List of Tables ______________________________4
List of Figures _____________________________5
Introduction _______________________________8
The Big Picture ____________________________9
IPP Timing and Multiple Frequencies__________11
Software Domains _________________________12
Piraq DSP code ___________________________12
Linux Piraq Driver ________________________12
Linux MAPR applications ___________________12
System Timing ____________________________13
Timers ____________________________________13
  Gating Synchronization Errors _____________13
Wavetable ________________________________14
Splitting the gates between the DSPs. _________15
  Wavetable, coding, and multiple frequencies_15
Pulse Coding for Dummies _________________16
  The Goal _________________________________16
  The Nitty Gritty __________________________16
  So What? _________________________________17
  Coherent Integration ______________________17
  Complementary Coding ____________________18
  Practical Considerations __________________18
  Range Truncation _________________________19
  Verification of the technique _______________19
    Aggregate SNR improvements ______________19
    Hard Target _____________________________19
    Injecting a Pulse _________________________19
    Using a full length injected pulse _________21
    Partial Decoding _________________________21
    Using the coded transmit pulse _____________21
    Test Procedure __________________________22
DSP Software ______________________________24
  Data Structures __________________________24
  Processing Flow __________________________25
  Data Upload ______________________________26
  FifoRead() ______________________________26
  Control Variables _________________________26
Linux _____________________________________28
  Configuring and building the kernel _________28
    Background ______________________________28
The procedure _____________________________28
System Time ______________________________29
The Linux PIRAQ Driver _____________________30
  Data upload from PIRAQ _________________30
Application Software _______________________33
  Profiler File _____________________________33
  Coff Library ______________________________33
  Piraq Interface ___________________________33
  Mapr _______________________________________33
  Mapr Display ______________________________33
Odds and Sods ____________________________35
  Piraq A/D Calibration _____________________35
  Piraq Card Funnies _________________________35
  Magma PCI Expansion Unit _________________36
  Linux _________________________________36
  Dell Workstation________________________36
  VMWare _________________________________37
  Connectors______________________________37
List of Tables

Table 1. Wave table digital I/O usage......................15
Table 2. Code Pairs..................................................18
Table 3. Sums of odd and even codes......................20
Table 4. Decoding a sub-pulse.................................20
Table 5. Decoding a full pulse.................................21
Table 6. Partial decoding ........................................21
Table 7. Maximum gates per DSP .........................25
Table 8: PIRAQ driver interface functions..............30
Table 9. dBm versus A/D counts.........................35
List of Figures

Figure 1. MAPR block diagram...............................10
Figure 2. IPP and sample timing..............................10
Figure 3. DSP data organization..............................24
Figure 4. maprdisplay thread assignments...............32
Figure 5. PIRAQ breakout connector ......................37
Figure 6. Wave Table Timing.................................38
Introduction

The next version of MAPR is based on an Intel machine running Linux, and using the NCAR PIRAQ for the initial radar processing. This development required creating a Linux device driver to run the PIRAQ cards, and porting/adapting the Solaris x86 based MAPR code to Linux.

Adoption of the PCI based PIRAQ cards has significant benefits for MAPR. The PIRAQ IF sections provide improved signal characteristics. The DSPs provide for increased computational horsepower and flexibility. For instance, since individual data samples are delivered to the DSP, alternative coherent integration strategies (as opposed to simple block averaging) can be explored. The increased PCI bus bandwidth, over that of the ISA bus, means that the Linux side will be spending less time doing the data uploads. Additionally, the count of specialized boards is halved, since the separate coherent integrator cards are eliminated.
The Big Picture

A block diagram of the Mapr system is given in Figure 1. The system is divided between the Piraq cards and the workstation, which communicate across the PCI bus. The individual components are described in greater detail in later sections.

The general data flow is from top to bottom, beginning with the RF signal, at the IF frequency, being fed to the digital IF section of the Piraq. The timer and wavetable control the overall timing of the Piraq card. Baseband measurements, integrated across the radar range gates, are transferred to the Piraq FIFOs. The FIFOs are dynamic memories which allow samples to back up when the downstream DSP processing is unable to handle them. Thus the FIFOs provide a throughput buffering function very early in the data transmission path.

Each Piraq has two parallel DSPs available for onboard computations. The wavetable architecture of Piraq is very general, and data samples can be routed to either DSP in arbitrary schemes simply by the programming of the wavetable. The method used in Mapr is to route the lower half of the range gated samples to DSP0 and the upper half to DSP1.

In Mapr, the DSPs are used only to perform the coherent integration. A specified number of IPP samples are collected and added to create a single sample in time for each gate. A specified number of these integrated samples is then buffered up within the DSP and transmitted to the workstation in a single block for all gates.

Pulse coding is a technique used to increase the effective power output of the transmitter while retaining range resolution. It requires the transmission of differently phased radar pulses, and the subsequent separate coherent integration of the radar returns. Thus the DSPs will collect even and odd coded time series, and send these to the workstation.

Mapr can be configured to operate as a multiple frequency system. This requires collecting separate pulse coded time series for each frequency, and transmitting these to the workstation.

The software on the workstation side is divided between the Linux Piraq driver and the application code.

The driver controls hardware considerations of the Piraq board, handling interrupts, accessing control registers on the Piraq card, and performing data transfers across the PCI bus. Many of these functions depend on a programming protocol between the driver and the Linux driver, where the driver requires the existence of certain predefined variables in the DSP programs, placed at fixed locations in the DSP shared memory. The data transfer protocol is also mandated by the Linux driver, and relies on a strict interrupt and mailbox type of interaction between the driver and the DSP code. The upshot of all of this is that the DSP code must be written to strictly maintain these definitions and protocols. This is not difficult, since these dependencies are well isolated in the existing DSP code base.

The Piraq driver provides the standard Linux programming interface to the application software. Read(), mmap() and ioctl() calls are used for the various data transfer, Piraq configuration and wavetable programming activities.

The application software is used to configure and run the radar, to compute data products, to produce related graphics displays, and archive raw time series and computed products. Computations cover power spectra, SNR, and correlation functions.

References

1 The number of samples collected for integration is referred to as “Number of Coherent Integrations”, or NCI. The length of a blocked time series is often referred to as “POINTS”
Figure 1. MAPR block diagram

Figure 2. IPP and sample timing
IPP Timing and Multiple Frequencies

Mapr is typically used in a single frequency configuration, where all transmit pulses are at the same frequency. It can also be used in a multiple frequency mode, which is required to perform Frequency Domain Interferometry (FDI). Since Mapr also has multiple antennas and receiver chains, the analysis can even include Spaced Antenna FDI (SAFDI).

Figure 2 shows the transmit pulse and final data sample output for a single frequency and dual frequency configuration. Extrapolation to higher numbers of frequencies is straightforward.

In the diagram, time increases from top to bottom. The “Tx” line depicts the transmitter output, with each box labeled E or O representing a pulse at the IPP rate. Since Mapr implements complementary pulse coding, the transmit pulses alternate between the even(E) and odd(O) codes. The sub-elements in each transmit pulse are to indicate that phase coding takes place within each pulse.

The “data” line in the figure represents the resulting output data sample, after pulse decoding has been performed in the application software. Note that “NCI” is defined as the number of pulses (both even and odd) that go into the data point for a given frequency. When operating with multiple frequencies (NF), we can consider the data output to be a stream of interleaved time series, one per frequency, each having the same data rate.

The final data sample rate then is given as:

\[ f_{\text{sample}} = \frac{1}{\text{NCI} \times \text{IPP} \times \text{NF}} \]

This shows that to maintain an equivalent sample rate as the number of frequencies is increased, the NCI must be decreased by the corresponding factor. This will naturally lead to a decreased SNR for the individual frequency time series.

In the remainder of this application note, most of the discussion is presented in terms of a single frequency. When the system is operated in an FDI mode, the information should apply to each of the independent frequencies.
Software Domains

The MAPR software can be divided into three main categories. These are:

**Piraq DSP code**

This is the code running on the Piraq card. It is written in C, with a bit of TI C40 assembly language. This code handles the front end data sampling, filtering, coherent averaging, and first pass data analysis (e.g. FFT computation). Data are then transferred to the Linux host via memory on the PCI bus.

