IPN Progress Report 42-148 February 15, 2002

Turbo-Decoder Implementation for the Deep Space Network

K. Andrews,[1] V. Stanton,[1] S. Dolinar,[1] V. Chen,[2] J. Berner,[2] and F. Pollara[1]

Since their discovery in 1993, turbo codes have been found to provide excellent performance with relatively low complexity. Turbo codes have been added to the Consultative Committee for Space Data Systems (CCSDS) standards for telemetry channel coding, and turbo encoders are being included in spaceflight hardware and software. This article describes the design of the prototype turbo decoder for the Deep Space Network (DSN). The decoder uses eight digital signal processors (DSPs) operating together to perform the decoding, frame synchronization, and control and interfacing. The key algorithms are described here, with the modifications and optimizations used in the assembly language implementation.

I. Introduction

With the perspective provided by several years of research, the discovery of turbo codes in 1993 [1] is becoming accepted as a milestone in the field of error-correcting codes. Until their discovery, research had been devoted to developing codes with large minimum distance and finding maximum-likelihood decoders for them. This path was yielding diminishing performance improvements with decoders of ever-increasing complexity. Berrou, Glavieux, and Thitimajshima demonstrated two facts with their turbo codes: that a large minimum distance is not required if the distance spectrum is sufficiently sparse at the low end, and that a sub-optimum decoder can reduce the decoding complexity so dramatically and with sufficiently small performance losses that, when combined with a good code, it is superior to maximum-likelihood decoding methods in both performance and complexity. The potential advantages of turbo codes for deep-space communications were soon recognized [2], and JPL has contributed substantially to the research.

Current research is extending the range of capabilities of sub-optimal iterative decoders in many directions, including for high-rate codes, spectrally efficient applications, and low-complexity decoding. The turbo codes discovered first continue to stand out as excellent choices for deep-space communications. They are low-rate codes, ranging from $R = 1/2$ down to $R = 1/6$ and below, and are most applicable to noisy channels with E_b/N_0 as low as -0.2 dB. They are designed for additive white Gaussian noise (AWGN) channels and for binary phase-shift keyed (BPSK) modulation, the method of choice when

[1] Communications Systems and Research Section.

[2] Communications Ground Systems Section.

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

bandwidth is not constrained. Turbo encoders are sufficiently simple that they can be implemented readily in hardware or software on a spacecraft [3].

With these attributes, and the confidence instilled by a few years of research with the codes, JPL worked with the Consultative Committee for Space Data Systems (CCSDS), an international standards body, to add them to the standards in May of 1999. These are described in [4], so we merely summarize the key parameters here. The standardized turbo encoder, shown in Fig. 1, consists of two 16-state convolutional encoders, connected with an algorithmically described interleaver. The constituent convolutional encoders are terminated independently at the end of each block. Code rates close to 1/2, 1/3, 1/4, and 1/6 are achieved by appropriate puncturing of the output symbols.[3] Block lengths of 1784 through 8920 information bits are specified, to match those of the $(255, 223)$ Reed–Solomon code with interleaving depths of 1 through 5. Attached synchronization markers (ASMs) are appended to each encoded block for synchronization recovery.

