A 12-Channel Real-Time GPS L1 Software Receiver[1]

B.M. Ledvina, S.P. Powell, and P.M. Kintner,
Electrical and Computer Engineering, Cornell University

M.L. Psiaki,
Mechanical and Aerospace Engineering, Cornell University

Biography

Brent M. Ledvina is a Graduate Student with the Space Plasma Physics Group in the Department of Electrical and Computer Engineering at Cornell University. His areas of interest cover irregularities in the equatorial and midlatitude ionosphere and GPS technology. He has a B.S. in Electrical and Computer Engineering from the University of Wisconsin at Madison.

Mark L. Psiaki is an Associate Professor of Mechanical and Aerospace Engineering at Cornell University. He received a B.A. in Physics and M.A. and Ph.D. degrees in Mechanical and Aerospace Engineering from Princeton University. His research interests are in the areas of estimation and filtering, spacecraft attitude and orbit determination, and GPS technology and applications.

Steven Powell is a Senior Engineer with the Space Plasma Physics Group in the Department of Electrical and Computer Engineering at Cornell University. He has been involved with the design, fabrication, testing, and launch activities of many scientific experiments that have flown on high altitude balloons, sounding rockets, and small satellites. He has M.S. and B.S. degrees in Electrical Engineering from Cornell University.

Dr. Paul M. Kintner, Jr. is a professor of Electrical and Computer Engineering at Cornell University. His interests as a rocket scientist range from exploring the northern lights to understanding space weather. His recent work with GPS receiver design and scintillations led NASA to appoint him chair of the Living With a Star Geospace Mission Definition Team.

Abstact

A GPS receiver has been developed that runs 12 tracking channels in real-time using a software correlator. This work is part of an effort to develop a flexible receiver that can use new GPS signals as they become available without the need for new correlator hardware. The receiver consists of an RF front-end, a system of shift registers, a digital data acquisition (DAQ) card, and software that runs on a 1.73 GHz PC. The commercial RF front-end down converts the signal into a 2-bit digital data stream at 5.714 MHz. The shift registers parallelize the magnitude and sign data bit streams into separate words, which the DAQ reads into the PC's memory using direct memory access. The PC performs base-band mixing and PRN code correlations in a manner that directly simulates a hardware digital correlator. It also performs the usual signal tracking and navigation functions, under the control of a real-time Linux operating system.

[1]Patent Pending

The software correlator receives frequency commands for simulated carrier and code NCOs and, in effect, uses these to reconstruct carrier and code replicas which it mixes with the input data stream. The resulting signals are summed to produce the standard in-phase and quadrature, prompt and early-minus-late accumulations. These, along with the phases of the 2 NCOs, are sent back to the part of the code that executes the tracking loops and the navigation functions. The contributions of this work are a set of special high-speed algorithms for doing the correlations in software. They make use of bit-wise parallelism so that a single C-code command (partially) processes 32 samples at a time.

This system has been tested using a roof-mounted antenna. When operating with 12 channels, the entire receiver uses less than 50% of the capacity of the 1.73 GHz processor and navigates to an accuracy of 10 meters.

Introduction

A real-time software receiver architecture can provide GPS user equipment with operational flexibility that will prove more and more useful as time goes by. The current GPS system is slated to expand its capabilities to include new civilian codes on the L2 frequency and a new L5 frequency. A receiver that uses a hardware correlator will require hardware modifications in order to use these new signals. In the near term, a receiver designer will be faced with a complex trade-off in order to decide whether the extra complexity is worth the improved performance that will accrue only very slowly as new GPS satellites replace older models. A software receiver can use new signals without the need for a new correlator chip. New frequencies and new pseudo-random number (PRN) codes can be used simply by making software changes. Thus, software receiver technology will lessen the risks involved for designers during the period of transition to the new signals. Furthermore, a software receiver could be reprogrammed to use the Galileo system, GLONASS, or both, which provides an added benefit from the use of a software radio architecture. Thus, there are good reasons to develop practical real-time software GPS receivers.

A GPS receiver can be broken down into various components (see Figure 1). First, an antenna, possibly followed by a pre-amp, receives the L-band GPS signals. After the antenna comes an RF section that filters and down converts the GHz GPS signal to an intermediate frequency in the MHz range. The RF section also digitizes the signal. The next section is the correlator chip that separates the signal into different channels allocated to each satellite. A modern receiver has 10 or more channels. For each satellite, the correlator mixes the Doppler shifted intermediate frequency signal to base-band and correlates it with a local copy of a PRN code. The final components of the receiver involve software routines that track the signals, demodulate the navigation message, and compute the navigation solution.

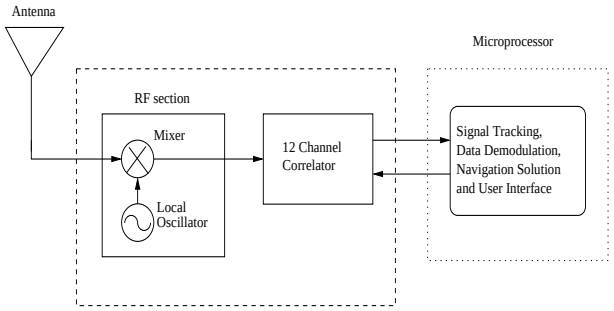

Figure 1: *A typical GPS receiver with special purpose hardware and general hardware separated.*

A software receiver differs from a standard receiver in one very distinct way (see Figure 2). The functions of the correlator chip are moved to software running on a general purpose processor. Doing so changes the components and layout of the receiver. The RF front-end is repackaged into a device called a bit-grabber, which outputs a binary bit-stream. A data buffering and acquisition system reads the bit-stream into a computer. The bit-stream is then available for processing by a software correlator running on the PC's processor.

The notion of a software GPS receiver has been around for several years. In the recent past, GPS software receivers have been developed that either post-process stored signals or operate in real-time. Previous real-time software receivers function with a limited number of channels (4-6) and require high-end computer speeds or DSP chips [*Akos et al.*, 2001a and *Akos et al.*, 2001b]. The work presented in this paper improves upon these previous works in two ways. First, the software receiver discussed here is

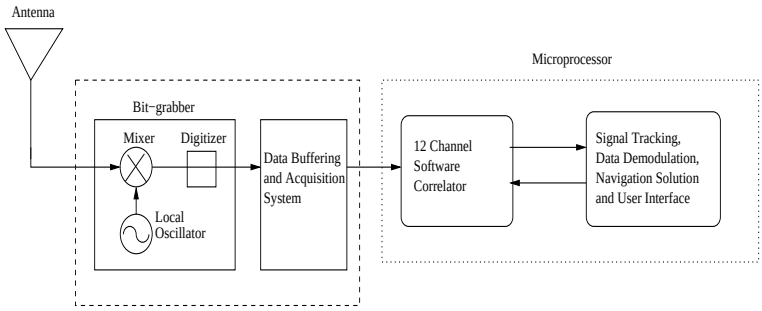

Figure 2: *A typical GPS software receiver showing the separation between special purpose hardware and general hardware.*

approximately 3 times faster than the results presented in *Akos* [2001a], thus enabling a 12-channel receiver. Second, this paper fully explains the algorithms used to compute the correlation accumulations that are required for acquisition and tracking.

