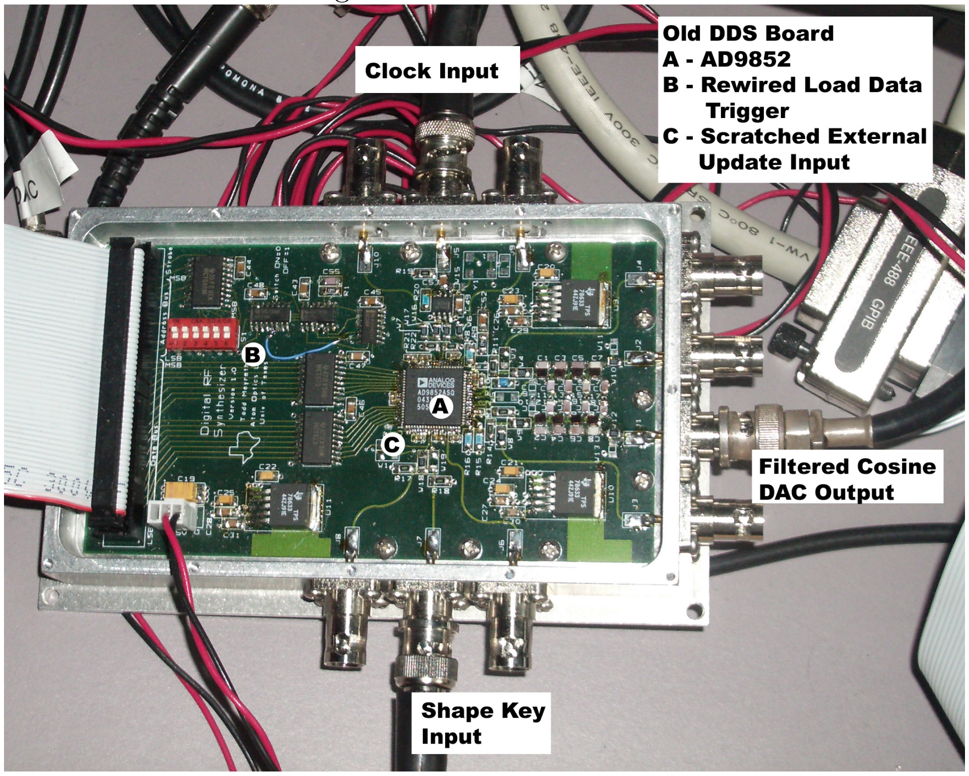
Sample programs written using the NI-DAQ card to test the DDS are explained in the Software sections.

Programming the DDS with instructions requires two main steps:

```
Write - Write the value to the AD9852's onboard buffer
Update - Move the data from the buffers to the AD9852's internal
registers.
```

Only when the data is written into the internal registers will we see effect of our program instruction. There are two ways to "update" the DDS, internal or external. In the default method, internal update, the DDS will simply move the data from its buffers to the registers every constant number of System Clock cycles. The constant is stored in a set of the AD9852's registers and can be modified by the user. When the system updates, it will send 8 pulses through the **I/O Update** pin on the AD9852 to inform the user that change has taken effect.
In external update mode, the user can control exactly when a change should take effect by sending a signal through the **I/O Update** pin on the AD9852. This pin is routed to the "Update option" on the

Figure 4.2: Old DDS Box


Clock Input
Old DDS Board
A - AD9852
B - Rewired Load Data Trigger
C - Scratched External Update Input
Filtered Cosine DAC Output
Shape Key Input
A
B
C

DDS board, which can be addressed by writing a 2 to the 3-bit options address in our instructions. [Figure 4.1]

The AD9852 chip has shape keying enabled by default. This means, if we don't disable it in our code or give it a proper shape keying signal, we will NEVER see a trace from the outputs. This was painstaking to debug.

4.2 Debugging

The first version of the DDS shown in [Figure 4.2] has several bugs which were very difficult to debug. I will document them here for future references as similar debugging procedures may be needed.

4.2.1 The Delay Line

The first serious problem was not a problem in the design but a blunder in building the board. Nevertheless, it was a gritty bug to find.

`Symptoms/Potential Problems:`

The Master Reset does not work properly. Later investigations reveal that it needs to be held for an excessive period of time before the board resets.

`Solution:`

After measuring some timing diagrams, we discovered that the 100ns delay line immediately after the comparator was not delaying the signal for 100ns but 100ms. In fact, the capacitor in the delay circuit turned out to be 1000 times the required value. Since tiny surface mount capacitors don't have actual capacitance values printed on them, it is usually impossible to measure its capacitance in a circuit. In this case, the RC constant could be measured.

This bug was a complete shock since the "load data" pin was also connected to the delay line and the NI-DAQ card surely could have sent the programming instructions to the DDS 2000 times over before the "load data" signal arrived even once. How were we able to program the DDS at all? The following bug miraculously complemented this problem.

4.2.2 The Load Data Pin

The following problem caused countless hours of headaches.

`Symptoms/Potential Problems:`

The output frequency does not lock to the reference clock. Certain sequences of instructions are not loaded. Some instructions are written to incorrect addresses of the AD9852 registers.

`Solution:`

At first it looked like no matter how we programmed it, whenever the DDS was placed in multiplier mode (where the system clock reference is multiplied internally by an integer from 4-20), the output frequency would fail to the lock to the reference frequency. It would generate a signal independent of the system clock frequency and slowly decrease.

This seemed like a possible bizarre malfunction of the AD9852's internal Phase-Lock Loop so we tried to replace the AD9852 chip. This wasn't an easy feat since the back of the chip was a giant heatsink and the only way to de-solder it was to either use a giant soldering iron and prod the chip out from the back or bake it out in an oven. The iron prod method requires clipping the IC's legs while baking risks solder spills on the rest of the connections on the board. Our electronic shop did this and they chose the former method.