The TI Code Composer tool is used as the development environment for the DSP codes. This tool is supported under Windows95/98, and so a W98 virtual machine is run under VMWare under Linux, to provide the development environment alongside of Linux. Apparently there is a port of gcc which will produce C40 object files, and so this may possibly provide an alternative development environment.

**Linux Piraq Driver**

This provides the operating system support for Piraq. Its most important function is to provide interrupt handling and data buffering so that Piraq data transfers can be performed in the multiprocessing environment. All Piraq functions are accessed through the driver. The Piraq driver is implemented as a Linux loadable module. It is a character type driver, written in C. Applications connect to the driver via the read(), ioctl() and mmap() interfaces.

**Linux MAPR applications**

These comprise a suite of programs that control operation of the radar, manage and perform the real-time data analysis, and provide graphical displays. Data acquisition and control of the Piraq cards is made through the Piraq driver. This code is implemented as a collection of C++ class libraries.

The class library is also aware of coff files. There should be no need for user applications to open the PIRAQ devices directly or make calls on the file descriptors; this should be left to the PIRAQ class libraries.
System Timing

Timers

The Piraq contains a versatile timing module, which is described in the Piraq manual. The description that follows here applies to the way in which the timer is used in the Mapr application.

The critical feature of the timing module is that it is used to generate a trigger signal that causes the wavetable to be executed. This means that once the trigger is generated, the wave table will be clocked at 48 MHz. The wave table is stopped only when it detects a stop bit in the table itself. Thus if the table is still running, generation of a new trigger should have no effect. Therefore, care must be taken to insure that the timer section and the wavetable operate correctly together.

The timer module actually contains six separate counters, which are counting an 8 MHz input signal. In addition to the wavetable trigger mentioned above, these counters are also used to create an additional signal known as the test pulse.

The Piraq timer module can be configured to either run continuously once it has been started, or to run once and then stop.

The test pulse generator is configured to produce a pulse of a specified width, at a certain time delay after the timer has started. Similarly, the wavetable trigger is also configured with a (different) start time delay.

The wavetable trigger can actually be configured to generate staggered intervals, e.g. a long interval followed by a shorter one. This would be used for generating a staggered IPP, for instance if one needed to run the radar in two different modes simultaneously. They are referred to as the even and odd triggers. Currently in Mapr, the two intervals are set to be identical.

In order to synchronize the four Piraqs in Mapr, the first Piraq acts as a master, with the additional Piraqs being slaved to this board. The timer modules are configured as follows:

- The even and odd IPP lengths are set equal on all of the Piraq cards.
- The timers on the master Piraq are programmed to start running on command; i.e. when the timer start bit is set by software. They are also configured to run continuously, automatically reloading the initial counter settings. Thus once the software has started the timers on the master Piraq, the wavetable trigger and the test pulse are generated at the proper IPP rate. The software must disable the timer to stop the IPP cycle.
- The Master Piraq test pulse generator is programmed to generate a pulse as soon as possible after starting; i.e. its counter is initialized to 1 (the minimum acceptable value).
- The timers on the slave Piraqs are configured to start by an external input, rather than on software command. The test pulse output from the master Piraq is fed into the external trigger input of the slave Piraqs. Thus the slave Piraq timers will run shortly after the master has been started. This delay is a consequence the master test pulse delay, which due to the logic design, will be at least 1 count.
- It takes a little time for the master test pulse to be generated and then for the slave timers to start running after receiving the pulse. The wavetable trigger delay is used to compensate for this discrepancy between the master and slaves. It is set to 1 for the slaves (the minimum possible), and set to 8 for the master. The latter value was determined empirically2.

Gating Synchronization Errors

A troubling artifact arises out of the current timing scheme. Recall that the wavetable is clocked by a 48 MHz signal, while the timer section runs on an 8 MHz clock. Since the master Piraq timer module is used to generate the synchronization on the slave Piraqs, the resolution of this is 1/(8MHz) (125 nS), which corresponds to 18.75 m of radar range. However, the wavetable has a resolution of 1/(48 MHz) (20.8 nS).

2 This should be verified more rigorously.
In Mapr, the gate sampling is programmed identically in the master and slave Piraqs. However, due to the resolution limitation of the timer modules, the gate sampling synchronization between Piraqs can be certain only to within 18.75 m. This gate sampling offset between Piraqs will be random, and will vary between 0 and 125 nS, in steps of 20.8 nS. This has been verified with test equipment. Mitch Randall has promised to find a solution to this problem.

**Wavetable**

The wavetable is essentially a clocked memory, containing 24 bit-wide words. When a given word in the wavetable is addressed, its contents are placed on output lines which are connected to various functions on the Piraq card. Some of the bits are used for logic functions such as controlling the digital IF section, gate sampling and DSP buffers, etc. Other bits are used to drive an ADC to produce an arbitrary analog waveform. Eight of the bits are brought off the card for digital output use. A couple of the bits are used to actually control the wavetable functions.

The wave table is clocked at 48 MHz; thus each word represents a time interval of 20.833 nS, which corresponds to a length of 3.13 m.

The digital output lines of the wave table can be used for any sort of external control function. In Mapr, some lines are used for diagnostic purposes; e.g. setting a bit to indicate when a gate is being sampled; etc. Others are used for external hardware control, such as phase coding and generation of the transmitter pulse.

**NOTE:** it appears that setting one of these bits in word 0 of the wave table does not work as expected. Setting the bit high seemed to cause the line to be turned off during the whole run of the wave table, and raised high outside of that.

The rules for wave table timing and their relationship to the MAC are given next. I determined these relationships experimentally. Mitch warns that the description given in his original documentation are not correct – they were design specs, but he said that he had to experiment later to determine the correct usage. Also, the following recipe may not be completely mandatory, but we will follow these guidelines in the MAPR application.

All parameters are given here in terms of the number of wave table entries. Thus a gate spacing of 50m would correspond to 8 wave table units.

- For safety, initialize the wave table past the end of the timing sequence by a few entries. Since the stop and rewind bits have latencies, if the next few following MACACC, MACOE and WREN bits are not programmed properly, spurious samples can be sent to the FIFOS, which really fouls up the DSP gate synchronization with the FIFOS. Initialize FIFO1, FIFO2 and MACACC to 1. Initialize the gain to whatever non-zero gain is being used. Initialize all other bits to zero.

- Gate spacing (GSPACE) must be even.

- On Mitch’s advice, the MACOE is not used to generate an EOF word. It is always set to zero. Instead, the gain is set to zero at the appropriate time in order to generate an EOF word with a value of zero.

Let $i_g$ represent the index in the wave table corresponding to the end of a given gate, and the EOF word. $i_g$ must be even valued.

- MACACC[$i_g$-G-1] = MACACC[$i_g$-G] = 0 forces a load of the MACC, to start the summation across the gate.

- All other MACACC bits were initialized to 1, so the MAC will be accumulating on the other entries.

- FIFO $[i_g] = FIFO[i_g+1] = 0$ writes the accumulation to the FIFO. Either or both FIFOs can be chosen.

- All other FIFO bits were initialized to 1, inhibiting the writes to the FIFOs at other times.

- To generate the zero valued EOF word, set GAIN[$i_g$-1…GAIN[wave end] = 0, for $i_g$ corresponding to the last real gate.

- To stop the table after the EOF word, where $i_g$ corresponds to the EOF gate, set STOP[$i_g$-2] = 1.

- To stop and rewind the table after the EOF word, where $i_g$ corresponds to the EOF gate, set STOP[$i_g$-2] = 1, REWIND[$i_g$-1] = 1.

**PW** = sub-pulse width
NCODE = code length
CPW = NCODE*PW = coded pulse width
FIRSTGATE = beginning of first gate. Must be even, and greater than CPW.
NGATES = number of gates
GATESPACE = gate spacing
IPPLEN = FIRSTGATE + (NGATES+1)*GATESPACE

Do we need to add anything after IPPLEN to account for latencies in reading the stop/rewind?

