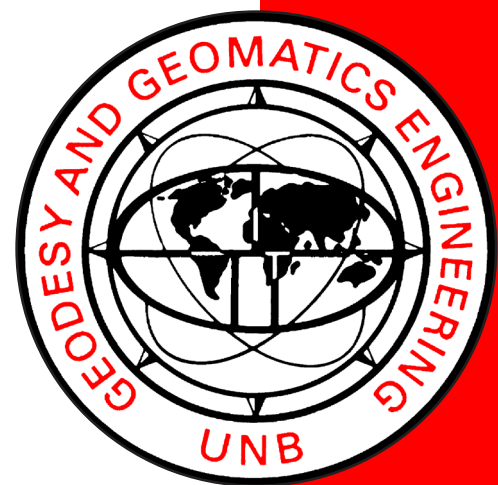


A REAL-TIME SOFTWARE GNSS RECEIVER DEVELOPMENT FRAMEWORK

DOUGLAS A. GODSOE

April 2010



A REAL-TIME SOFTWARE GNSS RECEIVER DEVELOPMENT FRAMEWORK

Douglas A. Godsoe

Department of Electrical and Computer Engineering
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada
E3B 5A3

April, 2010

© Douglas A. Godsoe, 2010

PREFACE

This technical report is a reproduction of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical and Computer Engineering, April 2010. The research was supervised by Mary Kaye, Department of Electrical and Computer Engineering and Dr. Richard Langley, Department of Geodesy and Geomatics Engineering and funding was provided by the Natural Sciences and Engineering Research Council of Canada.

As with any copyrighted material, permission to reprint or quote extensively from this report must be received from the author. The citation to this work should appear as follows:

Godsoe, D.A. (2010). *A Real-Time Software GNSS Receiver Development Framework*. Ph.D. dissertation, Department of Electrical and Computer Engineering, published as Technical Report No. 273 by the Department of Geodesy and Geomatics Engineering, University of New Brunswick, Fredericton, New Brunswick, Canada, 256 pp.

DEDICATION

To Pam, for encouraging me to start and supporting me as I finished. Thanks especially to Victoria, for pork chop sandwiches and always making me laugh. And especially to Allan, for being interesting when I needed a distraction, but stop taking all the ammo.

ABSTRACT

This dissertation provides the architecture and describes the development effort of a modular software-based real-time global navigation satellite system (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.

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.

PREFACE

When I began my thesis research I wanted to do work with digital-system design and development. I had a few ideas of specific areas that I intended to explore, such as system on a programmable chip (SoPC) and hardware/software co-design focusing on the practical applications of scientific theory for the development of buildable solutions. The reason I went into engineering in the first place is that I like to see designs implemented; the tangibility of seeing your work constructed and your ideas realized has a large appeal to me.

I needed to find a field of scientific research that used in some practical way the results of digital system design, so I took an interdisciplinary approach of trying to find a problem for a solution technology. I peaked in on the Chemistry department to find out what was going on in that realm with regards to electromagnetic chemical analysis, but as I listened to their explanations, I became uncomfortably aware that while I was familiar with all of the words they were using to describe their work, I had never heard them in that particular order and all together like that before, so I realized that I would be spending most of my time just trying to figure out the pertinent background material and very little on the digital design parts of the problem.

GPS uses a similar spread-spectrum signaling technology used by CDMA cell phones and wireless Ethernet. The basic signals concepts are not that difficult to understand, the solutions combine various aspects of both hardware and software, and as a bonus there's no obvious controls theory to any of it. As far as digital-system applications go, GPS is as

good as any other. Besides, recovering messages from space is kind of cool and people can relate to the work when you tell you're building a GPS thing.

Narrowing down the application area was one thing, but finding a contribution to make is something else. The work couldn't just be a project for the sake of building something; it had to have meaning and value to other people doing similar work. At first I thought of designing and implementing a multi-channel correlator using an FPGA and an assortment of hardware functional blocks, but I soon found out that several dozens of low-cost commercial ICs are available that do this part, already.

In order to get started with some form of GPS research, I looked for the work of others that I could reproduce and possibly enhance or extend. They all required some piece of special hardware or software, or the specific implementation details were so obscured that reproducing the results would be impossible. The really difficult thing in working with spread-spectrum communications is that when things are working properly everything looks like noise, just as it does when things are working improperly, and in practice it's hard to tell the two situations apart.

The idea for the design of the framework came from the realization that there would be a significant benefit to having a reference receiver to act as a starting point for future development work. All of the pieces would be implemented in a way that they could be readily customized to accommodate the integration of components from different sources.

After the initial dissertation proposal, a certain member of the supervisory committee suggested to me that perhaps the project was too big to be done alone—receivers are

complicated things and the reason that researchers tend to concentrate on one area is that it's too much work for one person to do the whole thing by themselves. Not knowing any better, I argued convincingly that it couldn't really be that bad. I should have let myself be talked down from that tree. Nothing works the way the theory in the books describes it does, and to make life worse, the essentials of the receiver are phase-lock and delay-lock loops in mutual feedback control with one another, so I spent hours revisiting the controls material I had hoped to avoid after all—I wanted neither signals or controls and I got both, instead.

In all of the literature, nobody really said how they did it. They have great ideas and algorithms on acquisition, but nothing but hand waving on implementing carrier tracking loops. What information was available was flawed, conflicted, and incomplete. Trying to re-implement someone else's research required having access to their code, their hardware, and their development and test environments. Also, since the work was never a complete receiver, it would have to be integrated with something else in order to get useable results. Each new report of something similar had to be looked at and picked apart to see if it really did what was claimed. Most of them were optimizations of offline receivers.

An array of MATLAB-based offline post-processing applications exists. However, to me, MATLAB obscures the details behind some of the important parts of an implementation. While it makes some things easier, understanding the details of how a particular filter or transform is implemented is important sometimes, and without that understanding applying the results obtained is pretty hard. Also, MATLAB functions on a

Java virtual machine so the performance is wanting, even for offline applications, let alone meeting any sort of real-time target.

The implications of not utilizing MATLAB for the work are numerous and inconvenient. MATLAB provides graphing and plotting capabilities that are time consuming to reproduce. There are also the pieces in the solution that will receive only a one-sentence mention that took many weeks to develop, test and debug. The *Complex* types and the DFT/FFT routines, for example, involved much testing and coding effort and are really only support functions for the receiver framework. There really has been no new work on the FFT since Cooley and Tukey, and what is represented as new is largely plagiarized directly from (1), which itself is adapted from the original version in FORTRAN—two-character variable names and all. No, seriously, go to your favorite search page and search for the following exact phrase and just see how many results are obtained:

“// here begins the Danielson-Lanczos section”

Even though I had tried to avoid signals and controls, learning that the z-transform shares the same generating function as a shift register for the PRN sequences, and that discrete control feedback loops can be analyzed and implemented in a similar manner to digital filters for signal processing was revealing. I’ve also noticed that the state feedback equations used in controls look like the representation of a Mealy state machine in digital logic systems. There is something intellectually reassuring in these similarities and representational dualities, if for no other reason than it is a form of brain-cell reuse.

I've chosen to include code that supports the theory or ideas as implementation examples. The books that I found most useful were the ones that included coding examples, even if they were in C, FORTRAN, or some flavor of BASIC. These were easy to translate into any language, and they made solid the abstract concepts the text was trying to present. References such as (2) provide no code or example implementations to support the work, and no matter how useful it was at some point in time, the development effort is largely lost forever. C# shares much of the language syntax of C/C++, so examples are included that have little to no dependency on the functions of the .NET base class library (BCL) for operational support. If a BCL function is required in the example, the operations are either well commented or the behavior of the code is evident from the names of the classes and methods invoked.

The Receiver Development Framework described in this thesis is really just a starting point for the exploration of signals-related control applications. The pipeline model and interoperability support features allow for a wide range of ideas to be tested, evaluated, and integrated into a huge variety of working solutions.

There's nothing like a fact you've learned yourself; believe in your own experiences.

ACKNOWLEDGEMENTS

I'd like to thank my supervisors, Prof. Kaye and Dr. Langley, for their support and guidance over the past few years. Dr. Tervo and Dr. Diduch have always been willing to make themselves available to help me out of the occasional cognitive bind, and for that I am greatly appreciative.

Table of Contents

| | |
|--|------|
| DEDICATION | ii |
| ABSTRACT | iii |
| PREFACE | iv |
| ACKNOWLEDGEMENTS | ix |
| Table of Contents | x |
| List of Tables..... | xiv |
| List of Figures | xv |
| Acronyms and Abbreviations..... | xxii |
| Chapter 1 Introduction | 1 |
| 1.1 Overview and Background | 4 |
| 1.2 Previous Work | 7 |
| Chapter 2 GPS Operation..... | 14 |
| 2.1 Trilateration | 14 |
| 2.2 System Overview | 16 |
| 2.2.1 System Architecture | 17 |
| 2.2.2 Signals | 18 |
| 2.2.3 Transmitted Data..... | 24 |
| 2.2.4 Position Determination..... | 26 |
| 2.2.5 Receiver operation | 28 |
| Chapter 3 Spread-spectrum Fundamentals | 30 |
| 3.1 Spread-Spectrum Types | 30 |
| 3.1.1 Frequency Hopping..... | 31 |
| 3.1.2 Direct Sequence | 32 |

| | | |
|---|--|-----|
| 3.2 | Transmitter and Receiver Architecture | 33 |
| 3.2.1 | Modulation | 33 |
| 3.2.2 | Demodulation | 34 |
| 3.3 | PRN Sequences and Generators | 35 |
| Chapter 4 Object-Oriented Analysis and Design | | 37 |
| 4.1 | Encapsulation..... | 37 |
| 4.2 | Inheritance | 41 |
| 4.3 | Polymorphism..... | 42 |
| 4.4 | Special Items | 45 |
| Chapter 5 Real-time Systems | | 47 |
| 5.1 | Definition of Real-time | 48 |
| 5.2 | Applications, Processes, and Threads | 52 |
| 5.3 | Scheduling | 55 |
| 5.4 | Synchronization | 64 |
| 5.5 | Architectural Modeling and Languages for Real-time | 68 |
| Chapter 6 Development Framework Overview | | 77 |
| 6.1 | Receiver Framework Architecture Diagram | 79 |
| 6.2 | Pipeline Processing Model | 83 |
| 6.2.1 | Synchronous Pipeline..... | 85 |
| 6.2.2 | Asynchronous Pipeline..... | 87 |
| 6.2.3 | Pipeline Component | 90 |
| 6.2.4 | Pipeline Container..... | 94 |
| 6.2.5 | Phase-Lock Loop Pipeline Component..... | 96 |
| 6.2.6 | Delay-Lock Loop..... | 103 |
| 6.2.7 | Numerically Controlled Oscillator..... | 108 |
| 6.2.8 | Data Demodulator | 110 |
| 6.2.9 | Signal Controller..... | 112 |
| 6.2.10 | PRN Code Generation | 114 |
| 6.3 | Common Types..... | 116 |
| 6.3.1 | Complex | 117 |
| 6.3.2 | DFT | 117 |
| 6.3.3 | FFT | 117 |
| 6.3.4 | Filter Classes..... | 118 |
| 6.3.5 | Frequency | 118 |

| | |
|---|-----|
| Chapter 7 Interoperability Support | 119 |
| 7.1 Interoperability Requirements | 119 |
| 7.2 Interoperability Layered Model..... | 121 |
| 7.3 Interoperability Scenarios | 125 |
| 7.3.1 Single Process..... | 126 |
| 7.3.2 Interprocess..... | 127 |
| 7.3.3 Interprocess with Remote Execution | 129 |
| Chapter 8 Signal Source Device Driver Interface | 131 |
| 8.1 Layered Device Driver Approach..... | 132 |
| 8.1.1 Signal Source Device Interface | 140 |
| 8.1.2 Signal Source Base Class | 142 |
| 8.1.3 Signal Source Derived Classes | 143 |
| 8.2 Device Interface State Models..... | 143 |
| 8.2.1 Signal Source System State Model..... | 143 |
| 8.2.2 State Model for USB Signal Source | 144 |
| Chapter 9 Acquisition and Tracking..... | 147 |
| 9.1 Acquisition | 147 |
| 9.2 Tracking | 151 |
| Chapter 10 Reference Implementation Results..... | 152 |
| 10.1 Pipeline Testing Configuration..... | 153 |
| 10.2 Real-time Performance Evaluation..... | 163 |
| 10.3 Interoperability Component Integration..... | 173 |
| Chapter 11 Conclusion | 182 |
| Appendices..... | 188 |
| Introduction to the Appendices | 188 |
| <i>Appendix A</i> — 3 rd -party Toolkit Interoperability | 189 |
| Library Background and History..... | 189 |
| Component Object Model..... | 193 |
| The .NET Framework..... | 197 |
| Integration of GPSTk..... | 198 |
| Additional Features..... | 207 |

| | |
|---|-----|
| <i>Appendix B— Tracking-loop Control Theory</i> | 211 |
| <i>Appendix C— Finite Fields and SSRGs</i> | 220 |
| <i>Appendix D— Background Signals Theory</i> | 238 |
| References | 245 |
| Index | 256 |
| Curriculum Vitae | |

List of Tables

| | |
|--|-----|
| Table 6-1—Typical PLL discriminator functions..... | 101 |
| Table 6-2—Typical DLL discriminator functions | 106 |
| Table 10-1—Visible satellites extracted from file captured on June 5, 2009 at 13:16 UTC..... | 162 |
| Table 10-2—Visible satellites extracted from file captured on June 10, 2009 at 15:52 UTC..... | 163 |
| Table 10-3—Performance testing on 40 seconds of data from the SiGe EK3 front-end hardware | 164 |
| Table 10-4—Satellite tracking results with a simulated signal | 171 |
| Table 10-5—Performance testing on 40 seconds of data from the simulated signal model | 172 |
| Table C-1—The equivalence between XOR and multiplication with $\{0, 1\} \leftrightarrow \{+1, -1\}$ mapping | 222 |
| Table C-2—Example 7-bit autocorrelation calculations. Each row represents one additional bit delay. | 232 |
| Table C-3—Correlation results..... | 232 |
| Table D-1—Relationships between the properties of the time and frequency domains | 239 |

List of Figures

| | |
|---|----|
| Figure 2-1—Trilateration: Determining the range to three transmitters at known locations permits an observer to calculate their position, point P | 15 |
| Figure 2-2—GPS Frequencies and Codes: Note that the L5 frequency and the L2C and M codes are only active on some satellites | 22 |
| Figure 2-3—C/A Code Generation: The C/A code is formed by the product of two sequences, G1 and G2 | 23 |
| Figure 2-4—GPS Navigation message structure | 25 |
| Figure 2-5—Fixing a position requires finding the pseudoranges to at least four satellites | 28 |
| Figure 2-6—GPS receiver block diagram | 29 |
| Figure 3-1—Frequency Hopping Spread Spectrum: The total bandwidth available is divided into multiple channels, and each channel is occupied randomly in turn by the modulated carrier signal for a short interval of time | 32 |
| Figure 3-2—Direct Sequence Spread Spectrum: The transmitted carrier frequency determines the position of the center of the spectrum, while the width (spreading) is determined by the chip rate (R_c) | 33 |
| Figure 3-3—BPSK DSSS Transmitter: The input data, $d(t)$, is combined with the carrier and then binary phase shift keyed with the pseudo-noise sequence | 34 |
| Figure 3-4—Direct Sequence BPSK receiver model | 35 |

| | |
|--|----|
| Figure 4-1—Static UML object model for a parking application..... | 39 |
| Figure 4-2—UML object-model showing inheritance..... | 42 |
| Figure 4-3—Each class derived from vehicle implements a specialized polymorphic Park method | 43 |
| Figure 4-4—An interface declaration and a class that implements it | 45 |
| Figure 6-1—Block diagram model of the Receiver Development Framework..... | 80 |
| Figure 6-2—CPU Workload with a capture-then-process signal processing approach..... | 84 |
| Figure 6-3—Pipeline structure with a common clock | 85 |
| Figure 6-4—Event-driven synchronous pipeline process | 86 |
| Figure 6-5—Asynchronous software pipeline model using event coupling between successive stages | 88 |
| Figure 6-6—Pipeline stage 1 event-handler structure with separate worker threads..... | 89 |
| Figure 6-7—Pipeline component object model..... | 90 |
| Figure 6-8—Example pipeline configuration showing feed-forward and feedback control objects with parallel pathways..... | 92 |
| Figure 6-9—PipelineContainer class diagram..... | 95 |
| Figure 6-10—A basic phase-lock loop | 97 |
| Figure 6-11—PI controller as the filter function for a PLL | 97 |

| | |
|--|-----|
| Figure 6-12—PLLPipelineComponent class diagram | 102 |
| Figure 6-13—DLL E, P, L correlator outputs under on-time (a), early (b), and late conditions (c) | 104 |
| Figure 6-14—DLL correlator block diagram | 105 |
| Figure 6-15—DLLPipelineComponent class diagram..... | 107 |
| Figure 6-16—NCOPipelineComponent class diagram | 109 |
| Figure 6-17—UML static object model for the demodulator component..... | 110 |
| Figure 6-18—SignalController object model | 113 |
| Figure 6-19—UML static object model for PRN code generators | 115 |
| Figure 7-1— Interoperability Layer..... | 121 |
| Figure 7-2—Layered Interoperability Model | 122 |
| Figure 7-3—Single process interoperability function call: custom marshaler in A, system marshaler in B..... | 127 |
| Figure 7-4—Interprocess with common data types and system marshaler | 128 |
| Figure 7-5—Interprocess interoperability between two systems with remote code execution ... | 129 |
| Figure 8-1—Layered Device Driver Model | 132 |
| Figure 8-2—LibUSB library function for starting a USB device. | 133 |
| Figure 8-3—Exported USB functions from Layer-2 | 134 |

| | |
|--|-----|
| Figure 8-4—Implementation of the Layer-2 USB function for device initialization..... | 134 |
| Figure 8-5—Layer-4 device wrapper declaration for the GN3S device driver | 135 |
| Figure 8-6—Layer-4 wrapper code implementation for the GN3S device driver initialization sequence | 136 |
| Figure 8-7—Abstract signal base class implementation UML static structure diagram..... | 137 |
| Figure 8-8—Explicit interface implementation | 139 |
| Figure 8-9—Generic signal source UML Statechart model | 143 |
| Figure 8-10—USB signal source internal state model..... | 145 |
| Figure 8-11—USB device state model as mapped into the system state model..... | 145 |
| Figure 9-1—Software-based signal determination | 148 |
| Figure 10-1—Tracking pipeline configuration used for testing | 154 |
| Figure 10-2—Time-domain view of input signal source (a) and input signal histogram (b) | 157 |
| Figure 10-3—Frequency-domain view of input signal | 158 |
| Figure 10-4—Correlation peak for PRN#18 detection | 159 |
| Figure 10-5—Frequency-domain view of recovered carrier for PRN #18..... | 159 |
| Figure 10-6—Navigation data signal from PLL output | 161 |
| Figure 10-7—Input I/Q signal from the simulated signal source used for testing..... | 167 |
| Figure 10-8—Double sided spectrum for simulated signal source..... | 168 |

| | |
|--|-----|
| Figure 10-9—Circular correlation peak detection using simulated signal source | 169 |
| Figure 10-10—Frequency-domain view of carrier using the simulated signal source | 170 |
| Figure 10-11—Time-domain view of recovered carrier using the simulated signal source..... | 171 |
| Figure 10-12—Precompiled header file stdafx.h used for the GNURadioParts library project .. | 176 |
| Figure 10-13—Header file GNURadioParts.h with the <i>UpdateOutput(...)</i> function exported from the GNURadioParts library project | 177 |
| Figure 10-14—CPP source file GNURadioParts.cpp showing the <i>UpdateOutput()</i> function implementation | 178 |
| Figure 10-15—CPP source file DllMain.cpp with the <i>gr_pll_refout_cc</i> instance initialization.. | 179 |
| Figure 10-16—C# source file GNURadioWrapper.cs that imports the GNURadioParts library | 179 |
| Figure 10-17—C# source file GNUPLLPipelineComponent.cs for invoking the GNURadioWrapper <i>UpdateOutput()</i> static method..... | 180 |
| Figure 10-18—PipelineContainer PLL member declaration and initialization for GNUPLLPipelineComponent class integration | 181 |
| Figure A-1—Acrobat Access system registry entry | 194 |
| Figure A-2—Acrobat Access version specific program ID | 194 |
| Figure A-3—Acrobat Access class ID key | 194 |
| Figure A-4—Acrobat Access class ID value..... | 195 |
| Figure A-5—Acrobat Access InprocServer32 sub-key..... | 195 |

| | |
|--|-----|
| Figure A-6—Acrobat Access executable file registry entry..... | 195 |
| Figure A-7—Visual Studio 2008 new project dialog | 200 |
| Figure A-8—Visual Studio 2008 new project dialog | 201 |
| Figure A-9—Exported code symbol C macro | 201 |
| Figure A-10—C++ class code exported using API macro | 202 |
| Figure A-11—Dialog for adding a DLL module export definition file | 203 |
| Figure A-12—Definition file exported symbols | 203 |
| Figure A-13—API Macro exported symbols using C naming styles..... | 204 |
| Figure A-14—C++ exported function implementation..... | 205 |
| Figure A-15—Exported symbols using C naming styles, without the use of the API macro | 205 |
| Figure A-16—Exported symbols using C naming styles | 205 |
| Figure A-17—C# class for accessing the functions exported from the GPSTk library | 206 |
| Figure A-18—C# event handler that invokes functions from the external GPSTk library | 207 |
| Figure A-19—C++ code for initializing a persisted class instance..... | 209 |
| Figure A-20—C++ code for accessing a persisted class instance | 209 |
| Figure B-1—PLL feedback control model..... | 213 |
| Figure B-2—Sample and hold representation | 214 |
| Figure B-3—PLL 1 st -order filter..... | 215 |

| | |
|---|-----|
| Figure B-4—A simple transfer function | 215 |
| Figure B-5—C# code for implementing the 1 st -order filter of Figure B-3 | 217 |
| Figure B-6—Arctangent discriminator function over the range $[-\pi, \pi]$ | 218 |
| Figure B-7—Product discriminator function over the range $[-\pi, \pi]$ | 218 |
| Figure B-8—Recovered carrier waveform, after code removal | 219 |
| Figure C-1—Fibonacci implementation of P(x) | 223 |
| Figure C-2—Galois implementation of P(x) | 224 |
| Figure C-3—GNSS implementation of P(x) | 224 |
| Figure C-4—SSRG configuration | 227 |
| Figure C-5—Example shift register | 229 |
| Figure C-6—Non-normalized autocorrelation | 232 |
| Figure C-7—Length-31 Gold code autocorrelation function | 233 |
| Figure C-8—GPS C/A-code generator configuration | 234 |

Acronyms and Abbreviations

| | |
|-------|---|
| ADL | Architectural description language |
| AGC | Automatic gain control |
| API | Application Programming Interface |
| BCL | Base class library |
| BPF | Band-pass filter |
| BPSK | Binary Phase-shift Keying |
| C/A | Coarse Acquisition |
| CCS | CORBA Component System |
| CCS | Calculus of communicating systems |
| CDMA | Code-division Multiple-access |
| CISC | Complex instruction set computer |
| CORBA | Common Object Resource Broker Architecture |
| CPU | Central processing unit |
| CSP | Communicating sequential processes |
| DDLL | Discrete delay-lock loop |
| DFT | Discrete Fourier Transform |
| DLL | Delay-lock loop or Dynamic-link library |
| DPLL | Discrete phase-lock loop |
| DSP | Digital Signal Processing or Digital Signal Processor |
| DSSS | Direct-sequence spread-spectrum |
| FFT | Fast Fourier Transform |
| FHSS | Frequency hopped spread-spectrum |

| | |
|------|--|
| FPGA | Field Programmable Gate Array |
| GNSS | Global Navigation Satellite System |
| GPS | Global Positioning System |
| HDL | Hardware description language |
| HPET | High-performance event timer |
| HTTP | Hyper-text transfer protocol |
| I, Q | In-phase (signal), Quadrature-phase (signal) |
| LNA | Low-noise amplifier |
| LPF | Low-pass filter |
| MMX | Multimedia extensions |
| NCO | Numerically controlled oscillator |
| ODBC | Open database connectivity |
| OLE | Object linking and embedding |
| OMT | Object modeling technique |
| O-O | Object-oriented |
| OS | Operating system |
| PLL | Phase-lock loop |
| PRN | Pseudo-random noise |
| RF | Radio-frequency |
| RISC | Reduced instruction set computer |
| RPC | Remote Procedure Call |
| RUP | Rational Unified Process |
| SDR | Software Defined Radio |
| SIMD | Single instruction multiple data |
| SOPC | System on programmable chip |

| | |
|-------|--|
| SSE | Streaming SIMD extensions |
| SSRG | Sequential serial shift register generator |
| TOA | Time of arrival |
| UML | Unified modeling language |
| VHDL | VHSIC hardware description language |
| VHSIC | Very high-speed integrated circuit |
| XML | Extensible mark-up language |

Chapter 1 Introduction

There are many expected and anticipated advantages of software Global Navigation Satellite System (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 a Programmable Chip (SoPC) philosophy, using one or more soft-core

processors in combination with various application specific logic blocks, bring the features and performance benefits of both hardware and software. However, they also suffer all the combined development 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 chipsets 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.

Eventually, everything falls out from having the local code and carrier precisely aligned with the received signal. As developed and tested, with a 2.4 GHz Pentium 4 Quad-core processor working in conjunction with a front-end sampler sampling at approximately 16 MHz, it first appears that there should be enough processing resources to manage the amount of work required. The processor clock is roughly 150x faster than the sampling-rate, which really implies, though, that there is only 150 clock-cycles worth of time in order to process one sample. The situation worsens when the level of application parallelism is not ideal (Chapter 5), and the multiple-processors are doing

other work unrelated to processing the input signal. Work such as running the operating system, responding to user input, and memory management require clock-cycles that need to be accounted for.

A real-time solution will require the use, management, and synchronization of multiple threads of processing. The synchronization and scheduling problems are non-trivial issues that cannot be simply dismissed.

The key features of this framework are:

- that it uses a high-level development language (C#) and feature-rich run-time environment;
- that it defines and implements a unique software-based pipeline processing model;
- that it is a flexible and easily adapted object model providing well-defined functional interfaces;
- that it provides an interoperability layer that directly supports integration with 3rd-party tools and other external software and hardware.

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.

The Receiver Development Framework is the outcome of an analysis process that was an attempt at standing back and taking a holistic view of the overall approach to GNSS

receiver application research and development. The significance of this work lies in the establishment of the collection of object models and base implementations for real-time receiver development. Without it, there is a limitation to the degree of improvement to software receiver performance that can be made through individual optimization efforts alone. By adopting the principles and integration philosophies embodied and presented in this work, world-wide efforts can be combined into a unified development model, which has the potential for enhancing researcher productivity through the reduction of redundant non-value-added activities.

1.1 Overview and Background

Global Navigation Satellite System (GNSS) is the general term given to the process of identification of user position through the relative location of known orbiting satellite platforms. There are currently two operational GNSS services: the Global Positioning System (GPS) funded by the US Department of Defense and the **Global'naya Navigatsionnaya Sputnikova Sistema**, which translates to Global Navigation Satellite System (GLONASS), developed by the former Soviet Union and now supported by the Russian Federation. Both GPS and GLONASS are operated under the joint control of military and civil agencies.

The European Union (EU) in collaboration with the European Space Agency (ESA) is currently in the process of deploying the **Galileo** satellite-based navigation system. Galileo is expected to be operated under the control of a civilian agency, but will undoubtedly also be used by military authorities. The Chinese have developed and deployed a regional satellite navigation system known as **BeiDou**, that unlike GPS,

interacts with the user to determine a position estimate. Another Chinese system—**Compass** or **BeiDou-2**, similar to GPS, is currently under development.

While the modulation and data encoding methods may differ in their specific implementations, GLONASS and Galileo are largely variations of GPS and share many similarities. As a result, mixed-constellation receivers have been developed that interoperate with navigation data from these systems.

Software Defined Radio (SDR) is a broad term that can apply to the different aspects of transceiver functions. SDR makes use of software digital signal processing (DSP) techniques to create or receive data streams that are converted to/from the analog domain as close as possible to the antenna in the signal path. For a receiver, the intelligence information or message from the carrier is extracted by means of extensive processing in software.

GNSS receivers based on SDR allow a flexible, customizable, and easily extensible solution in which the future requirements for system operation can be altered or updated after the system has been deployed. For example, a multi system, multiple constellation, SDR GNSS receiver could be developed that would allow the resolution of position from a mixed set of visible satellites when there is an insufficient number visible from a single system. A multiple-constellation system can provide higher positioning accuracies than a single constellation system, even when there is a sufficient number of satellite signals available from a single constellation.

The approach most often taken to software-centric signal processing is based on Fourier transform analysis. Large data sets in the time domain are captured from an input

signal sampling hardware device and then converted to the frequency domain through a Discrete Fourier Transform (DFT) or its fast (FFT) alternative. Once in the frequency domain, more complicated operations such as correlation and convolution can be replaced by multiplication. However, this approach is very difficult to make work in a real-time manner, especially at high sample rates and in situations where the signal needs to be processed for more than one data stream from more than one transmitting source. Analyzing signal data in parallel requires either that all tracked sources have access to copies of the incoming data stream, or that they have access to a shared buffering data structure. Maintaining multiple copies of the data can create memory resource issues, and sharing the data requires a synchronized or thread-interlock mechanism that limits peak system performance. Adding to this complexity is the further need of establishing a common reference time base or source that the data demodulators can work from, which serves only to increase the design headaches.

While frequency-transform features have been supplied, the Receiver Development Framework (RDF) takes an adaptable and flexible asynchronous pipeline approach to working with each sample as it arrives in the discrete time domain for most of the signal processing activities. Software representations of block diagrams for hardware-based receiver components can be developed and “plugged in” to the framework pipeline, making the development effort easier and faster. More importantly real-time performance has been considered essential and is achievable with the pipelined approach.

The framework provides implementations of many adaptable pieces that can be used to build a rich variety of functionality. The interoperability features of the framework support the direct integration of custom or commercial hardware and software

components at any point in the processing sequence. As a result, components can be described and simulated using software representations for baseline performance characterizations. Then, hardware versions can be synthesized using an HDL and FPGA toolset using parameters derived from the software components. These hardware pieces can be subsequently connected back into the pipeline, replacing the software component, so that performance comparisons for improvements from the baseline can be made. Alternatively, existing libraries of software can be directly integrated with the framework for testing without the need to rewrite previously tested code.

The framework was developed and tested with a GPS receiver implementation for measuring the required observables of code and carrier phase, and carrier frequency (Doppler) necessary for a pseudorange measurement. However, extensions to the framework are possible that would make it suitable for a wide range of applications, such as audio/video signal processing and feedback control systems.

The implementation of the framework has been developed using the Microsoft .NET Framework version 3.5 and the C# development language, with some device drivers written in C. However, as described, the framework could be developed in other languages and operating environments. The design is object-oriented, so tools and languages that directly support an object-based development paradigm will be more suitable for future adaptations.

1.2 Previous Work

While a software-only approach to GNSS has been predicted, highly anticipated, and widely researched for some time, only recently has the capability of general purpose

CPUs been up to the required signal processing performance demands. Recent publicly-disclosed SDR GNSS attempts, however, are non-real-time, have specific hardware dependencies, and often work with a single system on a single frequency. Other soft-GPS solutions are proprietary or largely commercial endeavors that poorly support the open environment needed for receiver research.

General background information on the design and operation of GPS receivers can be found in (3) and broader but related details on spread-spectrum communication systems in (4). Additional information on software signal processing methods are provided in (5), (6) and (7). These references develop some of the necessary theory for software-based GNSS receivers.

Many solutions are intended more for modeling the behaviors of individual system functional blocks for hardware implementations, and lack a whole-system approach for performance analysis. Taking a direct block-diagram approach towards receiver design, and then constructing it using iterative frequency-domain transformations, bypasses the opportunity for discrete optimizations and yields non-real-time performance characteristics. Solutions such as (8) possess real-time behaviors, but are based on specialized hardware chipsets, support only GPS, and lack extensibility for dual-frequency support.

Papers such as (9) discuss the current research activities, and provide some guidance to the architectural requirements, but lack documentation on any substantive implementations. The work discussed in (10) is an FPGA-based real-time GPS receiver connected to a PC for graphical display of the results. While solutions that include an

FPGA fabric for signal processing acceleration can be regarded as “reconfigurable,” modifications to the systems require non-trivial tools and skills. The limited extensibility of these systems negatively impacts their flexibility for use in future, wider research areas.

Reference (11) uses the SIMD MMX (single instruction multiple data multimedia extensions) instructions on the Intel x86 processor. By using C++ and inline assembler, the code gains a 70% performance improvement—which demonstrates that it is possible with targeted optimizations to achieve the necessary work throughput from a PC processor. However, the implementation does not represent a flexible application framework for GNSS research.

References (5) and (12) are non-real-time projects in that they are offline batch-oriented processes, requiring specific proprietary hardware. These are predominantly MATLAB simulations for functional analysis purposes. The work presented in (13) is a proprietary library of GPS receiver pieces and not a complete general-purpose solution framework.

Article (14) describes Galileo acquisition software techniques in a non-real-time environment and no concrete implementation details are provided. (15) represents the development and testing of a hardware simulator using non-real-time software techniques. (16) is also a non-real-time MATLAB GPS simulator for a single satellite signal.

The works of (17) and (18) are hardware-dependent FPGA-based special-purpose solutions and not flexible frameworks for further development. (17) is an attempt to

improve or offset limitations in GPS precision under weak-signal conditions using SDR for time-of-arrival (TOA) corrections, and (18) is a proposed reference implementation of a prototype SDR + FPGA architecture GNSS receiver.

References (2) and (19) develop the mathematical models for digital delay and phase-lock loops (DDLL and DPLL) for use in GPS receivers and present software simulations of the results. In particular, (2) states that the work included a software baseband receiver implementation, but it is developed for a simulated signal only and ignores the effects of the navigation message. The work of (20) was a study on post-processing data from an early unconventional civil GPS receiver, Satellite Emission Range Inferred Earth Surveying (SERIES), and was similar, in some respects, to other such data post-processing efforts at UNB.

Many of the so-called real-time systems have been developed more for real-time signal simulation graphing and plotting functions for the purposes of validation and verification. For others, the real-time performance attribute has never been demonstrated operationally, only modeled and simulated, such as the systems described in (21) and (22).

On-going research at the University of New South Wales (UNSW) Satellite Navigation and Positioning (SNAP) lab is taking the direction of developing combination hardware (FPGA) and software (C) receiver implementations (23) that are targeted for a single hardware configuration.

At the 2008 International GNSS Service (IGS) Workshop in Miami Beach, recommendations made in a presentation by researchers from Cornell University and

University FAF Munich (24) seem to support the need for further development of open software GNSS receivers. The presentation encourages the definition of an IGS-sponsored software receiver and the establishment of an IGS format for exchange of data among software receivers, and identifies the need for benchmark comparisons of software receiver performance to traditional commercial hardware devices.

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 (25) and issues with their testing protocols and challenges covered nicely in (26).

The variety of MATLAB-based solutions, such as those presented in (16) and included in (5), like the examples provided by (6) and (7), 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 (18), or the software defined radio receivers for GPS and GNSS discussed in (10) and (17) are cobbled together from the complex interconnection 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. In (27), a real-time 12-channel SIMD software correlator is described, but the work does not identify any of the challenges or their solutions for managing the necessary level of processing parallelism.

The contributions and key differences of the work presented in this dissertation from the previously identified literature are that

- it is entirely a software solution, with the exception of the hardware front-end, that does not rely on accelerator devices to function;
- it is an object-oriented solution architecture that defines the generalized set of interfaces and base-class implementations for future receiver development;
- it presents a working view of real-time systems and identifies the issues associated with managing shared resources across multiple processing pathways;
- it establishes a pipeline model for software-based signal-processing work that can be customized and readily adapted for use in a broad range of signal and control-related applications.

Most notably, the basis of the reference implementation of the framework is the signal processing methods presented in (5), but with greater consideration given in the

design of the modules and components for how these things will be shared and improved in the future. Rather than merely providing a library of code that others will need to pull apart to extricate and augment, the object-oriented nature of the signal processing pipeline components facilitates a much simpler approach to customization and specialization. Innovative new functionality will be able to be encapsulated in a self-contained assembly that can be made available to other researchers, who will only have to add the components directly to their own projects. The process will eliminate the lengthy setup and build times associated with the configuration of cumbersome non-homogeneous development environments.

Chapter 2 GPS Operation

The known propagation characteristics of radio waves make their use for obtaining position information as valuable as their use for communication. Radionavigation began with the development of Loran (long-range navigation) system during WWII, and subsequently accelerated with the advent of ground-based short-range line-of-sight navigation aids such as the VHF Omni-directional Radio Range (VOR), the Instrument Landing System (ILS), and the Microwave Landing System (MLS) (3).

2.1 Trilateration

Estimation of position based on distance measurements to reference points at known locations is called **trilateration**. By measuring the length of time taken for a radio signal to propagate from a transmitter to an observer, combined with the known speed of travel of radio waves, it is possible to determine the distance between the transmitter and the observer. A radionavigation system based on this idea is referred to as a **time-of-arrival** (TOA) system (3).

As illustrated in Figure 2-1 [*adapted from* (3)] if one can determine the distances to three radio wave transmitting towers operating at known locations, then one can unambiguously determine their own position.

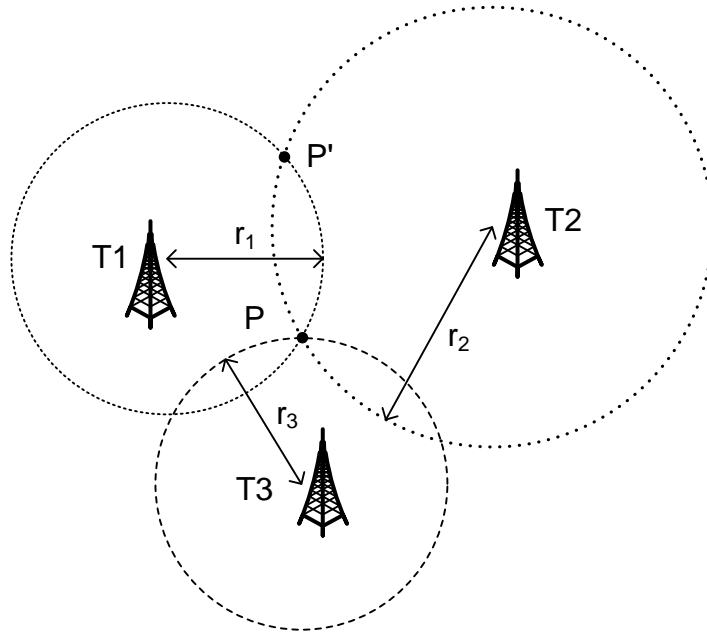


Figure 2-1—Trilateration: Determining the range to three transmitters at known locations permits an observer to calculate their position, point P

By measuring their distance, or range, to transmitter T1 with known coordinates, the observer's **line of position** (LOP) must lie on a circle of radius r_1 that is centered on T1. Range information from a second tower T2 gives a circle of radius r_2 , thereby reducing the uncertainty in the observer's actual location to the two points where the circles intersect, P and P'. While it may be possible to reject one of the points based on other physical information, it typically requires finding the range to a third transmitter to unambiguously determine the observer's position, point P.

To extend the system to 3-dimensional solutions, it is necessary that the angle of elevation between at least one of the transmitters and the observer be large (3). Since ground-based transmitters are limited by practical tower heights, the solutions obtained from them are restricted to 2-dimensions. In the case of a satellite-based transmitter, each range measurement would result in an LOP described by the geometry of a sphere, the

surface of which represents a potential area of position. Three intersecting spheres would identify a coordinate in 3-dimensions at or near the surface of the earth.

GPS is a trilateration-based TOA system. In order for a user of the system to make a position determination (position fix) it is necessary that information regarding the nature and orbit of the satellites that are visible to the receiver (in view) be communicated at the time that the fix is made.

At an approximate orbital radius of 26,560 km the GPS satellites move in space at about 4 km/s, yet their positions can be accurately predicted with an error of less than a few meters 24-48 hours in advance (3). The transmission times are imprinted on the signals using *nearly perfect and nearly perfectly synchronized* atomic clocks on the satellites. The precise estimation of the arrival times at the receiver is made possible by spread-spectrum signaling, which allows each satellite to transmit its unique signal on a shared frequency.

2.2 System Overview

GPS consists of a baseline constellation of at least 24 satellites (they try to keep 30 or more functioning at all times) that operate on the L1:1575.42 MHz, L2:1227.6 MHz, and L5:1176.45 MHz frequencies. All current satellites transmit a public C/A-code (Coarse Acquisition) on L1 and an encrypted P(Y)-code on L1 and L2. Presently, there is only one satellite actively transmitting on L5—full operational capability for the L5 signal is scheduled for 2018 when plans call for 24 L5-capable satellites to be on orbit. The latest generation of satellites, Blocks IIR-M and IIF (“R” is for *replenishment* and “F” is for *follow-on*), additionally transmit a new L2C civil code on L2 (7).

The navigation data transmitted by the satellites are **Binary Phase Shift Keying** (BPSK) encoded at 50 bps and broadcast using a direct-sequence spread-spectrum technique known as Code Division Multiple Access (CDMA) (5).

2.2.1 System Architecture

The components of the GPS architecture are organized into three functional groups or segments: the space segment, the control segment, and the user segment. Each segment serves a specific function or describes a particular operation of the complete system.

The **space segment** is where the satellites live. The baseline GPS constellation is comprised of at least 24 satellites that are distributed in six orbital planes inclined 55° to the equator. Each orbit has four primary satellites and room for several spares. The satellites are identified by a letter (A-F) for the orbital plane and a number (1-6) for the order of the satellite within the plane. With an orbital period of approximately 12-hours, the primary 24 satellites cover the earth in such a way that at least four, and possibly as many as 12, are visible at any time (7).

The **control segment** consists of the **Master Control Station** (MCS) located at Schriever Air Force Base in Colorado, with a backup at Vandenberg Air Force Base in California; several unmanned and remotely operated monitor stations spread around the world; and the ground antenna upload stations that receive telemetry from, and transmit commands to, the satellites. The role of the control segment includes monitoring and maintenance of the satellites' health, orbit, and time (GPS Time), and to predict the satellites' orbital information (the ephemerides) and clock parameters that are required by a receiver in order to make position determinations.

The **user segment** is the part of GPS that most people are familiar with, namely the receivers and devices. It is the responsibility of the receiver to identify and track in-view satellites, to detect and decode the navigation message, and to calculate a navigation solution. The unknown difference, or bias, between the satellite clock and that of the receiver is overcome by an additional parameter in the “fix” calculations, shown in Equation 2-3.

2.2.2 Signals

GPS uses basically two types of spreading codes and operates currently on the L1 and L2 frequency bands. Support for L5, as previously mentioned, is planned but is not fully operational. L1 contains a civilian-use C/A code and a military-only encrypted P(Y) code, while L2 contains only P(Y). There is now also a civilian code on L2—the L2C—however, a portion of the L2 frequency band is shared with Aeronautical Radiolocation services (ground radar) on a co-primary basis and may be susceptible to interference from non GPS sources (3).

Each satellite is assigned a unique ID code that it uses to identify its signal transmission. In order to make a position determination, a receiver must identify the code and then synchronize a local replica of it for at least three satellites (four are necessary to remove receiver timing biases), and track these signals for eighteen to thirty-seconds.

2.2.2.1 Frequencies

Currently, GPS signals are broadcast on two carrier frequencies that are known as Link-1 (L1) and Link-2 (L2). The frequency of L1 is 1575.42 MHz, and L2 is 1227.6

MHz. These signals are derived from a very accurate atomic clock on board the satellite operating at 10.23 MHz, and can be related to them by:

$$L1 = 154 \times 10.23 \text{ MHz} = 1575.42 \text{ MHz}$$

$$L2 = 120 \times 10.23 \text{ MHz} = 1227.6 \text{ MHz}$$

When the clock signal is generated, its frequency is adjusted to be slightly lower by 4.567×10^{-3} Hz in order to account for relativistic effects, making the satellite reference frequency 10.229999995433 MHz on the ground before launch, rather than 10.23 MHz. When received by a receiver, the signals should be at the correct frequencies. However, the relative motion between the satellite and the receiver can produce a Doppler shift in the frequencies by as much as ± 5 kHz (6).

The total radio-frequency power at the transmitter input port on the satellite is approximately 50 W (3), about half of which is allocated to the L1 C/A code. At the receiver antenna, the signal energy is less than the background noise level (≈ -160 dBW). As with all CDMA-based systems, it is the receiver's ability to reproduce the PRN sequence used by the transmitter that allows for the extraction of the signal buried beneath the noise (3).

Since the GPS satellites all transmit on the same carrier frequencies, there is the possibility that they will interfere with one another. In order to help avoid this interference, it is desirable that all signals appear to have the same power level at the receiver. Otherwise, a strong signal may cause a large cross correlation peak with a weak signal, and the receiver will miss the desired cross correlation peak in the weaker signal

(6). If transmitted isotropically, less power will be seen by a receiver from a satellite low on the horizon than from one directly overhead. Consequently, the satellite antennae are designed to concentrate more signal energy on the edges than in the middle of the beam.

The transmitted signals are BPSK modulated with the 50 bps navigation data and PRN chipping codes. The chip rate (frequency) determines the amount of signal spreading that occurs and creates a transmitted power spectral density, which is often modeled in the form of a sinc^2 function $\left(\frac{\sin x}{x}\right)^2$. The main lobes of the spectrum are located at the carrier frequency \pm the chipping rate, making the first null-to-null bandwidth equal to twice the chipping rate.

2.2.2.2 Spreading Codes

The present constellation of GPS satellites supports the coarse/acquisition (C/A) and the precision (P) codes. The C/A code is a 1023-chip code that is BPSK modulated onto L1 at a rate of 1.023 MHz, which makes the first null-to-null bandwidth of the primary signal lobe 2.046 MHz. With a 1.023 MHz chip rate and length of 1023 chips, the C/A code repeats itself every millisecond.

The P code is modulated at 10.23 MHz, making the main lobe null-to-null width of the spectrum 20.46 MHz. The P code is not directly transmitted but is first encrypted by a W code, the details of which are a classified **US military secret**, to generate the Y code. Referred to as the P(Y) code, it is not directly available to civilian users, and requires access to a cryptographic key that the military will only provide to authorized users (3). Civil dual-frequency receivers exist that acquire P(Y) measurements through sub-optimal semi-codeless techniques.

The P code is generated from two PRN sequences. One sequence is 15,345,000 chips in length, and the other is 15,345,037 chips long. These two numbers have no common factors and are, therefore, relative primes (6). The duration, or period, of the first sequence is 1.5 seconds $\left(\frac{15,345,000}{10.23 \times 10^6}\right)$ and the total combined code length of these two sequences is $1.5 \times 15,345,037$, or 23,017,555.5 seconds—just a bit (≈ 9 hours) longer than 38 weeks.

