Optimized Bit-Packing for Bit-Wise Software-Defined GNSS Radio

Zachary Clements, Peter A. Iannucci, Todd E. Humphreys
Radionavigation Laboratory
The University of Texas at Austin

Thomas Pany
Bundeswehr University Munich

BIOGRAPHIES

Zachary Clements (B.S., Electrical Engineering, Clemson University) is a graduate student in the department of Aerospace Engineering and Engineering Mechanics at The University of Texas at Austin, and a member of the UT Radionavigation Laboratory. His research interests currently include GNSS signal processing, anti-spoofing techniques, software-defined radio, and sensor fusion.

Peter A. Iannucci (BS, Electrical Engineering-Computer Science and Physics, MIT; PhD, Networks and Mobile Systems, CSAIL, MIT) is a postdoctoral research fellow in the Radionavigation Laboratory at The University of Texas at Austin, and a member of the UT Wireless Networking and Communications Group (WNCG). His current research interests include collaborative navigation, multi-spectral mapping, and re-purposing broadband Internet satellites for radionavigation.

Todd Humphreys (BS, MS, Electrical Engineering, Utah State University; PhD, Aerospace Engineering, Cornell University) is a professor in the department of Aerospace Engineering and Engineering Mechanics at The University of Texas at Austin, where he directs the Radionavigation Laboratory. He specializes in the application of optimal detection and estimation techniques to problems in secure, collaborative, and high-integrity perception, with an emphasis on navigation, collision avoidance, and precise timing. His awards include The University of Texas Regents' Outstanding Teaching Award (2012), the National Science Foundation CAREER Award (2015), the Institute of Navigation Thurlow Award (2015), the Qualcomm Innovation Fellowship (2017), the Walter Fried Award (2012, 2018), and the Presidential Early Career Award for Scientists and Engineers (PECASE, 2019). He is a Fellow of the Institute of Navigation and of the Royal Institute of Navigation.

Thomas Pany (Diploma in Physics; PhD Geodesy, Graz University of Technology; *venia legendi* in Navigation, Bundeswehr University), is a professor of satellite navigation at the Bundeswehr University Munich, Germany, since 2016. He has written numerous publications in the field navigation and he received five best presentation awards from the Institute of Navigation. His personal research focus is on GNSS signal processing and sensor fusion with special emphasis on software radio technology.

ABSTRACT

This paper proposes a framework for structuring data bit transfers from the radio frequency (RF) front-end to a general-purpose processor (GPP) in software-defined radio (SDR) for Global Navigation Satellite System (GNSS) applications. With the evolution of multi-antenna and multi-frequency GNSS SDRs, the packing and unpacking of data bits between the RF front-ends and GPP becomes increasingly complicated. ION's Metadata Standard provides a foundation for standardizing GNSS SDR output files but does not accommodate data packing formats that are efficient for processing by an important class of SDRs called bit-wise SDRs. Besides proposing an extension to the ION Metadata Standard that resolves this shortcoming, this paper treats the problem of bit-packing for bit-wise SDRs more generally: It develops a bit-packing scheme that is flexible enough to accommodate any practical combination of antennas, frequency bands, sampling rates, and quantization encodings while optimizing bit-wise SDR processing efficiency within the constraints of low-cost front-end hardware. The performance of the proposed scheme is presented in terms of reduced instructions per processed sample. Performance is validated experimentally by implementing the proposed scheme on a high-performance GNSS SDR whose dual-antenna, tri-band RF front-end was recently developed in house at the University of Texas Radionavigation Laboratory.

Preprint of the 2021 ION GNSS+ Conference St. Louis, MO, Sept. 21-24, 2021

INTRODUCTION

Within the past decade, software-defined radios (SDRs) have emerged as an especially valuable platform for GNSS research and development [1]. GNSS-SDRs implementations vary greatly, but are characterized by processing more-or-less-raw samples of radio frequency (RF) data from an analog front-end using general-purpose processors, either online (i.e. in real time) or offline (i.e. post-processing). Because they enable researchers to collect and share large raw-sample datasets, GNSS SDRs are an ideal tool for collaboration and repeatable, high-fidelity cross-verification within the GNSS community.

Lack of a standardized data format for raw RF data previously stymied this process. Properly importing a dataset into a software package different from that used for the original recording could be error-prone and tedious. To tackle this problem, the Institute of Navigation (ION) GNSS SDR Standard Working Group recently released their GNSS SDR Metadata Standard [2]. The Standard defines the structure of a machine- and human-readable auxiliary file to be distributed alongside raw-sample datasets. The introduction of this "metadata" is a much welcomed initiative that promises to eliminate formatting ambiguities and promote interoperability in GNSS-SDR research.

Not all GNSS-SDR implementations have quite the same requirements for their data formats. GNSS SDRs with software correlators may be sub-divided into byte-wise and bit-wise categories. Byte-wise SDRs represent each real or imaginary component of a front-end sample as a byte (e.g., the MuSNAT and the IFEN SX3 [3], [4]). As the smallest directly-addressable unit of computer memory and the smallest supported integer data-type on most modern architectures, bytes represent an inflection point in software complexity. Operations on narrower quantizations are less straightforward to implement.

Why might narrower quantization be desirable? The weak nature of GNSS signals-in-space and the typical hemispherical pattern of GNSS receiver antennas mean that as much as 99% of the power in a recorded GNSS-SDR waveform is additive white Gaussian noise (AWGN). At such a low signal-to-noise ratio (SNR), the formal information content of the desired signal cannot exceed a small fraction of a bit per sample. The bulk of the dataset is noise. Bit-wise SDRs exploit the uneven distribution of this fraction-of-a-bit of useful information among the output bits of the analog-to-digital converter (ADC). Under these conditions, there are diminishing returns to each bit of quantization after the first. Bit-wise SDRs therefore truncate samples, sometimes to a single bit, to reduce memory bandwidth and power consumption (e.g., the UT Austin GRID SDR [5]–[8]).

Version 1.0 of the Standard does not support the data formats that are most efficient for bit-wise SDR processing: those in which parallel planes of bits (e.g. sign bits, magnitude bits) from a single stream of RF samples are aggregated (grouped into runs) rather than collated (interleaved). This renders two recently-offered public GNSS datasets, the University of Texas Challenge for Urban Positioning (TEX-CUP) [9], and the ATX Urban Positioning Challenge Dataset [8], incompatible with the Standard. This paper proposes extensions to the Standard to improve compatibility with bit-wise SDRs.

Why should such an unusual bit-ordering be valuable? Since modern processors do not offer native arithmetic on data-types smaller than a byte, bit-wise SDRs rely instead on "bit-slicing": a digital logic circuit (AND, OR, NOT, XOR) for correlation is designed and implemented as a program with one Boolean instruction per gate. Each wire in the logic circuit is represented by a register-sized integer (a "word"), and the n^{th} bit of one word interacts only with the n^{th} bits of other words: that is, parallel bits flow through separate, parallel copies of the logic circuit. The correlator therefore operates on as many samples in parallel as there are bits in a word. It is this mapping of logical wires to register-sized integers that leads to the bit-ordering preferences of bit-wise SDRs. Just as a 2×2-bit adder circuit uses distinct wires for high and low bits, the bit-sliced implementation uses different registers and memory locations to hold (corresponding vectors of) high and low bits. What would be a single instruction (multiply and add bytes) in a byte-wise SDR becomes many instructions; but data parallelism is fully exploited, critical paths are short, and the processor can be kept busy with many inexpensive Boolean operations. Bit-slicing is simplest when the depth of quantization is just one or two bits, but the technique does not have a strict limit.

Rather than propose narrow extensions to the Standard supporting only existing bit-wise datasets, this paper aspires to greater generality: its first contribution is a scheme for encoding *arbitrary* packing of raw GNSS data sampled from potentially multiple antennas, frequency bands, quantization schemes, and sample rates. This paper lays out the proposed extensions to the Standard in concrete detail.

Second, this paper presents a scheme for efficient packing and unpacking of GNSS data streams. This scheme automatically generates compact packing logic and fast unpacking code from a description of the packed data format. Experimental results on x86-64 and ARM64 architectures demonstrate the tools' efficiency and flexibility.

As a final contribution, this paper presents the design of a new RF front-end, RadioLion, developed at the Radionavigation Laboratory (RNL) of The University of Texas at Austin. RadioLion simultaneously collects 1- to 3-bit quantized samples from three GNSS bands across two antennas. Data are streamed over USB to a general-purpose processor. The flexibility of the RadioLion has been the motivation for creating automated tools to explore alternative bit-packing schemes.

Rather than incorporating this flexibility only into the RNL's GRID software, it is the authors' wish that these solutions for efficient stream unpacking should be "up-streamed" to the ION Standard reference implementation, where they can be of universal benefit. This integration work is anticipated in the near future.

BACKGROUND AND RELATED WORK

Both bit-wise and byte-wise GNSS-SDR implementations exploit vectorized processor instructions, known as Single Instruction, Multiple Data (SIMD). SIMD instructions are ideal for calculations with high data parallelism, such as correlation, because they operate on multiple samples in the same cycle. Typical vector register sizes range from 128 to 512 bits, though the largest sizes are not available on all processors. Most x86-64 processors outside of datacenters, for instance, support up to 256-bit vector operations, while modern 64-bit ARM processors typically support only 128-bit SIMD instructions.

Depending on the CPU's particular SIMD instruction set, byte-wise SDRs can multiply and accumulate 16 samples per cycle per core [7], and bit-wise SDRs can multiply and accumulate 128 samples per core using a short sequence of SIMD XOR, `popcount`, and table look-up operations [7], [10]–[12]. The required memory bandwidth is lesser for bit-wise SDRs than for byte-wise SDRs. On the other hand, byte-wise SDRs eliminate quantization losses and are more easily adapted for use with non-binary modulation schemes like CBOC. Moreover, under non-AWGN conditions, greater quantization depth may be beneficial for e.g. adaptive notch filtering [13].

While graphics processing units (GPUs) might appear to be a compelling alternative, with their support for a stupendous amount of data-parallel computation in integer or floating-point formats [14], they suffer from high overhead in GPU/CPU communication, and are therefore of greatest use in the search phase of signal acquisition rather than during tracking.

BIT-PACKING FOR GNSS SDRS

Bit-packing within the Current Standard

The ION Metadata Standard aims to standardize the layout of binary data streams and files containing raw samples from GNSS RF front-ends. It also standardizes the specification of this layout, and of system parameters such as sample rate, IF center frequencies, bit-depth, and bit-packing schemes in the metadata associated with binary streams and files. The Standard aspires to be general enough to describe data taken from the numerous GNSS-SDR architectures used by researchers, industry professionals, and hobbyists.