The remaining portions of this paper explain the internal workings of a 12-channel real-time GPS software receiver and present experimental performance results for this system. The second section describes the hardware including the bit-grabber and PC. The third section reviews the structure of spread-spectrum signals and methods of acquisition and tracking. The next section presents a short description of an existing PC-based GPS receiver that has been modified to develop the present receiver. The fifth section presents an overview of the software correlator design. Section 6 gives a lengthy description of the mathematics behind base-band mixing and correlation leading into the implementation used in the software correlator. Section 7 discusses timing and measurements made by the software correlator. Section 8 describes how to keep all of the calculations in integer format. Section 9 presents some performance results. Section 10 gives a summary and concluding remarks.

System Configuration

Central to the software GPS receiver is the personal computer. The current system consists of a PC with a 1.73 GHz AMD Athlon processor running the RT-Linux operating system. RT-Linux is a hard real-time variant of Linux implemented as a set of patches to the standard Linux kernel. Due to its real-time optimized design, RT-Linux provides very low latency interrupt responsiveness along with the ability to execute threads at regular intervals. This translates into a highly efficient and responsive operating system that reliably executes time critical code. An additional feature of RT-Linux is that it keeps the functionality of Linux by running the kernel as the lowest priority thread. By retaining the functionality of Linux, it is very easy to develop, test, debug, and run real-time software. Another benefit of using Linux is that tools such as drivers, a C complier, and text editors are readily and freely available.

The next component of the software receiver is the bit-grabber. The bit-grabber consists of a Mitel GP2015 RF front-end. The front-end down converts the nominal 1.57542GHz GPS signal to an intermediate frequency of $(88.54/63) \times 10^6$ Hz $\cong$ 1.4053968254MHz and then performs analog-to-digital conversion. The resultant, digitized signal has two binary bits per sample corresponding to a sign and a magnitude. The possible values for the digitized signal are ± 1 and ± 3. Table 1 shows how to convert the binary sign and magnitude bits into integer values. The two binary bits are available as outputs from the bit-grabber. In order to provide accurate timing, the sign and magnitude bits are synchronized to a $(40/7) \times 10^6$ Hz $\cong$ 5.714 MHz clock signal, which is the third output from the bit-grabber card.

Another type of bit-grabber which uses a direct ADC down conversion implementation has also been used. The heart of this bit-grabber is an analog-to-digital converter that can function with an input bandwidth of up to 2 GHz and that can perform 8-bit conversion at continuous sample rates up to 1 GHz. The ADC samples the GPS signal at 5.714 MHz, which aliases the L1 carrier down

to a nominal frequency of $\cong$ 1.722 MHz. Each 8-bit sample gets processed by a separate logic unit to create sign and magnitude pairs with an appropriate input gain that minimizes the signal-to-noise ratio's digitization loss.

Sign	*Mag*	*Value*
0	0	-1
0	1	-3
1	0	+1
1	1	+3

Table 1: *Sign and magnitude combinations of the input GPS signal.*

A data acquisition system reads the digitized sign and magnitude bits from the bit-grabber into the PC. To make the process of reading data into the PC more efficient and to prepare for efficient correlation calculations, the DAQ card reads 32 bits of buffered samples at a time. The 32 bits consists of 16 sign bits and 16 magnitude bits. A series of shift registers buffer the data, packing the sign and magnitude bits into separate 16-bit words. A divide-by-16 counter converts the 5.714MHz clock down to 357.14KHz, which provides a signal indicating when the buffer is full.

The data acquisition system consists of a PC card and driver software. The card is a National Instruments PCI-DIO-32HS digital I/O card. Pertinent features of this card are the 32 digital input lines, direct memory access (DMA) and availability of a driver for RT-Linux. A suite of open source drivers and application interface software for DAQ cards known as COMEDI (COntrol and MEasurement and Device Interface) is freely available. COMEDI provides Linux/RT-Linux support for nearly one hundred DAQ cards spanning numerous manufacturers. One of the strong points of COMEDI is that it includes very general drivers, which are easily modifiable for specific applications. The demands of the software receiver necessitate certain modifications to the stock COMEDI driver for the PCI-DIO-32HS card. The modifications include increasing the number of input bits from 16 to 32, enabling DMA, and modifying the driver to support continuous interrupt-driven acquisition.

The software receiver is written entirely in C-code using tools available from standard Linux distributions. To promote portabilility of the software, no processor-specific assembly language or special instructions are used.

Review of the GPS Spread Spectrum Signal and Receiver Correlations

The received time-domain L1 coarse/acquisition (C/A) signal that gets output by the RF front-end is:

$$y(t_i) = \sum_j A_j D_{jk} C_j \left[0.001 \left(\frac{t_i - \tau_{jk}}{\tau_{jk+1} - \tau_{jk}} \right) \right] \times cos[\omega_{IF} t_i - \phi_j(t_i)] + n_j \tag{1}$$

where t_i is the sample time, A_j is the amplitude, D_{jk} is the navigation data bit, $C_j[t]$ is the C/A code, τ_{jk} and τ_{jk+1} are the start times of the received k^{th} and $k+1^{st}$ C/A code periods, ω_{IF} is the intermediate frequency corresponding to the L1 carrier frequency, $\phi_j(t_i)$ is the carrier phase perturbation due to accumulated delta range, n_j is the receiver noise, and the subscript j refers to a particular GPS satellite. The summation is over all visible GPS satellites. The negative sign in front of $\phi(t_i)$ comes from the high-side mixing that occurs in the RF front-end that has been used.

A GPS receiver works with correlations between the received signal and a replica of it. The correlations are used to acquire and track the signal. The replica is composed of two parts, the carrier replica and the C/A code replica. Two carrier replica signals are used, an in-phase signal and a quadrature signal. When mixed with the code replica they form the in-phase and quadrature replicas:

$$y_{Ij}(t_i) = C_j \left[0.001 \left(\frac{t_i - \hat{\tau}_{jk}}{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}} \right) \right] \times cos\{\omega_{IF} t_i - [\hat{\phi}_{jk} + \hat{\omega}_{Doppjk}(t_i - \hat{\tau}_{jk})]\} \tag{2}$$

$$y_{Qj}(t_i) = -C_j \left[0.001 \left(\frac{t_i - \hat{\tau}_{jk}}{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}} \right) \right] \times sin\{\omega_{IF} t_i - [\hat{\phi}_{jk} + \hat{\omega}_{Doppjk}(t_i - \hat{\tau}_{jk})]\} \tag{3}$$

where equations (2) and (3) apply during the k^{th} C/A code period. In these equations $\hat{\tau}_{jk}$ and $\hat{\tau}_{jk+1}$ are the receiver's estimates of the start times of the k^{th} and $k{+}1^{st}$ code periods, $\hat{\phi}_{jk}$ is the estimated carrier phase at time $\hat{\tau}_{jk}$, and $\hat{\omega}_{Doppjk}$ is the estimated carrier Doppler shift during the k^{th} code period.