Replacing the $40 chip was a bust as the same problem persisted. We went back to the drawing board and decided to experiment with the programming and double check for software errors. This gave us the first logical clue. We noticed that changing the order in which some instructions were entered allowed the DDS to function properly. The problem eventually was narrowed down to the fact that certain bits bus were conditionally over-writing values in the internal register. This was very odd as we tried to deduce functions for bit patterns that caused the haywire. However, no simple and consistent logic error could be concluded. Thus, we decided to logic probe the signals coming from the lines on the latch and their values on every possible junction and time. Everything agreed.

During the aforementioned logic tests, we sent the instructions one at a time and noticed a curious quirk. The DDS was receiving the program instructions regardless of toggling the "Load data" pin. This lead to the miraculous observation that the pin was vaguely hinted as **active low**, while the DDS board was designed to trigger it **active high.** Problem solved. We simply bypassed an inverter with a wire. Label B on [Figure 4.2].

4.2.3 The Update Pin

This problem was fixed before it could do any harm.

`Symptoms/Potential Problems:`

AD9852 or AND gate chip damaged.

`Solution:`

The I/O update pin, as described earlier, can be configured as both input and output. During internal update (default), the AD9852 will send out short pulses through the pin which is connected to the output of the AND gate. If this pin on the AND gate is low an any instance the I/O update pin is high, the 3.3V pulse will be shorted to ground and possibly damaging both chips.
To fix this, we simply scratched off the connection.[Figure 4.2] The new DDS versions will have fixed this problem by placing inline protection diodes.

4.3 RF Amplifiers

The for a chip that operates on 5 watts, the AD9852's output power is abysmal. Selecting a power amplifier for the DDS is still being debated. The most common application for the DDS in cold atom experiments is to drive AOMs (Acousto-optic modulator). Typical input power requirement is about 2 watt. Todd Meyrath and his experiments in Austin uses a 4 watt amplifier from Mini-Circuits in combination with attenuators while Dan Steck's group is supposedly building their own custom amplifier for project ZOINKS. If they are successful, we will most likely inherit their home-built amplifier for our experiments as they are significantly cheaper.

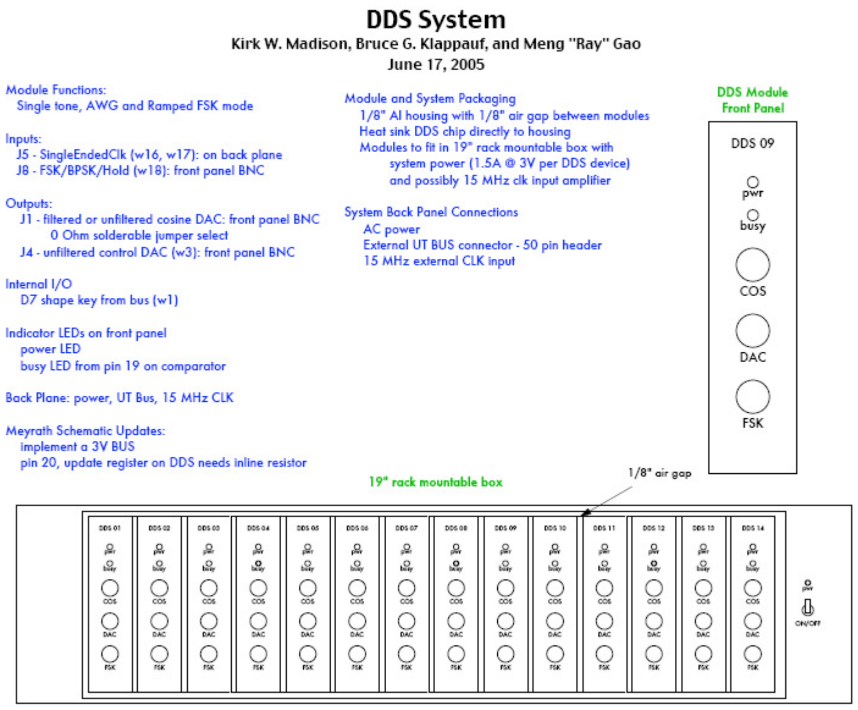
Another variable to consider is that the DDS's output power is so low (¡¡ 0dBm) that it doesn't even satisfy the input power ratings on most power amplifiers. Thus a pre-amplifier is needed. In the new DDS design by the UBC Electronic Shop, this pre-amp will be built onto the board.

A Mini Circuits 2-4 watt amplifier typically costs $500 USD.

4.4 The new DDS boards

The new and improved DDS boards will feature all the fixes to the bugs mentioned above. Todd Meyrath have already ”mass produced”

Figure 4.3: The New DDS System for QDG


DDS System

Kirk W. Madison, Bruce G. Klappauf, and Meng "Ray" Gao

June 17, 2005

Module Functions:
Single tone, AWG and Ramped FSK mode

Inputs:
J5 - SingleEndedClk (w16, w17): on back plane
J8 - FSK/BPSK/Hold (w18): front panel BNC

Outputs:
J1 - filtered or unfiltered cosine DAC: front panel BNC
0 Ohm solderable jumper select
J4 - unfiltered control DAC (w3): front panel BNC

Internal I/O
D7 shape key from bus (w1)

Indicator LEDs on front panel
power LED
busy LED from pin 19 on comparator

Back Plane: power, UT Bus, 15 MHz CLK

Meyrath Schematic Updates:
implement a 3V BUS
pin 20, update register on DDS needs inline resistor

Module and System Packaging
1/8" Al housing with 1/8" air gap between modules
Heat sink DDS chip directly to housing
Modules to fit in 19" rack mountable box with
system power (1.5A @ 3V per DDS device)
and possibly 15 MHz clk input amplifier

System Back Panel Connections
AC power
External UT BUS connector - 50 pin header
15 MHz external CLK input

the version 2.0 DDS boards. However, our UBC electronic shop is building a more complicated version. The notable difference is that it will replace the digital addressing logic ICs with a **CPLD**. The exact specifications are unknown as no design schematic have been produced yet. We have asked that they be at least compatible to two main things:

What is **CPLD?** Complex Programmable Logic Device allows digital logic to be programmed and modified at will. It is basically a giant set of primitive logic gates such as AND and OR

- The UT Bus System
- The new DDS rack mount shown in [Figure 4.3] (Also being designed by the electronic shop)

The rack mount box will allow up to 14 new DDS boards to be lined up in one enclosure and a set of power amplifiers in a separate box beside it. With each DDS board running at 5 watts and each amplifier at 70 watts, the DDS System will be the alpha energy monger on the block.