Relevant portions of an example bit-wise GNSS receiver architecture are shown in Fig. 1. Multiple GNSS antennas feed multiple RF front-ends whose digitized data are packed together, either for storage or for direct online streaming to software running on a general-purpose processor. The combination of all individual RF front-ends and the downstream multiplexing hardware can be thought of as a *composite* RF front-end whose output is a single serial stream of data. Each RF front-end produces digitized data for a particular combination of antenna and one or more frequency bands at a uniform sampling rate and with uniform quantization (e.g., 1-bit, 2-bit, etc.). In the Standard, the digitized output of an individual RF front-end is called a *stream*. (All ION Metadata Standard abstractions will be italicized in this paper to distinguish them from colloquial usage of the same term.) The ith *stream*'s sample rate is expressed as an integer multiple SF_i of a base rate f_s that is assumed to apply to the receiver system as a whole. Each sample in a stream can be either real or complex and may be encoded with one of 12 encoding schemes enumerated in the Standard.

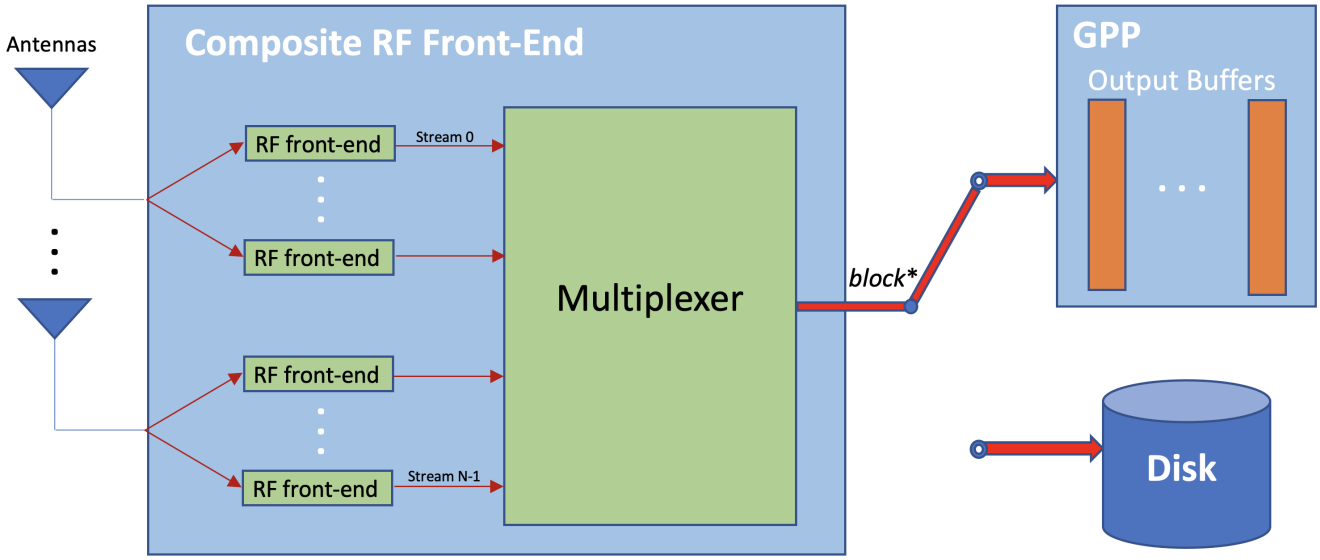


Fig. 1: General GNSS-SDR architecture. Antennas feed analog signals to the RF front-ends for signal conditioning, downmixing, and digitization. Each RF front-end produces a *stream* of data. The multiplexer combines *streams* into *lumps* for transport to a general-purpose processor. Before being written to disk for offline processing, these *lumps* may be further structured into *chunks*, *blocks*, and *lanes*. For online processing, the *streams* must be de-multiplexed on the general-purpose processor into various buffers for correlation. The multiplexer can be implemented in programmable logic (FPGA/CPLD and/or a microcontroller). In a bit-wise GNSS SDR, each output buffer receives a single bit-plane (e.g., all sign bits or all magnitude bits) from the samples of one *stream*.

As shown in Fig. 2, data from multiple streams collected over $t_s = 1/f_s$ seconds are multiplexed to form a *lump*. The Standard assumes all samples belonging to a *stream* are ordered chronologically and grouped contiguously within a *lump*.

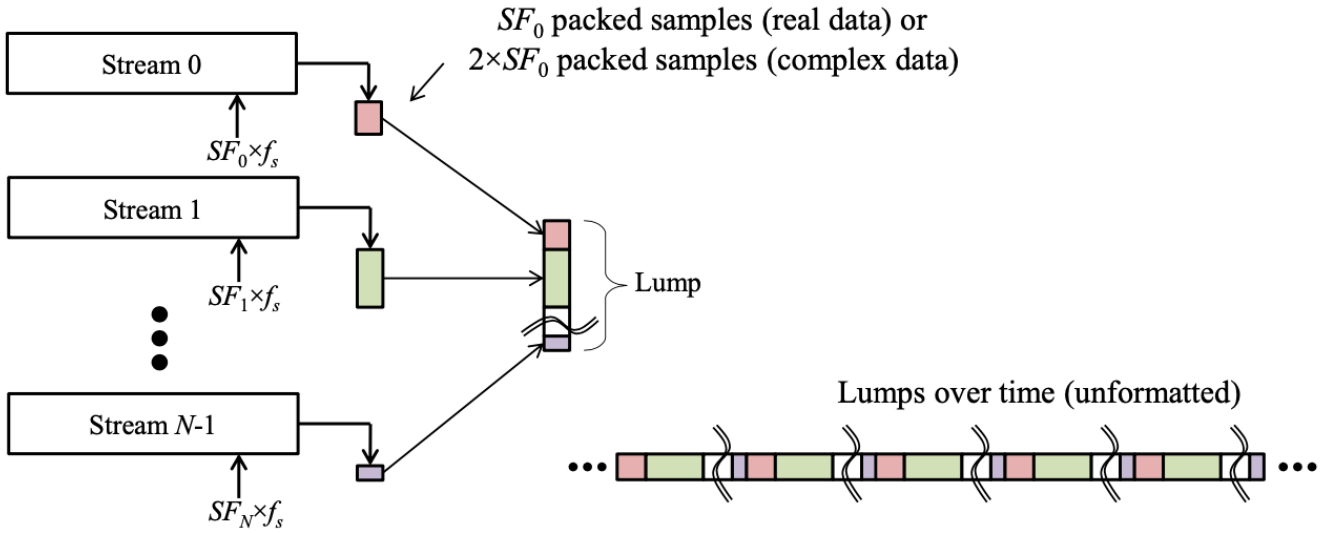


Fig. 2: A *lump* concatenates SF_0 samples from the 0th *stream* with SF_1 samples from the 1st stream, etc., up to the $(N-1)$th *stream*. Importantly, Version 1.0 of the Standard assumes that all samples belonging to a *stream* are ordered chronologically and grouped contiguously within a *lump*. Figure credit: The ION Metadata Standard Version 1.0.

One or more *lumps* may be packed in various ways into *words*, where a *word* refers to a 8-, 16-, 32-, or 64-bit standard unsigned integer data type. Such packing into *words* prepares the samples for writing and reading the samples to/from disk.

When packed into one or more *words*, a collection of *lumps* forms a *chunk*. All *words* within a *chunk* are assumed to be of the same type (e.g., all 8-bit unsigned integers). A nonzero number of sequential and contiguous *chunks*, plus an optional header or footer containing ancillary data (e.g., parity bits), are packed into a *block*. The structure of a *block* is presumed to remain constant for the entire data collection session. In other words, no dynamic *block* re-formatting is permitted by the Standard. One or more potentially heterogeneous *blocks* are transported via a *lane* to disk for post-processing or to a general-purpose processor for online correlation.

The authors of the Standard provide a reference implementation of *stream* packing and unpacking as a C++ library. The C++ reader is a normative reference and uses extensive for-loops for bit-shifting to isolate samples from *lumps* one-at-a-time. The use of the for loops renders the reader quite slow and it does not exploit advanced instructions (like SIMD) on the CPU.

Efficient bit-packing for bit-wise SDRs

The bit-wise parallel correlation technique first introduced in [10] operates on circular buffers containing digitized front-end data, as shown on the right-hand side of Fig. 1. Each bit in a circular buffer corresponds to a separate sample of data. Bits are ordered sequentially with no gaps. The Standard's *stream* abstraction can be thought of as being unloaded into a set of circular buffers, one for each of the *stream*'s bit-planes (each quantization bit). For example, a two-bit-quantized (four quantization level) *stream* maps to two circular buffers, one for the sign bit-plane and one for the magnitude bit-plane.

Bit-wise parallel correlation amounts to a highly optimized sequence of SIMD instructions that operate on the separate circular buffers to correlate 2^p samples in parallel against local code and carrier replicas. For the latest variant of RNL's GRID receiver, $p = 7$, meaning that 128 samples are correlated in parallel.

Despite the impressive efficiency of bit-wise parallel correlation, a bit-wise SDR's overall processing can be brought to a crawl if the operations required to unpack a *lane* into its corresponding circular buffers are too numerous or complex. For efficient unpacking, a *lane* should contain extended runs of bits destined for a particular circular buffer. For example, a *chunk* within a *lane* could be composed of a run of 8 sign bits for a particular *stream*, followed by 8 magnitude bits for the same *stream*, followed by similarly-arranged sign and magnitude bits for a total of four *streams*, as follows:

$$\underbrace{\overbrace{[S_1 \ldots S_8]}^{\textit{Stream 0}}[M_1 \ldots M_8]}_{\text{Byte: } \textit{uint8\_t}} \overbrace{[S_1 \ldots S_8][M_1 \ldots M_8]}^{\textit{Stream 1}} \overbrace{[S_1 \ldots S_8][M_1 \ldots M_8]}^{\textit{Stream 2}} \overbrace{[S_1 \ldots S_8][M_1 \ldots M_8]}^{\textit{Stream 3}}$$

One can readily appreciate how efficiently this *chunk* can be unpacked into its corresponding circular buffers for bit-wise correlation: unpacking simply involves byte-wise "read" and "copy" operations. The TEX-CUP and ATX Urban Positioning Challenge datasets adopt a format similar (but not identical) to the above.

