A Framework for Real-time GNSS Software Receiver Research

D.A. Godsoe, *University of New Brunswick*, Canada
M.E. Kaye, *University of New Brunswick*, Canada
R.B. Langley, *University of New Brunswick*, Canada

BIOGRAPHY

Douglas A. Godsoe is a Ph.D. student in the Department of Electrical and Computer Engineering at UNB. Since completing his B.Sc. in electrical engineering from UNB in 1993 he has been working in enterprise application architecture and development for a variety of industries. His current research interests include electronics and digital systems, real-time applications, embedded hardware-software co-design, and digital signal processing.

Mary E. Kaye is an associate professor in the Department of Electrical and Computer Engineering at UNB where she has been teaching and conducting research since 1979. She has a B.Sc. in electrical engineering from UNB and an M.Eng. from Carleton University, Ottawa. Prof. Kaye has been active in the development of embedded systems with real-time applications in the areas of control, signal processing and communications.

Richard B. Langley is a professor in the Department of Geodesy and Geomatics Engineering at UNB, where he has been teaching and conducting research since 1981. He has a B.Sc. in applied physics from the University of Waterloo and a Ph.D. in experimental space science from York University, Toronto. Prof. Langley has been active in the development of GPS error models since the early 1980s and has been a contributing editor and columnist for GPS World magazine since its inception. He is a fellow of the International Association of Geodesy, the Institute of Navigation (ION) and the Royal Institute of Navigation. He was a co-recipient of the ION Burka Award for 2003 and received the ION Johannes Kepler Award in 2007.

ABSTRACT

This paper outlines the architecture and on-going development effort of a modular software-based real-time GNSS receiver research framework using the Microsoft .NET Framework and the C# programming language. A pipelined signal-processing model is used to address key timing and inter-module synchronization challenges inherent in working with the parallelism required to simultaneously receive and process four or more satellite signals. An extensible interoperability layer provides clearly defined functional interfaces and simplifies the integration of existing hardware and software components with any stage in the signal pipeline. Various aspects of front-end hardware design requirements, as well as new acquisition and tracking mechanisms, are identified and discussed.

INTRODUCTION

There are many expected and anticipated advantages of software GNSS receivers over conventional hardware implementations. Among these benefits are lower cost, greater flexibility, easier updating or upgrading mechanisms, and better adaptability for supporting new signals and frequencies. Software receivers serve as fertile ground for researchers exploring the exciting possibilities of the development and testing of new signal processing techniques and ideas. Satisfying the computational requirements for a real-time software receiver has focused much of the current research and development effort on innovative algorithms aimed at reducing the necessary processing complexity. For the purposes of proof-of-concept testing, many of these ideas have been demonstrated using some combination of field programmable gate arrays (FPGAs) and commercial digital signal processors (DSPs) with software written in assembly language. PC-based demonstrations that take advantage of the MMX/SSE (streaming SIMD—single instruction, multiple data—extensions) instruction sets provided by the Pentium-4 microprocessor have required the use of optimized assembler code for their implementations.

While perhaps reconfigurable, hardware definition languages (HDLs), such as VHDL (very high speed integrated circuit hardware description language) or Verilog, are intended to describe hardware operations and are not widely considered to be software as the compiled binaries are not executed on a general purpose processor. Solutions based on a system-on-programmable-chip (SOPC) philosophy, using one or more soft-core processors combined with various application specific logic-blocks, bring the features and performance benefits of both hardware and software. However, they also suffer all the challenges of hardware and software systems, as well, in that they are often difficult to customize, requiring the support of a mix of non-integrated vendor-specific tools and components. Furthermore, solutions built from specialized DSP chip-sets using hand-optimized assembly languages and esoteric development tools require specialized software skill-sets to reproduce. These systems represent more of a one-off customized hardware implementation approach and generally fail to satisfy the adaptability and flexibility benefits expected from software receivers.

Using readily available tools and high-level programming languages makes the technology more accessible to would-be system implementers and brings the desired software receiver goals closer to realization. Beneficial side-effects include having access to larger data storage devices, network connectivity and XML-based web-service integration, links to GIS and mapping information, and support for rich application functionality that is difficult to provide through low-level code only.

The expected benefits of this framework development will be to establish a whole context for software receiver research and to provide a unified view of a software receiver implementation using tools and technologies that encourage the development of diverse feature-rich applications. Support from the .NET Compact Framework makes the move to mobile devices based on embedded versions of Windows a straight path to hand-held applications.

SOFTWARE & RECEIVERS

Since without additional context, software can mean different things to different people, for this paper software is considered to be a collection of algorithms expressed in a programming language that is compiled for execution on a general purpose processor. However, there are many types and models of microprocessors available, as well as different run-time configurations of hardware and software supporting various levels of operating system integration. For practical reasons, the topics of discussion will be restricted to the consideration of a PC-based computing system with an Intel Pentium-4 class of processor running a recent version of Microsoft Windows and the .NET Framework.

To refine scope still further, the murkiness of *real-time* and its accompanying modifiers of *hard*, *soft*, *near*, and *firm* needs to be clarified. Here, the critical distinction that is made between real-time and post-processing is that the real-time system has only the brief interval between sample arrivals to evaluate a new data point. Where the post-processing system has the advantage of being able to replay and "look" at the signal stream multiple times, the real-time system must process a sample before it is gone. Block-oriented real-time processing techniques must digest a block of samples in a time interval given by the product of the block length and the sampling period.

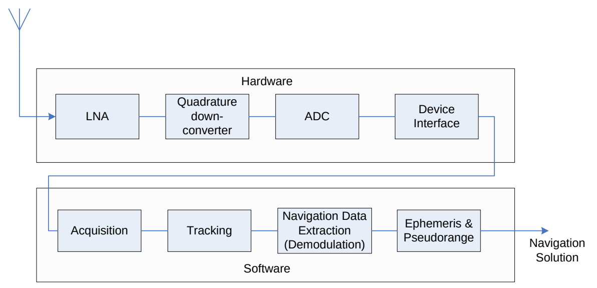

Figure 1—Software GNSS Receiver

As shown conceptually in Figure 1, the traditional model of a software receiver consists of a hardware front-end that performs signal pre-amplification and intermediate frequency (IF) down-conversion prior to sampling and analog-to-digital conversion. The resulting data-stream is then processed further by subsequent software stages. The obvious goal with a software receiver is to put as much processing functionality as possible into software modules in order to maximize the overall system flexibility and potential cost savings. Doing so also allows for, among other things, easier development and testing of methods for multipath amelioration, weak-signal acquisition and tracking, and support for new signals and spreading codes.