Chapter 5

The Control Software

To coordinate and control all the ZOINKS devices and run an experiment, some kind of software is required. Since cold atom experiments typically require large collections of linear instructions and very little feedback until the end, "control software" have been traditionally hard-coded using procedural languages such as C or Basic for specific experiments. However, as experiments become more complicated, implementing an object-oriented software structure may prove to be a more scalable and reusable solution.

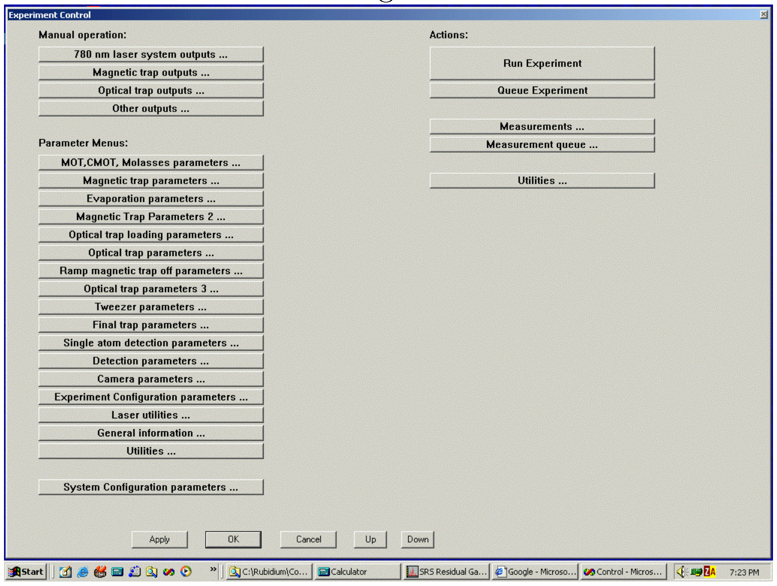
5.1 UT Control

The control software being used in Mark Raizen's lab at UT Austin was written by Florian Schreck. [Figure 5.1] The basic idea of this program was as follows:

1. Code experimental procedures
2. Enter parameters
3. Run experiment
4. Acquire data

Control, coupled with **Vision** (a data acquisition and analysis program also written by Florian) worked remarkably for the experiments in Austin. However, the feasibility of using Florian's program for the experiments at QDG have been debated countless number of times.

Figure 5.1: Florian's Control Program Screenshot from his webpage


Experiment Control
Manual operation:
780 nm laser system outputs ...
Magnetic trap outputs ...
Optical trap outputs ...
Other outputs ...
Parameter Menus:
MOT,CMOT, Molasses parameters ...
Magnetic trap parameters ...
Evaporation parameters ...
Magnetic Trap Parameters 2 ...
Optical trap loading parameters ...
Optical trap parameters ...
Ramp magnetic trap off parameters ...
Optical trap parameters 3 ...
Tweezer parameters ...
Final trap parameters ...
Single atom detection parameters ...
Detection parameters ...
Camera parameters ...
Experiment Configuration parameters ...
Laser utilities ...
General information ...
Utilities ...
System Configuration parameters ...
Actions:
Run Experiment
Queue Experiment
Measurements ...
Measurement queue ...
Utilities ...
Apply
OK
Cancel
Up
Down
Start
C:\Rubidium\Co...
Calculator
SRS Residual Ga...
Google - Microso...
Control - Micros...
7:23 PM

Here is a summary of the controversy:

`Pros:`

- **Control** is already written and can save time in the short term.
- **Control** is designed to run with many instruments common to cold atom experiments.

`Cons:`

- **Control** is built in Visual C++ and we want to eventually migrate to Linux.
- **Control** requires writing new experimental procedures into the code itself and requires to be recompiled everytime something changes.
- **Control** isn't properly documented, modifying it requires cloning Florian and have him work here as a robot.

A proposed plan for a new control software system to address the shortcomings of **Control** is discussed in the next chapter.

Chapter 6

Z.759

This project was developed with the DevC++/MingW environment allowing the programs to be ported to other operating systems without too much problems.

6.1 Architecture

The goal of **Z.759** is to establish a scalable framework for creating experimental control software. Part of the code I have already done, including a proof of concept UT bus test utility. However, there is still some debatable areas that needs to be thought through carefully before implementing. Thus like so many mentioned here, the future of this project is uncertain.
The main design aim of Z.759 revolves around several major concepts:

- All experimental sequences are written in XML
- There are 2 types of software device drivers. One type represent UT bus devices and do not actually drive any hardware but only produces binary/hex bit patterns that can be sent to the UT bus. The other type, such as the driver for the NIDAQ card, are drivers in the classical sense where they interact with hardware.
- The kernel code is only used for driving UT bus devices.

[Figure 6.2] shows a diagram of the program flow for Z.759.

A The main program will start by parsing the input XML documents specifying the experiment sequence (which represent functions and parameters) and the hardware configurations (devices and addresses).