Instead of using one code for 38 weeks, the P code is reset each week so that only a one-week-long portion is used, allowing for 37 different one-week-long codes. Each of the 32 possible satellite IDs is assigned to a different section of the code, with five sections being reserved for operational uses such as ground transmission. The time of the GPS Week must be known very accurately in order to perform signal acquisition; normally, the precise time is found by first acquiring the C/A code and then, using the known timing relationship between the two, locking onto the P(Y) code (6). The signals and contained codes are summarized in Figure 2-2.

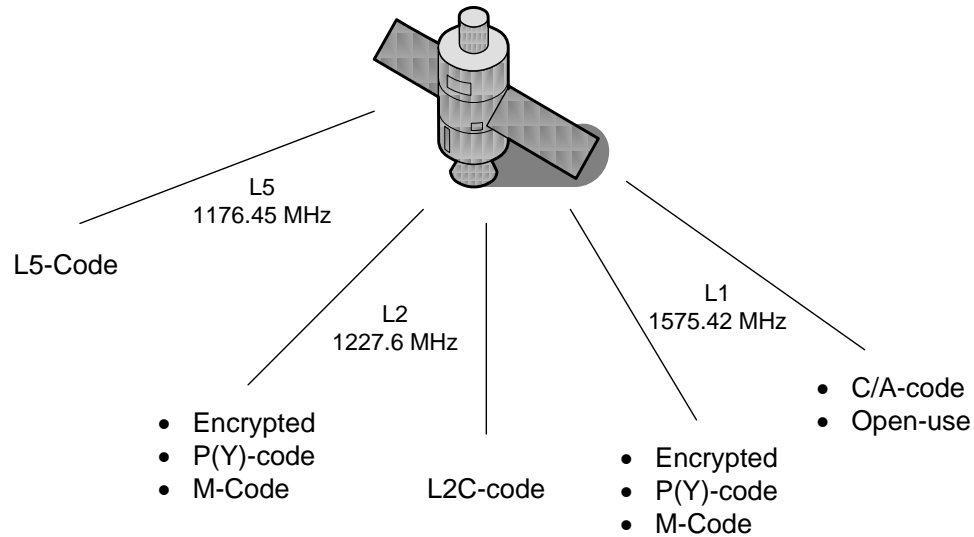


Figure 2-2—GPS Frequencies and Codes: Note that the L5 frequency and the L2C and M codes are only active on some satellites

The C/A codes belong to a family of PRN sequences known as *Gold codes*, named after Dr. Robert Gold, that are formed by the product of two maximal length codes, identified as G1 and G2. Gold codes are an important class of periodic PRN sequences that exhibit good periodic cross correlation and autocorrelation properties.

Each code generator is a 1,023-bit sequence formed by a 10-stage maximum length linear shift register that is driven by a 1.023 MHz clock. Maximum-length sequences (m-sequences) can be created by employing modulo-2 feedback from the shift register output and intermediate stages. The feedback tap positions, which determine the output pattern of the sequence, can be expressed as binary polynomials.

The generator function for G1 can be written as $1 + x^3 + x^{10}$, meaning the feedback is from bits 10 and 3. The corresponding polynomial for G2 is

$$1 + x^2 + x^3 + x^6 + x^8 + x^9 + x^{10}$$

2-1

so, feedback is from bits 2, 3, 6, 8, 9, and 10 (6).

A pair of taps is taken from stages in G2, modulo-2 added together, and then added to the output of G1 to form the C/A code. The positions of the tapped stages in G2 determine the satellite ID. Initially, all stages of both G1 and G2 are reset to the “1s” condition. Figure 2-3 shows the shift register configuration for satellite ID #2 using the G2 taps 3 and 7.

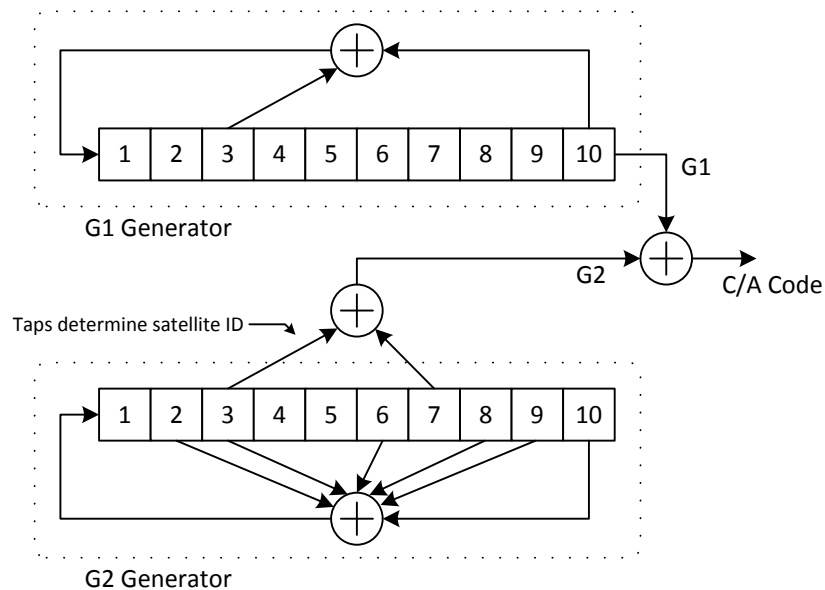


Figure 2-3—C/A Code Generation: The C/A code is formed by the product of two sequences, G1 and G2

Even though it is 1,023-chips long, when compared with the data rate of 50 bps, the C/A code would be considered a short code; the C/A code is repeated 20 times for each data bit. With a 38 week period, obviously the P code would be a long code. Longer codes with higher chipping rates are more desirable than shorter ones from a positioning

point of view since the length of the code limits the precision of the position that can be determined. Each C/A chip is approximately $1\mu\text{s}$ wide, which, when multiplied by the speed of light, corresponds to a distance of 300 m. While the signal travel time can be measured to a small fraction of a chip, however in general, the greater the ambiguity in the measurements of arrival time, the greater the uncertainty in positioning (3).

2.2.3 Transmitted Data

The data transmitted by the GPS satellites includes information regarding the health of the vehicle, its key orbital parameters, and the system time. A reduced precision version of orbital data for the entire satellite constellation, called an almanac, is also included to help receivers predict which satellites should be visible into the near future. While only a short amount of signal, about 1 ms, is required to determine which satellites are currently in view, it is necessary to track and demodulate the signal for up to 12.5 minutes in order to receive the entire navigation message, including the almanac, ionospheric correction data, Universal Time offset, and so on.

2.2.3.1 Navigation Message

As shown in Figure 2-4, the GPS navigation message is transmitted as words that are 30-bits in length. Ten words make up a sub-frame, and a frame is composed of five sub-frames. At 50 bps, each navigation data bit is 20 ms long, so it takes 600 ms to transmit a word, and six seconds to send a sub-frame. It takes 30 seconds to transmit all five sub-frames of a frame, and the entire message is 25 frames long, repeating every 12.5 minutes (6).

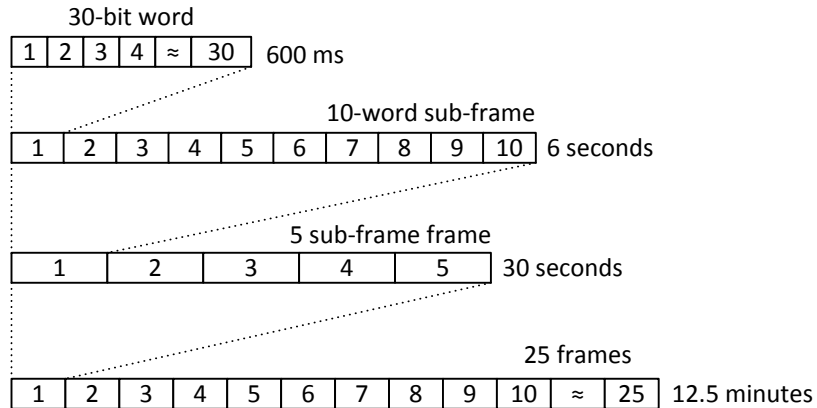


Figure 2-4—GPS Navigation message structure

As previously discussed, essential to the principle of trilateration is the need to know the exact location of the reference transmitters. The precise orbital information for each of the in-view satellites is contained in only the first three sub-frames of each frame of the navigation message, so a minimum of 18 seconds of data is required to accurately determine the satellite’s position. However, since the data is being continuously transmitted by the satellite, it is not possible for the receiver to know exactly when these specific sub-frames will be transmitted. To be certain of receiving all three ephemeris sub-frames, it is necessary to wait until all five sub-frames in the frame have been sent. Consequently, 30 seconds of the navigation message, at a minimum, must be recovered to be guaranteed the delivery of sub-frames one through three.

To help ensure error-free recovery of the transmitted navigation message, the data bits are first encoded using a Hamming (32, 26) error detection code (7), meaning for each 32-bits sent, 26 of them are data. Each word in the navigation message contains 30-bits—24 are data and six are parity bits. In order to perform a parity check, eight parity bits are used by incorporating the last two parity bits from the previous word in the

parity-generation algorithm. There are 30 parity generating equations: one for each data bit, and one for each parity bit. Complete details on the data contained in the navigation message are defined in the GPS Interface Specifications (28), while interpretations and explanations can be found in references (5), (3), (6), and (7).

2.2.3.2 Ephemerides

Each satellite transmits key orbital parameters in sub-frames 1-3 that a receiver uses to correctly determine its position. These parameters, called an *ephemeris*, are predicted by the Master Control Station based on code and carrier phase measurements at the monitor stations. Parameter sets covering the next fourteen days are uploaded to the satellites daily. The data set that a satellite broadcasts changes every two hours, and without daily refreshing would deteriorate over time (3).

2.2.3.3 Almanac

In addition to its own ephemeris data, each satellite transmits as part of its navigation message, in sub-frames four and five, a catalog of a coarse version of the ephemerides of all satellites in the constellation known as an almanac. The almanac allows a receiver to determine approximately when a satellite will come into view above the horizon given a rough estimate of the user position. The almanac parameters are not required to be as accurate as the ephemeris, and serve only to let the receiver plan when to initiate satellite signal acquisition (3). With PC-based prediction software, users can also utilize the almanac data for observation planning purposes.

2.2.4 Position Determination

A receiver determines a user's position by first calculating the distance, or range, to the satellites that are in view based on

$$\rho = (\text{time of travel}) \times (\text{speed of light in a vacuum}) \quad 2-2$$

The time of travel is derived from code and carrier phase measurements; however, as a side effect of the unsynchronized clock in the receiver there is a timing bias between the receiver and the satellite. The offset from GPS Time for each satellite clock is included in sub-frames 1-3 of the navigation message, so the same bias can apply to all signals at the receiver. The measured value of ρ will be either too large or too small by some amount, and is referred to as a pseudorange. As shown in Figure 2-5, by measuring the pseudoranges to at least four satellites ($\rho^{(k)}$ for $k = 1 \dots 4$) the $X, Y,$ and Z coordinates of the position can be determined and the timing bias, b , can be calculated. To do so requires solving a set of four equations for four unknowns in the form of

$$\rho^{(k)} = \sqrt{(x^{(k)} - x)^2 + (y^{(k)} - y)^2 + (z^{(k)} - z)^2} + b \quad 2-3$$

Theoretically, solving this set of nonlinear equations will result in two possible solutions. However, only one solution is near the earth's surface and the other is in space. The iterative methods typically used for solving these equations begin with initial conditions at the center of the earth, guaranteeing convergence on the correct solution rather than the one in space (6).

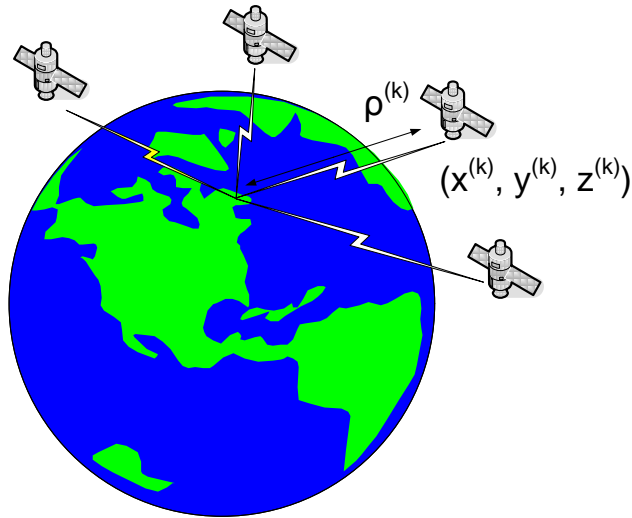


Figure 2-5—Fixing a position requires finding the pseudoranges to at least four satellites

There are many sources of error in making position determinations, and the above model serves only as a most basic of starting points.

2.2.5 Receiver operation

The role of the receiver is to determine the user's position, velocity, and time. It does this by processing and separating the signals transmitted by the satellites, measuring signal transit times and Doppler shifts, and decoding the navigation message to determine the satellites' position, velocity, and time parameters (3).

To acquire a signal, the receiver generates a local replica of a known C/A code and attempts to align it with the signal from a satellite by sliding it in time and computing the cross correlation between the two. When the codes are aligned, a peak will appear in the correlator output. Code tracking is performed through the feedback mechanism of a delay-lock loop (DLL). Adjustments from the DLL keep the code aligned with the incoming signal (2). The aligned code is used to de-spread the signal, leaving only the carrier modulated with the navigation data. The carrier frequency and phase are tracked

using a phase-lock loop (PLL), which essentially extracts the data bits of the navigation message by identifying the phase reversals caused by the data bits. A receiver block diagram is presented in Figure 2-6.

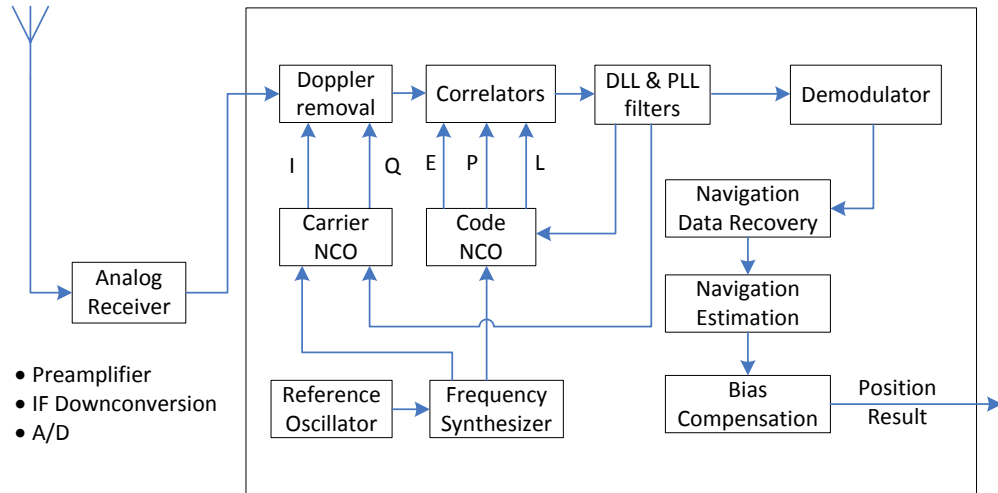


Figure 2-6—GPS receiver block diagram

The time shift required to align the receiver-generated code to that of the satellite is the apparent transit time of the signal, modulo 1 ms. The code chips are generated at known instances according to the satellite clock. The time of reception is determined by the receiver clock and the receiver can determine when the chip was generated essentially by reading the satellite time (3) from the navigation message.

The whole system, including the ranging capability and the navigation message transmission is based on spread-spectrum CDMA data communication.

Chapter 3 Spread-spectrum Fundamentals

Spread-spectrum transmission is a signaling technique that utilizes a specially constructed **pseudo-random sequence** to modulate an information carrier in such a way that the signal energy is spread over a much wider bandwidth than that of the original information-bearing signal. A receiver uses a locally generated and synchronized version of the modulating sequence in order to de-spread the signal and extract the information content (29). The elements of the **pseudo-random-noise** (PRN) sequence, the ones and zeros, are called **chips** in order to differentiate them from the data bits that carry information. The rate at which the spreading code is applied to the signal is referred to as the **chip rate**. This chapter presents general information on spread-spectrum communications that is relevant to its application in GPS transmitters and receivers.

3.1 Spread-Spectrum Types

There are two frequently used flavors of spread-spectrum, namely **frequency hopping**, where the signal is rapidly jumped between different frequencies within the allocated bandwidth; and **direct sequence**, where the digital data is directly combined with a higher frequency coding signal. While these two methods cover most applications, there is also one other: **time hopping**, in which the signal is transmitted in short, pseudo-random bursts and the receiver knows when to look for it. Time hopping is not widely implemented, and will not be discussed any further. Although frequency hopping is not used by GPS, it is described for completeness.

With either system type, due to the increased bandwidth of the spread-spectrum signal, the power spectral density is reduced, so the signal appears as so-called “white” noise to an unsynchronized receiver. This noise-like characteristic makes spread-spectrum systems less likely to suffer negative performance effects in the presence of deliberate or accidental narrow-band noise and interference sources and makes the signal harder to intercept without *a priori* knowledge of the signal structure.

3.1.1 Frequency Hopping

When a pseudo-random sequence is used to shift the carrier frequency of an information-modulated signal such that the transmitted signal occupies many different frequencies, each for a short period of time, this is frequency-hopping spread-spectrum, or FHSS.

As shown in Figure 3-1, the FHSS transmitted bandwidth is determined by the lowest and highest hop frequencies, while the number of hop positions is determined by the system bandwidth in relation to the individual hop bandwidth. At each hop, the FHSS signal is a narrowband transmission with all power concentrated on one channel; averaged over time, the transmitted signal occupies the entire spread-spectrum bandwidth. The hopping pattern is determined by the pseudo-noise sequence and is typically not numerically channel sequential.

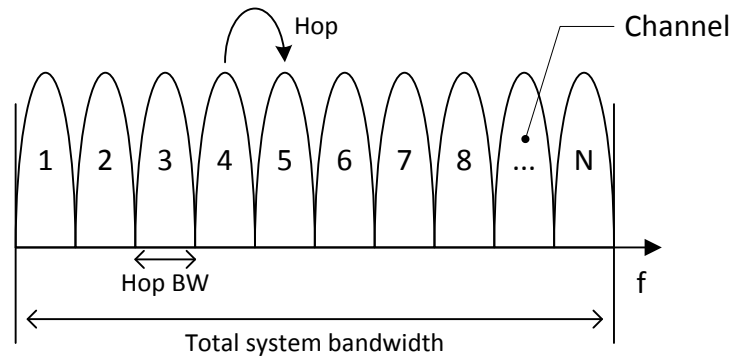


Figure 3-1—Frequency Hopping Spread Spectrum: The total bandwidth available is divided into multiple channels, and each channel is occupied randomly in turn by the modulated carrier signal for a short interval of time

3.1.2 Direct Sequence

With **direct-sequence spread-spectrum**, DSSS, a pseudo-random noise sequence is used to shift the phase of the modulated signal at a rate (chip rate) that is a multiple of the information rate (bit rate), and establishes the degree of signal spreading. The maximum chip rate (R_c) is determined by the design of the system and limits the transmitted bandwidth. Unlike with FHSS, where all of the signal energy is concentrated in a narrow band for a short interval of time, DSSS places all of the signal energy across all of the bandwidth all of the time. To recover the signal, a receiver must be synchronized to the spreading sequence; an unsynchronized receiver will detect only noise.

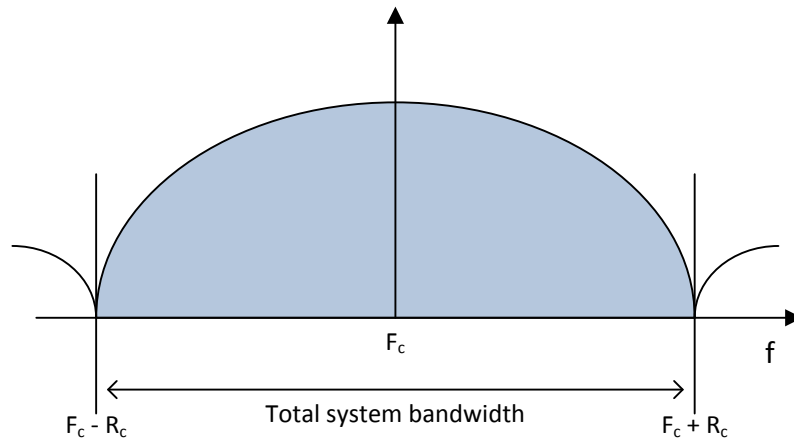


Figure 3-2—Direct Sequence Spread Spectrum: The transmitted carrier frequency determines the position of the center of the spectrum, while the width (spreading) is determined by the chip rate (R_c)

Figure 3-2 gives the general appearance of the frequency spectrum of a DSSS signal that is transmitted at a carrier frequency of F_c . The effect of the chip rate is to spread the signal energy across the entire available bandwidth.

3.2 Transmitter and Receiver Architecture

Since GPS uses the direct-sequence form of spread-spectrum, the following discussion on transmitters and receivers will highlight the basic principles of their operation in DSSS applications, and emphasizes the specifics of the **Binary-Phase-Shift-Keying** (BPSK) PRN chip modulation employed by the GPS satellite transmitters.

3.2.1 Modulation

The stages of a DSSS modulator are represented schematically in Figure 3-3 (4). The **nonreturn-to-zero** (NRZ) binary data [actually, a binary 0 maps to a +1 and a binary 1 to a -1] are **phase-shift modulated** onto a radio-frequency (RF) carrier. This operation can be represented by a multiplication function that shifts the carrier phase by either 0 or $-\pi$

radians. The resulting product is then multiplied by the pseudo-random noise, $PN(t)$, pattern (also -1, +1) such that whenever the product of $d(t)$ and $PN(t)$ is -1, the carrier is phase shifted by $-\pi$ radians.

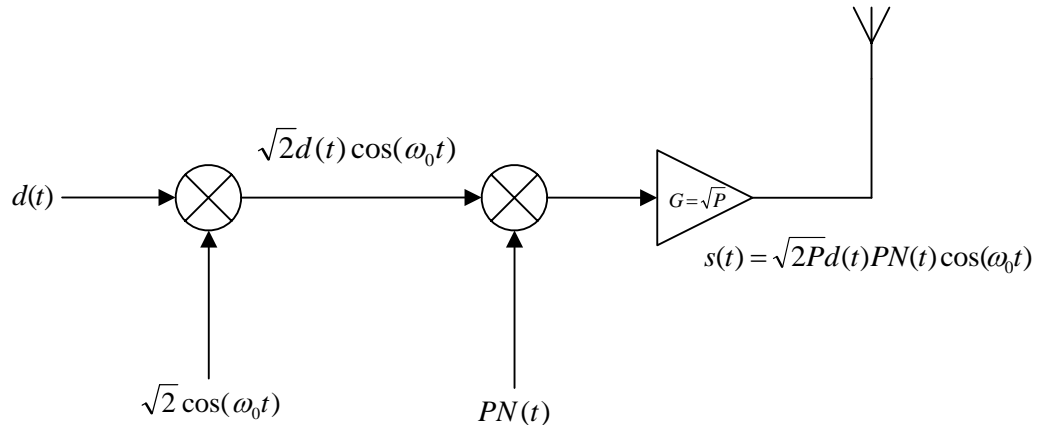


Figure 3-3—BPSK DSSS Transmitter: The input data, $d(t)$, is combined with the carrier and then binary phase shift keyed with the pseudo-noise sequence

3.2.2 Demodulation

Once modulated and transmitted, the signal cannot be detected by a conventional narrow-band receiver. In order to detect and demodulate the data contained in the signal it is necessary to remove not only the carrier, but also the spreading code. Often referred to as de-spreading, the signal detection and demodulation process requires that locally generated versions of the carrier and PRN sequence be kept in synchronization with the received signal. Carrier and code removal are codependent activities. In the case of a Doppler-shifted signal, a close estimate of the carrier frequency is needed in order to identify the presence and offset of a particular spreading code, and an aligned code-removed version of the signal is necessary in order to accurately determine the signal frequency. Current methods for acquiring a signal are different from tracking one. Some

receiver references, such as (3) and (29), discuss the first problem of removing the RF carrier and then generating a synchronized PRN sequence. Others, for example (4) and (6), consider the problem of removing the spreading PRN code first and then the carrier. Either way, it is essential to remove the RF carrier component first, and then determine the required code and phase synchronization necessary to fully demodulate the data. This approach is modeled in Figure 3-4 where the signal is first amplified by a low-noise amplifier (LNA), mixed with a locally-generated RF carrier, and then band-pass filtered (BPF).

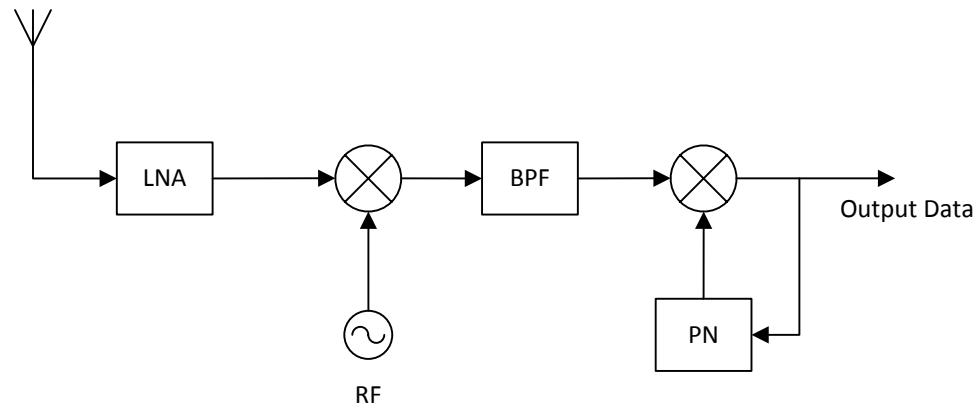


Figure 3-4—Direct Sequence BPSK receiver model

3.3 PRN Sequences and Generators

Essential to spread-spectrum communication implementations is the ability to generate a deterministic pattern, or sequence, of binary 1s and 0s that exhibits random noise-like properties that can be used to spread the bandwidth of the signal energy. As the pseudo-random epithet implies, these sequences are predictable when the generating code is known, but appear random to an observer when the code is unknown. Without the

ability to predict the sequence, it would be impossible for the receiver to generate a local replica to use to demodulate the data.

Besides a noise-like appearance, other characteristics of an ideal spreading code are a zero cross correlation with other codes, maximum (peak) autocorrelation for zero delay, and zero autocorrelation for all non-zero delays. Additionally, ideal odd-length codes will exhibit a balance property in that the number of 1s exceeds the number of 0s by one, and that the probability distribution of consecutive patterns of all 1s or all 0s (run-lengths) behaves in a $\frac{1}{2^R}$ manner (29), where R is the length of a run.

The cross correlation function can be used to find and track code synchronization. A distinct peak will appear in a correlator output when the local version of the code is time aligned with the received code sequence. With an odd-length balanced code, there will be one more 1 in the sequence than zeros, such that the normalized result of the correlation will be 1.

The length of a PRN code, long or short, is considered in terms of the associated data rate. A short code has the same pattern or portion of the PRN sequence repeating for each data symbol, whereas a long code is much longer than a data symbol (29).

Most practical codes utilize **maximum-length sequences** (*m-sequences*) that are derived from the outputs of linear feedback sequential shift registers implementing functions determined by the properties of irreducible primitive binary polynomials. The employed algorithms are described and supported by extensive theoretical work in ring and finite-field arithmetic theory, some of which can be found in references (4), (29), and (30), with highlights provided in Appendix C.

Chapter 4 Object-Oriented Analysis and Design

Much of the flexibility and extensibility of the Receiver Development Framework is derived from leveraging object-oriented design techniques and the languages used to implement them. In order to provide a basic appreciation for these concepts, the following sections will briefly discuss the underpinnings of object-oriented analysis and design and will provide working definitions for commonly used keywords and phrases.

Unlike procedural-based structured programming techniques that emphasize functions over data, where the focus is on black-box representations of processes, an object-oriented approach is to view the problem as a set of items and related behaviors, objects and methods, that are essential to the system being analyzed.

In the C language, the basic unit of work is the function. In an object-oriented solution it is the class method. In a parking-lot application, for example, the basic elements might be the *Lot*, the *Space*, and *Vehicle* classes. Each class within the application has methods that perform transformations on the internal state information or other object data, and that are used to send and receive messages from outside objects.

Encapsulation, Inheritance, and Polymorphism are the three defining traits of object-oriented solutions.

4.1 Encapsulation

A class is a design description for the creation of an object, while an object is an instance of a class that has been constructed from the description. Classes describe the

basic components, their operations, and characteristic features of the system. A class name declaration represents the boundary of encapsulation for critical internal data stores and the available methods that can directly modify or manipulate their values.

Encapsulation reduces potential coupling between unrelated entities, minimizes unintended side effects, and helps to ensure integrity of the object.

The class description may include the declaration of member variables that hold the state of the object. Two instances of the same class may have different states, depending on the values of their member variables. Members can be declared as being private, internally accessible only by other class members, or public, accessible to external users of the class instance. Private members that are accessible only through a public get/set accessor method are called properties. For example, a class of type parking *Space* may have a private *MaxWidth* member that is accessed by calling *get_MaxWidth* and *set_MaxWidth* property methods.

Objects can also encapsulate instances of other classes, either of the same or different type. In the *Parking* application example, a class of type *Lot* may hold references to many *Spaces*, and each *Space* could contain a *Vehicle* that was placed there by calling the *Vehicle::Park* method. Object references may take the form of C pointers or arrays, but may also be implemented in a language-specific manner through specialized containers or mail slots that have been likewise implemented using object-oriented techniques. Combining classes in such a manner results in what is called a ***has a*** object relationship: a *Space has a Vehicle*, or a *Lot has a* (or many) *Spaces*.

The process of identifying the properties, attributes, behaviors, and relationships of the set of objects essential to an application is referred to as object modeling. There are many methodologies and diagrammatic representations for communicating an object model. The Unified Modeling Language (UML) evolved from the collaboration of early pioneers in the field of object modeling. UML is a complex and mammoth toolset for modeling the structural, behavioral, and activity characteristics of applications and their objects. Mastering the intricacies and vagaries of UML, not to mention its many inconsistencies, is a significant challenge for those interested in doing so. The basics of UML for creating representations of object models is becoming the common vernacular in the domain of object oriented analysis and design. As such, only a limited subset of the UML nomenclature will be utilized in the presentation of the receiver framework.

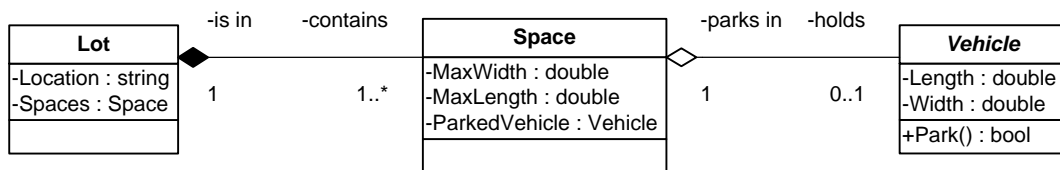


Figure 4-1—Static UML object model for a parking application

Figure 4-1 represents a UML class diagram that shows the static structural relations between classes in the hypothetical parking application. From left to right the diagram reads, a *Lot* contains one or more *Spaces* and each space holds zero or one *Vehicle*. When read from right to left, a *Vehicle* parks in one space, and a *Space* is in one *Lot*. The solid black diamond next to the *Lot* class indicates UML **composition** and a life-cycle dependency between the *Lot* object and the *Spaces* that it contains. If the *Lot* were destroyed, the *Spaces* would also cease to exist—without the *Lot*, the *Space* has no

meaning. The same dependency does not exist between the *Space* and *Vehicle* relationship; the *Space* can be destroyed, but a *Vehicle* can continue to exist independently of the space it was once parked in. In the UML diagram symbology, an unfilled diamond is used to differentiate an **aggregation** relationship from a stronger **composition** relationship; however, they are both used to indicate the presence of whole-part structures between classes.

Strict enforcement of encapsulation inevitably leads to a problem of **state persistence**. Suppose the *Vehicle* class has a private data member, *IsParked*, that retains the current parked status of the vehicle instance—true/false or yes/no. Once the *IsParked* internal variable is set upon completion of the *Parked* method, other class members (not shown) can make use of this state to adjust their internal actions where appropriate. However, the value of *IsParked* will be retained only if the object is kept active in system memory. Once the application ends, or if the object needs to be relocated or transmitted over a network, there is no guarantee that the proper value will be restored when the application is restarted and its objects are reloaded. State persistence is a multi-dimensional problem in that there are many causes, many solutions, and many data integrity issues to address along with various timing and performance considerations.

It is also necessary to point out that multiple valid object models can be derived for a given problem domain, depending on the viewpoint and expectations of the solution. The formal demonstration of completeness and correctness of an object model is both difficult and often unnecessary due to the flexible nature of an object-oriented implementation approach. Once the top-level objects and their relations have been established, sorting out the microscopic details of the various interaction aspects is more easily approached in an

iterative manner, with each iteration attempting to resolve issues that are identified by the previous version of the model.

4.2 Inheritance

Once the required functionality has been implemented in an encapsulated class structure, the manner in which the behavior is altered or extended is through object inheritance. Inheritance is a mechanism whereby one class description is used as a starting point for another, necessarily related class. Members and functional implementations from the original class, called the *base class* or *parent class*, are available to the new class, called the *derived class* or *child class*. Inheritance allows the base-class methods to be extended and enhanced for specialization in derived classes, without violating the rules of encapsulation of the base class.

The UML class diagram shown in Figure 4-2 represents a possible inheritance-based extension to the *Vehicle* class for the parking application. The *Vehicle* base is used to derive implementations of a specialized family of classes, including a *Car*, a *Bus*, and a *Truck*.

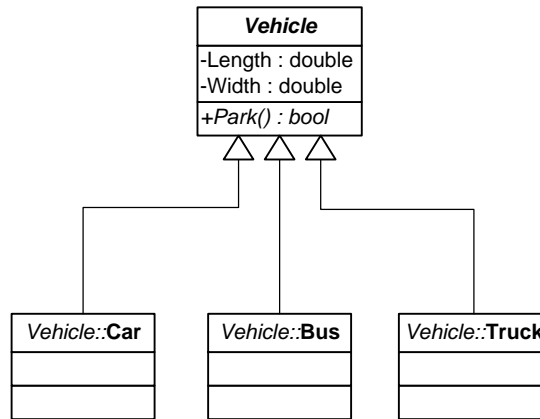


Figure 4-2—UML object-model showing inheritance

The diagram represents an object-oriented hierarchical taxonomy and provides little insight into the behavioral or timing relationships between the various classes. Only the semantics that a *Car* is a type of *Vehicle*, for example, are represented by this static model view. The behavior of the *Park* method in each derived *Vehicle* type is inherited from the base *Vehicle* class.

Inheritance from classes can be carried out at any level of the object hierarchy. A specific *Car* type may be derived by further inheriting from the *Vehicle::Car* class, such as a *Vehicle::Car::Sedan*. Subtypes of *Sedan* can be viewed as *Cars* or *Vehicles*, depending on the application context, and all object instances can be parked in a *Space*.

4.3 Polymorphism

Polymorphism allows an object to change its behavior at runtime depending on the type of object that has been constructed. When a base class provides a declaration of a method that is marked as *virtual*, derived classes may alter the default functionality of the base by **overriding** the method and providing a new method definition in its place.

However, users of the base class need not know any of the type-specific implementation details of the derived class.

The inheritance diagram for the *Vehicle* class shown in Figure 4-3, for example, indicates that the *Vehicle* base class supports a virtual *Park* method that has been redefined in the *Car*, *Bus*, and *Truck* classes. Each of these classes has provided a specialized version of the *Park* method that meets the individual need of the class type.

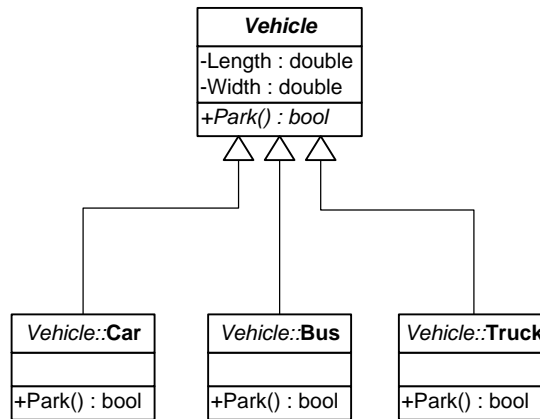


Figure 4-3—Each class derived from vehicle implements a specialized polymorphic Park method

Polymorphism occurs when an instance of a *Space* class, containing only the information provided by a *Vehicle* reference, calls the *Vehicle::Park* method, which automatically invokes the *Car::Park* implementation. The mechanism by which the method call is resolved to the correct implementation of the derived class is through **late binding**. While early binding seeks to resolve methods and their definitions when the application is compiled and linked, late binding provides only a loose mapping in the form of a **virtual table** between a method invocation and the code that implements its behavior such that the call can only be resolved when the application is executed at run time.

The significant benefit of polymorphism is realized when an entirely new *Vehicle* type is added to the design, after the application has been completely written and tested. A *Motorcycle* may be derived from the *Vehicle* base type that overrides and implements the *Park* method, and any existing code that calls a *Vehicle::Park* method will require no changes in order to support the new type of vehicle. All of the necessary specialization code is self-contained in the new type implementation. By extending the application through inheritance and polymorphism, any errors or issues can be clearly isolated to problems with the new components, simplifying the debugging efforts. If the application worked correctly before the new type was introduced, broken functionality simply becomes a case of *post hoc, ergo proptor hoc*.

Virtual methods that are declared in a base class but contain no implementation are considered to be *abstract methods*. Class specifications that contain only abstract methods are referred to as *pure abstract* or *pure virtual* classes. Instances of these classes cannot be created directly, but must first be derived from, and all methods implemented, before an object can be instantiated. Classes that are derived from abstract classes, and provide complete method bodies, are called *concrete classes*.

Pure virtual class declarations can be used to describe a component **interface** or signature. Calling methods on concrete classes through the abstract class type represents a formal declaration of the kinds of methods a family of classes must support. In terms of UML, an interface is described by the diagram of Figure 4-4.

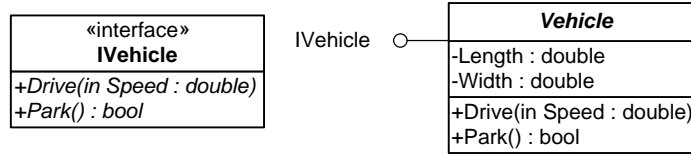


Figure 4-4—An interface declaration and a class that implements it

On the left, the *IVehicle* interface (the de facto standard for interface names is that they start with an uppercase *I*) declares the methods and their parameters that implementers of the interface must support. On the right, the *Vehicle* class indicates that it implements the *IVehicle* interface through the named **lollipop** symbol attached to the edge of the class diagram.

Like classes, interfaces can be composed and extended through inheritance. Classes may implement multiple interfaces, according to the needs of the application.

4.4 Special Items

Sometimes it is required that object instances of one class type be informed of situations or conditions that occur in another type. While it would be possible to have an object maintain references to all of the external recipients in a list or a collection, doing so requires a level of runtime determinism that is usually not possible. Instead, it is more desirable to have the sources of events, or **publishers**, be bound to event sinks, the **subscribers**, through some nondeterministic means that ensures loose coupling between the two parties, allowing subscribers to select which notifications they would like to receive from individual publishers. The scheme by which the required linkage is established is through *loose coupled events*.

In an event-driven application, subscriber objects register with the publisher the class method that should be invoked, or called, by the publisher when the event occurs. To ensure type safety and compatibility between the event and the method that is called, the publishing class must specify what the exact method signature should look like by defining a **delegate** type to be used during the registration process. Only if the method indicated by the subscriber matches the delegate description will the event registration succeed. Subscribers remove themselves from publisher's events in a similar way, placing the control of the notification mechanism within the realm of the subscriber.

Events and delegates work in a fashion that is not unlike hardware-based interrupts. In fact, many of the sources of software-based events involve external hardware actions, such as mouse button clicks, keyboard key presses, and other such activities. Events can signal when a process has started, when it has finished, or when a condition has been satisfied, such as a data value meeting a predefined criterion.

Chapter 5 Real-time Systems

There are special requirements that one must take into account when designing and developing applications for real-time operations. Potentially an issue at a fundamental level is the lack of consensus among various implementation communities on the essential distinguishing characteristics of a real-time system and the traits that make them special from their alternatives.

Real-time system implementations often take on a similar architectural form as many multithreaded applications, and likewise encounter the same analysis complexity and operational problems. Discussions related to non-real-time applications also apply to real-time systems, as well, but need to include additional time constraints imposed by considering the real-time schedule constraints.

Good system design practices dictate applications of modular construction consisting of multiple role-specific processes and interacting threads. Scheduling execution times for the threads and synchronizing the required level of interaction to support the appropriate types of interprocess communication create additional analysis and design complications. The diversity of the architectural patterns and choices for real-time systems necessitates a formal analysis methodology to sort through. Detailed design analysis requires the support of capable modeling tools, and well-defined models should provide a clear view to an implementation plan. The evaluation of suitable software languages for real-time systems represents another set of factors that must be considered prior to the start of development.

This chapter provides an informal working definition of real-time systems and attempts to place these solutions in a context that is suitable for the subsequent discussions on application structure, scheduling, synchronization, various modeling options, and the identification of essential real-time language features.

5.1 Definition of Real-time

While superficially trivial, possibly one of the more awkward difficulties in discussions on real-time systems lies in the nuances of establishing a broad-ranged widely-applicable definition of *real-time* as compared to *non-real-time*. Definitions derived from distinctions based on application-dependent operational requirements or consequences of failure (critical vs. noncritical) are necessarily subjective and largely a matter of opinion. As such, the same system implemented for one environment as real-time may be classified as non-real-time when applied to another. The opposite may also be true, depending on how the system requirements are formulated. Relaxing a performance constraint can allow a system previously considered non-real-time to operate in a real-time environment without making any changes to the underlying system design characteristics.

The evolving body of literature on real-time systems has partially managed to avoid directly addressing the issue of the meaning intended by the term's use by creating the fuzzy conceptual modifiers of *hard*, *soft*, and *near* real-time. These modifiers provide the individual reader with the latitude to infer a definition that is almost a matter of personal taste. As a result, the published work must be evaluated against a subjective continuum that ranges from *not-real-time-enough* to *more-real-time-than-necessary*. This *Goldilocks phenomenon* makes applicability of the conclusions and consequences either too

restrictive (narrow focus) or too broad (far reaching), with only a small subset of the audience feeling just right.

Conceptually, the notions of *hard*, with strict timing constraints and no permissible violations, or *soft*, involving perhaps some level of transaction based user interaction, real-time systems tend to define more of the execution model of the operations and are not confined to a specific realm of operating environment. If a timing constraint may be violated without affecting the validity of the produced result, then it could be argued that the constraint is poorly defined or fundamentally over specified.

Real-time system definitions that are performance based or described through some type of behavioral determinism, such as the system must be fast and responsive, suffer the recursive problem of subsequently trying to apply concrete meanings to other possibly even more abstract ideas. From a pragmatic point of view, systems that are slow and unresponsive are rarely deemed engineering successes. An expectation exists with users and designers alike that all systems should be fast and responsive. The terms *fast* and *responsive*, however, are relative measures that require an absolute baseline metric in order to unambiguously determine a frame of reference for performance comparison purposes.

If performance becomes the essential property in the determination of real-time, then the importance of competing factors, like flexibility, maintainability, and scalability become deprecated and the total quality of the system design will ultimately suffer. Regardless of requirements, designers normally strive for the best overall performance, as measured by the number of generated outputs in the fewest work cycles possible. There

are many high-performance, high-workload applications possessing great computational complexity that would not generally be thought of as real time. Conceivably, real-time systems with weak or slack timing constraints may need little performance, but must still meet the imposed timing requirements. So, while sometimes key, performance is neither necessary or sufficient in the classification of real-time systems.

Real-time distinctions based on the embedded versus non-embedded (or workstation) nature of an application contain implementation requirements that are more reflective of the resource limitations often encountered in working with dedicated special-purpose hardware. Design challenges resulting from limited memory, single low-speed processor, and minimal operating system support may reduce the richness of features that are implementable in an embedded solution, but care should be taken not to confuse the verifiable application requirements with the restrictions imposed by the underlying environment. If the removal of the embedded components of a real-time system design eliminates the driving requirements for a real-time solution, then the view must be taken that the root system is not fundamentally real-time in nature. Furthermore, distributed or networked systems that are composed of multiple interconnected nodes, which may or may not be implemented as embedded devices, could still be required to operate in real time. The emergent properties of such systems makes it necessary to take a holistic view of the complete operating characteristics before making a real-time behavior determination. The diversity of real-time systems covers a wide range of technology scale from small, dedicated hardware devices to large, complex distributed applications.

Inevitably, one must conclude that *real time* is a multi-dimensional *N-space* problem domain, with each dimension comprised of a solution subspace of varying boundaries

encompassing a region of some volume. Each design challenge, such as scheduling, thread management, synchronization, and resource allocation has a field of suitable potential implementations available. An optimum system attempts to minimize, in some quantitatively measurable sense, a representative cost function while simultaneously maximizing the value of a critical figure of merit.

System designers generally have an intuitive, although informal, understanding in-the-large of what is and is not real time. These systems are often driven by a precisely timed external stimulus or event and operate on the expectation regarding the ability of the system to produce a result or outcome before the next event arrival. Behavioral predictability is critical for design acceptance, and as a result the extensive evaluation of processor performance and utilization warrant considerable review and characterization. In contrast, coarse-grained batch-oriented computation models, where input arrival processing can be suspended indefinitely without compromising the validity of the resulting outputs, are usually thought of as non real time.

In practice, solution analysis and design patterns that satisfy the majority of real-time operational needs typically yield additional beneficial side-effects in the areas of extensibility, flexibility, and maintainability. Even in the absence of specific real-time requirements, adopting a real-time system architecture can enhance the overall quality and long-term value of a solution. As a result, one should attempt to take the broadest conceivable definition of a real-time system early on in the design life cycle, without creating unwarranted functional restrictions or unnecessary implementation complications. Every computing system can be regarded as having some characteristic

that is real time in nature, regardless of the ambiguous or subjective properties of real-time system definitions.

5.2 Applications, Processes, and Threads

System modularity represents a design outcome in a divide-and-conquer approach to solution analysis. By carving a set of problem specifications into groups of related functional pieces, highly complex problems may be decomposed into simpler representations of smaller interdependent components. Each smaller component may, if necessary, be further reduced into tinier subcomponents until the details of implementation become apparent. The final solution is then composed through aggregations of the smaller pieces and by defining the interface boundaries where the various modules interact with one another.

Real-time systems can be represented or described through modular containers of functionality with differing levels of granularity. Once designed, the resulting system modules are then implemented as one or more applications, processes, and threads.

An *application* is a top-level system comprised of one or more substituent elements. Applications may be composed of multiple dedicated *processes*, and each process may have multiple *threads*. A *process* can be thought of as a task in that it represents a subunit of work that is smaller than an application. Due to the complexity and high clock speed of modern microprocessors, distinctions between processes and tasks based on the number of physical processors in the system, like those found in (31), are synthetic and outdated. Processes within an application can usually be configured to operate with unique identities for the purposes of resource access and security.