Several non-commercial research-oriented reference implementations of post-processing receivers are available, and their number seems to be growing daily. When combined with a low-cost hardware front-end, MATLAB versions such as [2] provide an excellent starting point for the exploration and study of the processing mechanics involved with signal acquisition, tracking, and navigation message recovery. However, a MATLAB application operating on a Java virtual machine is a non-ideal solution architecture for the basis of a real-time GNSS receiver. Versions developed in C/C++ and assembler attempt to address environment performance issues, however C and native assembler don't interoperate easily with other development languages. Building useable high-level applications in the C language is a time-consuming endeavor.

In a post-processing receiver application, sampled data is typically first stored to a file that is then used as an input to the system modules. The input data represents a static snap-shot of the in-view satellites in the sky at the time the sample was recorded. While the acquisition process may subsequently take hours to run, the results obtained are valid for the life of the file; if the file contents or analysis software haven't changed, neither should the processing results. Inter-module synchronization is achieved by running dependent tasks sequentially, fully completing one operational step before starting another. So, for example, satellite tracking does not begin until the acquisition stage has analyzed the signal file and produced the output of a collection of objects that can be tracked.

Conceptually, real-time and post-processing software receivers share the same basic processing stages, however receivers attempting to operate in real-time must analyze a newly arrived individual data point in the time interval between sampling instances. The code has only a small moment of time to determine the impact this new data will have on the receiver state and output. Real-time operation requires the synchronization of multiple overlapping dependent tasks with tight completion deadlines that are executed on a predictable schedule. If an acquisition stage precisely determines the initial code delay of a signal to be tracked, but the handoff between acquiring the signal and the tracking process takes an unpredictable or excessive amount of time, the change in signal with the passage of time invalidates the result and necessitates re-acquiring the signal all over again.

The current approach towards realizing real-time software receiver operation most often involves optimizing the implementation of the various receiver sub-modules to improve execution performance. There are two drawbacks to this situation, though. Firstly, optimizing individual isolated receiver pieces leaves the individual optimizer feeling rather isolated. Determining the performance gain from enhancements in one component becomes difficult without the ability to easily integrate the implementation back into a working system for comparison purposes. Secondly, while overall performance is important, it alone is not the whole picture in meeting the needs of a real-time system.

Achieving real-time operational characteristics will require more than an optimization effort on existing post-processing algorithms. Even in a multiple-processor-core PC environment, performing calculations in long-running iterative loops pins a workload to a single processor while the others remain essentially idle. To avoid this situation and to maximize the ability of the system to schedule execution resources in a fine-grained manner requires the use of a carefully designed multi-threaded application. Managing and synchronizing thread execution and cross-thread interactions becomes an essential element in the software receiver solution architecture. Additional considerations regarding such things as the timing of the transition between signal acquisition to tracking, tracking-loop stability analysis, and the implications of the front-end sampling-rate limitations must also be evaluated.

Post-processing receivers have the advantage of being able to work away on the signal data starting from different initial offsets and cross-comparing the results to determine the location of navigation data-bit edges. A real-time receiver must make these determinations along the way, and somehow compensate for timing synchronization errors without corrupting the received navigation message. The more sophisticated the edge detection and timing extraction algorithms become, the less likely it is that they will be able to be executed under the real-time constraints.

Both varieties of software GNSS receivers, in general, suffer from basic implementation challenges that would be straightforward to overcome in hardware-based designs. With hardware receivers, inter-module synchronization is readily achieved by connecting dependent components to a common clock source. Measuring incoming signal frequency and phase, implementing feedback control, and the processing of the integro-differential equations present in tracking-loop filters all occur quite naturally with coils and capacitors in hardware, but require many lengthy calculations when performed in software. Even a basic shift-register generator can involve complex code and large data structures to implement efficiently in software. When necessary, depending on the discriminator function used, the synchronization of a software phase-lock loop (PLL) with the output of a data-bit recovery demodulator creates another challenge. In this case, it is the hardware that is soft and the software that is hard.

Happily, there are things that are much simpler to achieve using software instead of hardware. Providing better and easier support for making changes to functionality implementation is one of them. Visualizing the internal system state through introspection and reflection during testing and debugging in an integrated development environment is another. Software implementations allow the direct access to observable system states within the receiver that would otherwise require intrusive instrumentation to realize in hardware.

RELATED WORK

General background information on the design and operation of GPS receivers can be found in [8] and broader but related details on spread-spectrum communication systems in [5]. Additional information on software signal processing methods are provided in [2], [15] and [17]. These references develop some of the necessary theory for software-based GNSS receivers.

There are so many software receivers of the post-processing non-real-time variety that a comprehensive evaluation of the nature and characteristics of all of them would be practically impossible. Nearly any programming language that can open a file and perform basic mathematical operations can be used for the development of these projects. A good overview of the current state of software GNSS receivers can be found in reference [1] and issues with their testing protocols and challenges are covered nicely in [9].

The variety of MATLAB-based solutions, such as those presented in [13] and included in [2], like the examples provided by [15] and [17], are post-processing receivers and possess no real-time design intentions or characteristics.

There are also highly optimized receiver components that are written in some combination of assembler and C++ claiming real-time performance benefits. However the designs and methods of implementation of test-beds such as [4], or the software-defined-radio receivers for GPS and GNSS discussed in [11] and [3] are cobbled together from the complex interconnection of a veritable Jenga® tower of hardware-dependent system prototyping components. Solutions such as these are difficult to repeat and customize, and it is even harder to incorporate their presented generalized conclusions into a specific application development and testing environment.

Any relevant discussion on threading, parallelism, and the required level of inter-process communication and data structure synchronization are conspicuously absent from the vast multitude of emerging real-time software receiver implementation papers.

THREADS & PROCESSORS

A *thread* is a conceptual construct that is used to model the path a computer processor takes when performing the instructions contained in a set of code in the form of a program. Threads represent a basic unit of software execution.

The state of a thread consists of all the CPU registers—instruction counter, stack position, status flags, accumulator and other general purpose registers—that hold the current values of an executing program's internal variables. State reduction is achieved by externalizing these internal registers to reserved areas of system memory where they can be saved and later retrieved. Multiple blocks of memory can be used to simultaneously maintain the state of several threads, which gives rise to a multi-threaded run-time environment. When one thread can no longer make progress or if another more important (higher-priority) thread needs to run, the current thread's state is stored and the former state of the next thread ready-to-run is loaded and the processor resumes execution.

By managing multiple threads in such a manner, a single high-speed processor can appear to perform several tasks in parallel. Of course, the resulting parallelism is in appearance only in that one processor may only execute one thread's instructions at a time. To improve performance, multiple processors may be used to execute an application, where each available processor manages a different set of threads for execution.

A thread is a synthetic construct that has been invented in order to optimize the utilization of a limited number of processor clock cycles. If a thread is waiting for an external or shared resource, such as the completion of an I/O operation or reading from a comparatively slow memory location, rather than wasting time spinning (or idling) the current thread it is better to do other work instead. A context switch is performed by the operating system whenever progress is no longer being made on the current thread's execution while it waits to acquire a resource. Threads provide, when properly used, a means of creating fine-grained independent workers for long-running or background activities.