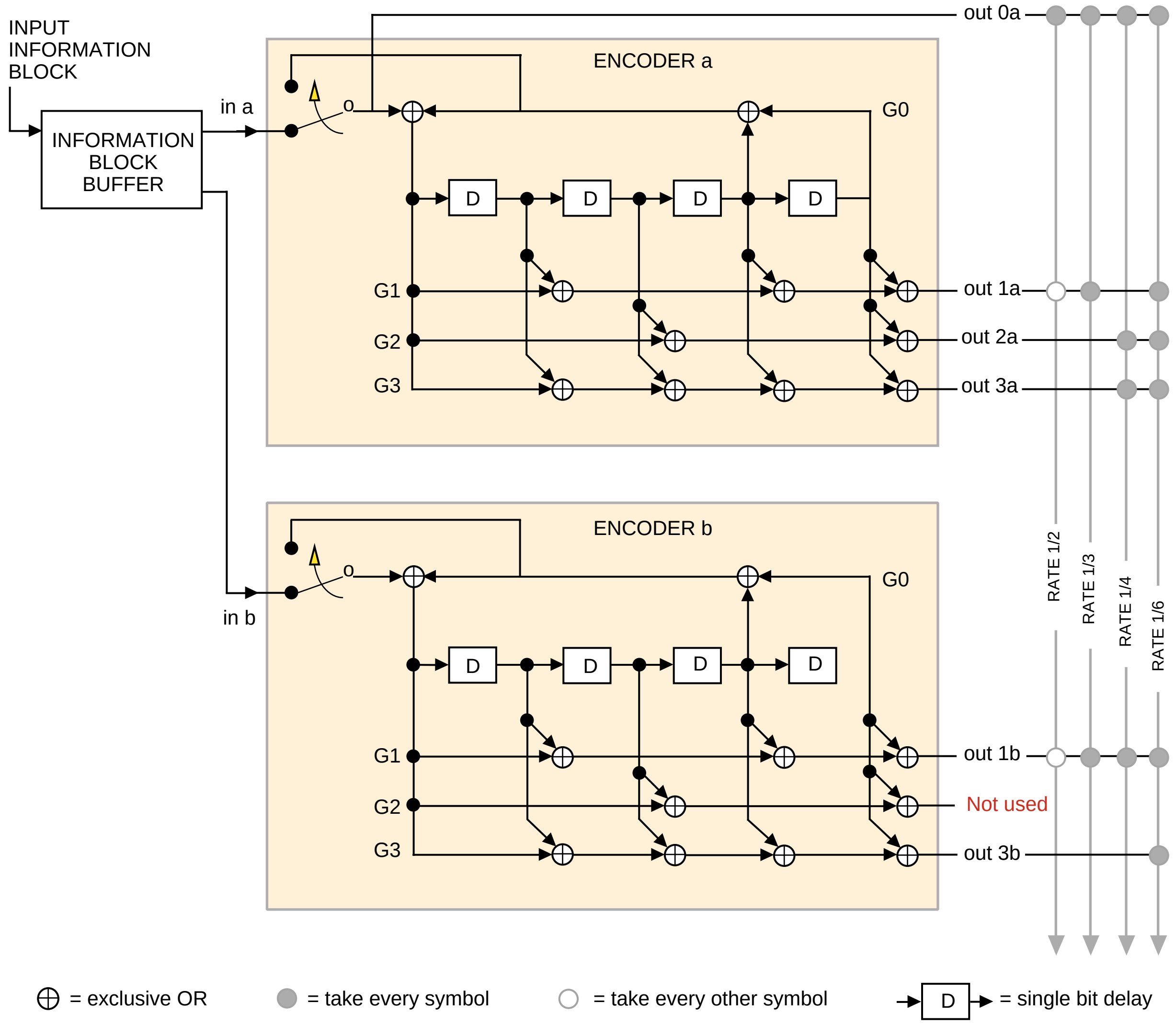

Fig. 1. CCSDS turbo encoder (reproduced from [4]).

[3] The CCSDS turbo codes use four termination bits to return the constituent convolutional encoders to their zero states, reducing the true rate to $N/(N+4)$ times the nominal rate, where N is the block length.

Figure 2 compares the performance of representative CCSDS-standardized turbo codes to familiar codes currently in use. Figure 2(a) shows several codes with rates near 1/2: the constraint-length-7, rate-1/2 convolutional code concatenated with the (255, 223) Reed–Solomon (RS) code using interleaving depth 1 (rate 0.4373, length 1784), the same code with interleaving depth 5 (rate 0.4373, length 8920), and the (rate-0.4989, length-1784) and (rate-0.4998, length-8920) turbo codes. Figure 2(b) compares several rate-1/6 codes: the constraint-length-15, rate-1/6 convolutional code concatenated with the (255, 223) Reed–Solomon code using interleaving depth 1 (rate 0.1458, length 1784), the same code with interleaving depth 5 (rate 0.1458, length 8920), and the (rate-0.1663, length-1784) and (rate-0.1666, length-8920) turbo codes.

