Journal of Global Positioning Systems (2008)
Vol. 7, No. 2 : 156-168

Journal of Global Positioning Systems
2008
© CPGPS

Architecture and Benefits of an Advanced GNSS Software Receiver

Mark G. Petovello, Cillian O'Driscoll, Gérard Lachapelle, Daniele Borio and Hasan Murtaza
Position, Location And Navigation (PLAN) Group, Department of Geomatics Engineering
University of Calgary

Abstract

This paper describes a GNSS software receiver architecture and the associated benefits in terms of algorithm flexibility and processing efficiency. For the latter, different signal processing algorithms and implementations are considered including processing with a Graphics Processing Unit (GPU); a novel implementation in the GNSS community. The massively parallel processing capability of the GPU is demonstrated relative to other processing optimizations. Sample results of GPS processing are presented including centimetre level positioning. Results obtained with some of the Galileo and GLONASS signals are also included to demonstrate the flexibility of the receiver.

Key words: GNSS, Software Receiver

1. Introduction

Software-based GNSS receivers have been receiving considerable attention in the past several years. Not only do such receivers provide an excellent research tool for investigating and improving GNSS receiver performance in a wide range of conditions, they are also gradually becoming commercially viable, with some companies having already released products to the market (IFEN 2007, Morton 2007, NXP 2007, Scott 2007, CSR 2008, Fastrax 2008). The above advantages are further highlighted by the proliferation of new systems and signals. In contrast, the primary drawback of software receivers is the computational requirements needed to implement the receiver in the first place. In particular, with GNSS sampling rates generally exceeding 4 Msps (samples per second), processing requirements are indeed extreme for the receiver's signal processing operations.

The major objective of a software receiver is therefore to efficiently implement the high rate computations while maintaining the desired flexibility inherent in a software-based approach. Unfortunately, these two objectives are generally at odds with an improvement in one aspect often occurring at the expense of the other. Traditional "hardware-based" GNSS receivers can be viewed as an extreme example of this where the most computationally intense processing is performed using very efficient hardware (i.e., application specific integrated circuits, or ASICs) which is inherently inflexible.

This paper discusses the general design, implementation and testing of a software-based GNSS receiver that addresses the above challenges. The software GSNRx™ (GNSS Software Navigation Receiver) was developed in C++ and is flexible enough to allow for a wide range of configurations involving different processors, receiver architectures, and acquisition and tracking strategies. With this in mind, the objectives of the paper are two-fold; first, to describe and rationalize the general architecture of the software, and second, to show some sample results obtained with the receiver.

There are two main contributions of the work. First, by presenting the overall software architecture and the underlying motivation for it, it is hoped that readers will gain some insight into the practical implementation issues regarding software receivers. Second, the implementation of a Graphics Processing Unit (GPU) for data processing is presented as a means of improving processing efficiency, even with high sample rates. To the authors' knowledge, this is the first time such an implementation has been used for GNSS software receivers.

The paper begins with a general overview of the software receiver architecture and its corresponding benefits in terms of processing efficiency and algorithm flexibility. The paper discusses how different processors can be incorporated into the receiver and the benefits realized. For example, the receiver can be configured to use a "pure software" approach, or, if available, any other co-processors such as an FPGA (Field Programmable Gate Array) or a GPU (Graphics Processing Unit). The latter is discussed in detail, as this represents a novel implementation for software receivers. It is also demonstrated how the choice of processor can be optimized by making use of any suitable instruction sets available on Intel processors. Following the description of the software, some sample results will be presented that demonstrate the software's capability.

2. GNSS Receiver Methodology

This section describes the basic GNSS receiver methodology, as it applies to the software architecture described in this paper. Algorithm details are available in the cited references. Alternatively, several references on GNSS signal processing in general are available in the public literature including, for example, Van Dierendonck (1995), Misra & Enge (2001), Ma et al (2004), Tsui (2005), Ward et al (2006)and Borre et al (2007). Sample results for more advanced receiver architectures are presented later on, but these architectures are not discussed in detail and the reader is referred to the cited material for more information.

GNSS signal tracking is achieved by generating a local signal within the receiver that matches the incoming signal as closely as possible. This process can be roughly broken down according to Fig. **1** (the acquisition process is roughly similar but some components are omitted for clarity). The different color boxes correspond to the rate at which the operations are performed, as described below. The signal is received at the antenna and is down converted to a lower intermediate frequency (IF) and sampled in the front-end. The front-end and antenna are the only hardware that is strictly necessary in a GNSS receiver. The samples are then passed to any number of individual channels in parallel, each of which is responsible for tracking a given signal that involves Doppler removal and correlation (DRC) also called baseband mixing and de-spreading – tracking error determination and updating of the local signal generator. The remaining steps include the extraction of the navigation data bits (if present on the signal), measurement generation and computation of the navigation solution.

The largest challenge associated with software based GNSS receivers is the computational requirements. To this end, the various operations are divided into high, medium and low rate categories (respectively denoted as red, blue and green boxes in Fig. **1**). In this context, high rate refers to operations performed at the MHz level; typically 4-50 MHz. Medium rate operations are generally performed at a rate of 50-1000 Hz. Low rate operations are generally performed at 20 Hz or less. The various operations are discussed briefly in the following sub-sections.

High Rate Operations

These operations are performed at the sampling rate of the incoming data; typically at 4 Msps or higher. The local signal generation involves computing (i) the sine and cosine of a carrier wave at a particular frequency with a particular starting phase, and (ii) the ranging code starting from a particular code phase. To minimize processing requirements, the sine and cosine of the carrier signals are often generated beforehand and stored in memory for later use (e.g., Ledvina et al 2003, Petovello & Lachapelle 2008). The ranging code may also be computed ahead of time, but is often computed online. It is noted that in most cases, several code phase-shifted versions of the ranging code are required for tracking.

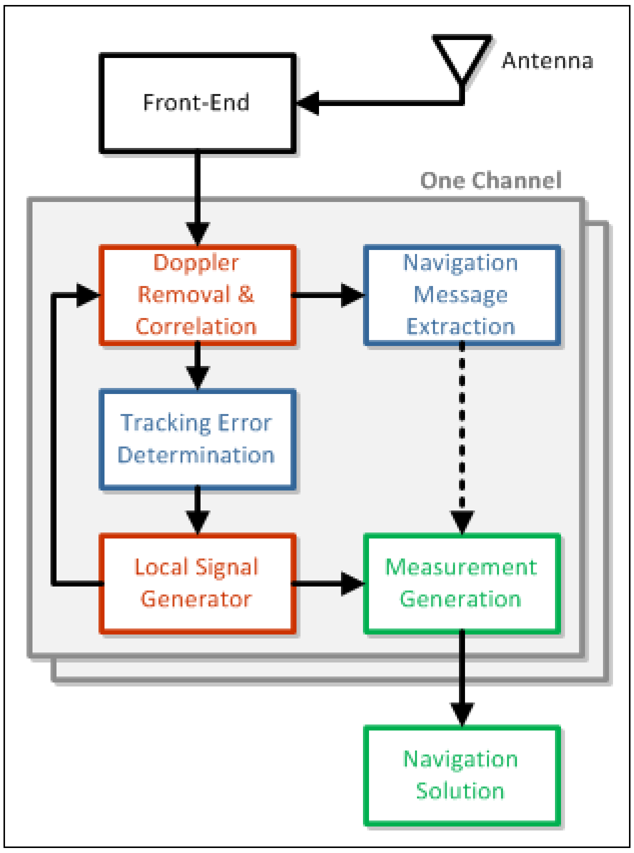

Fig. 1 General overview of GNSS signal tracking (boxes in red, blue and green represent processes performed at high, medium and low rates respectively)