Table 1. Wave table digital I/O usage

<table>
<thead>
<tr>
<th>Line</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>sync</td>
<td>Pulsed at wave table location 1 (not 0!)</td>
</tr>
<tr>
<td>D1</td>
<td>hpa</td>
<td>Power supply control for transmitter HPA</td>
</tr>
<tr>
<td>D2</td>
<td>freset</td>
<td>Pulsed in order to reset frequency selector</td>
</tr>
<tr>
<td>D3</td>
<td>phase</td>
<td>Low to select even code phase; high to select odd code phase</td>
</tr>
<tr>
<td>D4</td>
<td>blank</td>
<td>Raised when RFSWITCH is set to one</td>
</tr>
<tr>
<td>D5</td>
<td>tx</td>
<td>Transmit pulse</td>
</tr>
<tr>
<td>D6</td>
<td>fnext</td>
<td>Pulsed to bump frequency selector</td>
</tr>
<tr>
<td>D7</td>
<td>gates</td>
<td>Raised when FIFO 1 or FIFO 2 are sampling</td>
</tr>
</tbody>
</table>

Wavetable, coding, and multiple frequencies

The wavetable is used to generate the transmitter pulse, the sub-pulse phase coding, and the switching between even and odd complimentary codes. Referring to the single frequency case in Figure 2, we see that the even and odd codes alternate after every IPP. The wave table thus has two nearly identical segments in it. The gate sampling will be duplicated between these segments. However, the even phase code will be specified in the first segment, and the odd phase code will occur in the second segment. The other difference is that the first segment will end with only a stop bit set, so that the table stops there at the end of the IPP. The next IPP trigger from the timer module will cause the second segment to run. The second segment then has both a stop bit and a rewind bit set, to force the wavetable back to the beginning of the first segment, to wait for the next IPP trigger.

In the case of multiple frequencies, the even and odd wave table segments are duplicated in succession for each frequency. All segments end with only the stop bit set, except for the last segment, which will have the rewind bit set. The successive pairs of even and odd segments will use the D2 and D6 digital output lines to select the transmit frequency.

Splitting the gates between the DSPs.

The DSPs will each process half of the gates. If there are an odd number of gates, the first DSP will get the extra one. The second DSP will always get the EOF word.

The second DSP will also get extra gates necessary to perform full decoding at its lowest gate, corresponding to the middle of the profile. It will receive NCODE − 1 gates. Therefore, the number of gates read by the first DSP will be:

NGATES/2 + NGATES % 2

The number of gates read by the second DSP will be:

NCODE − 1 + NGATES/2 + 1
Pulse Coding for Dummies

This section will present a basic explanation of pulse compression. The same information is presented throughout the literature, but usually the treatment assumes that the reader already understands key assumptions at one point or another. The following sections will describe the basic approach in the simplest terms. Since I was so confused by the terminology at first, somewhat pedantic definitions of terms will be given as the description proceeds.

The Goal

In a pulsed radar system, there is an inherent tradeoff between the range resolution and the transmitted power. Higher resolution requires a shorter transmit pulse, but this decreases the transmitted power, and leads to a corresponding decrease in the SNR of the return signal. Pulse compression techniques provide methods for increasing the average transmitted power while retaining a given range resolution. In general, a longer pulse is transmitted but the return signal is still sampled at the desired resolution. The result is that returns from a desired range will appear in adjacent range gates. The trick is in coding the long pulse so that the returns for a single gate can be extracted from these adjacent gates.

Thus, the technique of transmitting a long modulated pulse, and then manipulating the return signals to achieve a finer range resolution, is termed pulse compression, since the long pulse has been effectively compressed.

The key to the technique is the modulation of the transmit pulse. This is typically done by defining a series of values that defines the modulation. The long pulse length, divided by the desired range resolution (in time), defines the length of the series. The series is referred to as the code. Pulse compression, achieved by modulation of the transmitted pulse by a code, is known as pulse coding.

The Nitty Gritty

First, we will define an n-length code as a sequential series of positive and negative ones:

$$ C = \{ c_1, c_2, \ldots, c_n \}, \quad c_j \in \{1, -1\} $$

The transmitted radar pulse will consist of a long pulse, divided equally into n sub-pulses. The phase of each sub-pulse is set according to the corresponding code value $c_j$. Thus the phase is 0° for a $c_j$ of 1, and the phase switches to 180° for $c_j$ of -1. This is known as bi-phase coding.

A fixed period after the transmit pulse, the sampling begins, and the receiver voltages are measured at the same interval as the sub-pulse duration. Choose time $t_i$ such that the return of the last sub-pulse, i.e. the pulse with code $c_n$, is received for the target response of gate $g_i$. Because this time corresponds to the last sub-pulse in the sequence, the sample at that time will also contain returns for earlier sub-pulses reflected from higher gates, since they have had time to travel further. Therefore, the measured receiver voltage $v_i$ will contain contributions from n gates:

$$ v_i = c_n g_i + c_{n-1} g_{i+1} + \ldots + c_2 g_{i+n-2} + c_1 g_{i+n-1} $$

Next we will cross correlate a series of voltage measurements with the transmitted code. It may not be immediately obvious that this is useful, but it will soon be seen that this has important properties. Perform the correlation using the measurements made prior to and including $v_i$:

$$ G_i = c_1 v_{i-n+1} + c_2 v_{i-n+2} + \ldots + c_{n-1} v_{i-1} + c_n v_i $$

We will substitute the voltage/gate response relationship into this equation, showing each term, and obtain:

$$ G_i = + c_1 (c_n g_{i-n+1} + c_{n-1} g_{i-n+2} + \ldots + c_2 g_{i-1} + c_1 g_i) $$
$$ + c_2 (c_n g_{i-n+2} + c_{n-1} g_{i-n+1} + \ldots + c_2 g_i + c_1 g_{i+1}) $$
$$ + \ldots $$
$$ + c_{n-1} (c_n g_{i-1} + c_{n-1} g_i + \ldots + c_2 g_{i+n-2} + c_1 g_{i+n-1}) $$
$$ + c_n (c_n g_i + c_{n-1} g_{i+1} + \ldots + c_2 g_{i+n-2} + c_1 g_{i+n-1}) $$

The key point here is that each voltage measurement at the nominal gate $g_i$ contains contributions from the target return at that height, and heights above it. Therefore including terms from the $n-1$ heights that are nominally below $g_i$ will collect all measurements that include returns from the desired height. We see that $G_i$ contains measurements that bracket the gate.
Next we rearrange the terms and collect the factors. Note that some extra terms have been included in order to better illustrate the result:

\[
G_i =
+ (c_i c_n) g_{i-n+1}
+ (c_i c_{n-1} + c_2 c_n) g_{i-n+2}
+ ...
+ (c_i c_2 + c_2 c_3 + ... + c_{n-i} c_n) g_{i-n}
+ (c_i c_1 + c_2 c_2 + ... + c_{n-i} c_n) g_i
+ (c_2 c_1 + c_3 c_2 + ... + c_n c_{n-1}) g_{i+1}
+ ...
+ (c_{n-i} c_1 + c_n c_{n-2}) g_{i+n-2}
+ (c_n c_1) g_{i+n-1}
\]

Naturally, we see that \( G_i \) contains a sum of returns from gates above and below \( i \). Notice that relative amplitudes of the contribution of each gate take the form of the coefficients of the auto-correlation function (ACF) of the code series itself. This is a very useful result.

Right off the bat, since an ACF is always maximum at zero lag, the maximum amplitude in \( G_i \) will be contributed by gate \( g_i \). We also know that this amplitude will be equal to \( n \), since the code values are either 1 or \(-1\). Hence, we have achieved our objective of measuring a return (for the desired gate) that has \( n \) times the signal which would be achieved with a simple short pulse of length equal to the gate size.

However, we see that \( G_i \) is also contaminated by influences from adjacent gates. These unwanted contributions are referred to as sidelobes, I suppose because the graph of the ACF shows them as non-zero values out to either side of the main peak at zero lag. They are further known as range sidelobes, since they correspond to contamination from adjacent ranges. Incidentally, these sidelobes should not be confused with antenna pattern sidelobes.

The graph of the ACF is a very convenient way to intuitively gage the behavior of a given code series. At this stage it is useful to present a typical code, and its corresponding ACF. For example, the code series \(-1, -1, -1, +1\) has an ACF of \{-1, 0, 1, 4, 1, 0, -1\}:

We see that the sidelobes are any non-zero values occurring outside of lag zero.