A typical receiver computes the estimates $\hat{\tau}_{jk}$, $\hat{\tau}_{jk+1}$, $\hat{\phi}_{jk}$, and $\hat{\omega}_{Doppjk}$ by various means that are described in [*Van Dierendonck*, 1996]. These include open-loop acquisition methods and closed-loop signal tracking methods such as a delay-locked loop to compute $\hat{\tau}_{jk}$ and $\hat{\tau}_{jk+1}$ and a phase-locked loop or a frequency-locked loop to compute $\hat{\phi}_{jk}$ and $\hat{\omega}_{Doppjk}$. The software receiver developed here uses standard techniques for forming these estimates. These techniquires are not discussed in detail here.

The receiver uses the carrier and code replicas to compute the following in-phase and quadrature correlation accumulations:

$$I_{jk}(\Delta) = \sum_{i=i_k}^{i_k+N_k} y(t_i) C_j \left[0.001 \left(\frac{t_i + \Delta - \hat{\tau}_{jk}}{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}} \right) \right] \times cos\{\omega_{IF} t_i - [\hat{\phi}_{jk} + \hat{\omega}_{Doppjk}(t_i - \hat{\tau}_{jk})]\} \quad (4)$$

$$Q_{jk}(\Delta) = - \sum_{i=i_k}^{i_k+N_k} y(t_i) C_j \left[0.001 \left(\frac{t_i + \Delta - \hat{\tau}_{jk}}{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}} \right) \right] \times sin\{\omega_{IF} t_i - [\hat{\phi}_{jk} + \hat{\omega}_{Doppjk}(t_i - \hat{\tau}_{jk})]\} \quad (5)$$

where i_k is the index of the first RF front-end sample time that obeys $\hat{\tau}_{jk} \leq t_{i_k}$ and $N_k + 1$ is the total number of samples that obey $\hat{\tau}_{jk} \leq t_i < \hat{\tau}_{jk+1}$. The time offset Δ causes the replica PRN code to play back early if it is positive and late if Δ is negative. One of the main contributions of the present work is developing an efficient technique for the receiver to accumulate I_{jk} and Q_{jk} in software.

Use of Previously Existing GPS Receiver Software

Previous work is important to the implementation of this real-time GPS software receiver. The Mitel GPSArchitect GPS receiver was ported to RT-Linux [*Ledvina et al.*, 2000] and is herein referred to as Cascade. The Cascade GPS software coupled with the Mitel chipset (GP2015 RF front-end and GP2021 correlator) on an ISA card forms a GPS receiver for the PC. Since Cascade provides standard GPS functions (signal tracking, data demodulation, navigation solution, etc.) and is designed to interact with the GP2021, it is included as part of the real-time software receiver. Thus, no new developments have been needed for standard receiver functions such as code and carrier frequency steering during acquisition and tracking.

The software correlator is an independent RT-Linux module. The interface for interacting with this module has been designed to be similar to that of a hardware correlator. In fact, the software correlator closely mimics the GP2021 correlator in order to allow the Cascade GPS software to be merged with the software correlator. When using a hardware correlator, the receiver software interacts via I/O memory to read from and write to the correlator's registers. The software correlator uses an analogous strategy by implementing a shared memory buffer that both the correlator and the Cascade software can access. The memory buffer is implemented using the "mbuff" driver included with RT-Linux. This driver is ideal for real-time situations since it allows sharing of memory between kernel modules and restricts the Linux kernel from swapping the shared memory space to disk.

Thus, the software correlator has been designed as an independent module that interacts with other parts of the receiver according to well defined interface specifications. This modular approach provides flexibility in the internal workings of the receiver. One benefit of this modularity is that the mixing methods and correlation routines are transparent to the other standard software modules. This enables quick changes in correlator design that do not significantly affect other parts of the GPS receiver.

Software Correlator Design

Since a software correlator, as compared to a hardware correlator, does not process each channel in parallel the correlator calculations of a multi-channel software receiver represent a heavy computational burden. Therefore, it is important to explain the step-by-step process of software

correlation. An outline of the software correlator's functions that need to be completed every millisecond is given below.

1. Obtain the most recent carrier and code frequencies from the acquistion or tracking loops.
2. For each channel, first mix the signal to base-band using the most recent carrier frequency and the accumulated carrier phase. Then, compute in-phase and quadrature prompt and early-minus-late correlations using the most recent code frequency and the accumulated code phase.
3. Store the prompt and early-minus-late *I's* and *Q's* for use by the acquisition or tracking loops.
4. Repeat Steps 1-3 for each channel.
5. If a measurement time (denoted as a t_{TIC}) occurred, then store the current measurement data including C/A code phases, epoch counters, carrier phases, and carrier Doppler shifts.
6. Sleep for the remainder of the millisecond.

An examination of the correlation timing requirements is in order. To correlate one millisecond of data on 12 channels, computations must be completed in less than one millisecond. In order to leave computational time for other aspects of the GPS receiver, it is advisable to limit the processing of 12 channels to less than 750 microseconds.

Mathematical Methods of Software Correlation

A correlator has three main functions. First, it mixes a signal to base-band using the estimated carrier Doppler shift and carrier phase. Second, it mixes the base-band signal with a replica of the C/A code using the estimated code phase and code chipping rate. Third, it sums the resulting signal over a C/A code period. Since the received L1 signal has an uncertain carrier phase, the correlator computes both in-phase and quadrature accumulations, as defined in equations (4) and (5).

Base-Band Mixing

Base-band mixing is a multiplication of an input signal by a complex exponential where the frequency of the complex exponential approximately matches that of the input signal. The resultant signal is centered at base-band. A complex signal can be broken down into cosine and sine components, resulting in separate in-phase and quadrature components.

In typical terrestrial, marine and aeronautical applications, the Doppler shift can vary over a $\pm$ 10kHz range about the intermediate frequency. If one wants to implement a phase-locked loop, then the frequency of the mixing signal must be controllable to within a few millihertz. Furthermore, the mixing signal must have a continuously varying phase.

In a hardware correlator, local oscillators generate cosine and sine signals that have precise frequency control and a continuous phase. This strategy is not feasible for a software correlator. Generating cosine and sine signals on the fly with the correct frequency and phase would be too time consuming. Instead, the software correlator generates cosine and sine signals on a grid of frequencies off-line. These signals are stored in memory for later recall.

A strategy is needed in order to minimize the number of sine and cosine signals that must be stored. The signals must be stored on a time grid of points sampled at the RF front-end sampling frequency of 5.714 MHz, and the signals must last for a C/A PRN code period, i.e., for 0.001 sec. It would take tens of gigabytes of memory or more in order to brute-force store all frequencies on a 1 mHz grid ranging from -10 KHz to $+10$ KHz, not to mention the question of storing a grid of possible starting phases at each frequency point.