The DRC operation requires projecting the incoming signal on to the locally generated carrier and then correlating the result with the local ranging code. Overall, this requires six multiplications and four additions per sample, per satellite, per code phase (Petovello & Lachapelle 2008). Frequency domain methods (e.g., van Nee & Coenen 1991) are also commonly used for the DRC operation; although mostly used for acquisition, they are also sometimes used for signal tracking as well (Tsui 2005).

The high rate operations, by far, represent the largest computational burden on the receiver. The software architecture should therefore allow for many different processing options in this regard, as will be discussed below.

Medium Rate Operations

The medium rate operations are well understood algorithms and are performed at rates of about 50 Hz to 1 kHz. These operations can be summarized as follows

- Tracking error determination uses a discriminator and loop filter pair to first measure the error (offset) between the incoming and local signals and then filter the result to minimize noise (e.g., Ward et al 2006). Kalman filter-based algorithms may also be implemented here (e.g., Psiaki & Jung 2002, Ziedan & Garrison 2004, Petovello et al 2008a).

- Navigation message extraction is performed only with those signals that broadcast navigation data. The entire operation can be further broken down into bit synchronization, navigation message (frame) synchronization and finally data extraction. For signals that contain a secondary code instead of a navigation message (e.g., the new GPS L5 signal), synchronization with the secondary code is akin to the bit synchronization process.

Low Rate Operations

The low rate operations involve generation of the carrier phase, carrier Doppler and pseudorange measurements and the subsequent computation of the navigation solution. Although some receivers output measurements at 100 Hz, typical rates for mass-market receivers are closer to 1 Hz.

Except in the case of vector-based tracking or ultra-integration with inertial measurement units (IMUs) (e.g., Petovello et al 2008a), the low rate operations are performed independent of the high and medium rate operations described above.

3. Software Architecture

The GSNRx™ software was developed in C++ using a highly modular object-oriented approach. The software was originally written to acquire and track GPS L1 C/A code signals but has since been modified to track many other signals and to use more advanced receiver architectures (more details in the results section). Because of its class-based structure, the architecture will herein be described in terms of "objects" (i.e., instantiated classes).

General Structure

The general architecture adopted for the GSNRx™ software receiver is shown in Fig. **2**. As with Fig. **1**, the boxes refer to the rate at which the operation is performed. Although Fig. **2** is a bit of an abstraction, the basic concept holds true. Before describing the objects in more detail and discussing how they interact, a few things are worth pointing out. First, the term "programmer" is used instead of "user". This is intentional because the purpose of the software is to allow for flexibility in developing and testing various signal tracking algorithms and receiver architectures. That said, the user could effectively be given the same control as the programmer via an appropriate user interface.

The second thing to notice is that in many cases the programmer has control over what objects are created and/or how objects are created. In this way, the software favors an object composition approach. In other words, all classes that adhere to a well defined interface can be used interchangeably allowing the programmer to "create" a particular receiver implementation by simply instantiating the objects with the desired functionality (at compile time or run time, whichever is preferred).

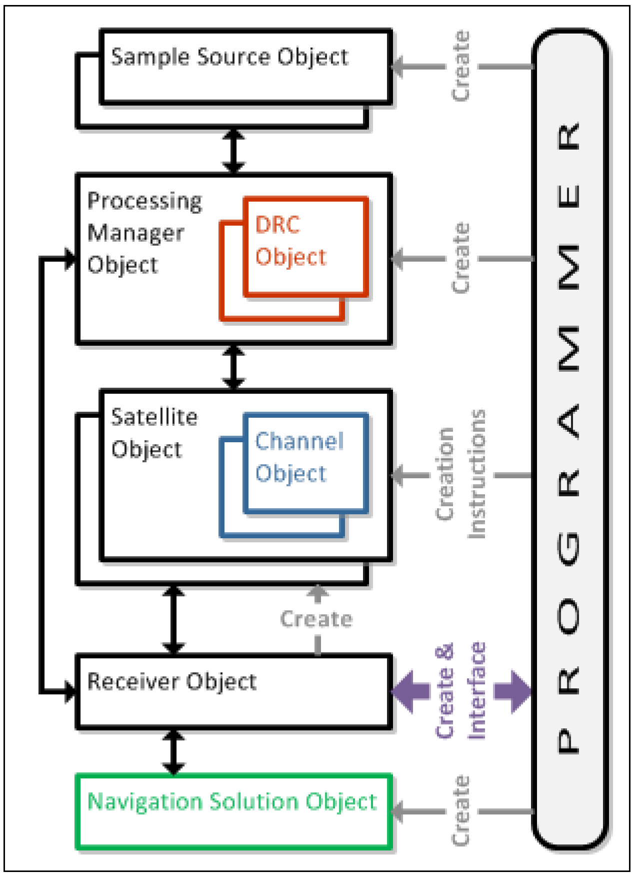

Fig. 2 General Software Architecture of GSNRx™ (colors are used to represent the rate at which each operation is performed, as per Fig. 1)

The third point of interest is that once the necessary objects are created, the programmer only interfaces with a single object— the receiver object. Not only does this improve the readability of the code, but it also simplifies the debugging process because side-effects are avoided.

The final point is that the high, medium and low rate operations are now completely separate. The importance of this will become evident as more details of the implementation are described, as below.

Object Descriptions
The main objects in Fig. **2** are described briefly below. The following section then describes how the objects interact.

Sample Source: A general repository of IF data samples within a given frequency band. The samples can be either real or complex and may be obtained from any practical source (e.g., read from file in post-mission or loaded directly from an analog to digital converter in real-time).

Signal Object: Although not shown in Fig. **2**, a signal is described by its carrier frequency and a ranging code. An example of a signal would be the GPS L1 C/A code, the Galileo E1b code or Galileo E1c code.

Channel Object: A channel is an object that is solely responsible for tracking one or more signals. The flexibility of the channel to handle a wide range of signal combinations (with some limitations) is a major advantage because it allows for more sophisticated tracking algorithms such as data/pilot combining (e.g., Mongrédien et al 2006, Muthuraman et al 2007, Muthuraman et al 2008) or multi-frequency tracking (e.g., Gernot et al 2008a, Gernot et al 2008b). The inputs into the channel are the correlator outputs from the DRC objects (described below).