**So What?**

The beauty of this whole scheme is that if a good code can be found; i.e. if its ACF “looks good”, then the impact of the sidelobes can be minimized, while retaining the contribution for the central gate. Great effort has been spent searching for codes with good characteristics. Barker, poly-phase, and pseudorandom codes are common families of codes. For instance, the Barker codes are widely known. It turns out that the amplitudes of all sidelobes in a Barker code are either one or zero, and the longer the code, the greater the proportion of sidelobes is equal to zero. For example, the 5 term Barker code \{+1, +1, +1, -1, +1\} has an ACF as shown in

**Coherent Integration**

At this point, it is worth mentioning coherent integration in relation to pulse coding.

Barker (and other codes) can be applied on individual and successive transmit pulses, and can be decoded on the corresponding individual returns. If the decorrelation time of the atmospheric phenomena is substantially longer than the IPP, then the pulses can be coded, the returns coherently integrated, and the decoding performed at the much slower rate of the coherent integration period. Thus the computational
cost of the decoding can be decreased by a factor equal to the number of coherent integrations. However, there will be no escaping the influence of the unwanted sidelobes.

Coherent integration is a crude form of filtering. The integration is simply a sum of all of the measurements made over the period. The number of terms can be quite small, e.g. less than 10 in a RASS measurement, or might increase up to several hundred or more depending on the radar configuration. It is frequently mentioned that this type of “boxcar averaging” has the effect of diminishing the spectral power as the signal Doppler shift gets further away from zero. At the nyquist limit, the power at the extreme frequencies will be 4 db down from the center3.

Perhaps there may be some advantage to using a more selective filter in place of the coherent integration. Typically the coherent integration has been performed by hardware-based adders, however in the Piraq based profiler design, an FIR or IIR filter could be substituted. It is not obvious if this filter would be done on I and Q separately (in parallel), or if it must be performed (somehow) on the complex time series.

**Complementary Coding**

Complementary codes provide a clever method for achieving (theoretically) zero contamination from the range sidelobes. Simply put, complementary codes are a pair of code sequences, whose individual ACFs have crummy sidelobe characteristics, but when summed they have zero sidelobes for all ranges. The Golay codes are one such family of sidelobe canceling binary codes4.

The technique is simple. A transmit pulse is modulated by the first code, and the returns are measured and decoded. The next pulse is modulated by the complimentary code, sampled, and decoded using the complimentary code. When the two results are summed, all range sidelobes should cancel out. The radar runs continuously in this manner, alternating between the primary code and its compliment.

---

3 We need a reference for this statement.

4 Are there others? What are their names?
2. The cross-correlation starts with the first code multiplying the measurement at time \( t_i \) and progresses up to the last code multiplying the measurement at \( t_i \).

3. Since the codes are either positive or negative one, multiplication may not be required. Instead, the code could simply determine whether the measurement is to be added or subtracted from the sum. The TMS320 conditional add and subtract instructions may be useful here.

4. Examination of the equations shows that the coherent integration may be performed in a block, and the decoding only needs to be carried out on the resulting sum. Four running sums are maintained during the coherent integration: I and Q for the first code, and I and Q for the complimentary code.

5. Since the results of two ACFs are added, the amplitude of lag 0 will be \( 2^n \). However, double the number of pulses has been transmitted. It may be a reasonable idea to simply scale the final values by the number of pulses that went into the total sum.

**Range Truncation**

As described above, the decoding process involves combinations of the returns from adjacent lower gate samples. This means that for (say) a code length of 4, there will not be enough gates to do full decoding of the lowest three gates. This leads to the caveat usually given that the minimum range of the radar is reduced when pulse coding is utilized. There is at least one technique, discussed in Ghebrebrhan[, which claims to be able to perform full decoding of these truncated ranges. Otherwise, one can simply use the partially decoded values as is, or choose to discard the lowest (truncated) ranges.

**Verification of the technique**

There are a lot of things that have to happen correctly in order for pulse coding to work properly. The transmit pulse needs to be phase modulated correctly, and with alternating codes on successive pulses. The sampled receiver data then needs to be organized so that alternating returns are summed during the coherent integration, and the correct code needs to be applied properly to the correct sum, and with the proper order in relation to the gate order. There are many places where the process can get fouled up. Since the atmospheric returns are extremely weak to start with, simple examination of the SNR levels during normal operation will not easily show that the procedure is working correctly, and in fact are not at all useful for this purpose.

Alternative approaches are thus needed to verify the operation of the pulse coding.

**Aggregate SNR improvements**

One method is to run the profiler with constant operating parameters, under nearly stationary atmospheric conditions, and vary the code length. The SNR values are recorded for all gates, and then plotted as a function of height. The family of SNR values should be a function of code length, i.e. the SNR values should increase 3 db for each doubling of the code length. This technique seems to work fairly well.

**Hard Target**

There may be methods that involve examining the radar returns from a hard target at a known distance. Perhaps the antenna would need to be tilted vertically so that a known target could be identified.

**Injecting a Pulse**

Another approach is to synthetically inject a pulse into the receiver chain at a known time. (The Piraq wavetable can be used to generate the pulse in relation to the IPP.) It could be difficult to generate a pulse that is phase coded as though it were the return from a phase coded transmit pulse. However, since we know the mathematics of pulse decoding, we can figure out what the decoded sampling of a non-coded pulse should look like, and see if this agrees with the sampling of the simple injected pulse.

Assume that we will inject a pulse whose width matches the range gate, as if we were not doing any pulse coding; i.e. there will be no phase coding of this pulse. Thus the pulse width is equal to what we have earlier called the sub-pulse.

Assume also that the sampled value of the pulse will be P. The sampled noise, in gates that don’t have the pulse, will have an average value that we will call N.

Recall that the measurements maintain two separate arrays of gate measurements, one for the even IPP pulse, and the other for the odd pulse. Note that this
is done even for no pulse coding; i.e. where the code length is one. To set the stage, the uncoded sampling of the gates should then look as follows, where \( g_i \) is the gate where the pulse is injected:

\[
\begin{array}{ccccccc}
& N & N & N & P+N & N \\
\hline
t & g_{i-3} & g_{i-2} & g_{i-1} & g_i & g_{i+1} \\
\end{array}
\]

The pulse decoding algorithm multiplies the first code times the even array of gates, multiplies the complimentary code times the odd array, and then sums the two. Let’s superimpose the code coefficients for a code length of 4, above the sampled data to show what the decoding products will be for each code. The odd codes are denoted with primes:

\[
\begin{array}{cccc}
c'_1 & c'_2 & c'_3 & c'_4 \\
N & c_2 & c_3 & c_4 \\
\hline
t & g_{i-3} & g_{i-2} & g_{i-1} & g_i & g_{i+1} \\
\end{array}
\]

Since the injected pulse is identical for both the even and odd IPP, the decoded value for the gate of interest should be:

\[
G_i = \sum_{j=1}^{n-1} (c_j + c'_j)N + (c_n + c'_n)(P + N)
\]

The n-1 gates above \( g_i \) will have similar decoded values, except that the two codes which multiply \( P \) are successively shifted towards the first code in the sequence. All other gates will have decoded values which are simply the sum of the even and odd codes, multiplied by \( N \). This will hold for the partially decoded gates at the bottom as well, where we just sum fewer code terms.

So, we can tabulate what the relative values will be for the decoding of a simple injected pulse, for the \( n \) gates at and above the injected gate, and for all other gates as well. Note that it makes the procedure particularly simple to just add the even and odd coefficients before doing the multiplication. We use the code sequences given earlier.