Unfortunately, a *chunk* formatted as above violates Version 1.0 of the ION Metadata Standard because it does not adopt one of the Standard's accepted *stream* encodings, which assume that samples appear sequentially. The Standard would have the 2-bit-quantized data formatted instead, e.g., as the following *chunk*:

$$\underbrace{[\overbrace{SM}^{\textit{Stream 0}} \overbrace{SM}^{\textit{Stream 1}} \overbrace{SM}^{\textit{Stream 2}} \overbrace{SM}^{\textit{Stream 3}}]}_{\text{Byte: } \textit{uint8\_t}}$$

This encoding is particularly inefficient for unpacking into circular buffers for bit-wise parallel correlation because each bit must separately be shifted, masked, copied, and re-packed into bytes. In fact, *the required number of operations per bit to unpack such data exceeds that of bit-wise parallel correlation on the same data*, an unacceptable situation for bit-wise SDRs.

One could imagine extending the Standard's list of acceptable encodings to include an abstract encoding type that accommodates runs of, say, 8 bits from the same *stream* bit-plane within a *lump*, as in the efficient format above. As a consequence of this extension, the Standard's insistence on samples appearing sequentially in a *lump* would have to be relaxed. But such a narrow extension to the Standard would be a missed opportunity to think more broadly about bit-packing schemes that are at once highly efficient for bit-wise SDRs, information-dense, flexible, and respectful of hardware constraints on low-cost composite RF front-ends. The following section explores development of such a general bit-packing scheme.

A PROPOSAL FOR OPTIMIZED BIT-PACKING FOR BIT-WISE SDRS

As a prelude to the development of a general bit-packing scheme for bit-wise SDRs, the next subsection offers some example cases that the scheme should be able to accommodate.

Example Cases

The Standard should be at least as flexible as the SDR platforms it serves. Future low-cost SDRs with low-end hardware require clever bit-packing schemes to make the most of limited resources. In addition to running bit-planes, speculative bit-packing schemes may include other features that are currently not accounted for in the standard. SDR systems may seek to employ non-collated bit-planes and interleaving, sample clock phase offsets for staggering multiple channels, and periodic masking of non-GNSS data bits.

Higher quantization levels (bit-depths) reduce SNR loss and provide more interference resistance, but come at the cost of more expensive hardware and software requirements. The power level of GNSS signals captured on Earth's surface is incredibly weak, so GNSS receiver designers attempt to eliminate as much SNR degradation as possible. Quantizing analog signals into digital signals incurs a loss in SNR [15]. Low-cost GNSS receivers are often handcuffed in performance because of their limited on-board resources. Designers of low-cost GNSS receivers often opt for one-bit quantization because of its simple implementation for a low cost in SNR [16]. The implementation requirements of higher quantization systems can be alleviated by allowing interleaving of *streams* within *lumps*. Interleaving allows data bits to be sent more quickly, reducing multiplexer storage requirements.

Future GNSS systems may want to include staggered sampling across channels and streams. Such a scheme could potentially be beneficial in the event of GNSS interference. Staggering samples from multiple antennas might grant greater observability in the case of jamming or intermodulation distortion by having a "virtual" sampling frequency faster than either ADC clock. The ability of staggered sampling provides the versatility needed for advanced signal processing techniques such as sparse sampling and compressed sensing. Continuous wave frequency modulated jammers are common sources of anti-GNSS jamming in contested areas. Interference waveform characterization can be achieved through sparse reconstruction methods, which is applied to obtain non-parametric instantaneous frequency estimation. This can further be exploited to enhance jammer localization and suppression [17], [18]. Highly customized sampling schedules may want to use staggered sampling to implement these sparse sampling algorithms.

An example scheme that exploits staggered sampling is shown in Table I, with rows indexing "frames", or cycles of a clock corresponding to the least common multiple of all of the sample rates. Note that L2 and L5 are sampled at the same rate on both antennas, but not at the same times. This represents a virtual doubling the number of independent looks at the L2 channel, which could be exploited to increase interference observability. This scheme is not achievable under the current standard.

TABLE I: Representation of potential SDR system with staggered sampling

Frame	Antenna 1			Antenna 2			Data	
	L1	L2	L5	L1	L2	L5	Bits Sampled	Total Bits
0	3-bit	1-bit		2-bit			6	6
1	3-bit		1-bit	2-bit	1-bit		7	13
2	3-bit	1-bit		2-bit		1-bit	7	20
3	3-bit			2-bit	1-bit		6	26
4	3-bit	1-bit	1-bit	2-bit			7	33
5	3-bit			2-bit	1-bit	1-bit	7	40

Proposed Metadata Standard Extensions

This section details two independent sub-proposals for extending version 1.0 of the ION GNSS-SDR Sampled Data Metadata Standard. The first addresses shortcomings in describing periodic, synchronous timing relationships among heterogeneous *streams*, such as staggered sampling. The second sub-proposal introduces means to override the default layout of *stream* sample bits within a *lump*.

Proposal #1: Flexible Sample Timing: Periodic, synchronous timing relationships between *streams* could be encoded in many equivalent ways. A straightforward approach amenable to clear documentation is as follows:

The *stream* object gains two new attributes.

TABLE II: Definition of new *stream* attributes

Attribute	Description	Class	Enumeration	Required	Default
delayticks	Time elapsed from start of *lump* until first sample from this *stream*, measured in ticks of a clock with frequency **freqbase** × **delayfactor**.	unsignedInt		No	0
delayfactor	Factor by which delay units are shorter than base clock ticks.	unsignedInt		No	1

Now, rather than the i^{th} sample from a *stream* being sampled at time

$$t_{stream,i} = \frac{1}{system.\mathbf{freqbase}} \times \left(\frac{i}{stream.\mathbf{ratefactor}} \right),$$

it is instead sampled at time

$$t_{stream,i} = \frac{1}{system.\mathbf{freqbase}} \times \left(\frac{i}{stream.\mathbf{ratefactor}} + \frac{stream.\mathbf{delayticks}}{stream.\mathbf{delayfactor}} \right).$$

It is required that

$$0 \leq stream.\mathbf{delayticks} < stream.\mathbf{delayfactor}.$$

Proposal #1 would look like this in the schema definition for *stream* objects:

```
<xs:element default="0" maxOccurs="1" name="delayticks" type="xs:unsignedInt"/>
<xs:element default="1" maxOccurs="1" name="delayfactor" type="xs:unsignedInt"/>
```

Proposal #2: Flexible Lump Layout: In the existing Standard, a *stream* object encodes a mapping between RF samples and bit-strings of length *stream*.**quantization**[1]. Similarly, a *lump* object encodes an (as-yet trivial) mapping between *lists* of bit-strings of length *stream*.**quantization** representing samples, and a single combined bit-string representing the entire *lump*.

The mapping between bit-strings and octet-strings is specificed in the *chunk* object, and is not affected by this proposal.

The *lump* object gains a new property.

TABLE III: Definition of new *lump* attribute

Attribute	Description	Class	Enumeration	Required	Default
layout	Explicit layout of bits in packed representation of *lump*.	*LumpLayout*		No	See text.

In the absence of an explicit *lump*.**layout**, the layout defaults to the old behavior. This may be summed up as follows: first, within each *stream* in the *lump*, packed bit-strings representing the *stream*.**ratefactor** samples in the *lump* are concatenated in the order specified by *lump*.**shift**. Next, these concatenated bit-strings from the various *streams* are further concatenated in the order the *streams* are enumerated in the *lump*.

A new sub-type of *MetadataElement* is introduced, *LumpLayout*. A *LumpLayout* is primarily an ordered sequence of one or more *bit* elements, each specifying a stream, a sample of that stream, and a bit-plane of that sample to be included in the *lump*. However, two edge cases motivate a slightly more complex definition of *LumpLayout*.

First, certain bits of certain samples may not appear in the packed *lump* at all. For instance, a three-bit quantized *stream* sampled twice per *lump* (**ratefactor** of 2) might have its least-significant bit omitted from every other sample. These bits are discarded during packing, or *punctured*.

[1]The distinction between real- and complex-valued samples and the associated factor of 2× applied to each appearance of *stream*.**quantization** are omitted for simplicity of exposition.

Second, certain bits in the packed stream may not correspond to any sample, but instead are *padding* or have an application-specific meaning. Nominally, these bits are discarded during unpacking.

To capture these cases, a *LumpLayout* contains a sequence of one or more *bit* elements, possibly interspersed with *pad* and/or *puncture* elements.

TABLE IV: Definition of *LumpLayout* attributes

Attribute	Description	Class	Enumeration	Required	Default
bit	One bit from one sample of one *stream* appearing in the packed *lump*.	*LumpBit*		Yes	
pad	One bit not from any *stream* appearing in the packed *lump*. Can be used with `fill="..."` attribute to override default empty-bit interpolation during packing.	*LumpBit*		No	
puncture	One bit from one sample of one Stream *not* appearing in the packed *lump*. Can be used with `fill="..."` attribute to override default empty-bit interpolation during unpacking.	*LumpBit*		No	
extra	If present, a unique identifier for the format of data stored in *pad* bits. Optionally links to human-readable documentation.	URI		No	Pad bits are arbitrary.

The *bit*, *pad*, and *puncture* elements share the same new datatype, *LumpBit*.

TABLE V: Definition of *LumpBit* attributes

Attribute	Description	Class	Enumeration	Required	Default
stream	Index of the *stream* from which this bit is taken, if any.	unsignedInt		No	
sample	Index of the sample (within one period of the base clock) from which this bit is taken, if any.	unsignedInt		No	
plane	Index of the bit within the bit-string representation of the sample, if any.	unsignedInt		No	
fill	How an empty bit is interpolated during packing (*pad*) or unpacking (*puncture*).	BitFillMethod	`"0"`, `"1"`, `"extend"`, `"extra"`	No	See text.

- *LumpBit*.**stream** indexes into the *streams* of the *lump*.
- *LumpBit*.**sample** indexes into the *stream*.**ratefactor** samples collected from this stream during one cycle of the base clock.
- *LumpBit*.**plane** indexes into the bits of the packed (bit-string) representation of the sample as produced by the *stream*. (n.b. plane 0 is the right-most and least-significant bit in the bit-string.)
- *LumpBit* (*puncture*).**fill** defaults to `"extend"` if *stream*.**encoding** is two's complement, and `"0"` otherwise.
- *LumpBit* (*pad*).**fill** defaults to `"0"`.