Several efforts are currently under way to include turbo encoders on board spacecraft. The System Interface Assembly (SIA), under the aegis of X2000, and the Spacecraft Transponder Module (STM) projects are both hardware application-specific integrated-circuit (ASIC) implementations of the encoder. Several mission teams, including the Mars Exploration Rover (JPL) and Mercury Surface, Space Environment, Geochemistry and Ranging (MESSENGER) (The Johns Hopkins University Applied Physics Laboratory), are considering software implementations. By 2003, each Deep Space Network (DSN) station will be equipped with a turbo decoder for use with the Block V receivers. The prototype decoder, which is the subject of this article, is essentially complete.

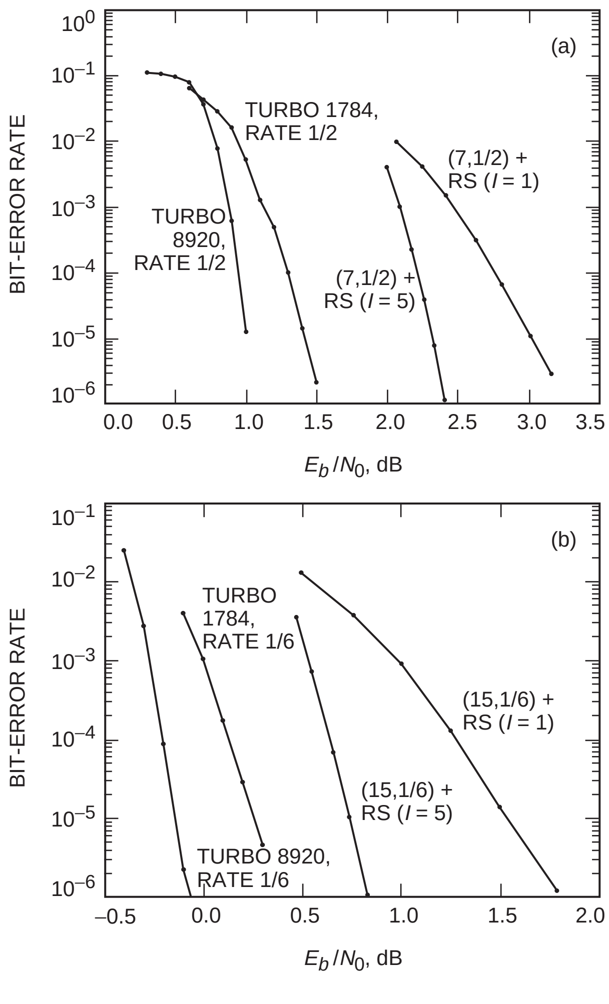

Fig. 2. Bit-error rates for (a) several codes with rates near 1/2 and (b) several codes with rates near 1/6.

The turbo-decoder system is implemented in software, with tasks distributed among eight high-performance Texas Instruments TMS320C6201 digital signal processors (DSPs) on circuit boards in a Versa Module Eurocard (VME) chassis. In the next section, the hardware and the partitioning of the software among the DSPs are described. In the remaining sections, we describe the key algorithms and aim to give a sense of their implementations through a couple of examples. The tailoring of an algorithm to use the available memory resources efficiently is exhibited with "windowing" in the turbo-decoder algorithm. Assembly language implementation is illustrated with the core of the frame-synchronization algorithm.

II. System Architecture

As shown in Fig. 3, the turbo-decoder system's purview extends significantly beyond that of simply decoding the error-correcting code. The input data are a stream of symbols, quantized into 8-bit digital numbers, normally generated by the Block V receiver; for self-tests, a "loopback" data source built into the interface board can be used. Alternatively, a DSP can generate simulated noisy codewords for more extensive performance analysis. The system attaches 24-bit time tags to each symbol, documenting their arrival time to the nearest 0.1 microsecond (modulo 1 second) for time-sensitive applications. After scaling the symbols, the system establishes frame synchronization by searching for the known symbol pattern of the attached synchronization markers with a correlating detector. It resolves the 180-deg phase ambiguity present in BPSK suppressed-carrier modulation according to the detected ASM polarity, and