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.

“A thread is a lightweight process with a reduced state.” (31)

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 multithreaded runtime environment. If 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 ready-to-run thread is loaded and the processor resumes execution.

By managing multiple threads in this 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 utilized to execute an application, where each available processor manages a different set of threads for execution. The scheduling mechanisms responsible for setting and controlling the sequence of running threads are considered later in this document in Section 5.3.

Articles such as (32) suggest the abandonment of threads as a model of computation in favor of weakly defined alternatives. The view is promoted that multithreading represents a nondeterministic approach to problem solving and should be used only when chaos and randomness are desirable system characteristics. In this view, even for a small number of programming instructions, the permutations of their arrangement is large, but only one sequence represents the correct solution to the problem. As a result, the odds of finding the right solution are small. Adding hard-to-model thread interaction serves only to create another layer of complexity that reduces the odds of success still further. However, the argument as presented is flawed in that, due to the commutability and idempotence properties of many of the required operations, there would exist many more than one solution to the problem (*e.g. $A \times B \equiv B \times A$ —two solutions*) and that even a small amount of skill and intellect can greatly increase the possibility of finding one of the correct instruction sequences. While it is unlikely that typing random letters on a keyboard will result in a well-formed grammatically correct sentence, there is no need to consider abandoning words as a method of communication due to the complexity of a written language.

However complicated thread interaction modeling and prediction may be, one must accept that the idea of a thread is a consequence of the key features of the von Neumann computing architecture (33):

- Data and instructions are stored in a single read/write memory;
- Memory contents are addressable by location;
- Execution occurs in a sequential fashion.

A thread is a synthetic device that has been fabricated 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.

For security purposes, threads will inherit the identity of their creating process when accessing shared system resources. The specific authentication and access control mechanisms are dependent on the nature and level of operating system access control support.

5.3 Scheduling

Scheduling is the act of determining the order in which the threads (or tasks) within a process are made running on the CPU. Schedulers may be grouped into one of two broad categories: *static* or *dynamic*. A static scheduler is usually a person who makes up the thread list ahead of time and figures out their ideal execution sequence in order to guarantee that all threads finish their work in time and that there are no constraint violations or resource deadlocks. The task list is then hard-coded into the work queue of the implemented system so that the threads always run in the same predetermined order.

For a static schedule to function as expected, reliable information regarding the resource requirements and processing time for each thread needs to be available during the early stages of system development. 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 verification of the achievability of the schedule often requires the use of NP complete (nondeterministic polynomial time) schedule analysis algorithms, while the schedule optimization process is typically NP hard.

Deterministic analysis techniques that are based mainly on worst-case execution times and task deadlines do not allow deadline violations in data processing. These scheduling techniques are inappropriate for use with highly variable workloads. However, the majority of workload types are inherently variable and the worst-case execution time is often significantly different from the average-case when average processor utilization is less than 100% (34).

The idea behind the work of (34) is to improve the results of deterministic analysis by providing a means of task characterization that more accurately represents the degree of workload variability. Using a grouped probability distribution method, *histogram-based*

models are developed that describe tasks as a discrete statistical probability mass. The developed models can then be used to aid in the scheduling process. Choosing the correct number of task classes that balances result precision with the analysis complexity is critical to the success of the approach. The paper shows that *workload isolation* is a desirable property of scheduling algorithms that simplifies analysis and makes the outcome algorithm independent.

The *Spring Architecture* (35) stresses predictability and flexibility in real-time system design by defining three types of tasks: *critical*, *essential*, and *nonessential*. Each task type is handled differently using an *a priori* scheduling method to ensure critical tasks will always meet their deadlines. 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 make 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 (36):

- An in-order front-end that fetches instruction streams from memory. Four instruction decoders handle up to five instructions per cycle. Decoded instructions (called μ ops) are fed to an out-of order execution unit four at a time.
- An out-of-order execution engine that issues up to six μ ops per clock cycle. The μ ops are reordered to execute as soon as operand sources are ready and execution resources are available.
- An in-order instruction retirement unit that ensures μ op execution results are processed and completed in a sequence consistent with the original program order. A peak instruction retirement rate of up to four μ ops 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 μ ops 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.” (36)

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.

Obtaining a deterministic result from a statistical approach is a difficult thing for static scheduling to successfully achieve. Additionally, static execution schedules are not very flexible in that minor changes to system requirements can mean huge new efforts to rework the execution plan. With small, relatively simple systems with little to no OS support, however, static scheduling can produce acceptable results.

In a dynamic scheduling environment, tasks are allowed to run according to the availability of system resources and free processor clock cycles. Dynamic scheduling may be either *preemptive*, where tasks are periodically interrupted so the system may check if something else needs to run, or *non-preemptive*. Both preemptive and non-preemptive systems require high degrees of cooperation between interacting threads, but the amount of cooperation required is typically greater for the non-preemptive case. The cooperation comes in the form of threads that appropriately signal their resource usage, wait for other threads to release shared resources before proceeding, don't block unnecessarily or longer than required, and release control of the processor from time to time during long-running loops of operations. Obviously, this type of cooperation must be preplanned and developed into the system at implementation time.

Preemptive dynamic schedulers have an additional requirement of needing a source of periodic interrupts, usually in the form of a hardware interval timer. The interrupt service routine for the timer event will determine if another ready-to-run thread is waiting and should be made running. If so, a context switch is performed by saving the current thread's state and loading the previously saved state from the next thread to run into the processor. The interval of the timer's timeout determines the time resolution of the thread schedule. If the interval is too large, threads will potentially run for an excessive length of

time, while other threads will miss critical completion deadlines. If the interval is too small, threads will be interrupted more frequently than necessary and the increase in overhead will limit their ability to make execution progress.

Threads may be managed according to their importance so that higher priority threads receive greater access to the system processor. When all threads share the same fixed priority, only a single work queue is required and threads can be serviced in a round-robin fashion. With multiple thread priorities, or priority classes, a separate queue is required for each shared priority level and waiting threads of the highest priority are dispatched first. If priorities are not shared, each thread must have a unique priority, and one work queue maintained in order sorted by priority is sufficient.

Priority-based scheduling creates an implementation challenge with many solutions and algorithms that cover the gradient from simple and fast to complex and not so fast. Many of the more complex approaches aim at detecting and avoiding threads that deadlock over access to shared critical sections. A deadlock occurs when one thread acquires a resource needed by another thread, and the other thread is blocking the processor while it waits to acquire the resource it needs. The result is that neither thread can make progress. A related problem occurs when higher priority threads are blocked while waiting for lower priority threads to release resources, which leads to a situation known as *priority inversion*. Static priorities can also lead to thread starvation, where lower-priority threads never gain access to the processor due to the continued preemption by higher-priority tasks. One solution to these problems lies in the introduction of some form of priority inheritance, where a thread's running priority is dynamically varied from its base priority. Wait time, acquired resource priorities, and the priorities of other

waiting threads can be used as factors in calculating the current thread's actual running priority.

Preemptive operating systems providing guaranteed response times may use *Rate Monotonic Scheduling* (RMS) with static priorities. Threads are assigned priorities based on the execution time of their work: the shorter the time, the higher the priority. Rate monotonic analysis is used in the development of these systems to provide scheduling guarantees for a specific application. Assumptions for a simple RMS implementation include no resource sharing (so, deadlocks shouldn't happen) and free context switches. One must carefully evaluate whether these assumptions are legitimate for their particular application.

In systems requiring mutual access to shared resources, arbitrary preemption of tasks introduces the need for non-trivial resource access protocols, which can degrade system performance and complicate system analysis and design (37). However, a fixed-priority non-preemptive solution may lead to scheduling conflicts. An analysis of the worst-case response times under fixed priority with deferred preemption scheduling (FPDS) and a continuous time model is presented in (37). The central thesis of the work is to show that previous analyses were fundamentally either too pessimistic or overly optimistic.

The benefit of dynamic scheduling over static is a reduction in the level of information required on the behavior of the processes involved. Dynamic plans can also be more flexible in accommodating changes and insertion of new jobs. However, it can be more problematic to guarantee critical deadlines under a dynamic scheduling plan, which usually results in contingency increases during design to ensure sufficient resource

allocations. Dynamic scheduling also increases the overhead work required in preemptive systems by periodically running a schedule and dispatch cycle. Increasing the sophistication of the scheduler algorithm directly increases the corresponding amount of overhead.

A method for the insertion of a random task within a predefined schedule of jobs, such that existing real-time constraints are not violated, is presented in (38). The jobs (or tasks) are assumed to be non-preemptive and must execute in the order given by the schedule, although their start time can be delayed. The objective is to determine a suitable point of insertion in the schedule for the new job without compromising critical deadlines. A reference algorithm, without the real-time constraint, is provided and subsequently shown to be $O(n^2)$ in complexity. The algorithm is then divided into an *offline* part, where the schedule is determined and pre-calculated, and an *online* part, for when a job must be inserted. With the applied modifications, both the online and offline parts are shown to be $O(n)$, which would make the method a reasonable solution for accommodating a mix of static and dynamic scheduling, even for a suitably large number of tasks. The work has the potential of bringing together the best of both static and dynamic scheduling plans.

A further complication in the areas of real-time system analysis and design is the variable frequency clocks for thermal management available in low-power devices (39). Predicting system behavior while simultaneously balancing temperature-imposed clock-speed limitations results in conflicting operational characteristic requirements that are not addressed by existing design methodologies. The work of (39) is an effort to resolve this problem through a calculus of reactive speed scaling. Although pointing out traditional

worst-case execution scenarios do not apply in temperature-constrained situations, the paper does not address schedulability analysis under the resulting clock-speed constraints.

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 multiprocessor 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, (36) 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, discussed next, can reduce the effective level of parallelism and diminish the potential performance gain through processor scaling.

5.4 Synchronization

With a hardware-based design, getting multiple 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 harder to achieve; it is very difficult to run multiple code modules simultaneously with a high

degree of timing precision. In single-processor multithreaded 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 temporal behaviors and interactions are critical and essential 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.

Consider the situation where two threads, *thread-A* and *thread-B*, execute the C instruction $n++$, meaning take the current value of n , add one to it, and store the result back in the memory location identified by the variable n . Once thread-A reads the current value of n , it is then pre-empted by thread-B, which also reads the current value of n . When thread-A resumes execution, it increments its local copy of n and saves the new value to the memory location of n . Thread-B then does likewise, but since it is working from an outdated copy of n it is overwriting the work of thread-A, and the resulting value of n is now one too small. Thread-B should have been kept from accessing the value of n from the point where thread-A initially read the value through to the time when the update was completed.

Preventing such race conditions across two or more threads requires the use of *synchronization primitives* in the form of *locks* or *signals*. A lock can be used to allow a single thread to gain exclusive access to a shared resource (a mutex), or to allow a fixed number of threads access to a limited number of resources. A signal acts as a wait handle, providing a predetermined point in the code execution path where multiple threads can wait for one another before proceeding. It is important to remember that thread synchronization is cooperative. If a thread bypasses a synchronization mechanism and accesses the protected resource directly, the synchronization mechanism will not be effective. Errors or exceptions occurring in the code after a thread acquires a lock must be adequately handled and recovered so that the lock is released before the thread terminates. Otherwise, the lock will never be released and any threads already waiting to acquire the resource will block indefinitely, leading to a hung application.

In many ways, it would be better to have no interaction or shared resources between threads, but this would limit the flexibility of solutions. For instance, in a system that consists of one module for reading inputs and another module that processes these values and calculates an output, since the input arrivals are potentially asynchronous in nature, and the amount of time required to process the data could be variable, running both input and output operations on a single thread can create undesirable characteristics. The output could be delayed due to the timing of the input, or the input module could appear to be unresponsive due to the long processing time required for calculating each output. Separating the work into two threads allows each component to run at its own pace, determined by either the arrival rate of the inputs or the processing time requirement of

the outputs. However, this design requires the use of a shared data structure between the two threads.

Even with non-shared resources, race conditions can unexpectedly occur with numeric types that are wider than the system data bus. Changing the value of a 16-bit integer on an 8-bit system, or 64-bit integer on a 32-bit system, requires reading and updating two consecutive memory locations. A thread context switch that occurs in the middle of the update cycle can cause the final value written to be in error.

Processing long-running tasks on the user-interface thread can make an application look like it has become nonresponsive or completely broken, which eventually leads to a poor user experience. Performing slow network activities, such as downloading large files, in the background can appear to improve system performance since the processor will be able to do other useful work while delayed tasks are waiting on external events.

When poorly thought out and improperly implemented, the consequences of using multithread synchronization primitives can be excessive deadlocks and a priority inversion, as previously described. Creating, maintaining, and synchronizing another thread creates an additional system workload overhead. The performance improvement of the new thread has to outweigh its overhead in order to yield a net benefit. An implementer may feel that churning out threads to do work in the background will necessarily improve the application's performance. However, this improvement will only be realized if all threads are doing useful work and are not competing for shared resources. Once threads need to stop, wait, and synchronize with other tasks, the associated performance penalties can outweigh the potential performance gains. No

amount of increase in processor clock speed will make threads that are constantly blocking run any faster.

Solutions designed for multithreading should prefer fine-grained locks and lock only the smallest possible part that needs exclusive access, not entire methods or class data structures. Locks should be acquired as late as possible in the code path and released as soon as the work is completed. The number of threads should not exceed the availability of dependent resources. When necessary, add a thread per available resource and then scale out the number of resources to reduce performance bottlenecks. Avoid creating threads on a per request basis, and instead manage long-running threads and message queues. Considering the performance implications, design and implement lock-free alternatives such as those presented in (40). Many of these access algorithms use a timestamp or thread ID to mark the value most recently written. Before updating with a new value, the ID of the current value is compared with the value the thread obtained when the data was initially read. If these are different, another thread has made a change, so the new value should be reread and the work recalculated. These are not perfect solutions, but they solve many types of performance problems related to locking critical sections in many circumstances.

5.5 Architectural Modeling and Languages for Real-time

Computer programming languages are oriented towards solving a problem through the sequential execution of a series of instructions. Concurrency and parallelism lack the support of available direct language constructs. Imperative languages like C++ are wanting in the amount of declarative expressions available for the parallel operations required of real-time systems. High-level-language keywords and constructs that say

“Run these things together and put the combined results here,” are either nonexistent or poorly supported by application development tools. Manual coding efforts to coerce a better optimized parallel execution plan, using tactics such as loop unrolling where one large loop is broken into multiple smaller loops, can yield significant performance improvements but they result in code that is harder to support and maintain. During implementation, developers are rarely aware of the underlying execution-time operating environment, which requires different versions be implemented for the specific number of processors available at run time.

Establishing some form of system model should not be an uncommon activity during analysis and design stages of any system development methodology. Behavioral modeling is especially critical in the design of real-time systems because the available languages and tools are presently incapable of automatically revealing timing constraint violations and general resource contention issues. As such, it is necessary to develop models that permit the exploration of critical and essential features regarding system operation. Good models will expose otherwise hidden information regarding the characteristics of the system and potential modes of operational failure. Using the information obtained from the modeling exercise allows a designer to explicitly specify how the various components will fit together. These same models can also be used to verify system behaviors post-implementation. Ideal modeling tools allow the direct translation from design artifact to executable code.

While essential to the design process, structural models depicting the physical composition and relationships between the key solution modules are insufficient for building real-time systems. Some form of dynamic modeling, showing the causal patterns

between external events or triggers and actions, is necessary to fully characterize the system. Various types of statecharts and data-flow diagrams can be used to show state transitions in the context of pre and post conditions as well as any actions that are performed during the transition. Timing constraints can be more difficult to capture, but analysis by some type of Petri-net is sometimes worthwhile. Petri-nets are better for determining the existence of self (internal) loops and modeling the synchronization requirements between distributed tasks, and are usually easier to construct than queuing nets.

Object-oriented analysis and design practices for real time is visited by *Rumbaugh et al.* (41) and partially addressed through the definition of the *Object Model* (structures and relationships), the *Dynamic Model* (events and states), and the *Functional Model* (operations and their data flows). The work was transformational at the time of publication in that it provided an easy to read and understand class-oriented symbology along with a consistent set of rules for their construction. However, it lacked the necessary semantics to properly describe the interdependencies of concurrent threads of execution, and instead repackaged classic computer science constructs, such as statecharts, to fill in the missing pieces.

The development of the *Unified Modeling Language* (UML) (42) was the synthesis and evolution of various ideas on object-oriented practices. The initial impetus behind UML was to provide a unification of the best parts of a number of software and relational database modeling and diagramming methods, such as the *Object Modeling Technique* (OMT), from names like *Booch, Gane & Sarsen, Jackson*, and others including *Rumbaugh* (41). Over time, UML has become excessively large and unwieldy,

embodying an incomprehensible degree of complexity. With UML, there are many equivalent ways to express the same model, resulting in several types of documentation inconsistencies. Over time, UML has essentially turned into a basis for heavyweight and expensive analysis methodology tools for commercial sale rather than a cohesive system modeling language. *Rational Rose* from IBM (43) is one example of such a toolset that has been corporately acquired and twisted to support the *Rational Unified Process* (RUP). A RUP implementation requires significant investment in skills and training to properly execute, often out of proportion to the end system being developed. A methodology is usually represented by a gated process, a sequence of tasks and outputs, that is presumed to deliver a consistently repeatable outcome. If, in order to produce results, a methodology is overly dependent on the experience and skills of the people involved in its execution, then it must contain indefinable and uncodifiable properties, which limit its repeatability. Once the project development emphasis has shifted towards a specific methodology implementation, the big process is no longer about doing the system design work.

When a minimalist view of UML is taken, where only the parts of UML that are necessary to communicate the ideas that one wishes to represent are used, the results can be satisfactory. Capturing design requirements in the form of *use cases* and presenting high-level structures and interactions in the form of *object models* and *activity diagrams* is generally a worthwhile exercise. While essential for functional analysis and requirements verification, however, once these models have been completed, the path to object identification and implementation is usually unclear. One approach to object identification from use cases in real-time embedded systems is discussed in (44). Since

there is typically no one-to-one mapping between the two models, activity diagrams are first constructed from the use cases, which then serve as the basis for object identification.

Architectural description languages (ADL) based on Milner's (45) *Calculus of Communicating Systems* (CCS) or *Communicating Sequential Processes* (CSP) can be used to specify the implementation of a programmable architecture for both hardware and software. *Wright*, *Darwin*, or *Piccola* are typical ADL examples (46). ADLs differ from UML in that they focus on the descriptions of components rather than on the whole solution. Simulation using ADLs allows for design-space exploration and evaluation of candidate architectures at a level of abstraction that prevents binding to specific point solutions (47). Critics of an ADL-based approach (48) are instead seeking architectural *design* languages that, rather than describing the current practice, aid in the identification of the characteristics of correct solutions for future practices.

Prototyping system behavioral requirements from a well-defined CCS calculus to C# and .NET are discussed in (46) where CCS processes and actions are mapped to C# classes and methods, respectively. By describing the interactions across *input ports* and *output ports*, the manner in which messages are passed between processes and how they communicate with each other can be examined.

In a model-driven development exercise, integration with reusable standards-based commercial components can reduce solution costs and time to market factors. Overall, however, the integration process requires simplification and automation. Auto-generation tools for synthesizing artifacts from models that support middleware component

technology aid in system realization. The challenges associated with one approach for component modeling in distributed real-time embedded systems are identified and partially addressed in (49). The platform-independent design language developed uses Java and the *Component Object Resource Broker Architecture (CORBA) Component System (CCS)*.

A component model framework implementation for extending C++ to support concurrency, thereby integrating object-orientation and concurrency, is developed in (50). Heavily influenced by *Concurrent Pascal*, the work involves the definition of an *active object* that combines the concept of an object with that of a process, and claims a 50% reduction in code size as a beneficial side effect. A reference implementation is given in the form of a CD player developed for a commercial manufacturer.

Various programming languages are available to choose from for the development of real-time systems. While it is possible to use nothing but assembler, high-level languages such as C offer advantages of portability, maintainability, and developer productivity over processor-specific assembly language implementations. While historically the choice most often made has been C, C++ and an object-oriented paradigm are becoming increasingly more prevalent (50) and accepted in industry. Not all features are available in every language, so hybrid assembler approaches are not atypical of many real-time solutions.

A listing of language features necessary to support real-time programming are identified in (51). These features are grouped into four categories: *essential*, *primary*, *secondary*, and *performance*. The essential language features given for real-time system

implementation include the ability to access and control hardware, availability of bit manipulation instructions, support for interrupt handling, and accessibility of pointers for use with dynamic data structures. Fundamentally, these essential features are more representative of specific patterns of solutions for traditional hardware-centric systems than requirements for real-time system development. Access and control of hardware can be, and should be in a good design, implemented in a device-appropriate way and then abstracted for use by higher-level functions, thus removing the need for general programming languages to support these low-level operations. Such designs create better modularity and reusability, and allow the intervention of security access control mechanisms of an underlying operating system. Interrupts and their handling are usually a way of gaining control and somehow manipulating a type of thread creation mechanism. Since the processor state is stored and then retrieved during an interrupt event, the execution model is a kind of special-purpose *micro-thread*. Access to hardware interrupts or external events is better done through a functional abstraction, with or without operating system support, rather than directly through high-level code. The use of pointers for creating and accessing dynamically created data structures, such as linked lists, can likewise be implemented through abstractions in the form of reference types or smart pointers, which require no direct memory access capabilities on the part of the high-level language. Indeed, many of the essential features that are identified in (51) are not critical or essential real-time language requirements at all.

When developing real-time applications, programming languages need to be able to create and control tasks at the thread level, they need to be able to influence the scheduler (if one is used) either directly or via setting thread priorities, and they require diverse

types of atomic inter-thread synchronization primitives. These things are really more operating system issues and not limitations of a particular programming language. While some languages may be better suited than others for solving a particular problem, as long as a language is supported by the operating system environment and libraries of thread management and synchronization (and possibly timing control) functionality are available, then in actuality any high-level language can be used to develop real-time applications.

The particular choice of implementation language is more often determined by operating system support and the manner in which the application and the OS will be linked together. If the linking, the combining of the operating system with the application, is done during or immediately after compile time (early binding), then the source language of the operating system will predicate the application language. The C and C++ languages don't interoperate well with other languages without help—a large portion of the development of Windows™ has involved getting C to play nice with other development languages. Operating systems requiring early binding that have been written mainly in C will carry with them language specific interoperability restrictions.

If the operating system provides an application loader and supports late binding, the options for development languages become somewhat broader. More sophisticated environments will provide memory management features, such as garbage collection to automatically reclaim unused memory references. These services can limit the predictability of real-time operations by introducing a potential near-random workload on the system.

The .NET Framework provides support for creating and managing threads and thread priorities, as well as a variety of mutex types and exception-safe critical-section lock devices. Many of the built-in collection classes (arrays and lists) include a synchronized interface for cross-thread call capabilities. Asynchronous method calls, callbacks, and software-based events are all integral parts of the run-time environment. While the real-time suitability assessment of (52) is based on version 1.0 of the .NET Framework (version 4.0 will be released shortly), it does correctly identify a weakness in the inability to predict or specify when a scheduled thread will start. Presently, there is a timing ambiguity over when a started thread will actually begin executing. If better control is required, it may be possible to use the *High Performance Event Timer* (HPET) (53) on newer PC system boards as a work around to this problem.

Chapter 6 Development Framework Overview

The GNSS Receiver Development Framework project has been designed and developed to address the issues of thread management, interprocess 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 polymorphism;
- 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 offline post-processing tools and utilities, the framework is intended to deliver the critical performance characteristics necessary to achieve real-time receiver operation.

A key requirement in the framework 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 models for benefit evaluation. By leveraging object-oriented design techniques such as polymorphism, model 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 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 specific requirements. This framework is not intended to deliver a toolkit or a collection of library components that must be incorporated into other applications—instead, it is a development guidance reference for how receiver

applications should be structured and constructed to achieve the most worthwhile results in the shortest timeframe with the greatest opportunity for reuse and extension.

Since the operation of a receiver is to undo that which the transmitter does, the receiver components and object models provided, or subsequently developed, could also be modified and extended to create simulation-based signal generators for testing receiver operations under controlled input conditions. A mathematical model of a signal could easily be implemented that creates a binary file to be used as an input source for testing receiver performance.

6.1 Receiver Framework Architecture Diagram

The receiver framework diagram, shown in Figure 6-1, serves as the solution overview and a roadmap to the following documentation. Each functional block is documented in greater detail in sections that describe the operational requirements, characteristics, and interactions with other system components. While the intention of the framework is to provide a development tool for the research of GNSS and other spread-spectrum receivers, the modular definition of the components allows for a kind of plug-and-play approach to the overall system implementation.

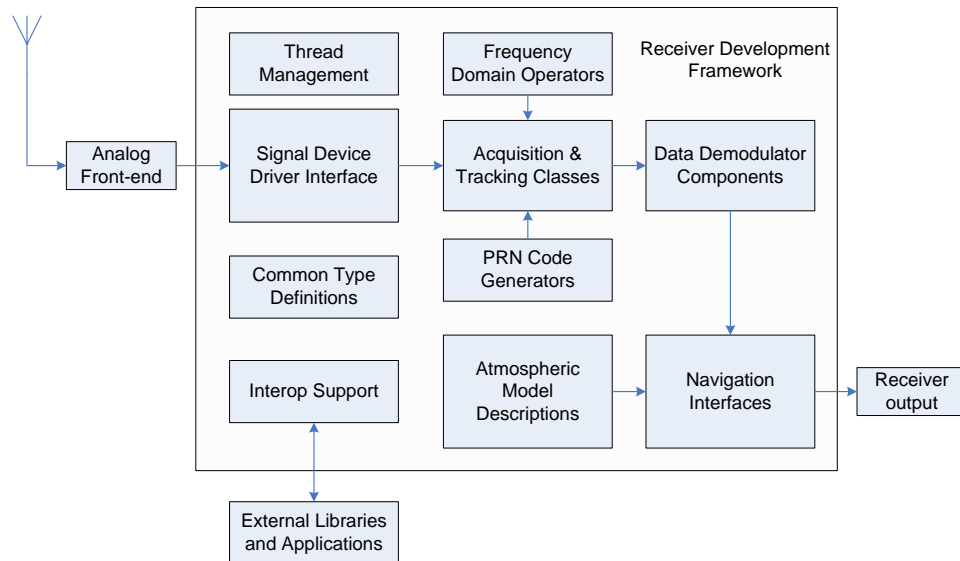


Figure 6-1—Block diagram model of the Receiver Development Framework

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 is for this reason that the system was developed in such a way that it more closely resembles the block diagrams of the hardware that it represents.

The object-oriented design takes advantage of the reusability available through inheritance and polymorphism. 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 subsystems.

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. Support for multithreaded processes is largely dependent on the underlying operating system characteristics.

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 phase-lock loop (PLL), delay-lock loop (DLL), and a numerically-controlled oscillator (NCO) are provided as part of this module. Solution implementations vary widely in their approach to signal

detection and demodulation, so these components are offered as reference utilities that may be used, altered, extended, and replaced as needed.

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.

Atmospheric models and navigation message interfaces are identified as parts of the framework, but no reference implementation has been provided as of yet. Libraries from other sources such as the GPSTk toolkit (54) from the University of Texas at Austin have been integrated into the framework, and support for these functions could easily be extended. Details of this integration work are provided in Appendix A.

6.2 Pipeline Processing Model

The preferred batch-oriented approach (6) to software-based post-processing signal demodulation is a compute-intensive solution, which often requires several hours of offline execution time to analyze even just 30-40 seconds worth of captured and stored data. Efforts to achieve real-time performance through optimization of the individual post-processing stages usually follow a similar sequential program flow as was originally provided, without creating extensions for parallelism. As such, it is unlikely that these works will ever manage to attain the throughput required for a real-time signal processing application.

By dividing the signal processing activities into two parts, data capture followed by iterative calculations—*capture-then-process*—the resulting CPU workload becomes non-

uniform with time, as illustrated in Figure 6-2. The sampling task is time critical but low workload, while the calculations are not time critical but they are high workload.

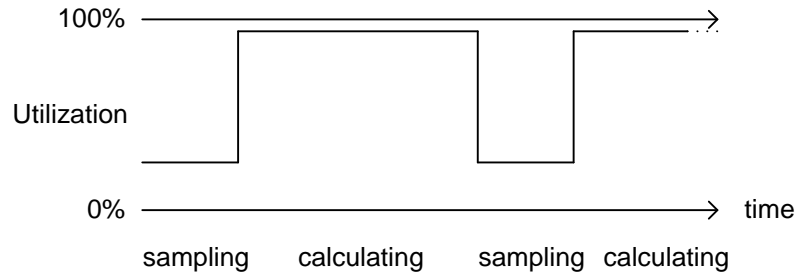


Figure 6-2—CPU Workload with a capture-then-process signal processing approach

When a real-time solution is attempted using this processing model, it becomes necessary to run both the sampling and calculating activities in parallel. Individually, it may be the case that neither activity exceeds the performance capacity of the system processor, but when combined the deficiency in spare CPU cycles makes itself apparent. One approach to contend with this workload balance issue is to run each activity on system threads with suitably higher or lower priorities.

If the sampling work is performed on a higher-priority thread, the calculation activities will be preempted often and require an excessive amount of time to complete. As a result, the signal samples will be arriving at a rate faster than the system can process them and will have to be queued or buffered. Keeping the system synchronized under these conditions is a challenging exercise. With limited system memory resources, signal buffering will eventually overflow and the application will have to cease functioning.

Likewise, if the sampling work is performed on a lower-priority thread than the execution of the calculation activities, giving more processor time to run the

computational workload, signal samples will undoubtedly be missed. At best, dropping input data samples will cause the system to lose signal synchronization, but it can also be a pernicious and hard to detect source of data errors.

Overall, to execute the capture-then-process computational model in real time is extremely challenging and requires careful performance tuning and optimizations in order to realize adequate results. It is for reasons of complexity and performance that previous solutions based on this approach have not been real time.

6.2.1 Synchronous Pipeline

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 during 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.

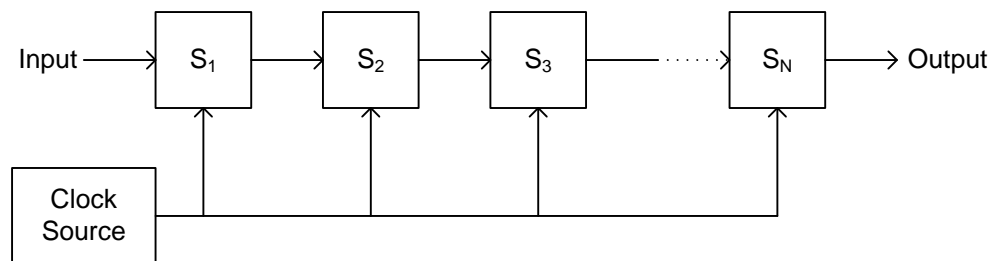


Figure 6-3—Pipeline structure with a common clock

An N-stage pipeline is shown in Figure 6-3. While it takes an amount of time equal to N stages to initialize and fill the pipeline, once filled a new result is produced on each

clock cycle. It is the common shared clock that makes this a synchronous pipeline, the rate of which must be lower than the maximum input-to-output delay of the slowest block in the chain. In hardware, the clock source is usually the output of an 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.

Without access to a shared time reference, pipelines are difficult to build using software constructs. However, the same pipeline structure as presented in Figure 6-3 can be achieved in software applications with events and event handlers. To do so requires that 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 6-4.

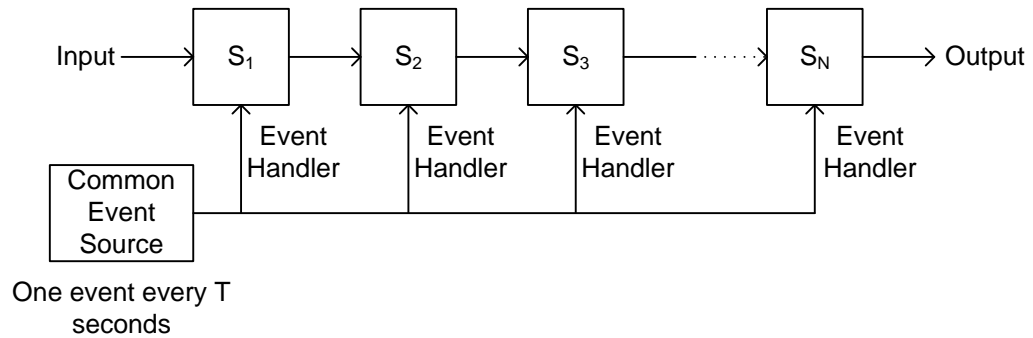


Figure 6-4—Event-driven synchronous pipeline process

An event, such as a signal sample arrival from a common source, is raised every T seconds, making the event frequency $E_f = 1/T$. The N -stages of the pipeline are filled in $N \times T$ seconds and a new result is produced every T seconds. Basically, the structure mimics the properties of a hardware-based implementation, including running at the slowest stage performance level. The throughput benefit of this model results from the

more evenly distributed workload due to the processing that occurs between the event-trigger intervals.

Unfortunately, with software-based events, the 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 additional semaphores, wait handles, or other shared synchronization primitives, such as a synchronized queue between stages, to reliably execute in software, each of which negatively impacts system performance and increases implementation complexity.

6.2.2 Asynchronous Pipeline

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 6-5. 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 stable. 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.

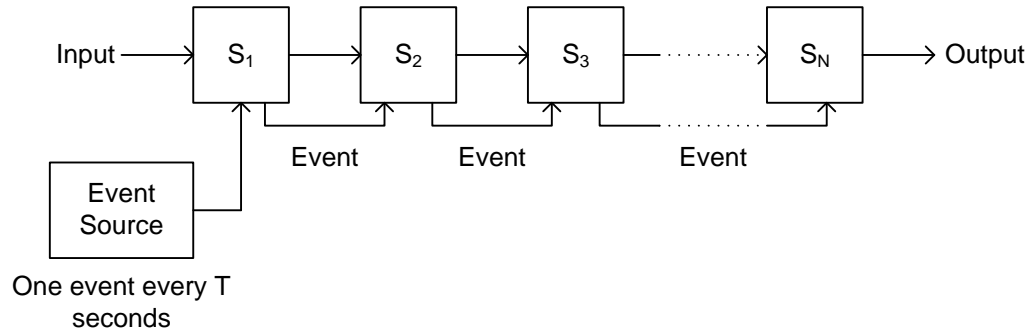


Figure 6-5—Asynchronous software pipeline model using event coupling between successive stages

As shown in Figure 6-5, the pipeline is initiated by the event source at stage S_1 occurring at regular intervals every T seconds. After responding to the event and updating its output, S_1 then raises its own event at time $T + \tau_1$, where τ_1 is the S_1 processing delay. The new event signals to downstream subscribers that the outputs from stage S_1 have been updated and are stable. Each stage repeats this sequence, cascading the event and the data sample along the way.

The total N -stage pipeline propagation delay, τ_{total} , can be given by

$$\tau_{total} = \sum_{k=1}^N \tau_k \quad \mathbf{6-1}$$

As long as τ_{total} is kept to less than the event arrival interval T , the performance characteristics of the asynchronous pipeline are equivalent to those of the synchronous one in that the N -stages of the pipeline are filled in $N \times T$ seconds and a new result will be produced every T seconds, without the drawback of having an unpredictable event delivery order. The delivery sequence is entirely determined by the pipeline organization,

and events can be triggered at different rates that are appropriate for the output timing characteristics of the component.

For real-time operation, it is important 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 6-6 for the pipeline stage S_1 . The stage properties include input, output and time data values that are inherited from the component base class.

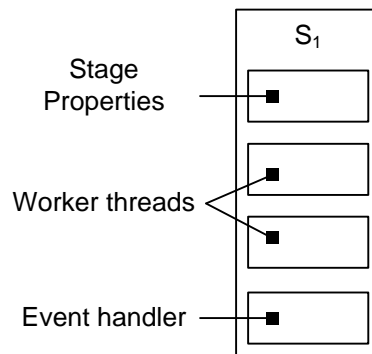


Figure 6-6—Pipeline stage 1 event-handler structure with separate worker threads

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.

6.2.3 Pipeline Component

Each component of the pipeline is derived from a common base class, shown in Figure 6-7, that implements the *IPipelineComponent* interface representing 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.

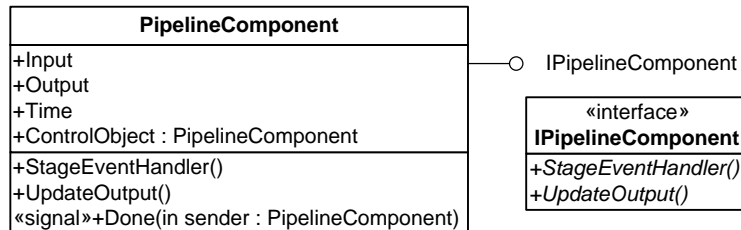


Figure 6-7—Pipeline component object model

The *PipelineComponent* base class provides default behaviors for the stage’s *StageEventHandler* and *UpdateOutput* methods. New pipeline components can be defined by inheriting from this base and overriding the functionality of either *StageEventHandler* or *UpdateOutput*, or both. Components can be made to behave as

basic gain stages, where the output is a simple product of the input, or they can perform more complex functions such as summers and integrators.

The *ControlObject* property of a *PipelineComponent* can be used to externally regulate the output of a stage. Controlling transfer-function properties based on input signal characteristics, such as gain or correlator detection thresholds, or adjusting frequency and phase for signal mixing, are the expected typical uses of the *ControlObject* property. If a specific type of controller object is required by a stage, such as the output of a low-pass filter for a voltage-controlled oscillator in a phase-lock loop, the provided base-type reference may be appropriately cast to the required derived type.

Since the *PipelineComponent* event signal, *Done*, includes a parameter that is a reference to the signal sender, the output value is actually passed from one stage to the next in the event itself. Consequently, pipelines can be organized with multiple parallel pathways by assigning the event handlers from two or more components to the same event source. Specialized observer components can tap into pipeline outputs at any stage, recording time and sample values to disk or sending them to a graphical display object for plotting.

Figure 6-8 shows an example of a system with a forward control path for gain, a feedback path for regulation, and a signal tap for data recording. An input stage sends a data value through an event to a mixer and automatic-gain control (AGC) stages. The AGC stage sets its *Output* property based on the average level of the input, and although it may raise an event, it has no listeners so no additional action is triggered. The mixer stage produces an output based on some function of the input signal and its

ControlObject property, the output of a downstream phase detector. The mixer then raises its *Done* event. An integrator stage responds to the event from the mixer and uses the input value and the output from its *ControlObject* property, the AGC component, to determine the length of integration time.

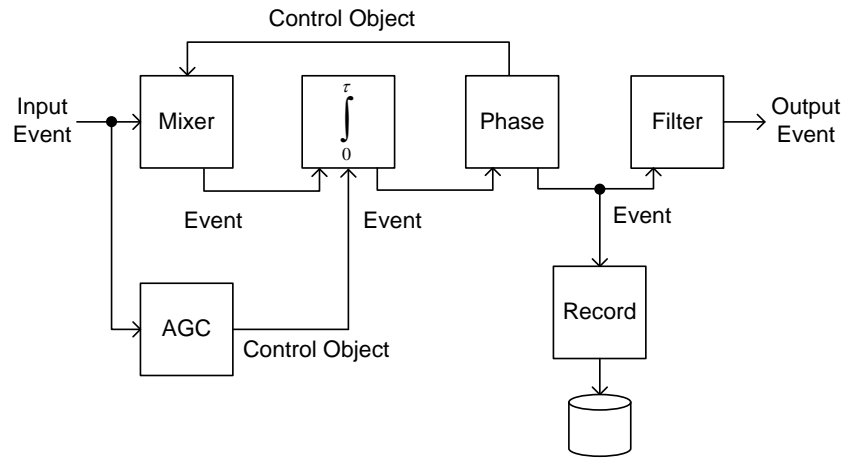


Figure 6-8—Example pipeline configuration showing feed-forward and feedback control objects with parallel pathways

The integrator updates its output and triggers an event for the Phase detector. The phase detector output is looped back to control the *Output* property of the earlier mixer stage, and its triggered event passes along an output to the filter component that creates the final result. The phase *Done* event is also connected to a data record object that copies the *Time* and *Output* information from the event to a persistent data store, such as a file or relational database.

Care must be taken not to stall the main event thread executing the *StageEventHandler* by invoking long-running or overly involved operations. Lengthier processing loops, when required, can be executed by creating and initializing a separate worker thread.

6.2.3.1 Properties

Input: the data value passed from the output of the previous stage that is valid for the current time index.

Output: that data value that will be passed to the next stage in the pipeline after the *Done* event is raised. Typically,

$$Output = f(Input, Time, ControlObject.Output, instance\ state\ variables)$$

Time: a relative or absolute time value for the sample currently being processed. Outputs that are calculated from time-based functions use the value of this property as the time parameter. The difference in time between two successive samples will be the stage-1 event source's event rate, T .

ControlObject: allows feed-forward or feedback control from an external pipeline component. The *ControlObject.Output* property may be used in the calculation of the stage's *Output* value.

6.2.3.2 Methods

StageEventHandler: the virtual event handler for the *Done* event. The default implementation provided by the *PipelineComponent* base class, which can be overridden in derived classes, performs the following sequence of operations:

- 1) Sets the *Time* property to the event sender *Time*
- 2) Sets the *Input* property to the event sender *Output*
- 3) Calls the virtual method *UpdateOutput* to change the *Output* property
- 4) Raises the *Done* event and forwards a reference to itself as the sender object

This default component behavior is equivalent to the unit delay (z^{-1}) transfer function.

UpdateOutput: virtual method that is used to calculate the *Output* property based on the *Input*, *Time*, and *ControlObject* property values. The base-class version of *StageEventHandler* calls the *UpdateOutput* method after changing the *Input* and *Time* properties and before it raises the *Done* event. The default function performed by *UpdateOutput* is to set the $Output = Input$, thereby acting as a single-stage delay. The majority of new classes derived from *PipelineComponent* should only need to provide a definition for *UpdateOutput* and inherit all other attributes from the base class.

6.2.3.3 Events

Done: signals to other stages that this object has finished updating its *Output* property. A reference to the sender object is passed as a parameter.

6.2.4 Pipeline Container

The pipeline components are configured into the desired sequential structure, dependent events connected, and control objects assigned as part of a *PipelineContainer* class. All that is required to run multiple pipelines in parallel is to create multiple container instances and connect them to the front-end event source.

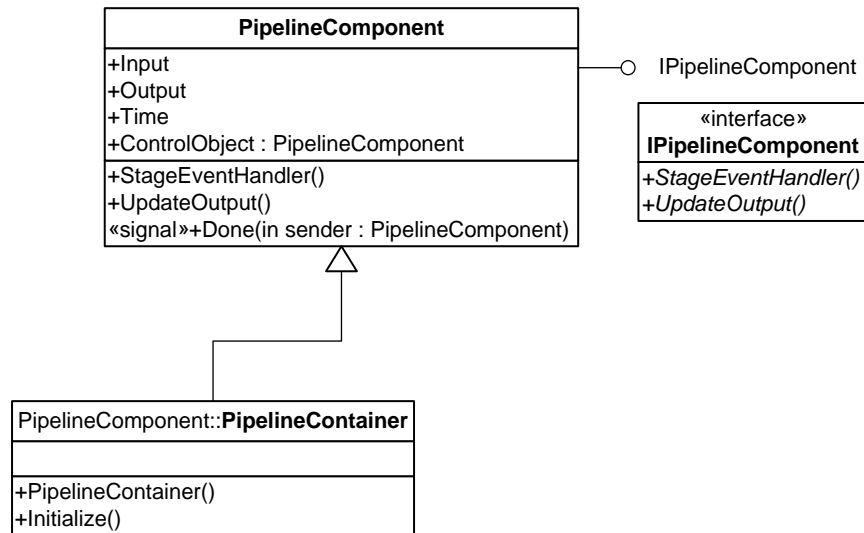


Figure 6-9—PipelineContainer class diagram

The *PipelineContainer* type is derived from the *PipelineComponent* so that larger pipelines can be comprised of smaller ones; pipelines may contain stages that are themselves pipelines.

6.2.4.1 Properties

Inherits the properties of *Input*, *Output*, and *Time* from *PipelineComponent*. Derived types may need to add properties that expose selected state information of the contained *PipelineComponents* as computational results to objects that created the pipeline. In the case of the reference GPS application, a derived *PipelineContainer* class is defined that extracts the carrier and code phases and carrier Doppler values, as well as the latest output navigation data bits, for a single tracked satellite as properties of the container.

6.2.4.2 Methods

PipelineContainer: a class constructor method that gets called when an instance is created. The various constituents of the pipeline will need to be initialized and the events

hooked together and this is a good place to do it. The bulk of the work may be put into the *Initialize* method, described below, so that the pipeline may be reinitialized without having to be recreated.

Initialize: used to reinitialize the pipeline configuration.

6.2.4.3 Events

PipelineContainer inherits the *Done* event from *PipelineComponent*. Additional events to notify external objects of changes in the pipeline state may be added to derived class instances as necessary.

6.2.5 Phase-Lock Loop Pipeline Component

The purpose of the PLL is to accurately track the frequency and phase of the incoming signal carrier after the DLL has removed a properly aligned code. If the location of the navigation data bits is known, a coherent PLL can be used. However, usually this is not the case, so a Costas-type or squaring discriminator is required. 1st and 2nd-order PLLs are described and characterized in (6), and the PLL implementation provided by the receiver framework is based largely on that work. Reference (55) provides an analysis of the performance characteristics of GPS weak-signal tracking using a 3rd-order PLL. An optimal loop filter for the discrete-time PLL operating at steady-state is obtained in (56) using Wiener's analysis of the minimum-mean-squared error of the phase difference between the input signal and the PLL output. Many of the texts consider only the analog nature of PLLs, and ignore the consequences of their discrete characterizations, such as (3) and (4).

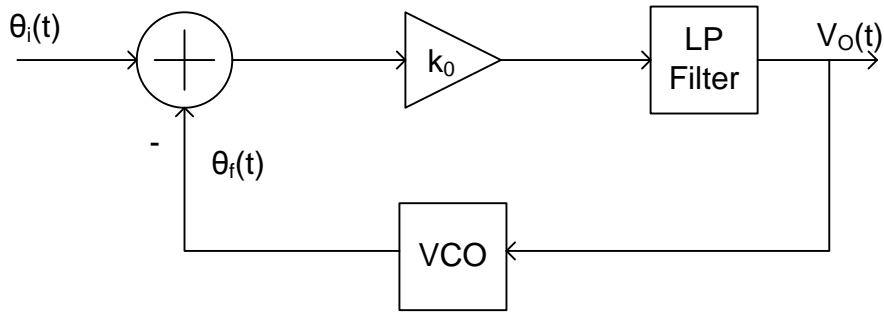


Figure 6-10—A basic phase-lock loop

Figure 6-10 shows the time-domain representation of a basic PLL. The output of the phase comparison is the difference between the phases of the input signal and the output of the voltage-controlled oscillator (VCO). The transfer function of the low-pass (LP) filter is given by

$$F(z) = C_1 + \frac{C_2}{1 - z^{-1}} \quad 6-2$$

which is simply a first-order PI (proportional integral) controller.