Satellite Object: A satellite contains one or more channel objects. Satellite objects are responsible for handling satellite-specific information (e.g., different ephemeris messages from different channels). Satellites are created by the receiver on an as-needed basis.

DRC Object: This is an object that performs the DRC operations for a given signal. The algorithm used for this purpose is not defined (e.g., time-domain or frequency-domain, etc.) so long as the interface specifications are met. To this end, the input to the DRC is the sample source and the corresponding signal information from the channels. The outputs are the desired correlator values (more details below).

Processing Manager: The role of the processing manager is to manage the relationships between the channels, signals and sample sources. In so doing, the processing manager has the ability to determine what DRC objects are used for processing. This has major advantages as it allows for highly optimized processing to take place without any modifications to the rest of the code. Several examples of this will be presented later.

Navigation Solution: This object is responsible for computing the position, velocity and time solution (along with any other parameters of interest), typically using least-squares or Kalman filtering estimation algorithms. The navigation solution may also incorporate other sensor information, such as from an IMU, if desired. Having the navigation solution separate from the signal processing components of the receiver (except is some advanced receiver architectures) allows different processing models to be used interchangeably. This is important if an estimation algorithm is better suited to certain applications or operational conditions.

Receiver Object: This is the class that encompasses the entire receiver functionality. As described above, composing the receiver using a variety of objects allows different functionality to be included with only minimal modifications. As stated previously, certain combinations of objects could potentially be selected by the user using an appropriate user interface. The receiver performs all of the necessary high-level operations such as determining what satellites should be acquired and tracked, maintaining the receiver time and interfacing with the user. To this end, the programmer can instruct the receiver how new satellite objects should be created. The receiver object is also responsible for the implementation of vector-based and ultra-tight receiver architectures (e.g., Petovello et al 2008a). In this case, the navigation solution is used to drive the local signal generator directly; the difference between the architectures being that that ultra-tight receiver incorporates an IMU in the navigation solution (see details of navigation solution object) whereas the vector-based receiver does not.

Object Interaction
The general flow of the software is to first create the necessary objects and compose the receiver. The receiver object is informed of what sample sources are available and is also given access to the processing manager. The receiver then creates (allocates) satellite objects as needed based on assumed satellite visibility. As satellite objects are created, information about their channels (and corresponding signals) are passed to the processing manager, which, as described above, is responsible for maintaining the relationships amongst the sample sources, signals and DRC objects. Similarly, as satellites are removed (e.g., because they fall below the local horizon), they are removed from the receiver and the processing manager is informed accordingly.

As samples become available the receiver is told to process the samples, which it does by using the processing manager. To this end, a more detailed view of the interaction of the processing manager, sample sources, channels and signals is shown in Fig. 3. For clarity, only a single sample source, satellite, channel and signal are shown although in practice there may be multiple of each. The processing manager uses the sample source and signal parameters to perform the DRC computations on each signal within the channel. Once the DRC computations are complete, the processing manager forwards all correlator outputs to the channel for processing (tracking). The channels are responsible for informing the processing manager of how many samples to process at a time.

Also shown in Fig. 3 are correlator requests. Correlator requests are initiated by the channel (which has full knowledge of the tracking status for each signal) and are used to request the necessary correlator outputs (code phase and/or frequency offsets relative to the prompt corrrelator) needed for acquisition or tracking. The processing manager is responsible for satisfying the requests of the channels. An example of when a correlator request would be necessary is if early and late correlator spacing is to be narrowed (to mitigate multipath effects) as the tracking status of a particular signal is improved. The key point however, is that the channel completely determines what is needed for tracking the signals contained within it. This is a highly modular structure that is readily modified to accommodate a very wide range of tracking scenarios.

Advantages of the Proposed Architecture

In addition to the flexibility associated with the "composition" approach used in the software and the "autonomous" nature of the channel objects, the proposed architecture offers one other major advantage in terms of processing efficiency. Specifically, it was found that in order to best optimize DRC processing (the high rate computations) on a general processor, all of the data necessary for the DRC computations should be available simultaneously. If this is possible, several optimization approaches can be considered including

Using processor-specific optimizations such as the single instruction, multiple data (SIMD) instruction set available on x86 processors (Pany et al 2003, Heckler & Garrison 2004, Charkhandeh 2007). This is often termed "vectorizing" the processing. Some processors and compilers do this automatically.

- Implementing a multi-threaded architecture which is particularly well suited to multi-core processors.
- Using co-processors such as a field programmable gate array (FPGA), digital signal processor (DSP) or graphical processing unit (GPU).

To date, all of the above optimizations have been implemented (in terms of co-processors, only a GPU implementation is currently complete). Of particular interest here is the use of a GPU which, to the authors' knowledge, has not been previously applied to GNSS software receivers. A GPU is typically used to do computationally expensive graphics coordinate transformations and color shading of polygons in real-time for video games. More recently, they have started to be used for general scientific simulations with great results. A further benefit is that their price-to-performance ratio is several orders of magnitude better than traditional supercomputers.

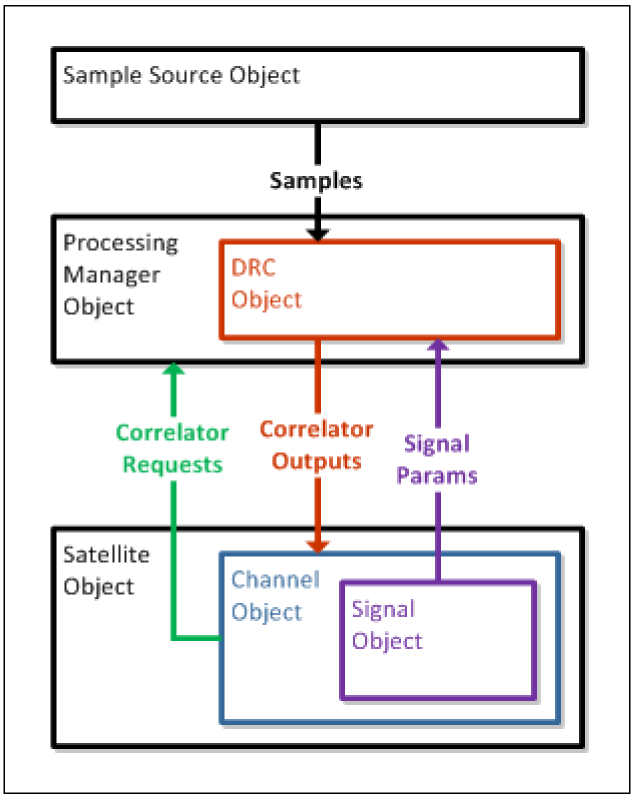

Fig. 3 Interaction of processing manager and associated objects

However, before discussing the GPU implementation in detail (see next section), we first look at different DRC algorithms and their performance using different optimization strategies. In particular, to demonstrate the efficiency of different DRC algorithms, the GSNRx™ software was run using the "rigorous", "table" and "new" DRC algorithms. The "new" algorithm is proposed in Petovello & Lachapelle (2008), which also describes the other two algorithms.