So, for instance,

```
<lump>
  <stream>...</stream>
  <stream>...</stream>
  <layout>
    <bit stream="0" sample="0" plane="0"/>
    <bit stream="0" sample="1" plane="0"/>
    <bit stream="0" sample="2" plane="0"/>
    <bit stream="0" sample="3" plane="0"/>

    <bit stream="0" sample="0" plane="1"/>
    <bit stream="0" sample="1" plane="1"/>
    <bit stream="0" sample="2" plane="1"/>
    <bit stream="0" sample="3" plane="1"/>

    <bit stream="1" sample="0" plane="0"/>
```

```
    <bit stream="1" sample="1" plane="0"/>
    <bit stream="1" sample="2" plane="0"/>
    <bit stream="1" sample="3" plane="0"/>

    <bit stream="1" sample="0" plane="1"/>
    <bit stream="1" sample="1" plane="1"/>
    <bit stream="1" sample="2" plane="1"/>
    <bit stream="1" sample="3" plane="1"/>
    <pad/>
    <pad/>
    <pad/>
    <puncture stream="0" sample="0" plane="2" fill="0"/>
    <puncture stream="0" sample="1" plane="2" fill="1"/>
    <puncture stream="0" sample="2" plane="2" fill="0"/>
    <puncture stream="0" sample="3" plane="2" fill="1"/>
  </layout>
</lump>
```

TABLE VI: Enumeration of *LumpBit* **fill** attribute

XML String	Description
`"0"`	Fill with zero when necessary.
`"1"`	Fill with one when necessary.
`"extend"` (only with *puncture*)	Extend the least-significant unpunctured bit to this bit when necessary.
`"extra"` (only with *pad*)	Do not interpolate—padding bits are meaningful.

A future extension might introduce another bit filling method, `"bernoulli"`, with a numeric parameter **p** defaulting to 0.5. However, it would be premature to introduce such a feature without carefully considering the resulting loss of determinism.

Packing: Packing of a *lump* occurs as if each sample from each *stream* was explicitly and individually packed into a bit-string of length *stream*.**quantization** according to the attributes of the *stream*, and then the *LumpLayout* indexed into those packed bit-strings.

Unpacking: Unpacking proceeds as if the *LumpLayout* was used to explicitly and individually reconstruct the packed bit-strings representing each sample from each *stream*, and then the attributes of the *stream* were used to decode those packed bit-strings.

Restrictions: It is an error to use **fill**=`"extend"` with a *stream*.**encoding** other than two's complement, or when all higher-order bits are punctured.

It is required that

$$0 \leq \textit{LumpBit}.\textbf{stream} < \text{number of } \textit{streams} \text{ in } \textit{lump}$$
$$0 \leq \textit{LumpBit}.\textbf{sample} < \textit{stream}.\textbf{ratefactor}$$
$$0 \leq \textit{LumpBit}.\textbf{plane} < \textit{stream}.\textbf{quantization}$$

It is an error for any two *bit* or *puncture* elements in the same *LumpLayout* to have exactly the same (**stream**, **sample**, **plane**) combination.

It is *not* an error for a valid (**stream**, **sample**, **plane**) combination to be absent from the *LumpLayout*. In this case, omitted bits are assumed to be punctured, and default empty-bit interpolation behavior is invoked when needed. (See default value of *LumpBit*.**fill** attribute above.)

Lump size: The *lump* is permitted to be larger or smaller than the sum of its *streams*' **quantization** depths. The actual size of a packed *lump* in bits may be found by counting its *bit* and *pad* elements.

Amendment to §6.2.6: The language from this section would be amended as follows:

As such, **freqbase** represents the rate at which *Lumps* are produced within the *System*. This should ordinarily be set to the greatest common divisor of the sample rates of the *Streams*. When explicit *Lump* layout is in use, a longer repetition period (and hence lower **freqbase**) may be required.

Interpretation of padding bits: This is left up to implementations, with the following caveats:

- If a data source assigns semantic significance to padding bits (e.g. IMU measurements or timestamps), it should mark these bits with **fill**="extra" and include a URI in the *LumpLayout*.**extra** attribute that uniquely identifies the padding-bit data format or format version in use.
- A data sink should not interpret padding bits unless the *LumpLayout*.**extra** attribute is present and contains a URI known to the reader.
- A data converter might need to produce output in a format which ostensibly includes semantically-meaningful padding bits, but whose *LumpLayout*.**extra** URI the converter does not recognize. In this case, it *may* copy these bits verbatim from a data source with an identical *LumpLayout*. Otherwise, it should fill the bits with zeros and generate output metadata with (1) the *LumpLayout*.**extra** attribute removed and (2) "0" substituted for "extra" in each *LumpBit*.**fill** attribute.

Interpolation of punctured bits: During unpacking, a data reader may use the *LumpBit*.**fill** attribute of punctured bits to fill in gaps in the reconstructed packed-sample bit-strings. The ordering of <puncture/> elements within the *LumpLayout* has no effect.

Schema Definition: Proposal #2 would look like this in the schema definition, omitting annotations:

```
<xs:complexType name="Lump">
 <xs:complexContent>
  <xs:extension base ="MetadataElement">
   <xs:sequence minOccurs="0">
    <xs:element minOccurs="1" maxOccurs="unbounded" name="stream" type="Stream"/>
    <xs:element minOccurs="0" maxOccurs="1" name="shift" type="Alignment"/>
    <xs:element minOccurs="0" maxOccurs="1" name="layout" type="LumpLayout"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="LumpLayout">
 <xs:complexContent>
  <xs:extension base ="MetadataElement">
   <xs:sequence minOccurs="0">
    <xs:choice maxOccurs="unbounded">
     <xs:element minOccurs="1" maxOccurs="unbounded" name="bit" type="LumpBit"/>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="pad" type="LumpBit"/>
     <xs:element minOccurs="0" maxOccurs="unbounded" name="puncture"
                      type="LumpBit"/>
     <xs:element minOccurs="0" maxOccurs="1" name="extra" type="xs:anyURI"/>
    </xs:choice>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:simpleType name="BitFillMethod">
 <xs:restriction base="xs:string">
  <xs:enumeration value="0"/>
  <xs:enumeration value="1"/>
  <xs:enumeration value="extend"/>
```

```
  <xs:enumeration value="extra"/>
 </xs:restriction>
</xs:simpleType>

<xs:complexType name="LumpBit">
 <xs:sequence/>
 <xs:attribute name="stream" type="xs:unsignedInt"/>
 <xs:attribute name="sample" type="xs:unsignedInt"/>
 <xs:attribute name="plane" type="xs:unsignedInt"/>
 <xs:attribute name="fill" type="BitFillMethod"/>
</xs:complexType>
```

Further Extensions

The authors of this paper considered a third extension to the Standard introducing a mechanism to designate specific GNSS bits in a serialized stream as having been *overwritten* by other data (the term "clobbered" is sometimes used). Such a bit-mask might be implemented as an additional per-*block* parameter to the demultiplexer. This was judged to be of secondary importance for the current work—but the motivation is documented here regardless for further discussion.

The RadioLion uses separate hardware components for data serialization and USB communication. Moreover, within the USB component, not all functions are fully programmable. In order to provide reliable detection of downstream packet over- or under-runs, the USB micro-controller writes a time-stamp into each packet. Unfortunately, the hardware is not capable of inserting this time-stamp without overwriting GNSS data.

Because these timestamps replace GNSS bits, they cannot simply be designated as header or footer bits in a *block*. If they were, a software receiver would suffer periodic phase trauma due to mis-counting of received samples. On the other hand, only two bytes out of every 1024 bytes are overwritten in this fashion. The degradation in SNR due to a receiver's ignorance of this detail is therefore likely to be small. It might be more elegant for the metadata to annotate this correctly, but it would make little difference to quantitative results.

Notation

This subsection introduces a concise vectorial notation for representing a composite front-end's sampling schedule. It is less flexible than the representation proposed above, but it aids in describing the algorithms presented in the next section.

From the perspective of the multiplexer, details of antennas and channels are irrelevant. Each *stream* has an index i, bit-depth b_i, decimation factor d_i, and decimation phase p_i. The multiplexer is a synchronous digital state machine clocked at the least common multiple of the sample rates of the *streams*. It collects a sample of b_i bits (counting both I and Q components) from stream i in cycle j iff $j \bmod d_i = p_i$. The overall sampling scheme is then summarized by the vectors $(\boldsymbol{b}, \boldsymbol{d}, \boldsymbol{p})$.

For example, the sampling scheme in Table I is represented as:

$$\boldsymbol{b} = \begin{bmatrix} 3 & 1 & 1 & 2 & 1 & 1 \end{bmatrix}, \qquad \boldsymbol{d} = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \end{bmatrix}, \qquad \boldsymbol{p} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 2 \end{bmatrix}$$

ALGORITHMS

This section discusses optimized bit-packing and bit-unpacking algorithms for bit-wise SDR. The objective for bit-packing is to multiplex *streams* in an efficient manner to meet stringent hardware constraints. The bit-unpacking algorithm performs fast bit permutation and transposition on a packed input stream to produce one or more output streams (often directed to first-in-first-out queues or circular buffers).

Optimized Bit-packing for Bit-wise SDR

Serialization and de-serialization (or packing and unpacking) schedules for sample bits have grown increasingly cumbersome to design and maintain across software, firmware, and hardware description code-bases. This problem is exacerbated when contemplating additional channels, antennas, quantization, and sample frequency options. In the presence of hard resource

limits driven by hardware cost and weight optimization, the design of packing schedules becomes quite important: in the RNL's case, these limits affect both bus throughput—motivating careful consideration of which bits of which samples are most valuable—and multiplexer size—ruling out haphazard solutions. This section presents a systemic, rational, and self-optimizing approach to designing bit-packing schemes for bit-wise GNSS-SDR implementations.

The first limitation considered is the throughput of the data bus. This represents the maximum capacity of data transfer and is the upper bound on the throughput of a *lane*. USB 2.0 and 3.0 are particularly common and well-supported options, offering maximum signalling rates of 480 Mbps and 4.8 Gbps, respectively. However, because of unacceptable L-band noise coupling between USB 3.0 components and nearby GNSS antennas, the RadioLion uses USB 2.0. Byte-wise SDRs with multiple *streams* and high sample rates will quickly exhaust the throughput of USB 2.0, and may be better served by e.g. gigabit Ethernet or PCI Express.

The USB bus is host-driven, meaning that a device like the RadioLion cannot simply transfer samples as they arrive. Instead, the host periodically polls the USB device, which responds to indicate how many "IN" transfers the host must initiate to drain the device's buffers. Thus, before samples can be transferred over the USB bus, they must be serialized into memory. Similar restrictions apply to many other high-speed multi-purpose data buses: sampled data must first be serialized into a packet in memory, so that when a bus transfer begins, an integer number of complete packets may be transferred at the highest possible rate, and then the bus may be released for other purposes.