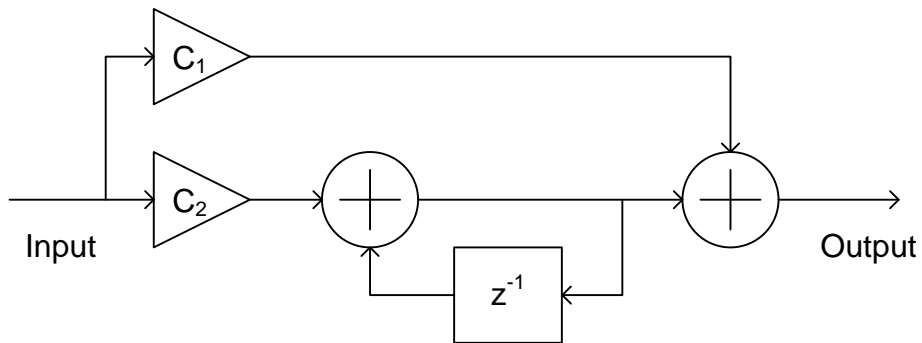


Figure 6-11—PI controller as the filter function for a PLL

The VCO can be replaced with

$$N(z) = \frac{\theta_f(z)}{V_{o(z)}} \equiv \frac{k_1 z^{-1}}{1 - z^{-1}} \quad 6-3$$

The PLL transfer function can be written as

$$H(z) = \frac{\theta_f(z)}{\theta_i(z)} = \frac{k_0 F(z) N(z)}{1 + k_0 F(z) N(z)} \quad 6-4$$

Substituting equations 6-2 and 6-3 into 6-4 results in

$$H(z) = \frac{k_0 k_1 (C_1 + C_2) z^{-1} - k_0 k_1 C_1 z^{-2}}{1 + [k_0 k_1 (C_1 + C_2) - 2] z^{-1} + (1 - k_0 k_1 C_1) z^{-2}} \quad 6-5$$

From which, it can be shown (6) that

$$C_1 = \frac{1}{k_0 k_1} \frac{8\zeta \omega_n T_S}{(4 + 4\zeta \omega_n T_S + (\omega_n T_S)^2)} \quad 6-6$$

$$C_2 = \frac{1}{k_0 k_1} \frac{4(\omega_n T_S)^2}{(4 + 4\zeta \omega_n T_S + (\omega_n T_S)^2)} \quad 6-7$$

Where the natural frequency, ω_n , and the damping factor, ζ , can be found from

$$\omega_n = \sqrt{\frac{k_0 k_1}{\tau_1}} \quad 6-8$$

$$\zeta = \frac{1}{2} \omega_n \tau_2 \quad 6-9$$

Additionally,

$$\tau_1 = \frac{T_S}{C_2} \quad \mathbf{6-10}$$

$$\tau_2 = \frac{2\tau_1 C_1 + T_S}{2} \quad \mathbf{6-11}$$

The desired loop characteristic usually specified is the noise bandwidth, B_n , which for the second-order system implemented is equal to

$$\begin{aligned} B_n &= \int_0^{\infty} |H(\omega)|^2 d\omega \\ &= \frac{\omega_n}{2} \left(\zeta + \frac{1}{4\zeta} \right) \end{aligned} \quad \mathbf{6-12}$$

Typical values for the noise bandwidth are in the range of 15-25 V/\sqrt{Hz} , and the damping factor is usually specified to be the critically-damped value of 0.7; i.e., the poles of the characteristic equation will be real and equal.

There are many different, but equivalent, ways to code a digital filter and control loop, so the stability effects due to the loop gains k_0 and k_1 are not able to be easily generalized. Ensuring stability for the PLL involves evaluating a modified Routh-Hourwitz criteria using a bilinear transform to the ω -domain (not the best choice for a discrete system), or preferably conducting Jury's stability test (57). A few of the controls related details for tracking loop implementations are provided in Appendix B.

Loop stability as related to the signal sample rate, T_S , for the classic Type-2 PLL has been evaluated in (58). The open-loop gain, G_{OL} , is given by

$$G_{OL}(z) = (\omega_n T_S)^2 \frac{z \left(\frac{1}{2} + \frac{\tau_2}{T_S} \right) + \left(\frac{1}{2} - \frac{\tau_2}{T_S} \right)}{(z - 1)^2} \quad \mathbf{6-13}$$

and the resulting gain margin is,

$$G_M = -20 \log(\zeta \omega_n T_S) \quad \mathbf{6-14}$$

However, the gain margin is only defined provided that $\omega_n T_S < 4\zeta$.

In the situation where the PLL is operated in a coherent integrate-and-dump configuration, it is important to realize that the value of T_S will correspond to the integration interval (typically, the length of a spreading code) and not the actual sampling rate. As a consequence, the gain margin may be considerably smaller than expected and the combined code and carrier loop could exhibit a loss of signal tracking due to loop instability.

The most frequently encountered PLL phase discriminator functions (59) are provided in Table 6-1. In the table, the terms I^k and Q^k are the in-phase (real) and quadrature (imaginary) components of the signal. The recovered data bit stream is taken from the I arm of the output of the PLL, since every bit-transition will cause a 180° phase reversal in the carrier and will be seen as an abrupt change in the polarity of the PLL output.

| Discriminator | Description |
|--|--|
| $\theta = \text{sign}(I^k)Q^k$ | Computationally simple, output is proportional to the sine function |
| $\theta = I^k Q^k$ | Medium calculation complexity, output is proportional to sine function |
| $\theta = \tan^{-1}\left(\frac{Q^k}{I^k}\right)$ | High computational workload, output is the phase error |

Table 6-1—Typical PLL discriminator functions

The *PLLPipelineComponent* class diagram is shown in Figure 6-12. As the name suggests, the *PLLPipelineComponent* inherits from the *PipelineComponent* base class so that instances may be created and configured to participate in the signal processing pipeline. Consequently, the *PLLPipelineComponent* has access to the default Input, Output, and Time properties, as well as sending and receiving pipeline events.

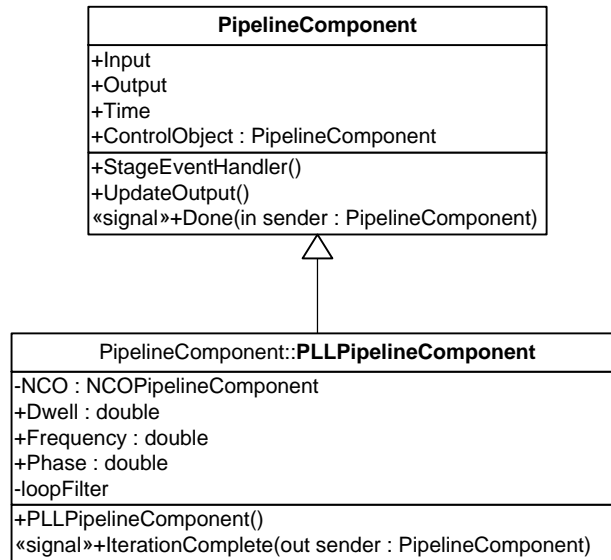


Figure 6-12—PLLPipelineComponent class diagram

6.2.5.1 Properties

NCO: A private reference to an instance of a numerically-controlled oscillator component. The PLL controls the frequency and phase properties of this object, which is also shared with the DLL component.

Dwell: A public property for controlling the length of time the PLL integrates the product of the incoming signal and the NCO output before evaluating the phase error and adjusting the frequency and phase of the NCO.

Frequency: Since the NCO is a privately held reference, the *Frequency* property is used by other pipeline members to view the current frequency of the tracked signal.

Phase: Used by other pipeline members to view the current phase of the tracked signal.

loopFilter: A private reference to an instance of a *Filter* class that performs a smoothing operation on the output of the discriminator function. The output of the filter is used to adjust the NCO frequency and phase.

6.2.5.2 Methods

PLLPipelineComponent: Class instance constructor that is used to initialize the various internal states.

6.2.5.3 Events

IterationComplete: In addition to the inherited *Done* event, the class also raises this event to signal to interested listeners that the amount of time equal to the dwell time has passed since the start of the last integration period. The firing of this event represents the completing of the *dump* part of *integrate and dump*.

6.2.6 Delay-Lock Loop

Following the acquisition process, the DLL receiver component performs the cross correlation between the incoming carrier, that has been mixed with the signal from the NCO and the locally-generated PRN code in order to produce a code-removed version of the signal that is then fed to the PLL for message bit recovery. Correlation can be performed as multiplication in the frequency domain, or as integration in the time domain. Usually, the integration method is harder (or, at least more mundane), so many post-processing receivers are implemented using the frequency-transform-based approach. The pipeline processing model of the receiver framework makes performing the integration a fairly simple and natural operation. After each event from the predecessor stage, the input value is added to a sum accumulator, and a test is made comparing the Time property to the integration interval end time. If the integration time

is complete, the total sum is normalized by multiplying the sum by the sampling time interval (dividing by the sampling rate) and the result is then produced as an output, after which the accumulated sum is reset to zero. A multichannel software correlator using a combined DSP and PC environment and a 12 MHz sampling rate is tested in (59).

The DLL keeps the locally-generated replica of the code time-aligned with the code in the carrier by maintaining three correlation results with an on-time (Prompt, P), an advanced (Early, E), and a delayed (Late, L) version of the local code. When the E correlation result is higher than the P result, the indication is that the local code is running behind since the advanced version fits better. Likewise, when the L result is higher, the local code is running advanced, since the delayed version fits better. The usual spacing of the early and the late codes is $\frac{1}{2}$ -chip interval ahead and behind the prompt timing.

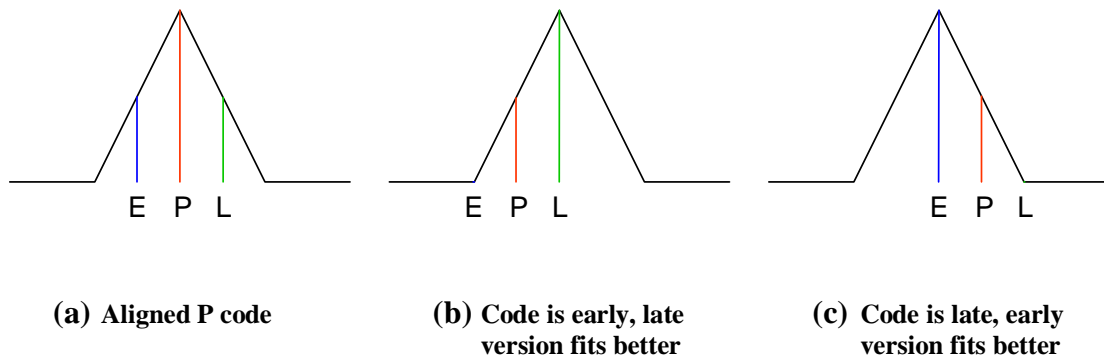


Figure 6-13—DLL E, P, L correlator outputs under on-time (a), early (b), and late conditions (c)

Depicted graphically, the DLL operates as shown in Figure 6-13. The triangular shaped outline represents the envelope of the expected range of values of the PRN cross correlation function. The goal is to keep the P correlator output in the apex of the triangle with the E and L values on either side as seen in (a). When the timing of the local code

generator advances, situation (b), getting ahead of where it should be, the output from the L correlator peaks higher than that of either the E or P correlators—the late version fits better, so the local code must be ahead of the received code. Feedback control from a discriminator function will cause the timing of the code generator to slow down, bringing it back into alignment. The reverse happens when the E correlator output peaks (c).

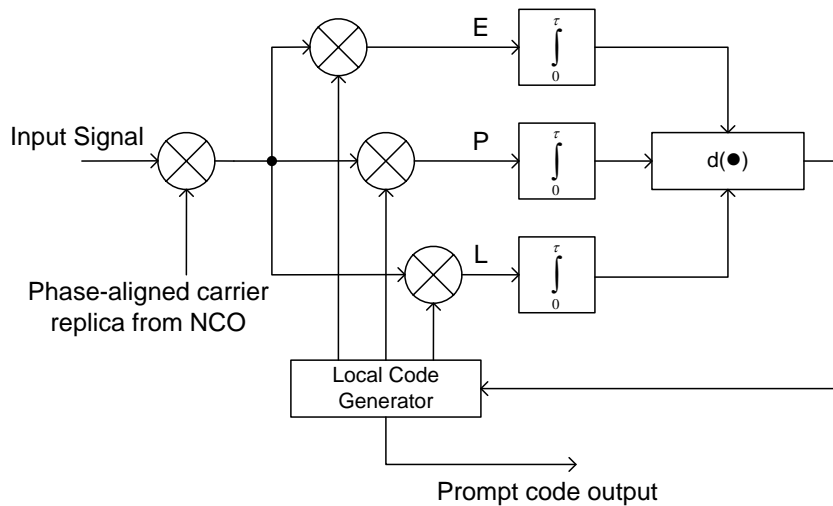


Figure 6-14—DLL correlator block diagram

A block diagram of the correlator section of a DLL is presented in Figure 6-14. The input signal from the front-end sampler is mixed with the output from the PLL controlled NCO and multiplied by early, prompt, and late code sets. After integrating for an interval of time, τ , usually equal to the code duration, the correlator outputs are then compared with a discriminator function to produce the feedback variable.

Various DLL discriminators are given in Table 6-2 from (59). In the table, the terms I_E, I_L, I_P and Q_E, Q_L, Q_P are the in-phase (real) and quadrature (imaginary) components of the Early, Late and Prompt correlator outputs.

| Type | Discriminator | Description |
|--------------|---|--|
| Coherent | $d = I_E - I_L$ | Simple discriminator, can only be used when the carrier phase and data-bit locations are known |
| Non-coherent | $d = (I_E^2 + Q_E^2) - (I_L^2 + Q_L^2)$ | Early power minus late power |
| | $d = \frac{(I_E^2 + Q_E^2) - (I_L^2 + Q_L^2)}{(I_E^2 + Q_E^2) + (I_L^2 + Q_L^2)}$ | Normalized early minus late power |
| | $d = I_P(I_E - I_L) + Q_P(Q_E - Q_L)$ | Dot product |

Table 6-2—Typical DLL discriminator functions

The receiver framework *DLLPipelineComponent* class diagram is provided in Figure 6-15. The reference implementation uses the normalized $E - L$ discriminator function to keep a PRN code generator instance aligned with the received signal. The *Output* property is set to the *Input* property times the PRN sequence value for the current sample time.

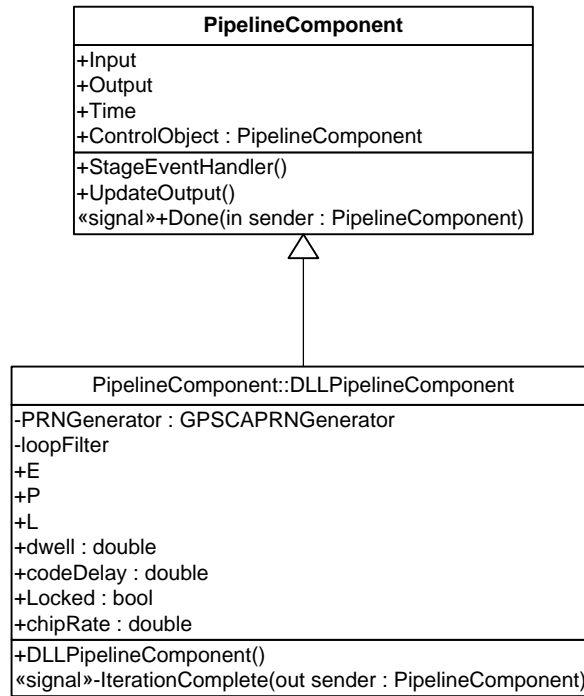


Figure 6-15—DLLPipelineComponent class diagram

The *DLLPipelineComponent* is derived from the *PipelineComponent* base class and as such, inherits that class’ properties, methods, and the *Done* event. It is then extended with DLL-specific characteristics.

6.2.6.1 Properties

PRNGenerator: A private reference to one of the *PRNGenerator* classes that will produce the correct code value in a sequence for a given point in time.

loopFilter: A private reference to an instance of a *Filter* class that performs a smoothing operation on the output of the discriminator function. The output of the filter is used to adjust the PRN generator delay property.

E: the current value of the integrator for the Early code correlator.

P: the current value of the integrator for the Prompt code correlator.

L: the current value of the integrator for the Late code correlator.

dwell: the length of time, τ , that the correlator will integrate for before invoking the discriminator function and adjusting the PRN code delay.

codeDelay: the current tracked PRN delay value. This output is used to calculate the signal transit time, modulo 1 ms, for pseudorange measurements.

Locked: indicates that the DLL is actively tracking a signal. If this property changes to false, the signal must be reacquired.

chipRate: the current PRN chip rate. With large Doppler shifts, the code generator chipping rate needs to be adjusted in order to maintain better tracking control.

6.2.6.2 Methods

DLLPipelineComponent: class constructor that is used for initializing the instance state variables.

6.2.6.3 Events

IterationComplete: an event that is raised upon complete of the integration interval.

6.2.7 Numerically Controlled Oscillator

The numerically-controlled oscillator generates the local version of the Doppler-adjusted signal that is mixed with the incoming signal prior to the DLL. There are several vector-based SIMD implementations of optimized algorithms, such as (27), but the reference implementation operates simply with the *sin* and *cos* math library functions.

The class diagram is given in Figure 6-16.

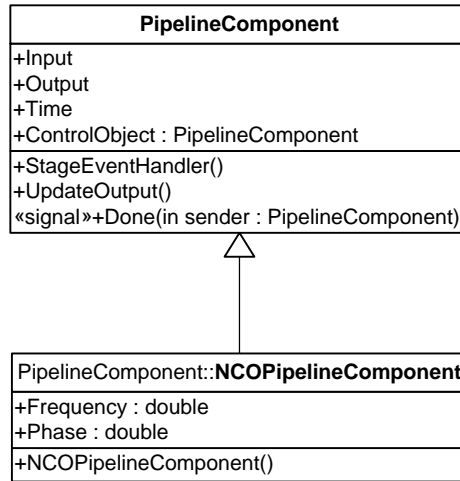


Figure 6-16—NCOPipelineComponent class diagram

6.2.7.1 Properties

The *Input*, *Output*, *Time* properties are inherited from the *PipelineComponent* base class.

$\text{Output.Real} = \text{Math.Cos}(2.0 * \text{Math.PI} * \text{Frequency} * \text{Time} + \text{Phase});$

$\text{Output.Imag} = \text{Math.Sin}(2.0 * \text{Math.PI} * \text{Frequency} * \text{Time} + \text{Phase});$

Frequency: The current frequency of the output signal.

Phase: The current phase of the output signal.

6.2.7.2 Methods

NCOPipelineComponent: The class constructor used for initialization.

6.2.7.3 Events

The *Done* event from the base class indicates the output value has changed.

6.2.8 Data Demodulator

The data demodulator component is connected to the *IterationComplete* event of the PLL, where it integrates the output for the bit rate time interval; 20 ms for the 50 bps data rate of the C/A signal. At the end of each integration period, a comparison is made to determine if the accumulated value is largely positive or negative. Positive results are then mapped to an output of binary 0, negative results produce a binary 1 (Appendix C). As each data bit is extracted, the demodulator raises an *IterationComplete* event that can be used to activate a navigation message formatting component.

The demodulator class diagram is given in Figure 6-17.

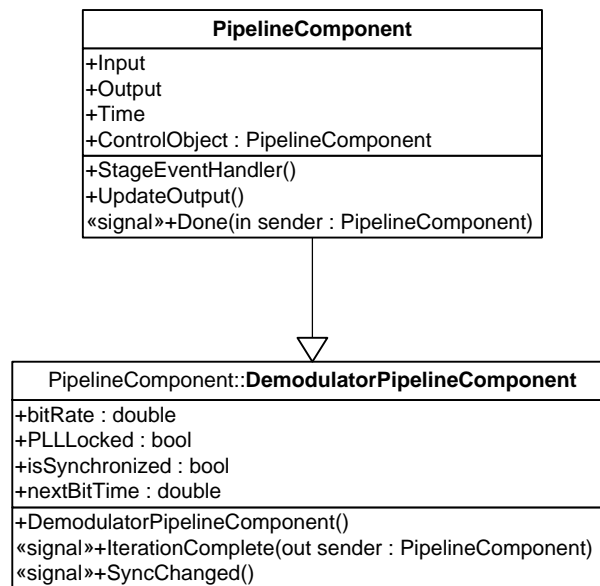


Figure 6-17—UML static object model for the demodulator component

6.2.8.1 Properties

The *Input*, *Output*, and *Time* properties are inherited from the *PipelineComponent* base class. The *Output* value is determined by accumulating the values from the PLL output and generating a 1 for a negative result, and a 0 for a positive result. It is possible,

depending on the PLL discriminator used, that the polarity of the bits may change between words. The last two bits in the second word of each sub-frame (bits 59 and 60 from the sub-frame start) should produce a negative result and can be used to correct inverted polarity conditions.

bitRate: The rate at which the binary symbols are received. The default value is 50 bps.

PLLLocked: An input condition indicating that the PLL is properly tracking a legitimate signal. This flag must be set to **true** before the data bit output events will occur.

isSynchronized: The default initial condition for the demodulator is to integrate over the entire bit interval without regard for the location of the start position. However, better results will be obtained when the position of the bit edge can be located. This flag signals a condition in which the demodulator has been able to make a reasonable estimate of the edge location and the *nextBitTime* property can be used to control the integration span of the PLL in a feedback loop.

nextBitTime: The expected time of the next bit transition condition. This value can be used to control a PLL for situations where the discriminator function is sensitive to bit transitions.

6.2.8.2 Methods

DemodulatorPipelineComponent: The class constructor used to initialize variable instances.

6.2.8.3 Events

IterationComplete: An event that occurs when the specified bit time interval has passed with the *PLLLocked* flag set; signals that a new data bit is ready.

SyncChanged: This event indicates that either a new value for *nextBitTime* has been calculated or that the *isSynchronized* property has changed. The demodulator component needs to verify that the estimate of the location of the bit edges is legitimate by looking for transitions when they are expected to occur.

6.2.9 Signal Controller

An instance of the *SignalController* class maintains an initialized reference to a *SignalBase* abstract class object that serves as the connection point to the receiver's input signal. Any class that either inherits from the *SignalBase* class or provides an implementation for *ISignalSource*, described in the Signal Source Device Driver Interface section, may be used as the signal reference. The object model for the *SignalController* class is provided in Figure 6-18.

The *SignalController* class is derived from the *PipelineComponent* base class and provides the signal sample interconnection and time synchronization to the rest of the processing pipeline. As a result of *SignalController* being a type of *PipelineComponent*, a *SignalController* instance may be joined to a pipeline at any stage and not just at the front-end acting as the primary signal input. Combining a *SignalController* with a signal connected through an interoperability component (see section on Interoperability Support) in the middle of a pipeline would allow, for example, an external physical hardware clock circuit to participate as part of the signal processing chain. Information

regarding the frequency and sampling rate of the underlying *SignalBase* object is made available through access to the *SignalSource* class property.

A *SignalController* instance is expected to run asynchronously on a separate thread until it is sent a *Stop* signal from the containing parent application. Running a *SignalController* in this manner allows the priority of thread to be adjusted according to the demands of the input signal source.

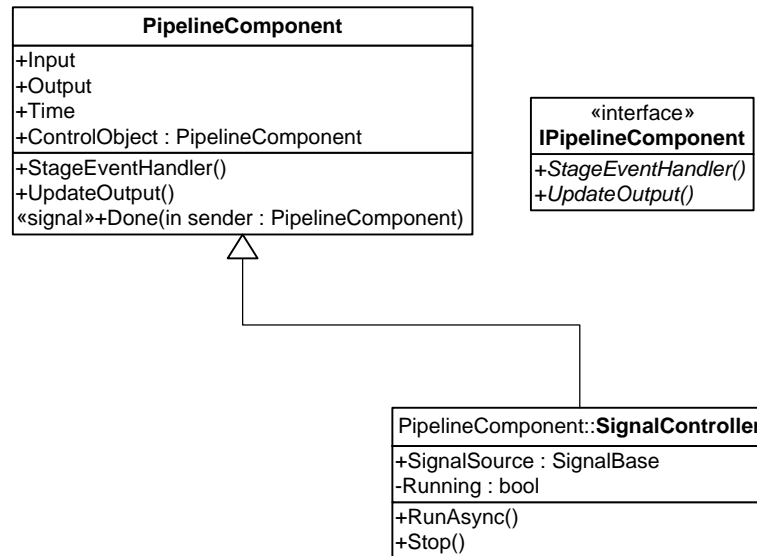


Figure 6-18—SignalController object model

In addition to the properties, methods, and events inherited from the base *PipelineComponent*, the *SignalController* class provides the following extensions.

6.2.9.1 Properties

SignalSource: a reference to a *SignalBase* object (Chapter 8) that is used to generate the input sample data and time values. When a *SignalController* is connected as the primary input stage to a pipeline container, in each sample interval the *Time* property is

incremented by an amount equal to the reciprocal of the sampling rate property defined in the *SignalSource*.

Running: a private property used by the class to determine if it should terminate any threads or operations that are currently executing.

6.2.9.2 Methods

SignalController: The class constructor that is used for initialization.

RunAsync: resets the Input and Time properties, initializes and starts the *SignalSource*. It then executes a loop that repeatedly sets the Output to the *ReadData* method result from the *SignalSource*, raises the *Done* event, and then updates the *Time* property. This method only returns after the *Stop* method is called.

Stop: sets the *Running* property to false, which halts the *RunAsync* thread if it is running.

6.2.9.3 Events

The Done event is inherited from the *PipelineComponent* class. Derived classes may extend the signals of the base-class definition.

6.2.10 PRN Code Generation

The classes that are responsible for the generation of the pseudo-random noise sequences that are necessary to de-spread the received signal implement the *IPRNGenerator* interface. Shown in Figure 6-19 is the UML object model for an inheritance hierarchy of GNSS PRN code generator classes.

index in the table as the code starting point. Time values passed into the *SequenceValue* method are reduced modulo sequence duration to compute the index of the chip location for the time provided.

6.2.10.1 Properties

ID: represents the identifier or starting seed value of the PRN sequence.

SequenceLength: the total number of chips in the PRN sequence.

SequenceDuration: the time interval, or repeat rate, of the sequence; the sequence length divided by the chipping rate.

ChipRate: the frequency, or time-base, of the chip sequence.

ChipDuration: the width in time of each chip in the sequence, equal to the sequence duration divided by the sequence length, or the reciprocal of the chip rate.

6.2.10.2 Methods

SequenceValue: returns -1/+1 value of the sequence for the specified time index. The returned value should be calculated on a *modulo-ChipDuration* basis and should support negative time inputs as indicating a time offset from the end of the sequence.

6.3 Common Types

The common types module includes the various types that have been provided in order to support, leverage, or enhance certain characteristics of the targeted runtime environment. By providing them in this fashion, portability issues are lessened should someone decide to undertake the endeavor of migrating the framework to a different environment.

6.3.1 Complex

The *Complex* type is a structure that provides a Real + Imaginary number system and defines the related operators. The signal processing pipeline classes work with *Input* and *Output* properties that are of type *Complex*.

6.3.2 DFT

The *DFT* class provides a *Transform* method that accepts and returns an array of Complex numbers. The *Transform* method is a brute-force, non-optimized discrete implementation of the Fourier transform.

$$H(f) = \sum_{k=0}^{k < N} \sum_{n=0}^{n < N} h_n e^{-2\pi i f n k / N} \quad 6-15$$

where f is the desired frequency bin and N is the number of samples.

Although slower than the FFT, discussed next, the DFT will operate on sampled data sets of any arbitrary length. It is sometimes faster to perform a DFT transformation on non-power of two data lengths rather than padding the input to the next highest power of two.

6.3.3 FFT

The *FFT* class provides a *Transform* method that accepts and returns an array of Complex numbers. The *Transform* method is a fast-optimized discrete implementation of the Fourier transform based on the Cooley and Tukey Radix-2 Butterfly technique (60). This algorithm requires that the input data be, or be made to be, a power of two in length. Data sets that are shorter should be extended and padded with zeros before calling the *Transform* method.

6.3.4 Filter Classes

The *Filter* classes include moving average, 2nd- and 3rd-order discrete filter implementations. These filter implementations are used primarily in the PLL and DLL integrator functions.

6.3.5 Frequency

Provides a collection of methods for performing correlation, convolution and their respective inverses, along with data-windowing functions, on arrays of *Complex* data. These can be used for signal analysis purposes.

Chapter 7 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 met.

7.1 Interoperability Requirements

To satisfy general reuse expectations and to support specialty libraries and tools from 3rd-parties, either open-source or commercial in nature, the receiver framework provides a generalized method of integrating non-core code through its interoperability service interface. Such code can be locally or remotely executed, or even called via a web service or an equivalent remote procedure call (RPC) mechanism. In order to connect a plug-in

component, that component has to conform to or be made to look like it supports, certain essential interoperability requirements:

- 1) **An external component has to be externally callable.** That is, the component needs a layer to provide some type of formal application programming interface (API) definition. It can be a library module, an application, or a system-level service, but it has to include a means of invoking the functions and receiving returned results. Stand-alone applications that only interact with the outside-world through an event-driven user-interface, make for poor interoperability candidates.
- 2) **External components must have data types that support the necessary marshaling services,** and the types have to have equivalent representations in both environments; i.e., they must be blittable data types. [Data marshaling is the act of moving and initializing data elements from one region of memory to another, typically between processes.] Types that are of different sizes, byte-ordering (endianness), or are ambiguous require special handling and possibly the implementation of a custom marshaler. Ambiguous types have either multiple representations that map to a single type, or they are missing type information, such as the size of an array.
- 3) The external **code must be available to the system at run time.** For library code that has been statically linked, the code will become part of a user application that must be present on the path when the code is called.
- 4) **The code has to be native to the operating system that supports the application,** or accessible through a service that will make it look like native code. Interpreted code, such as Java or MATLAB, requires invoking the

functionality through a runtime interpreter application that must be accessed by a user-provided mechanism.

Although not a critical requirement, ideally, the external code **will not be required by the application to maintain state** between calls. Multiple round-trip calls to library functions that expect the persistence of state between calls will have to externalize the information necessary to exchange and reinitialize the state-related variables from call-to-call. Such a requirement can be difficult to implement efficiently, particularly in a multithreaded or multiuser environment.

7.2 Interoperability Layered Model

The relationship between the interoperability layer and the other framework components can be conceptualized as shown in Figure 7-1. 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.

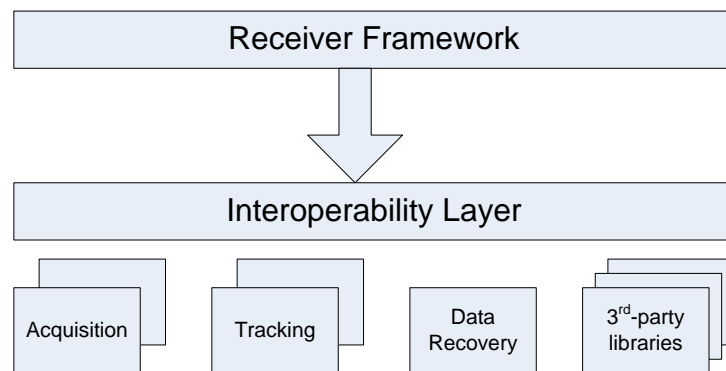


Figure 7-1— Interoperability Layer

Figure 7-2 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.

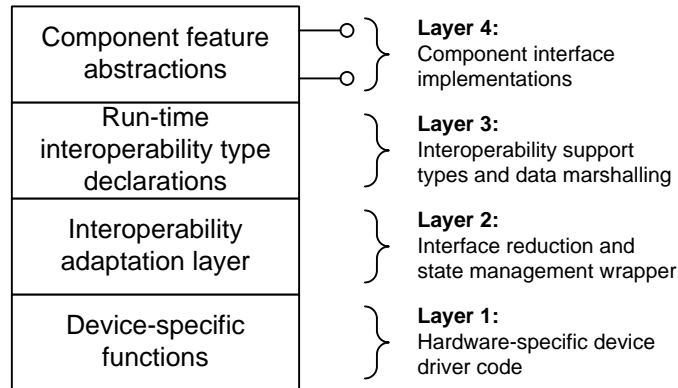


Figure 7-2—Layered Interoperability Model

At the bottom is the Layer-1 hardware-specific, often vendor provided, specialized device driver application or library code, and is usually only required when connecting to physical hardware. The hardware drivers are typically written in C and assembler, and as a result require the support of other C-language constructs in order to invoke their functionality. The exported functions and data structures frequently contain naming conventions and other attributes that tie them directly to either the hardware or the system bus connection they represent. Calling these functions or creating related type instances directly by their name makes the code explicitly hardware dependent and limits application reusability.

Layer-1 consists of libraries, applications, and configuration utilities that expose fundamental low-level operations to the upper layers in a uniform and consistent way. A familiar example would be a driver for a printer device. While perhaps not a potential

hardware component for a receiver, the abstraction that is achieved is equivalent in that any application with a print capability can direct its output to the printer through the device driver without any specific information on how the printer happens to be attached to the system.

Layer-2 is identified as an interoperability adaptation layer, the purpose of which is to hide some of the low-level implementation details required to perform high-level functional tasks. Depending on the complexity of the device operations, it is sometimes necessary to create an additional C-library wrapper to transform and minimize the exported functions and their data types. State encapsulation and persistence management can be implemented at this layer. While it is optional, this layer can be particularly beneficial when the source code for the device is unavailable, or when the device is shared by multiple applications and the interface signature is immutable (non-changeable), thereby making it impossible to be changed for a specific need or requirement.

Layer-3 provides the runtime interoperability type and function declarations, specifies the names and locations of related hardware libraries, and describes the nature and direction of the required data-marshaling services (in, out, in/out.) This layer is optional if support for the device is integrated as a library type of the runtime environment or operating system, or when creating a virtual or simulated device. It is usually necessary when connecting to a physical device through a device driver provider.

Layer-4 is the critical part that provides the functional view of the device to dependent applications. It is implemented through interfaces and base classes that are

used to define general capabilities for specific hardware-centric device descriptions. If a base class implements an interface, the derived class inherits that implementation.

Since the nature and operational characteristics of the interface requirements for any individual device can vary greatly, even within a device family, only general implementation strategies can be discussed for the bottom three layers. The top layer, due to its application-wide pervasiveness warrants the most attention.

The most likely candidate that will be encountered in developing and extending the receiver framework using the interoperability model will be for the front-end signal hardware. Signal sources from different types of hardware front-ends, including simulated and file-based data, may be connected to the receiver framework through the interoperability 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.

7.3 Interoperability Scenarios

Different interoperability situations call for different interoperability support structures in order to satisfy the specific requirements at hand. When an application calls a method or procedure that has been packaged into an external library, the code that is eventually executed requires runtime system resources such as memory and processor time, in the form of a scheduled thread, in order to do the requested work. The required resources may be taken directly from the calling application—the code can be loaded into the application’s address space and the instructions can execute with the active thread. When resources are shared in such a manner, the code is considered to be executing *in-process* since it appears to the system as if it is part of the original application process.

For security and other memory management related reasons, however, it is often not appropriate for an external piece of code to share the same memory and other resources as the application that has invoked the operation. Such circumstances require that memory be allocated by the global system memory manager and the necessary data values must be copied between the two memory spaces in a way that makes the values look like they are sharing the same storage locations.

An executing thread takes on the security context, the identity and credentials, of the process that created it. The active security context can be used to limit the amount of access that an executing thread has within the system. Switching a thread between one process context and another incurs a potentially significant overhead and corresponding performance penalty. Keeping calls within a single process, with or without data marshaling, typically represents the most ideal situation from a performance point of

view. Making calls into a functional library or an application-mode device driver is usually a single process activity.

Calls that invoke operations residing in external libraries or applications can also be made between two processes as an *interprocess* operation, such as calling code in a system service or other application where dedicated resources are doing work. In addition to the aforementioned data marshaling requirements, these calls also necessitate thread synchronization operations and security context changes. While there is an additional overhead to consider, interprocess execution can be invaluable in situations involving trusted subsystem models, accessing dedicated resource pools, or for connection and state management activities. Interprocess operations can also be combined with a network layer redirector to create a distributed execution environment, where code can be executed remotely, such as performing database queries or accessing a hardware resource on an external server. The flexibility of interprocess operations often makes up for their potential performance downside.

7.3.1 Single Process

Figure 7-3 represents the most often encountered configuration for interoperability between legacy or special-purpose library code; straight calls to library functions or application-mode device drivers.

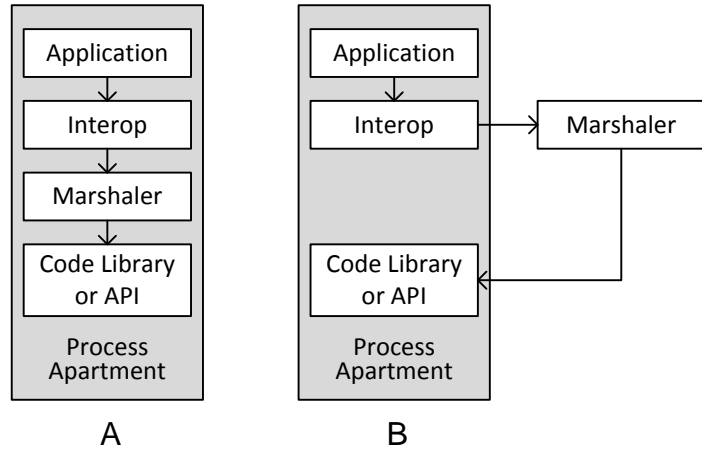


Figure 7-3—Single process interoperability function call: custom marshaler in A, system marshaler in B

The interoperability layer provides the runtime callable wrappers that specify the names and locations of library binaries as well as supplying the mappings between function names and interface signatures (function prototypes.) When library code is loaded and executed in the memory space of the main application process it takes on whatever threading model is active when the call is made. Thread synchronization requirements depend on the application and any shared data structures. Object state persistence is the responsibility of either the application or the interoperability layer implementer.

An in-process custom marshaler is shown in A, while a solution with the system default marshaler is shown in B. System provided marshalers usually run in the system process.

7.3.2 Interprocess

Figure 7-4 shows an example of the interoperation between an out-of-process application or system service and the system framework receiver application. Kernel-

mode device drivers, host applications, or installable services provide an isolated execution environment and security context that require system-level marshaling and synchronization services. The marshaler is required to create and initialize copies of the involved data structures between the two processes.

The threading model of the client (Process 1) must support that of the server, Process 2, since the library code is executed within the Process 2 apartment or boundary. Process 2 could represent a persistent data store and the marshaler would be an Open Database Connectivity (ODBC) or Object Linking and Embedding (OLE) data source provider.

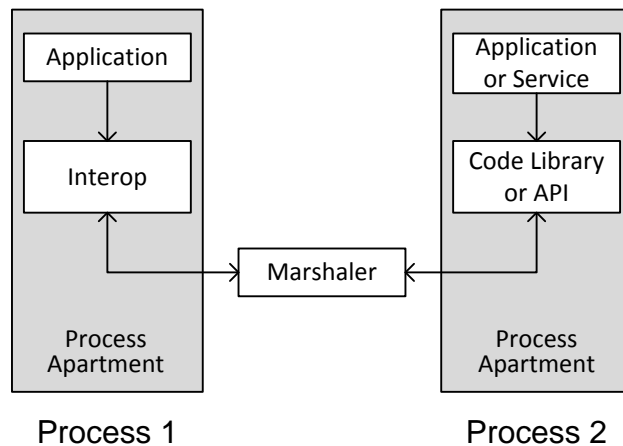


Figure 7-4—Interprocess with common data types and system marshaler

State persistence between calls would have to be supported intrinsically by the design of Process 2, or extrinsically by Process 1 in the implementation of the interoperability layer. Depending on the nature of the services provided by Process 2, state persistence may either be not supported or not required.

7.3.3 Interprocess with Remote Execution

A distributed application solution can be integrated with the system framework through a standard network remote procedure call (RPC), named pipe, or sockets-based approach. Pipes and RPC methods are session-layer network protocol connections, while a socket operates at the transport layer. Remote services that operate over application-layer protocols, such as XML-based Web Services over HTTP, can also be used to extend the interoperability layer to services hosted on networked machines located anywhere in the world. The pipe or socket layer may not necessarily be a native application component and therefore would not require interoperability support from the system framework. However, such support would be needed if the communication components involved were part of a customized connection library application or message exchange protocol.

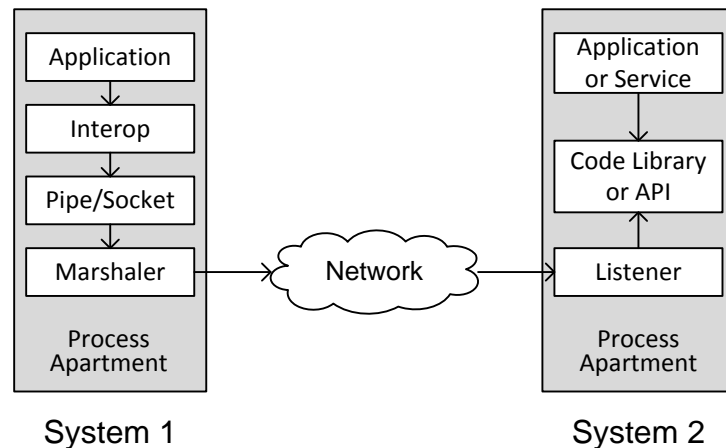


Figure 7-5—Interprocess interoperability between two systems with remote code execution

The connection hierarchy shown in Figure 7-5 is a generic view of the remote execution of code between two network-attached systems. The systems involved may be

of completely different architectures, as long as there is a possible mapping between any data types that are exchanged.

The end-to-end round-trip latency introduced by the network conversation must not adversely affect any of the desired real-time performance characteristics expected of the receiver implementation. While it may be technically possible to use a hardware-based correlator located on the opposite side of the world through a network accessible interoperability layer, the delays in sending, processing, and receiving the results will likely negatively impact the system performance.

Chapter 8 Signal Source Device Driver Interface

The most frequently encountered need for connecting external hardware will be for the signal front-end sampling device. This section provides the details of implementing a signal source using the layered-interoperability model, showing the required adaptations for a specific device; the *SiGe SE4110L-EK3 USB (61) Link-1 (L1)* receiver front-end that has been used for framework testing.

The purpose of the device driver components is to provide the interfaces and abstract base classes required to permit the necessary functional virtualization. The design goal is to support a broad range of hardware-oriented analog-to-digital front-end devices in a manner that is both internally consistent and externally flexible in order to isolate changes and to minimize the effects on unrelated system components.

Internally, the focus is on the common characteristics of these devices. Aspects such as the sampling rate and intermediate frequency are made accessible through device class properties, while specific hardware features, such as bus connection type or individual chipset registers, are kept deliberately opaque. Externally, by acting as a data-stream endpoint, the framework supports a variety of highly dissimilar devices, even those that are file based or entirely simulated. The source for the receiver connection sink can be implemented in the manner most appropriate to the device at hand.

The same principle of abstraction applies to the device state, as well. While it may be necessary for an individual hardware device to traverse several unique states on the path

from initialization to data transfer, these fine-grained states can be represented by aggregation into a smaller number of more coarse-grained system-level states. The messages defined by the supported interface types act as signals that trigger a state transition, such as starting a signal capture or data transfer process.

8.1 Layered Device Driver Approach

The key to achieving system-level hardware-isolation is a layered approach to device-independent abstraction. A device driver model based on the interoperability model layers of abstraction is shown in Figure 8-1. The model is comprised of four layers, but depending on the characteristics of the implemented device, it will not always be necessary to provide all the layers; once again, only the top level, Layer-4, is mandatory.

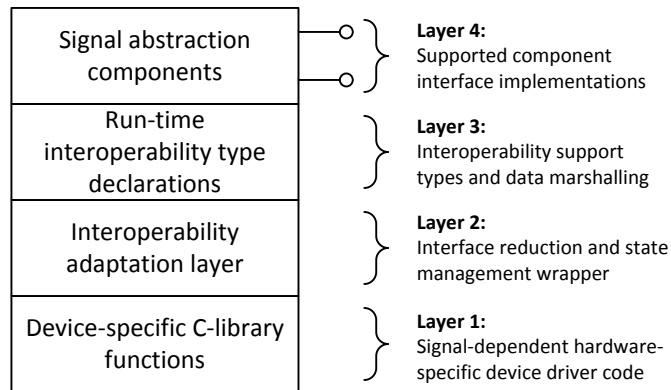


Figure 8-1—Layered Device Driver Model

Layer-1 implementation consists of the OpenSource LibUSB USB (62) device driver code library for Win32 applications. LibUSB allows user applications to access any USB device on a Windows system in a generic way without having to write custom kernel-mode device drivers. The exported device library functions are called by including the `usb.h` C header file and linking to the `LibUSB.lib` library.

The code from the library for the device *start* function is shown in Figure 8-2. It is not obvious from the code alone what is happening at the physical device connection since the goal of this function is to provide a level of abstraction between the device details and the higher level functionality.

```
bool fusb_ephandle_win32::start () {
    if (d_started)
        return true;    // already running

    d_started = true;

    d_curr = d_nblocks-1;
    d_outstanding_write = 0;
    d_input_leftover =0;
    d_output_short = 0;

    if (d_input_p){    // fire off all the reads
        int i;

        for (i=0; i<d_nblocks; i++) {
            usb_submit_async(d_context[i], (char * ) d_buffer+i*d_block_size,
                d_block_size);
        }
    }

    return true;
}
```

Figure 8-2—LibUSB library function for starting a USB device.

Layer-2 provides the connection to the underlying device library by encapsulating and selectively exporting only those higher-level functions that are required to be accessed. The exported functions for the USB signal device are declared in a C header file, shown partially in Figure 8-3.

```

GN3LIB_API void InitializeTheDevice(void);
GN3LIB_API void ReleaseTheDevice(void);
GN3LIB_API void StartDataClock(void);
GN3LIB_API void StopDataClock(void);
GN3LIB_API void ReadDeviceData(LPBYTE buffer, int size);
GN3LIB_API DWORD GetDeviceStatus(void);
GN3LIB_API void ReadGnssRFData(HANDLE hDevice, LPBYTE buffer, int size);

```

Figure 8-3—Exported USB functions from Layer-2

The implementation of one of the functions, *InitializeTheDevice*, is shown in Figure 8-4. This function is responsible for creating an instance of a pointer or Win32 handle to the correct USB device instance, and persisting its initialized state information.

Collectively, these functions are exported from an application dynamic-link library.

```

GN3LIB_API void InitializeTheDevice(void) {
    if (fx2 == NULL) {
        fx2 = usb_fx2_find(&num_str, vid_str, pid_str, debug);

        if (!fx2) {
            throw ERROR_NOT_SUPPORTED;
        }

        if (usb_fx2_configure(fx2, &fx2c) != 0) {
            throw ERROR_NOT_SUPPORTED;
        }
    }

    deviceinitialized = true;
}

```

Figure 8-4—Implementation of the Layer-2 USB function for device initialization

Layer-3 would be responsible for ensuring type compatibility between the unmanaged C runtime and the managed .NET Framework common language runtime. For this particular device, the types used are directly compatible between the two environments, so type conversions are not necessary.