Scheduling is the act of determining the order in which the threads (or tasks) within a process are made running on the CPU. A reliable real-time system implementation requires a deterministic scheduler; a thread that is scheduled to begin executing must do so in a predictable manner and it must complete its work in a fixed amount of time.

There are many factors that can influence the length of time a particular task needs in order to complete its work. Even without thread interactions for non-processor resources, factors such as differences in the instruction counts for conditional code branches, iterations of loop-counts based on external variables, the order in which memory has recently been accessed, and the specific processor architecture all influence the number of clock cycles (i.e. time) required for a task to complete. As a result of these extrinsic variations, statistics on the best-case, worst-case, and average completion times need to be collected and used in some way in order to evaluate the viability of a particular schedule.

The application development methodology discussed in [14] stresses predictability and flexibility in real-time system design. Implementations utilizing processor architectures that are *Complex Instruction Set Computer*-based (CISC) with their variable instruction lengths and pipeline depths are difficult to analyze and evaluate for worst-case execution times. Furthermore, the times arrived at would be large compared with average execution times, yielding an overly pessimistic schedule. On the other hand, favoring predictability and low-variance, the philosophy of simpler designs in *Reduced*

Instruction Set Computer (RISC) architecture machines makes the performance characteristics easier to analyze.

For the PC, the Intel Core 2 Duo, a CISC-based processor, provides two logical processors in a physical package. Each processor has a separate execution core and first-level (L1) cache. Both cores use a shared second-level (L2) cache, but the full capacity of this cache can be used by one logical processor if the other processor is inactive. The Core 2 Quad processor consists of two identical copies of the dual-core modules. Each logical processor in both the dual and quad core packages accesses the outside world through a shared system bus.

The pipelined micro-architecture of the Intel Core contains [6]:

- An in-order front-end that fetches instruction streams from memory. Four instruction decoders handle up to five instructions per cycle. Decoded instructions (called μops) are fed to an out-of order execution unit four at a time.
- An out-of-order execution engine that issues up to six μops per clock cycle. The μops are re-ordered to execute as soon as operand sources are ready and execution resources are available.
- An in-order instruction retirement unit that ensures μop execution results are processed and completed in a sequence consistent with the original program order. A peak instruction retirement rate of up to four μops per cycle can be attained.

Each processor core is able to fetch, dispatch, execute, and retire up to four instructions per system clock cycle. When an instruction sequence causes the processor to wait for a shared resource, the execution core performs other instructions rather than sitting idle. For semantically correct execution, the results of instructions must be committed in original program order before they are retired.

Due to the high level of pipelining in the processor, instruction timing data from Intel is specified in the form of *latency* and *throughput* values. Latency is the number of clock cycles necessary in order for the execution core to complete the execution of all of the μops that form an instruction. Throughput refers to the number of clock cycles required to wait before the same instruction can be accepted again. These values are implementation dependent in that they can vary between different core models.

Other factors affecting these timings can include:

- The memory type the instructions came from and any cache replacements or memory write-backs that are subsequently required.
- Activities on other cores, what instruction sequences they are executing and have recently completed.
- Code optimizations across all other running threads.
- Whether the processor is operating in 32-bit or 64-bit execution mode.

As a result of these variations, the often-used means of determining total execution times by summing the clock cycle counts for a series of instructions is not valid for a modern superscalar processor.

"Due to the complexity of dynamic execution and out-of-order nature of the execution core, the instruction latency data may not be sufficient to accurately predict realistic performance of actual code sequences based on adding instruction latency data." [6]

Since reliable and precise clock-cycle counts for instruction execution times are not available, results derived from generalized assumptions on how long individual operations take are inadequate to predict required performance levels. Statistical measurements on execution times gathered with an application profiler must be used instead. The outcome of such an analysis effort is a stochastic schedule—ranges of probability—that is only valid for a given execution environment and instruction sequence, making it impossible to accurately determine the process run-time duration.

Of course, it would also be naïve to assume that a quad-core processor has four times the capability that a single-core processor would have. The levels of parallelism implicit in the algorithms and the amount of synchronization overhead required for their execution limit the potential performance gain possible with multi-processor systems. In general, increasing the number of processors does not proportionally increase the processing performance of the system, depending on the degree to which portions of the application can be executed in parallel. Assuming a 50% parallel workload, [6] calculates the expected performance improvement using two physical processors to be only 33% compared to using a single processor, with four processors providing no more than a 60% improvement over a single processor. In practice, it can be very difficult to determine with any certainty the actual degree of parallelism present, since the final application that runs is often some mix of user, library, and operating system code. However, improper use of thread synchronization can reduce the effective level of parallelism and diminish the potential performance gain through processor scaling.

With a hardware-based design, getting two or more modules to execute in-step with one another is basically a matter of running wires from their respective clock inputs to the output of a common clock source. Similar synchronous behaviors in software are much harder to achieve; it is very difficult to guarantee that multiple code modules will run simultaneously with a high-degree of timing precision. In single-processor multi-threaded environments, concurrent execution is obtained by interleaving the instructions from different threads according to some schedule. With a multiple-processor solution, multiple threads may run together over time, but predicting and guaranteeing their short-term temporal behaviors and interactions are critical and non-trivial design challenges.

Unanticipated process interactions across shared data structures can cause unpredictable, seemingly random, results. The manifestation of these interactions most often comes in the form of spurious data corruption and system crashes. Debugging and proactively eliminating the side-effects from poorly behaved threads requires a great deal of time and testing in order to achieve a satisfactory level of application performance and reliability.

The end result of attempts to characterize and manage the vagaries of multiple interacting threads is the realization that achieving some measure of real-time software receiver operation will require much more than additional processors and complex algorithm optimizations.

RECEIVER DEVELOPMENT FRAMEWORK

The GNSS Receiver Development Framework project has been designed and developed to address the issues of thread-management, inter-process communication, and module synchronization associated with the levels of parallelism required for real-time software-based GNSS receivers. The main goals of the framework's object-oriented design are

- to be developed using a modern high-level language with tools that are intended for the implementation of feature-rich applications;
- to provide a modular component model that supports a high-degree of reuse through inheritance and poly-morphism;
- to integrate with other 3rd-party hardware and software components in as simple a manner as possible;
- to act as an extensible baseline receiver reference.

Serving as the focal point for customization and functional composition, the receiver development framework provides the essential aspects for object creation (*instantiation*), system *orchestration*, signal detection, synchronization, and tracking. Unlike off-line post-processing tools and utilities, the framework is intended to deliver the critical performance characteristics necessary to achieve real-time receiver operation.