Packets: The precise serialization of bits into a packet is highly implementation-dependent: for instance, whether bits are packed tightly with no wasted space, or whether padding may be present; whether headers or footers are present, and so forth. For the purpose of this section, suppose that there are no headers, footers, or wasted space in a packet.

The functional component that serializes bits from the individual ADC channels to the USB packet queue is the multiplexer. Unlike a general-purpose processor, the multiplexer does not have random access to its output memory. It can only choose, at each clock cycle, whether to write to the sequentially next output address, and what to write to that address. Its view of the input streams is similarly synchronous: it must decide, at each clock cycle, whether to capture the value of each bit presented to it by each of the individual streams' ADCs. Even when sample rates and quantizations are homogeneous across streams, if the number of bits collected in one clock cycle is not exactly the size of a single write to the output memory, the multiplexer must route bits appropriately to and from its internal storage in each cycle.

The sampling and packing schedules followed by the multiplexer must be periodic and known to the downstream GNSS-SDR software. Packing should abide by an economical schedule that transfers data bits promptly, while maintaining low storage and routing costs on the multiplexer.

The rest of this section discusses generating optimized bit-packing schemes by minimizing a set of heuristics. This is followed by presenting two different bit-packing schemes for a hypothetical sampling configuration. The first scheme presents the bit-packing scheme under the current Standard. The second scheme depicts an optimized bit-packing scheme generated by minimizing heuristics.

Optimization Heuristics: By minimizing a set of heuristics modeling hardware resource utilization, one searches for a bit-packing scheme that is *feasible*. A bit-packing scheme is feasible if it does not exceed the multiplexer storage capacity at any given instance. One further searches for a scheme that is *optimal*, leaving maximum hardware resources available for other purposes. To guide the search towards the set of feasible solutions, and then further towards more optimal solutions, one considers storage usage, storage input to output routing cost, host demultiplexing cost, and transfer latency. It is imperative to minimize the amount of temporary storage used given the constrained hardware. As soon as the multiplexer accumulates enough data bits to fill a packet, the data should be sent. This way, the storage registers occupied by the transferred bits are freed to store new data bits. The storage schedule aims to minimize the amount of unique routes between sample streams and storage registers because altering routing paths inside the multiplexer is computationally expensive. Packing data bits as runs of a particular bit-plane minimizes the demultiplexing cost. A sampled data bit is defined to be "prompt" if it is transferred in the same frame in which it is sampled, or "delayed" if it is transferred in a frame later than it was sampled. Bits should be transferred in the same frame to reduce system latency. Therefore, the number of bits that are delayed should be minimized.

For any given set of parameters, there are a multitude of schedules that can be constructed. Automatically generated schedules are validated by a satisfiability solver that ensures each bit appears in schedule exactly once and that no bit is transferred

before it arrives. Schedules are generated, validated, and then given a rating based on the heuristics listed above. The schedule with the best score (lowest) is taken as the bit-packing scheme.

Comparison of Bit-packing in the Standard against Optimized Scheme : The following example compares the bit-packing scheme required under the current Standard to an optimized bit-packing scheme for a multiplexer with minimal storage. Consider a composite front-end with two antennas, each feeding three RF front-ends at the L1, L2, and L5 frequencies, for a total of six *streams*. A hypothetical sampling scheme is shown in Table VII where each *stream* is sampled at a different rate and with different bit-depths. This scheme can be expressed as:

$$\boldsymbol{b} = \begin{bmatrix} 3 & 1 & 1 & 2 & 1 & 1 \end{bmatrix}, \qquad \boldsymbol{d} = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \end{bmatrix}, \qquad \boldsymbol{p} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

TABLE VII: An example sampling schedule of a hypothetical composite front-end.

Frame	Antenna 1			Antenna 2			Data	
	L1	L2	L5	L1	L2	L5	Bits Sampled	Total Bits
0	3-bit	1-bit	1-bit	2-bit	1-bit	1-bit	9	9
1	3-bit			2-bit			5	14
2	3-bit	1-bit		2-bit	1-bit		7	21
3	3-bit		1-bit	2-bit		1-bit	7	28
4	3-bit	1-bit		2-bit	1-bit		7	35
5	3-bit			2-bit			5	40

In this example, 8-bit packets are taken into consideration. This sampling schedule will begin repeating after six frames and collect a total of 40-bits over the interval. The bits collected are aggregated into five packets to form a *lump*. Table VIII shows how many total bits are collected on each *stream* within a *lump*. Table IX shows how the 40 bits must be packed over the five packets under the current Standard. The entirety of channel 0's data is required to be packed and sent before any of the data on the other channels because the Standard does not allow interleaving between streams. This results in a total of 22 bits required to be temporarily stored on the multiplexer. These bits are also delayed by multiple frames. Additionally, the routing path in the multiplexer changes between every frame. These characteristics make for an undesirable bit-packing scheme.

TABLE VIII: Example of six streams multiplexed into a single *lump* under the current standard.

Channel 0	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5
18 bits	**3 bits**	**2 bits**	**12 bits**	**3 bits**	**2 bits**

TABLE IX: The corresponding transfer schedule for the *lump* in Table VIII. Each databit is represented as a three digit key. The first digit represents which channel it came from, the second digit represents which frame it was sampled in, and the third digit represents which bit in the quantization it is. All of the sampled data on channel 0 must be transported in a contiguous manner before any of the other channels can be sent. Data on channels 1, 2, 3, 4, and 5 must be temporarily stored on the multiplexer. Because channel 0 collects data up until the last frame, the earliest data from the other channels can be sent is the last frame.

	Bit Index							
	0	1	2	3	4	5	6	7
Packet 1	000	001	002	010	011	012	020	021
Packet 2	022	030	031	032	040	041	042	050
Packet 3	051	052	100	120	140	200	230	300
Packet 4	301	310	311	320	321	330	331	340
Packet 5	341	350	351	400	420	440	500	530

An optimized bit-packing scheme's transfer and storage schedule are shown in Tables X and XI. Arriving samples are either transferred promptly or stored only for a single frame. For example, there are nine bits sampled in the first frame, filling a packet. Eight bits are immediately transferred and the extra bit is stored on the multiplexer. This optimized bit-packing scheme takes advantage of the *LumpLayout* extension offered earlier in order to interleave *streams*. This specific storage schedule only needs six single bit registers of storage at any given moment, significantly less than the original scheme. Furthermore, each data bit is transferred in the same frame that it was sampled or delayed by only a single frame. The latency is significantly reduced compared to the bit-packing scheme specified by the standard. The routing between sample register to storage register generally remains static, reducing the routing cost. Storage requirements are minimized while transferring data promptly. Although the serialized stream does not contain long runs of bit-planes, it can still be efficiently unpacked as described in the following sections.

TABLE X: An optimized transfer schedule for the example.

	Bit Index							
	7	6	5	4	3	2	1	0
Packet 1	500	100	002	000	400	301	001	200
Packet 2	300	012	022	010	020	311	310	011
Packet 3	330	331	021	120	420	321	320	230
Packet 4	530	032	030	040	440	341	031	041
Packet 5	340	042	052	140	050	351	350	051

TABLE XI: An optimized storage schedule for the example.

	Multiplexer single-bit registers					
	5	4	3	2	1	0
Frame 0						300
Frame 1	010	310	012	311	011	300
Frame 2		420	120	321	021	320
Frame 3	030	530		032	031	
Frame 4			140	042		340
Frame 5						

Look-up table based Bit-unpacking

In principle, the bit unpacking problem can be formulated as a function $u(\cdot)$ that maps the value l of the *lump* depending on the stream index s (i.e. antenna and frequency band) and the frame index f to the sample value v,

$$v_{f;v} = u(l; f, v)$$

The size of the look-up table (LUT) is given by the bit-width of l multiplied by the number of streams, number of frames per *lump* and the number of bits to represent v. If this size matches the system capabilities (e.g. on high performance PCs a few Megabyte can often be afforded), the implementation of a look-up table is possible. Some CPUs even support vectorized access to those tables, e.g. in the Haswell generation of Intel CPUs, Intel introduced a new CPU instruction that can load a (256-bit wide AVX2) SIMD register from multiple independent locations in memory, that is, each 32-bit or 64-bit element of the register can be loaded from random locations in memory with a single instruction [19].

The LUT is precomputed based on the bit-encoding scheme. This approach does support all ways of bit encoding. If several samples v need to be combined to a single byte/word after decoding, this can be achieved via a simple OR operation as the values obtained for different frames are already at the correct bit position. In practice the size of the *lump* needs to be representable by 16-bits or less.

Parallel Bit-unpacking

Consider the following "brute-force" algorithm: for each *lump* in the input *lane*, iterate over the bits of the *lump*. For each bit, load the first incomplete 64-bit word from the appropriate output buffer, then shift, mask, and OR the input bit into the output word, and finally write the output word back into memory:

Under very generous assumptions about optimizations, the brute-force algorithm performs a minimum of five operations (load, shift, mask, OR, store) per input bit. The goal of efficient bit-unpacking is to achieve (far) less than five operations per bit.

Using one 16-bit LUT for each of N_{out} output streams, and supposing that incomplete output words can be stored in registers rather than spilled to memory, one achieves roughly $(3/16)N_{\text{out}}$ operations per bit (look-up, shift, OR). For twelve output streams, this is 2.25, for a 55% improvement over the brute-force algorithm. While look-up tables are a tried-and-true method

Algorithm 1: Brute-force bit unpacking.

```
Input  : packedWords : list(word)
Input  : lumpLayout : list(bit | pad)
Input  : outStreams : list(list(bit))
Output: unpackedWords : list(list(word))

shiftReg ← 0;
shiftRegBits ← 0;
unpackedWords ← [[] for i in range(len(outStreams))];
outCounter ← [0 for i in range(len(outStreams))];
for i ← 0..len(packedData) − 1 do
    j ← i  mod len(lumpLayout);
    b ← lumpLayout[j];
    if shiftRegBits == 0 then
        shiftReg ← next(packedWords);
        shiftRegBits ← 8 ∗ sizeof(word);
    end
    if b is not pad then
        v ← shiftReg & 1;
        for k ← 0..len(outStreams) − 1 do
            count ← outCounter[k];
            l ← count  mod len(outStreams[k]);
            if outStreams[k][l] is b then
                m ← count  mod (8 ∗ sizeof(word));
                if m = 0 then
                    outStreams[k].append(0);
                end
                outStreams[k][−1] ← outStreams[k][−1] | (v << m);
                outCounter[k] ← count + 1;
            end
        end
    end
    shiftReg ← shiftReg >> 1;
    shiftRegBits ← shiftRegBits − 1;
end
```

for accelerating bit manipulation routines, they do suffer from practical table and cache size limitations. In order to further exploit the implicit parallelism in the bit-unpacking problem, one must explore further aggregation of input bits beyond sizes convenient for LUTs.