Table **1** shows the average time to perform the DRC processing on 1 ms of data for eight satellites using the different DRCs with different sampling rates. The results were obtained using a single thread on an Intel Xeon quad-core processor. Each processor runs at 1.6 GHz with a 1.066 GHz system bus and 32 kB of L1 cache. Furthermore, each processor has hyper-threading capability, for a total of eight virtual processors.

According to the processor manufacturer, hyper-threading provides "*more efficient use of processor resources*" (Intel 2009), but in practice it has been observed to provide roughly twice the processing capability of a regular processor (i.e., equivalent to using only 50% of the processing core). Comparing the different DRC algorithms, the "table" algorithm performs best. This is somewhat surprising because Petovello & Lachapelle (2008) showed that the "new" algorithm has fewer computations. The difference in performance is explained by the optimizations performed by the compiler and/or processor (different results have been obtained on different processors), and it is clear that these can be significant and should be considered when maximizing processing throughput. Furthermore, for the "table" and "new" algorithms, vectorization provides roughly 50% improvement in processing time.

Table 1 - Average Time to Perform the DRC Processing on 1 ms of Data for Eight Satellites Using Different DRC Algorithms

DRC Algorithm	**Average Time (ms)**	
	5 Msps Data	**25 Msps Data**
Rigorous	1.58	8.06
Rigorous Vectorized	1.29	6.64
Table	1.13	5.54
Table Vectorized	0.56	2.72
New	1.45	6.35
New Vectorized	0.67	3.39

To improve the processing times shown in Table 1, multi-threading was also implemented. In Fig. **4**, the average processing time for 1 ms of data for eight satellites is shown as a function of the number of threads used. The performance of multi-threaded code scales well up to four threads, after which the performance increases are marginal. This is likely due to the fact that, of the eight processors mentioned above, half can be considered "virtual" (i.e., they are not "real" processors). It is expected that these results would scale if more processors were available.

4. DRC Processing Using A GPU

This section presents the details of the GPU implementation for the DRC processing. To this end, two features of GPUs stand out as being particularly useful for the problem at hand. First, they allow a very high degree of parallelism, typically several hundreds or thousands of threads running simultaneously. This is in contrast to the several tens of threads found in a typical central processing unit (CPU). The GPU architecture is limited to executing kernels which perform the same operation on a large data set. While this is sufficient for digital signal processing (as is the case in the current context), it is not at all suitable for implementing general purpose software. The second feature of interest is that GPUs have devoted more silicon area to computationally expensive arithmetic functions. A typical CPU has more than 50% of its area devoted to memory controllers and cache memory. In contrast, a GPU has very little on board cache, devoting extra silicon to arithmetic functions instead. As a result, memory access latency is very high (200-300 clock cycles), but other traditionally expensive operations have been made extremely cheap; on the order of four clock cycles (e.g., sin/cos computations, thread context switching, floating point re-sampling and interpolation).

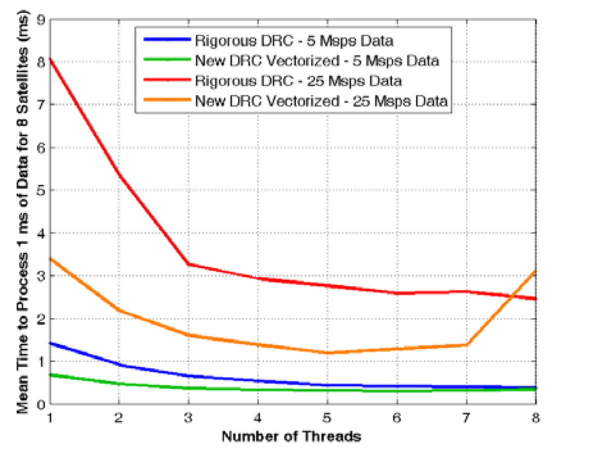

Fig. 4 Mean Time to Perform the DRC Processing on 1 ms of Data for Eight Satellites Using Different DRC Algorithms and Data Rates

5. DRC Processing Using A GPU

This section presents the details of the GPU implementation for the DRC processing. To this end, two features of GPUs stand out as being particularly useful for the problem at hand. First, they allow a very high degree of parallelism, typically several hundreds or thousands of threads running simultaneously. This is in contrast to the several tens of threads found in a typical central processing unit (CPU). The GPU architecture is limited to executing kernels which perform the same operation on a large data set. While this is sufficient for digital signal processing (as is the case in the current context), it is not at all suitable for implementing general purpose software. The second feature of interest is that GPUs have devoted more silicon area to computationally expensive arithmetic functions. A typical CPU has more than 50% of its area devoted to memory controllers and cache memory. In contrast, a GPU has very little on board cache, devoting extra silicon to arithmetic functions instead. As a result, memory access latency is very high (200-300 clock cycles), but other traditionally expensive operations have been made extremely cheap;

on the order of four clock cycles (e.g., sin/cos computations, thread context switching, floating point re-sampling and interpolation).

As mentioned above, relative to a standard CPU, a GPU has considerably more execution units, and each unit is also able to run considerably more threads simultaneously. For example, the NVIDIA 8800GTX GPU used in this work has 16 execution units, each able to run 768 threads simultaneously, for a (theoretical) total of over 12 thousand threads. However, sharing data between multiple processors requires that the processing be partitioned among the different processors and then merged back; a process that can also produce a bottleneck.

In order to better explain how the GPU is used in the software receiver, we recall that to compute any given correlator value, every sample being processed must be multiplied by a local carrier and local code and then the results (one per sample) have to be added up. Obviously, if all of this processing is performed in a single thread on a single processor, there is no benefit to be gained. Instead, within the GPU, the samples are divided into N_s contiguous "slices". Then, for each slice of data, the DRC processing is divided into N_t threads, with each thread processing a subset of the samples within the slice. This concept is shown graphically in Fig. **5**. Both N_s and N_t are design parameters, and for efficiency, should be selected to be powers of two. Each thread then performs the following computations for each sample it is responsible for processing and sums the result:

- Compute of the local code and carrier phase
- Perform the Doppler removal and multiply by the local code

Using the above approach, the GPU effectively divides the processing for a single correlator into a total of $N_s \times N_t$ threads. The result of each thread's processing must then be "reduced", that is, summed to get the final correlator value. However, given the vast number of thread processors on a GPU, this can be highly inefficient and may result in a bottleneck. Fortunately, the NVIDIA architecture provides a mechanism for guaranteeing that a group of threads all have access to a fast "shared" memory. This group of threads is referred to as a "block". Within a block, the reduction process (i.e., adding the results of all the threads) is very efficient because of the shared memory. In GSNRx™, each block is responsible for processing one slice of data for a single correlator, and has been optimized using the techniques of sequential addressing and loop unrolling described by Harris (2007). Finally, because each block only processes a single slice of data, a second reduction is required to obtain the final correlator value by summing the results of all N_s blocks/slices.