**Table 3. Sums of odd and even codes**

<table>
<thead>
<tr>
<th>( n )</th>
<th>Even code, odd code and sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C +1</td>
</tr>
<tr>
<td></td>
<td>C' +1</td>
</tr>
<tr>
<td></td>
<td>Sum +2</td>
</tr>
<tr>
<td>2</td>
<td>C -1,-1</td>
</tr>
<tr>
<td></td>
<td>C' -1,+1</td>
</tr>
<tr>
<td></td>
<td>Sum -2,0</td>
</tr>
<tr>
<td>4</td>
<td>C -1,-1,+1,-1</td>
</tr>
<tr>
<td></td>
<td>C' -1,-1,+1,+1</td>
</tr>
<tr>
<td></td>
<td>Sum -2,-2,0,0</td>
</tr>
<tr>
<td>8</td>
<td>C -1,-1,+1,+1,-1,-1,+1</td>
</tr>
<tr>
<td></td>
<td>C' -1,-1,+1,+1,+1,+1,+1,-1</td>
</tr>
<tr>
<td></td>
<td>Sum -2,-2,-2,-2,0,0,0,0,0,0,0</td>
</tr>
<tr>
<td>10</td>
<td>C -1,+1,+1,+1,-1,-1,-1,-1,-1</td>
</tr>
<tr>
<td></td>
<td>C' -1,-1,-1,+1,+1,+1,+1,+1,-1</td>
</tr>
<tr>
<td></td>
<td>Sum -2,-2,-2,-2,-2,0,0,0,0,0,0,0</td>
</tr>
</tbody>
</table>

There is a very important point to remember about the generation of the synthetic sub-pulse. That is that the nominal gate sampling of gate \( g_i \) occurs at the appropriate time interval in relation to the timing of the last sub-pulse in the coded transmit pulse. Thus, when generating the injected sub-pulse, it is important to increase its start time by the total code length.

**Table 4 shows the expected response for the decoding of a single sub-pulse.**

**Table 4. Decoding a sub-pulse**

<table>
<thead>
<tr>
<th>( n )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noise</td>
<td>+2N</td>
<td>-2N</td>
<td>-4N</td>
<td>-4N</td>
<td>6N</td>
<td>-8N</td>
</tr>
<tr>
<td>i+16</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
</tr>
<tr>
<td>i+14</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
</tr>
<tr>
<td>i+13</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
</tr>
<tr>
<td>i+12</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2P</td>
</tr>
<tr>
<td>i+11</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2P</td>
</tr>
<tr>
<td>i+10</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
</tr>
<tr>
<td>i+9</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2P</td>
</tr>
<tr>
<td>i+8</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2P</td>
<td>-2P</td>
</tr>
<tr>
<td>i+7</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
<td>2P</td>
</tr>
<tr>
<td>i+6</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
<td>0P</td>
</tr>
</tbody>
</table>
Using a full length injected pulse

We obtain a different response pattern, if we inject a pulse of a different width. The following table illustrates the pattern that should be observed, if the pulse width is equal to the code length times the gate spacing; i.e. it is equal in length to a coded pulse. Remember however, that it has not been phase modulated. Table 5 lists the expected response for the decoding of the full length pulse.

Partial Decoding

The partial decoding of the noise signal, or an arbitrary signal that is present in all gates, will yield a particular response pattern as well. This is shown in Table 6.

Table 5. Decoding a full pulse

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noise</td>
<td>+2N</td>
<td>-2N</td>
<td>-4N</td>
<td>-4N</td>
<td>6N</td>
<td>-8N</td>
</tr>
<tr>
<td>i+5</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
<td>2P</td>
</tr>
<tr>
<td>i+4</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2P</td>
<td>0P</td>
</tr>
<tr>
<td>i+3</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
<td>...</td>
<td>2P</td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>...</td>
<td>...</td>
<td>-2P</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>...</td>
<td>-2P</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>2P</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6. Partial decoding

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noise</td>
<td>-2N</td>
<td>-4N</td>
<td>-4N</td>
<td>6N</td>
<td>-8N</td>
<td>-8N</td>
</tr>
<tr>
<td>14</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-6N</td>
</tr>
<tr>
<td>13</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-4N</td>
</tr>
<tr>
<td>12</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2N</td>
</tr>
<tr>
<td>11</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-4N</td>
</tr>
<tr>
<td>10</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2N</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-6N</td>
</tr>
<tr>
<td>8</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>8N</td>
<td>-2N</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-4N</td>
<td>6N</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2N</td>
<td>4N</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>0</td>
<td>4N</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2N</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-4N</td>
<td>2N</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>-2N</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>+2N</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Using the coded transmit pulse

An elegant approach is to simply feed the coded, but delayed, transmit pulse back into the receiver. The decoding should then produce a single pulse at the same gate, no matter which code length is used. The strength of the signal should increase linearly with the code length.

The pulse must be delayed by an appropriate amount so that it will appear in the desired gate. At the moment, the MAPR software allows the transmit pulse to be arbitrarily delayed, using the txdelay parameter.

The 915 MHz signal could be injected at the antenna input to the receiver. However, care must be taken to limit the signal level appropriately. Additionally, this would not really produce a signal, since the frequency offset would be zero.

An alternative method, which was tested successfully with MAPR, is to use the phase modulated 60 MHz
IF signal, and feed that into the IF input. In order to produce a detectable frequency, the phase modulation is applied to a $60 + \Delta f$ signal generated by a signal generator. The signal generator must be locked to the external 10 MHz MAPR reference oscillator.

In general, encouraging results can be obtained with this procedure. The min/max profile plot of raw time series was used for the verification. Start by transmitting an uncoded pulse. Adjust the output strength of the signal generator so that a strong value, say around +/-10 counts, is seen in the gate. Adjust the system delay via the delay parameter, so that the pulse is centered at the correct gate corresponding to the transmit delay.

As the coding length is increased, the amplitude of the signal will increase at the same rate. We did see some leakage in gates above the test gate, but the amplitude seemed to be down by about a factor of 10 or more. The cause of the leakage is not known.

Examining the SNR plots gave confusing results. Even though the time series amplitudes behaved as expected, the calculated SNR did not show the same behavior. If the signal strength is large, as for the verification using the min/max plots, the SNR values are extremely large and off scale on the plot. We tried turning down the output signal from the generator to where the SNR values are in range, e.g to an output level of –70 dbm. In this case the SNR values do not track the code length in the correct manner.

The smallest range on the min/max plot is currently +/- 1 count. This should be modified so as to allow for smaller ranges.

**Test Procedure**

A signal generator is substituted for the 60 MHz input to the phase shift box. The signal generator uses the MAPR 10 MHz reference for a time base reference. Set the signal generator to 60 MHz, plus or minus a few cycles. If it is set to exactly 60 Mhz, the base band signal will be at DC, and you will wonder why nothing is happening.

Put a drawing here of the test setup.

1. Measure the output amplitude of the MAPR 60 MHz signal.

2. Adjust the output level of the signal generator to be approximately the same.

Connect the signal generator to the phase shift box.

3. Connect the phase shift box output (PO) to one of the PIRAQ IF input channels on Mike’s mystery box. Optional, but desirable, is to insert some pads here so that the input signal does not saturate the receiver. The saturation will appear as clipping in the time series display. The test works without the padding, but the numerical results are liable to be a bit off.

4. Configure MAPR with values for IPP, NCI and NPOINTS that produce a time series length of one second. This makes the time series display show the same number of wavelengths as the signal generator is configured for.

5. Set the transmit delay to a large number, so that the TX pulse will appear in the gate sampling. Set the code length to one.

6. Run maprdisplay, and observe the time series. A sine wave matching the frequency of the generator (minus 60 MHz) should be seen. Adjust the tx delay until the observed signal peaks at a given gate. Note the signal amplitude using the min-max graph on the left side of maprdisplay.

7. Now successively increase the code length. The decoded signal should remain as a single peak in the same gate. The amplitude of the signal will increase linearly with the code length.

8. An extra test involves injecting a tx pulse that has not been coded. Simply disconnect the phase input to the phase shift box. The profiles of amplitude with height should match those found in Table 5.

9. Repeat the test, with different values for the tx delay. The signal should move cleanly from one gate to the next, when tx delay is changed by the amount equal to the gate spacing. Be sure to verify correct behavior as you move the signal across the mid-point in height of the gates. Since the data are split between the two DSP’s on the Piraq card at this point, this region in the data handling is particularly susceptible to programming errors.
DSP Software

Some givens:

- Each DSP on the Piraq will be loaded with the same program.
- The DSP will be aware of which one it is, i.e. DSP 0 or DSP 1.
- DSP 0 will process the lower half of the gates; DSP 1 will process the upper half. If there are an odd number of gates, DSP 0 will process the extra gate.
- For a code length of $n$, $n-1$ extra gates are processed by DSP 1. These are the gates below the midpoint of the profile, and are required for the full decoding of the midpoint gate.