Figure 6.1: Z.759 Kernel UML Diagram

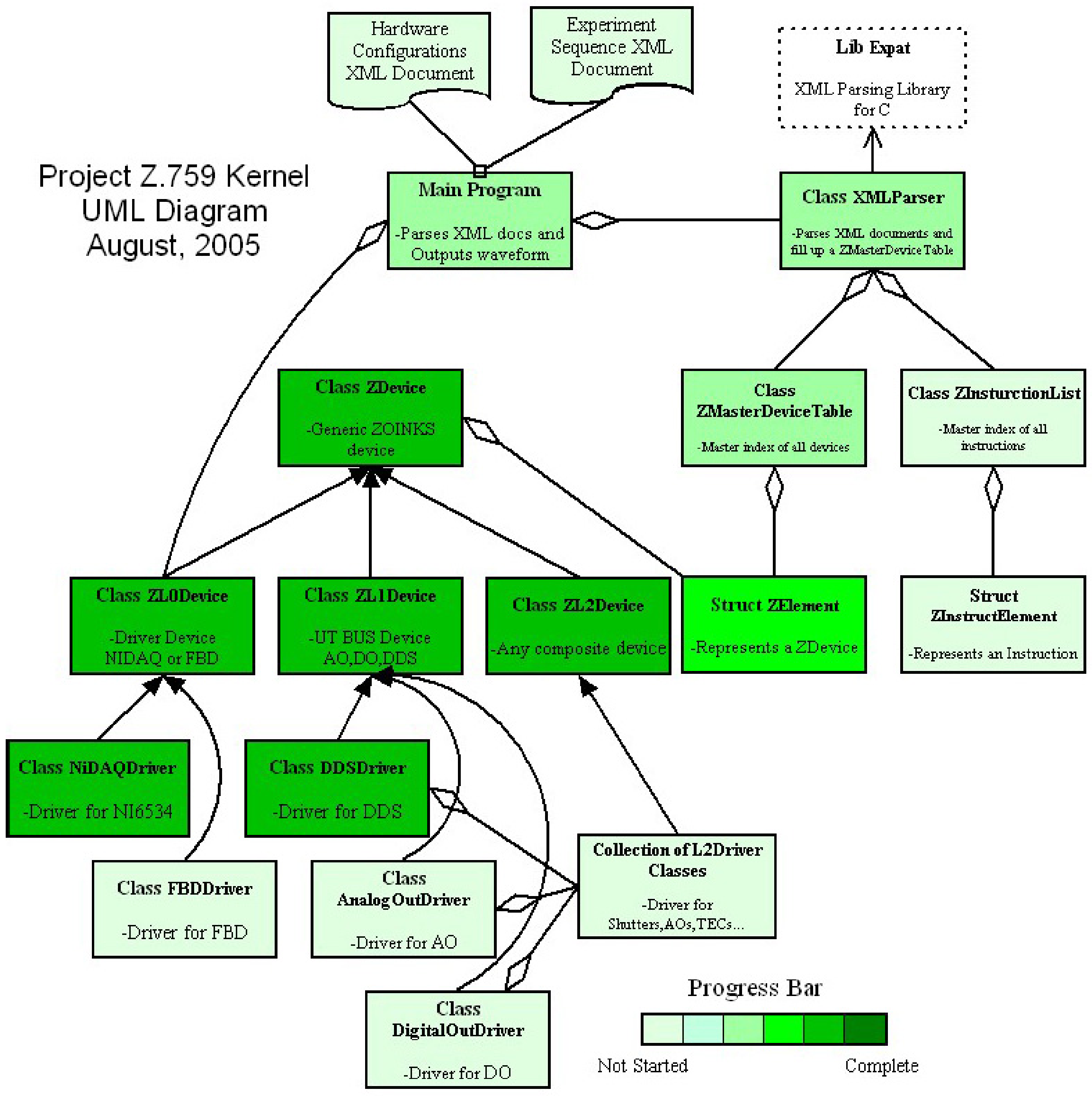
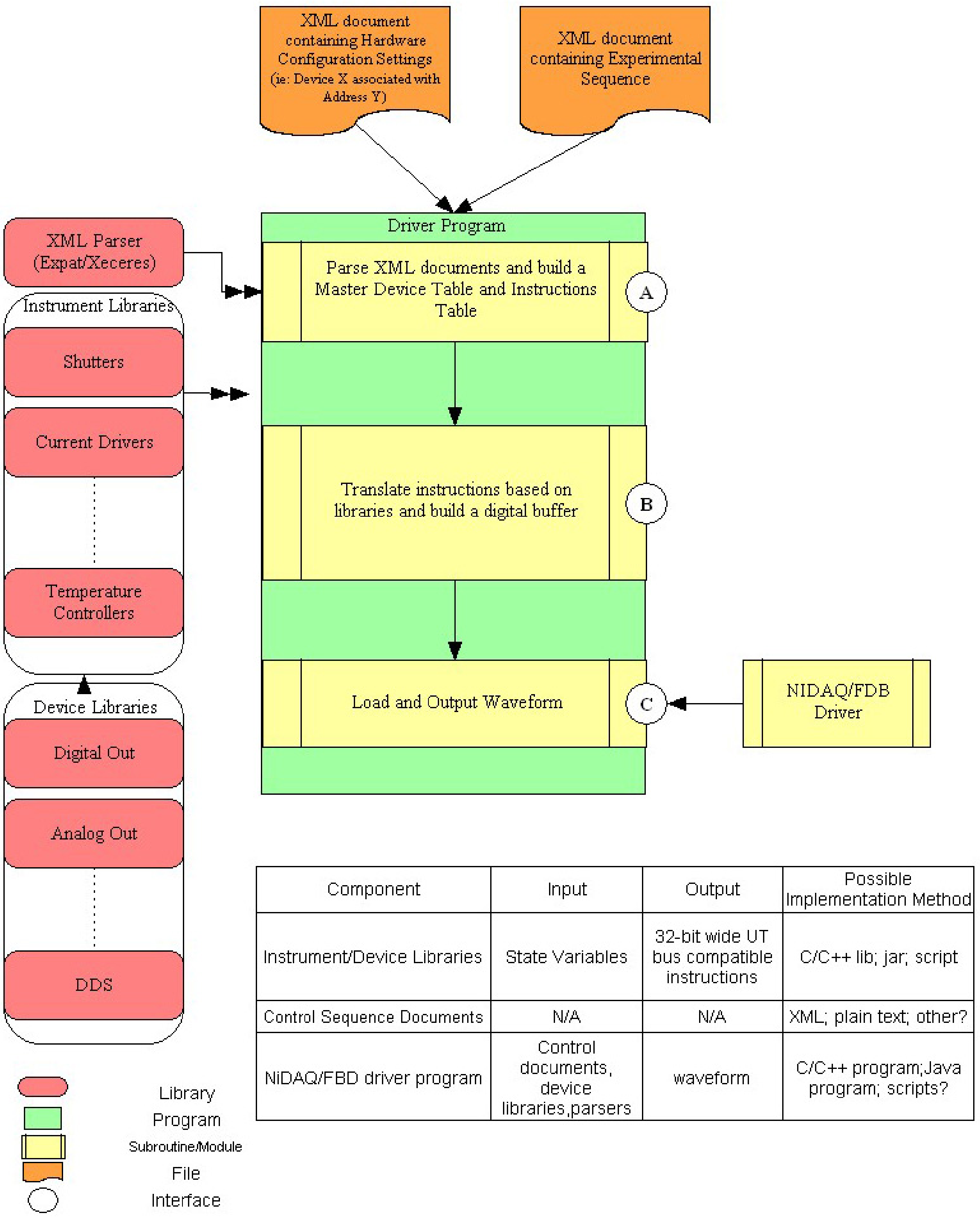