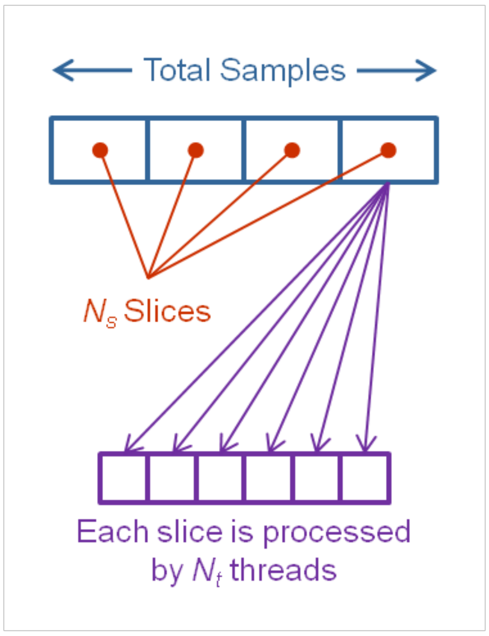

Fig. 5 Diagram Showing How a Group of Samples is Processed in the GPU

The above description only applies to a single correlator value. However, this can be easily extended to include multiple correlator values at once. In this case, processing is performed across a two-dimensional grid of blocks, where the first dimension is the number of correlator outputs and the second dimension is the number of slices (as discussed above). Conceptually, this is shown in Fig. **6**. From the figure, it should be clear that the GPU processing paradigm is highly flexible and scales easily with the number of correlators required and the number of slices the data is divided into (but not the number of samples). In fact, once these parameters are determined (e.g., based on tracking algorithms employed, number of satellites in view, etc.) the GPU takes care of dividing the processing in the most efficient manner possible. In other words, the programmer is allowed to determine the inputs and desired outputs, and then, by using the *same kernel function*, the GPU handles the core processing steps in a transparent manner.

Prior to executing the above processing, all of the samples have to be loaded onto the GPU along with the ranging codes, and the tracking parameters for each correlator to be computed. The transfer of data to the GPU can be executed asynchronously, meaning that the CPU can continue to operate as the samples are loaded onto the GPU. Once this is complete, the execution of the processing is initiated by specifying the kernel function to run, in addition to the grid of blocks and slices discussed above and shown in Fig. **6**.

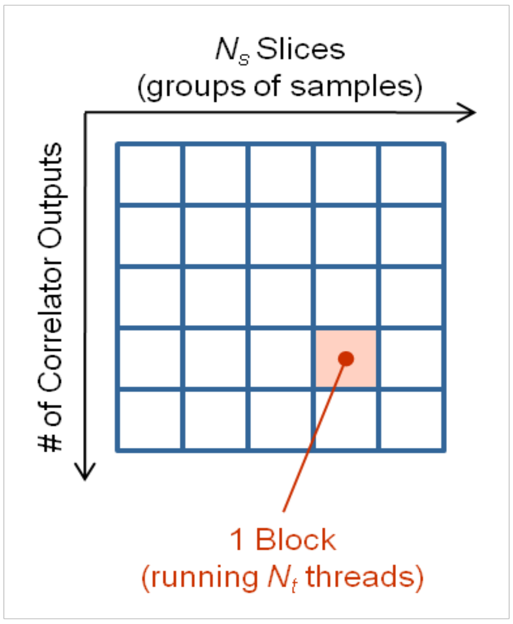

Fig. 6 Computation Grid for GPU Processing

Two final points regarding the GPU implementation are in order before presenting some results. First, GPU programming philosophy is quite distinct from that of CPU programming. In particular, the local code and carrier values are calculated independently for each sample, rather than by incrementing previous values. Second, these local replica values are calculated "on-the-fly" rather than being pre-computed and stored. Specifically, for the local carrier, sin/cos function calls are made explicitly. For a standard CPU, this approach is computationally inefficient, however, due to its design, the GPU executes these function calls much more efficiently. For the local code replica, the code phase (i.e., index into the ranging code that is uploaded to the GPU) is computed independently for each sample using the initial code phase (i.e., at the beginning of the samples to be processed), the code Doppler and sample period. Although this requires more computations than using a lookup table, for example, this is computationally feasible because of the highly parallel structure of the GPU.

To demonstrate the benefit of the GPU, Fig. 7 shows the average DRC processing time for 1 ms of data on eight satellites as a function of the number of threads and slices using 25 Msps data. As can be seen from the plot, there is a tradeoff between the number of slices and the number of threads with the best performance occurring, in this case, with 16 slices and 64 threads per block (although other combinations provide nearly the same performance). Of greater interest however, is that most combinations can process the data in less than 1 ms, suggesting that real-time capability is possible. In contrast, with reference to Fig. **4** (note the different y-axis scales), none of the DRC algorithms were able to process the 25 Msps data in real-time on a general CPU. In other words, the GPU offers the possibility of processing higher data rates in less time, and thus realizing the benefits of the increased signal bandwidths, and at the same time doing it more quickly than with a CPU.

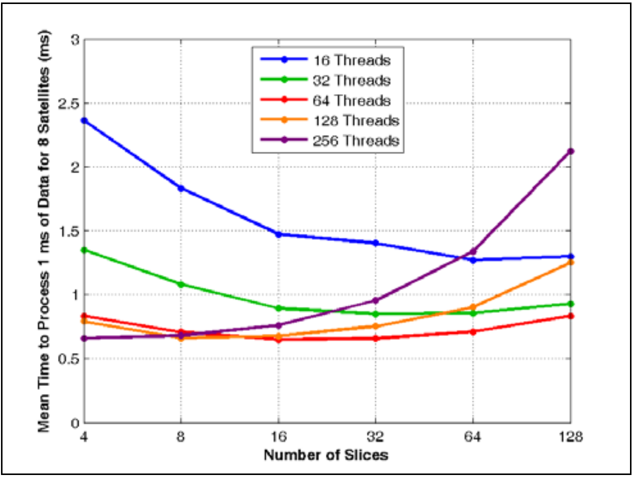

Fig. 7 Mean Time to Perform the DRC Processing on 1 ms of 25 MHz Data for Eight Satellites Using a GPU with Different Numbers of Threads and Slices

6. Software Status and Sample Results

The GSNRx™ software is currently able to acquire and track several signals, as summarized in Table 2. A GPS L1 and Galileo E1 receiver is also working. To date, the navigation solution is only enabled for the GPS L1 signal and the two GLONASS signals because these are the only signals available on a sufficient number of satellites. However, the capability to compute a solution using the other signals is ready and requires final testing with live satellites. In addition, Kalman filter-based, vector-based and ultra-tight architectures (e.g., Petovello et al 2008a) are available for GPS L1 and work is ongoing to incorporate the other signals as well.