An important aspect of the framework's design is providing for the integration of external hardware and software components in a seamless and consistent manner. Doing so allows for the immediate reuse of existing solution pieces, while simultaneously supporting the externalization of any newly developed features. As such, receiver algorithms may first be developed and tested as software within the framework and then migrated into hardware representations that can be hooked back into the receiver object model for further testing and evaluation.

The receiver framework supports the development and integration of toolkits of a variety of implementation types for each component category. Baseline performance measurements and operational characterizations with one implementation strategy can be made and used for direct comparison to alternate methods for benefit evaluation. By leveraging the advantages of object-oriented design techniques, comparisons can be made with minimal code changes simply by overriding the implementation of a class virtual method and invoking the base-method at run-time.

The desire in the creation of the receiver framework was not to refine or improve upon existing signal analysis algorithms, but rather to give them a better place to live. The longer-term vision is to produce a number of Visual Studio project templates that may be used as starting points for receiver development. It was decided to avoid basing the design on the capabilities of toolkit-centric scripting languages, such as MATLAB, for the previously mentioned performance issues. Of course, this decision carries a few consequences, not the least of which are the many features of convenience, such as graphing and plotting functionality, and frequency domain filters and transforms that are immediately absent. While the product of this framework is not intended to be simply another signal-processing toolkit or a collection of library components that can be incorporated into other applications, it does provide implementations for many of the essential signals-related transforms and mathematical operations. Simple graphing objects have also been developed that can be placed within the user interface as visual elements. Data collected from the receiver may also be exported and analyzed separately by other applications as required.

The reference implementation provided is only one way, not necessarily the best, of achieving a signal detection and tracking objective. However, it is the generalized set of interfaces and abstract classes that give the framework its flexibility and offers the greatest value to its consumers. The design of the framework establishes the philosophy of defining an interface, declaring an abstract base that implements it, and creating derived types that satisfy solution requirements. The aim of the framework is to serve as a development guidance reference for how receiver applications should be structured and assembled to achieve the most worthwhile results in the shortest timeframe with the greatest opportunity for reuse and extension.

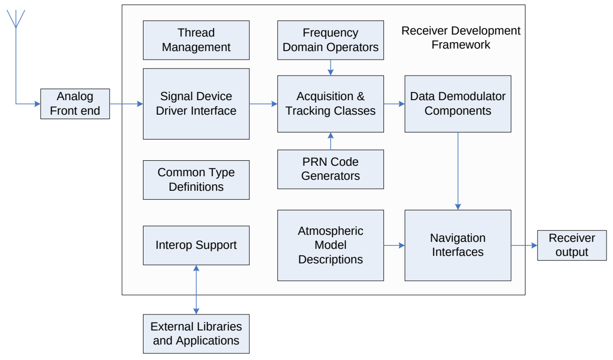
Thread Management
Frequency Domain Operators
Receiver Development Framework
Analog Front end
Signal Device Driver Interface
Acquisition & Tracking Classes
Data Demodulator Components
PRN Code Generators
Common Type Definitions
Interop Support
Atmospheric Model Descriptions
Navigation Interfaces
Receiver output
External Libraries and Applications

Figure 2—Receiver Development Framework

The top-level system diagram is shown in Figure 2. Each box in the development framework diagram represents a collection of related pieces that combine to produce the expected output from the corresponding module. All that is necessary to change or extend a component is to implement the classes of objects that support the required interfaces and methods. While the greatest flexibility will be achieved when all of the components are developed in software, there is nothing in the overall design that precludes the substitution of a specialized hardware device in place of a class method or an entire class implementation, as long as the hardware fully supports the expected input and output parameters of the method being replaced. It's for this reason that the system was developed in such a way that it more closely resembles the block diagrams of hardware that it represents.

The object-oriented design takes advantage of the reusability available through inheritance and poly-morphism. Types that are derived from a common base-class can be thought of as new implementations of the base-class functionality. Invokers see no difference in the calling semantics, and implementers can leverage any suitable functionality existing in the base-class. Reuse exists, therefore, at two levels.

The analog front-end is used to tune, band-limit, and sample the incoming signal so that it can be brought into the system in the form of a stream of binary data. It is connected to the framework through a set of signal device driver interface components that are used to structure the properties and attributes of the device data source into a format that is compatible with the dependent sub-systems.

Interoperability support features allow previously developed functional libraries and external hardware devices to be tied into the system in a uniform and consistent manner without a great deal of effort or intellectual overhead. Although dependent on the polymorphic behavior of the framework's object-oriented design, the interoperability layer allows for high-degrees of reuse and application flexibility.

While not necessarily an integral part of the receiver framework in that they borrow from and must be supported by operating system constructs, the thread management components represent a collection of work-queues and synchronization primitives that are required to help ensure the desired real-time performance objectives of the system. Events and delegates available to other system components are considered part of thread management services, and patterns for their use are provided.

The frequency operators section refers to the frequency-domain essentials of signal processing. These include FFT/DFT functions and their inverses, filtering functions, and other operations necessary to transform input signals into useable data. Reference implementations have been provided as parts of the receiver framework; however, these can easily be replaced with alternate software algorithms or with customized external hardware blocks.

Classes and supporting interface specifications for items that are commonly encountered or shared between operations such as the Complex data type and system status enumerations are provided in the common type declarations of the receiver framework. Data types and structures that bridge between functional modules can be considered as part of the common type library.

Pseudo-random noise (PRN) code generators are required to reproduce an exact replica of the sequence of binary chips that was originally used by the transmitter to spread the signal. Only the GPS C/A codes are currently provided, but different code types and generating methods are supported by extension.

The signal acquisition and tracking module contains the classes that are responsible for finding the presence of PRN sequences in the received signal and keeping the locally generated sequences in synchronization. The acquisition process attempts to discover a transmitted sequence by cross-correlating the incoming signal with each possible spreading code while looking for a peak in the correlator output. After signal acquisition is completed, a collection of objects is returned for tracking—each object representing a detected PRN sequence in the input signal. Object models for a PLL, a delay-lock loop (DLL), a numerically-controlled oscillator (NCO), and a reference time-base are provided as part of this module.

The data demodulator components contain the code that is required to extract, verify, and process the recovered navigation data message. Any required data formatting and validation functions may be included in this block as well.

THE PIPELINE COMPONENT MODEL

An often used construct of high-performance computer architecture is the sequential pipeline, where functional blocks are linked together in a chain and driven by a common clock. Each block in the chain achieves some measure of work in the time interval between clock pulses (ticks) contributing to the overall operation. New operations are started at the beginning of the pipeline while in-progress operations occur at successive stages. The output of one block becomes the input for the next, and the final result is taken from the output of the last stage in the sequence. In hardware, the clock source is usually the output of a local oscillator or some other reference signal that is physically wired to a control point on each stage that latches the input data between clock transitions.