A method has been developed that allows the receiver to accumulate I and Q values using stored carrier replicas that fall only on a rough frequency grid and that all start with a phase of zero. The rough frequency grid has a spacing of 175 Hz, and the resulting storage requirements are on the order of 323 Kbytes. The resulting accumulations are

$$I_{gjk}(\Delta) = \sum_{i=i_k}^{i_k+N_k} y(t_i) C_j \left[0.001 \left(\frac{t_i + \Delta - \hat{\tau}_{jk}}{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}} \right) \right] \times cos[(\omega_{IF} - \omega_{gjk})(t_i - t_{0gjk})] \quad (6)$$

$$Q_{gjk}(\Delta) = - \sum_{i=i_k}^{i_k+N_k} y(t_i) C_j \left[0.001 \left(\frac{t_i + \Delta - \hat{\tau}_{jk}}{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}} \right) \right] \times sin[(\omega_{IF} - \omega_{gjk})(t_i - t_{0gjk})] \quad (7)$$

where ω_{gjk} is the grid frequency that is closest to the estimated frequency $\hat{\omega}_{Doppjk}$ and where t_{0gjk} is the time at which this carrier replica has zero carrier phase. These accumulations are then rotated in order to create accurate approximations of what would have been computed had the estimated carrier phase time history in equations (4) and (5) been used:

$$I_{jk}(\Delta) = I_{gjk}(\Delta) cos(\Delta\phi_{avgjk}) + Q_{gjk}(\Delta) sin(\Delta\phi_{avgjk}) \quad (8)$$

$$Q_{jk}(\Delta) = -I_{gjk}(\Delta) sin(\Delta\phi_{avgjk}) + Q_{gjk}(\Delta) cos(\Delta\phi_{avgjk}) \quad (9)$$

where $\Delta\phi_{avgjk}$ is the average phase difference between the grid carrier phase and the estimated carrier phase averaged over the accumulation interval:

$$\Delta\phi_{avgjk} = \omega_{gjk} \left(\frac{\hat{\tau}_{jk} + \hat{\tau}_{jk+1}}{2} - t_{0gjk} \right) - \hat{\phi}_{jk} - \hat{\omega}_{Doppjk} \left(\frac{\hat{\tau}_{jk+1} - \hat{\tau}_{jk}}{2} \right) + \omega_{IF} t_{0gjk} \quad (10)$$

The validity of equations (8) and (9) is dependent on the assumption that

$$1 - cos \left[\frac{1}{2} \left(\omega_{gjk} - \hat{\omega}_{Doppjk} \right) \left(\hat{\tau}_{jk+1} - \hat{\tau}_{jk} \right) \right] << 1 \quad (11)$$

Given a 175 Hz grid spacing and a nominal C/A PRN code period of 0.001 sec, the maximum value on the left-hand side of inequality (11) is 0.04, which respects the assumed limit.

Note that equations (8) and (9) can be derived from equations (4) and (5) as follows: First, one adds and subtracts the carrier phase of the grid signal in the arguments of the cosine and sine terms into sums of products of cosine and sine terms. Second, one uses trigonometric identities to split the cosine and sine terms into sums and products of cose and sine terms. In each product, one of the terms involves an argument like the arguments in the trigonometric terms in equations (6) and (7). The other trigonometric terms are then approximated by either $cos(\Delta\phi_{avgjk})$ or $sin(\Delta\phi_{avgjk})$. These approximations are valid because of the inequality in equation (11) and because the average of $sin(\omega_{gjk} - \hat{\omega}_{Doppjk})[t_i - \frac{1}{2}(\hat{\tau}_{jk} + \hat{\tau}_{jk+1})]$ over the accumulation interval is zero.

A decrease in C/No is expected from using an inexact frequency. The worst-case decrease is expressed as a function of the frequency grid spacing Δf and is given by

$$\Delta SNR = 20 log_{10} \left(\frac{sin(\pi \Delta f T)}{\pi \Delta f T} \right) \quad (12)$$

where Δf is in units of Hz, and T is the integration period. Thus, a Δf of 175 Hz causes a worst-case SNR loss of 0.44 dB for $T = 0.001$ sec.

The cosine and sine signals on the grid are stored with a 2-bit binary sign and magnitude representation. The format of this representation is defined in Table 2. This format assumes that the cosine and sine signals have an amplitude of 2.4. Figure 3 shows how to sample a sine wave to generate the optimal 2-bit representation, that has the minimum least square error.

Sign	*Mag*	*Value*
0	0	-1
0	1	-2
1	0	+1
1	1	+2

Table 2: *Sign and magnitude combinations of the stored intermediate-frequency carrier sine wave.*

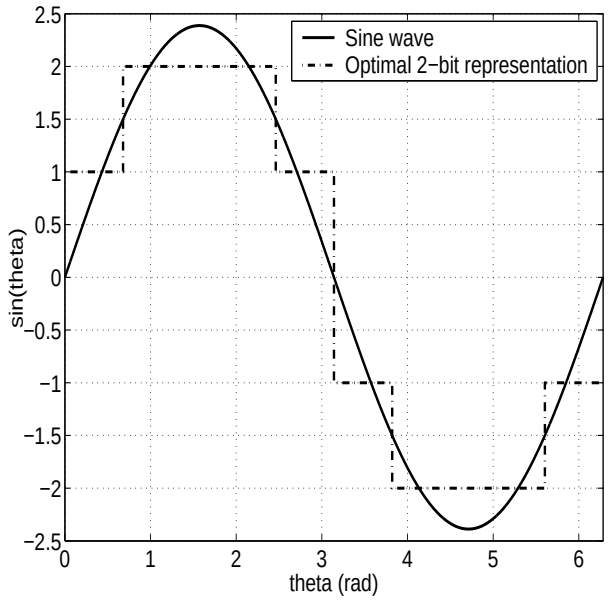

Figure 3: *Illustration of how to sample a sine wave for a 2-bit representation.*

Sign	*High Mag*	*Low Mag*	*Value*
0	0	0	+1
0	0	1	+2
0	1	0	+3
0	1	1	+6
1	0	0	-1
1	0	1	-2
1	1	0	-3
1	1	1	-6

Table 3: *Sign, high-magnitude, and low-magnitude combinations of the base-band mixed signal.*

A simple EXCLUSIVE OR multiplication of sign bits and a redefinition of data bits accomplishes base-band mixing. Multiplication of the RF front-end output representation of Table 1 by the sine wave representation of Table 2 yields a result that can take on the values −6,−3,−2,−1,+1,+2,+3, and +6. These can be represented by 3 bits according to the scheme in Table 3. The high magnitude bit of Table 3 is simply the magnitude bit of the RF front-end output from Table 1, and the low magnitude bit of Table 3 is the magnitude bit of the base-band mixing sine wave from Table 2. Thus, these two magnitude bits are available without the need for computation. The sign bit can be computed by executing an EXCLUSIVE OR operation between the sign bits of the Table-1 RF front-end data and those of the Table-2 base-band mixing signal data. Notice how the sign bit value's relationship with the actual sign gets reversed from that of Tables 1 and 2.