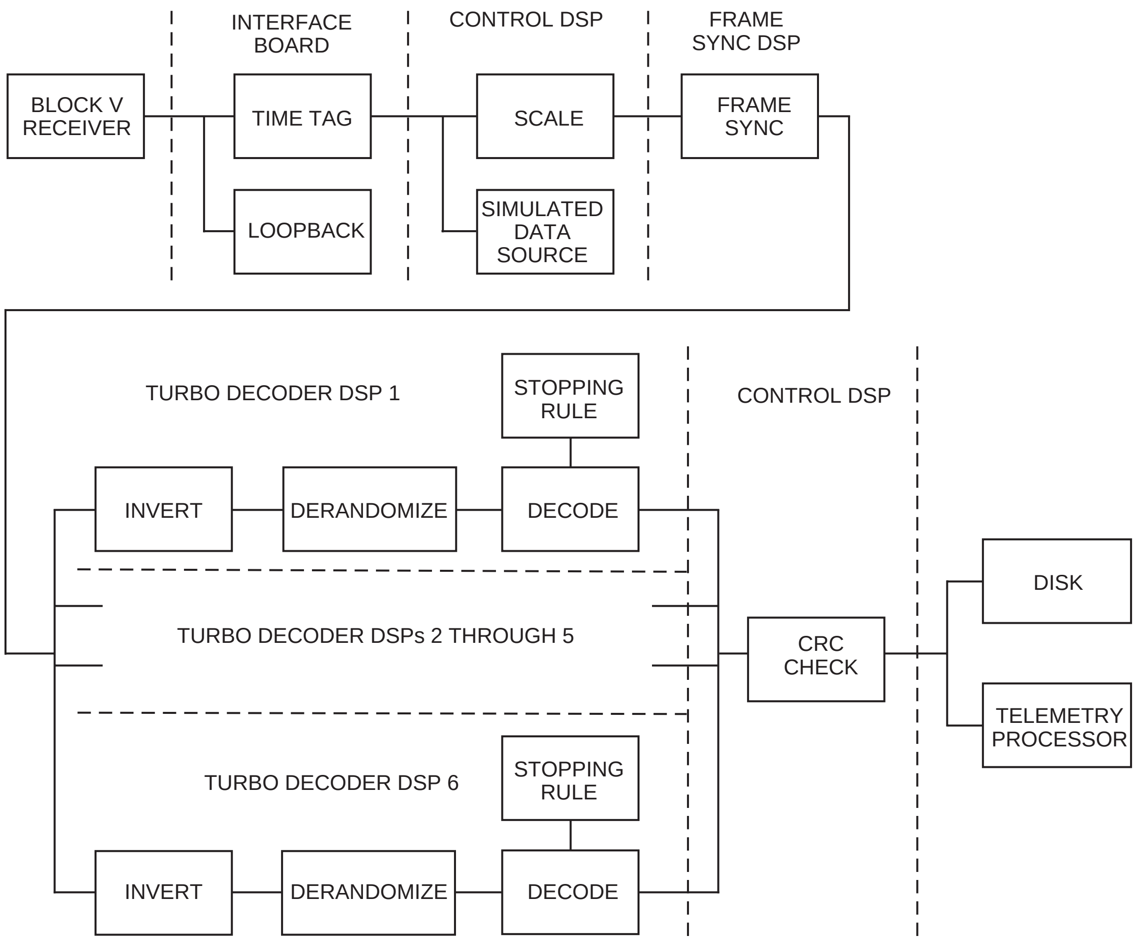

Fig. 3. Block diagram of the turbo decoder system.

inverts the data if necessary. If a pseudo-randomizer was used when encoding the data (to ensure a sufficient bit transition density for symbol synchronization at the receiver), this is taken care of next. Turbo decoding is then performed. This is an iterative process, and by selecting the number of iterations performed, one can trade bit-error rate against decoding complexity and, hence, against the sustainable data rate. The number of iterations can be specified manually or a “stopping rule” can be used to select this number automatically on a block-by-block basis, based on how successful the decoder believes it is. Turbo decoders, unlike Reed–Solomon decoders, generally cannot detect whether a decoding error has occurred. When greater error-detection power is desired, turbo codes are concatenated with an outer cyclic redundancy check (CRC) code. After turbo decoding, the system can verify the CRC. The resulting decoded data, with a header containing auxiliary information, are delivered to the data sink. In normal operation, this is the telemetry processor (TLP); for testing of the prototype system, the output data can also be fed directly to a hard disk for later analysis. Error-rate statistics are also accumulated.