Figure 6.2: Z.759 Software Architecture


XML document containing Hardware Configuration Settings (ie: Device X associated with Address Y)
XML document containing Experimental Sequence
Driver Program
Parse XML documents and build a Master Device Table and Instructions Table
A
Translate instructions based on libraries and build a digital buffer
B
Load and Output Waveform
C
NIDAQ/FDB Driver
XML Parser (Expat/Xeceres)
Instrument Libraries
Shutters
Current Drivers
Temperature Controllers
Device Libraries
Digital Out
Analog Out
DDS
Library
Program
Subroutine/Module
File
Interface

Component	Input	Output	Possible Implementation Method
Instrument/Device Libraries	State Variables	32-bit wide UT bus compatible instructions	C/C++ lib; jar; script
Control Sequence Documents	N/A	N/A	XML; plain text; other?
NiDAQ/FBD driver program	Control documents, device libraries,parsers	waveform	C/C++ program;Java program; scripts?

B Once the parsing is complete, the program will have a master list of all the devices in the system and instructions the user wishes to execute. Based on the device driver libraries, the program can then translate the instructions into a pure binary form that can be sent directly to the UT bus as a digital wave pattern.

C Finally, the program can choose to physically output the pattern using an appropriate UT bus driver (NIDAQ or FBD).

6.1.1 XML

What is XML? The Extensible MarkUp Language is very similar to HTML but allows for custom tags. Thus XML can be viewed as a **language-building language**.

XML has caught on during the recent years as a very popular method for building data storage models and custom markup languages. XML's main advantage to this project its readability to both human and machine. [Figure 6.3]

- human interpretability is trivial and can be enhanced by writing GUIs to produce a visual representation of the content.
- machine interpretability has been made simple by many publicly available XML parsers such as **Expat** or **Xceres**.

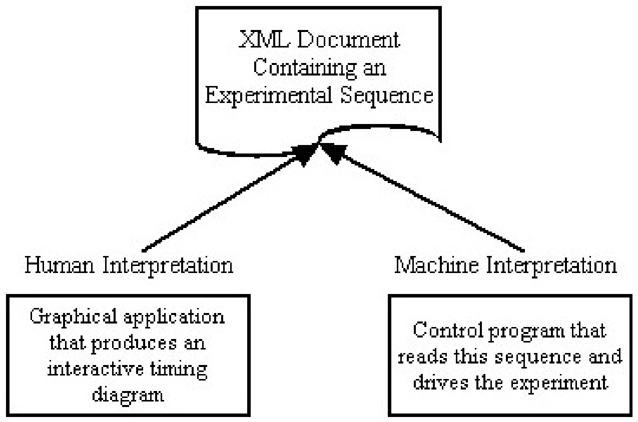
Nevertheless, we still need to design a set of rules and tags in this new language. Lets call it the **Experimental Sequencing Language, or ESL**. There is still considerable work to be done on building this pseudo language. However, below demonstrates the starting point. Here is a block of ESL code that will declare all the devices in the system.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<HCXML>

<L0DEVICE DID="0" TYPE="NICARD"  NAME="NICARD0" ADDR="0"
OSPEED="20MHZ">

    <L1DEVICE DID="1" TYPE="DO" NAME="DOBOARD0" ADDR="3"/>
    <L1DEVICE DID="2" TYPE="DO" NAME="DOBOARD1" ADDR="4"/>
```

Figure 6.3: Human Vs. Machine on XML


XML Document Containing an Experimental Sequence
Human Interpretation
Machine Interpretation
Graphical application that produces an interactive timing diagram
Control program that reads this sequence and drives the experiment

```
</L0DEVICE>

<L0DEVICE DID="3" TYPE="FBD" NAME="FBD0" ADDR="1">
    <L1DEVICE DID="4" TYPE="AO" NAME="AOBOARD0" ADDR="2"/>
    <L1DEVICE DID="5" TYPE="DDS" NAME="DDSBOARD0" ADDR="1"/>
</L0DEVICE>


<L2DEVICE DID="6" TYPE="SHUTTER" NAME="SHUTTER1" USES="1"/>
<L2DEVICE DID="7" TYPE="NEWDEVICE" NAME="NEWDEVICE1" USES="4 5"/>