Table 2 - Current Status of GSNRx™ Software

Signal	**Status within GSNRx™**
GPS Signals	
L1	Acquire, Track and Navigation Solution
L1C	Work is ongoing
L2C	Acquire and Track
L5	Acquire and Track
Galileo Signals	
E1b/c	Acquire and Track
E5a	Acquire and Track
E5b	Acquire and Track
GLONASS Signals	
L1	Acquire, Track and Navigation Solution
L2	Acquire, Track and Navigation Solution

The software was designed to be able to interface with samples from any front-end hardware, but has not, as yet, been tested in a real-time configuration for any specific front-end. However, given that the results of the previous section show that real-time processing is possible, work is ongoing to have the software work in real-time with a commercially available L1 front-end.
An exhaustive list of references related to the GSNRx™ software (and GNSS signal acquisition and tracking algorithms in general) is beyond the scope of this paper. Interested readers are referred to the PLAN Group website (http://plan.geomatics.ucalgary.ca), which provides access to papers and theses involving software receiver development and testing (as well as all other research topics).

The following sub-sections present some sample results to demonstrate the flexibility of the GSNRx™ software. All of the data processed was collected using a National Instruments front-end system that allows for collection of data on up to three frequency bands at a time (the actual frequency bands used will be clear in the following discussions) using a selectable bandwidth and sampling rate. That said, similar results would be expected with other front-ends. Finally, the results are included mostly to show the flexibility of the software and are therefore presented with minimal explanation.

GPS Results
The most fundamental assessment of a receiver is the standalone positioning error. To this end, Table 3 shows the L1 C/A Code position error statistics for a 15-minute data set collected in open sky conditions. The position solution is accurate to the metre level, as expected given the low level of ionospheric activity during the test and the relatively benign multipath environment in which the data was collected.

Table 3 - GPS L1 Standalone Position Error Statistics

Direction	Mean (m)	RMS (m)
North	-1.4	2.5
East	1.2	1.7
Vertical	-1.1	3.0

The GSNRx™ software is also able to accurately track the carrier phase of the signal, thus allowing high accuracy carrier phase positioning. To illustrate, Fig. **8** shows the RTK position errors as a function of time for a pedestrian-based DGPS test (described in Petovello et al 2007a). For the portion of data shown, the signals were collected in an open sky environment. It is worth noting that the antenna was experiencing peak-to-peak accelerations of about 10 m/s^2 in each coordinate direction throughout the test. In spite of this relatively large level of acceleration, the position errors are still at the centimetre level, as is typical with RTK systems. The error statistics for the data shown in Fig. **8** are given in Table 4.

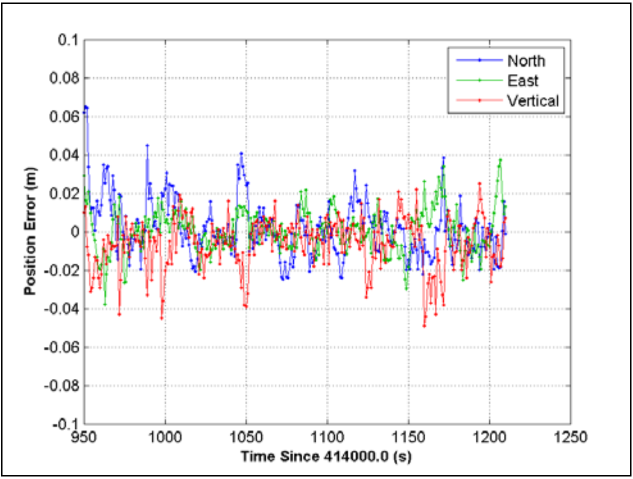

Fig. 8 DGPS L1 RTK Positioning Errors in Open Sky Environment

Table 4 - DGPS L1 RTK Position Error Statistics in Open Sky Environment

Direction	Mean (cm)	RMS (cm)
North	0.2	1.6
East	0.1	1.1
Vertical	-0.6	1.5

In addition to the traditional signal tracking algorithms used to generate the above results, considerable work has also gone into testing new receiver architectures (e.g., Petovello et al 2007a, Petovello et al 2007b, Petovello et al 2008a, Petovello et al 2008b). Two of the most promising architectures are the Kalman filter-based architecture and the ultra-tight integration of GNSS and inertial measurement units (IMUs). A Kalman filter-based receiver replaces the conventional discriminator/loop filter pair with a Kalman filter (although other estimation algorithms could also be used). In an ultra-tight architecture, the IMU measures and compensates for the user's motion, allowing the tracking loops to have a narrower bandwidth. Both the Kalman filter-based and ultra-tight integrations have proven useful when tracking weak GNSS signals. To illustrate this, data was collected on a pedestrian and a variable attenuator was used to slowly reduce the received signal power by 1 dB every 4 s. As the signal power was reduced, different receiver architectures failed at different times. Fig. **9** shows a "histogram" of the horizontal position error as a function of attenuation for different receivers (again, for DGPS L1 RTK positioning). The plot shows the number of epochs whose horizontal position error exceeds a given threshold for all attenuation values up to that shown on the x-axis. Initially, all solutions are able to provide highly accurate solutions, so the number of epochs where the position error exceeds the thresholds is zero. Then, at some point in time (level of attenuation) the receiver “fails” such that corresponding position error exceeds the specified thresholds and never recovers. In this context, “failure” consists of a cycle slip at best, or complete lock of loss at worst. When this happens, the number of epochs where the position exceeds a given threshold increases linearly (with a few minor exceptions). With this in mind, for epochs with an attenuation of 20 dB or less, the standard receiver has about 24 epochs where the horizontal error exceeds 0.1 m. In contrast, for the same level of attenuation, the Kalman filter-based and ultra-tight architectures have one and zero epochs respectively where the horizontal error exceeds 0.1 m. The reason for the improvement with the ultra-tight approach is because in an ultra-tight architecture, the inertial data is used to compensate for receiver motion, thus improving the tracking capabilities of the receiver (*ibid.*).

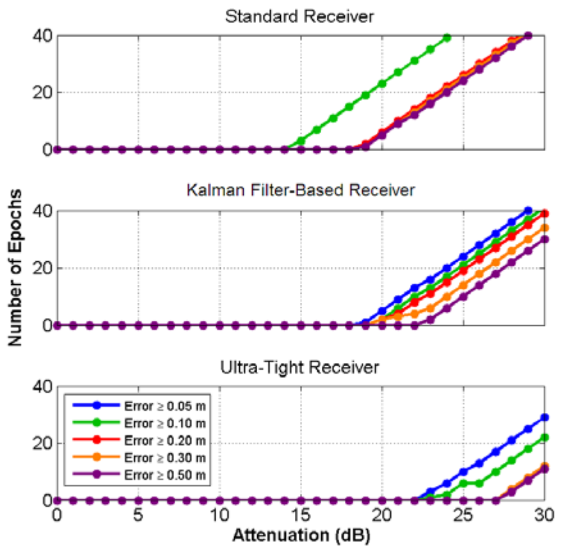

Fig. 9 Horizontal Position Error Histogram for Standard, Kalman Filter-Based and Ultra-Tight Receiver Architectures during Signal Attenuation (lines are plotted in order of increasing position error and thus lines for larger errors may hide those for smaller errors)

New GNSS Signals

As mentioned above, only the GPS L1 signal is fully deployed. The other signals in Table 2 are still not fully available and a fully operational receiver for these signals is not yet feasible (except for the GLONASS signals). Nevertheless, GSNRx™ offers the opportunity to develop, implement and test the acquisition and tracking algorithms for these new signals prior to their full deployment. In so doing, once the signals are available on orbit, the software receiver can be easily extended to take full advantages of these signals, thus reducing product lead time. Included below are some sample results from some ongoing testing and development associated with new GNSS signals and/or systems.