Layer-4 provides the final connection between the device library and the receiver framework. The functions declared in the runtime callable wrapper shown in Figure 8-5 meet the requirements of the interface specification for the USBSignalSource type, as shown in the class diagram of Figure 8-7, by utilizing the .NET platform-invoke (P/Invoke) mechanisms.

```
public class GN3S {  
  
    [DllImport("GN3Lib", EntryPoint = "?InitializeTheDevice@@YAXXZ")]  
    public static extern void InitializeTheDevice();  
  
    [DllImport("GN3Lib", EntryPoint = "?ReleaseTheDevice@@YAXXZ")]  
    public static extern void ReleaseTheDevice();  
  
    [DllImport("GN3Lib", EntryPoint = "?StartDataClock@@YAXXZ")]  
    public static extern void StartDataClock();  
  
    [DllImport("GN3Lib", EntryPoint = "?StopDataClock@@YAXXZ")]  
    public static extern void StopDataClock();  
  
    [DllImport("GN3Lib", EntryPoint = "?ReadDeviceData@@YAXPAEH@Z")]  
    public static extern void ReadDeviceData([Out] Byte[] buffer,  
                                             [In] int size);  
  
    [DllImport("GN3Lib", EntryPoint = "?GetDeviceStatus@@YAKXZ")]  
    public static extern uint GetDeviceStatus();  
  
}
```

Figure 8-5—Layer-4 device wrapper declaration for the GN3S device driver

The implementation of the initialization function is shown in Figure 8-6. The wrapper functionality can now be called by simply including the assembly into any development project utilizing a .NET-compatible language. All of the exported types and functionality are visible to consumers of the assembly without the need to include any of the C language-specific header or library files. However, it is necessary to have the USB device driver (LibUSB) properly installed, the USB front-end connected, and the Layer-2 library accessible to the calling application when the code is run. See Appendix A for additional details on P/Invoke and 3rd-party library interoperability requirements.

```

/// <summary>
/// Initializes the hardware device and readies it for data capture
/// </summary>
public override void Initialize() {
    GN3S.InitializeTheDevice();
    GN3S.StopDataClock();
    //Signal the device Reset event:
    Reset(this, new EventArgs());
}

```

Figure 8-6—Layer-4 wrapper code implementation for the GN3S device driver initialization sequence

In general, there are two forms of implementation patterns that achieve the level of isolation and abstraction that is expected of Layer-1. The first, shown in Figure 8-7, uses an abstract base class that provides interface amalgamation, and default method definitions when default component behavior can be defined without loss of generality. Marking the base class as abstract prevents it from being created directly and constrains its use for derived types only. Each derived receiver class follows the *is a* idiom for object inheritance; for example, a *USBSignalSource* is a *SignalBase* type. Only one class definition is required for use in host applications, where the object characteristics are determined at run time through late binding and polymorphism.

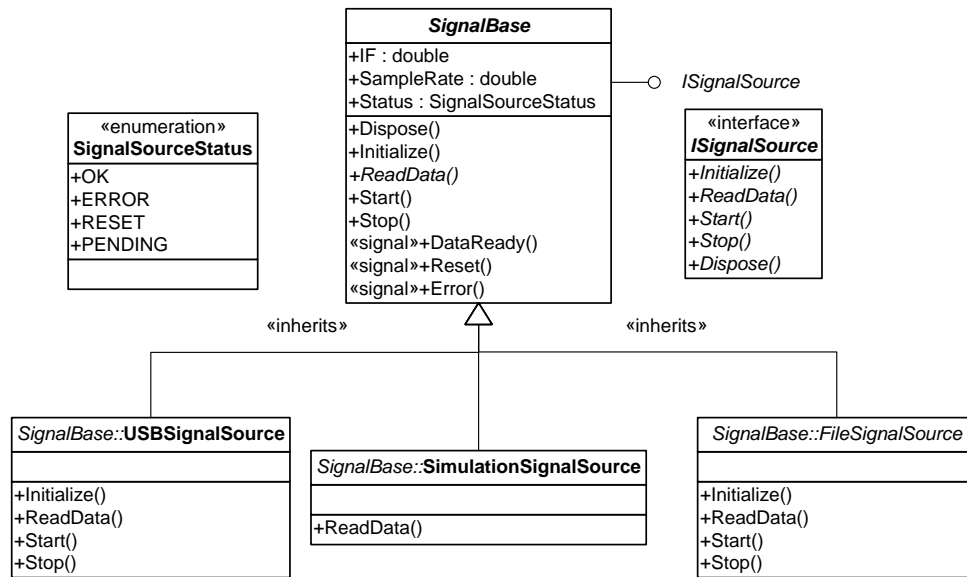


Figure 8-7—Abstract signal base class implementation UML static structure diagram

The interface declaration, *ISignalSource*, describes the features that would be expected of any generic signal class. Objects that implement this interface must provide a definition of all methods and operations that users of the interface expect—the interface represents a contract between the component and its clients or users.

To further refine the desired functional features, an abstract signal base class, *SignalBase*, is declared that implements the *ISignalSource* interface. Since *SignalBase* is abstract, it can only be used as a contract, of sorts, in the definition of derived classes. A class derived from *SignalBase* will inherit the interfaces and default method definitions that it supports, but must also provide any bodies for methods not already present in the base class.

Variable declarators of the base type are assigned to references of derived or concrete type specifiers through an instance identifier. Messages sent via method calls made on the

base-class type are resolved at run time to the corresponding derived-class implementations.

Object creation follows the usual instantiation pattern:

```
SignalBase inputsignal = new USBSignalSource ();
```

All dependent code that relies on the general behaviors of a *SignalBase* instance can be obtained by invoking the same methods on the signal reference; in the code above, the derived *USBSignalSource* instance would provide the device specific functionality.

```
inputsignal.Initialize ();
```

The previous code excerpt demonstrates the explicit creation of a derived-class instance by name. Creating object references in this manner requires the identity of the type to be known at the time that the application is compiled. A **class factory** is a generalized mechanism that creates object instances from a codified ID or some other form of unique class identifier. If complete type isolation is necessary, a factory could be implemented either as a helper class or an extension method that would defer the final derived-type resolution until run time execution.

A second form of signal source implementation can be crafted to achieve the goal of device independence. Figure 8-8 shows a *USBSignalSource* class that directly implements the *ISignalSource* interface without inheriting from the *SignalBase* abstract class. This class would need to implement all of the methods described in this interface since interfaces provide no default implementations. Any new, non-USB, types of signal source defined in a similar manner would not be able to take advantage of methods

implemented in the *USBSignalSource* class, so the pattern would have to be repeated for each device type.

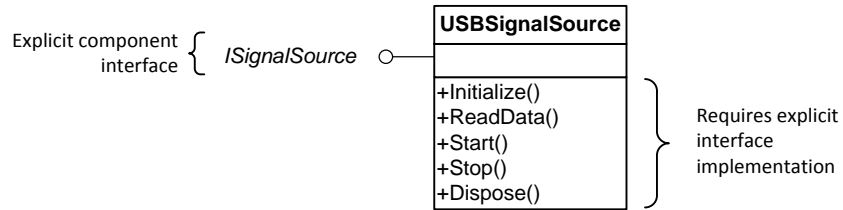


Figure 8-8—Explicit interface implementation

By implementing interfaces in this way, the *USBSignalSource* class *looks like* a generic signal source but inherits no default behaviors or method implementations. However, other USB signal input device types with shared or similar characteristics could be derived from the non-abstract *USBSignalSource* class.

The following code declares an identifier of the interface type *ISignalSource* and then creates an instance of a *USBSignalSource* with the *new* keyword:

```
ISignalSource inputSignal;  
USBSignalSource input = new USBSignalSource();
```

The declared interface, receiver, is then assigned to the *USBSignalSource* instance after casting it to the interface type:

```
inputSignal = input as ISignalSource;
```

Methods are then called directly on the interface to achieve the required polymorphic behavior:

```
inputSignal.Initialize();
```

8.1.1 Signal Source Device Interface

ISignalSource is the standard signal interface type that represents the most abstract form of the operations any signal source must support in order to be compatible with the system as presently defined. Future changes to this interface description will affect all classes and their derived subclasses that support it. To that end, it is necessary to keep the basic interface as uncomplicated as possible to avoid creating a burden for future component developers. While it may be tempting to add extra properties and methods, only those that are absolutely necessary, and not just nice to have, should be incorporated into the common signal interface.

8.1.1.1 Properties

IF: provides the value of the intermediate frequency, or center frequency, of the down-converted signal source. This value is used in various parts of the receiver in order to set the reference frequency of the carrier mixer stages and to calculate the Doppler shifts of the incoming signal.

SampleRate: indicates the sampling frequency of the incoming data stream. In the case of an external hardware signal sampling device, this value is determined by the device characteristics. For a simulation-based signal, the sampling rate must be specified to meet the Nyquist criteria—twice the highest frequency component of the signal.

Status: provides an indicator for the operational status of the device. The value is an enumerated system type, discussed later.

8.1.1.2 Methods

The methods of the *SignalSource* classes setup, activate, deactivate, and tear down the connections and resources for signal sampling. The related state transitions involved are described in the device *State Model* section.

Initialize: acquires the necessary system resources, such as memory and device handles, and configures any internal data structures used by the device. This method must be called by consumers of the signal source to ensure the device is properly configured, but the class implementation may not be required. A call to initialize should leave the device in the *Suspended* state.

Dispose: stops the device if it is currently active and releases any resources previously acquired through initialize.

ReadData: while the signal source is in the *Data Transferring* state, transfers an element of data from the signal source. This method may be implemented to support block transfers of data involving more than one sample, or can support a time-indexed value where the current time is passed as an input parameter.

Start: activates the device for data sampling. The device is put into the *Data Transferring* state.

Stop: suspends device data sampling. The device is put into the *Suspended* state.

8.1.1.3 Events

In the class UML diagram, <<signals>> indicate the published events that may be used to notify external objects of internal conditions or state transitions that have occurred.

DataReady: raised when a sample unit of signal is available to be read. Data readers may either block by calling the *ReadData* method, or be asynchronously notified that data is ready by responding to this event.

Reset: raised when the device has been reset, allowing external system components to reinitialize the device and reestablish their own internal state variables.

Error: occurs when the device has encountered an unrecoverable error that requires external intervention in order to correct.

8.1.1.4 Status Enumeration

SignalSourceStatus enumerated type provides an indication through the device *Status* property of the operational state of the signal source. Valid values are: OK, ERROR, RESET, PENDING.

8.1.2 Signal Source Base Class

SignalBase implements *ISignalSource* and provides default implementations of *Initialize*, *Start*, and *Stop*. Not all signal sources will require bodies for these methods and providing a default implementation eases the work necessary to derive a new concrete type from the base class. Default properties for *IF* and *SampleRate* properties cannot be provided as they are always dependent on the signal source.

8.1.3 Signal Source Derived Classes

Derived implementation classes for a USB-attached signal sampler, a file-based signal data source, and a simulation signal model are provided as references. Further details are provided in the Reference Implementation, Chapter 10.

8.2 Device Interface State Models

Just as important as the ability to abstract the functionality of the device, the state transitions must also be reduced to a common set that the system can manage and maintain. The signal device system-state model identifies the representative states and the conditions that must be met for a state to occur. It is necessary that hardware devices be mapped from one state space to the other; from an internal to an external representation.

8.2.1 Signal Source System State Model

The signal source state model that is defined by the system is represented as a UML Statechart in Figure 8-9.

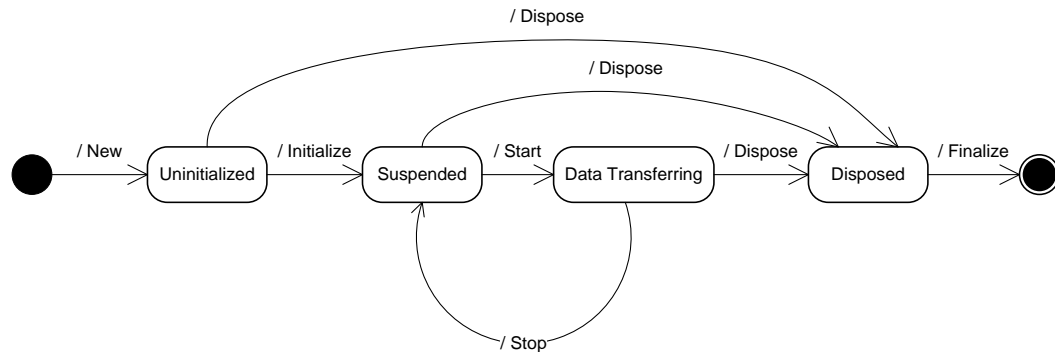


Figure 8-9—Generic signal source UML Statechart model

There are four states identified in the model:

- **Uninitialized:** The representing object has been created from the global resource pool and any required memory has been allocated. Access mechanisms for the hardware have not been established, device or file handles are not yet created.
- **Suspended:** Hardware and operating system primitives, such as device handles, have been initialized and are ready to go, awaiting a signal to start reading data. The process of initialization has reset base stream positions to their beginning. In the case of a Stop message, the current stream position, for byte-stream oriented devices, is maintained.
- **Data Transferring:** Data is actively being collected or generated and is either buffered internally or moved to the system. This is an active state where work is being performed by the device.
- **Disposed:** The resources previously acquired by the object and underlying hardware have been deactivated and released, ready to return to the system resource pool. The objects cannot be reclaimed or reinitialized from this terminal state. Reactivation implies recreation of all dependent objects.

8.2.2 State Model for USB Signal Source

Figure 8-10 represents an internal state model view of a USB-attached digitizing device. This representation does not make visible any of the details of the USB interface protocol specification and has been simplified to show only a high-level logical characterization of the device operation. There is a greater number of states and state transitions identified in this diagram than the previously discussed system-state model

supports. It is necessary, therefore, that a mapping and minimization effort first be performed in order to coerce the model's shape into something the system supports.

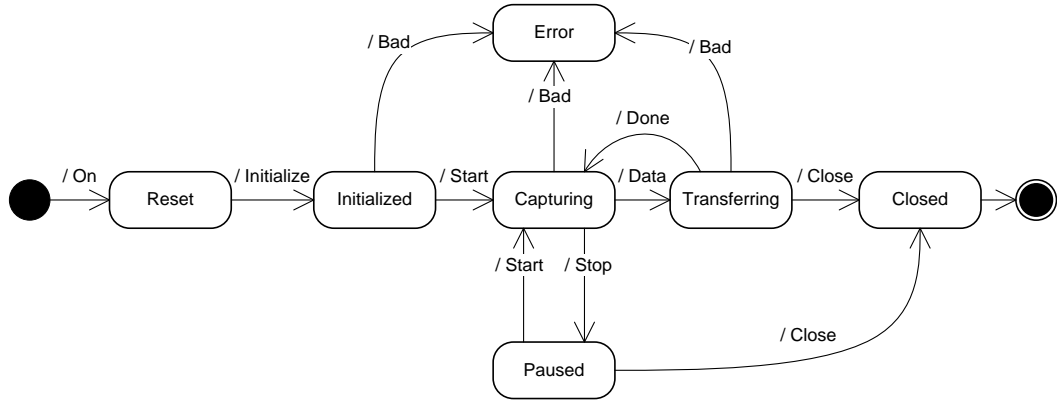


Figure 8-10—USB signal source internal state model

One such possible system mapping is demonstrated in Figure 8-11. The labeled gray outlines indicate the aggregate state boundaries. The exact boundaries appropriate for aggregation and the manner in which state information is maintained or persisted are largely dependent on the specific implementation and device operational requirements.

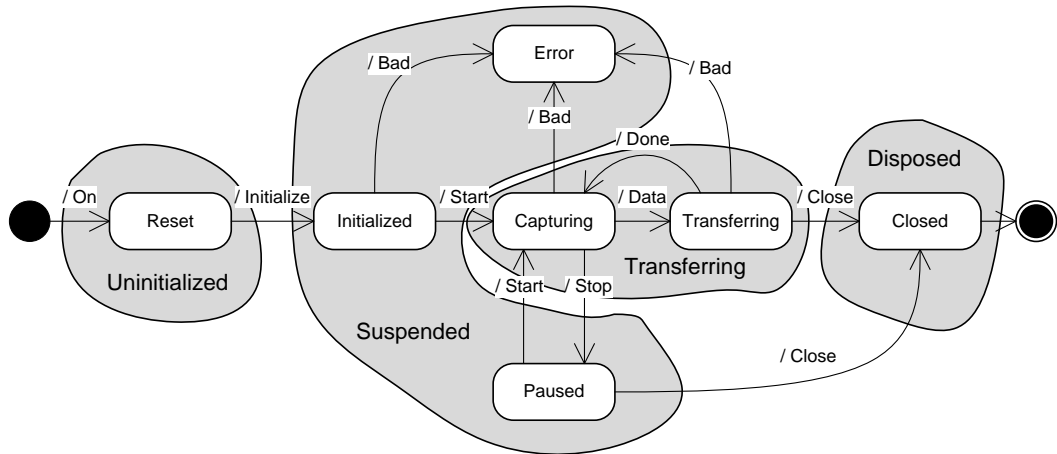


Figure 8-11—USB device state model as mapped into the system state model

The state reduction mechanisms can be structured as code in any layer of the device interface model, but should be done so in the most appropriate manner for the individual device. As defined, the system state model may hide device-specific capabilities, making them invisible to client applications; such is the nature of hierarchical entity abstraction.

Chapter 9 Acquisition and Tracking

The acquisition components are required to process the input signal source and determine which transmitters are visible and to make an accurate initial measurement of the code delay and Doppler values. The output of acquisition is a collection of tracking objects for each signal transmitter to be tracked.

The event-driven pipeline model, discussed in the Pipeline Processing Model section, is used for the signal processing activities necessary for signal acquisition and tracking. Each acquisition and tracking component is derived from a common abstract base class that implements the *IPipelineComponent* interface. Instances of these components are then organized into a pipeline by setting up the event sources and their handlers appropriately as part of a pipeline container class. Where necessary, the *ControlObject* property of the classes is set to enable any feed-forward or feedback linkages between the various pipeline stages. Each instance of the pipeline container performs the signal detection and processing needed to track a single transmitter. Multiple transmitters are tracked by creating multiple instances of the pipeline container and connecting them to the same *SignalController* stage *Done* event.

9.1 Acquisition

There are different methods available for acquisition; the parallel approach of circular correlation is implemented in the Receiver Development Framework and explained in detail here.

Typical software-based receivers for GPS applications (5) (6) (7) perform acquisition by analyzing a block of sampled data along a 2-dimensional plane with code delay running along one axis and Doppler shift along the other. This analysis is repeated for each satellite signal the receiver is attempting to acquire. The Doppler axis divides the expected intermediate frequency $\pm 5\text{-}10$ kHz shift range into bins 500 Hz to 1 kHz apart, while the code axis is determined by the length of the code—1023 chips, 1 ms, in the case of the GPS C/A code. The approach is shown in Figure 9-1.

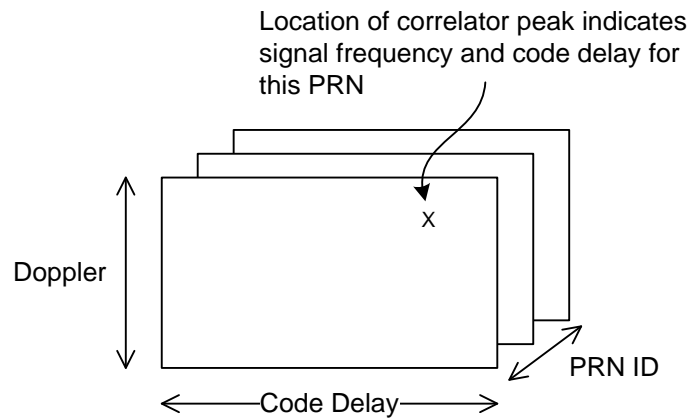


Figure 9-1—Software-based signal determination

Each sample in the data block is multiplied by $\cos(\omega t)$ and $\sin(\omega t)$, where $\omega = 2\pi(IF + f_D)$; IF is the intermediate frequency of the signal source, and f_D is the Doppler frequency value. The resulting real and imaginary (in-phase and quadrature) values are cross-correlated with a generated copy of the PRN code for delays ranging from 0 to the sequence length. The discrete form of the correlation function between two sample sets of length N , $x(n)$ and $y(n)$ can be written as:

$$r(n) = \sum_{i=0}^{N-1} x(i)y(n+i) \quad 9-1$$

If $x(n)$ and $y(n)$ are both real, the transform-pair relationship can be used to find the correlation (Appendix D) in the frequency domain as:

$$|R(k)| = |X(k)Y^*(k)| \quad 9-2$$

The complex conjugate of the frequency-transformed PRN sequence is multiplied point-by-point with the transform of the signal data. The result is then inverse-transformed and the resulting vector is scanned to locate the index of the largest peak above the noise floor. If located, the index of the peak can be used along with the signal sampling frequency, f_s , to calculate the time delay of the start of the sequence.

$$\text{code time offset} = \frac{\text{peak index}}{f_s} \quad 9-3$$

Estimating the amount of noise in the signal in order to determine the required peak size required for detection can be difficult to do reliably. If the noise floor estimate is too low, the probability of a false detection is increased; if set too high, signals may be overlooked. The approach taken in the code of (5) is to compare the magnitude ratio of the largest peak to the next larger peak, and declaring a detection only if the result is greater than an acquisition threshold parameter (default = 2.5). However, if it happens that the sample contains a navigation data bit transition, the effect is to create double peaks in the output of the correlator, which can reduce the effectiveness of this detection strategy. The process is usually run twice on consecutive data sets and the results averaged before evaluating the detection metric.

The magnitude of the peak is then stored in the table at position $(f_D, delay)$. The calculations are repeated until all $f_D \times delay$ table locations are filled. Once completed, the table is then scanned to find the coarse Doppler frequency and code delay that resulted in the best, or highest, correlation output. These values are for one PRN or satellite ID; the entire process must be repeated for every satellite that is expected to be visible.

Once the coarse Doppler and code delay are found, the original sample data is multiplied by the local code of the correct delay. Since the sample duration is short compared to the data bit time interval, the result should contain the carrier only. Multiple blocks of samples, up to half the length of a data bit to help minimize the possibility of the presence of a bit transition, are then cascaded together and processed to find the fine carrier frequency and phase values. The phase angle, θ_m , for a data set at time m can be found from the highest frequency component in the DFT result (6).

$$\theta_m(k) = \tan^{-1} \left(\frac{Im(X_m(k))}{Re(X_m(k))} \right) \quad 9-4$$

Repeating the calculation to find the phase angle from data set n , θ_n , taken a short time later can be used to find the fine frequency (6).

$$f = \frac{\theta_n(k) - \theta_m(k)}{2\pi(n - m)} \quad 9-5$$

Once the code delay and the carrier frequency and phase are known, an instance of the tracking loop components is initialized and started with these values as input parameters.

9.2 Tracking

The essential elements are the delay and phase-lock loops for tracking code and carrier phases. These are feedback control systems where the plant under control is the frequency of a numerically controlled oscillator. Without the ability to track the signal for the time required, it would not be possible to receive the navigation message.

The usual approach is to model the control loops as continuous functions of time using s-domain representations and assuming a conversion to the sampled z-domain exists. When demonstrated, the transform from the s-domain is done with Tustin's bilinear transform with,

$$s = \frac{2(z-1)}{T_s(z+1)} \quad 9-6$$

where T_s is the sampling time interval = $1/f_s$.

While other transforms exist, this one maps all stable s-domain systems into stable z-domain (the unit disc).

A potential drawback of this approach is that it assumes a very high sampling rate for the system and fails to account for the effect the sampling delay has on system stability. The sampling rate is not necessarily the signal sampling rate of the front-end hardware, but for the purposes of the control-loop operation, it is the pre-detection integration time of the non-coherent and so-called integrate and dump integrators. As a result, the models derived are less than optimum, provide no guidance or insight on parameter selection, and are potentially unstable.

Chapter 10 Reference Implementation Results

The component models that have been developed have been extensively tested using a combination of sampled data from live satellites and simulated signals. The processing results from one sample file to another are consistent, so only two sets of sampled data outputs are presented. These are the from signal data files that were captured on June 5 and 10, 2009, at Fredericton, NB.

The front-end that was used for testing is less than ideal for a real time applications, and it was never intended to be used for this purpose. However, the development of the receiver framework was already well into testing before the shortcomings of the device were fully realized. At that time, a critical design goal of the receiver framework was to allow for hardware independence and spending additional effort adapting to new hardware was not germane to the framework fundamentals. A great deal of time went into trying to find ways to overcome the limitations of the hardware, but the device was never designed to work in real time applications.

As it turns out, PLLs are a challenge to implement well and efficiently in software. Testing has revealed some interesting dynamics related to the implementation in the receiver. There is still lots of work to be done in this area, and understanding all of it will involve reading and writing many more papers.

Real time performance is not achieved with the front-end as tested, however a simulated signal using I/Q sampling, a 4-bit signed representation and byte-packing was used to demonstrate the real time capabilities of the application framework.

As a demonstration and further proof of the interoperability features of the framework a PLL component from the open-source GNU Radio project has been integrated. Also, adapting the front-end to use a simulated signal source with different characteristics from the other device also represents the equivalent of replacing the hardware entirely, further supporting the design flexibility of the framework.

10.1 Pipeline Testing Configuration

Evaluation of the receiver framework has been conducted with the post-detection tracking configuration shown in Figure 10-1. Using an *SiGe SE4110L-EK3 USB (61)* 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 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.

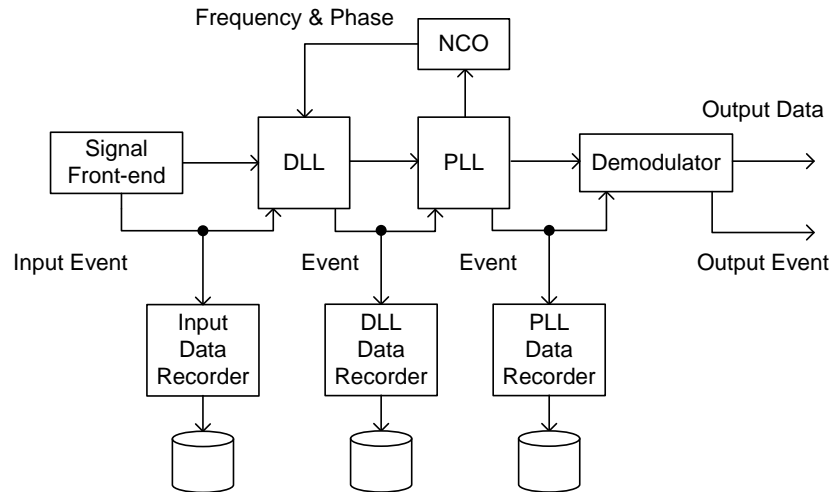


Figure 10-1—Tracking pipeline configuration used for testing

The DLL module (Section 6.2.6) 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 signal. The code-removed output from the DLL is passed to a PLL that tracks changes in carrier phase using an arctangent discriminator function, discussed in Section 6.2.5, 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. For testing, the outputs from the DLL and PLL components are also fed to data recorder objects, in a manner similar to the input signal, for offline graphical analysis.

Referring to the layered interoperability model shown in Figure 7-2 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.

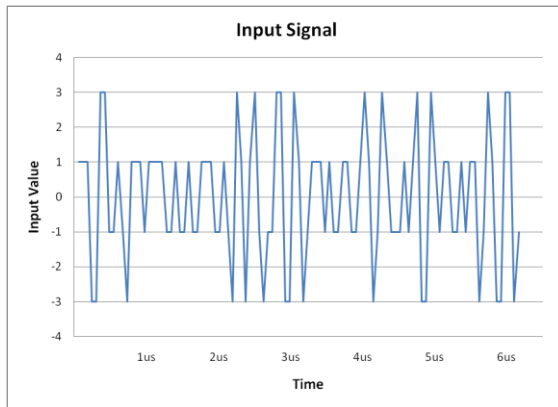
Persisting and reporting on the status of the device, such as current error conditions, and combining the multiple steps required to detect, enable, and activate the hardware into single functional commands is handled in Layer-2. In order to start the sampling process, the device needs to be either reset 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 *ISignalSource* interface methods *Start()*, *Stop()*, *Initialize()*, 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 marshaler, 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 runtime 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 10-1. 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 timestamped copy of the data it requires

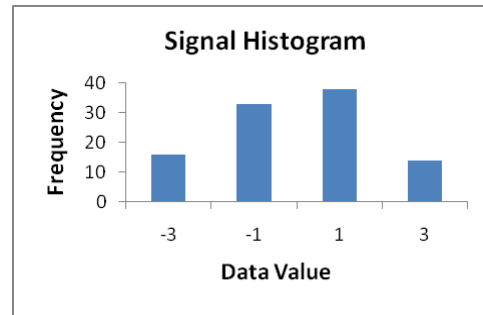
to complete its specified task. Information regarding the incoming sample rate, IF, data bit rate, PRN code delay, and carrier frequency and phase are properties of their representative component classes and are made visible to the receiver pipeline container.

In order to produce the repeatable set of results presented here, the input signal from the front-end was initially recorded to two data files (June 5, 2009 at 13:16 UTC and June 10, 2009 at 15:52 UTC, located at UNB Fredericton, 45.9499N 66.6425W) which were then used as input sources for subsequent analysis. The front-end hardware utilized was originally designed for data capturing in a post-processing application and is not entirely adequate for real-time signal processing. Due to a limitation in the device's embedded software, it suffers from an inability 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 $\approx 4x$ the intermediate frequency, the minimum phase difference between samples of $\pi/2$ 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. The post-processing software intended for use with this device (5) avoids this problem by tracking the signal on a longer time base, integrating the phase error over some interval. However, the longer the integration interval, the more accurate the frequency estimate needs to be, due to a narrowing of the *sinc* function, requiring more signal processing time.

An excerpt of one of the input signals is shown in Figure 10-2 (a). The signal exhibits the expected noiselike appearance, bounded by the -3 to +3 input voltage range.



(a)



(b)

Figure 10-2—Time-domain view of input signal source (a) and input signal histogram (b)

The distribution of the input values covering the same time period is shown in the signal histogram of Figure 10-2 (b). 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.

The frequency domain view of a signal is provided in Figure 10-3, which was obtained by performing ensemble averaging on ten consecutive 4096-length Fast Fourier Transforms (FFTs) of the signal data. The resulting periodogram shows the input signal energy spread over about a 4 MHz bandwidth that is roughly centered on the ≈ 4 MHz IF.

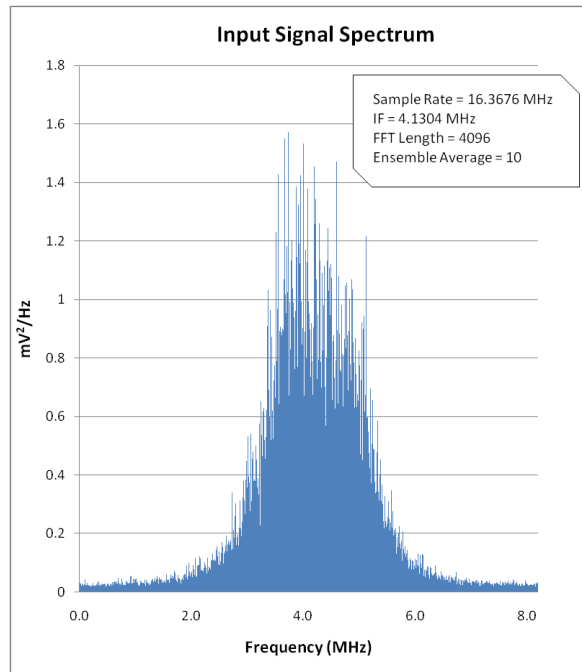


Figure 10-3—Frequency-domain view of input signal

Using the circular correlation method as described previously in section 9.1, the presence of PRNs #18, #21, #22, and #26 were detected in the signal recorded on June 5, 2009 and the initial code phase (time delay) was found for each satellite. The location of the peak for PRN #18 shown in Figure 10-4 corresponds to a code delay of about 917 μ s.

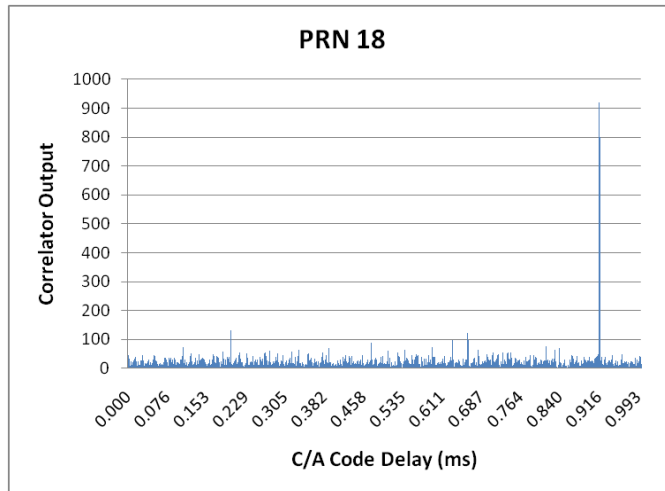


Figure 10-4—Correlation peak for PRN#18 detection

The initial carrier-frequency estimate (IF + Doppler) was found by removing the C/A code from ten milliseconds of sampled data and then frequency transforming the result, looking for a spectral peak. Figure 10-5 reveals a peak for PRN #18 of 4.1304 MHz + 2584.0 Hz.

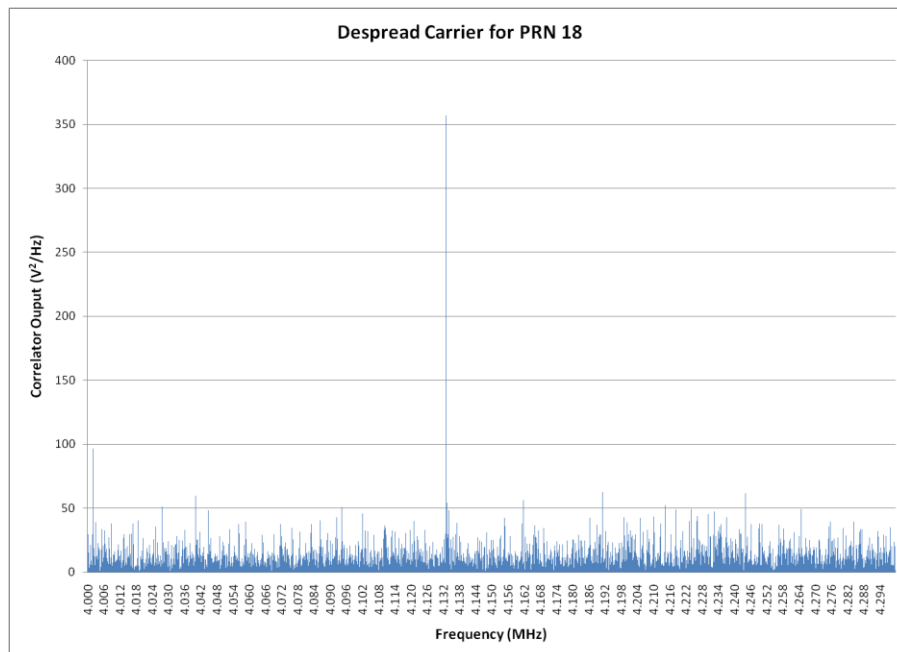


Figure 10-5—Frequency-domain view of recovered carrier for PRN #18

On June 5, 2009 at 10:16 ADT (13:16 UTC), PRN18 had a radial velocity with respect to Fredericton, NB of -409 m/s, which would give it a theoretical Doppler shift of

$$\begin{aligned}df &= -\frac{dv}{c} * f && \mathbf{10-1} \\&= -\frac{-409}{299\,792\,458} \times 1575.42 \times 10^6 \\&= 2149 \text{ Hz}\end{aligned}$$

The difference between the measured (2584 Hz) and calculated (2149 Hz) values can most likely be caused by an inaccuracy in the sampling frequency of the front-end hardware. Since the sampling rate specification is approximately four times the intermediate frequency, an error of only 100 Hz in sampling frequency would cause a difference in IF of 400 Hz. The front-end is based on the SiGe 4110L chipset, but the sampling rate accuracy is determined by the nature of the reference clock circuitry provided in the design. These details and variances are not indicated in the product datasheets, so it is difficult to determine if the calculated errors would be considered within specification.

Additional errors in the Doppler calculation come from numerical precision restrictions resulting from the length of the frequency FFT transform analysis. All of the signal energy from a narrow band of frequencies is placed into a single *bin* the width of which is determined by the size of the FFT. It is not possible to distinguish one frequency from another unless their difference is greater than the minimum bin width. This error source is discussed in greater detail in section 10.2, next.

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 10-6, 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.

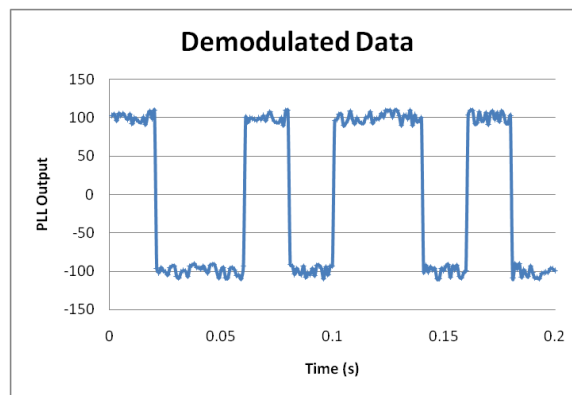


Figure 10-6—Navigation data signal from PLL output

At 50 bps and using the mapping of $\{-1, +1\} \rightarrow \{1, 0\}$ the signal of Figure 10-6 corresponds to the output of ten data bits $\{0110100101\}$. The data output is a raw stream of bits that would have to be aligned to the preamble character (0x8B) at the start of each of the sub-frames in the navigation message in order begin to extract the satellite ephemeris parameters (future work).

The results presented graphically above represent the output of a single instance of the satellite signal tracking loop from the file sampled on June 5, 2009. Multiple loop instances are created and initialized to track multiple signals simultaneously. However

the results obtained thus far using the SiGe front-end hardware are not produced in real time, as desired.

The data recording objects have a tendency to increase the system workload by creating high priority operating system threads that perform the necessary file output operations, which limits the available processor cycles for tracking signals—these recorders can be removed when they are not required for testing. Additionally, the PLL as implemented uses the most computationally intensive discriminator function in order to avoid the need for data bit timing synchronization. More work is required to identify and implement a better PLL software processing model.

The results obtained from processing the signals recorded on June 5, 2009 are summarized in Table 10-1. The average error between the theoretical Doppler and the measured value is about 390 Hz higher than predicted, which can potentially be caused by the previously mentioned clock frequency error, however the standard deviation of the errors is 102 Hz. The samples are gathered over a relatively short timeframe, and one would not expect such a large amount of variability in the clock uncertainty.

| PRN # | Code Delay (ms) | Doppler (Hz) | Radial Velocity (m/s) | Theoretical Doppler (Hz) | Error (Hz) |
|--------------|------------------------|---------------------|------------------------------|---------------------------------|-------------------|
| 18 | 0.917 | 2584 | -409 | 2149 | 434 |
| 21 | 0.094 | 664 | -74 | 389 | 275 |
| 22 | 0.043 | 3263 | -525 | 2759 | 504 |
| 26 | 0.428 | 1842 | -287 | 1508 | 334 |

Table 10-1—Visible satellites extracted from file captured on June 5, 2009 at 13:16 UTC

Typically, sampling clock circuits are off by some measurable amount and that amount drifts slowly with time and temperature. The frequency resolution of the software detection method is about 65 Hz; the standard deviation would be expected to fall within a range of \pm that amount.

| PRN # | Code Delay (ms) | Doppler (Hz) | Radial Velocity (m/s) | Theoretical Doppler (Hz) | Error (Hz) |
|-------|-----------------|--------------|-----------------------|--------------------------|------------|
| 9 | 0.940 | -2427 | 486 | -2554 | 127 |
| 12 | 0.514 | 2373 | -361 | 1897 | 476 |
| 14 | 0.907 | 2038 | -299 | 1571 | 467 |
| 18 | 0.009 | -1607 | 349 | -1834 | 227 |
| 22 | 0.171 | 477 | -12 | 63 | 414 |
| 26 | 0.566 | -2520 | 496 | -2606 | 86 |
| 30 | 0.509 | 3747 | -638 | 3352 | 394 |

Table 10-2—Visible satellites extracted from file captured on June 10, 2009 at 15:52 UTC

The results of processing the data captured on June 10, 2009 are summarized in Table 10-2, yielding an average Doppler error of 313 Hz and a standard deviation of the errors of 163 Hz. Again, these values are higher than anticipated.

10.2 Real-time Performance Evaluation

Real-time performance testing has been conducted with files captured from the SiGE *SE4110L-EK3 USB* signal front-end and a pipeline configuration without the data recording objects of Figure 10-1 attached. For the 16 MHz sampling rate, 40-seconds of captured signal data results in a file that is 625MB in size. The *SignalSource* component that reads the data from the file and raises an empty event (with no attached listeners), performing no processing, requires approximately 18 seconds to reach the end of file.

With the addition of a single satellite tracking loop, it takes 57 seconds of processing time to get to the end of the file, which is longer than the original file duration. If the signal were being received in real time, the processing activities would obviously not be maintaining pace with the sample arrival events. Connecting a second tracking loop worsens this timing situation, requiring 79 seconds to process the entire file. The results obtained from this testing are summarized in Table 10-3.

| Activity | Time (s) |
|----------------------|----------|
| File reading only | 18 |
| Track one satellite | 57 |
| Track two satellites | 79 |

Table 10-3—Performance testing on 40 seconds of data from the SiGe EK3 front-end hardware

The high sampling rate and low information density of the samples, 2-bits of signal information for every stored byte, make for a cumbersome large file to process in real time. Also, due to the low ratio of sampling rate to IF, the PLL processing component frequently loses its lock during processing operations and has to perform extra steps to reacquire the carrier phase. The lack of the pipeline recording objects makes it difficult to demonstrate the PLL's loss of lock, however during testing the *Locked* property of the *PLLPipelineComponent* was frequently observed to transition to the FALSE state.

A more desirable front-end would utilize an I/Q sampler operating at a lower sample rate, but with an IF that has been down-converted to a value much closer to baseband. Each I and Q sample would be represented by a signed (two's complement) 4-bit binary sequence with I+Q data packed together in an 8-bit byte formatted as: $Q_5Q_2Q_1Q_0 I_5I_2I_1I_0$,

where the subscripted S denotes the sign bit and 2, 1, and 0 represent the three magnitude bits.

The signal characteristics described and evaluated in (2) make use of a sampling rate of 2.1518 MHz and an IF of 17.248 kHz. The reduction of sampling rate places a limitation on the precision with which the position of the C/A code alignment can be found, however. The ambiguity of the code phase depends on the commensurability ratio of the sampling rate over the chip rate; the higher the common factors between the sampling rate and chip rate, the greater the potential error. At 2.1518 MHz the resulting error in positioning accuracy is calculated to be approximate 1.3 cm (2).

For testing, a 40-second signal was simulated using these characteristics and containing the same satellite signals (PRN #s 18, 21, 22, and 26) as the June 5, 2009 recorded signal file previously analyzed. The code delay values used were the same values measured in the original signal file, however the Doppler values simulated were the theoretical predicted values based on the satellites' orbital velocities. The reason for using the predicted Doppler values instead of the measured values was to ascertain to what extent the measured values obtained in Table 10-1 were affected by numerical issues with the software and how much they were influenced by clock or other inaccuracies in the hardware.

The I and Q components of the test signal were created using the following numerical equations:

$$I = Signal \times Cos(2\pi \times (IF + Doppler) \times t) \times PRN(t - CodeDelay) \times NavData(t) + noise(t) \quad \mathbf{10-2}$$

$$Q = Signal \times Sin(2\pi \times (IF + Doppler) \times t) \times PRN(t - CodeDelay) \times NavData(t) + noise(t) \quad \mathbf{10-3}$$

where *Signal* is the relative scale factor for the signal amplitude, *Doppler* is the positive or negative Doppler shift amount, *PRN* is the chip code value (± 1) for the specified time minus the code delay, and the *NavData* is the stream of navigation data bits (± 1) at a 50 bps rate. Noise was added to the signal using a uniform random number generator.

The calculated I and Q values were then scaled to the same ± 3 Volt range as the original SiGe front-end hardware, and quantized into a 4-bit representation. The 4-bit Q nibble was shifted left by 4-bits, and then combined (bitwise OR) with the I nibble before the merged byte value was written to disk. The lower sampling rate and higher information density resulted in a 40-second file that was only 80 MB in size. A section of the simulated signal is shown in Figure 10-7.

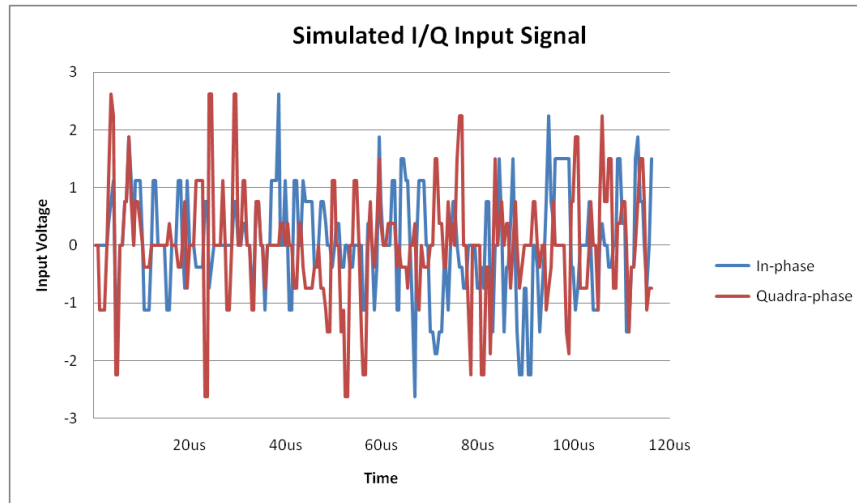


Figure 10-7—Input I/Q signal from the simulated signal source used for testing

The calculated spectrum for the simulated signal is shown in Figure 10-8. A longer length FFT was used in order to increase the frequency resolution of the result because of the lower IF and sampling rates involved. Since the L1 signal is down-converted closer to baseband, some of the signal energy from the lower sideband region is lost, reducing the overall SNR. Also, since the input data are complex values, assumptions regarding the symmetry of the negative frequencies in relation to the positive ones no longer apply. Hence, the spectrum of Figure 10-8 is shown double sided.