</HCXML>
```

Eventually, all possible tags and parameters will have to be formalized. But currently, anything shown in the sample here is prone to change as new features are implemented.

6.1.2 the Master Tables

During the parsing, each valid device element, "**ZElement**" encountered is stored in a "Master Device Table"; each valid instruction, "**ZInstructElement**", is stored in a "Master Instructions Table". Instantiation of the device objects will be done when the associated

"**ZElement**" is created.

A ZElement looks like this:

```
typedef struct ZElement{

    DEVCLASS theclass;
    string name;
    int32_t ID;
    int32_t addr;
    string type;
    void* obj;
};
```

The **ZMasterDeviceTable** and **ZMasterInstructionList** classes [Figure 6.1] are **singleton** classes, which means only one static instance of each can exist in a program. Both tables will use a map structure to associate each element by a key. Unless this software will be implemented for an overwhelming number of devices, searching optimizations are not taken into consideration.

What is **singleton**? Singleton is a design pattern in software engineering that forces users to only be able to create the instance once through a static function and making the constructor private. Since this type of class occurs so common, it has been given a generic name.

6.1.3 the Device Drivers

The drivers are represented in a 3 layer hierarchy [Figure 6.1]. A **"ZDevice"** is at the top, representing any device that can exist on or be controlled by the UT bus.

These devices are divided into 3 sub classes:

- L0 Device - a bus driver, NIDAQ card or FBD
- L1 Device - a UT bus device, Analog/Digital Out or DDS

- L2 Device - a device that is driven by any combination of L1 Devices.

It should be emphasized again that the "Device Drivers" in Z.759 **DO NOT** necessarily mean they interact with hardware. The drivers for L1 and L2 devices will produce 32-bit wide UT bus instructions while L0 device drivers actually interface with hardware.

Here is how the program flow looks from a device point of view. For simplicity, lets assume there is only 1 L0 device in the system:

1. the L0 device is created
2. a set of L1 and L2 devices are created and associated with the L0 device.
3. the parser encounters a set of instructions and ask the L1 and L2 device drivers to generate the corresponding UT bus instructions (integers).
4. the main program asks the L0 device to output these instructions to the UT bus.

Since each L0 device physically drives a collection of L1 and L2 devices, each set of L1 and L2 devices must be associated with an L0 device. This association must be made very clear in the Master Device Table as well.

6.2 Problems

This section discusses some of the unresolved problems encountered while implementing this project.

6.2.1 State Dependence

Some functions in an experimental sequence may be state dependent. Ie: they require information about the previous state of the device. Common examples of the state dependent function include increment/decrement a value, toggle a switch etc.

Here even the digital out boards physically require state information to drive individual lines since all 16 lines are modified together by

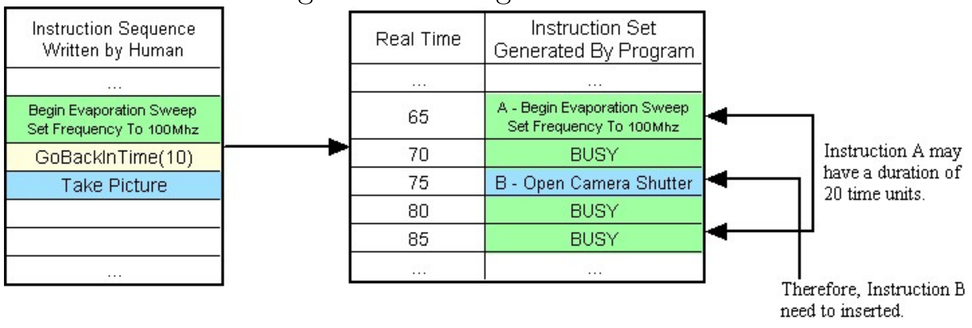
1 instruction.