To begin, Fig. **10** shows the acquisition plot for the GIOVE-A (Galileo test satellite) E1b signal employing a BOC(1,1) ranging code. The characteristic side peaks of the signal are clearly visible on each side of the main peak. Also, the sin(x)/x shape is visible in the frequency domain. The results were obtained using two 4-ms coherent integration intervals which are then added non-coherently. Following acquisition, the signal is also able to be tracked (results not shown due to space limitations).

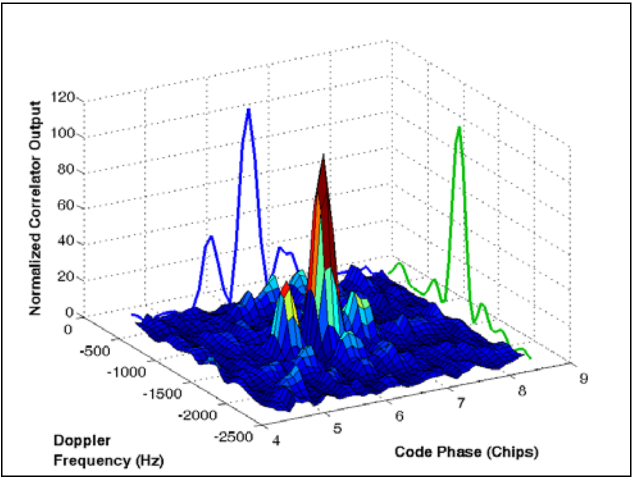

Fig. 10 Acquisition Plot for GIOVE-A E1b Signal with BOC(1,1) Ranging Code (blue line is the projection of the peak in the code phase domain and the green line is the projection of the peak in the frequency domain)

The GSNRx™ software has also been used to acquire and track the GIOVE-A E5b signal. The E5b signal was selected because the Calgary International Airport’s distance measuring equipment (DME) falls in this band and it was desired to see if the resulting interference could be effectively mitigated within the receiver. To this end, the upper plot in Fig. **11** shows the power spectral density (PSD) of the original signal as well as the PSD after applying a notch filter inside the receiver. The effect of the DME interference is effectively eliminated by the notch filter. The lower plot in Fig. **11** shows the estimated C/N_0 for the two signals and it is obvious that the notch filter allows for better signal tracking. The average improvement in C/N_0 is about 2 dB, which is significant.

As a final example, the GSNRx™ software has been used to track signals from the Russian GLONASS system. More specifically, algorithms have been developed to track the civilian signal on both L1 and L2 (Abbasian Nik & Petovello 2008). Table 5 shows the position error statistics for the L1 and L2 position solutions. No GPS measurements were included in these results. For the L1 solution, the position error is very similar to the solution obtained using data collected from a NovAtel OEMV2 receiver (using the same five satellites in both solutions). For the L2-only solution, only four L2-capable satellites were available and the position errors are larger because of satellite geometry degradation, but are still of reasonable magnitude and compare favorably with those of the L1-only solution computed using the same satellites (to remove the effect of satellite geometry).

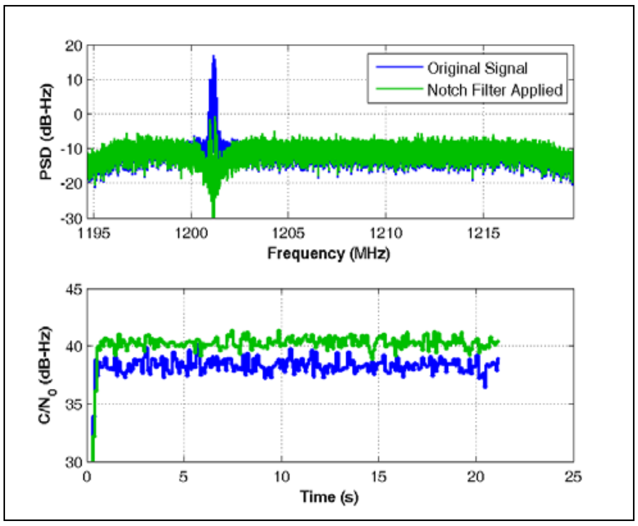

Fig. 11 Power Spectral Density and Estimated C/N$_0$ for GIOVE-A E5b Signal with and without a Notch Filter to Mitigate DME Interference

Table 5 - Standalone GLONASS L1 and L2 Position Error Statistics

Solution	**RMS Error (m)**		
	North	**East**	**Vertical**
5 Satellite Solution			
GSNRx™ (L1-Only)	2.1 m	2.8 m	7.5 m
NovAtel (L1-Only)	0.5 m	1.7 m	2.8 m
4 Satellite Solution			
GSNRx™ (L1-Only)	3.6 m	2.2 m	7.8 m
GSNRx™ (L2-Only)	4.6 m	4.2 m	14.2 m

Summary

From the above, the software architecture clearly provides considerable flexibility to acquire and track new signals and to implement different receiver implementations. This capability is critical in a research environment but is also of interest to agencies wishing to test various algorithms prior to finalizing an implementation in hardware. Another application is products with low replacement rates (e.g., vehicles) that want to incorporate positioning capability now and in the future but would like to easily upgrade functionality in the future as new technologies become available.

7. Summary and Future Work

This paper presented the overall architecture of the GSNRx™ software receiver. The primary benefit of the architecture was shown to be the flexibility it provides for implementing advanced receiver architectures such as ultra-tight integration with an IMU, and for developing and testing algorithms to acquire and test new signals. In addition, the software is structured to allow processing optimizations to be implemented using whatever resources may be available. Herein, the use of vectorization, multi-threading and a GPU were shown to provide various levels of processing improvements. In particular, the GPU was shown to provide considerable processing improvements, and these are expected to become more significant as more signals need to be tracked simultaneously.

Future work will focus on refining existing algorithms while at the same time incorporating functionality to acquire and track the new signals that will soon be available.

For licensing information, please contact the authors.

REFERENCES

Abbasian Nik, S. and M.G. Petovello (2008) ***Multichannel Dual Frequency GLONASS Software Receiver***, Proceedings of ION GNSS 2008, Savannah, GA, Institute of Navigation, In press.

Borre, K., D. Akos, N. Bertelsen, P. Rinder and S.H. Jenson (2007) ***A Software-Defined GPS and Galileo Receiver,*** A Single-Frequency Approach, Boston, Birkhäuser.

Charkhandeh, S. (2007) ***X86-Based Real Time L1 GPS Software Receiver***, M.Sc. Thesis, Geomatics Engineering, University of Calgary.