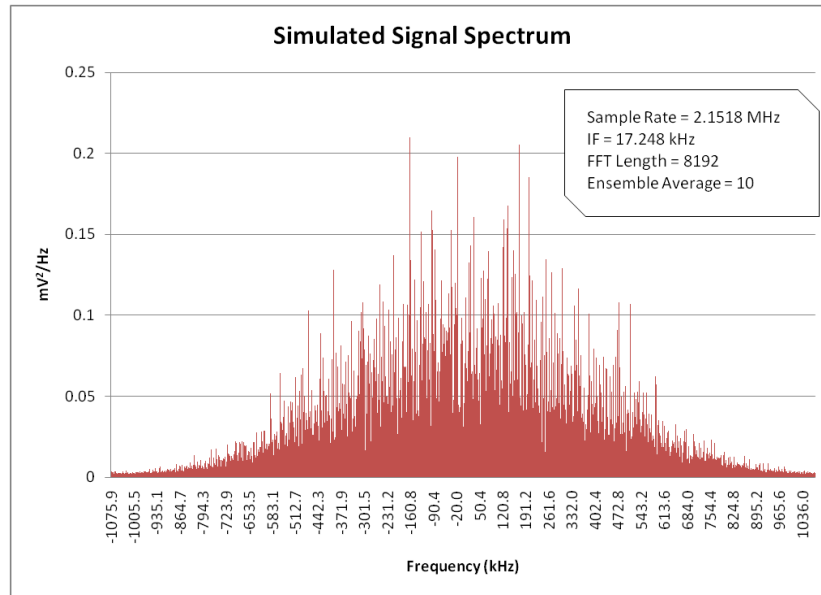


Figure 10-8—Double sided spectrum for simulated signal source

In order to accommodate the packed byte format of this file type, a new *FileSignalSource* class (Figure 8-7) was created that inherits default behaviors and the *ISignalSource* interface implementation from *FileSignalSource*, but overrides the *ReadData* method. The *QuadratureFileSignalSource* class provides the facility to read a byte of data from the specified file, unpack the I and Q values, and return a complex sample with real and imaginary parts. The properties for IF and sampling rate are specified and returned according to the indicated values. An instance of this type is then created and used to initialize the pipeline component *SignalController*. The rest of the testing application remains unchanged since all of the operations and behaviors are encapsulated and abstracted through the related base class declarations.

As was done previously, all PRN sequences present in the signal were detected, the code delays measured and the Doppler amounts determined during an initial acquisition

process. Figure 10-9 shows the same peak detected for PRN #18 at 0.917 ms in the simulated signal, the same as Figure 10-4 for the real signal.

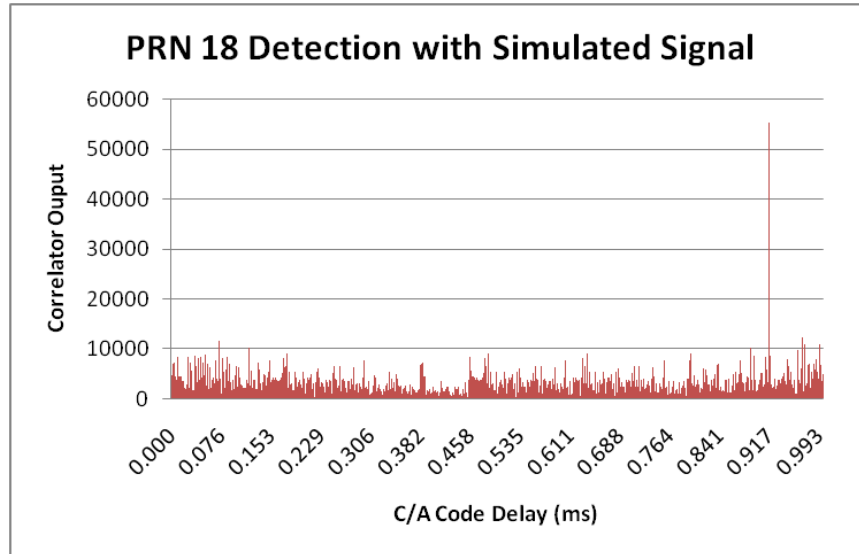


Figure 10-9—Circular correlation peak detection using simulated signal source

Likewise, after the code delay has been found, the Doppler shift can be determined by locating a peak in the frequency transformed version of a longer signal sample that has been multiplied by the detected PRN code of the correct delay. The signal peak is obvious in the chart of Figure 10-10, located at the expected value of about 19.4 kHz (17248 + 2149 Hz).

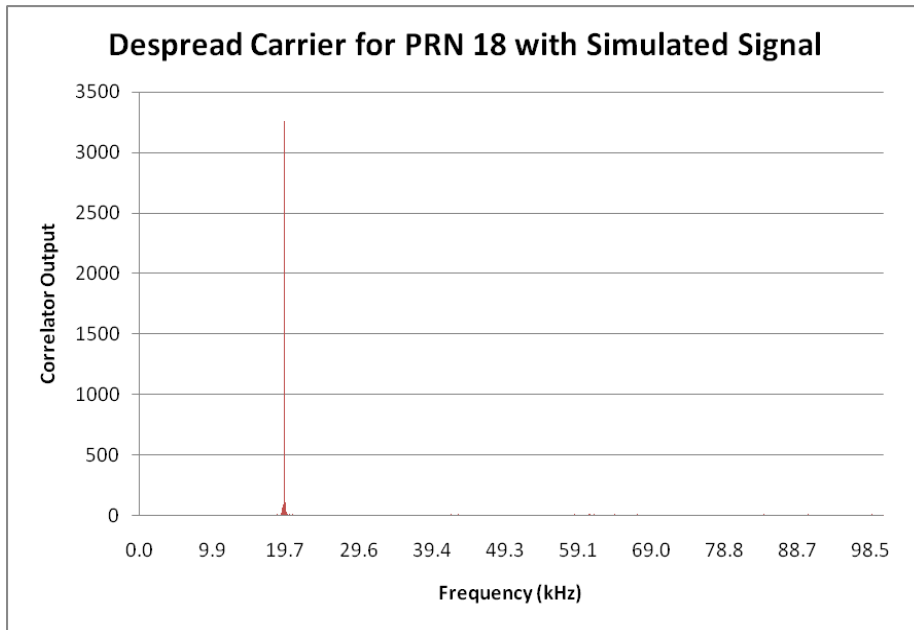


Figure 10-10—Frequency-domain view of carrier using the simulated signal source

Finally, if desired, the carrier can be recovered and viewed in the time-domain by band-pass filtering the signal and taking the inverse Fourier transform of the result. The resulting output of this operation is shown in Figure 10-11, however it is usually not a necessary step for the purposes of tracking the received signal and is shown here for illustrative purposes only.

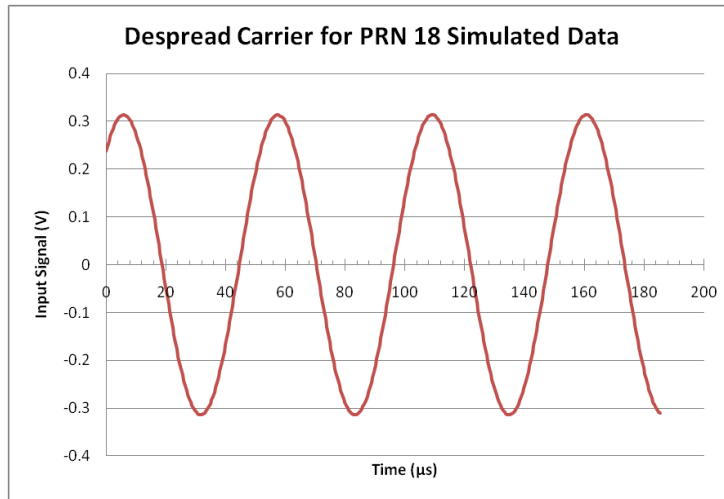


Figure 10-11—Time-domain view of recovered carrier using the simulated signal source

The results of the signal acquisition and detection processing are summarized in Table 10-4. All of the PRNs contained in the signal were identified with the correct delays. The errors between the actual Doppler values used in the simulated signal and the Doppler as measured can be directly attributed to limitations in the precision of numerical signal processing, since there is no hardware involved or other external physical phenomenon at play.

| PRN # | Code Delay (ms) | Actual Doppler (Hz) | Measured Doppler (Hz) | Error (Hz) |
|-------|-----------------|---------------------|-----------------------|------------|
| 18 | 0.917 | 2149 | 2124 | -25 |
| 21 | 0.094 | 389 | 417 | 28 |
| 22 | 0.043 | 2759 | 2781 | 22 |
| 26 | 0.428 | 1508 | 1533 | 25 |

Table 10-4—Satellite tracking results with a simulated signal

The average Doppler error for the simulated signal is 12.5 Hz, while the standard deviation is 25 Hz. The transform-based frequency measurement has a maximum

resolution that is determined by the ratio of the sampling frequency to the number of data bins in the transformation. The greater the number of bins, the higher the frequency resolution obtained, at a cost of increased computation time. For these tests, the corresponding ratio is

$$\begin{aligned} \text{Frequency resolution} &= \frac{2.1518 \text{ MHz}}{32\,768} && \mathbf{10-4} \\ &= 65.67 \text{ Hz per bin} \end{aligned}$$

The computed errors are all within this limiting value.

Similar to before, real time performance measurements were obtained using this simulated signal source and are summarized in Table 10-5. Since the file was significantly smaller, a much lower amount of time was required just to read its contents; the file read operation without processing completed on average in less than two seconds. Tracking of one, two, and four satellite signals simultaneously was also achieved, all within an amount of time less than the actual recording duration of the file.

| Activity | Time (s) |
|-----------------------|----------|
| File reading only | < 2 |
| Track one satellite | 16 |
| Track two satellites | 29 |
| Track four satellites | 38 |

Table 10-5—Performance testing on 40 seconds of data from the simulated signal model

The additional information present in the I/Q format of the data significantly simplified the phase and frequency tracking operations of the PLL, which managed to

stay in a locked status through the entirety of the processing. I and Q processing offers advantages over real only samples in that the additional information provided can be used to calculate the frequency when the exact phase is unknown and to find the frequency by the rate of change of phase. The navigation data bits were simulated using an instance of a PRN class that had its chip rate set to 50. This configuration allowed the data bit edge for each simulated satellite to align properly with the chip edge, as it would in a real signal, but it does not encode an actual GPS navigation message. However, the phase of the carrier is not adjusted by the simulated code delay, and all signals are given the same navigation message and noise values. Furthermore, the simulation noise, usually modeled as a band-limited Gaussian process, was actually created using a random number generator with a uniform distribution, and the same noise value was added to both the I and Q signals. A better, more precise, way to create the simulation data would be to add a band-limited noise value to a simulated real signal and then, using an implementation of a Hilbert transform, form the Q signal component.

For the purposes of evaluating and testing the real time tracking capabilities of the framework, the signal as simulated should be sufficient.

10.3 Interoperability Component Integration

The motivation in providing an interoperability support layer for the receiver development framework was to enable hardware and software components developed and supported in different application environments a means of tying into the signal processing model. Likewise, application features of the framework can be enhanced by the integration of additional functionality that has previously been developed by other members of the receiver research community.

The integration and testing of the I/Q simulated signal source is an example of the capabilities and potential of this interoperability approach. Although the signal was simulated, the adaptations involved in the framework are identical to what would be required had the signal source been a physical piece of front-end hardware.

In order to accommodate the packed byte format of this file type, a new *FileSignalSource* class was created that inherits from *FileSignalSource*, gaining default behaviors and the *ISignalSource* interface implementation. By overriding the *ReadData* method, polymorphism allowed the correct method of the new class to be invoked at run time. As far as the rest of the processing components were involved, the details of the format and actual source of the new signal were irrelevant.

In a similar manner, any hardware or software component can be adapted to implement the appropriate component interface, either through inheriting a level of already existing functionality and overriding the differences in properties and methods, or by starting a new class type that implements the interface directly. Had the new signal source been a physical piece of hardware, the only additional complexity in creating support for it would be in the challenges associated with writing or finding a suitable Windows® device driver. The degree of complexity involved is usually determined more by the complications of working in the realm of the required 3rd-party tools and environments.

Appendix A provides background and details on the mechanics of working with 3rd-party application libraries and developing interoperability components in general. As a further example and demonstration of the steps involved, a phase-lock loop component

from the open-source **GNU Radio** (63) project will be integrated into the signal processing pipeline of the test configuration. Although the basic approach is the same, following the four-layer model presented in Chapter 7, the work of this section is not for the faint of heart.

GNU Radio consists of a mixed bag of hardware and software technologies. The project contains many worker classes and utilities for building SDR and other signal processing related applications. The first step in connecting to this ready-made functionality is to build (compile and link) the GNU GRC library called **general**; building it is hard (64):

“Considerable effort has been put into making the GNU Radio code portable among various operating systems, but there are several reasons why it cannot be “simply” compiled and run under Windows:

- *The build and install procedures [sic] are based on Linux scripts and tools*
- *Several third-party libraries are used, each with its own, often system-dependent, installation procedure*
- *Most GNU Radio applications must interface to hardware (e.g., a sound card or USRP) which require system-dependent drivers and installation procedures*
- *Because GNU Radio is written as an extension to Python, there are potential problems on Windows if different runtime libraries are used for GNU Radio and Python”*

Not to editorialize, but it’s clear that in their efforts to keep the *GNU Radio* project system agnostic, the implementers have instead made life difficult for everyone who

wishes to use it. Specific build instructions change with each version, so no additional details on building the imported libraries will be provided, here.

After getting the appropriate source files compiled into a static library, the next step is to create a new Windows DLL (this DLL is for *dynamic-link library*, not delay-lock loop) project; here it is called *GNURadioParts*. To this project was added the previously built *general.lib* to the list of linker inputs. Also added was the *gr_pll_refout_cc.h* file to the *stdafx.h* precompiled header file, as shown in Figure 10-12.

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
#pragma once

#include "targetver.h"
// Exclude rarely-used stuff from Windows headers
#define WIN32_LEAN_AND_MEAN
// Windows Header Files:
#include <windows.h>

// TODO: reference additional headers your program requires here
#include "..\..\..\gnuradio-3.2.2\gnuradio-
core\src\lib\general\gr_pll_refout_cc.h"
```

Figure 10-12—Precompiled header file *stdafx.h* used for the *GNURadioParts* library project

The purpose of creating this library is to produce the interoperability type adaptations and function exports (Layer-2 and Layer-3). The *GNURadioParts.h* file shown in Figure 10-13 will declare the exported function *UpdateOutput(...)* that takes a single complex value as an input and returns a complex value as an output.

```

// The following ifdef block is the standard way of creating
// macros which make exporting from a DLL simpler. All files
// within this DLL are compiled with the GNURADIOPARTS_EXPORTS
// symbol defined on the command line. This symbol should not be
// defined on any project that uses this DLL. This way any other
// project whose source files include this file see
// GNURADIOPARTS_API functions as being imported from a DLL,
// whereas this DLL sees symbols defined with this macro as being
// exported.
#ifdef GNURADIOPARTS_EXPORTS
#define GNURADIOPARTS_API __declspec(dllexport)
#else
#define GNURADIOPARTS_API __declspec(dllimport)
#endif

//GNU Radio PLL component declaration, initialized in DLLMain
extern GNURADIOPARTS_API gr_pll_refout_cc* PLLInstance;
//GNU Radio PLL component interface wrapper exported function
extern "C" {
    GNURADIOPARTS_API std::complex
        UpdateOutput(std::complex<double> input);
}

```

Figure 10-13—Header file `GNURadioParts.h` with the `UpdateOutput(...)` function exported from the `GNURadioParts` library project

The implementation of `UpdateOutput(...)` in `GNURadioParts.cpp` is shown in Figure 10-14. The simple template library (STL) complex template is used to map the input value to a C++ complex type. An instance of the PLL class has been previously declared in Figure 10-13 and initialized in Figure 10-15. All that is required is to forward (redirect) the input argument to the `work(...)` method of the `GNU Radio gr_pll_refout_cc` class. When the method call completes, the returned value is subsequently passed back to the caller from the upper layer.

```

#include "stdafx.h"
#include "GNURadioParts.h"
#include <complex>

// This is the exported GNU Radio PLL work function wrapper code
// The input and return types are from the STL complex library,
// which happen to be compatible with the Receiver Framework
// Complex type.
GNURADIOPARTS_API std::complex
UpdateOutput(std::complex<double> input) {

    std::complex<double> output;
    std::vector<gr_vector_const_void_star> input_items(input);
    std::vector<gr_vector_void_star> output_items(output);

    //The actual operations are performed in the call to the GNU
    //Radio PLL gr_pll_refout_cc class member, work.
    PLLInstance->work(int(1), input_items, output_items);

    return output_items;
}

```

Figure 10-14—CPP source file *GNURadioParts.cpp* showing the *UpdateOutput()* function implementation

The *DllMain* entry point in *DLLMain.cpp*, Figure 10-15, is called when the library module is loaded by the operating system. Rather than repeatedly initializing an instance of the PLL class, it is simpler and more resource efficient to create and initialize a shared instance when the application is first loaded. The instance is first declared in *GNURadioParts.h*, shown in Figure 10-13, but note that the instance marked as external (*extern* keyword). The actual instance is defined in *DLLMain.cpp*, but its existence needs to be communicated to the *UpdateOutput(...)* function in *GNURadioParts.cpp*.

```

// dllmain.cpp : Defines the entry point for the DLL application.
#include "stdafx.h"

GNURADIOPARTS_API gr_pll_refout_cc* PLLInstance;

BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
        PLLInstance = new gr_pll_refout_cc(float alpha,
        float beta, float max_freq, float min_freq);
        break;
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        delete PLLInstance;
        break;
    }
    return TRUE;
}

```

Figure 10-15—CPP source file DllMain.cpp with the gr_pll_refout_cc instance initialization

The implementation of the Layer-4 component wrapper, *GNURadioWrapper.cs*, is shown in Figure 10-16. This C# file declares the GNURadio namespace and creates the linkage between the *GNURadioWrapper* class and the *GNURadioParts* library.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace GNURadio {
    public class GNURadioWrapper {

        [DllImport("GNURadioParts")]
        public static extern Complex UpdateOutput(Complex input);
    }
}

```

Figure 10-16—C# source file GNURadioWrapper.cs that imports the GNURadioParts library

A new *PipelineComponent* class needs to be declared in order to connect the *GNURadioWrapper* class to the signal processing pipeline. This work is accomplished in *GNUPLLPipelineComponent.cs*, shown in Figure 10-17.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PipelineComponents {
    public class GNUPLLPipelineComponent : PLLPipelineComponent {

        //Non-default class constructor:
        public GNUPLLPipelineComponent(NCOPipelineComponent NCO,
            DemodulatorPipelineComponent Demod):base(NCO, Demod) {
        }

        //PipelineComponent inherited method:
        public override void UpdateOutput() {
            Output = GNURadio.GNURadioWrapper.UpdateOutput(Input);
        }
    }
}
```

Figure 10-17—C# source file GNUPLLPipelineComponent.cs for invoking the GNURadioWrapper UpdateOutput() static method

The class, *GNUPLLPipelineComponent*, inherits from the existing *PLLPipelineComponent* type, and overrides the non-default constructor and the *UpdateOutput()* method. The PLL maintains references to an NCO instance that is shared with the DLL (now this is a delay-lock loop), and a demodulator component that is used to convert the real portion of the PLL output into a binary data stream. The PLL also receives bit synchronization timing information as feedback from the demodulator.

The required adaptations in the existing *PipelineContainer* merely involve changing the type name of the PLL instance to the new class name. The pipeline holds a private member of the *PLLPipelineComponent* base type, and since the

GNUPLLPipelineComponent is derived from this class, it can be treated as an equivalent type. The modifications required are shown in Figure 10-18.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using PRNCodeGenerator;

namespace PipelineComponents {
    public class PipelineContainer : PipelineComponent {

        :
        :
        //PLL Class instance:
        private PLLPipelineComponent PLL;
        :
        :
        PLL = new GNUPLLPipelineComponent(NCO, Demodulator);
        :
        :
    }
}
```

Figure 10-18—PipelineContainer PLL member declaration and initialization for GNUPLLPipelineComponent class integration

Everything else in the existing application stays exactly the same since the declared type of the PLL is of the base type *PLLPipelineComponent*, but at run-time the actual derived type is *GNUPLLPipelineComponent*, and polymorphism ensures the correct method is executed. The *UpdateOutput()* method is invoked from within the *Done()* event handler of the base *PLLPipelineComponent* class.

The performance results with the simulated signal and the *GNU Radio* PLL are unchanged from the results previously presented in Table 10-5.

Chapter 11 Conclusion

The design and development of a real-time software GNSS receiver research framework has been demonstrated and tested. The pipelined processing model 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. The interoperability layer defined by the framework allows for the direct integration of external components from other sources, and the extensibility characteristics of the provided object-oriented design allow new functionality to be created while permitting high levels of code reuse.

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 these applications requires more than an overall optimization effort on the part of software and algorithm implementers.

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 reuse 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.

The tracking loops as tested and presented consist of only the most direct interpretation of the basic processing techniques and requires more sophisticated optimizations and improvements to enhance the overall performance characteristics. However, there are many researchers actively focusing on algorithm optimization, and by exploiting the integration goals of the framework, the results thus obtained can be incorporated into this solution for testing and evaluation.

The reference receiver needs the integration of an 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. The effective process for acquisition is equivalent to a receiver in a “cold-start” mode. Given an approximate estimate of location from the last known position, and a reasonably accurate system time, the search space and corresponding processing duration for identifying in-view satellites can be dramatically reduced.

Going forward, many of the list and collection types in the framework could benefit from the definition of C# generics, or templates in C++, to simplify the process of creation of new components. Also, 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 converts 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. The *SiGe EK3 USB* sampler used in this research operates at a ≈ 4 MHz IF and a ≈ 16 MHz sample rate (although a newer one has been made available that uses a sampling frequency of 8.1838 MHz and an IF of 38.400 KHz). For carrier frequency and phase tracking, the error in the angle between samples needs to be kept at less than $\pi/2$, but at four samples per signal cycle the minimum possible phase difference is $\pi/2$. Simply satisfying the Nyquist rate for choosing the sampling rate is not sufficient for many control-related applications (65) (66). Increasing the sampling rate improves the stability and tracking capabilities of DLL and PLL loops (57). 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 (19). A baseband software processor for GPS was developed and evaluated in (2) using an IF of ≈ 20 kHz and a sampling rate of ≈ 2 MHz, but only tested under simulated signal conditions and without the presence of a navigation message. Also, reducing the sampling rate reduces the precision with which the C/A code delay can be found, so a careful balance must be struck between performance and accuracy.

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 multimedia time reference purposes, which could also be used for front-end signal 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 even a de facto agreement from the community on the

establishment of a binary file format that defines an embedded header for holding an information structure that includes byte ordering (endianess), sample rate, and other information required for cross-platform and hardware compatibility.

It is also worth mentioning that the event-driven pipeline model developed for the receiver does not need to be limited to real-time systems. Highly complex calculations may benefit from the ability to create multiple parallel instances of a sequential processing chain, even though real-time operation would not be achievable due to the computational load. There are many possible applications of the software pipeline architecture for controls and signal processing work. When combined with the memory, data storage, and network access capabilities of a PC, well, one's imagination becomes the limiting factor.

The classes of supporting components developed as part of this framework can also provide benefits to other applications as well. The PRN generator classes and signal sources, for example, can be used outside of the framework pipeline and can act as input signal simulators for the development and testing of a wide range of utilities.

The significance of this work lies in the establishment of the collection of object models and base implementations for real-time receiver development. Without it, there is a limitation to the degree of improvement to software receiver performance that can be made through individual optimization efforts alone. By adopting the principles and integration philosophies embodied and presented in this work, world-wide efforts can be combined into a unified development model.

The release of this document corresponds to version 1.0 of the receiver framework. There is still much work to be done and many opportunities for making improvements. It is difficult to predict how consumers of any object-model will need to adapt and change its structure in the future. As the goal of establishing and distributing the framework proceeds, the feedback obtained from testers and users alike will serve as a means for enhancements and a better overall application environment.

Appendices

Introduction to the Appendices

These are the things that I had to search many texts to find out and the connections or discoveries I've made along the way. There is likely nothing new here, (except for Appendix A) and you've probably seen it already before. I'm including this material not necessarily to support the work, but to aid in someone starting out so that they don't have to go through it all themselves.

Appendix A—3rd-party Toolkit Interoperability

As an example of the approach of the receiver framework towards interoperability, a few details on the integration of the open-source GPS toolkit library, *GPSTk* (54), from the University of Texas at Austin, are provided in this appendix.

The GPSTk toolkit is written primarily in C/C++ and contains many features and functions for working with GPS almanac data files, performing date-time conversions, and extracting ephemeris data from the received satellite navigation message. The library comes with source code, compiled libraries for static linking, and several command line applications. Rather than rewriting and testing the ready-made functionality provided by the toolkit, it is far easier to make use of such resources as they are, in place.

The purpose of this appendix is to provide relevant background information and details on using C/C++ to create a native-code Windows library that exports functions and symbols that are consumable by .NET applications and is connected to the receiver framework. This approach may be used to provide a bridge between managed and unmanaged code that abstracts many of the type-conversion complexities and issues involved with the typical direct approach of invoking native code through the various *Interop* assemblies.

Library Background and History

In the past, application code was written and rewritten without regard for how it could be recycled and reused. In the interest of increasing code reuse, the idea of bundling collections of valuable and useful, but generically implemented, functions into shareable

redistributable software libraries eventually evolved. Source code, such as C/C++ and machine specific assembler files were compiled into object modules and then merged together into libraries. Typically, code with similar functional purposes was grouped together into a single aptly-named library. The librarian tool, **LIB**, was used to build and maintain these various libraries.

When an application required a piece of code that was available in one of the libraries, the developer would declare to the compiler that the functionality was implemented externally. The compiler would produce intermediate object files without immediately trying to locate all of the missing pieces. It was then up to the link operation to combine the object files and extract from the libraries the external functions. This process created a **statically linked** application: a copy of the function was taken from the code in the library and statically embedded into the application.

For all of the reuse benefits, static linking has a couple of disadvantages. If a large application consists of several executable modules that all share code from a common set of libraries, each of those modules will increase in size by the amount of code extracted from the libraries. When the applications run, they will each have their own copy of the code in memory and it will all be loaded whether or not the functions are ever actually called by the application. *{Just to clarify: an optimizing linker will not link to external library code that is never called by the application, but if an external function is called within a conditional statement that evaluates to false at run time, that function's implementation will be linked and loaded because it could have been called.}* Loading many copies of the same code into memory is not a very efficient use of available system resources.

What was needed was a way of dynamically linking the libraries into a shared module that could be loaded into memory just once and only when it was needed. Shared code could be written, compiled, and linked into a *dynamic-link library* (DLL) that could be accessed by several applications all at once, if needed. Any functions that were to be exported from the library would be added to a lightweight LIB file that contained just enough information to keep the linker happy—unlike their static counterparts, these LIBs contained no executable code. Application authors wishing to make use of the services provided by the DLL would add the LIB file to the list of linker inputs, and as long as the DLL could be loaded when the application needed it, calls into the functions would run as if they had been statically linked all along. Parameters surrounding the loading of the library into memory could be controlled to improve the application's startup time and to reduce its memory footprint.

In order to create the LIB file, exported function names first had to be either added to a module definition file, .DEF file, or the function entry points needed to be marked with the *_export* keyword (16-bit applications used *_export*, 32-bit versions later replaced this syntax with a *_declspec(dllexport)* tag.) A utility, *implib*, would produce the LIB file using either the DEF file or the DLL itself as an input. The DEF file provided better control of the export behaviors, where the *_export* approach was just simpler.

Consuming these libraries was not always an easy thing to do, especially if they had been developed by a 3rd-party independent software vendor (ISV). The function prototypes (names and parameters) must be declared before their use, along with any custom symbols, types, or enumerations that are required by the library interface. Typically, these things are placed in C-style .H header files that are <included.h> with the

application source code. It may seem very structured and abstract, providing the sense of separation between declaration and implementation, however, in practice the code frequently becomes an ugly mix of preprocessor directives, obtuse macros, conditional includes, and hard-coded file paths.

The .LIB files had to be added to the linker input list, and the DLLs needed to be available when the application ran. When trying to locate DLL modules, the runtime loader would only look in the current directory, usually where the application was started, and in the system directory. If something couldn't be found, the application would die in a most shameful manner. Keeping in mind that the libraries were intended to encapsulate shared functionality, applications that were modularized into multiple executables in self-contained directory structures would either require multiple copies of the shared DLL in multiple locations, or a single copy in the operating system root directory where they would often overwrite one another.

The problem with the C language is that it has a tendency to constantly pull the developer away from the abstract into the detail view. Most experienced C/C++ developers are aware when they are writing code what registers will hold variables, how values are being stored and what the stack-frame looks like at any point in time, among other minutiae not directly related to the task at hand.

Other more modern languages, however, are much better suited to developing feature-rich applications, where too much fretting over the compiler details just gets in the way of being productive and innovative. Unfortunately, there are development language issues to deal with because C header files are only useful to C language

compilers. When C code is included with a Visual Basic application, the result is general unhappiness. Interoperating C libraries with non-C languages has been problematic, to say the least.

Component Object Model

Using (67), in particular the *Abstract Factory*, *Factory Method*, and *Builder* patterns, Microsoft® developed the Component Object Model (COM) interface standard, which later became COM+ (the + is for *new and improved*) with the release of Windows 2000. The COM+ subsystem provides class factory methods that allow subclass types to be created through an interface provided by a virtual base class. The underpinnings of COM take advantage of the C++ multiple inheritance and polymorphism capabilities to define an abstract base class that all other COM-capable objects must derive from and implement. Pointers to the base-class type are passed into and out of method calls and eventually cast to a concrete implementation at run time.

COM-accessible components assign newly defined classes (types) a globally unique identifier, a *GUID*, and store information on how to create instances in a central searchable location, namely the system registry. An application can then ask the operating system to create an instance of a class by invoking a method on the abstract base (through the common interface) that returns a pointer to an object as the base type that can eventually be cast to the appropriate derived type. This pointer can be used to call methods on the specific class implementation. All of this plumbing eventually allows non-C languages to create instances of, and call methods on, classes written in C++.

So, for an example, the Acrobat Access reader plug-in has a registry entry that looks like Figure A-1.

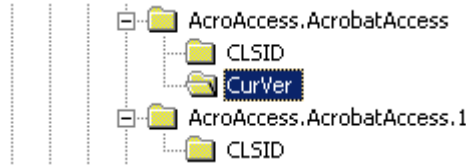


Figure A-1—Acrobat Access system registry entry

As shown, *AcroAccess.AcrobatAccess* is the version independent program ID (progID). An application wishing to create classes of types supported by this library can refer to the progID by name. The current version, *CurVer*, key contains the version specific program ID, *AcroAccess.AcrobatAccess.1*, as shown in Figure A-2.

| Name | Type | Data |
|---|--------|----------------------------|
|  (Default) | REG_SZ | AcroAccess.AcrobatAccess.1 |

Figure A-2—Acrobat Access version specific program ID

The class ID key, *CLSID*, of this entry contains the GUID of the executable that holds the implementation runtime of the class library, as shown in Figure A-3 and Figure A-4.

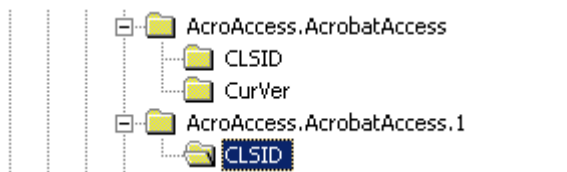


Figure A-3—Acrobat Access class ID key


| Name | Type | Data |
|---|--------|--|
|  (Default) | REG_SZ | {C523F39F-9C83-11D3-9094-00104BD0D535} |

Figure A-4—Acrobat Access class ID value

The value of *{C523F39F-9C83-11D3-9094-00104BD0D535}* given in Figure A-4 references another registry key that contains, among other information, the name and location of the executable, shown in Figure A-5.



Figure A-5—Acrobat Access InprocServer32 sub-key

Where, the *InprocServer32* sub-key provides the name and path of the executable file, as shown in Figure A-6.



| Name | Type | Data |
|--|--------|--|
|  (Default) | REG_SZ | C:\Program Files\Adobe\Acrobat 7.0\Reader\plug_ins\Accessibility.api |
|  ThreadingModel | REG_SZ | Apartment |

Figure A-6—Acrobat Access executable file registry entry

The COM model defines two pure-abstract base-classes that work in similar but somewhat different manners, *IUnknown* and *IDispatch*. C++-style languages that support *v-table* method calls, such as C++, use the *IUnknown* interface. Other late binding scripting languages, like Visual Basic, VB Script, and Java Script, use the

IDispatch interface, which inherits from and extends *IUnknown*. The methods defined in these two interfaces are:

(68) *IUnknown*: *QueryInterface*, *AddRef*, and *Release*

(69) *IDispatch*: *GetTypeInfoCount*, *GetTypeInfo*, *GetIDsOfNames* and *Invoke*

So, to create an instance of the Acrobat Access class, the Win32 function ***CoCreateInstance*** is called with the version independent program ID *AcroAccess.AcrobatAccess*, and the returned *IUnknown* pointer can be cast to the specific *AcroAccess* plug-in type. From there, *QueryInterface* can be called to get references to any other functional interfaces that the class implements. All of the registry lookups and object instantiation details are left to the operating system to deal with.

Declaring an interface using the C++ syntax for an abstract class with pure-virtual methods is awkward and long winded. It also doesn't support the additional GUID information required for COM registration. The *Interface Definition Language (IDL)* leverages the C++ syntax and supports the extra attributes and keywords necessary for declaring object IDs and interface types. The Microsoft IDL compiler, ***MIDL***, processes this information and generates C files that are then compiled and linked into the class library. All of the type information exposed by the COM interfaces is listed in a type-library file, .TLB, that client applications can use to create references to the exported types.

MIDL code is the standard way of creating COM/COM+ compatible interfaces in C/C++. Further details on COM and the MIDL tools can be found in (70).

The .NET Framework

The .NET Framework includes a feature-rich *Base Class Library* (BCL) as a foundation for application development, and a managed code execution environment that provides system services for library dependency, versioning, and runtime security.

The *Common Language Infrastructure* (CLI) specification has been standardized under ECMA-335 and ISO/IEC 23271:2006 (71) and (72). The CLI encompasses a *Common Language Specification* (CLS), a *Common Type System*, and a *Virtual Execution System*. The Common Intermediate Language (CIL) (73) (CIL—formerly MSIL) is the lowest-level human-readable language in the CLI and the .NET Framework. Any development language that targets the .NET Framework runtime must be capable of generating CIL-compatible code. The Microsoft-defined .NET languages are C#, Visual Basic .NET, managed C++/CLI, and J#.

Using the .NET-aware languages, high-level code is compiled into assemblies of instructions in the CIL format. These assemblies can be either .EXE or .DLL files similar to native code, but they contain no CPU-specific binary compatible operations. Instead, the code is loaded and further compiled into binary code by the *just-in-time* (JIT) compiler of the .NET runtime the first time the application is executed. Each loaded assembly contains meta-data that describes the version specific details, dependency requirements, symbol exports, and any other applied attributes.

Exported types are visible through reflection, and consuming functionality exposed from a library is as simple as adding a reference to it in a project. Since there are no requirements for forward declarations of types or function prototypes, there is no need for

header or other types of include files. There are no .LIBs or exported type libraries, and no digging through the registry looking for interface IDs. More importantly, side-by-side version execution provides the capability to load two different versions of a DLL with the same file name without conflicts. ISVs can install their application-specific files into an isolated folder without stomping on other similarly-named system files; the .NET Framework will sort out all the details when the assemblies are loaded.

Memory management is provided by the runtime in that all class instances are created on a managed system heap. As references to objects go out of scope, memory will be automatically released by a garbage collection process that is provided to eliminate application memory leaks. Along with improved type safety and security checking, application stability is greatly enhanced. These changes, coupled with the Framework's base class library, allow software developers to focus on delivering a higher level of application functionality.

Integration of GPSTk

A .NET application can access Win32 native-code C library functions through platform invoke, *P/Invoke*, by specifying the function as external and by applying attributes that provide the name of the library where the function implementation can be located. Using P/Invoke requires class-method declarations for each of the C functions that one wishes to use, and all the data types that those functions require, in the .NET environment. However, not all C/C++ native types map directly to CLS compatible types, particularly where multiple levels of indirection are present, like passing the address of a pointer to a structure that contains a pointer to a void pointer. It's not always easy or necessarily possible to resolve these incompatibility issues.

It is possible to build a C/C++ DLL and use it to interconnect to the plumbing of the desired C-only API, exposing a much simpler interface with intrinsically compatible data types. Quite often, the overall goal is to achieve a single purpose task that is accomplished through two or more C function calls, where each call requires references to data structures that bear little relevance to the top layer. Rather than redefining types and function signatures for each piece, the easier solution is to write a single C function interface that instantiates the required data types and invokes the correct sequence of API calls. Only this function then needs to be exported from the library, and any input parameters that are necessary can often be passed using simpler data types, but more complex structures can be accommodated, as well.

Much has already been written on using `P/Invoke` and the ***DllImport*** attributes to link to unmanaged Win32 library code, but little information is provided on the requirements for writing an unmanaged C/C++ DLL that exports CLR-accessible symbols. Surprisingly, the rules for building a DLL in C haven't changed in many years, but some of the details have been obscured through the mists of time and legend.

The GPSTk library comes complete with several command line example applications that take lists of input arguments, but the whole thing looks very UNIX'y. Without having to rewrite the functionality or going through time-consuming hoops to get to it, an adapter DLL is created to call into the C-only GPSTk library.

Referring to the layered interoperability model discussed in Chapter 7, Layer-1 represents the actual GPSTk library code. The first step in creating the Layer-2 adaptation

involves starting a new Win32 DLL project in Visual Studio (any C++ compiler tool for Windows can be used), as shown in Figure A-7.

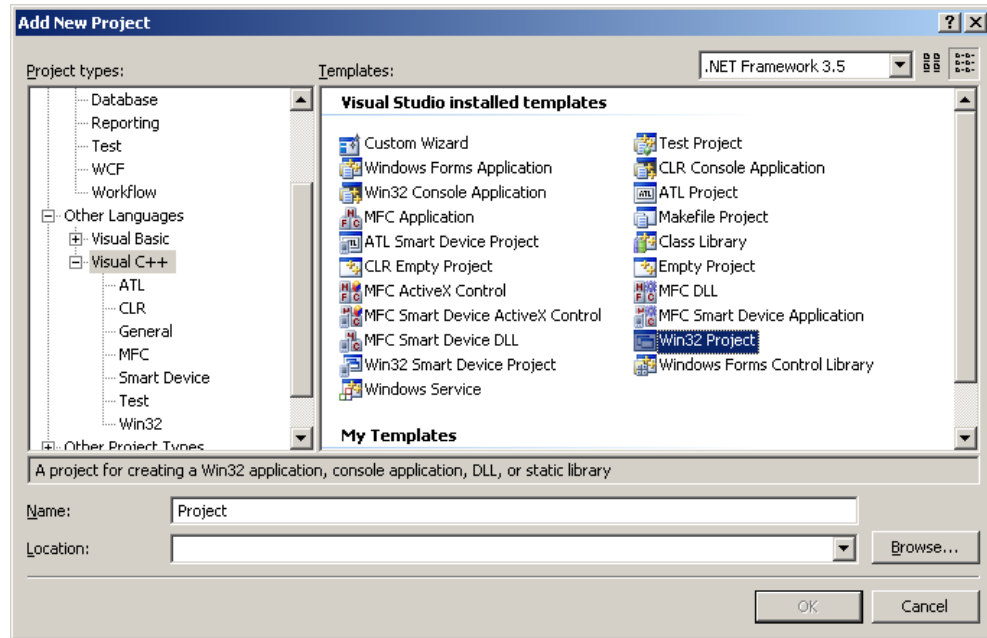


Figure A-7—Visual Studio 2008 new project dialog

After selecting the Win32 Project type, giving the project a name, and pressing OK, additional project options can be specified, as shown in Figure A-8. This dialog allows the specification of the application type (DLL) and an option to export symbols.

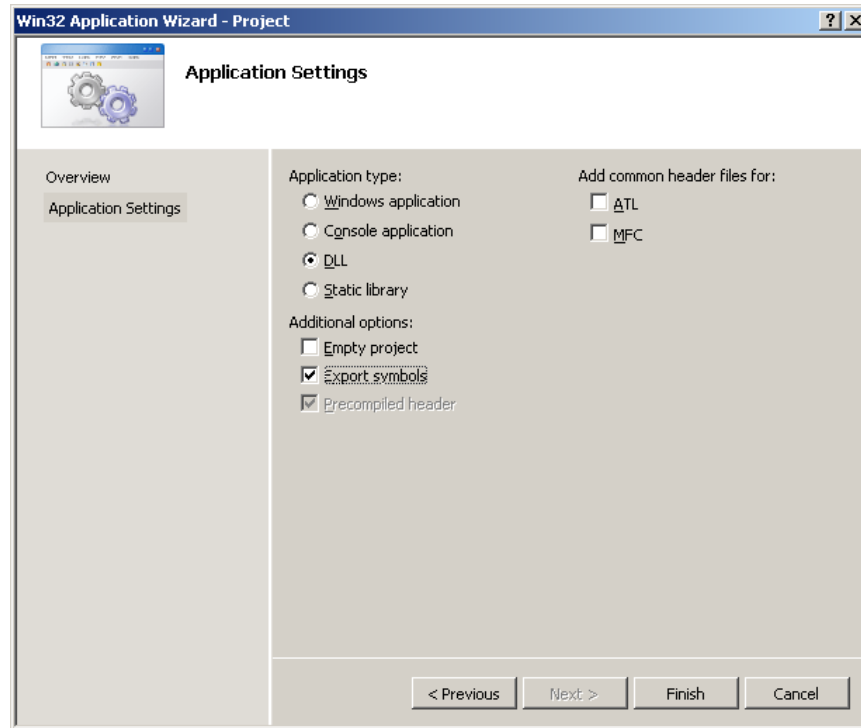


Figure A-8—Visual Studio 2008 new project dialog

If the *export symbols* option is selected, the *new project* wizard will generate a .h file (*projectname.h*) that contains the following code.

```
// The following ifdef block is the standard way of creating macros which
// make exporting from a DLL simpler. All files within this DLL are
// compiled with the GPSTKLIB_EXPORTS symbol defined on the command line.
// This symbol should not be defined on any project that uses this DLL.
// This way any other project whose source files include this file see
// TESTWIN32LIB_API functions as being imported from a DLL, whereas this
// DLL sees symbols defined with this macro as being exported.
#ifdef GPSTKLIB_EXPORTS
#define GPSTKLIB_API __declspec(dllexport)
#else
#define GPSTKLIB_API __declspec(dllimport)
#endif
```

Figure A-9—Exported code symbol C macro

As explained by the code comments, if the header file with the macro is included in a project that defines the GPSTKLIB_API symbol, the default will be to export declared functions from the library, otherwise the functions will be imported.

At this point, there are two choices for creating function exports. In the source code, create functions or classes with name declarations by adding GPSTKLIB_API before the function or class name, like in Figure A-10.

```
class GPSTKLIB_API SystemTime
    : public UnixTime
{
    public:
    //Further details elided..
```

Figure A-10—C++ class code exported using API macro

Alternatively, a *module definition file* (DEF) can be added to the project and the exports can be created manually. With this method, it's possible to explicitly specify which symbols are exported rather than exporting the whole class. Also, the symbol that is added to the generated .LIB file has to be searched for by name when the DLL is loaded. For performance reasons, it is sometimes desirable to specify an ordinal value for linkage rather than a text name.

The module definition file can be added to the project by adding a new item and selecting the Module Definition File (.def) icon (Figure A-11), giving the new file a name, and pressing Add. A blank DEF file will be inserted into the project.

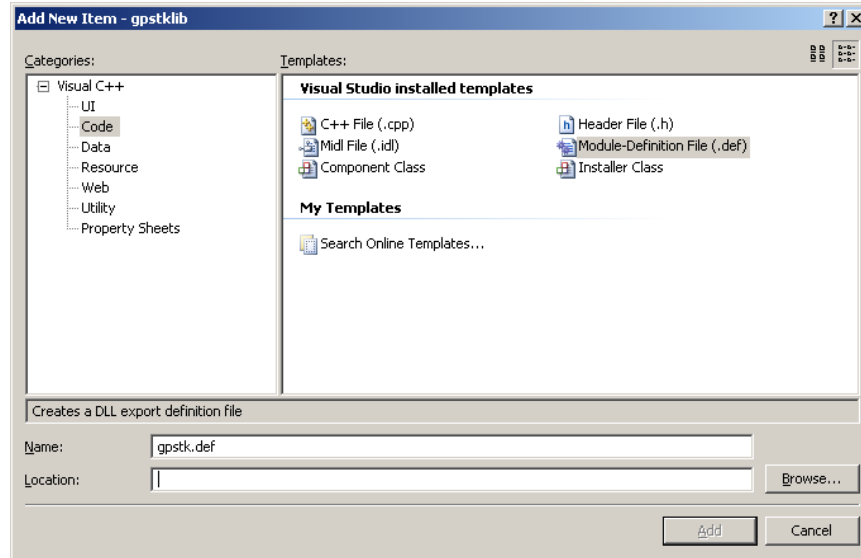


Figure A-11—Dialog for adding a DLL module export definition file

The newly created DEF file, Figure A-12, can be edited to supply the properties of the required exports. The LIBRARY section specifies the name of the output LIB file, and the EXPORTS part supplies the names of the exported symbols.

```

LIBRARY      "gpstklib"

EXPORTS
?GPSfullweek@DayTime@gpstk@@QBEFXZ      @100
?GPSsow@DayTime@gpstk@@QBENXZ          @200
?dayOfWeek@DayTime@gpstk@@QBEFXZ      @300
??0DayTime@gpstk@@QAE@XZ              @400
?day@DayTime@gpstk@@QBEFXZ            @500

```

Figure A-12—Definition file exported symbols

The function names are the result of the C++ name mangling mechanism that combines information about the class, method name, and input parameters to generate a unique signature. The easy way to get these names to add to the EXPORTS section is to simply copy them from the linker error output or the build log file the first time the

application is built. The other, slightly harder, way is to configure the linker to generate a map file that contains the mangled names of all the exportable symbols.

The @numbers after the symbol names are optional; they cause the linker/loader to resolve symbols by ordinal rather than by name. Calling performance is better but if the exports change, dependent projects will need to be re-linked. The numbers can be grouped and incremented according to any desired scheme. It is also possible to use a combination of both methods, DEF file and the export macro, to get pretty much any set of desired results.

Another way of declaring the exported functions is to add to a header file, like the *projectname.h* file that defines the import API macro, C-style prototypes for the necessary functions, as shown in Figure A-13.

```
extern "C" {
    GPSTKLIB_API short GPSfullweek();
    GPSTKLIB_API short GPSday();
    GPSTKLIB_API double GPSsecond();
    GPSTKLIB_API short day();
}
```

Figure A-13—API Macro exported symbols using C naming styles

The *extern "C"* prevents the name mangling, and is optional. These functions can be defined in a C/CPP source file, such as in Figure A-14.

```

using namespace std;
using namespace gpstk;

GPSTKLIB_API short GPSfullweek() {
    DayTime dt;

    return dt.GPSfullweek();
}

```

Figure A-14—C++ exported function implementation

This function simply creates an instance of the GPSTk class *DayTime* as defined in the library, and invokes the corresponding method of the class. The exported functions may also be declared with the GPSTKLIB_API attribute, Figure A-15, as long as the unmangled exports are put into the module definition file as in Figure A-16.

```

extern "C" {
    short GPSfullweek();
    short GPSday();
    double GPSsecond();
    short day();
}

```

Figure A-15—Exported symbols using C naming styles, without the use of the API macro

```

LIBRARY      "gpstklib"

EXPORTS
    GPSfullweek      @600
    GPSday           @700
    GPSsecond        @800
    day              @900

```

Figure A-16—Exported symbols using C naming styles

This approach, however, tends to flatten the existing hierarchical class structure making it look more like a C API than a C++ object-oriented solution. This technique will only work in situations where the class encapsulates only functionality, in that member fields and variables do not need to maintain their values between calls. A

possible solution is discussed later for those situations where persisting class state is necessary.