Mixing of the Base-Band Signal with a Local C/A Code

Both prompt and early-minus-late correlations are needed to track the carrier frequency, carrier phase, and code phase in a GPS receiver. The prompt correlations are defined by equations (4) and (5) with $\Delta = 0$. The early-minus-late correlations are $I_{jk}(\Delta_{eml}/2) - I_{jk}(-\Delta_{eml}/2)$ and $Q_{jk}(\Delta_{eml}/2) - Q_{jk}(-\Delta_{eml}/2)$, where Δ_{eml} is the spacing between the early and late PRN code replicas.

A hardware correlator generates in real-time a particular C/A code replica at the correct Doppler shifted frequency and phase. This approach is too time consuming in a software correlator. Instead, it is better to generate the C/A codes off-line and store the C/A code replicas in a memory table, called the PRN code table. Storing all 32 C/A codes on a 2-dimensional grid of possible phases and Doppler shifts would require a large amount of memory, on the order of several gigabytes.

The required amount of storage can be greatly reduced by making simplifications. First, the prompt code is stored as a single sign bit. This representation is shown in Table 4. The early-minus-late code, on the other hand, is stored in a two-bit representation (actually a 1.5 bit representation). It has a sign bit and a zero-mask bit, as denoted in Table 5.

Sign	*Value*
1	+1
0	-1

Table 4: *Sign bits of the prompt C/A code.*

The next simplification in the PRN code table is to ignore code Doppler shift variations. All signals in the table are assumed to have zero Doppler shift; i.e., all C/A codes in the table assume that $\hat{\tau}_{jk+1} - \hat{\tau}_{jk} = 0.001$ sec. The code phase errors due to this assumption are eliminated by choosing a replica code from the table whose midpoint

Sign	*Zero Mask*	*Value*
X	0	0
0	1	-2
1	1	+2

Table 5: *Sign and zero mask combinations of the stored early-minus-late C/A code replica.*

occurs at the desired midpoint time $(\hat{\tau}_{jk} + \hat{\tau}_{jk+1})/2$. The only other effect of this assumption is a small correlation power loss, which is no more than 0.014 dB if the magnitude of the Doppler shift is less than 10 KHz.

The PRN code table must include a selection of phases as measured relative to the sample times of the RF front-end outputs. The particular RF front-end that has been used has a sample spacing of 175 nsec. The PRN code table includes 14 different phases with respect to these sample times. This translates into a code phase spacing of 12.5 nsec, which equals a pseudorange measurement digitization level of 3.8 m. Thus, the maximum measurement error is half of this digitization level, or 1.9 m.

The prompt and early-minus-late C/A code replicas can be mixed with the base-band code by bit re-definitions and a simple EXCLUSIVE OR operation. Suppose that one has a 3-bit base-band signal that is represented as in Table 3 and a prompt replica of the C/A code as represented in Table 4. Then the product of the two signals can be found by forming the EXCLUSIVE OR of the two inputs' sign bits to produce the sign bit of the 3-bit representation given in Table 6. The high and low magnitude bits of this mixed signal equal the high and low magnitude bits of the base-band signal from Table 3. Note that the Table 6 representation is identical to that of Table 3 except for the inversion in the meaning of the sign bits.

The mixing of the early-minus-late code with the base-band signal is also accomplished by an EXCLUSIVE OR operation on the two signals' sign bits in conjunction with a transcription of the high magnitude, low magnitude, and zero mask bits. The resulting representation is given in Table 7.

Sign	*High Mag*	*Low Mag*	*Value*
0	0	0	-1
0	0	1	-2
0	1	0	-3
0	1	1	-6
1	0	0	+1
1	0	1	+2
1	1	0	+3
1	1	1	+6

Table 6: *Sign, high-magnitude, and low-magnitude combinations of the fully mixed prompt integrand.*

Sign	*High Mag*	*Low Mag*	*Zero Mask*	*Value*
X	X	X	0	0
0	0	0	1	-2
0	0	1	1	-4
0	1	0	1	-6
0	1	1	1	-12
1	0	0	1	+2
1	0	1	1	+4
1	1	0	1	+6
1	1	1	1	+12

Table 7: *Sign, high-magnitude, low-magnitude, and zero-mask combinations of the fully mixed early-minus-late integrand.*

Bit-Wise Parallel Storage and Accumulations of Correlations.

One can exploit the simple representations of signals in terms of 1 to 4 bits by using bit-wise parallelism to perform the necessary calculations. Bit-wise parallel operations work with representations of the data that store successive samples in successive bits of a word. For example, 32 samples of the RF front-end output are stored in 2 32-bit words. One word stores the 32 sign bits of the 32 samples, and the other word stores the 32 magnitude bits. The stored tables of the base-band mixing cosine and sine waves have their sign and magnitude bits stored in separate words, with each 32-bit word storing 32 sign or magnitude bits that tabulate to 32 successive samples of the corresponding cosine or sine wave. Similarly, the stored tables of the prompt and early-minus-late codes store sign or sign and zero-mask bits in words with each word storing 32 samples worth of data. By this means the EXCLUSIVE OR operations that are involved in mixing operate on 32 samples at a time because the processor has a bit-wise EXCLUSIVE OR command and other bit-wise

commands that operate in parallel on each of two input arguments' 32 bit pairs.

The final operation in the correlation calculations is to sum the results over all of the samples in a given estimated PRN code period. This operation requires additional bit-wise parallel operations followed by operations that form totals over the bits in a given word. This approach starts by performing bit-wise parallel Boolean logic for each of the 8 possible values in the right-hand column of the prompt integrand representations in Table 6. A 32-bit value word is computed for each 32 samples and each row of Table 6. It contains ones for the sample times when the actual integrand equals the corresponding value in the right-hand column of Table 6 and zeros for the remaining times when the actual integrand does not equal this value. The 8 value words corresponding to the 8 possible Table-6 values are formed as follows:

$$MINUSONE = NOT(SIGN)\ AND\ [NOT(HIGHMAG)\ AND\ NOT(LOWMAG)] \quad (13)$$

$$MINUSTWO = NOT(SIGN)\ AND\ [NOT(HIGHMAG)\ AND\ LOWMAG] \quad (14)$$

$$MINUSTHREE = NOT(SIGN)\ AND\ [HIGHMAG\ AND\ NOT(LOWMAG)] \quad (15)$$

$$MINUSSIX = NOT(SIGN)\ AND\ [HIGHMAG\ AND\ LOWMAG] \quad (16)$$

$$MINUSONE = SIGN\ AND\ [NOT(HIGHMAG)\ AND\ NOT(LOWMAG)] \quad (17)$$

$$MINUSTWO = SIGN\ AND\ [NOT(HIGHMAG)\ AND\ LOWMAG] \quad (18)$$

$$MINUSTHREE = SIGN\ AND\ [HIGHMAG\ AND\ NOT(LOWMAG)] \quad (19)$$

$$MINUSSIX = SIGN\ AND\ [HIGHMAG\ AND\ LOWMAG] \quad (20)$$

These operations can be carried out in 15 binary operations if one takes advantage of redundancy by storing common intermediate results.