CSR (2008) ***CSR eGPS: Fast and reliable positioning - everywhere,*** Retrieved March 5, 2009, from http://www.csr.com/egps/.

Fastrax (2008) ***Smart Positioning with Fastrax Software GPS Receiver***, Fastrax Ltd. 2008.

Gernot, C., K. O'Keefe and G. Lachapelle (2008a) ***Combined L1 / L2C Tracking Scheme for Weak Signal Environment,*** Proceedings of ION GNSS 2008, Savannah, GA, Institute of Navigation, In press.

Gernot, C., K. O'Keefe and G. Lachapelle (2008b) ***Comparison of L1 C/A-L2C Combined Acquisition Techniques,*** Proceedings of European Navigation Conference, Toulouse, France.

Harris, M. (2007) ***Optimizing CUDA, SUPERCOMPUTING 2007 Tutorial***, Retrieved March 5, 2009, from http://www.gpgpu.org/sc2007/.

Heckler, G.W. and J.L. Garrison (2004) ***Architecture of a Reconfigurable Software Receiver***, Proceedings of ION GNSS 2004, Long Beach, CA, Institute of Navigation, 947-955.

IFEN (2007) ***NavX®-NSR - GPS/GALILEO NAVIGATION SOFTWARE RECEIVER***, IFEN, GmbH. 2007, Brochure for NavX®-NSR.

Intel (2009) ***Hyper-Threading Technology***, Retrieved 24 March, 2009, from http://www.intel.com/technology/platform-technology/hyper-threading/index.htm?iid=tech_product+ht.

Ledvina, B.M., S.P. Powell, P.M. Kintner and M.L. Psiaki (2003) ***A 12-Channel Real-Time GPS L1 Software Receiver***, Proceedings of ION National Technical Meeting, Anaheim, CA, Institute of Navigation, 767-782.

Ma, C., G. Lachapelle and M.E. Cannon (2004) ***Implementation of a Software GPS Receiver,*** Proceedings of ION GNSS 2004, Long Beach, CA, Institute of Navigation, 956-970.

Misra, P. and P. Enge (2001) ***Global Positioning System Signals, Measurement, and Performance***, Lincoln, MA, Ganga-Jamuna Press.

Mongrédien, C., G. Lachapelle and M.E. Cannon (2006) ***Testing GPS L5 Acquisition and Tracking Algorithms Using a Hardware Simulator,*** Proceedings of ION GNSS 2006, Fort Worth, TX, Institute of Navigation, 2901-2913.

Morton, J. (2007) ***Expert Advice: Software Defines Future, GPS World System Design and Test News***, Retrieved January 7, 2008, from http://sidt.gpsworld.com/gpssidt/Expert+Advice+%26+Leadership+Talks/Expert-Advice-mdash-Software-Defines-Future/ArticleStandard/Article/detail/445464?contextCategoryId=35358&searchString=software%20receiver.

Muthuraman, K., R. Klukas and G. Lachapelle (2008) ***Performance Evaluation of L2C Data/Pilot Combined Carrier Tracking***, Proceedings of ION GNSS 2008, Savannah, GA, Institute of Navigation, 9 pages.

Muthuraman, K., S.K. Shanmugam and G. Lachapelle (2007) ***Evaluation of Data/Pilot Tracking Algorithms for GPS L2C Signals Using Software Receiver,*** Proceedings of ION GNSS 2007, Fort Worth, TX, Institute of Navigation, 11 pages.

NXP (2007) ***NXP Software teams with Mango Research on high performance Personal Navigation Device,*** Retrieved March 5, 2009, from http://www.software.nxp.com/?pageid=140.

Pany, T., S.W. Moon, M. Irsigler, B. Eissfeller and K. Fürlinger (2003) ***Performance Assessment of an Under Sampling SWC Receiver for Simulated High-Bandwidth GPS/Galileo Signals and Real Signals***, Proceedings of ION GPS/GNSS 2003, Portland, OR, Institute of Navigation, 103-116.

Petovello, M.G. and G. Lachapelle (2008) ***Centimeter-Level Positioning Using an Efficient New Baseband Mixing and De-Spreading Method for Software GNSS Receivers,*** Journal on Advances in Signal Processing (JASP), In Press.

Petovello, M.G., C. O'Driscoll and G. Lachapelle (2007a) ***Ultra-Tight GPS/INS for Carrier Phase Positioning In Weak-Signal Environments***, Proceedings of NATO RTO SET-104 Symposium on Military Capabilities Enabled by Advances in Navigation Sensors, Antalya, Turkey, NATO, 18 pages.

Petovello, M.G., C. O'Driscoll and G. Lachapelle (2008a) ***Carrier Phase Tracking of Weak Signals Using Different Receiver Architectures,*** Proceedings of ION National Technical Meeting, San Diego, CA, Institute of Navigation, 781-791.

Petovello, M.G., C. O'Driscoll and G. Lachapelle (2008b) ***Weak Signal Carrier Tracking Using Extended Coherent Integration with an Ultra-Tight GNSS/IMU Receiver***, Proceedings of European Navigation Conference, Toulouse, France, 11 pages.

Petovello, M.G., K. O'Keefe, G. Lachapelle and M.E. Cannon (2007b) ***Consideration of Time-Correlated Errors in a Kalman Filter Applicable to GNSS***, Journal of Geodesy, In Press.

Psiaki, M.L. and H. Jung (2002) ***Extended Kalman Filter Methods for Tracking Weak GPS Signals,*** Proceedings of ION GPS 2002, Portland, OR, Institute of Navigation, 2539-2553.

Scott, L. (2007) Directions 2008: ***Software-Defined Radio Role to Grow, GPS World System Design and Test News,*** Retrieved January 7, 2008, from http://sidt.gpsworld.com/gpssidt/Receiver+Design/Directions-2008-Software-Defined-Radio-Role-to-Gro/ArticleStandard/Article/detail/476704.

Tsui, J.B.-Y. (2005) ***Fundamentals of Global Positioning System Receivers: A Software Approach,*** Hoboken, NJ, John Wiley & Sons, Inc.

Van Dierendonck, A.J. (1995) ***GPS Receivers, Global Positioning System: Theory and Applications,*** B. W. Parkinson and J. J. Spilker, Jr., American Institute of Aeronautics and Astronautics, Inc. I, 329-407.

van Nee, D.J.R. and A.J.R.M. Coenen (1991) ***New fast GPS code-acquisition technique using FFT,*** Electronics Letters, 27(2), 158-160.

Ward, P.W., J.W. Betz and C.J. Hegarty (2006) ***Satellite Signal Acquisition, Tracking, and Data Demodulation,*** Understanding GPS Principles and Applications, E. D. Kaplan and C. J. Hegarty, Norwood, MA, Artech House, Inc., 153-241.

Ziedan, N.I. and J.L. Garrison (2004) ***Extended Kalman Filter-Based Tracking of Weak GPS Signals under High Dynamic Conditions***, Proceedings of ION GNSS 2004, Long Beach, CA, Institute of Navigation, 20-31.