While modern general-purpose processors operate efficiently on words of 64-512 bits, it is not obvious how to fully exploit this capability to operate upon a large number of input bits at once. This is because the *lump* may not place input bits destined for a particular output stream at convenient locations, and may not even have a length divisible by the word-size of the processor (e.g. 64 bits). In the extreme case, one has a number of potentially-complex, interleaved, repeating patterns of input bits to be re-organized into separate, contiguous output bit-streams.

Ideally, one should like to operate on so many *lumps* at a time that the number of bits yielded towards each output stream is a multiple of the word size. For instance, the *lump* shown in Table X could be aggregated 32 times to produce a unit of computation of size 1280 bits, from which each output stream draws an integer multiple of 64 bits.

Consider the general problem to be as follows. At design time, one is given a *LumpLayout* and an output specification. The output specification consists of a list of output bit-stream descriptions, each of which in turn consists of an ordered list of bit identifiers. These bit identifiers specify which bits from which streams are to be extracted (at run-time) from each *lump* and placed into the corresponding output queue. From these specifications, one one wishes to produce efficient code. Suppose that the *lump* has a packed size of ℓ bits and the CPU architecture provides a file of F SIMD registers, each of size R.

As an initial step, bits from the *LumpLayout* may be "matched up" with bits from the output specification. Each reduced bit-stream description now consists of a list of bit-indices within the *lump*, ranging from 0 to $\ell - 1$. Each such list need not be monotone increasing, but distinct output bit-stream descriptions should be disjoint.

According to the O.H.I.O. rule of thumb[2], one should prefer that at run-time, each size-R portion of the input stream is loaded into the processor exactly once, and each size-R portion of each output stream is stored to memory exactly once. Under the given parameterization, one may load up to $\lfloor F \times R/\ell \rfloor$ *lumps* into the CPU at a time.

Then one may compute

$$L = \operatorname{lcm}(R, \ell, \texttt{len}(o) \text{ for } o \in \text{output stream descriptions})$$

The resulting value of L is a batch size, in bits, corresponding to integral number of register-sized loads from the input stream and an integer number of register-sized stores to each output stream. If $L > R \times F$ does not fit in the SIMD register file, then further subdivision may be required; but for simplicity of exposition this case will not be considered in what follows.

Permutation Networks

The approach taken here is to consider this as an instance of large-scale bit permutation networks, as described in Knuth [20] p. 145–149 with further citations to Duguid, Le Corre, and Slepian. These networks consist of simple two-input, two-output "crossbar" subunits that conditionally swap pairs of inputs. Depending on which crossbars are enabled at run-time via bit-masks, the network can be reconfigured to perform any permutation of a given size.

The two-input, two-output crossbar is the base case $P(2)$ of an inductive construction for larger networks. In outline, the inductive step is as follows (see Figure 12 on page 147 of [20]). Suppose $P(n)$ is a network that supports all possible permutations of n elements. Then the network $P(2n)$ may be constructed from two copies of $P(n)$ plus $2n$ additional crossbars. First, divide the $2n$ inputs into two sublists A and B by parity. Next, apply a crossbar between A_i and B_i for $i = 1 \ldots n$. Denote the list of first outputs of these n crossbars by C and the list of second outputs by D. Pass C and D through separate instances of $P(n)$ to obtain lists E and F. Apply a crossbar between E_i and F_i for $i = 1 \ldots n$ to obtain G and H. Finally, interleave the elements of G and H to produce the output of $P(2n)$.

Proofs can be found in §3.3 of [21].

In effect, the network $P(2^d)$ places input bits on the corners of a hypercube in d dimensions. In stage i of execution, the coordinate axis numbered $k = (d-1) - |(d-1) - i|$ is selected, and crossbars are applied across each of the 2^{d-1} edges parallel with coordinate axis k. So, for instance, with $d = 3$, the 8 vertices would undergo swaps along x, then y, then z, then y, and finally x again.

The network $P(n)$ ultimately consists of $2\log_2 n - 1$ stages, each consisting of $n/2$ independent crossbar elements for an overall complexity of $C(n) = n\log_2 n - n/2$ crossbars. Since each crossbar has exactly one bit of configuration state (swap or don't swap), and a general permutation network must support $n!$ different permutations, such networks necessarily obey $C(n) \geq \log_2 n!$. For this reason, $P(n)$ is asymptotically as small as possible for general permutation networks: applying Stirling's approximation,

$$C(n) = n\log_2 n - n/2 \geq \log_2 n! \approx n\log_2 n - n/\ln 2 + 1/2\ln(2\pi n)$$

The gap is roughly $0.943n$.

This is not to say that networks tailored for *particular* permutations cannot be smaller. For the purposes of this paper, however, the only tool used to optimize the network beyond the construction of Knuth is the C compiler—which, it must be said, does a remarkably good job on permutations with substantial power-of-two structure.

Algorithm 2 finds crossbar parameters for $P(n)$ to achieve a particular permutation σ.

Auto-generating Vectorized Code

Performing a permutation on $(R \times F)$ bits stored across F SIMD registers requires two basic primitives: inter- and intra-register crossbar stages. Inter-register swaps operate on pairs of registers indexed i and j with $j - i$ a power of two. These swaps do not require bit-shift instructions: the requisite shifts are accomplished by indexing.

[2]Only Handle It Once

Algorithm 2: Cycle-chasing algorithm for configuring Knuth-style bit permutation networks.

Input : n : int
Input : $\sigma\,[0..(n-1)]$: int
Output: xbar $[0..(2\log_2 n - 2), 0..(n/2-1)]$: bool

```
1  logn ← log2 n;
2  xbar[:logn − 1, :] ← 2;                                                    /* sentinel value */
3  a[:] ← [0..(n − 1)];
4  p[:] ← σ[:];                              /* At each stage, p maps from layer output to layer input */
5  r[p] ← a[:];                                        /* r maps from layer input to layer output */
6  for d = 0..logn − 2 do
7  |  s ← n >> (d + 1);
8  |  pair ← (a & (s − 1)) | (a & ∼ (s | (s − 1))) >> 1;
9  |  xii ← xbar[d, :];                                                       /* By reference */
10 |  xjj ← xbar[ − (d + 1), :];                                             /* By reference */
11 |  i0 ← 0;
12 |  while i0 < n do
   |  |  /* Loop over potential cycle start points                                          */
13 |  |  i ← i0 ^ s;
14 |  |  j ← (xjj[0] & 1) * s;
   |  |  /* Chase cycle                                                                    */
15 |  |  while xii[pair[i]] == 2 do
16 |  |  |  xii[pair[i]] ← bool((i ^ j) & s) == xjj[pair[j]] & 1;
17 |  |  |  j ← r[i ^ s];
18 |  |  |  xjj[pair[j]] ← bool((i ^ j) & s) == xii[pair[i]] & 1;
19 |  |  |  i ← p[j ^ s];
20 |  |  end
21 |  |  i0 = ((i0 | s) + 1) & ∼ s;
22 |  end
23 |  lIdx ← a ^ (xii[pair] * s);          /* maps from layer input to next layer input; idempotent */
24 |  rIdx ← a ^ (xjj[pair] * s);         /* maps from next layer output to layer output; idempotent */
   |                                                  /* Transform invariants for next layer */
25 |  p[:] ← lIdx[p[rIdx]];                                  /* layer output -> layer input */
26 |  r[:] ← rIdx[r[lIdx]];                                  /* layer input -> layer output */
27 end
28 xbar[logn − 1, :] ← (p[::2] ≠ a[::2]);
29 xbar[:, :] ← bool(xbar[:, :]);
```

Intra-register swaps may be implemented with shifts, but this is not always necessary. Both x86-64 and ARM64 feature variations of a "shuffle" operation, whereby a single SIMD register may be viewed as an array of 64-, 32-, 16-, or 8-bit words, and a new vector may be constructed using a vector of indices into the register's component words. There does not seem to be any advantage to using the more coarse-grained variants of these instructions, so one need consider only the byte-shuffling instruction. These byte-shuffling primitives save two instructions per register in most intra-register stages. Shifts are only used for the innermost five stages, which require shifts by four, two, one, two, and four bits, respectively, and cannot be implemented with byte shuffles.

Because each crossbar stage involves only pairwise operations between registers or between bits in a register, there is enormous data parallelism within the inner loop. Modern compilers and processors are able to exploit this parallelism by re-ordering instructions, allowing them to be "pipelined" together for maximal efficiency.

The auto-generated inner-loop code contains no branches and compiles cleanly under `-std=c99` standard to nearly 100% SIMD instructions through the use of the semi-portable `vector_size` attribute, supported by both GCC and Clang compilers. Numerical results for code size are presented in the Experiments and Results section below.

RADIOLION

Across the domains of communications and PNT, wireless standards and waveforms are changing faster than ever before. Special-purpose digital logic designed for a particular version of a standard is often obsolete before it hits the market. To keep pace, designers incorporate software-defined receiver functionality. The Radionavigation Laboratory has had great success in the field with the RadioLynx, a custom, in-house GNSS-SDR platform. The RadioLynx is a four-channel receiver (two antennas $\times$ two bands). Its successor, the RadioLion, is a new low-cost, dual-antenna, triple-frequency receiver front-end. (Fig. 3)

Features and uses

The RadioLion's six individually-configurable channels operate at up to 44 Msps with a variety of real and complex quantization modes and depths up to three bits. The tools developed in this paper for optimizing bit-packing schedules and auto-generating unpacking code aid in making full use of the RadioLion's rich configuration space. Without these tools, developers would be unable to explore more than a small number of sampling strategies due to the need to hand-tailor code for each strategy to perform packing on the programmable logic device and unpacking on the general-purpose processor.

With simultaneous access to three GNSS bands, GNSS-SDR receivers built on the RadioLion will be better able to compensate for ionospheric delays [22] and multipath [23] than is possible with the RadioLynx. The device optionally includes one of a family of MEMS inertial measurement units (IMUs) specialized for applications in aerial robotics, automotive navigation, and virtual reality.