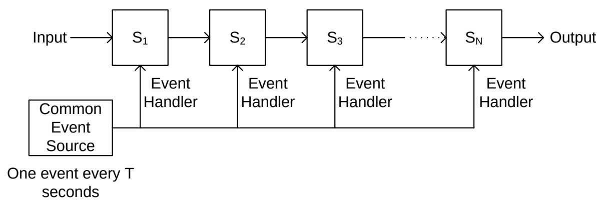

Figure 3—Synchronous pipeline model

Without access to a shared time reference, pipelines are difficult to build using software constructs. However, the same pipeline structure can be achieved in software applications with events and event-handlers. To do so requires event-handlers from multiple object instances be assigned to respond to a single shared event-source. The shared event-source serves an equivalent role as the common clock in the hardware version, such as the configuration shown in Figure 3. The throughput benefit of this model results from the more evenly distributed work-load due to the processing that occurs between the event-trigger intervals.

Unfortunately, with software-based events, the operating system makes no guarantee as to the order of delivery of the event signals to the various subscribed handlers. The order in which the event-handlers are registered with the event-source can influence but not fully determine which handlers will receive the event notification first. If the last stage is triggered to run out of sequence with the rest of the pipeline, the resulting output would be a repeat from the previous cycle and obviously an error. A synchronous pipeline requires semaphores, wait-handles, or other shared synchronization primitives to reliably execute in software, each of which negatively effects system performance and increases implementation complexity.

The receiver framework makes use of an innovative event-driven asynchronous pipeline model for the processing activities involved in signal acquisition and tracking, as shown in Figure 4. Each stage in the pipeline is notified by an event from the preceding stage that the next signal sample is available for processing. The stage reads the passed-along prior stage's output value as its input and updates its current time. The stage then performs a small amount of processing on the sample and sets its output property to the newly calculated result. Finally, the stage signals, through a new event to the successor downstream stage, that it has completed its processing chore and its output is ready. As a result, each sample is time-stamped as it arrives and is allowed to ripple through the pipeline without blocking or interfering with the processing activities of the antecedent stages.

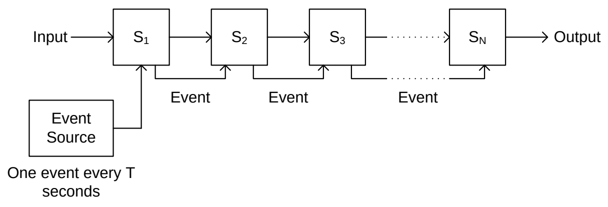

Figure 4—Asynchronous pipeline model

The modular definition of the components allows for a kind of plug-and-play approach to the overall system implementation. The event delivery sequence is entirely determined by the pipeline organization.

It is important, however, to keep each stage's event-handler computationally simple and to limit the overall length of the pipeline in order to minimize the total processing delay. Tasks that need to run longer or do more work than is practical in the event-handler should do so on blockable worker threads that are created and started when the object instance is initialized. The basic pipeline stage element is shown conceptually in Figure 5. The stage properties include the input, output and time data values that are inherited from the component base-class.

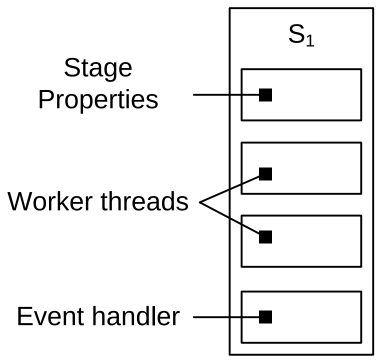

Figure 5—Pipeline stage model

Unlike other software-based signal processing models that operate by collecting a large number of samples and then processing them in bulk, the pipelined approach allows a single sample to trickle through the system and be processed in real-time. Each pipeline stage in the receiver framework has, in addition to an input and output, a control object access point that optionally allows feedback from downstream or external components so that its behavior can be regulated by the outputs of other objects. The pipeline component model allows one to easily mimic hardware timing behaviors and functions in software. As defined, the model more closely resembles a discrete time-domain representation of a feedback control system block diagram, which minimizes the need for processor intensive transform-based analysis of large blocks of signal data.

Figure 6 shows an example set of components arranged as a pipeline. The components support control from feedback and feedforward objects, such as frequency and phase adjustment from a PLL and gain control from a signal level detector.

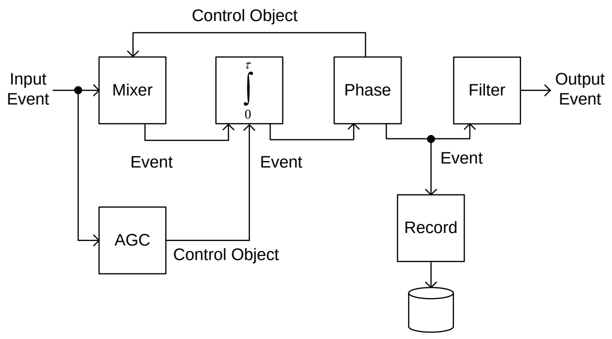

Figure 6—Example pipeline configuration

Each component of the pipeline is derived from a common base-class implementing a typed interface that represents the minimum functionality required to participate in the receiver pipeline structure. Generalizing the component description in this manner allows the overall configuration of the pipeline to be highly adaptable, easily supporting the creation and integration of new components. However, if needed, existing components can be enhanced through extension to support additional specialized properties and methods as required, without significant loss in flexibility. The default component behavior is a unit delay (z^{-1}) transfer function.

The pipeline components are configured into the desired sequential structure with their events connected and control objects assigned as part of a *Pipeline Container* class. All that is required to run multiple pipelines in parallel is to create multiple container instances and connect them to a shared front-end event source. The pipeline container type is derived from the base pipeline component class so that larger pipelines can be comprised of smaller ones, meaning pipelines may contain stages that are themselves pipelines.

RECEIVER INTEROPERABILITY SUPPORT

The interoperability support features provide guidelines, class templates, and other resources for the integration of external hardware or software components through a consistent set of interface wrappers. If required, it is possible to implement in external hardware intermediate parts of a receiver that are connected through a suitable device-driver interface, and to make them behave as if the work were performed by an internal application component. This capability allows hardware functions to be initially defined and tested in software, and then implemented in hardware. Once implemented, the hardware can then be plugged into the framework replacing the software version of the component for relevant performance evaluation comparisons. Software functions built using other implementation tools or technologies (languages, etc.) may also be combined with the core application framework in a similar manner. Any component within the system can be implemented externally through the interoperability interface, provided all critical timing requirements are satisfied.

The relationship between the interoperability layer and the other framework components can be conceptualized as shown in Figure 7. While the receiver framework provides interface specifications and type declarations, the interoperability layer provides a means of sending and receiving messages to and from external or 3rd-party components.