A few years ago, a decoder was built using field programmable gate arrays (FPGAs) for a complex turbo code based on two 64-state convolutional codes. This project found that the decoding algorithm was sufficiently complex and required enough accesses to mass (off-chip) memory that it could not be parallelized effectively. Accepting that the decoding algorithm must be performed essentially serially, commercial processors provide higher clock speeds at feasible development costs than a custom design could. Digital signal processors are general-purpose microprocessors that have been optimized for computationally intensive tasks. The Texas Instruments TMS320C6201 (’C6201) is among the most capable fixed-point DSPs available today [8]. This DSP uses an entirely new architecture unencumbered with backwards-compatibility issues, and the outlook is favorable for a series of higher performance devices in the same family within the next few years.

A few manufacturers offer VME circuit boards with several ’C6201 DSPs, memory, and support circuitry. These combine excellent signal-processing resources with the industry-standard high-performance VME bus, and provide the appropriate hardware for system development. The prototype is built with two Monaco VME boards manufactured by Spectrum Signal Processing, each of which provides four DSPs, memory, and VME interfaces. There are two additional boards in the VME chassis. A general-purpose single-board computer provides a rudimentary user interface, disk storage, and Ethernet access. A simple custom interface board captures the quantized received symbols from the Block V receiver, attaches a time tag to each one, and passes them to the DSPs. The interface board can also be commanded to generate a symbol sequence for testing without an external data source.

Each DSP runs at 200 MHz and can execute up to eight instructions simultaneously in every clock cycle, although there are complex restrictions on their selection. Each DSP also provides four direct memory access (DMA) channels and a collection of peripherals. While these resources are substantial, it is difficult to use them efficiently: compilers are rather poor at this optimization, and the assembly language programmer finds it maddening. Because speed is a primary goal for the turbo decoder, most of the code is written in assembly language and meticulously hand optimized. Because the optimization makes the resulting code nearly unreadable and quite unmodifiable, segments of the code that are not time-critical are written either in C or in assembly language without optimization.

The primary computational tasks of the decoder are frame synchronization, turbo decoding, and control and coordination. These tasks are divided among the eight DSPs as shown in Fig. 3. The control DSP handles data transfer and coordination among the DSPs and performs various other minor tasks. The frame-synchronizer (sync) DSP identifies the attached synchronization markers to determine the location and polarity of each turbo-coded block. These blocks are distributed to six decoder DSPs, each of which runs identical software and performs the iterative turbo decoding. At the 200-MHz clock rate, each decoder DSP can maintain a data rate of 50 kbits/second while performing 10 iterations on each block. This gives a system throughput of 300 kbits/second at 10 iterations, and this increases proportionally as the clock rate is increased and as the average number of iterations is decreased with a stopping rule.

The basic skeleton of the turbo-decoder algorithm is written in C to preserve readability of the code and to permit minor modifications; all the computation routines are written in optimized assembly. All the software for the frame-synchronizer DSP is written in assembly; the skeleton is not optimized, but all the computation is. Software for the control DSP is written in assembly language because it deals extensively with interrupts and hardware resources, which higher level languages do not handle as readily; the assembly is not optimized because speed is not critical.

III. The Turbo-Decoder DSPs

Turbo decoding is performed iteratively according to the equations given in the Appendix, as shown in Fig. 4. In the first half-iteration, an attempt is made to decode the first constituent convolutional code from its received data, y, independent of information $\tilde{y}$ for the second code, using the BCJR soft-decision algorithm (named for Bahl, Cocke, Jelinek, and Raviv [5], and also known as the "forward–backward" algorithm). This algorithm works forwards through the block, iteratively computing $\alpha_j(s)$ for each message bit j and each trellis state s, and backwards, iteratively computing $\beta_j(s)$ similarly. These are combined to form reliabilities λ_j, each representing the estimated probability that the jth bit is a binary 1. These probabilities are modified by subtracting a term common to both constituent decoders, and the resulting extrinsic information μ_j is interleaved (permuted according to permutation π) and used as an aid in decoding the second convolutional code during the second half-iteration. A new set of extrinsic information is computed, deinterleaved by π^{-1}, and passed back to the first decoder for use in the second iteration. This process is repeated for some number of iterations, and usually the message estimates converge to the correct decoded sequence. The number of iterations can be fixed, or determined by a stopping rule based on the reliabilities at the end of each iteration. Upon completion, a final message estimate $\hat{m}$ is computed.