The RadioLion can be built with one of three timing sources: a low-cost temperature-compensated crystal oscillator (TCXO), a high-end programmable MEMS TCXO, or an external reference input. For timing output, it offers a small number of general-purpose I/O (GPIO) ports connected to each of the microcontroller and the programmable logic device, which can be used to provide clock or pulse-per-second outputs.

One motivator for this work has been the RadioLion's limited available storage for multiplexing. *Streams* are multiplexed into *lumps* on a Xilinx CoolRunner II Complex Programmable Logic Device (CPLD). CPLDs differ from FPGAs in their more limited logic capacity and are specialized for minimal pin-to-pin latency. Memory inside the CPLD is limited to 128 one-bit registers. Of these, 57 are reserved for I/O and not available to store sample bits during multiplexing. Facing such storage constraints on multiplexing, it was natural to ask how best to use available hardware resources to minimize the computational and power costs of software unpacking.

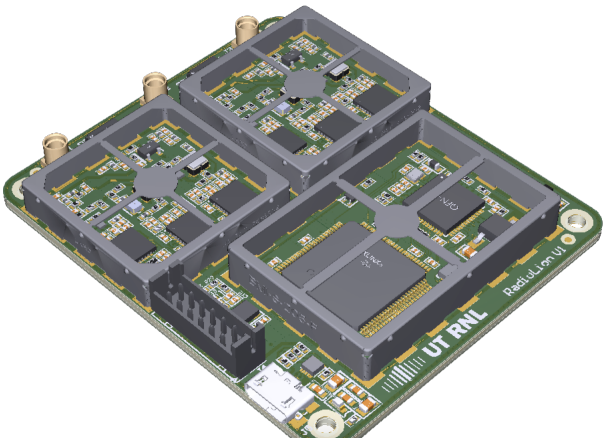


Fig. 3: RadioLion

Fig. 4: Quadcopter equipped with a RadioLion

RadioLion Applications: Small, Lightweight CDGNSS-SDR: The Carrier-phase Differential GNSS (CDGNSS) positioning technique provides users with centimeter-level accuracy. Autonomous Unmanned Air Vehicles (UAVs) are reliant on lightweight CDGNSS-SDRs for centimeter-level precise positioning. The RadioLion's compact size (5.6 cm $\times$ 6.5 cm) and low mass (24 grams) make it suitable for these micro-aerial (Fig. 4) platforms. Larger commercial urban aerial vehicles depend on CDGNSS for precise positioning to ensure safety of operation [24]. The RadioLion is able to counter challenging urban multipath environments by exploiting triple-frequency signal processing techniques. Additionally, the RadioLion's dual-antennas and onboard IMU provide robust defense mechanisms against GNSS spoofing [25]. The RadioLion has a place in non-traditional GNSS receiver markets such as augmented and virtual reality. Open World Virtual Reality (OWVR) is an outdoor Virtual Reality (VR) concept that combines precise GNSS positioning and a smartphone-grade inertial sensor to provide globally-referenced centimeter-and-degree accurate tracking of the VR headset [26].

"First light" experiments with the new hardware

The baseline configuration for the RadioLion includes all six channels with two-bit quantization. On both antennas, the L1 and L2 channels are sampled at 10 MHz and the L5 channels are sampled at 20 MHz. In the sampling nomenclature presented earlier, this strategy can be expressed as:

$$\boldsymbol{b} = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 & 2 \end{bmatrix}, \qquad \boldsymbol{d} = \begin{bmatrix} 2 & 2 & 1 & 2 & 2 & 1 \end{bmatrix}, \qquad \boldsymbol{p} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where the order is: primary antenna L1, L2, L5 followed by secondary antenna L1, L2, L5. The multiplexer packs data from the six *streams* into 64-bit words. Each 64-bit word consists of four sign and four magnitude bits from the L1 and L2 *streams*, and eight sign and eight magnitude bits from the L5 *streams*. The packed 64-bit word is formatted as follows:

Bits 0–31:

$$\overbrace{[S_1 \ldots S_4 M_1 \ldots M_4]}^{\textit{Stream 0}} \overbrace{[S_1 \ldots S_4 M_1 \ldots M_4]}^{\textit{Stream 1}} \overbrace{[S_1 \ldots S_4 M_1 \ldots M_4]}^{\textit{Stream 2}} \overbrace{[S_1 \ldots S_4 M_1 \ldots M_4]}^{\textit{Stream 3}}$$

Bits 32–63:

$$\overbrace{[S_1 \quad \ldots \quad S_8][M_1 \quad \ldots \quad M_8]}^{\textit{Stream 4}} \overbrace{[S_1 \quad \ldots \quad S_8][M_1 \quad \ldots \quad M_8]}^{\textit{Stream 5}}$$

For this baseline scheme, the demultiplexer operates on a batch of 512 bits comprised of eight-sequential 64-bit words from the multiplexer. This yields 32 sign and magnitude bits from the L1 and L2 *streams* and 64 sign and magnitude bits from the L5 *streams*. The demultiplexer unpacks the data bits into their corresponding output buffers by using a series of loads, shifts, masks, ORs, and stores. These instructions efficiently operate on nibbles and bytes rather than individual samples because the input data are packed in runs of particular bit-planes. Unpacking the baseline case (without exploiting parallelization) results in .81 operations per bit to demultiplex a batch of 512 bits into their corresponding output buffers. This number is the same for both x86-64 and ARM64 processors. It is more efficient than brute force and LUT based demultiplexing because the demultiplexing instructions operate on runs of signs and magnitude bits. The number of operations per bit can be further decreased by exploiting the parallelized bit-unpacking algorithm described above.

EXPERIMENTS AND RESULTS

Random permutations are the "worst case" scenario for bit-unpacking. The demultiplexing instructions cannot operate on multiple bits simultaneously because random permutations lack structure (e.g. runs of bit-planes). Permutations on strategically packed data with runs of bit-planes are much easier to perform. The permutation algorithm can permute multiple bits simultaneously because runs of bits are destined for the same output buffer. Additionally, they do not need to be shuffled because they are already ordered in the manner they should appear in the output buffer. This enables the demultiplexer to operate on entire runs of bits rather than operating on a single bit.

The efficiency of the parallelized unpacking algorithm presented earlier is highlighted in Tables XII on both x86-64 and ARM64 processors. These tables report the performance of the unpacking algorithm when permuting a batch of a random sequence of bits. Most of the operations are SIMD instructions, indicating that the implementation is fully vectorized. The four to five remaining instructions are attributed to loop counting and branching. The parallelized unpacking algorithm is remarkably more efficient than brute force and LUT based demultiplexing.

The results of a trade study comparing multiplexer storage requirements and demultiplexing efficiency on the baseline RadioLion sampling configuration is presented in Table XIII. The aggregation factor controls the lengths of runs of bits from the same bit-plane, but it also controls the size and thus cost of the multiplexer. Greater aggregation allows for longer runs of bit-planes to be packed, reducing the work that must be done during unpacking.

A direct comparison between the parallelized and non-parallelized unpacking scheme is made. The parallelized unpacking scheme takes .19 operations per bit on x86-64, which is a 4.3 times improvement over the non-parallelized .81 operations per bit. The best case scenario for $4\times$ aggregation factor on x86-64 is to use parallelized demultiplexing while operating on a batch size of 1024 bits. A batch size of 1024 bits allows the processor to operate on runs of 64-bits across each bit-plane.

The $1\times$ aggregation factor is the minimal storage case, yielding one sign and magnitude bit from the L1 and L2 *streams* and two sign and magnitude bits from the L5 *streams*. In this case, the multiplexer only needs four bits of storage. The

TABLE XII: Inner-loop complexity for auto-generated unpacking code on x86-64 and ARM64 architectures. To simulate worst-case conditions, a random bit-shuffle is used in place of the packing format.

Batch size (bits)	SIMD + Scalar Instructions		Instructions / Bit	
	x86-64	ARM64	x86-64	ARM64
512	156+4	335+3	0.32	0.66
1024	328+4	748+5	0.34	0.74
2048	691+4	1654+5	0.34	0.81
4096	1661+4	3806+5	0.41	0.93
8192	3988+4	9231+5	0.49	1.13

resulting serialized input stream is similar to the purely random case because the L1 and L2 *streams* do not produce runs of bit-planes. Even in this near "worst case" scenario, the parallelized unpacking algorithm performs significantly better than brute force, LUT, and non-parallelized unpacking. On the opposite end, the $8\times$ aggregation factor yields eight sign and magnitude bit from the L1 and L2 *streams* and 16 sign and magnitude bits from the L5 *streams*, but requires 116 bits of storage. This case is extremely efficient because the unpacking algorithm can permute the serialized input stream with byte-wise operations. Demultiplexing a $8\times$ aggregation factor is 6.5 times faster than a $1\times$ aggregation factor, but requires an extra 112 bits of storage.

TABLE XIII: Cost of unpacking in instructions-per-bit data with varying permutation and multiplexer size, assuming a baseline format.

Architecture	Batch size (bits)	Aggregation Factor			
		$1\times$	$2\times$	$4\times$	$8\times$
x86-64	512	.26	.21	.19	.04
	1024	.34	.25	.14	.06
	2048	.34	.31	.21	.07
	4096	.40	.34	.27	.10
	8192	.41	.40	.29	.12
ARM64	512	.49	.41	.42	.26
	1024	.62	.44	.36	.38
	2048	.61	.57	.39	.28
	4096	.63	.56	.50	.33
	8192	.64	.62	.52	.48
Multiplexer cost (bits)		4	20	52	116

In order to expedite unpacking, the multiplexer requires more storage to produce longer runs of bit-planes. However, the monetary cost of upgrading the CPLD in the RadioLion is non-negligible. The 128-bit variant of the CPLD costs 90% more than the 64-bit CPLD. Furthermore, the 256-bit CPLD costs 115% more than the 128-bit CPLD. The full bill of materials for the RadioLion may increase upto 16% when upgrading from the 128-bit to the 256-bit CPLD. The 128-bit CPLD was determined to be a "happy medium" for the RadioLion as it provides sufficient storage for copious bit-packing schemes while keeping costs low.

CONCLUSION

This paper concretely described two independent sub-proposals for extending version 1.0 of the ION GNSS-SDR Sampled Data Metadata Standard. The first addressed shortcomings in describing periodic, synchronous timing relationships among heterogeneous streams, such as staggered sampling. The second sub-proposal introduced means to override the default layout of stream sample bits within a *lump*. In addition to the proposed extensions to the ION Metadata Standard, this paper treated the problem of bit-packing and bit-unpacking more generally.