Data Structures

The following discussion holds for all code lengths greater than 1. For the non-coded situation; i.e. a code length of one, the data are not divided between even and odd buffers, and so the data capacities are twice as large. In addition, the PCI data bandwidth is halved, since only half as much data is transferred to the host.

A data buffer has the following architecture:

![Figure 3. DSP data organization](image)

DSP Data Buffer Organization

$$2 \times (\text{points} \times (\text{gates} \times (\text{I and Q}))))$$

Three buffers are allocated in a buffer pool, and rotated through during operation. The first is used to collect the coherently integrated I and Q values. When coherent integration for all points is complete, this buffer is handed off to the analysis step, which calculates the FFT. The buffer is then given to the data transfer task, which uploads it to the host.

It may appear that the buffers are twice as large as necessary. However, this is required in the coherent integration phase, so that both complimentary codes are collected separately. The double sized buffers are retained for the following steps, since we may be transmitting both the raw time series and the analyzed results to the host. Keeping the extra buffers makes this convenient.

Thus, the number of gates and points that can be processed will be limited according to:

Points x gates $< 512K / (3 * 2 * 2)$,

Points * gates $< 43690$.

For a given time series lengths, the maximum number of gates per DSP will be:
Table 7. Maximum gates per DSP

<table>
<thead>
<tr>
<th>Length</th>
<th>Max. number of gates per DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>341</td>
</tr>
<tr>
<td>256</td>
<td>170</td>
</tr>
<tr>
<td>512</td>
<td>85</td>
</tr>
<tr>
<td>1024</td>
<td>42</td>
</tr>
<tr>
<td>2048</td>
<td>21</td>
</tr>
<tr>
<td>4096</td>
<td>10</td>
</tr>
</tbody>
</table>

Note that this is assuming that the full 512K words of DSP local ram are available for the buffers. If other structures are allocated in this memory, the values will change.

Also note that the gate limit must include the extra gates being processed by DSP 1 when the code length is greater than 1.

As stated previously, DSP 0 will sample the lower half of the total gates, or half plus one if the number of gates are odd. DSP 1 will sample the upper half of the gates, plus the overlapping lower gates necessary for full decoding.

The pulse-decoding algorithm is applied on the host system; both the even and odd data are transferred to the host.

The following diagram depicts the data buffer organization. I data for gate 0 of the even code begins at location 0, with I and Q pairs progressing for \( n \) points in time. All of the even code data is in the lower half of the buffer. Data for the odd code occupies the upper half. The brackets on gates 0, 1 and 2 demonstrate that those gates would be partially decoded for a code length of 4.

**Processing Flow**

The DSP code is organized in the common foreground/background architecture.

The `fifoRead()` interrupt service routine (ISR) is the foreground task, which reads the fifos when data are available, and stuffs it into the current fifo buffer. The ISR de-multiplexes the pulse coding; i.e. it sends sets of gates to alternating sections within the buffer.

The ISR also performs the coherent integration (i.e. summing) of returns over the coherent integration count. If FIR or IIR filtering were to be implemented in place of the coherent integration, it would take place in this routine as well.

When a complete series of points for both pulse codes has been collected into one buffer, the ISR sets the `dataReady` variable, to indicate that the buffer is full. At this time, it also moves to the next buffer in the pool, and initializes the entire buffer to zero.

The background task is the main C program. It is running in a loop, testing the `dataReady` variable. When it sees that a buffer is full, it applies the analysis to that buffer. The analysis will include the pulse decoding and FFT generation. When the analysis is finished, the buffer is handed off to the `linuxCommUpload()` routine, which begins the data upload to the host. Note that although the data buffers on DSP 1 contain the extra overlapping gates below the altitude midpoint, these gates are not analyzed (outside of the pulse decoding), and are not transferred to the host by DSP 1.

Throughout the background processing, there are calls to the `linuxCommTask()` routine. This checks to see if data is waiting to be uploaded to the host, and that the host is ready to receive it. If the host is ready, a transfer is initiated. See the following section on data upload for more details on the data transfer process.

Note that the whole process works through the series of rotating buffers. Thus, each step is using the buffer that the previous step has just finished using. There is no attempt to buffer data between steps; i.e. to allocate extra buffers to hold data between steps. This is because the DSP processing is completely deterministic. The data are essentially clocked through the system by the IPP and gate sampling rate. Because these are fixed rates, there are no activities that take a variable length of time. If each activity is unable to keep pace with the preceding one, then the DSP is simply overloaded. Extra buffering would not be helpful in this case.

---

6 The exception to this is the Linux upload time, which depends on the Linux interrupt response. This is discussed in the Data Upload section.
Data Upload

Data transfers from the DSP to the host take place through a portion of the Piraq dual ported ram (DPR) that is visible to the PCI bus. The procedure is simple. The DSP places the data in the DPR, sets a variable (also in the DPR) that gives the size of the transfer, and raises an interrupt on the PCI bus. Before raising the interrupt, it also sets a transfer active flag in the DPR that indicates that a transfer is in progress.

When the Linux Piraq driver services the interrupt, it reads the data across the PCI bus from the Piraq DPR. When the data have been read, the driver clears the transfer active flag.

Thus, the DSP can determine when the transfer is complete by examining the transfer active flag. This data transfer acknowledgment must take place by polling, since there is no way for the Linux driver to generate an interrupt on the DSP.

The data upload is initiated on the DSP by a call to `linuxCommUpload()`. This call can specify an arbitrary length of data to be transferred. `linuxCommUpload()` will initialize a pointer word count, and then call `linuxCommTask()`. This second routine is used to transfer data up to the host in chunks that are the size of the DPR buffer. It copies a chunk from the local ram to the DPR, and generates the PCI interrupt. Successive calls to `linuxCommTask()` cause successive chunks to be transferred, until the entire data set is sent.

If the host is not ready to receive data when `linuxCommTask()` is called, i.e. the transfer active flag is still set, then the routine returns without doing anything.

We see that the data transfer task proceeds in a polled manner. `linuxCommTask()` should be called throughout the data analysis portion of the background processing, so that data transfers through the DPR can happen as soon as the host is ready for them. These calls are inexpensive if the host is not ready, and so they only impose overhead if it is time to send another chunk of data to the host.

Of course, there is opportunity here for data overruns. This would occur for instance, if the Piraq driver were slow in responding to the PCI interrupt. `LinuxCommUpload()` is written so as to ignore an upload request, if there is one currently active. If this condition occurs, a data overrun flag is incremented on the DSP. This flag is routinely probed by the Piraq driver, so that the condition can be detected.

In addition, the Piraq driver allocates 100 buffers for each DSP, so that even if the MAPR application code is getting behind, a significant amount of data from the DSP can be buffered on the Linux side, without causing overruns on the DSP itself.

FifoRead()

The fifoRead() interrupt routine is triggered by a fifo half full signal.

Each invocation of fifoRead() will read all of the gates for I and Q from the two fifos. These values will be added to the running sums of gates for the current point in time. If the eof flag is set for this DSP, an eof word will be read. If FIR or IIR filtering is used instead of coherent integration, it would take place at this step. Note that this will require some extra buffer space allocation to hold the running terms of the filter. This space would be of size: terms*gates*(I and Q)

Control Variables

Main interface variables in the DSP code; set by the MAPR application code:

- `gates`: the number of gates to be sampled. This does not include the overlapping gates that DSP 1 uses for decoding. It is up to DSP 1 to calculate its buffer sizes with room for the overlapping gates, and to read the extra decoded gates from the fifos.
- `eof`: set true if the eof word is to be sampled. Typically only set for DSP 1.
- `points`: number of points in the time series
- `nci`: the number of coherent integrations.
- `codeLength`: the length of the code series. It may be 1, 2, 4, 8, 10 or 16.
Linux

Configuring and building the kernel

Warning – this documentation may become stale as we continue to upgrade and move forward with Linux kernels and discover other idiosyncrasies in the hardware. Take the following with a grain of salt.

Background

These are the steps that you follow to configure, build and install a new Linux kernel. There are a couple of directories that are key to this process:

- /usr/src - below this will be found individual directories corresponding to the Linux version. The label is typically the Linux version number, such as 2.2.5. A link named Linux will point to the currently active version; i.e. the Linux source version that application code is compiled against.

- /usr/src/Linux/arch/i386/boot - will contain bzImage, after the build is completed.