The operations for the early-minus-late integrand are similar. All of the values double in this case, i.e., the MINUSSIX word becomes the MINUSTWELVE word. Also, an additional AND operation must be performed with the ZERO MASK bits of Table 7 in order to mask out sample times when the early and late PRN codes cancel each other. If one takes advantage of the fact that the early-minus-late HIGHMAG and LOWMAG words are the same and if one ANDs the zero mask words with the SIGN and NOT(SIGN) words before ANDing the results with the HIGHMAG and LOWMAG results, then the early-minus-late integrands can be computed at a cost of only 11 additional binary operations.

Additional zero-masking occurs in the first and last words of an accumulation interval. This is true because the start and stop times of an accumulation interval do not normally fall at the boundaries of data words. Therefore, the bits in the first word that precede the accumulation interval need to get zero-masked as do the bits in the last word that come after the end of the accumulation interval.

The accumulation operation must sum the number of 1 bits in each of the 8 value words. These are no such summation operations in a standard microprocessor's instruction set. Therefore, the summations are accomplished using a table look-up. The value word is used as the address in the memory table, and the table's output is set up to deliver the number of 1 values in the address. A 16-bit table has been used. This gives it a memory size of 2^{16} or 64 Kbytes, which makes it able to fit into the microprocessor's cache and allows for very fast execution. Suppose that this operation is called BITSUM. Then it can be used to compute the accumulation in equation (6) as follows:

$$
\begin{aligned}
I_{gjk}(0) = 6 * \sum_{l=1}^{N_w} [&BITSUM(PLUSSIX_{jl}) \\
&- BITSUM(MINUSSIX_{jl})] \\
+3 * \sum_{l=1}^{N_w} [&BITSUM(PLUSTHREE_{jl}) \\
&- BITSUM(MINUSTHREE_{jl})] \\
+2 * \sum_{l=1}^{N_w} [&BITSUM(PLUSTWO_{jl}) \\
&- BITSUM(MINUSTWO_{jl})] \\
+ \sum_{l=1}^{N_w} [&BITSUM(PLUSONE_{jl}) \\
&- BITSUM(MINUSONE_{jl})]
\end{aligned}
\tag{21}
$$

The prompt quadrature and early-minus-late in-phase and quadrature accumulations can be computed using the same operations, but with differing value words that correspond to their respective integrands.

Computation Time Savings

The bit-wise parallel operations save computation time in comparison to integer mathematical correlation operations. Integer mathematics requires 6 multiplications and 4 additions per sample (except for the last sample) in order to form the 4 required accumulations for each channel. At a sampling rate of 5.714 MHz this translates into 57136 integer operations per PRN code period. The bit-wise parallel method uses mostly simple logic and table look-up operations in order to form the 4 accumulations. It uses 6 EXCLUSIVE OR operations and 52 additional bit-wise logic operations per word. It uses 32 bit summation operations, and 32 additions per summation word (actually, it only requires 16 summations for the last word). Suppose that the nominal word length is 32 bits but that the summation words are only 16 bits long. Then there are 180 words and 360 summation words in a typical accumulation interval. If one totals the necessary operations along with some overhead that occurs at the first and last words, then the new method requires 33496 operations per PRN code period. Thus, there is a savings of almost a factor of two in the operation count. The bit-wise method's logic and table look-up operations may execute more rapidly than multiplication operations on a typical micro-processor, which would further increase the time savings. Additional speed-up may come about because of a reduced number of accesses to non-cache memory. The net speed-up is a factor of about 2.1 as measured on a 1.73 AMD Athlon GHz processor.

Note that this algorithm can be adapted to work with a different number of bits in the representation of the RF front-end output and of the cosine and sine mixing signals. An increase above 2 bits will make the logic more complex and will decrease the time savings versus straight integer arithmetic. A decrease to a 1-bit representation will do the opposite. For example, if the RF front-end uses one-bit digitization rather than two-bit digitization, then the operation count will decrease by a factor of almost 2 for the new method, which will make it about 4.2 times faster than straight integer arithmetic.

Another method of creating the carrier replicas exists. This method adds a small computational slow-down to the software, but reduces the number and length of signals stored. Instead of storing a millisecond of the carrier signals on a coarse grid of frequencies, it is possible to store only the values of cosine and sine over a period of 0 - 2π. Then, to generate the carrier replica, one needs to compute the argument of the cosine and sine functions in equations (2) and (3). The 2π modulus of the argument is then used as the index into the stored cosine and sine signals. This step adds the extra requirement of computing the argument in real-time, however, as compared to storing the signals ahead of time. If the cosine and sine arguments are specified as floating-point numbers, these computations produce a significant slow-down. If the arguments are expressed as 64-bit integers, on the other hand, the overall computational time of the integer-based correlation algorithms frows by less than 5% in comparison to the method that uses a pre-computed grid of carrier replicas. This method of generating the cosine and sine signals is not easily implemented in the bit-wise algorithms because of the additional cost of packing the representations into bit-wise parallel words after they get computed.

Storage Requirements

The pre-computed base-band mixing signals and PRN codes require a certain amount of memory. Each replica signal must occupy 180 32-bit words in order to be guaranteed to cover the full 5714 RF front-end samples that occur in one PRN code period for any possible code period start time within the 32 samples of the initial word. Thus, 180*4=720 bytes are required for each bit of each signal that must get stored. The sine and cosine waves each have two-bit representations, which translates into a storage requirement of 2880 bytes for the carrier replicas at a given Doppler shift. There are 115 Doppler shifts that must be stored in order to cover the -10 KHz to $+10$ KHz range with a 175 Hz grid spacing. This translates into 323 Kbytes of storage for all of the carrier replica signals.

The pre-computed PRN codes also require a significant amount of storage. The prompt code has a 1-bit representation, and the early-minus-late code has a 2-bit representation. This translates into a total of 2160 bytes for a single code phase of a single PRN number. The table must include 14 different code phases per RF front-end sample multiplied by 32 RF front-end samples per word, which yields a storage requirement of 945 Kbytes per PRN code and 30 Mbytes for all 32 PRN codes.

Note that it is possible to reduce these storage requirements by a factor of 32 if one does not store different code replicas for the 32 different possible locations within a data word of the first RF front-end sample at an accumulation interval. The memory savings comes at the cost of additional bit-shifting operations that are needed in order to 1-bit align the code replica's start bit with the estimated start bit in the incoming data word stream. Experience with the 1.73 GHz AMD Athlon processor indicates that this added computational cost is minimal.

The micro-computer stores the most recent 21 msec of RF front-end data in a circular buffer. This allows it to process the differing code periods for different satellites during different iterations of a regularly scheduled program thread. This buffer occupies 30 Kbytes of memory.