This paper contributed tools for the development of efficient new bit-packing schemes for next-generation bit-wise SDR receivers. Future low-cost hardware-constrained receivers require optimized bit-packing schemes. A parallelized bit-unpacking algorithm exploiting SIMD extensions was developed, tested, and analyzed. This scheme automatically generated compact packing logic and fast unpacking code. Experimental results on x86-64 and ARM64 architectures demonstrated the tools' efficiency and flexibility. Furthermore, the unpacking algorithm is not limited to GNSS SDR—any SDR platform that requires unpacking a single serialized stream into multiple output buffers can utilize it.

Finally, this paper presented the development of the RadioLion, a low-cost dual-antenna, tri-band GNSS-SDR front end developed in-house at the Radionavigation Laboratory. The RadioLion is a highly-reconfigurable front-end that is capable of automatically producing both VHDL code for packing and C code for unpacking RF data. This paper presented a trade study comparing multiplexer storage requirements to the efficiency of unpacking.

ACKNOWLEDGMENTS

Research support was provided by the U.S. Department of Transportation (USDOT) under the University Transportation Center (UTC) Program Grant 69A3552047138 (CARMEN), and by affiliates of the 6G@UT center within the Wireless Networking and Communications Group at The University of Texas at Austin.

REFERENCES

[1] Curran, J. T., Fernández-Prades, C., Morrison, A., and Bavaro, M., "The Continued Evolution of Software-Defined Radio for GNSS," *GPS World*, Vol. 29, 2018, pp. 43–49.

[2] Gunawardena, S., Pany, T., and Curran, J., "ION GNSS software-defined radio metadata standard," *Navigation, Journal of the Institute of Navigation*, Vol. 68, No. 1, 2021, pp. 11–20.

[3] Heinrichs, G., Restle, M., Dreischer, C., and Pany, T., "NavX-NSR–A Novel Galileo/GPS Navigation Software Receiver," *Proc. ION GNSS 2007*, Institute of Navigation, Fort Worth,Texas, Sept. 2007.

[4] Pany, T., Kraus, T., Schütz, A., Arizabaleta, M., Dötterböck, D., and Damph, J., "Recent Enhancments of the Multi-Sensor Navigation Analysis Tool (MuSNAT)," *Proceedings of the 34nd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2021)*, 2021.

[5] Humphreys, T. E., Ledvina, B. M., Psiaki, M. L., and Kintner, Jr., P. M., "GNSS Receiver Implementation on a DSP: Status, Challenges, and Prospects," *Proceedings of the ION GNSS Meeting*, Institute of Navigation, Fort Worth, TX, 2006, pp. 2370–2382.

[6] Lightsey, E. G., Humphreys, T. E., Bhatti, J. A., Joplin, A. J., O'Hanlon, B. W., and Powell, S. P., "Demonstration of a Space Capable Miniature Dual Frequency GNSS Receiver," *Navigation*, Vol. 61, No. 1, Mar. 2014, pp. 53–64.

[7] Humphreys, T. E., Bhatti, J., Pany, T., Ledvina, B., and O'Hanlon, B., "Exploiting multicore technology in software-defined GNSS receivers," *Proceedings of the ION GNSS Meeting*, Institute of Navigation, Savannah, GA, 2009, pp. 326–338.

[8] Humphreys, T. E., Murrian, M. J., and Narula, L., "Deep-Urban Unaided Precise Global Navigation Satellite System Vehicle Positioning," *IEEE Intelligent Transportation Systems Magazine*, Vol. 12, No. 3, 2020, pp. 109–122.

[9] Narula, L., LaChapelle, D. M., Murrian, M. J., Wooten, J. M., Humphreys, T. E., Lacambre, J.-B., de Toldi, E., and Morvant, G., "TEX-CUP: The University of Texas Challenge for Urban Positioning," *Proceedings of the IEEE/ION PLANSx Meeting*, 2020.

[10] Ledvina, B. M., Psiaki, M. L., Powell, S. P., and Kintner, Jr., P. M., "Bit-Wise Parallel Algorithms for Efficient Software Correlation Applied to a GPS Software Receiver," *IEEE Transactions on Wireless Communications*, Vol. 3, No. 5, Sept. 2004.

[11] Psiaki, M. L., "Real-Time Generation of Bit-Wise Parallel Representations of Over-Sampled PRN Codes," *IEEE Transactions on Wireless Communications*, Vol. 5, No. 3, March 2006, pp. 487–491.

[12] Ledvina, B., "Efficient Real-Time Generation of Bit-Wise Parallel Representations of Oversampled Carrier Replicas," *Aerospace and Electronic Systems, IEEE Transactions on*, Vol. 47, No. 4, Oct. 2011, pp. 2921–2933.

[13] Borio, D. and Gioia, C., "GNSS interference mitigation: A measurement and position domain assessment," *NAVIGATION*, Vol. 68, No. 1, 2021, pp. 93–114.

[14] Pany, T., Dötterböck, D., Gomez-Martinez, H., Hammed, M. S., Hörkner, F., Kraus, T., Maier, D., Sánchez-Morales, D., Schütz, A., Klima, P., et al., "The Multi-Sensor Navigation Analysis Tool (MuSNAT)–Architecture, LiDAR, GPU/CPU GNSS Signal Processing," *Proceedings of the 32nd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2019)*, 2019, pp. 4087–4115.

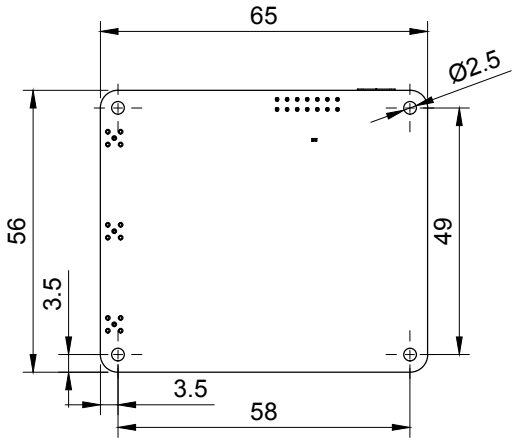
[15] Hegarty, C., "Analytical model for GNSS receiver implementation losses," *Navigation, Journal of the Institute of Navigation*, Vol. 58, No. 1, 2011, pp. 29.

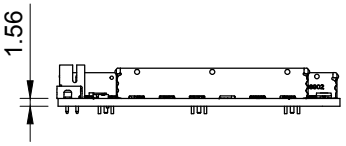
[16] Humphreys, T. E., "Interference," *Springer Handbook of Global Navigation Satellite Systems*, Springer International Publishing, 2017, pp. 469–503.

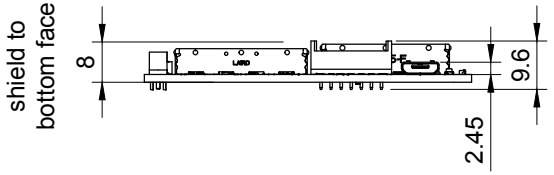
[17] Zhang, Y. D., Amin, M. G., and Wang, B., "Mitigation of sparsely sampled nonstationary jammers for multi-antenna GNSS receivers," *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2016, pp. 6565–6569.

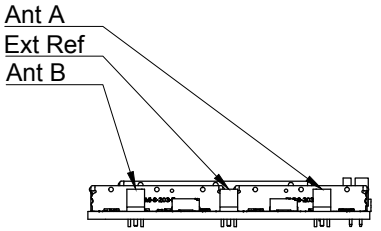
[18] Amin, M. G., Wang, X., Zhang, Y. D., Ahmad, F., and Aboutanios, E., "Sparse arrays and sampling for interference mitigation and DOA estimation in GNSS," *Proceedings of the IEEE*, Vol. 104, No. 6, 2016, pp. 1302–1317.

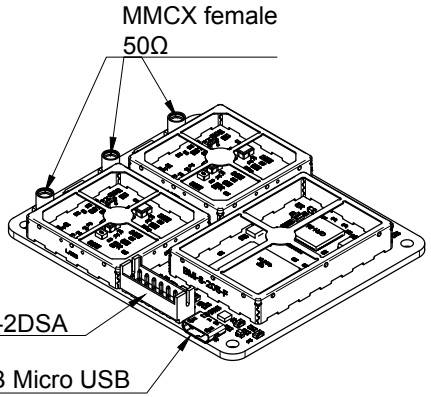
[19] Cree, M. J., "An Exploration of Using the Intel AVX2 Gather Load Instructions for Vectorised Image Processing," *2018 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, IEEE, 2018, pp. 1–9.
[20] Knuth, D., *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*, No. pt. 1, Pearson Education, 2014.
[21] Beneš, V., *Mathematical Theory of Connecting Networks and Telephone Traffic*, ISSN, Elsevier Science, 1965.
[22] Elsobeiey, M., "Precise Point Positioning Using Triple-Frequency GPS Measurements," *The Journal of Navigation*, Vol. 68, No. 3, 2015, pp. 480–492.
[23] Strode, P. R. and Groves, P. D., "GNSS multipath detection using three-frequency signal-to-noise measurements," *GPS Solutions*, Vol. 20, No. 3, 2016, pp. 399–412.
[24] Holden, J. and Goel, N., "Uber Elevate: Fast-Forwarding to a Future of On-Demand Urban Air Transportation," *Uber Inc., San Francisco, CA*, 2016.
[25] Psiaki, M. L. and Humphreys, T. E., "GNSS Spoofing and Detection," *Proceedings of the IEEE*, Vol. 104, No. 6, 2016, pp. 1258–1270.
[26] Humphreys, T. E., Kor, R. X. T., and Iannucci, P. A., "Open-World Virtual Reality Headset Tracking," *Proceedings of the ION GNSS+ Meeting*, Online, 2020.

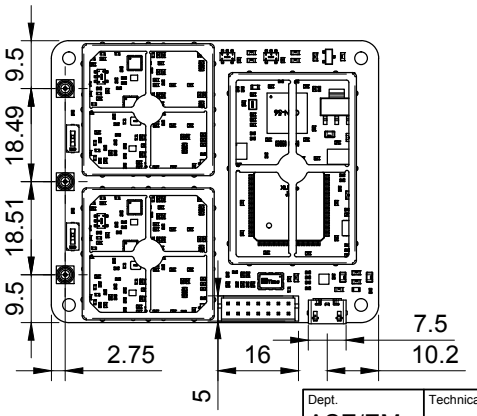












All dimensions in mm

Dept. ASE/EM	Technical reference	Created by Peter A. Iannucci 8/20/21	Approved by
		Document type Mechanical	Document status DRAFT
		Title RadioLion	DWG No.
			Rev. 0 / Date of issue 8/20/21 / Sheet 1/1