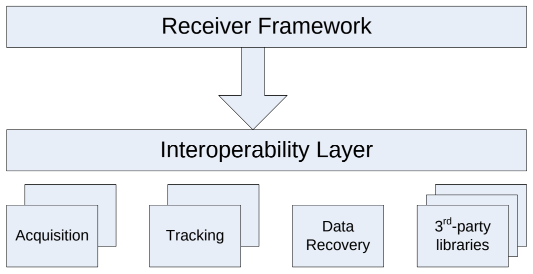

Figure 7—Interoperability Layer

Figure 8 shows the various strata that make up a representative interoperability implementation. Depending on the device or component-level technology involved, not all layers in this four-layer model will need to be provided. At a minimum, only Layer-4 is required with layers one through three providing hardware service abstraction, state management, and data type compatibility, respectively.

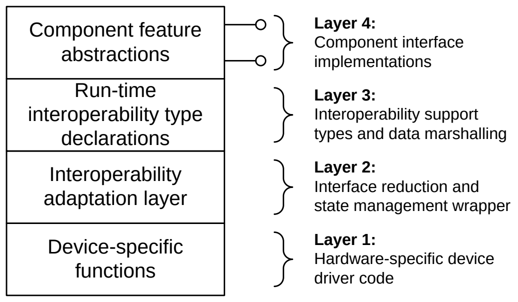
Component feature abstractions
Run-time interoperability type declarations
Interoperability adaptation layer
Device-specific functions
Layer 4: Component interface implementations
Layer 3: Interoperability support types and data marshalling
Layer 2: Interface reduction and state management wrapper
Layer 1: Hardware-specific device driver code

Figure 8—Layered interoperability model

Signal sources from different types of hardware front-ends, including simulated and file-based data, may be connected to the receiver framework through the inter-operability layer. Low-level device driver code for detecting, initializing, and activating the front-end is specified at Layer-1. Any code that is necessary for amalgamating multi-step operational sequences into a single high-level step is developed for Layer-2, as is the persistence of device state information. Any required numeric type conversions or data formatting issues are resolved at Layer-3. Finally, Layer-4 represents the connection point to the receiver framework for the signal source.

The benefit of the layered interoperability model is that there typically is no need to repetitively coerce internal data representations to fit different application programming interface signatures. System-level modules need only support the externally visible level of abstraction through the appropriate interface specification. As a result, connecting components from different sources should require no hacking and patching of someone else's code to support additional functionality.

TESTING RESULTS

Evaluation of the receiver framework has been conducted with the post-detection tracking configuration shown in Figure 9. Using an SiGe SE4110L-EK3 USB Link-1 (L1) receiver front-end, the signal from the antenna (not shown) is down-converted to an intermediate frequency (IF) of 4.1304 MHz, and then sampled and digitized at a sample rate of 16.3676 MHz. The event-source for the arrival of a new sample from the front-end is connected in parallel to a DLL module that follows the code delay, and a signal-recorder object that writes the sample data to a file on disk.

The DLL component produces early (E) and late (L) C/A PRN code sequences by multiplying the input signal with the output of an NCO block. A prompt (P) sequence is kept time-aligned with the received code by adjusting the code delay amount with a normalized E – L feedback loop. Using the locally generated P values, the code is removed from the carrier. The code-removed output from the DLL is passed to a PLL that tracks changes in carrier phase using an arctangent discriminator function and adjusts the output of the NCO. Finally, the phase transitions caused by the presence of the navigation data-bits in the signal are forwarded to a demodulator component that is used to extract the 50 bps data stream. The outputs from the DLL and PLL components are also fed to data recorder objects in a manner similar to the input signal.

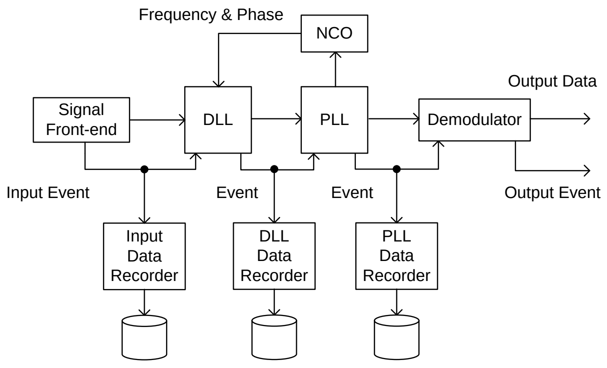

Figure 9— Tracking pipeline configuration used for testing

Referring back to Figure 8 for a moment, the code for initializing the front-end hardware, starting and stopping the sampler, and conducting sample data transfers over the USB link represents Layer-1 functionality. Persisting and reporting on the status of the device, such as current error conditions, and combining the multiple steps required to detect, enable, and activate the hardware into single functional commands is handled in Layer-2. In order to start the sampling process, the device needs to be either reset or reinitialized, and then sent a byte-oriented command sequence to turn-on the sampler; all of this step-by-step activity was rolled into the single statement commands *Start*(), *Stop*(), *Reset*(), and *ReadData*(). The 2-bit (magnitude + sign) data values from the analog-to-digital-converter (ADC) in the front-end that are transmitted over the USB connection to the PC as unsigned 8-bit bytes are translated into a (sign + magnitude) representation by mapping the values {11, 01, 00, 10} to {-3, -1, 1, 3} at Layer-2. The byte-sized data types used by the front-end are directly compatible with the interoperability data marshaller, so the conversion services of Layer-3 were not required. Had the front-end utilized more complex structures involving indirect pointers or multi-byte character strings, these data types—referred to as non-blittable, since their representations in memory are not consistent between run-time environments—would need to be converted to more basic types before being exposed to Layer-4 where the hardware makes the final connection to the receiver framework.

Only a single instance of the tracking pipeline is shown in Figure 9. For tracking multiple signal sources (satellites), instances of the pipeline are created and initialized for each tracked object, all connected to the shared signal front-end. Passing the data through the system in this manner eliminates the need for maintaining large arrays of samples that are synchronized across threads and aged out of memory when the last process is completed. Each stage in the pipeline has its own time-stamped copy of the data it requires to complete its specified task. Information regarding the incoming sample rate, IF, data bit-rate, PRN code delay, and carrier frequency and phase are properties of their representative component classes and are made visible to the receiver pipeline container.

In order to produce a repeatable set of results, the input signal from the front-end was initially recorded to a data file, which was then used as the input source for subsequent analysis. Furthermore, the front-end hardware utilized was originally designed for data-capture in a post-processing application and is not entirely adequate for real-time signal processing. The device suffers from an inability, by design, to capture data for more than 40 seconds without requiring a reset. At ≈16 MHz, the sample-rate is excessively high such that on a 2.4 GHz Pentium 4 processor only 150 (2400 / 16 = 150) system clock-cycles worth of time is available between samples to complete all processing stages. However, with a sample-rate of ≈4x the intermediate frequency, the minimum phase difference between samples of π/2 makes accurately tracking the carrier phase on a sample-by-sample basis either impossible or results in compromised long-term stability of the PLL module.