One logical method of implementing state dependent functions is to write state variables into each device driver. The variables are changed everytime to reflect the changes in state. Also the initial state of the devices can be stored in an XML block somewhere. However, this implementation must be considered very carefully because it effects another intricate detail, the **GoBackInTime** function.

6.2.2 GoBackInTime?

Upon first hearing of the **GoBackInTime** function, one might be left dazzled and confused. This section will try to clarify the mystery its significance in this project.

Figure 6.4: Going back in time

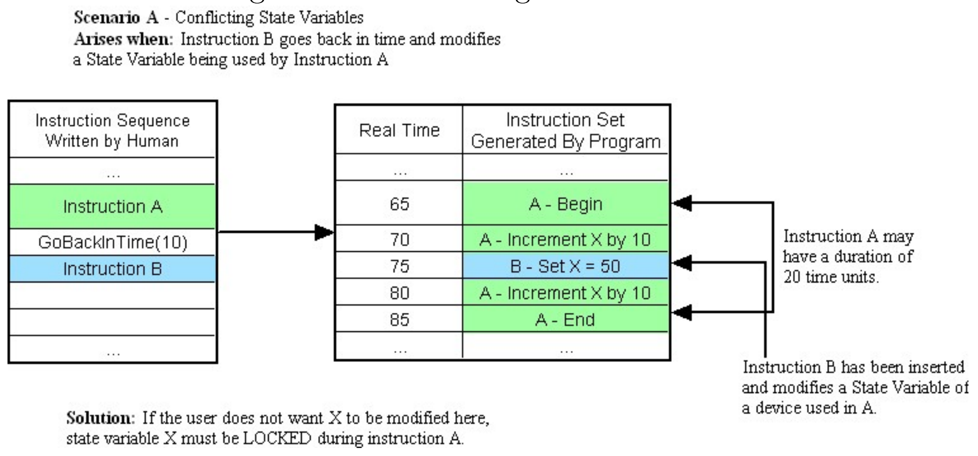

Instruction Sequence Written by Human
...
Begin Evaporation Sweep Set Frequency To 100Mhz
GoBackInTime(10)
Take Picture
...
Real Time
Instruction Set Generated By Program
...
65
A - Begin Evaporation Sweep Set Frequency To 100Mhz
70
BUSY
75
B - Open Camera Shutter
80
BUSY
85
BUSY
Instruction A may have a duration of 20 time units.
Therefore, Instruction B need to inserted.

GoBackInTime was coined by Florian Schrek in writing his control software. In writing control sequences, it is often the case that we might need to specify an event to occur x time units **BEFORE** another event. For example, if we have a camera and there is a known mechanical delay between when the shutter is triggered and when the shutter actually opens, we need to compensate for that by "going back in time" and insert the open shutter instruction.[Figure 6.4]

Some ambiguities arise when we play with time and space, even in a program. Consider the following situations:

It hasn't been decided yet how **GoBackInTime** will be implemented but the solution will definitely need to handle these cases.

Figure 6.5: Conflicting State Variables



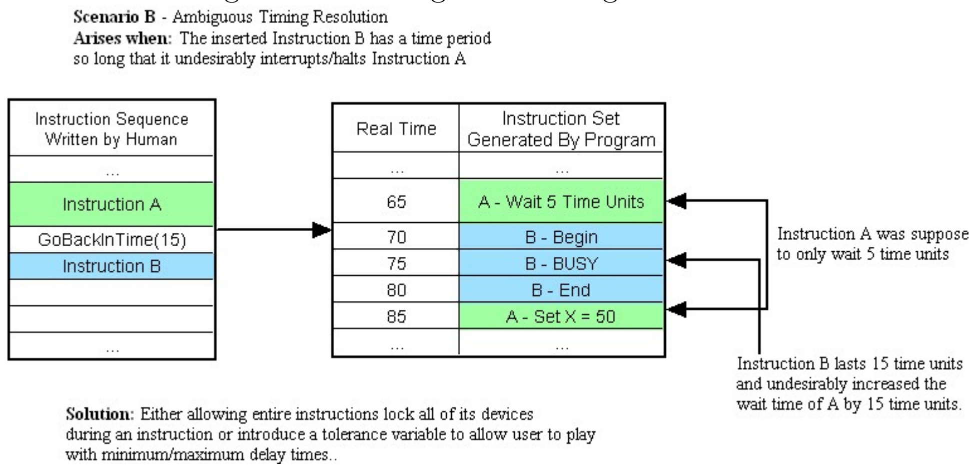
Scenario A - Conflicting State Variables
Arises when: Instruction B goes back in time and modifies a State Variable being used by Instruction A

Instruction A may have a duration of 20 time units.

Instruction B has been inserted and modifies a State Variable of a device used in A.

Solution: If the user does not want X to be modified here, state variable X must be LOCKED during instruction A.

Figure 6.6: Ambiguous Timing Resolution



Scenario B - Ambiguous Timing Resolution
Arises when: The inserted Instruction B has a time period so long that it undesirably interrupts/halts Instruction A

Instruction A was suppose to only wait 5 time units

Instruction B lasts 15 time units and undesirably increased the wait time of A by 15 time units.

Solution: Either allowing entire instructions lock all of its devices during an instruction or introduce a tolerance variable to allow user to play with minimum/maximum delay times..

6.2.3 L2 Drivers

The L2 drivers are meant to be representations of devices being driven by the UT bus L1 devices. It should be designed so that users can easily modify and/or add new drivers with ease. Therefore, their implementation should be simple and modularized.

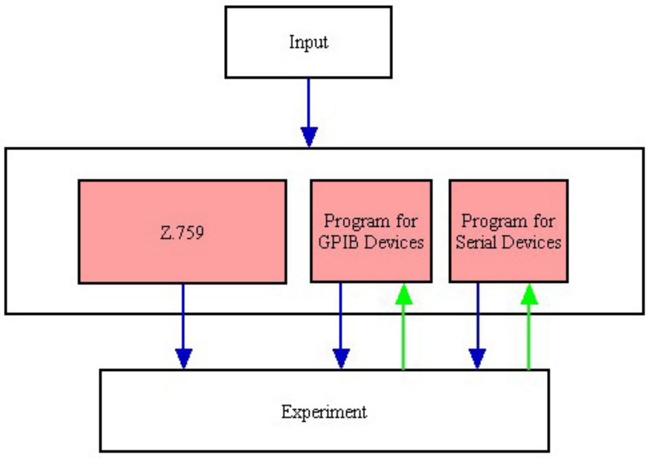
Thus it is questionable whether writing L2 drivers as classes in C++ in a fashion similar to L0 and L1 drivers is feasible. If so, a template for these classes must be made as clean as possible.

We could also consider a different implementation: allowing the user to define L2 devices and functions in XML. This involves formalizing another set of rules for **ESL** and adds another layer of complexity to the parser.

6.2.4 Implementing Non-UT bus devices