Code, Carrier Phase, and Carrier Frequency Measurements

Navigation calculations require measured values of the PRN code phase, carrier phase, and carrier frequency. The measurements for each satellite must occur at the exact same time. The TIC function provides a periodic timing scheme to synchronize these measurements at time t_{TIC}. At time t_{TIC} the TIC function latches all of the C/A code phases, carrier phases, and carrier frequencies along with the code epoch counters, and it makes these available to the remaining GPS receiver software. The GPS receiver uses the code phase and epoch counters to compute the pseudorange to each satellite.

The software correlator keeps track of the code and carrier phase of each signal as determined by the code chipping rate and the carrier Doppler shift inputs. Suppose that $\hat{f}_{cjk}$ is the receiver's estimated code chipping rate for satellite j during its k^{th} PRN code period and suppose that $\hat{\omega}_{Doppjk}$ is the associated carrier Doppler shift. $\hat{f}_{cjk}$ will have been determined either by an acquisition search procedure, or if tracking, by a delay-locked loop. Likewise, $\hat{\omega}_{Doppjk}$ will have been defined by an acquisition procedure or, if tracking has commenced, by a phase-locked loop or a frequency-locked loop. The software correlator uses these two quantities to update its self initialized code and carrier phases according to the formulas:

$$\hat{\tau}_{jk+1} = \hat{\tau}_{jk} + \frac{1023}{\hat{f}_{cjk}} \tag{22}$$

$$\hat{\phi}_{jk+1} = \hat{\phi}_{jk} + \hat{\omega}_{Doppjk+1}(\hat{\tau}_{jk+1} - \hat{\tau}_{jk}) \tag{23}$$

In this software receiver t_{TIC} occurs at the millisecond boundaries. At each time t_{TIC} the code phase of each signal is computed in the following manner (referring to Figure 4):

$$\hat{\psi}_{jTIC} = 1023\left(\frac{t_{TIC} - \hat{\tau}_{jk+1}}{\hat{\tau}_{jk+2} - \hat{\tau}_{jk+1}}\right) \tag{24}$$

where $\hat{\psi}_{jTIC}$ is the code phase in chips of signal j at time t_{TIC}. The epoch counters, which are simply a running total of the number of code periods, are incremented at each code start/stop time.

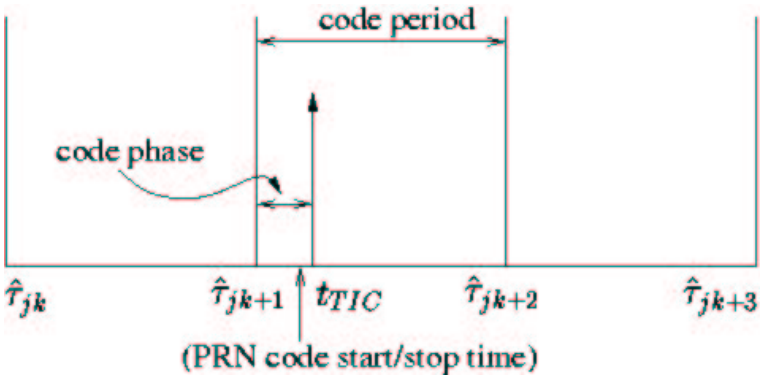

Figure 4: *A schematic diagram illustrating the code phase measurement.*

The carrier phase calculation at time t_{TIC} is similar to the code phase calculation:

$$\hat{\phi}_{jTIC} = \hat{\phi}_{jk+1} + \hat{\omega}_{Doppjk+1}(t_{TIC} - \hat{\tau}_{jk+1}) \tag{25}$$

where $\hat{\phi}_{jTIC}$ is the carrier phase at time t_{TIC} of Figure 4. The Doppler shift that gets returned at this time is $\hat{\omega}_{Doppjk+1}$.

Fixed-Point Computations

Real-time software is fastest when using fixed-point computations. Floating-point operations, such as addition, multiplication, and division, take much longer than their integer equivalents. For example, on an AMD Athlon one floating-point division takes 8 clock cycles while a fixed-point division takes 4-5 clock cycles. Furthermore, floating-point variables require more space in memory than do integer variables. Thus, it is worthwhile to carry out the majority of the calculations in an integer-based format. The mixing and correlations are already in integer format, but computations like calculating the code start/stop times and the angle of rotation for I and Q are inherently floating-point calculations.

No process exists for blindly converting floating-point computations into fixed-point equivalents. Any reasonable process includes determining the maximum values and the desired minimum resolution of the calculations. These figures help to determine the required sizes of the integers. Maximum values are important because many parameters, such as the code start/stop times, increase continually over time, which makes overflow a concern. Resolution is important because the required accuracy of a computation must be maintained when using an integer format.

As an example, consider the C/A code start/stop times. The first step is to define how precise the times must be. This depends on how precisely the pseudorange must be measured. Assume that the pseudorange must be measured to within 0.5 meters or $\cong 1.667$ nsec. The maximum value for a 32-bit unsigned integer is $2^{32} \cong 4 * 10^9$, which implies that a 32-bit representation of the code start/stop times would overflow in 7 seconds of operation. Since the code start/stop times continually increase over time, a 64-bit unsigned integer is more appropriate. A 64-bit integer also allows for an increase in the precision of the start/stop times. A good compromise between precision and avoidance of overflow is to count start/stop times in units of 100's of picoseconds. In this case overflow will occur after 58.5 years of continuous operation.

Converting the I and Q rotation angle into a fixed-point equivalent is more complicated. This is so because of the continually growing nature of the carrier phase angle and because of the large intermediate frequency that gets multiplied by a time in the last term of equation (10). In the former case, however, modulo arithmetic is useful since $\Delta\phi_{avgjk}$ is modulo 2π, and thus the terms that compose it can be computed modulo 2π.

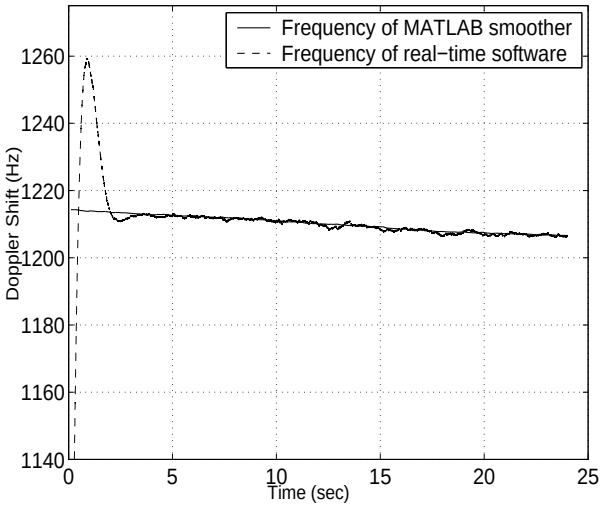

Figure 5: Frequency response of the real-time software receiver's FLL compared with that of a MATLAB smoother.