- /boot - the final kernel image, system map, and initrd image (if needed) are placed here.

- /lib/modules - contains directories where the modules are installed. The directory names usually reflect the version and extra version designations. Note that before the "make modules_install" is performed, the module directory name should be changed if you want to save an old set of modules, or the directory should be erased so that the existing modules are not carried forward into the new system.

Key files that are used during the process are:

- /usr/src/linux/arch/i386/bzImage - the new kernel image, which will be copied and renamed in /boot. The new name will be vmlinuz plus some variant on the version. Lilo will refer to this file name.

- /etc/lilo.conf - configuration file for the lilo program, which creates the boot loading setup. Multiple kernels can be configured to be chosen at boot time.

Note that we are not supporting the loading of modules at boot time. I could not see any need to do this, and we don’t have to endure the frustration of trying to work with the confusing initrd mechanism.

The procedure

1. Logon as root, and make sure that the link /usr/src/linux points to the desired source code version directory.
2. cd /usr/src/linux and "make mrproper".
3. "make xconfig" - this will open a configuration screen. There are only a few things that I have been changing in the configuration options:
   - Enabling symmetric multiprocessing
   - Turning off the automatic power management (APMD) options
   - In the low-level aic7xxx SCSI driver, change it from "m" (module) to "y". This insures that the SCSI driver is configured into the kernel, rather than having it load from an initrd file system at boot time.\(^7\)
   - Enable IP aliasing. It is not really needed, but it gets rid of a bunch of superfluous error message that show up in the log relating to the lo device.
4. edit the makefile, and set the EXTRAVERSION symbol. This is the way that you differentiate your multiple configurations of the same Linux source code versions. The final version identifier for the kernel and the modules will be the concatenation of the source code version and EXTRAVERSION; e.g. 2.2.5-piraq.

\(^7\) For some reason, loading of the SCSI driver from initrd would not work. The kernel would report that the scsi driver aic7xxxx had an undefined symbol "__global_sti". Making it resident in the kernel solved the problem, and also eliminated the need for an initrd image.
Commands that take a version parameter (such as mkninitrd) will use this version number to locate the correct modules directory. Likewise, version.h will contain this identifier.

5. "make dep" - bring the dependencies up to date.

6. "make bzImage" - this will build a new kernel.

7. "make modules" - this will compile the modules. After this is done, change to /lib/modules, and rename the module directory if you want to keep it, or remove it if you want to replace it.

8. "make modules_install" - this will copy the modules to /lib/modules/<version>.


10. Edit /etc/lilo.conf if you are adding (or removing) a boot choice.

11. "lilo" - will run lilo to create the boot information. **This step is crucial!**

12. Reboot and try out the new kernel.

**System Time**

When Linux boots up, it reads the real Time Clock (RTC) and sets the system time to this value. From then on, the RTC generates clock tick interrupts at fixed intervals, and Linux counts these to establish the system time. While Linux is running, the user can adjust the system clock by several means, such as rdate to another machine, using date directly, or using nntp. When the system is shutdown, the current system time is assumed to be the best guess at the time, and the RTC is set to the system time. The RTC then keeps track of the time until the system is booted again.

The Piraq driver is pretty heavy handed – it does a lot of memory reading across the bus during the interrupt service routine. While this is happening, other interrupts may be disabled, causing Linux to miss the RTC timer tick interrupt. When mapr is running with a large data transfer load, the system clock can loose many seconds per hour due to this problem.
The Linux PIRAQ Driver

The driver for PIRAQ is configured as a character device. It provides control of the PIRAQ through ioctl, reads from the file descriptor, and memory mapped I/O.

The ioctl interface is used for all manipulation of the PIRAQ status/control register. This may turn out to be cumbersome, in which case the status/control register interface will be moved to memory mapping in the user space. In this case all of the ioctl based functions would become memory map based.

<table>
<thead>
<tr>
<th>Function</th>
<th>Interface</th>
<th>Related Control Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP program download</td>
<td>Memory mapped</td>
<td></td>
</tr>
<tr>
<td>DSP run and reset</td>
<td>ioctl()</td>
<td>DSP RESET</td>
</tr>
<tr>
<td>Board reset</td>
<td>ioctl()</td>
<td>PCI SYSRESET</td>
</tr>
<tr>
<td>Data upload from PIRAQ</td>
<td>read()</td>
<td></td>
</tr>
<tr>
<td>Timer programming</td>
<td>Memory mapped</td>
<td></td>
</tr>
<tr>
<td>Timer</td>
<td>ioctl()</td>
<td>TP SOURCE, TRESET, TMODE, ODD TP, EVEN TP, ODD TRIG, EVEM TRIG, DELAY SIGN</td>
</tr>
<tr>
<td>Wave table programming</td>
<td>Memory mapped</td>
<td>RAM BUS</td>
</tr>
<tr>
<td>PLL programming</td>
<td>ioctl()</td>
<td>PLL CLOCK, PLL LE, PLL DATA</td>
</tr>
<tr>
<td>LED control</td>
<td>ioctl()</td>
<td>PC LED</td>
</tr>
<tr>
<td>Phase lock</td>
<td>ioctl()</td>
<td>PHASE LOCK</td>
</tr>
</tbody>
</table>

Table 8: PIRAQ driver interface functions

Data upload from PIRAQ

The basic scheme for data upload from PIRAQ is based on interrupt processing and the read() system call. When the PIRAQ has data to deliver to Linux, it generates an interrupt. The driver processes this interrupt, and transfers a single block of data from PIRAQ to an internal buffer.

If a process is waiting to read data, the buffer contents are returned to the read() call. If no process is waiting for data, and interrupts continue to be generated, successive buffers in the driver are filled. The number of driver buffers is fixed, and if all of them become filled before being read by a process, they are overwritten sequentially, and a buffer overrun status flag is set.

Thus, proper operation calls for the user process to issue a read() before starting the data collection task on the DSP.

The data transfer collaboration between the driver and the DSP requires coordination via the dual ported ram. There is a fixed address in the DPR which points to a structure in the DPR that is used for the driver/DSP communication. This structure will contain mailbox variables and pointers to buffers used for the data transfer. Once again these buffer pointers refer to areas in the DPR.

Note that the DSP and the driver locate the DPR in different address spaces. To the DSP, the DPR area begins at 0x80000000. To the driver, the DPR is at an arbitrarily remapped segment of virtual memory. For consistency, the pointers in the driver/DSP communication area will actually be offsets from the beginning of that DSP's DPR. It is important to remember that each of the two DSPs sees an identical DPR address space, whereas the driver sees one space, with half of it allocated to each DSP.

The driver/DSP communication structure is defined in the header PiraqDSP.h. This file is visible both to the driver source code and the DSP source code. A
library of C functions is coded for the DSP to implement the interface to the driver, and these routines must be included with all DSP applications running in the Linux environment.
Figure 4. maprdisplay thread assignments
Application Software

Profiler File

This library provides classes for storing raw and processed data in netCDF files.

When the object is deleted, the file is closed.

A sync() member function is provided to force writing of the current state of the file.

Coff Library

Piraq Interface

Mapr

The classes in this library are used to run the mapr data acquisition system, and perform the real-time calculations. The higher level classes include:

- Mapr – creates a MaprMaster and a MaprCruncher, and then coordinates them. It loops, waiting for data from MaprMaster, and then delivering it to the MaprCruncher thread. Mapr is derived from Thread.

- MaprMaster – initialize the Piraq timers and wave tables, download the dsp programs, run the dsp programs, and fetch data from the Piraq cards. Note that MaprMaster is not derived from Thread; i.e. it runs in the thread that created it.

- MaprCruncher – derived from Thread, this class runs in its own thread in order to perform the realtime calculations, without blocking the operation of the data acquisition activity beforemod by Mapr/MaprMaster. It can optionally send graphics data and notifications to another task via a MaprLink.

- Thread – A class used to create a new thread which the object will run in. It is derived from the Runnable class, which has semantics similar to the Java class of the same name. A particular style of use is required for Thread and derived classes. For of all, Thread or derived classes are singletons. To initialize the object, the instantiate() member of the class is called. When it is time to create the new thread, the run() member is then called. Kill() and join() are used to terminate the thread of execution. I’m not sure why Thread has to be a singleton.