One might ask how the other experimental control devices fit in with Z.759. **The answer is they don't.** Z.759 is designed to be a module that can be easily incorporated into other programs.[Figure 6.7] It is up to the user to decide whether a master control program should be written to encompass all the experimental controls, such as Florian's **Control**.

Figure 6.7: Overall Control Setup



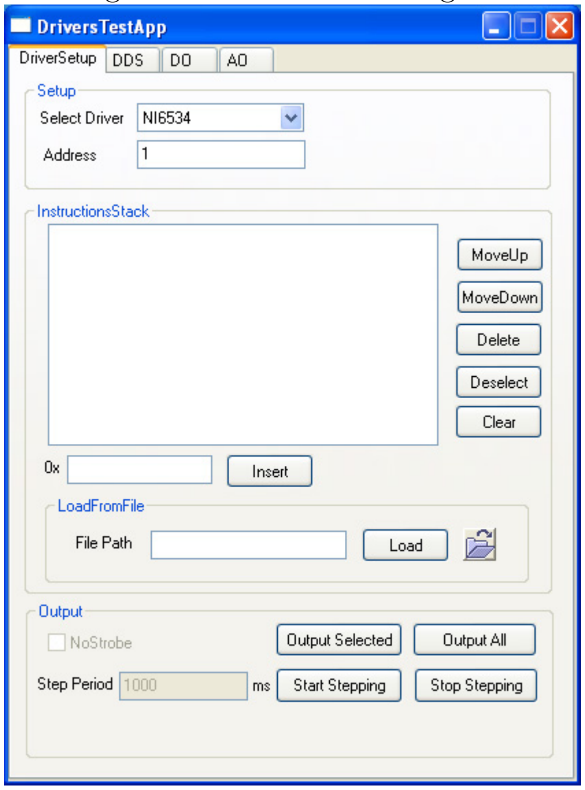
What if we want to time an event of a non-UT bus device to Z.759? **Use a digital trigger.** Most GPIB or serial lab instruments will have trigger inputs to allow for an event to occur at precisly the moment it receives the trigger signal.
The timing mechanism of the UT bus driver is by far superior to other protocols such as GPIB or serial. Don't even consider software timing. There is no guaranteed consistency nor precision.

6.3 Demo Utility

A working demo utility [Figure 6.8] based on Z.759 has been written to:

- Demonstrate some of the concepts in Z.759

Figure 6.8: The Demo Program



Select Driver NI6534

Address 1

Step Period 1000 ms

- Serve as a testing utility for UT Bus devices

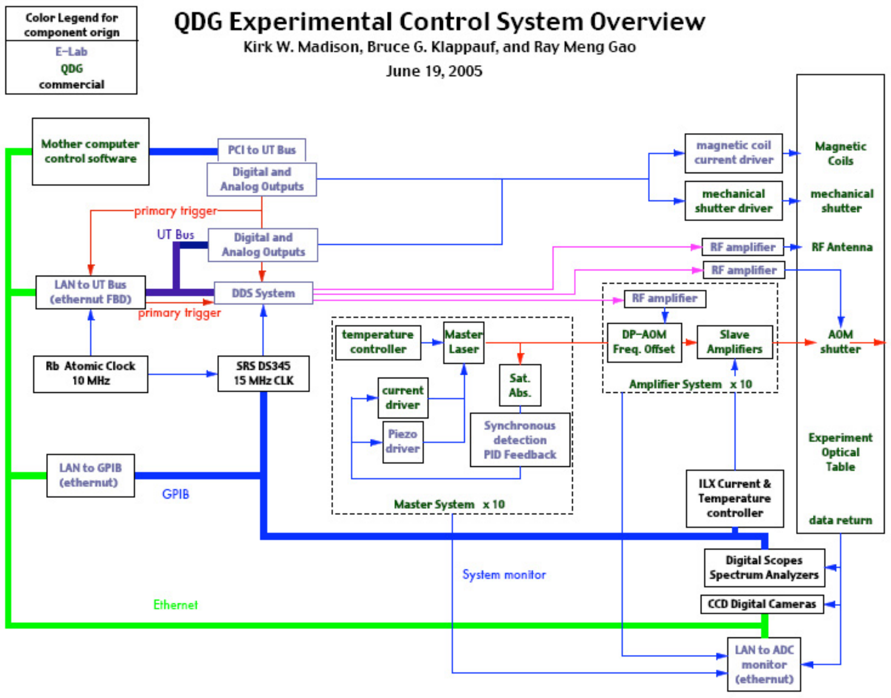
The demo program uses the L0 and L1 drivers to demonstrate the basic output functions of each UT bus device. For example, to output a sin wave with the DDS, the user can:

- Set the parameters such frequency,clock,mode etc.
- Produce the corresponding output buffer with the DDS driver. (UT Bus instructions)
- Modify the instructions at will. (If the user knows what they mean)
- Output them with the NiDAQ driver.

Chapter 7

QDG and ZOINKS: The Big Picture

Figure 7.1: Overview of QDG Experimental Control System



[Figure 7.1] is an extremely useful diagram drawn up by Kirk showing the entire experimental setup goals at QDG. Evidently, one can also see the number of ZOINKS projects in this picture and their significance to the QDG experiments.

Appendix A

List of References

All the references here are either hyperlinks to webpages or the **Refs** directory under the directory of this manual.

Home of Control All the manuals for the UT device drivers and Control are available here. But I've put copies in the **Refs** directory under the directory of this manual.
`http://george.ph.utexas.edu/~control/`

Home of QDG Control The control projects being developed at QDG.
`http://www.phas.ubc.ca/~qdg/ControlSystem/`

Home of ZOINKS The project list for ZOINKS. `http://atomoptics.uoregon.edu/~zoinks/`

Home of Dan Steck's Control project The project list Dan's lab.
`http://www.steck.us/control/`

AD9852 Manual The manual for the AD9852 chip used in the DDS.

AD9852 Manual The National Instruments manual for the Ni653X series cards.

Ethernut-GPIB Manual The Ethernut-GPIB manual.

Bibliography

[Control] Todd Meyrath, Florian Schreck, *A Laboratory Control System for Cold Atom Experiments*, http://george.ph.utexas.edu/ control/, 2004, Atom Optics Laboratory Center for Nonlinear Dynamics and Department of Physics University of Texas at Austin

[Steck] Daniel A. Steck, *Ethernet-Based Experiment Control Architecture*, http://www.steck.us/control/

[AD9852] Analog Devices *CMOS 300 MSPS Complete DDS*, 2004.

[NIDAQMANUAL] National Instruments Corporation, *NI 653X User Manual for Traditional NI-DAQ High-Speed Digital I/O Devices for PCI, PXI, CompactPCI, AT, EISA, and PCMCIA Bus Systems*,2005