The next step involves the creation of the Layer-4 interoperability layer component. A class named GPSTK is declared as shown in Figure A-17. Each library function that needs to be called has to be added as a *public static extern member* of this class, and each requires a *DllImport* attribute from the *System.Runtime.InteropServices* namespace that provides, at a minimum, the name of the library where the function is implemented. The symbol that will be searched for in the library will be taken from the given function name using the unmangled (extern "C") name, but an entry point can be specified in the case where the .NET function and the export names are different.

```
using System.Runtime.InteropServices;

namespace GPSTKLib {
    public class GPSTK {

        [DllImport("gpstklib")]
        public static extern short GPSfullweek();

        [DllImport("gpstklib")]
        public static extern short GPSday();

        [DllImport("gpstklib")]
        public static extern double GPSsecond();

        [DllImport("gpstklib")]
        public static extern short day();

    }
}
```

Figure A-17—C# class for accessing the functions exported from the GPSTk library

In this case, the link will be to the symbols that are exported as the non-class wrapper implementations, as in Figure A-15. Since these are static members that belong to the class, code can invoke the corresponding methods through the GPSTK class name

without explicitly creating an instance, as shown in the button click event handler of Figure A-18.

```
private void button1_Click(object sender, EventArgs e) {  
    label15.Text = GPSTK.GPSfullweek().ToString();  
    label16.Text = GPSTK.GPSday().ToString();  
    label17.Text = GPSTK.GPSsecond.ToString();  
    label18.Text = GPSTK.day().ToString();  
}
```

Figure A-18—C# event handler that invokes functions from the external GPSTk library

Additional Features

When a DLL is available, but there is no export LIB file, it is possible to make one by using the LIB utility with the /def option if a DEF file is used, or by providing the name of the existing DLL itself. The result will be a LIB file describing the public exports that can be used to link C/C++ code to the DLL. Only symbols exported from the DLL at implementation time will be visible to the caller application.

A list of function names for a specific DLL can be found by running one of a variety of command-line tools. For example, you can use the following to obtain function names exported from the gpstklib.dll library,

dumpbin /exports gpstklib.dll or

link /dump /exports gpstklib.dll

When creating exports using the DEF file approach it is sometimes convenient to produce a complete list of all the private and public symbols being generated. This list can be obtained by getting the linker to generate a map file. In the DLL project property page, expand linker, select debugging, and in Map file name put in something like:

\$(OutDir)\\$(ProjectName).map

The created DLL has to reside in the application folder, where the EXE is launched, or on the path somewhere. A convenient place for these custom DLLs is usually the %WINDIR% directory—this is not a good place to install non-system DLL's. Deployment pain should be felt by the developer and not the end-user.

As previously mentioned, class member variables don't persist state across calls. Calling a couple of field set accessors and then invoking a *DoMagic()* operation may not result in the expected behavior. Actually, calling into class methods without an actual instance can be a hazardous thing to do since the v-table mechanism may not have been properly initialized when the call is made. It is possible, though, to add an exported global instance to the library and initialize it in the DllMain function of DllMain.cpp. DllMain is the DLL's equivalent to the standard C main function application entry point and is called by the system whenever the DLL is loaded or unloaded from memory. The function includes a flag that is passed to indicate the reason for the call. Testing the value of this flag can be used to determine when to create an instance, and when to delete it, as shown in Figure A-19.

```

GPSTKLIB_API DayTime* anInstance;

BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved )
{
    switch (ul_reason_for_call)      {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
        anInstance = new DayTime;
        break;
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        delete anInstance;
        break;
    }
    return TRUE;
}

```

Figure A-19—C++ code for initializing a persisted class instance

The global instance can be accessed from one of the application modules exported from the DLL, such as in Figure A-20.

```

void SetValue(double newValue) {
    anInstance->setMJD(newValue);
}

```

Figure A-20—C++ code for accessing a persisted class instance

To prevent memory leaks, it's necessary to delete the instance when the process or thread unloads the DLL, which will happen after a short period of non-use, typically just a few minutes without making a call. State will not persist across instances, for obvious reasons.

Depending on the requirements, a mechanism may have to be invented that will determine the validity of the class instance before the application relies on state values. This class factory will have to do reference counting and should be able to distinguish

between object instances. Since these requirements could apply to more than one type that is available in the library, it would be a good design idea to possibly assign an identifier to each class or type, something that could be passed into the factory method as an ID. The construction process could look up the type information from some sort of table indexed by the identifier and use the details it finds there to create and cast a pointer to the corresponding type. However, at this point, something about the design becomes oddly familiar in that this work would just be a reinvention of COM. In the end, the same problems recur and similar solutions result.

At first glance, one may be inclined to think that the COM design is overly-complex and largely unnecessary. As the generality of the problem becomes more clearly understood, one can better appreciate features provided by the COM infrastructure. If the interoperability of a native C++ application requires the creation of an abstract factory, the details of the COM and IDL technologies should be thoroughly explored before proceeding with a custom solution.

Appendix B—Tracking-loop Control Theory

This section is not intended to be a comprehensive treatment of feedback control theory. Its purpose is to provide a summary of the relevant sections of references (57), (66), and (65), along with information gathered from multiple conversations with Dr. C.P. Diduch, to aid in the understanding of the control problems associated with the PLL and DLL carrier phase and code delay tracking.

Out of all the challenges encountered in software receiver design, the combined code and carrier tracking loop problem represents the area for the greatest system performance improvement opportunities. Most of the time, the PLL control model is explained in the literature using continuous analog functions that are represented in the s -domain, under the assumption that a suitable transform to the discrete z -domain exists. However, it is not always the case that such a transformation will provide a suitable basis for analysis and implementation.

In the continuous domain, the basic element is the integrator, whereas in the discrete domain the basic element is the time delay, or memory, of T_S seconds.

$$\int \Rightarrow \frac{1}{s} = s^{-1} \rightarrow \frac{z}{z-1} = \frac{1}{1-z^{-1}} \quad \mathbf{B-1}$$

The most often encountered transformations between the continuous and discrete domains are the forward difference, or Euler's method,

$$s = \frac{z - 1}{T_s} \quad \text{B-2}$$

The backward difference transformation,

$$s = \frac{z - 1}{zT_s} \quad \text{B-3}$$

And Tustin's approximation, the trapezoidal, or the bilinear transform,

$$s = \frac{2(z - 1)}{T_s(z + 1)} \quad \text{B-4}$$

The bilinear (Tustin's) is probably the most widely used, since it has the advantage of mapping the left half of the s-plane into the unit disc on the z-plane, which ensures that all stable continuous systems will result in stable discrete systems. All of these transformations create, however, a distortion of the frequency scale as given by B-5.

$$\omega = \frac{2}{T_s} \tan\left(\frac{\omega' T_s}{2}\right) \quad \text{B-5}$$

Where ω' is the unwarped frequency in the continuous-time system, and ω is the warped frequency in the discrete-time system.

For the case of the PLL, the NCO is modeled as shown in Figure B-1.

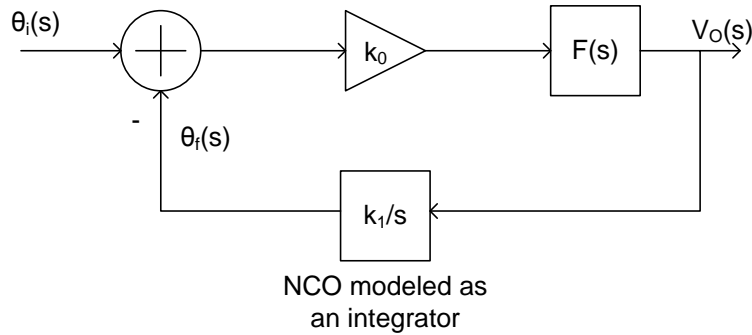


Figure B-1—PLL feedback control model

The usual 1st-order equivalent model for the NCO is the transfer function given in Equation B-6.

$$N(z) = \frac{\theta_f(z)}{V_o(z)} = \frac{k_1 z^{-1}}{1 - z^{-1}} \quad \text{B-6}$$

A close inspection of B-6 and comparing with the integrator model of B-1 reveals an additional z^{-1} term in the numerator. The existence of this term is explained by evaluating the model of the NCO, and including the effects of the sample and hold operation on the transfer function.

Firstly, the k_1/s element in Figure B-1 is the internal representation of the input signal characteristic modeled as a constant, such that the steady-state error goes to zero; if the signal/error is not a constant, the model is incorrect (more on this point momentarily) and the error will not be adequately removed. Secondly, the discrete sample-hold effects need to be considered, as shown in Figure B-2.

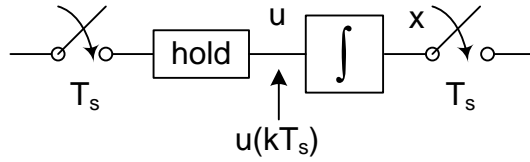


Figure B-2—Sample and hold representation

$$\dot{x}(t) = u(t)$$

B-7

$$= Ax(t) + Bu(t); A = 0, B = 1$$

$$x(kT_s + T_s) = e^{AT_s}x(kT_s) + \left(\int_0^{T_s} e^{An} dnB \right) u(kT_s)$$

B-8

$$x(kT_s + T_s) = x(kT_s) + T_s u(kT_s)$$

B-9

$$x(k + 1) = x(k) + T_s u(k)$$

B-10

$$zx = x + T_s u$$

B-11

$$x = \frac{T_s z^{-1}}{1 - z^{-1}} u$$

B-12

So, the discrete form of a sampled integrator given in equation B-12 is different from the relationship indicated by B-1. However, if all the work is performed and the state maintained internally to the sample and hold elements, then the extra z^{-1} term in the numerator represents an additional delay and isn't really necessary. Also, the model's plant is the filter and not the NCO, but this point will only matter if there is an external disturbance at the output that is not correctly fed-back to the compensator.

The loop filter, $F(s)$, shown in Figure B-1 is essentially an integrator and is intrinsically non-stable. To improve the stability requires a modification of the gain in the feedback element, as shown in Figure B-3.

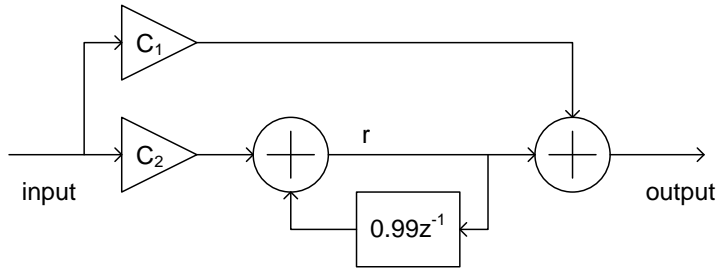


Figure B-3—PLL 1st-order filter

Implementing discrete filters in software requires a bit of manipulation of the desired transfer function. A transfer-function block diagram is shown in Figure B-4.

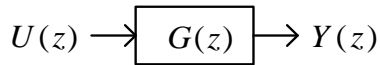


Figure B-4—A simple transfer function

In general, there can be multiple inputs and multiple outputs, so $U(z)$, $G(z)$, and $Y(z)$ may be state-variable vectors. For the linear time-invariant case,

$$G(z) \equiv \frac{Y(z)}{U(z)} = \frac{b_0 + b_1z^{-1} + b_2z^{-2} \dots}{a_0 + a_1z^{-1} + a_2z^{-2} \dots} \quad \text{B-13}$$

$$Y(z)[a_0 + a_1z^{-1} + a_2z^{-2} \dots] = U(z)[b_0 + b_1z^{-1} + b_2z^{-2} \dots] \quad \text{B-14}$$

Now, since

$$Y(z) \rightarrow Y(k) \quad \text{and,}$$

$$z^{-1}Y(z) \rightarrow Y(k - 1) \qquad zY(z) \rightarrow Y(k + 1)$$

$$z^{-2}Y(z) \rightarrow Y(k - 2) \qquad z^2Y(z) \rightarrow Y(k + 2)$$

So,

$$\begin{aligned} a_0Y(k) + a_1Y(k - 1) + a_2Y(k - 2) \dots \\ = b_0U(k) + b_1U(k - 1) + b_2U(k - 2) \dots \end{aligned} \qquad \mathbf{B-15}$$

$$\begin{aligned} Y(k) = \frac{1}{a_0} [-a_1Y(k - 1) - a_2Y(k - 2) \dots + b_0U(k) + b_1U(k - 1) \\ + b_2U(k - 2) \dots] \end{aligned} \qquad \mathbf{B-16}$$

The difference equation of B-16 gives the current output as a weighted sum of the current and past inputs as well as limited series of previous outputs. A minimal representation of an n^{th} -order system will contain only n delays. The prior inputs and outputs need to be persisted as state variables in software.

As each new input arrives, an updated output is calculated, and then, of course, there is a shift of last value = new value,

$$Y(k - 1) = Y(k)$$

$$Y(k - 2) = Y(k - 1)$$

$$U(k - 1) = U(k)$$

$$U(k - 2) = U(k - 1)$$

The code for implementing the filter of Figure B-3 is given in Figure B-5. The constants C1 and C2 are held in private class member variables that are calculated

whenever the filter characteristics (sampling rate, damping factor, natural frequency, gain) are changed.

```
public double Filter(double input) {  
    r = C2 * input + 0.99 * r;  
    output = C1 * input + r;  
  
    return output;  
}
```

Figure B-5—C# code for implementing the 1st-order filter of Figure B-3

The frequency response of the transfer function can be found by letting $z = e^{j\omega T_s}$ and computing the magnitude of the result for each value of ω .

Now, returning to Figure B-1 and Equation B-6. The objective of the PLL is to track small changes in carrier phase by computing the error angle through some form of a discriminator function. Two commonly used discriminators are shown plotted in Figure B-6 and Figure B-7, both of which have linear ramp characteristics over a small range of error input that crosses through the origin. Assuming that the acquisition stage of the GNSS receiver has provided the tracking loop with a good estimate of the initial carrier frequency (Doppler) and phase, the local carrier phase error will be close to zero and the 1st-order model of the discriminator error as a constant will be valid. If the phase error, for some reason, becomes large however, the discriminator error will be better modeled as 2nd-order ($1/s^2$), which results in a non-zero steady-state error. So, the PLL will quickly lose its ability to adequately track the dynamic phase changes and the signal will need to be reacquired.

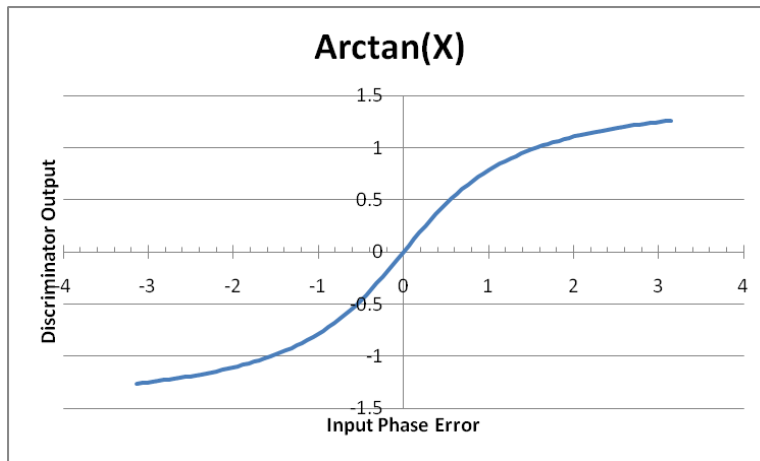


Figure B-6—Arctangent discriminator function over the range $[-\pi, \pi]$

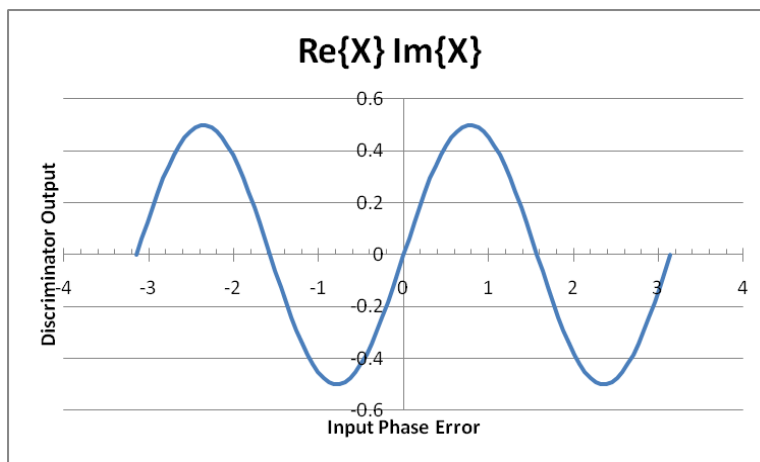


Figure B-7—Product discriminator function over the range $[-\pi, \pi]$

The *SiGe SE4110L-EK3 USB (61) Link-1 (L1)* receiver front-end used for testing the software Receiver Development Framework provides 3.96 samples per carrier cycle. A time-domain plot of the post-DLL code-removed carrier is shown in Figure B-8. The data has been plotted as straight lines with the actual sample points indicated by (blue) markers.

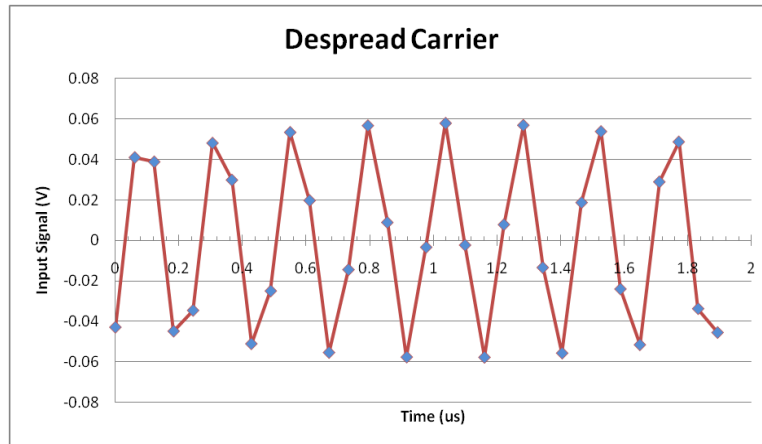


Figure B-8—Recovered carrier waveform, after code removal

At this sample rate, the phase error between samples will always be large and the discriminator function will not be operating in the linear region as desired on a sample-by-sample basis. It becomes necessary to integrate the error function over some interval of time in order to compute the average error, keeping the 1st-order approximation valid. This approach is suitable for post-processing applications, but potentially limits the ability of real-time implementations to accurately track the phase of the input signal.

More work is required to develop and test higher-order models that are capable of representing the PLL functions with greater accuracy. The potential to improve weak signal tracking performance, especially under high-dynamic (large Doppler-rate (74)) conditions, is great. Also, increasing the signal lock-in range of the PLL, reduces the precision and processing workload required of the initial acquisition stage.

Appendix C—Finite Fields and SSRGs

The intention of this appendix is to serve as an introduction to finite field theory as it applies to sequential shift register generators (SSRGs) as sources for pseudo-random noise (PRN) sequences that are used as spreading codes in direct-sequence-spread-spectrum systems. Much of this section is based on Chapter 2 from reference (4), however the text has several mistakes and mismatches in the equations and diagrams, as well as a few awkward language issues, that make it somewhat confusing to follow at times.

This appendix should serve to clarify a few of the points of confusion and provide a more substantive basis for system implementation. The goal is to remove some of the stumbling points, without creating new ones. The treatment of this material here is deliberately lacking in rigor and formality since much of that is provided by the referenced text, but the gist of the content is included for future use after the text has been returned to the library.

A finite field, or Galois field of q elements, denoted by $GF(q)$, has special properties and is defined in such a manner that there is only one way of constructing it. For any field, two operations exist, addition and multiplication, with their results always being in the field. The field must contain the additive and multiplicative identities (i.e. 0 and 1), and for every value in the field there must also exist the corresponding additive and multiplicative inverse. The usual associative, commutative, and distributive properties

must also apply. For their use in SSRGs we are concerned only with polynomials defined in the field of $GF(2)$.

A polynomial, $f(x)$, of degree n is irreducible, or prime, if it is not divisible by any polynomial of degree less than n but greater than zero. Like prime numbers, irreducible polynomials cannot be factored. Obviously, only odd polynomials, $x^n + \dots + 1$, can be irreducible, otherwise x would be a factor. Since the polynomial coefficients must be in $GF(2)$, not all decimal prime numbers correspond to irreducible polynomials—don't make that mistake.

An irreducible polynomial of degree n is primitive if it divides $x^m + 1$ for no m less than $2^n - 1$. Obtaining a maximal sequence length of $N = 2^n - 1$ for every n requires that the characteristic polynomial be primitive. There will always be one more one in the sequence than zero, since the all-zero state is not used in the output (it is terminal, resulting in no further changes in state.)

The various texts and articles on the subject of 0 and 1 binary valued transformations to the required +1 and -1 values differ and are usually incorrect. While an obvious choice would be to simply keep the 1 as +1 and shift the 0 down to the level of -1 (it would seem sensible enough), the only mapping between these value representations that preserves the equivalence of the exclusive-or and multiplication operations is that of

$$\{ 0, 1 \} \leftrightarrow \{ +1, -1 \}$$

Under this mapping, the results of an exclusive-or operation on two numbers is equivalent to multiplication, as show in Table C-1. The results column of both tables are equivalent when $0 \leftrightarrow +1$ and $1 \leftrightarrow -1$.

| X | Y | $X \oplus Y$ | X | Y | $X \times Y$ |
|-----|-----|--------------|-----|-----|--------------|
| 0 | 0 | 0 | +1 | +1 | +1 |
| 0 | 1 | 1 | +1 | -1 | -1 |
| 1 | 0 | 1 | -1 | +1 | -1 |
| 1 | 1 | 0 | -1 | -1 | +1 |

Table C-1—The equivalence between XOR and multiplication with {0, 1} ↔ {+1, -1} mapping

If the message data and PRN chipping/spreading code are mixed with the carrier as {0, 1} binary data, then the exclusive-or operation is required to spread the signal.

Otherwise, when {+1, -1} signals are used, the required operation is multiplication. The following function assumes that the binary data sequence, $d(t)$, and the spreading code, $PN(t)$, have been mapped to {+1, -1}

$$s(t) = \sqrt{P} \times d(t) \times PN(t) \times \cos(\omega_c t) \quad \text{C-1}$$

When binary {1, 0} values are used for the message data and chip code, they must be first modulo-2 added together before bi-phase modulating the carrier, like so:

$$s(t) = \sqrt{P} \cos(\omega_c t - [d(t) \oplus PN(t)] \times \pi) \quad \text{C-2}$$

Both functions behave in a similar manner in that they both produce a -180° phase shift in the carrier, ω_c , when the message data and spreading chip are different from each other, and a 0° phase shift when they are the same.

The function that maps binary data, $b(t)$, to {+1, -1}:

$$a(t) = \frac{1 - b(t)}{2} \quad \text{C-3}$$

Linear feedback shift register generators for PRN sequences are created by arranging feedback from various shift stages with taps defined according to a carefully selected polynomial. Take the following $GF(2)$ polynomial, $P(x)$, for example:

$$P(x) = x^{10} + x^9 + x^8 + x^6 + x^5 + 1 \quad \text{C-4}$$

This polynomial may be represented by the binary sequence [11101100001] and implemented as a linear feedback shift register by connecting the register cells corresponding to the powers of x to the feedback path through modulo-2 addition (XOR). This configuration is the Fibonacci form, shown in Figure C-1, and is characterized by feedback being taken from various taps in descending powers of x (from left to right) and applied to the most significant bit (MSB). The output is taken from the least significant bit (LSB).

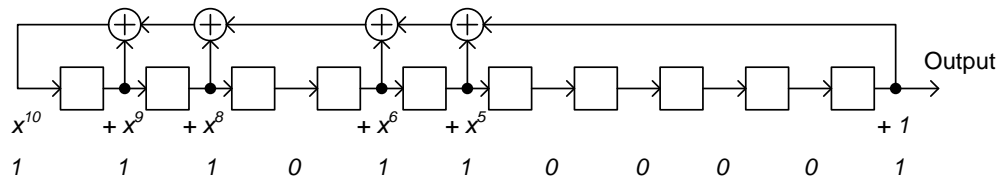


Figure C-1—Fibonacci implementation of $P(x)$

Starting with a single 1 in the right-most cell, the initial part of the produced sequence starts like:

1000000001110101001010001101101010111011110100011110101000001110...

An alternate configuration of the same polynomial is the Galois form, show in Figure C-2. In this form, the polynomial is written in reverse order, in increasing powers of x , and feedback is applied at multiple points along the length of the shift register.

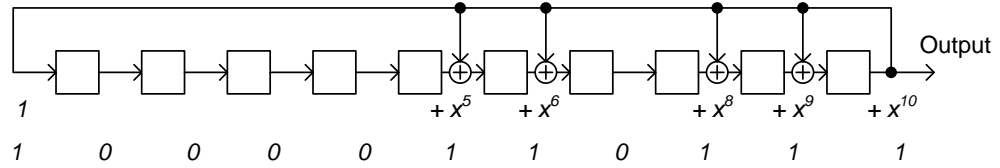


Figure C-2—Galois implementation of P(x)

The output is taken from the most significant bit (highest order term), and starting with a single 1 in the right-most cell the initial part of the sequence looks like:

1101010010100011011010101110111101000111101010000011101100110000...

It can be shown that for a given set of initial conditions, both shift register configurations are equivalent and will produce the same binary sequence, although they may differ in phase.

Now, for the part that can be confusing: The GNSS community uses a representation convention that is a hybrid of Fibonacci and Galois. As can be seen in Figure C-3, the feedback arrangement is that of Fibonacci, but the polynomial is written in increasing order as it is with Galois.

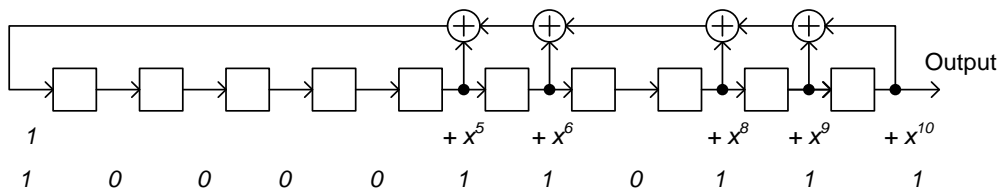


Figure C-3—GNSS implementation of P(x)

The sequence that results is the reverse sequence generated by a Fibonacci implementation, which is the same as using the reciprocal polynomial, $\frac{1}{P(x)}$. The

reciprocal of a primitive or irreducible polynomial is itself primitive or irreducible and will generate the reverse sequence.

The reciprocal of a polynomial can be evaluated using:

$$\frac{1}{G(x^N)} = x^N G\left(\frac{1}{x}\right) \quad \text{C-5}$$

So, for example, if

$$P(x) = x^{10} + x^9 + x^8 + x^6 + x^5 + 1 \quad \text{C-6}$$

then,

$$\frac{1}{P(x)} = x^{10}(x^{-10} + x^{-9} + x^{-8} + x^{-6} + x^{-5} + 1) \quad \text{C-7}$$

$$= x^{10} + x^5 + x^4 + x^2 + x + 1 \quad \text{C-8}$$

Implementing this polynomial as a feedback shift register using either the Fibonacci or Galois forms will result in the same output sequence as that of Figure C-3.

$GF(2)$ polynomial algebra is handled in a manner similar to traditional algebra, but keeping the coefficients in the field. Polynomials may be added and subtracted, or multiplied:

$$\begin{aligned} (x + 1)(x^3 + x^2 + 1) &= x^4 + x^3 + x + x^3 + x^2 + 1 \\ &= x^4 + x^2 + x + 1 \end{aligned} \quad \text{C-9}$$

Division can be conducted in the familiar long-method using pencil-and-paper, once again keeping the coefficients within the field.

$$\begin{array}{r}
 x^3 + x^2 + 1 \overline{) \begin{array}{r} x + 1 \\ x^4 + x^2 + x + 1 \\ \underline{x^4 + x^3 + x} \\ x^3 + x^2 + 1 \\ \underline{x^3 + x^2 + 1} \\ 0 \end{array}}
 \end{array}
 \tag{C-10}$$

The derivative of a GF(2) polynomial exists, with differentiation carried out in the usual way, except with the term coefficients kept within the field:

$$\begin{aligned}
 \frac{d}{dx}(x^N + x^{N-1} + x^{N-2} \dots x + 1) \\
 = Nx^{N-1} + (N-1)x^{N-2} + (N-2)x^{N-3} \dots + 1
 \end{aligned}
 \tag{C-11}$$

Where the coefficients $(N - k)$ are interpreted modulo-2. Now, since $(2 \bmod 2)$, $(4 \bmod 2)$, $(6 \bmod 2)$, and $(r \bmod 2) = 0$ {for r even}, even-order powers of x have a derivative of zero, while the derivative of odd-orders of x are x^{N-1} . An example,

$$\frac{d}{dx}(x^7 + x^6 + x^3 + x^2 + x + 1) = x^6 + x^2 + 1
 \tag{C-12}$$

Referring to Figure C-4, a generalized SSRG configuration model, expressions for the characteristic equation and characteristic polynomial can be obtained.

One would expect that the value immediately preceding a_0 would naturally be a_{-1} , however with the subscripts defined in the left-right manner above, the sequence would actually run $a_{-1}a_{-2}a_{-3} \dots a_{-n+1}a_{-n}a_0a_1a_2a_3 \dots$ such that the n^{th} stage's connection and initial condition are on the right, as shown in the diagram.

The sequence generating function is given by

$$G(x) = \sum_{k=0}^{\infty} a_k x^k \quad \text{C-17}$$

By combining equations C-16 and C-17, the output sequence can be determined as

$$G(x) = \sum_{i=1}^n c_i x^i \sum_{k=0}^n a_{k-i} x^{k-i} \quad \text{C-18}$$

Rearranging gives

$$G(x) = \frac{\sum_{i=1}^n c_i x^i [a_{-i} x^{-i} + a_{-i+1} x^{-i+1} + \dots + a_{-1} x^{-1}]}{\sum_{i=0}^n c_i x^i} = \frac{g(x)}{f(x)} \quad \text{C-19}$$

As an example, the shift register shown in Figure C-5 with feedback taps of:

$$c_0 = c_1 = c_3 = c_4 = 1, \text{ and } c_2 = 0$$

having initial conditions:

$$a_{-1} = a_{-2} = a_{-3} = 1, \text{ and } a_{-n} = 0$$

can be described by the polynomial:

$$f(x) = 1 + x + x^3 + x^4$$

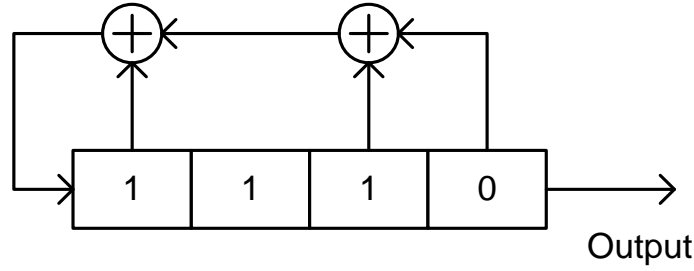


Figure C-5—Example shift register

The output sequence after the passing of the initial conditions can be found by evaluating equation C-20

$$\begin{aligned}
 g(x) &= c_1x(a_{-1}x^{-1}) + c_2x^2(a_{-2}x^{-2} + a_{-1}x^{-1}) \\
 &\quad + c_3x^3(a_{-3}x^{-3} + a_{-2}x^{-2} + a_{-1}x^{-1}) \\
 &\quad + c_4x^4(a_{-4}x^{-4} + a_{-3}x^{-3} + a_{-2}x^{-2} + a_{-1}x^{-1})
 \end{aligned}
 \tag{C-20}$$

$$g(x) = x(x^{-1}) + x^3(x^{-3} + x^{-2} + x^{-1}) + x^4(x^{-3} + x^{-2} + x^{-1})
 \tag{C-21}$$

$$g(x) = 1 + 1 + x + x^2 + x + x^2 + x^3
 \tag{C-22}$$

$$g(x) = x^3
 \tag{C-23}$$

By dividing $g(x)$ by $f(x)$ in ascending order, one obtains the following power-series in x :

| | | | | |
|---------------------|-------------------------|-------------------------|------------------------|-------------|
| | $a_0a_1a_2a_3$ | a_4 | $a_5a_6a_7a_8a_9\dots$ | |
| | 0001 | 1 | 10001 | |
| | $x^3 + x^4 + x^5 + x^9$ | | | |
| $1 + x + x^3 + x^4$ |) | x^3 | | |
| | | $x^3 + x^4 + x^6 + x^7$ | | |
| | | <hr/> | | |
| | | $x^4 + x^6 + x^7$ | | |
| | | $x^4 + x^5 + x^7 + x^8$ | | C-24 |
| | | <hr/> | | |
| | | $x^5 + x^6 + x^8$ | | |
| | | $x^5 + x^6 + x^8 + x^9$ | | |
| | | <hr/> | | |
| | | x^9 | | |

The coefficients of the result, $\{a_0a_1a_2 \dots\} = \{0001110001 \dots\}$, represent the output sequence after the initial conditions have passed.

The characteristic phase of an SSRG is the set of initial conditions that results in an output pattern such that if every second value in the sequence is sampled, the result is the same as the original output sequence. The characteristic phase of $f(x)$ can be found from

$$g(x) = f(x) + \frac{d[xf(x)]}{dx}, \quad \text{for } f(x) \text{ of even degree} \quad \text{C-25}$$

Or, from

$$g(x) = \frac{d[xf(x)]}{dx}, \quad \text{for } f(x) \text{ of odd degree} \quad \text{C-26}$$

The characteristic phase of the generator of Figure C-5 with $f(x) = 1 + x + x^3 + x^4$ can be found using equation C-25, since the polynomial is of even degree

$$g(x) = 1 + x + x^3 + x^4 + \frac{d}{dx}[x + x^2 + x^4 + x^5]$$

$$= x + x^3$$
C-27

Evaluating $G(x)$ in a similar manner to C-24 results in the following power series

$$G(x) = \frac{g(x)}{f(x)} = \frac{x + x^3}{1 + x + x^3 + x^4}$$
C-28

$$G(x) = 0 + 1x + 1x^2 + 0x^3 + 1x^4 + 1x^5 + 0x^6 + 1x^7 + 1x^8 + 0x^9 + 1x^{10} \dots$$

Which corresponds to an output sequence of [01101101101...] Starting the shift register of Figure C-5 with a left-to-right initial condition of [1010] and sampling the 2nd, 4th, 6th, 8th, and 10th bits in the output sequence yields [10110], which contains the same repeating pattern (underlined) as the first.

The correlation of two binary sequences can be found by performing a symbol-by-symbol sum for {0, 1} values or product for {+1, -1} values. In the case of the product, the correlation is the sum along the horizontal of the result. Dividing by the sequence length, N , gives the normalized correlation, that for a maximum length sequence will be either $-1/N$ or 1 since there is always one more -1 than +1. A useful trick for finding the correlation using XOR operations is to subtract the number of “ones” in the bitwise result from the number of “zeros.” An example that shows the calculation of the autocorrelation for a 7-bit sequence with successive bit delays is given in Table C-2.

| | $\sum 0's - \sum 1's$ | | $\Sigma =$ |
|--|-----------------------|--|------------|
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \hline 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$ | $7 - 0 = 7$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \hline 1\ 1\ 1\ 1\ 1\ 1\ 1 \end{array}$ | 7 |
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 1\ 0\ 0\ 1\ 0\ 1\ 1 \\ \hline 0\ 1\ 0\ 1\ 1\ 1\ 0 \end{array}$ | $3 - 4 = -1$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times -1\ 1\ 1\ -1\ 1\ -1\ -1 \\ \hline 1\ -1\ 1\ -1\ -1\ -1\ 1 \end{array}$ | -1 |
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 0\ 0\ 1\ 0\ 1\ 1\ 1 \\ \hline 1\ 1\ 1\ 0\ 0\ 1\ 0 \end{array}$ | $3 - 4 = -1$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times 1\ 1\ -1\ 1\ -1\ -1\ -1 \\ \hline 1\ -1\ 1\ -1\ -1\ -1\ 1 \end{array}$ | -1 |
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 0\ 1\ 0\ 1\ 1\ 1\ 0 \\ \hline 1\ 0\ 0\ 1\ 0\ 1\ 1 \end{array}$ | $3 - 4 = -1$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times 1\ -1\ 1\ -1\ -1\ -1\ 1 \\ \hline -1\ 1\ 1\ -1\ 1\ -1\ -1 \end{array}$ | -1 |
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 1\ 0\ 1\ 1\ 1\ 0\ 0 \\ \hline 0\ 1\ 1\ 1\ 0\ 0\ 1 \end{array}$ | $3 - 4 = -1$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times -1\ 1\ -1\ -1\ -1\ 1\ 1 \\ \hline 1\ -1\ -1\ -1\ 1\ 1\ -1 \end{array}$ | -1 |
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 0\ 1\ 1\ 1\ 0\ 0\ 1 \\ \hline 1\ 0\ 1\ 1\ 1\ 0\ 0 \end{array}$ | $3 - 4 = -1$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times 1\ -1\ -1\ -1\ 1\ 1\ -1 \\ \hline -1\ 1\ -1\ -1\ 1\ 1\ 1 \end{array}$ | -1 |
| $\begin{array}{r} 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ \oplus 1\ 1\ 1\ 0\ 0\ 1\ 0 \\ \hline 0\ 0\ 1\ 0\ 1\ 1\ 1 \end{array}$ | $3 - 4 = -1$ | $\begin{array}{r} -1\ -1\ 1\ 1\ -1\ 1\ -1 \\ \times -1\ -1\ -1\ 1\ 1\ -1\ 1 \\ \hline 1\ 1\ -1\ 1\ -1\ -1\ -1 \end{array}$ | -1 |

Table C-2—Example 7-bit autocorrelation calculations. Each row represents one additional bit delay.

The correlation result for each bit delay and a graph corresponding to these results is given in Table C-3 and Figure C-6. Note how the delayed autocorrelation function is periodic in the sequence length.

| Bit-shift amount | Correlation |
|------------------|-------------|
| 0 | 7 |
| 1 | -1 |
| 2 | -1 |
| 3 | -1 |
| 4 | -1 |
| 5 | -1 |
| 6 | -1 |
| 7 | 7 |

Table C-3—Correlation results

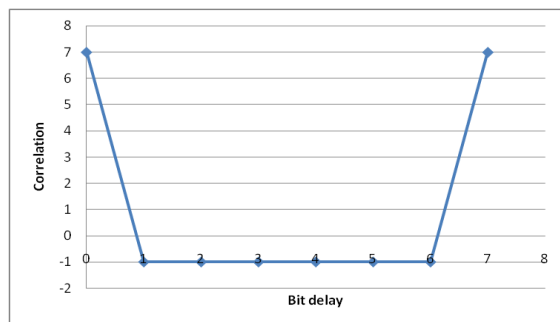


Figure C-6—Non-normalized autocorrelation

Not all sequences have these ideal autocorrelation characteristics. Although, the family of Gold codes are considered to be non-ideal sequences due to their imperfect autocorrelation properties, they are relatively easy to generate, there are many of them, and they are close enough to ideal for most applications. New sequences can be selected from existing generators by simply changing the positions of the polynomial taps. Figure C-7 shows a representative autocorrelation function for a length-31 Gold code.

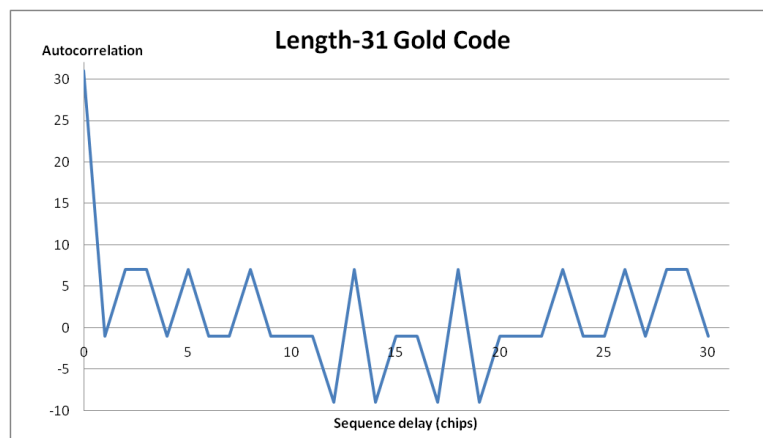


Figure C-7—Length-31 Gold code autocorrelation function

What follows is a complete list of all 1023-bits (chips) of each of the C/A codes for the GPS satellites. The generator configuration for these codes is shown in Figure C-8. The G1 and G2 generators are initialized to the all 1's condition, and the taps on the top of G2 are adjusted to determine the satellite ID. The values in the list are given in hexadecimal as arrays of 32-bit unsigned integers and include the transient pattern that results from the initial conditions. The bits are arranged starting with the left and moving to the right, such that the most significant bit is the first one in the sequence. Since there are 32 x 32-bit values, the listing actually provides 1024 bits in total, however the very last (least significant) bit of the very last value is unused and is always set to one (which

is also the same as the first bit in the sequence). These tables are used in the PRN Code Generator class, *GPSCACodeGenerator* (Section 6.2.9), to produce the early, prompt, and late code sequences.