Performance Results

A sample screen-shot from the real-time software receiver is provided in Figure 8. This figure shows the receiver tracking 9 channels. The antenna used is an L1 antenna with a pre-amp that has 26 dB of gain and is roof-mounted on Rhodes Hall at Cornell University. The receiver has a position accuracy on the order of 10-15 meters, which is is comparable to other receivers that use hardware correlators.

The tracking loop performance of the software receiver code has been evaluated by comparing it to a software receiver implemented in MATLAB that uses smoother-based carrier and code tracking loops and that operates on the same data in an off-line mode. Figure compares the Doppler shift of the carrier from the real-time software receiver with that of the MATLAB smoother. The mean frequency error deviation after the transient period is less than 2Hz. Thus, the real-time software receiver's FLL functions properly with the software-computed accumulations.

It is important to compare the tracking and navigation performance between the software receiver and a receiver that uses a hardware correlator. A reciever that uses the Mitel GP2021 digital hardware correlator was used for the comparison. This receiver uses the same Cascade GPS software as the software receiver. Both receivers also use the Mitel GP2015 RF front-end. The receivers are set up to run at the same time and are connected to the same roof-mounted antenna. Performing a simple side-by-side visual comparison shows that SNR values differ by less than 1 dB and that the navigation solutions differ by no more than 5-10 meters.

An important measure of the efficiency of the algorithms used in the software correlator is the average duration per millisecond required to base-band mix and correlate 12 channels. The current software correlator requires 390 microseconds, which is a 39% duty cycle. However, the algorithms are not optimized. We are aware of numerous increases in speed that will most likely reduce this value by about 5-10%. Furthermore, as faster PC's become available, the processing time will continue to decrease.

It is important to compare the software receiver presented in this paper with the one presented in *Akos et al.* [2001a]. A proper comparison gives a sense of the efficiency of the algorithms implemented in this receiver. *Akos et al.* [2001a] show a plot of the required computation time for their 6-channel software receiver to process 1 second of GPS data as a function of x86 microprocessor speed. From the plot, the computation time of the receiver at an RF front-end sampling frequency of 5.714 MHz on a 1.73 GHz processor is about 1 second. The software receiver discussed in this paper processes 6 channels in about 0.23 sec. *Akos et al.* [2001b] mentions a potential speed improvement that may decrease the computation time by 30%. Taking this into consideration, the software receiver described in this paper is still over 3 times faster.

Two different bit-grabbers have been tested with the real-time software receiver. The first one uses an analog down conversion scheme, while the other one implements a direct ADC down conversion. Previously, *Akos and Tsui* [1996] presented an implementation of a direct ADC down conversion GPS front-end. To evaluate the front-end, they stored and off-line processed 3 msec of GPS data. In contrast, the direct ADC front-end discussed in this paper has been tested with the real-time software receiver

Lat	42.44354	Spd	0.5	SVs	8	CTrack	FLL	Date	17/10/02
Lon	-76.48143	Hdg	327.0	Nav	3D	GDOP	1.9	GPS	19:58:11
Alt	269.6560	ROC	-0.7	HI ELEV		DO	-393.0	OscErr	0.25

CH	*SV*	*ELV*	*AZI*	*DOPP*	*NCO*	*UERE*	*SF*	*PRerr*	*PRRerr*	*LOCK*	*SNR*	*iS4*
1	1	67	237	-528	-927	4	1	9.0	0.7	CCBF	16.9	-1.000
2	22	64	47	-1644	-2045	4	1	14.3	0.6	CCBF	18.0	-1.000
3	3	50	152	2174	1778	2	1	5.5	-0.4	CCBF	18.4	-1.000
4	25	22	106	-2650	-3050	2	1	42.9	0.2	CCBF	13.2	-1.000
5	17	1	62	1722	1331	2	1	7.5	-0.4	CCBF	8.1	-1.000
6	15	2	81	2278	1887	2	1	-5.2	-0.7	CCBF	7.4	-1.000
7	27	12	295	2969	2575	0	1	0.0	0.0	CCBF	8.0	-1.000
8	13	44	303	1856	1866	2	0	0.0	0.0	C	14.9	-1.000
9	31	22	185	3860	3464	2	1	-11.0	-0.5	CCBF	15.9	-1.000
10	–	–	–	–	–	–	-	–	–		–	–
11	–	–	–	–	–	–	-	–	–		–	–
12	20	4	219	-3086	-3483	2	1	27.5	0.3	CCBF	9.6	-1.000

Table 8: *Screenshot of a the software GPS receiver.*

and ran continuously for several hours. The performance results are similar to those of the analog RF front-end.

Summary and Concluding Remarks

A 12-channel real-time software GPS L1 receiver that runs on a common PC has been implemented and tested. The hardware consists of an RF bit-grabber card, a data acquisition system, and a PC with a 1.73GHz AMD Athlon processor running RT-Linux. The software consists of the data acquisition code, the software correlator, and GPS software that provides the typical GPS functions such as navigation and tracking. The software correlator, running on the PC's processor, consumes about 39% of the CPU capacity, leaving the PC time to perform other tasks. Furthermore, optimizations exist that may decrease the CPU usage by about 5-10%.

The software correlator algorithms have been tested in depth. They have been compared to both a hardware correlator and a non-real-time software receiver implemented in MATLAB. These comparisons show that real-time software correlation can be implemented without loss of performance.

ACKNOWLEDGEMENTS

Research at Cornell University was funded by the Office of Naval Research under grant N00014-92-J-1822.

REFERENCES

1. Akos, D.M. and J.B.Y. Tsui, Design and Implementation of a Direct Digitization GPS Receiver Front End, *IEEE Trans. on Microwave Theory and Techniques, 44*(12), 1996, pp. 2334-2339.

2. Akos, D.M., P.-L. Normark, P. Enge, A. Hansson, and A. Rosenlind, Real-Time GPS Software Radio Receiver, *Proc. of the Institute of Navigation National Technical Meeting*, Long Beach, CA, January 22-24, 2001, pp. 809-816.

3. Akos, D.M., P.-L. Normark, A. Hansson, A. Rosenlind, C. Stahlberg, and F. Svensson, Global Positioning System Software Receiver (gpSrx) Implementation in Low Cost/Power Programmable Processors, *Proc. of the Institute of Navigation National Technical Meeting*, Salt Lake City, UT, September 11-14, 2001, pp. 2851-2858.

4. Ledvina, B.M., F. Mota, and P.M. Kintner, A coming of age for GPS: A RTLinux GPS receiver, Workshop on Real Time Operating Systems and Applications, *Proc. of the Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop* (in conjunction with IEEE RTSS 2000), Orlando, FL, November 27-28, 2000.

5. Van Dierendonck, A.J., "GPS Receivers," in *Global Positioning System: Theory and Applications, Vol.*

I, Parkinson, B.W. and Spilker, J.J. Jr., eds., American Institude of Aeronautics and Astronautics, (Washington, 1996), pp. 329-407.