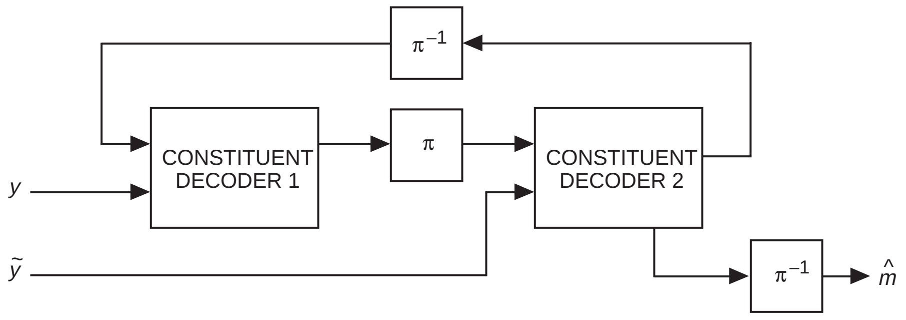

Fig. 4. The iterative turbo decoder.

There are two common, theoretically equivalent implementations for the decoder. In the "probability" form, extrinsic $\bar{\mu}_j(+1)$ and $\bar{\mu}_j(-1)$ are estimated probabilities. In the "logarithmic" form, the extrinsic μ_j represents a log likelihood ratio (LLR):

$$2\mu_j = \log \frac{\bar{\mu}_j(+1)}{\bar{\mu}_j(-1)}$$

Similar transformations are made to each of the variables, and the equations are modified accordingly. There is a developing consensus in the research community that the logarithmic form is less sensitive to the rounding errors introduced by fixed-point arithmetic and, depending on implementation details, the required arithmetic may be simpler. The Appendix gives the turbo-decoding algorithm equations in logarithmic form.

While implementation of the algorithm is essentially straightforward, efficient memory use and numerical precision require consideration. Each DSP contains 64 kbytes of internal data random access memory

(RAM) that it can access in a single clock cycle[4]. Megabytes of off-chip memory are available, but they impose 16, 18, or more wait states, entirely wasting that many clock cycles. Thus, it is very desirable to perform all computation using internal memory. Simultaneously, the data for future computations can be transferred from external to internal memory by DMA in the background, without imposing any significant penalty on the central processing unit (CPU) speed. Because the CCSDS codes use 16-state convolutional encoders, decoding a codeword of N message bits requires regular access to well over $16N$ intermediate values (the β's). The internal memory is too small to hold this many 16-bit quantities, even for the smallest turbo codes.

To address this memory problem, we break each block of N data bits into k "windows" as shown in Fig. 5. During each half-iteration, the windows are processed sequentially from left to right. The inductive "forward" α computation for each window can be initialized from the final values computed in the previous window. Except for the last window, however, initial values for the "backwards" β computations are not available. Research has shown that by computing β's for some trellis sections beyond those within the window, the algorithm will find approximately the correct initial conditions. Generally, when more than $5 \times 2^{\nu}$ extra β's are computed (where $\nu = 4$ is the memory of the convolutional codes in the CCSDS standard), the performance loss is negligible; below that, there is a trade-off between computation and performance. Information about the trellis state at the end of the last window is provided by the termination symbols. Because these symbols are generated separately by the two constituent convolutional encoders, no extrinsic information is generated by iterative decoding. Thus, the decoder uses them to compute initial β values for the last window before beginning the iterations, and the termination symbols are not used again. In Fig. 5, the sequence of α and β computations is shown from top to bottom, below the partitioned block. The amount of available memory determines $V + W$, the number of β values that are stored, where W is the window size and V is the overlap length used for determining the initial β values.