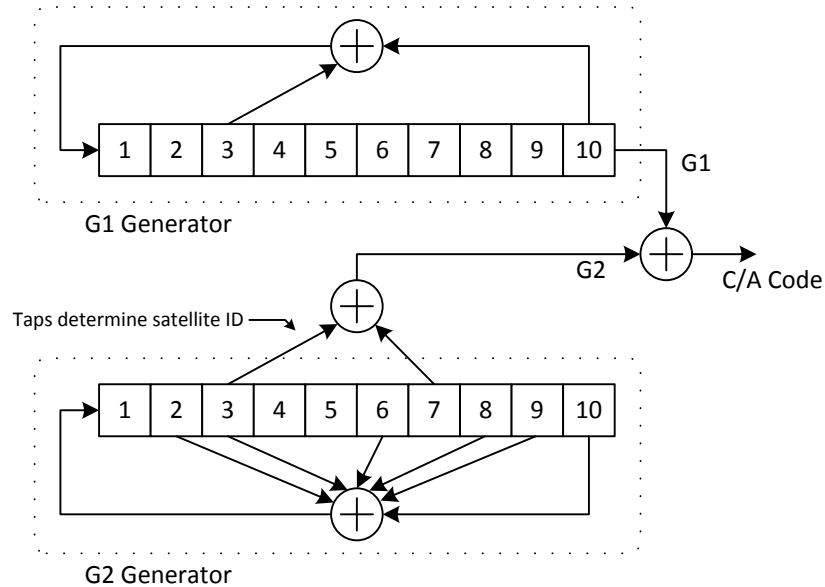


Figure C-8—GPS C/A-code generator configuration

A lookup table of C/A-code PRN sequences: Not just filler, these are hard to find in their complete form.

```
//Satellite 1: PRN 2 + 6
{ 0xC83949E5, 0x13EAD115, 0x591E9FB7, 0x37CAA100, 0xEA44DE0F, 0x5CCF602F, 0x3EA62DC6,
0xF5158201, 0x031D81C6, 0xFFA74B61, 0x56272DD8, 0xEEF0D864, 0x906D2DE2, 0xE0527E0A,
0xB9F5F331, 0xC6D56C6E, 0xE002CD9D, 0xA0ABAE94, 0x7389452D, 0x0ADAD8E7, 0xB21F9688,
0x7D5CC925, 0xFF87DE37, 0x2C3950A5, 0x7E3DA767, 0xEFA31F01, 0x28B444D8, 0x1DA3448E,
0x2CC9E6FC, 0xCA69AF36, 0xA778D442, 0x24E1CA21 }

//Satellite 2: PRN 3 + 7
{ 0xE4383E99, 0x6FCB2FF4, 0xB088B1E3, 0x06708E23, 0xA782D0D2, 0xF0E0F8DC, 0xC6B80F1A,
0x2E4666D3, 0x05E24A8C, 0x0AA09E09, 0x7FFBAB54, 0x152AA123, 0x21425370, 0x3954DC1E,
0xEC088B5C, 0xD4993B7D, 0x60979C2C, 0x346B025D, 0x8A5BA5B3, 0xE088AD28, 0xE9D923EB,
0xEF63AEA6, 0xF352EBC1, 0xEFB7A3E6, 0xDE216B43, 0xCCB30610, 0xFD67D33A, 0x8BDOCF A0,
0x9E997DCB, 0x69ECE796, 0xE66BEF69, 0x4B40A191 }

//Satellite 3: PRN 4 + 8
{ 0xF2388527, 0x51DBD084, 0x4443A6C9, 0x1EAD99B2, 0x0161D7BC, 0x26F734A5, 0x3AB71E74,
0x43EF94BA, 0x069DAF29, 0x702374BD, 0x6B15E812, 0x68C79D80, 0xF9D5EC39, 0x55D78D14,
0xC6F6376A, 0x5DBF10F4, 0xA0DD34F4, 0xFE0B5439, 0x76B2D5FC, 0x95A197CF, 0x443A795A,
0x267C1D67, 0x7538713A, 0x8E70DA47, 0x0E2F0D51, 0xDD3B0A98, 0x178E18CB, 0xC0E90A37,
0x7B13050, 0xB82E43C6, 0xC6E272FC, 0xFC901449 }
```

```

//Satellite 4: PRN 5 + 9
{ 0x9F38D8F8, 0x4ED3AF3C, 0x3E262D5C, 0x12C3127A, 0xD210540B, 0x4DFCD299, 0xC4B096C3,
0x753B6D8E, 0x87225DFB, 0xCD6281E7, 0x6162C9B1, 0x563103D1, 0x159E339D, 0xE3962591,
0xD3896971, 0x192C0530, 0x40F86098, 0x9B3B7F0B, 0x08C66DDB, 0x2F350ABC, 0x92CBD402,
0xC2F3C487, 0xB60D3C47, 0x3E936697, 0xE6283E58, 0xD5FF0CDC, 0x62FAFD33, 0x6575E8FC,
0x6B25169D, 0x50CF11EE, 0xD6A6BC36, 0x27784EA5 }

//Satellite 5: PRN 1 + 9
{ 0x96C46C57, 0x1EAEF9BE, 0xA3EE4593, 0xAF648601, 0xC0E7BD26, 0xDD96F3DC, 0x9B671012,
0x51FE1CB6, 0x8E7746E2, 0x81684D78, 0xFC4CC05F, 0xF3165F24, 0xA9F565A4, 0xC0605DA1,
0x1EDA361D, 0x21C3310A, 0x4134A2DA, 0xAA3AEE16, 0x55FF13F2, 0x480DB857, 0xB73A26778,
0xFAA74E83, 0xA73E8EF1, 0xC485DD92, 0x6606D1D7, 0x51C0D0E1, 0x503F37B5, 0x52300C95,
0xD3E4CD28, 0xD53C0F91, 0xD562E7AD, 0x4D04D4E5 }

//Satellite 6: PRN 2 + 10
{ 0xCB46AC40, 0x69693BA1, 0x4DF0DCF1, 0x4A279DA3, 0x32D36146, 0x304C3125, 0x145891F0,
0x7C33A988, 0xC357291E, 0x35C71D05, 0xAACE5D97, 0x9BD9E283, 0x3D8E7753, 0x294DCDCB,
0x3F9F69CA, 0xA71215CF, 0x300CAB8F, 0xB123A21C, 0x99608EDC, 0x41E31D70, 0xEB37DB13,
0xAC9E6D75, 0xDF0E43A2, 0x9BE9E57D, 0x523CD01B, 0x9382E1E0, 0xC1226A8C, 0x2C196BAD,
0x610FE821, 0x664637C5, 0x5F66F69E, 0xFFB22EF3 }

//Satellite 7: PRN 1 + 8
{ 0x967FD269, 0x0E51894A, 0x68F96F8B, 0x727317A7, 0x23E0D3F0, 0xCA5A8A20, 0x94767E7F,
0xF80C75B5, 0xF192E398, 0x0282F96C, 0x120F8622, 0x1E2AFCFC, 0x3E4A2CC8, 0x4331578B,
0xE0660094, 0x07E8B8CA, 0x0B9C7A10, 0xCA6C8AEA, 0xBC8F5C87, 0x61375FFA, 0x5498D6B1,
0xE5148F05, 0xCDA47590, 0x03FC7C42, 0x6860C3C6, 0xD9CC580B, 0xB9F4C6FE, 0x6BF59BCC,
0xFBA956F9, 0x17985FB1, 0x5CFF721A, 0x9DB10CC9 }

//Satellite 8: PRN 2 + 9
{ 0xCB1B735F, 0x611683DB, 0x287B49FD, 0x24AC5570, 0x4350D62D, 0x3BAA0DDB, 0x13D026C6,
0xA8CA9D09, 0x7CA5FBA3, 0x7432470F, 0xDDEFFEA9, 0x6D47B36F, 0x7651D3E5, 0x68E548DE,
0x40C1728E, 0x3407D12F, 0x1558C7EA, 0x81089062, 0xEDD8A966, 0xD57E6EA6, 0x1A9A83F78,
0x23478DB6, 0xEA433E12, 0x78553595, 0x550FD913, 0x5784A595, 0xB5C79229, 0xB0FBA001,
0xF52925C9, 0x87141FD5, 0x1BA83C45, 0x17E8C2E5 }

//Satellite 9: PRN 3 + 10
{ 0xE5A923C4, 0x56B50693, 0x883A5AC6, 0x0FC3F41B, 0xF308D4C3, 0xC3524E26, 0xD0030A9A,
0x00A9E957, 0x3A3E77BE, 0xCF6A183E, 0x3A1FC2EC, 0xD4F114A6, 0xD25C2C73, 0xFDF0F4774,
0x9092CB83, 0x2DF065DD, 0x9A3A9917, 0xA4BA9D26, 0xC5735396, 0x0F5AF608, 0x3D9BA954,
0x406E0CEf, 0x79B09BD3, 0x4581917E, 0xCBB85479, 0x90A0DB5A, 0xB3DE3842, 0x5D7CBDE7,
0x72691C51, 0xCF523FE7, 0x38039B6A, 0xD2C425F3 }

//Satellite 10: PRN 2 + 3
{ 0xD1289C36, 0x84084766, 0xD2302DE7, 0xACFD0285, 0xA2E49F1C, 0x67E40F8E, 0x551675C4,
0x1BED7A42, 0xFEDE52EA, 0xA30F2815, 0x0861B654, 0xF34B803B, 0xA18ADDDE, 0xA5EBC8AD,
0x7051FECE, 0x50408661, 0x4AD29E24, 0xADB25920, 0x81062773, 0xF7FF6AEA, 0xF6373D72,
0x8D86EDBD, 0x51A0DF1D, 0x8F5A7924, 0x27AE54C2, 0x2E9ECBA5, 0xC32AF355, 0x776460F0,
0xE3CFFF56, 0xA184282B, 0x41FF947B, 0xBCA98C01 }

//Satellite 11: PRN 3 + 4
{ 0xE8B0D470, 0xA43A64CD, 0x751FE8CB, 0x4BEB5FE1, 0x03D2F05B, 0x6D754F0C, 0x7360231B,
0x593A1AF2, 0xFB03A31A, 0x24F4AFB3, 0x50D8E692, 0x1BF70D0C, 0xB9B1AB6E, 0x1B88074D,
0x08DA8DA3, 0x1FD3CE7A, 0xB5FFB5F0, 0xB2E7F987, 0xF31C149C, 0x9E1A742E, 0x4BCD7616,
0x970EBCEA, 0xA4416B54, 0xBE063726, 0x72E89291, 0x2C2DEC42, 0x88A888FC, 0x3EB35D9F,
0xF91A711E, 0x5C1A2418, 0x15284F75, 0x87648281 }

//Satellite 12: PRN 5 + 6
{ 0xFA1AE242, 0x3C2FFDF2, 0x4F43FB16, 0x01A5E60A, 0x7B045C29, 0x2A99BF6D, 0xE9C69DC3,
0x28E47286, 0xF89A279E, 0x46F78D89, 0xEAAA1AC0, 0xD58668DA, 0xF3A2CD9A, 0x6B211345,
0x2ABDE8CE, 0xEBFEB871, 0xB5A26AEF, 0xBA9841FD, 0x96978190, 0xF091BCFD, 0x3A4EC17D,
0x9CE88014, 0xA3C9DC62, 0x6AFF03A7, 0xCDD1A402C, 0x6DD8B648, 0xFF892BC2, 0xC82D0C73,
0xB2C5D5A8, 0x1DB2A10D, 0x6A765431, 0x14714661 }

//Satellite 13: PRN 6 + 7
{ 0xFD29EB4A, 0xF829B987, 0x3BA603B3, 0x9D472DA6, 0xEF2291C1, 0xCBCB977D, 0xAD085718,
0xC0BE9E90, 0xF82199A0, 0x5608FD7D, 0x21BD30D8, 0x0891F97C, 0x10A5A34C, 0x7CED6AB9,
0x25AC86A3, 0x420CD172, 0xCA47CF95, 0x3972F5E9, 0x78D4C7ED, 0x1DAD1F25, 0xADF18811,
0x1FB98A3E, 0x5D75EAE8, 0x4CD48A67, 0x87B298E6, 0x0D8ED2B4, 0x16F964B7, 0xE117EBDE,
0x519F6461, 0x0201608B, 0x00ECAF50, 0xD308E7B1 }

```

```

//Satellite 14: PRN 7 + 8
{ 0xFEB06FCE, 0x9A2A9BBD, 0x81D4FFE1, 0x53364870, 0xA531F735, 0xBB628375, 0x8F6F3275,
0x3493E89B, 0xF87C46BF, 0x5E774507, 0x4436A5D4, 0x661A31AF, 0x61261427, 0x770B5647,
0x22243195, 0x96F5E5F3, 0x75B51D28, 0x7887AFE3, 0x0FF564D3, 0xEB334EC9, 0xE62E2CA7,
0x5E110F2B, 0x222BF1AF, 0DFC14E87, 0xA2E6F483, 0x3DA5E0CA, 0x6241430D, 0x758A9808,
0xA0323C85, 0x8DD88048, 0x35A1D2E0, 0x30B43759 }

//Satellite 15: PRN 8 + 9
{ 0xFF7CAD8C, 0xAB2B0AA0, 0xDCED81C8, 0x340EFA9B, 0x8038444F, 0x83360971, 0x9E5C80C3,
0xCE85539E, 0x7852A930, 0xDA48993A, 0x76F36F52, 0x515FD5C6, 0xD9E7CF92, 0xF2F84838,
0x21E06A0E, 0xFC897FB3, 0xAA4C7476, 0xD87D02E6, 0x3465B54C, 0x907C663F, 0xC3C1FEFC,
0x7EC54DA1, 0x9D84FC0D, 0x964BACF7, 0xB04CC2B1, 0xA5B079F5, 0x581D50D0, 0x3FC421E3,
0xD8E490F7, 0xCA347029, 0xAF076C38, 0x416A5F2D }

//Satellite 16: PRN 9 + 10
{ 0xFF9ACCAD, 0xB3ABC22E, 0x72713EDC, 0x8792A3EE, 0x12BC9DF2, 0x9F1C4C73, 0x96C55998,
0xB38E0E1C, 0xB845DEF7, 0x18577724, 0xEF918A11, 0x4AFD27F2, 0x05872248, 0x3001C707,
0xA00247C3, 0x49B73293, 0xC5B0C0D9, 0x88005464, 0xA9ADDD83, 0x2DDBF244, 0xD13617D1,
0xEEAF6CE4, 0xC2537ADC, 0xB2EDDCF, 0xB919D9A8, 0xE9BAB56A, 0xC533593E, 0x9AE37D16,
0x648FC6CE, 0xE9C20819, 0x62543354, 0x79856B17 }

//Satellite 17: PRN 1 + 4
{ 0x9B8044FC, 0xE45E239A, 0x3B406292, 0x85C7E528, 0x41BE2ED5, 0x7857CE08, 0x3F8C8EA5,
0xDC94DB92, 0xF0B840FB, 0x2B03A0FF, 0xE1AA471F, 0xCABE1762, 0x89C7460F, 0x674F988D,
0xF9CC6B79, 0x80F55E4D, 0x4BA5E258, 0x8C4CB8C9, 0x17287342, 0x4DD049A7, 0x3039E0DE,
0xA21E1E45, 0x4F8203C6, 0DCBEAB22, 0xD8651E37, 0x294BA38C, 0x1FAC7FAE, 0xAD1D2741,
0xCCB16D8F, 0xA7263C7E, 0xBC87F969, 0xF0FE9F81 }

//Satellite 18: PRN 2 + 5
{ 0xCDE4B815, 0x941156B3, 0x01A7CF71, 0xDF762C37, 0xF27FA8BF, 0xE2ACAFCF, 0x462D5EAB,
0xBA86CA1A, 0xFC30AA12, 0xE0F2EBC6, 0x243D1E37, 0x8715C6A0, 0x2D976686, 0xFADA2F5D,
0x4C144778, 0xF789226C, 0xB5440BCE, 0xA2188973, 0x380B3E84, 0x430DE588, 0xA8CA1F8C,
0x80C2C516, 0xAB500539, 0x17F45E25, 0x0D0D37EB, 0xAFC75856, 0x66EBCE81, 0xD38FFE47,
0x6EA53872, 0xDF4B2E32, 0xEB9479FC, 0xA14F0B41 }

//Satellite 19: PRN 3 + 6
{ 0xE6D6C661, 0x2C36EC27, 0x9CD41980, 0x722EC8B8, 0x2B9F6B8A, 0xAFD11F2C, 0xFAFDB6AC,
0x898FC2DE, 0xFA74DF66, 0x050A4E5A, 0xC6F6B2A3, 0xA1D82E41, 0x7FBF76C2, 0x3410F4B5,
0x16F85178, 0x4C371C7C, 0x4A34FF05, 0xB53291AE, 0x2F9A9867, 0x4463339F, 0x64B3E4CF,
0x91ACAB8F, 0x59390646, 0xF25124A6, 0xE7B92305, 0xEC8125BB, 0x5A481616, 0x6CC692C4,
0x3FAF128C, 0x637DA714, 0xC01DB9B6, 0x0997C121 }

//Satellite 20: PRN 4 + 7
{ 0xF34FF95B, 0x7025316D, 0xD26DF2F8, 0xA482BAFF, 0xC76F0A10, 0x096FC75D, 0x2495C2AF,
0x100B46BC, 0xF956E5DC, 0x77F61C94, 0xB79364E9, 0xB2BEDA31, 0xD6AB7EE0, 0x53759941,
0x3B8E5A78, 0x11E80374, 0x358C8560, 0x3EA79DC0, 0xA4524B16, 0xC7D45894, 0x828F1AC8,
0x191B9E6B, 0xA00D87F9, 0x008399E7, 0x12E32972, 0xCD221B4D, 0xC419FA5D, 0xB3622485,
0x972A07F3, 0x3D66E387, 0xD5D95993, 0x5DFBA411 }

//Satellite 21: PRN 5 + 8
{ 0xF98366C6, 0x5E2CDFC8, 0xF5310744, 0xCFD483DC, 0x31173ADD, 0x5A30AB65, 0xCBA1F8AE,
0xDCC9048D, 0xF8C7F881, 0x4E8835F3, 0x8F218FCC, 0xBB0DA009, 0x82217AF1, 0x60C72FBB,
0x2D355FF8, 0x3F078CF0, 0x0A50B852, 0xFB6D1BF7, 0xE1B622AE, 0x060FED11, 0x719165CB,
0xDD400501, 0xDC97C726, 0xF9EAC747, 0xE84E2C49, 0x5DF38436, 0x8B310C78, 0x5CB07FA5,
0x43688D4C, 0x926B41CE, 0x5F3B2981, 0xF7CD9689 }

//Satellite 22: PRN 6 + 9
{ 0xFCE52908, 0xC928289A, 0x669F7D9A, 0xFA7F9F4D, 0xCA2B22BB, 0xF39F1D79, 0xBC3BE5AE,
0x3AA82595, 0x780F762F, 0xD2372140, 0x1378FA5E, 0x3FD41D15, 0xA86478F9, 0xF91E74C6,
0x2668DD38, 0x28704B32, 0x15BEA6CB, 0x998858EC, 0x43441672, 0x66E237D3, 0x881E5A4A,
0x3F6DC8B4, 0xE2DAE749, 0x055E6817, 0x9518AED4, 0x959B4B8B, 0x2CA5776A, 0xAB595235,
0x2949C813, 0x45ED90EA, 0x9A4A1188, 0xA2D68FC5 }

//Satellite 23: PRN 1 + 3
{ 0x8CF7833E, 0xFBB03D03, 0x59A52189, 0x2735D1F4, 0x2153F417, 0x81D8F189, 0xDDA14310,
0xE2D9FBFD, 0x0C0CEFAB, 0x56552262, 0x29C288A2, 0x6D1A6C70, 0x7E2E6B9F, 0x0D6EDDD2,
0x2E4ABA5D, 0x45846644, 0x1EBEFB14, 0x86802754, 0x39219DE7, 0x6A8CBC1B, 0x5B6FD9FD,
0x54662E88, 0x1CDD6FFE, 0x338A9123, 0x14A75C06, 0x28DABED1, 0x26D256C9, 0x95AFCC64,
0xC50217B7, 0xF3AC386F, 0x8F354F93, 0xE6459A01 }

```

```

//Satellite 24: PRN 4 + 6
{ 0xF1A101A3, 0x33D8F2BE, 0xFE315A9B, 0xD0DCFC64, 0x4B72B148, 0x565E20AD, 0x18D07B19,
0xB7C2E2B1, 0x06C07036, 0x785CCCC7, 0x0E9E7D1E, 0x064C5553, 0x88565B52, 0x5E31B1EA,
0xC17E805C, 0x89462475, 0x1F2FE649, 0xBFFE0E33, 0x019376C2, 0x633FC623, 0x0FE5DDEC,
0x67D49B72, 0x0A666A7E, 0x1D651EA7, 0x2B7B6134, 0xED1038E6, 0x63363F71, 0x547479E1,
0x361C68B4, 0x37F7A305, 0xF3AF0F4C, 0x1F2CC4A1 }

//Satellite 25: PRN 5 + 7
{ 0xF8F41ABA, 0x7FD23E21, 0x631F5375, 0x75FBA091, 0xF719E771, 0x75A8589D, 0xD5832475,
0x8F2DD68B, 0x070CB274, 0x495D5DDA, 0x53A70337, 0x6174E7B8, 0xAD5FE828, 0x66653BEE,
0xD04D32EA, 0x73509F70, 0x9F0109C6, 0x3BC1D20E, 0x3356BC44, 0x547A224A, 0xB7240659,
0xE227860D, 0x09A231E5, 0x771984E7, 0xF482086A, 0x4DEA95E3, 0x58A6EEEE, 0x2F3B5117,
0x13F3BAEF, 0x1723E18F, 0x4C0002EE, 0x56A626D1 }

//Satellite 26: PRN 6 + 8
{ 0xFC5E9736, 0xD9D7586E, 0xAD885782, 0x27680EEB, 0x292C4C6D, 0xE4536485, 0xB32A8BC3,
0x935A4C96, 0x07EAD355, 0x51DD9554, 0xFD3BBC23, 0xD2E8BECD, 0x3FDB3195, 0x7A4F7EEC,
0xD8D4EBB1, 0x0E5BC2F2, 0x5F167E01, 0xF9DE3C10, 0xAA345907, 0x4FD8D07E, 0x6B44EB83,
0x20DE0932, 0x88401C28, 0xC227C9C7, 0x9B7EBCC5, 0x1D97C361, 0xC56E8621, 0x929CC56C,
0x010453C2, 0x8749C0CA, 0x13D7843F, 0x726357E9 }

//Satellite 27: PRN 7 + 9
{ 0xFE0BD1F0, 0x8AD5EB49, 0x4AC3D5F9, 0x8E21D9D6, 0x463699E3, 0xACAFA89, 0x807E5C18,
0xD9D618198, 0x8799E3C5, 0xDD9DF113, 0xAA75E3A9, 0x8B269277, 0xF6995D4B, 0xF45A5C6D,
0xDC98071C, 0xB0DE6C33, 0x3F1DC5E2, 0x18D1CB1F, 0xE6852BA6, 0xC209A964, 0x05749D6E,
0x41A2CEAD, 0x48B10ACE, 0x18B8EF57, 0xAC80E692, 0xB5A96820, 0x8B8AB246, 0x4C4F0F51,
0x887FA754, 0x4F7CD068, 0xBC3C4757, 0xE001EF75 }

//Satellite 28: PRN 8 + 10
{ 0xFF217293, 0xA354B2DA, 0xB96614C4, 0x5A853248, 0xF1BBF324, 0x88D0358F, 0x99D437F5,
0x1A7C671F, 0xC7A07B8D, 0x9BBD330, 0x01D2CC6C, 0xA7C1842A, 0x92386B24, 0xB350CD2D,
0x5EBE714A, 0x6F9CBB53, 0x8F181813, 0xE8563098, 0x40DD92F6, 0x04E115E9, 0x326CA618,
0xF11CAD62, 0xA8C981BD, 0x75F77C1F, 0xB77FCBB9, 0x61B63D80, 0x2CF8A875, 0xA326EA4F,
0x4CC25D1F, 0x2B665839, 0xEBC9A6E3, 0xA930B33B }

//Satellite 29: PRN 1 + 6
{ 0x95E656ED, 0x6C52AB70, 0xD28B93D9, 0xBC027271, 0x69F3B504, 0xBAF39E28, 0xB6111B12,
0x0C2103BE, 0xF1CF3C87, 0x0AFD4116, 0x7784132E, 0x70A1342F, 0x4FC99BA3, 0x48D76B75,
0xE7EEB7A2, 0xD3118C4B, 0xB46EA8AD, 0x8B99D0E0, 0xCBAAFFB9, 0x97A90E16, 0x1F477207,
0xA4BC0A10, 0xB2FA6ED4, 0x90E9B8A2, 0x4D34AFA3, 0xE9E76A75, 0xCD4CE144, 0xFF68E81A,
0x0A040E1D, 0x9841BF72, 0x69B20FAA, 0x7E0DDC21 }

//Satellite 30: PRN 2 + 7
{ 0xCAD7B11D, 0x501712C6, 0x754237D4, 0x4394E79B, 0x66596557, 0x03FE87DF, 0x02E39470,
0x52DC260C, 0xFC8B142C, 0xF00D9B32, 0xEF2A342F, 0x5A025706, 0xCE900850, 0xED1656A1,
0x43052915, 0x5E7B4B6F, 0xCAA1AEB4, 0x21F23D67, 0xD64878F9, 0xAE314650, 0x3F7551AC,
0x0393CF3C, 0x55EC33B0, 0x31DFD7E5, 0x47A5EF21, 0xCF913CAA, 0x8F9B81F4, 0xFAB519EA,
0x8DFF89BB, 0xC0F8EFB4, 0x810E829D, 0x6636AA91 }

//Satellite 31: PRN 3 + 8
{ 0xE54F42E5, 0x4E35CE1D, 0x26A6E5D2, 0xBC5FAD6E, 0x618C0D7E, 0xDF780B24, 0xD89AD3C1,
0x7DA2B4D5, 0xFA290079, 0x0D75F620, 0xA37D27AF, 0xCF53E692, 0x0E3CC1A9, 0x3FF6C84B,
0x1170E64E, 0x98CE28FD, 0xF5C62DB8, 0xF4C7CBA4, 0x58BB3B59, 0xB2FD6273, 0x2F6C4079,
0xD0042DAA, 0x26671D02, 0x6144E046, 0xC2ED4F60, 0xDCAA17C5, 0x2EF031AC, 0xF85BE112,
0xCE024A68, 0xECA447D7, 0xF550C406, 0xEA2B11C9 }

//Satellite 32: PRN 4 + 9
{ 0xF2833B19, 0x4124A070, 0x8F548CD1, 0xC3BA0814, 0xE266B96A, 0x313B4D59, 0x35A67019,
0xEA1DFDB9, 0x79780A53, 0xF3C9C0A9, 0x8556AE6F, 0x85FB3E58, 0x6E6AA555, 0xD686873E,
0x384A01E3, 0x7B949934, 0xEA75EC3E, 0x9E5D30C5, 0x9FC29A89, 0xBC9B7062, 0xA760C893,
0x39CFDCCE1, 0x1FA28A5B, 0x49097B97, 0x00491F40, 0x55378272, 0xFE45E980, 0xF92C9D6E,
0xEFFCAB81, 0x7A8A13E6, 0x4F7FE74B, 0x2C25CC65 }

```

Appendix D—Background Signals Theory

This appendix provides, without much explanation, a few of the more useful expressions and operations of basic signal theory. Most of this information can be found in references (75), (60), (76), (77), and (78). Reading it may feel like a shotgun blast.

The Fourier transform and its discrete equivalent are given by,

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{2\pi jft} dt \approx H_n = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi jkn / N} \quad \mathbf{D-1}$$

And the inverse is given by,

$$h(t) = \int_{-\infty}^{\infty} H(f)e^{-2\pi jft} dt \approx h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi jkn / N} \quad \mathbf{D-2}$$

Where,

$$\Delta = \text{the sample interval, } T_s = \frac{1}{f_s}$$

$$h_k \equiv h(t_k), \text{ the sample value at time } t_k$$

$$t_k \equiv k\Delta, \text{ the time at the } k^{\text{th}} \text{ sample, } k = 0, 1, 2, \dots, N - 1$$

$$H(f_n) \cong \Delta H_n \text{ with } fn \equiv \frac{n}{\Delta N}; n = -\frac{N}{2} \dots \frac{N}{2}$$

The Nyquist frequency is given by

$$f_c = \frac{1}{2\Delta} = \frac{f_s}{2}$$

D-3

Table D-1 provides a summary of the relationships between the time and frequency domains.

| If: | Then: |
|------------------------------|---|
| $h(t)$ is real | $H(-f) = H(f)^*$ |
| $h(t)$ is imaginary | $H(-f) = -H(f)^*$ |
| $h(t)$ is even | $H(-f) = H(f) \Rightarrow H(f)$ is even |
| $h(t)$ is odd | $H(-f) = -H(f) \Rightarrow H(f)$ is odd |
| $h(t)$ is real and even | $H(f)$ is real and even |
| $h(t)$ is real and odd | $H(f)$ is imaginary and odd |
| $h(t)$ is imaginary and even | $H(f)$ is imaginary and even |
| $h(t)$ is imaginary and odd | $H(f)$ is real and odd |

Table D-1—Relationships between the properties of the time and frequency domains

Some other useful relations are given in equations D-4 to D-7.

Time scaling

$$h(at) \leftrightarrow \frac{1}{|a|} H\left(\frac{f}{a}\right) \quad \mathbf{D-4}$$

Frequency scaling

$$\frac{1}{|b|} h\left(\frac{t}{b}\right) \leftrightarrow H(bf) \quad \mathbf{D-5}$$

Time shifting

$$h(t - t_0) \leftrightarrow H(f) e^{2\pi j f t_0} \quad \mathbf{D-6}$$

Frequency shifting

$$h(t) e^{-2\pi j f_0 t} \leftrightarrow H(f - f_0) \quad \mathbf{D-7}$$

Parseval's Theorem equates a signal's total energy in the time domain to the frequency domain.

$$\begin{aligned} \text{Total Power} &\equiv \int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |H(f)|^2 df \\ &\leftrightarrow \sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \end{aligned} \quad \mathbf{D-8}$$

Convolution means to smear out, or to spread a signal's energy across a specified response function. It can be found from:

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau)h(t - \tau)d\tau \quad \text{D-9}$$

Convolution in the time domain corresponds to multiplication in the frequency domain,

$$g * h \leftrightarrow G(f)H(f) \quad \text{D-10}$$

For the discrete form, the signal, $s(t)$, is represented by its samples, s_j , and the response, $r(t)$, is represented by samples r_k , as in D-11,

$$(r * s)_j \equiv \sum_{k=-\frac{M}{2}+1}^{\frac{M}{2}} s_{j-k}r_k \Leftrightarrow S_n R_n \quad \text{D-11}$$

M is some sample interval; typically M would equal N in the case of s_j being periodic in N. But, there are potential problems:

- The input signal is not always periodic
- The duration of the response does not equal the period of the data (N)

The length (duration) of the response is typically shorter than the length of the data set (i.e. $M < N$). Therefore, data must be padded on one end with zeros equal to the larger of the positive or negative duration of the response function, such that $M = N$.

Correlation calculates the extent to which one signal can be related to another. Also called the lag, it is the close cousin of convolution.

$$\text{Corr}(g, h)(t) = \text{Corr}(h, g)(-t) \quad \text{D-12}$$

$$\text{Corr}(g, h) \leftrightarrow G(f)H(-f) \quad \text{D-13}$$

If g, h are real, then,

$$H(-f) = H(f)^* \quad \text{D-14}$$

So,

$$\text{Corr}(g, h) \leftrightarrow G(f)H(f)^* \quad \text{D-15}$$

And, the autocorrelation,

$$\text{Corr}(g, g) \leftrightarrow |G(f)|^2 \quad \text{D-16}$$

Or, in its discrete form,

$$\text{Corr}(g, h)_j \equiv \sum_{k=0}^{N-1} g_{j+k} h_k \quad \text{D-17}$$

The DFT can be used to estimate the power spectral density (PSD) of a signal. For the one-sided PSD,

$$P_h(f) \equiv |H(f)|^2 + |H(-f)|^2 \quad 0 \leq f < \infty \quad \text{D-18}$$

If $h(t)$ is real, then

$$|H(-f)|^2 = |H(f)|^2 \quad \text{D-19}$$

So,

$$P_h = 2|H(f)|^2 \quad \text{D-20}$$

Since, D-21 is periodic in n ,

$$H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi jkn / N} \quad \text{D-21}$$

with period N , then n can vary between 0 to $N - 1$ (instead of $-\frac{M}{2}$ to $\frac{M}{2}$ as in Equation D-11), which is useful for zero-based arrays. The DFT, or FFT, then produces a periodogram such that:

The positive frequencies are located at $(0 < f < f_c) \rightarrow 1 \leq n \leq \frac{N}{2} - 1$

The negative frequencies are located at $(-f_c < f < 0) \rightarrow \frac{N}{2} + 1 \leq n \leq N - 1$

And, the most positive and negative frequencies ($f = f_c$ and $f = -f_c$) $\rightarrow n = \frac{N}{2}$

In order to get things to scale correctly when plotted,

$$P(0) = P(f_0) = \frac{1}{N^2} |C_0|^2 \quad \text{D-22}$$

$$P(f_k) = \frac{1}{N^2} [|C_k|^2 + |C_{N-k}|^2]; \quad k = 1, 2, \dots, \left(\frac{N}{2} - 1\right) \quad \text{D-23}$$

$$P(f_c) = P\left(f_{\frac{N}{2}}\right) = \frac{1}{N^2} \left|C_{\frac{N}{2}}\right|^2 \quad \text{D-24}$$

Note how the endpoints get special treatment; this is because these bins are either counted twice (most positive and most negative f overlap) or are only half as wide (at

$$f = 0). \text{ For } C(t) \text{ real, } |C_k|^2 = \left|C_{\frac{N}{2}}\right|^2$$

C/N or SNR is the ratio of signal carrier power to the white-noise power in a specified bandwidth (dB). C/N_0 is the ratio of signal carrier power to the white-noise power spectral density in a 1-Hz bandwidth (dB-Hz).

To convert C/N_0 to C/N , divide by the bandwidth,

$$C/N = C/N_0 B \quad \text{D-25}$$

Or, in dB, simply subtract $10 \log(B)$ where B is the bandwidth in Hz.

References

1. **Press, William H., et al.** *Numerical Recipes in C: The Art of Scientific Computing* 2nd. Ed. New York, NY : Cambridge University Press, 1992. 0-521-43108-5.
2. **Zhuang, W.** *Composite GPS Receiver Modeling, Simulations, and Applications (Ph.D. Dissertation)*. Fredericton, NB : University of New Brunswick, 1992. Thesis 5008.
3. **Misra, Pratap and Enge, Per.** *Global Positioning System: Signals, Measurements, and Performance, 2nd Ed.* Lincoln, MA : Ganga-Jamuna Press, 2006.
4. **Holmes, Jack K.** *Spread Spectrum Systems for GNSS and Wireless Communications*. Norwood, MA : Artech House, Inc., 2007.
5. **Borre, K., et al.** *A Software-Defined GPS and Galileo Receiver: Single-Frequency Approach*. Boston, MA : Birkhäuser, 2007.
6. **Tsui, James Bao-Yen.** *Fundamentals of Global Positioning System Receivers: A Software Approach*. New York : John Wiley & Sons, Inc., 2000.
7. **Ziedan, N.I.** *GNSS Receivers for Weak Signals*. Norwood, MA : Artech House, Inc., 2006.
8. *Development of the Open Source GPS Software Receiver Emulator*. **Kelley, C.W., Niles, F. and Baker, D.** Fort Worth, TX : The Institute of Navigation, September 2007. ION GNSS 20th International Technical Meeting of the Satellite Division.

9. *Creating a GPS Receiver from Free Software Components*. **Danielsen, T.** Fort Worth, TX : The Institute of Navigation, September 2007. ION GNSS 20th International Technical Meeting of the Satellite Division.
10. *FPGA-Based Architecture for High Throughput, Flexible and Compact Real-Time GNSS Software Defined Receiver*. **Sauriol, B. and Landry, R. Jr.** San Diego, CA : The Institute of Navigation, 22-24 January 2007. ION NTM 2007.
11. *Implementation and Testing of a Real-Time Software-Based GPS Receiver for x86 Processors*. **Charkhandeh, S., et al.** Monterey, CA : The Institute of Navigation, January 18-20 2006. ION NTM 2006.
12. *Improvements to 'A Software-Defined GPS and Galileo Receiver: Single-Frequency Approach'*. **Vinande, E. and Akos, D.** Fort Worth, TX : The Institute of Navigation, 25-28 September 2007. ION GNSS 2007.
13. **Krumvieda, K., et al.** A Complete IF Software GPS Receiver: A Tutorial about the Details. [Online] Data Fusion Corporation. [Cited: February 12, 2010.] <http://www.datafusion.com/gps/baselinereceiver.pdf>.
14. *Development of a One Channel Galileo L1 Software Receiver and Testing Using Real Data*. **Macchi, F. and Petovello, M.G.** Fort Worth, TX : The Institute of Navigation, 25-28 September 2007. ION GNSS 2007.
15. *Performance Evaluation of a GPS L5 Software Receiver Using a Hardware Simulator*. **Mongrédién, C., Cannon, M. E. and Lachapelle, G.** Geneva,

- Switzerland : European Group of Institutes of Navigation, 29 May-1 June 2007. ENC '07.
16. *Software Defined Radios: A Software GPS Receiver Example*. **Sharawi, M. S. and Korniyenko, O. V.** Amman, Jordan : IEEE/ACS, 13-16 May 2007. International Conference on Computer Systems and Applications, AICCSA '07.
17. *Integrated GPS/TOA Navigation using a Positioning and Communication Software Defined Radio*. **Brown, A. and Nordlie, J.** San Diego, CA : IEEE/ION, April 25-27 2006. Position, Location, And Navigation Symposium 2006.
18. **Dovis, F., Gramazio, A. and Mulassano, P.** Design and test-bed implementation of a reconfigurable receiver for navigation applications. *Wireless Communications and Mobile Computing*. 2002, 2, pp. 827–838.
19. *Discrete-time Phase and Delay Locked Loops Analyses in Tracking Mode*. **Sebesta, J.** Paris, France : World Academy of Science, Engineering, and Technology, Fall 2007, International Journal of Electronics, Circuits and Systems, Vols. 1, 4.
20. **Vassiliadou-Christoo, V.** *GPS Series Single Difference Observations Analysis and Software Developmen (M.Sc.E. Thesis)*. Fredericton, NB : University of New Brunswick, 1989. Thesis 4470.
21. **Pany, T.** Status of Software Receiver Technology at University FAF Munich and IFEN GmbH. *IGS Workshop 2008, Miami Beach, FL*. [Online] Institute of Geodesy and Navigation, 2008. [Cited: February 12, 2010.]
<http://www.ngs.noaa.gov/IGSWorkshop2008/docs/receivers-pany.pdf>.

22. *Signal and Algorithm Development Environment for SDR*. **Schiff, M.** Maclean, VA : IEEE, 28-31 October 2001. Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force.
23. SNAP Namuru Project. *University of New South Wales (UNSW)*. [Online] [Cited: February 12, 2010.] http://www.dynamics.co.nz/index.php?main_page=page&id=9.
24. **Humphreys, Todd E. and Young, Larry.** IGS Receiver Considerations. *2008 IGS Workshop, Miami Beach, FL*. [Online] Institute of Geodesy and Navigation, 2008. [Cited: February 12, 2010.] <http://www.ngs.noaa.gov/IGSWorkshop2008/docs/slidesHumphreys.ppt>.
25. **Baracchi-Frei, M., et al.** Real-Time Software Receivers: Challenges, Status, Perspective. *GPS World*. September 2009, Vol. 20, 9, pp. 40-47.
26. **Mitelman, A., et al.** Testing Software Receivers. *GPS World*. December 2009, Vol. 20, 12, pp. 28-34.
27. *Achieving Precise Real-Time GNSS Positioning with Software-based Receivers*. **Lu, D., et al.** Savannah, GA : The Institute of Navigation, 22-25 September 2009. ION GNSS 2009.
28. **GPS JOINT PROGRAM OFFICE.** IS-GPS-200 (Navstar GPS Space Segment/Navigation User Interfaces). *ARINC Inc.* [Online] Revision D, March 7, 2006. [Cited: January 17, 2010.] GPS Satellite Almanacs and SEM Program by ARINC. <http://www.arinc.com/downloads/is-gps-200mar06.pdf>. IRN-200D-001.

29. **Meel, J.** Spread Spectrum (SS): introduction. *De Nayer Instituut*. [Online] 1999.
[Cited: February 12, 2010.] http://sss-mag.com/pdf/Ss_jme_denayer_intro_print.pdf.
30. **Krishna, H., et al.** *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques*. Boca Raton, FL : CRC Press, 1994.
31. **Milenkovic, Milan.** *Operating Systems Concepts and Design, 2nd Ed.* New York, NY : McGraw-Hill Inc, 1992. 0-07-911364-8.
32. *The Problem with Threads*. **Lee, Edward A.** Washington, DC : IEEE Computer Society, May 2006, Computer.
33. **Stallings, William.** *Computer Organization and Architecture: Designing for Performance*. Upper Saddle River, NJ : Prentice-Hall, 2000. 0-13-081294-3.
34. *An analysis method for variable execution time tasks based on histograms*. **Vila-Carbó, Joan and Hernández-Orallo, Enrique.** Doetinchem, Netherlands : Springer Netherlands, January 2008, Real-Time Systems, Vols. 38, 1, pp. 1-37. 1573-1383.
35. *The Spring architecture*. **Stankovic, John.** Horsholm, Denmark : IEEE Computer Society, 1990. Proceedings - EUROMICRO '90 Workshop on Real Time. pp. 104-113. 9780818620768.
36. **Intel Corporation.** *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Santa Clara, CA : Intel Corporation, 2009. 248966-020.
37. *Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption*. **Bril, Reinder J., Lukkien, Johan J. and Verhaegh, Wim**

- F.J. Doetinchem**, Netherlands : Springer Netherlands, April 28, 2009, Real-Time Systems, Vol. 42, pp. 63-119.
38. *The one machine scheduling problem: Insertion of a job under the real-time constraint.* **Duron, C., Louly, M.A. Ould and Proth, J.M.** : ELSEVIER, December 16, 2009, European Journal of Operational Research, Vols. 199, 3, pp. 695-701. 0377-2217.
39. *Reactive speed control in temperature-constrained real-time systems.* **Wang, Shengquan and Bettati, Riccardo.** Doetinchem, Netherlands : Springer Netherlands, August 2008, Real-Time Systems, Vols. 39, 1-3, pp. 73-95. 1573-1383.
40. *Lock-Free Algorithms.* **Jones, Toby.** [ed.] Mike Dickheiser. Boston, MA : Charles River Media, 2006, Game Programming Gems, Vol. 6, pp. 5-15. 1-58450-450-1.
41. **Rumbaugh, James, et al.** *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ : Prentice-Hall, Inc., 1991. 0-13-629841-9.
42. Unified Modeling Language. *Object Management Group Inc.* [Online] April 27, 2009. [Cited: April 15, 2010.]
<http://www.omg.org/technology/documents/formal/uml.htm>.
43. Rational Rose. *IBM Corporation.* [Online] [Cited: November 12, 2009.] <http://www-01.ibm.com/software/awdtools/developer/rose/>.

44. *Deriving objects from use cases in real-time embedded systems.* **Kimour, Mohamed T. and Meslati, Djamel.** : Elsevier, 2005, Information and Software Technology, Vol. 47, pp. 533-541.
45. **Milner, Robin.** *A Calculus of Communicating Systems.* : Springer Verlag, 1980. 0-387-10235-3.
46. *Architectural Prototyping: From CCS to .Net.* **Rodrigues, Nuno F. and Barbosa, Luis S.** : Elsevier, 2005, Electronic Notes in Theoretical Computer Science, Vol. 130, pp. 151-167.
47. *Architecture description languages for programmable embedded systems.* **Mishra, P. and Dutt, N.** : IEE, May 2005, IEE Proceedings on Computer and Digital Technology, Vols. 152, 3.
48. *Computing for Embedded Systems.* **Lee, Edward A.** Budapest, Hungary : IEEE, May 21-23 2001. IEEE Instrumentation and Measurement Technology Conference.
49. *A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems.* **Balasubramanian, Krishnakumar, et al.** : Elsevier, 2007, Journal of Computer and System Sciences, Vol. 73, pp. 171-185.
50. *A C++ Framework for Active Objects in Embedded Real-time Systems-Bridging the Gap Between Modeling and Implementation.* **Caspersen, Michael E.** 1999. 32nd International Conference on Technology of Object-Oriented Languages and Systems - TOOLS. p. 52. 0-7695-0462-0.

51. *Languages for the programming of real-time embedded systems: A survey and comparison.* **Cooling, J.E.** : Elsevier Science, 1996, Microprocessors and Microsystems, Vol. 20, pp. 67-77.
52. *C# and the .Net Framework: Ready for Real Time?* **Lutz, Michael H. and Laplante, Phillip A.** : IEEE Computer Society, IEEE Software January/February 2003.
53. IA-PC HPET (High Precision Event Timers) Specification. *Intel Corporation.*
[Online] October 2004. [Cited: April 15, 2010.]
http://www.intel.com/hardwaredesign/hpetspec_1.pdf.
54. GPSTk. *The University of Texas at Austin.* [Online] [Cited: February 12, 2010.]
<http://www.gpstk.org/bin/view/Documentation/WebHome>.
55. *Carrier Loop Architectures for Tracking Weak GPS Signals.* **Razavi, Alireza, Gebre-Egziabher, Demoz and Akos, Dennis M.** : IEEE, April 2008, IEEE Transactions on Aerospace and Electronic Systems, Vols. 44, 2, pp. 697-710. 0018-9251.
56. *Wiener's Analysis of the Discrete-Time Phase-Locked Loop With Delay.* **Spalvieri, Arnaldo and Magarini, Mauizio.** : IEEE, June 2008, IEEE Transactions on Circuits and Systems--II:Express Briefs, Vols. 55, 6, pp. 596-600. 1549-7747.
57. **Phillips, Charles L. and Nagle Jr., H. Troy.** *Digital Control System Analysis and Design.* Englewood Cliffs, NJ : Prentice-Hall Inc., 1984. 0-13-212043-7.

58. **Crawford, James A.** Phase-Locked Loops-A Broad Perspective. *CommsDesign*.
[Online] May 5, 2004. [Cited: January 20, 2010.]
<http://www.commsdesign.com/showArticle.jhtml?articleID=19502344>.
59. *Real-time GPS Software Receiver Correlator Design*. **Tian, Jin, et al.** 2007. Second International Conference on Communications and Networking in China (CHINACOM '07). pp. 549-553.
60. **Gerald, Curtis F. and Wheatley, Patrick O.** *Applied Numerical Analysis, 4th Ed.* Don Mills, ON : Addison-Wesley Publishing Company, 1989. 0-201-11583-2.
61. GPS Product Details. *SiGe Semiconductor Corporate Web Site*. [Online] [Cited: February 12, 2010.] <http://www.sige.com/index.php/products/details/category/gps>.
62. LibUsb-Win32. *SourceForge.Net*. [Online] February 15, 2004. [Cited: February 12, 2010.] <http://libusb-win32.sourceforge.net/>.
63. GNU Radio. *GNU Radio Wiki*. [Online] [Cited: April 13, 2010.]
<http://gnuradio.org/redmine/wiki/gnuradio>.
64. Windows Installation. *GNU Radio Wiki*. [Online] [Cited: April 13, 2010.]
<http://gnuradio.org/redmine/wiki/gnuradio/WindowsInstall>.
65. **Astrom, Karl J. and Wittenmark, Bjorn.** *Computer-Controlled Systems-Theory and Design*. Englewood Cliffs, NJ : Prentice Hall, 1990. 0-13-168600-3.

66. **Houpis, Constantine H. and Lamont, Gary B.** *Digital Control Systems-Theory, Hardware, Software 2nd Ed.* New York, NY : McGraw-Hill, Inc., 1992. 0-07-030500.
67. **Gamma, E., et al.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA : Addison-Wesley Publishing Co., 1995. 0-201-63361-2.
68. IUnknown. *Microsoft Developer Network.* [Online] [Cited: February 12, 2010.]
<http://msdn2.microsoft.com/en-us/library/ms680509.aspx>.
69. IDispatch. *Microsoft Developer Network.* [Online] [Cited: February 12, 2010.]
<http://msdn2.microsoft.com/en-us/library/aa912027.aspx>.
70. Microsoft Interface Definition Language. *Microsoft Developer Network.* [Online] [Cited: February 12, 2010.] <http://msdn2.microsoft.com/en-us/library/aa367091.aspx>.
71. Common Language Infrastructure. *ECMA.* [Online] [Cited: February 12, 2010.]
<http://www.ecma-international.org/publications/standards/Ecma-335.htm>.
72. Common Language Infrastructure, ISO 23271:2006. *ECMA.* [Online] [Cited: February 12, 2010.]
<http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html>.
73. Common Intermediate Language/Microsoft Intermediate Language. *Microsoft Developer Network.* [Online] [Cited: February 12, 2010.]
<http://msdn2.microsoft.com/en-us/library/c5tkafs1.aspx>.

74. *Real-time Doppler/Doppler Rate Derivation for Dynamic Applications*. **Zhang, Jason, et al.** Calgary, AB : CPGPS, 2005, Journal of Global Positioning Systems, Vols. 4, no.1-2, pp. 95-105. 1446-3156.
75. **Couch II, Leon W.** *Digital and Analog Communication Systems, 5th Ed.* Upper Saddle River, NJ : Prentice Hall, 1993. 0-13-522583-3.
76. **Roden, Martin S.** *Analog and Digital Communication Systems, 3rd Ed.* Englewood Cliffs, NJ : Prentice Hall, 1991. 0-13-033325-5.
77. **Bateman, Andrew and Paterson-Stephens, Iain.** *The DSP Handbook*. Essex, England : Pearson Education Ltd., 2002. 0-201-39851-6.
78. **Jones, Douglas L.** Classical Statistical Spectral Estimation. *Connexions*. [Online] September 7, 2006. [Cited: February 3, 2010.] <http://cnx.org/content/m12014/1.3/>.

Index

- abstract methods44
- base class41
- characteristic equation 99, 226, 227
- characteristic phase230
- characteristic polynomial 221, 227
- class factory138
- COM193, 195, 196, 210
- Component Object Model193
- concrete classes44
- convolution 240, 241
- correlation 6, 19, 22, 28, 36, 103, 104,
118, 147, 148, 149, 150, 158, 185,
231, 232, 241, 242
- data marshaling 120, 125, 126
- delegate46
- derived class41
- DllMain208
- dynamic scheduling60
- dynamic-link library 134, 191
- event handler207
- Fibonacci 223, 224, 225
- Galois220, 223, 224, 225
- Galois field220
- generating function vii, 228
- globally unique identifier193
- IDispatch 195, 196
- inheritance 37, 41, 42
- interface44
- interface definition language196
- Interop189
- irreducible 36, 221, 225
- IUnknown 195, 196
- late binding43, 136
- librarian190
- lollipop45
- loose coupled events45
- marshaler 120, 127, 128
- module definition file 191, 202, 205
- native-code 189, 198
- Nyquist 140, 184, 239
- overriding42, 78, 90
- Parseval's Theorem240
- polymorphism 37, 42, 43
- polynomial . 22, 221, 223, 224, 225, 226,
228, 230, 233
- PRN sequences234
- pure abstract44
- pure virtual44
- reciprocal 116, 224, 225
- static linking190
- static scheduler55
- Tustin's approximation212
- type-library196
- Unified Modeling Language39, 70
- virtual vii, 42, 43, 44, 78, 93, 94, 123,
193, 196
- virtual-table43
- v-table*virtual table*

Curriculum Vitae

Candidate's full name: Douglas Allan Godsoe

Universities attended:

BScE (Electrical Engineering), University of New Brunswick, 1993

MEng (Electrical Engineering), University of New Brunswick, 2006-2007
(transferred into PhD program)

PhD candidate, Electrical & Computer Engineering, 2007-2010

Publications:

Godsoe, D.A., Kaye, M.E., Langley, R.B., "A Framework for Real-time GNSS Software Receiver Research," in *ION International Technical Meeting 2010*, San Diego, CA, 25-27 January, 2010.

Conference Presentations:

A Framework for Real-time GNSS Software Receiver Research

The Institute of Navigation International Technical Meeting

January 25-27, 2010