An excerpt of the input signal as recorded is shown in Figure 10. The signal exhibits the expected noiselike appearance, bounded by the -3 to +3 input range.

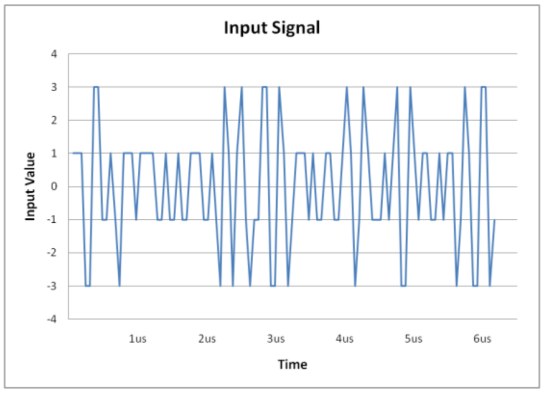

Figure 10—Time-domain view of input signal source

The distribution of the input values covering the same time period as Figure 10 is shown in the signal histogram of Figure 11. The appearance of this graph indicates that the input signal is not over-saturating the sampling hardware and that the signal level is reasonably well-balanced over the available range.

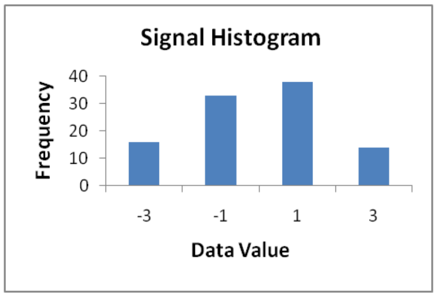

Figure 11—Input signal histogram

The frequency domain view of the signal is provided in Figure 12, which was obtained by performing ensemble averages on ten consecutive 4096-length Fast Fourier Transforms (FFTs). The resultant periodogram shows the input signal energy spread over about a 4 MHz bandwidth that is roughly centered on the ≈4 MHz IF.

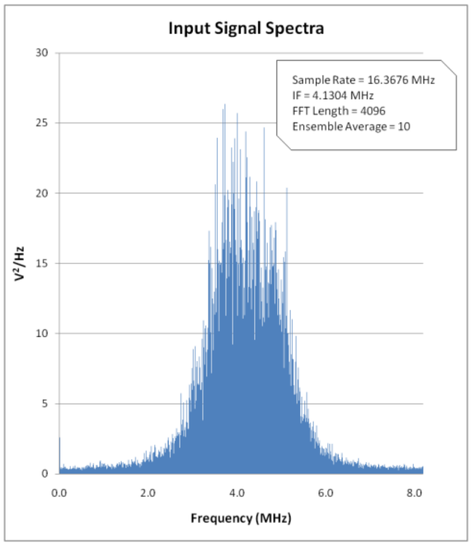

Figure 12—Frequency-domain view of input signal

Using the circular correlation approach as described in [15] the presence of PRN #18 was detected in the signal, and the initial code phase (time delay) was found. The location of the peak in Figure 13 corresponds to a code delay of about 917 μs.

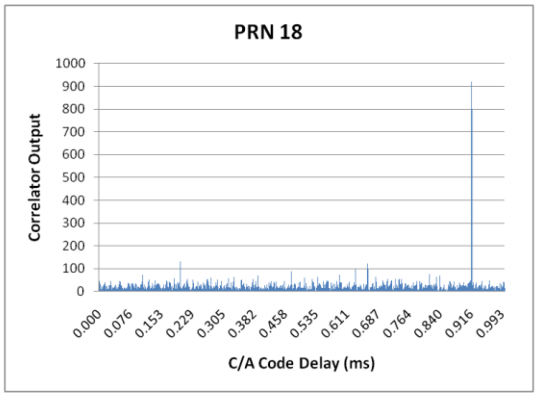

Figure 13—Correlation peak for PRN#18 detection

The initial carrier frequency estimate (IF + Doppler) was found by removing the C/A code from several milliseconds of sampled data and then frequency transforming the result, looking for a spectral peak. Figure 14 reveals a peak of 4.1304 MHz + 2584.0 Hz.

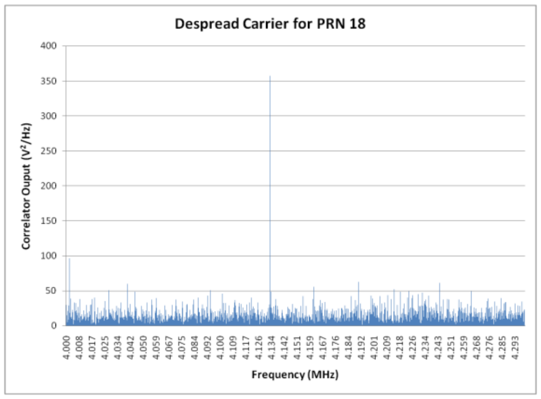

Figure 14—Frequency-domain view of carrier for PRN #18

For the purposes of tracking the signal, the initial code phase and carrier frequency estimates are passed to the pipeline components during initialization. A thread is created and attached to the main component collection for each PRN detected in the signal.

The output from the PLL, shown in Figure 15, is taken every millisecond and forwarded to the demodulator component. The normalized value is then converted to an appropriate binary one or zero at the expected data rate.

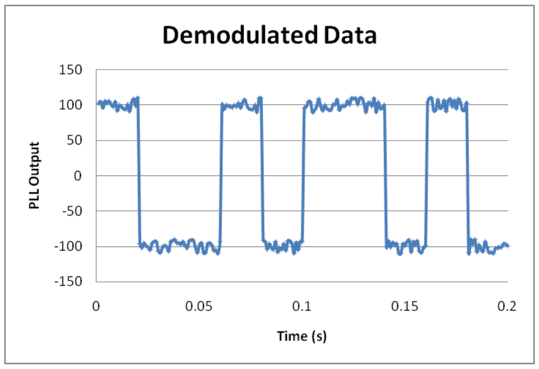

Figure 15—Navigation data signal from PLL output

At 50 bps and using the mapping of {-1, +1} → {1, 0} the signal of Figure 15 corresponds to the output of ten data bits {0110100101}.

FUTURE WORK & GENERAL CONSIDERATIONS

The tracking loop as tested and presented consists of only the most basic processing techniques and requires more sophisticated optimization and improvements to enhance the overall performance characteristics. However, by exploiting the integration intention of the framework, the methods of [7], or other better ideas for PLL and DLL tracking loops for example, can be incorporated.

The reference receiver needs the integration of almanac and other support material to aid in the initial detection process. There currently is no ability to make or incorporate in-view satellite predictions based on last known location and time-of-day information.