- MaprLink – This unfortunate class is used to provide a communications connection between threads that cannot share the standard thread IPC mechanisms such as semaphores and so on, which is the case for Qt 1.2 applications. It uses pipes for this purpose. The pipe file descriptor is made available, so that the thread can use select() or poll() to detect asynchronous messages sent by the other thread. Two pipes are created and used in this manner to establish LEFTLINK and RIGHTLINK communication paths. A thread sends a MaprCommToken via the notify() member. In addition, MaprLink provides member functions which allow GraphicsBuffer(s) to be transferred from one side to the other.

- MaprCommToken
- GraphicsBufferPool
- GraphicsBuffer

Mapr Display

- MaprDisplay
- StatusWidget
- DataDisplay
Odds and Sods

This is a collection of comments, gotchas and other observations gleaned during the Mapr Piraq development. Take them with a grain of salt; they can easily become irrelevant as the system evolves.

Piraq A/D Calibration

The IF section on the Piraq uses an Analog Devices AD6640 12 bit A/D. The device is designed for digital IF applications and can sample at up to 65 MSPS. It is sampled at 48MSPS on Piraq.

The analog input voltage range is +/- 1 V, which is measured across a 51\(\Omega\) resistor.

The twelve bit output on the A/D allows for counts of +/- 2048. The conversion from counts to voltage then is: \(V_{\text{volts}} = \frac{D_{\text{counts}}}{2048}\). The conversion to power is as follows.

\[
P_{\text{mW}} = \frac{1000 \times V^2}{51}
\]

\[
P_{\text{dBm}} = 10 \log\left(\frac{1000 \times V^2}{51}\right)
\]

\[
P_{\text{dBm}} = 10 \log\left(\frac{1000 \times D^2}{51 \times 2048^2}\right)
\]

\[
P_{\text{dBm}} = 20 \log(D) + 10 \log\left(\frac{1000}{51 \times 2048^2}\right)
\]

\[
P_{\text{dBm}} = 20 \log(D) - 53.3
\]

Solving for \(D\) as a function of power:

\[
D = 10^{\frac{P_{\text{dBm}} + 53.3}{20}}
\]

This gives the following values of dBm versus A/D counts:

<table>
<thead>
<tr>
<th>Power (dBm)</th>
<th>D (counts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-53</td>
<td>1.00</td>
</tr>
<tr>
<td>-50</td>
<td>1.46</td>
</tr>
<tr>
<td>-47</td>
<td>2.00</td>
</tr>
<tr>
<td>-43</td>
<td>3.00</td>
</tr>
<tr>
<td>-40</td>
<td>4.60</td>
</tr>
<tr>
<td>-33</td>
<td>10.00</td>
</tr>
<tr>
<td>-30</td>
<td>14.60</td>
</tr>
<tr>
<td>-20</td>
<td>46.20</td>
</tr>
<tr>
<td>-13</td>
<td>100.00</td>
</tr>
<tr>
<td>-10</td>
<td>146.00</td>
</tr>
<tr>
<td>0</td>
<td>462.00</td>
</tr>
<tr>
<td>10</td>
<td>1462.00</td>
</tr>
</tbody>
</table>

Piraq Card Funnies

Over the course of the development, there has been a constant battle with the Piraq cards. Often the problem has been faults in the driver, but the Piraq has caused some problems as well.

- There are several programmable logic devices on the cards. These all need to be programmed identically and with the most current configuration. At one point, the PAL associated with the PCI interface on one Piraq was wrong, and would cause a system lockup when the card generated an interrupt.

- Some of the Piraq cards are looking pretty ragged (physically). The card edge connectors are showing substantial wear. On one card, one of the tabs on the PCI connector is halfway worn off.

- The Piraq cards use a lot of power (approx. 25 W according to Eric). This is at the top end of the PCI spec. An installation of 4 cards could overwhelm a wimpy power supply.
supply. When four cards are installed in adjacent slots, they need to have extra cooling provided. Some sort of supplemental ventilation will need to be built into the Dell workstation if the Piraq cards are installed there.

**Magma PCI Expansion Unit**

- I have had trouble getting this unit to work with a full complement of Piraq cards. I am currently suspecting that the problem is due to the power issue mentioned previously, although there may be other reasons.

- Have now gotten it to work with 4 Piraqs, after Eric reprogrammed the PAL on one of the cards. This PAL was different than the others, and is responsible for the PCI interface configuration.

- I’ve found that sometimes the BIOS puts out an error message: “Plug and Play Configuration Error”. We would get a variable number of these. I moved the Piraqs around in different slots and the messages went away! The current slot assignments are: 2, 4, 5 and 6. The host interface card is in slot 1. I have no idea why this is happening; let’s hope that the cards stay happy in this configuration.

**Linux**

- During the development, we have seen occasions where the interrupts seem to be all fouled up. This seemed more prevalent when using the expansion unit. The interrupt assignment and running count can be seen via `cat /proc/interrupts`. Run the data acquisition program, and then look at the interrupt counts to verify that the correct number of interrupts are being generated, for the correct Piraq card. In particular, look for unusually large numbers of interrupts being registered for any IRQ. This indicates a stuck interrupt line. Since the PCI interrupts are level triggered, if the interrupt is held asserted, an avalanche of interrupts occurs because it is never de-asserted. This could be because a DSP truly generated an interrupt, but it was sent to the wrong IRQ and associated interrupt handler, or because the Piraq is simply not resetting the interrupt. I have seen Linux run along happily while registering 40,000 bogus interrupts per second. They don’t impact the machine too much because there is no processing associated with them.

- During the Linux installation, the installer was detecting the video card as an Nvidia. I thought that this was wrong, and manually selected the Diamond V770. The documentation shows that this card uses the Nvidia chipset, and so the automatically detected configuration is probably correct. I will try it on the next installation.

- There have been any number of unexplained sudden machine lockups. Everything is gone; ctrl-alt-f2 doesn’t shutdown the X server, there are no pings from outside, etc. Now some people have suggested that the X servers can cause this kind of behavior. As further evidence, I tried to do a dump to SCSI tape. Three times in a row, the system would hang, and always after writing 60% of the partition. The fourth time, I shutdown X and worked in the ascii mode, and the backup worked just fine.

These things make me suspect either the video driver, or the X server itself. I’ll try using the Nvidia configuration first.

- The general scuttlebutt says that power management should be disabled for SMP Linux systems.

- The boot complains about an unknown IO-APIC version number 0x20, and instructs to mail to the smp-linux newsgroup. Reading that newsgroup shows many reports of this problem, but no precise solutions, and perhaps it doesn’t hurt anything. There was a hint that the warning disappears in the 2.3 kernals. In any event, the IO-APIC can be disable by using the “noapic” boot flag if necessary.

**Dell Workstation**

The internal SCSI drive died. It began by occasionally having trouble spinning up immediately at boot time. It would display errors right after the SCSI bios announced itself and found the disk.
VMWare

- VMWareTools – I have consistently been unable to install the VMWare tools as instructed. Even though they are enabled, Explorer under Windows cannot find the (pseudo) floppy driver. Jeremy at VMWare hinted at a workaround on one of the VMWare newsgroups. This involved using mtools to read the floppy disk file image, and then extracting the tool files. This was done in Linux, and then I used ftp to transfer them to the Windows guest. This way I was able to install the VMWare tools.

- Initially, VMWare would not work. The package installed just fine, but everything ran incredibly slowly. After working for a month on this with the VMWare support, they determined that the following needed to be added to the .cfg file:

  `hostapic.ioapicversion=0x20`

  We also added:

  `aiovtdelay=0`

  It turned out that VMWare did not recognize the version of the IO-APIC, and so interrupts were not being handled correctly. Indeed, the Linux boot log showed that the IO-APIC version number was unrecognized.

Connectors

An adjacent card edge blank can carry three connectors which are attached to connectors on the Piraq board as follows:

![PIRAQ breakout connector](image)

The LO carries the 48 MHz oscillator. The DAC is the output of the D/A convertor driven by the wave table. D0 through D7 are the signals generated by the digital output word of the wave table.
Wave Table Timing

*Note that d0 is signalled only on alternating prf's, so that timing discrepancies related to the pulse coding can be identified.

Wave Table Timing.