Many of the list or collection types in the framework could benefit from the definition of C# generics or templates to simplify the process of creation of new components.

The development of a receiver pipeline graphical design utility will eventually be included as an essential part of the framework. Such a utility will assist in the visualization of the interconnections between the components and their event sources.

Some future considerations for software receivers include the design of front-end hardware that downconverts the input signal to a frequency that is closer to baseband in order to reduce the workload on the PC processor, and at the same time increasing the sampling rate as a factor of the IF requirement. Simply satisfying the Nyquist rate for choosing the sampling rate is not sufficient for many control-related applications. Increasing the sampling rate improves the stability and tracking capabilities of DLL and PLL loops [10]. The differences in the stability analysis of analog phase-lock loops (APLLs) and their discrete counterparts (DPLLs) are discussed for first and second-order systems in [12]. A baseband software processor for GPS was developed and evaluated in [16] using an IF of ≈20 kHz and a sampling rate of ≈2 MHz.

Many of the texts and references separate their discussions on the process of satellite signal acquisition from that of acquired signal tracking. The usual software model is that acquisition finds the PRN sequences corresponding to satellite transmissions in the incoming signal, and then returns a collection of objects for the tracking loops to follow over time. Tracking each detected satellite on an individual dedicated thread appears to be a good idea, but there is a great deal of variability in the starting time of a thread from its point of creation. The purpose of accurately finding the sub-millisecond C/A-code delay in the acquisition stage is lost when it takes 1-5 milliseconds for the tracking thread to begin execution. The transition between acquisition and tracking, therefore, must be viewed as a basic change of state in a continuous operation rather than a step in a longer sequential process. Methods need to be developed that support the seamless transition from acquired signal to tracked satellite.

The time-domain versions of acquisition processes generally require an excessive length of time in order to run. Frequency-based circular correlation methods for finding the initial code-phase parameter will require the incorporation of the output from a hardware counter resource that can serve as a relative timestamp for the incoming samples. The current code delay may then be calculated from the initial delay provided by the acquisition stage by using the difference in the counter values. Newer PC system boards include a High-Performance Event Timer (HPET) for multi-media time reference purposes, which could also be used for front-end signal sampling and timing.

The allocation of large blocks of memory for storing signal samples takes time and processor clock-cycles to achieve. These “background” system activities are usually unaccounted for, but need to be recognized in the overall receiver workload.

Finally, the flexibility of the receiver framework allows for a binary file to serve as the source for an input signal. Configuring the signal classes with the required information on the signal properties, such as IF and sample-rate, could be done automatically if there were an agreement on the establishment of a binary file format that defines an embedded header for holding an information structure that includes byte-ordering (endian-ess), sample-rate, and other information required for cross-platform and hardware compatibility.

CONCLUSION

Due to the complexity of modern microprocessors used in PC-based computing systems, achieving real-time software GNSS receiver operation will require algorithms with high-degrees of parallelism and carefully designed interprocess synchronization strategies. The nature of this work requires more than an overall optimization effort on the part of application and algorithm implementers.

This paper has provided an introduction to the on-going development of a real-time software GNSS receiver research framework. The design of the system eliminates the need for parallel access to large signal data structures and the requirement of creating multiple copies of objects in memory for parallel access across multiple processes.

Using a block-diagram model and a pipeline signal processing approach, the framework allows the development and testing of both software and hardware concepts in a consistent unified manner. An object-oriented implementation maximizes the potential for component re-use and enhances the system’s extensibility benefits.

The interoperability features of the framework allow for the integration of multiple types of component implementations from different solution sources. Combining individual efforts in such a manner allows for the best-of-everything system development model necessary to meet required performance objectives.

REFERENCES

[1] M. Baracchi-Frei, G. Waelchli, C. Botteron, and P. Farine, "Real-Time Software Receivers: Challenges, Status, Perspective," *GPS World*, vol. 20, no. 9, pp. 40-47, September 2009.

[2] K. Borre, D. Akos, N. Bertelsen, P. Rinder, and S. H. Jensen, *A Software-Defined GPS and Galileo Receiver: Single-Frequency Approach*. Boston: Birkhäuser, 2007.

[3] A. Brown and J. Nordlie, "Integrated GPS/TOA Navigation using a Positioning and Communication Software Defined Radio," in *Position, Location, And Navigation Symposium, 2006 IEEE/ION*, April 25-27, 2006.

[4] F. Dovis, A. Gramazio, and P. Mulassano, "Design and test-bed implementation of a reconfigurable receiver for navigation applications," *Wireless Communications and Mobile Computing*, no. 2, p. 827–838, 2002.

[5] Jack K. Holmes, *Spread Spectrum Systems for GNSS and Wireless Communications*. Norwood, Massachusetts: Artech House, Inc., 2007.

[6] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.: Intel Corporation, 2009.

[7] D. Lu, Y. Zhang, S. Lee, and C. Chen, "Achieving Precise Real-Time GNSS Positioning with Software-based Receivers," in *ION GNSS 2009*, Savannah, GA, 22-25 September 2009.

[8] Pratap Misra and Per Enge, *Global Positioning System: Signals, Measurements, and Performance, 2nd Ed.* Lincoln, Massachusetts: Ganga-Jamuna Press, 2006.

[9] A. Mitelman et al., "Testing Software Receivers," *GPS World*, vol. 20, no. 12, pp. 28-34, December 2009.

[10] Charles L. Phillips and H. Troy Nagle Jr., *Digital Control System Analysis and Design*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1984.

[11] B. Sauriol and R. Landry Jr., "FPGA-Based Architecture for High Throughput, Flexible and Compact Real-Time GNSS Software Defined Receiver," in *ION NTM*, San Diego, CA, 22-24 January 2007.

[12] J. Sebesta, "Discrete-time Phase and Delay Locked Loops Analyses in Tracking Mode," *International Journal of Electronics, Circuits and Systems*, vol. 1, no. 4, Fall 2007.

[13] M. S. Sharawi and O. V. Korniyenko, "Software Defined Radios: A Software GPS Receiver Example," in *International Conference on Computer Systems and Applications, AICCSA '07, IEEE/ACS*, 13-16 May 2007.

[14] John Stankovic, "The Spring architecture," in *Proceedings - EUROMICRO '90 Workshop on Real Time*, Horsholm, Denmark, 1990, pp. 104-113.

[15] James Bao-Yen Tsui, *Fundamentals of Global Positioning System Receivers: A Software Approach*. New York: John Wiley & Sons, Inc., 2000.

[16] W. Zhuang, "Composite GPS Receiver Modeling, Simulations, and Applications (Ph.D. Disertation)," Fredericton, NB, Thesis 5008, 1992.

[17] N.I. Ziedan, *GNSS Receivers for Weak Signals*. Norwood, MA: Artech House, Inc., 2006.