Turbo decoding requires that the input symbols be scaled according to the estimated signal-to-noise ratio (SNR). Because all computation is done using fixed-point arithmetic for speed, scaling before the quantization is also required to preserve the desired precision. The Block V receiver takes most of this burden because it has the SNR estimate available; the turbo decoder can optionally provide an additional

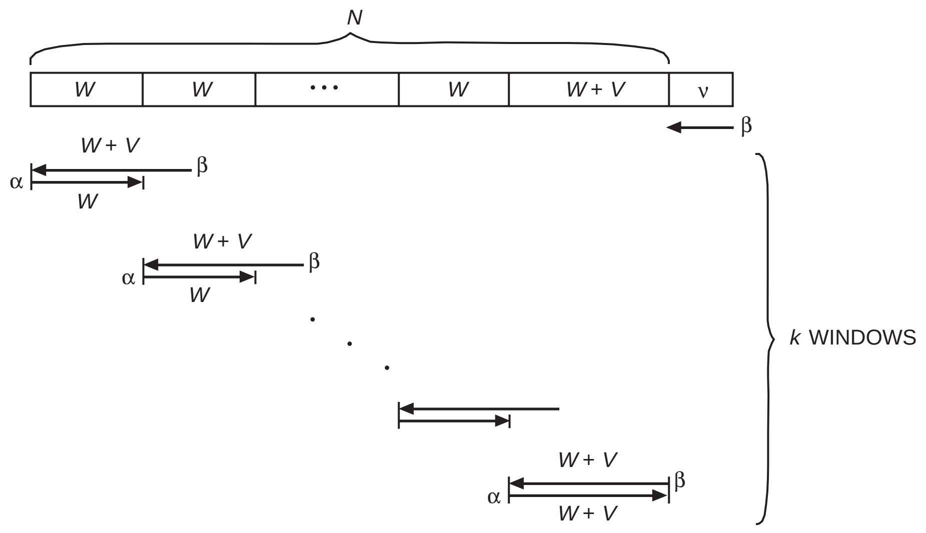

Fig. 5. Division of a turbo codeword into windows.

[4] More precisely, accesses to internal data RAM require no wait states (assuming bank hits are avoided), so while load instructions involve four delay slots, the carefully written program executes at full speed.

constant scale factor if desired. Together, they set the nominal BPSK values to $\pm 10/\sigma^2$, where σ is the standard deviation of the noise. These values are quantized, and all following arithmetic is performed using integers. For typical values of σ, this preserves about three bits of "soft" information.

As decoding proceeds, the extrinsic information exchanged between decoders usually grows in magnitude with each iteration, and this must be restricted to prevent numeric overflow. Moreover, the α and β values inductively computed by the BCJR algorithm must be renormalized regularly to prevent numeric overflow, and the required renormalization rate is proportional to the average magnitude of the extrinsics. There are anecdotal research results that show that clipping the extrinsic values to some limit also improves decoder performance and increases immunity to outlying received noisy symbols (due, perhaps, to non-Gaussian contributions to the noise). For all these reasons, clipping of the extrinsics is implemented, with a programmable level to permit making trades among performance, required renormalization rate, and immunity to outlying symbol values.

A "stopping rule" proves valuable for increasing a turbo decoder's average speed. When decoding a block of data, the reliabilities of the message bit estimates generally improve with each iteration. Depending on the particular values of the received noise, the estimated message is often entirely correct after a few iterations, but may not be until 10 or more, or never, in which case a decoding error is unavoidable. One could perform a fixed number of iterations on each block (typically 10), knowing that most decodable blocks will be correctly decoded by then. A superior method is to iterate only until the decoder is sufficiently "confident" in its estimates, up to some maximum number of iterations. In this way, most blocks are decoded in a few iterations, and the time saved can be used to perform extra iterations on the difficult blocks, resulting in a better decoder, or a faster one, or both.

This technique requires a stopping rule that determines when a sufficient confidence has been reached. Balancing implementation issues against performance, we consider Rule S_2 from [6], which stops decoding when

$$\min_{0<j\leq N}\left[|\lambda_{j,2}^{i}|\right] \geq \theta, \quad 1 \leq i \leq I_{max} \tag{1}$$

where N is the number of message bits in the block, $\lambda_{j,2}^{i}$ is the reliability of the jth bit after i complete iterations, θ is a predetermined threshold, and I_{max} is a fixed upper limit on the number of iterations. That is, decoding is stopped when all the reliabilities generated by the second constituent decoder have a magnitude exceeding θ. In implementation, we use a variant. During the last half-iteration performed, the decoder must store reliabilities λ_j rather than extrinsics μ_j, so to avoid preserving both, it helps to identify the last half-iteration in advance. To permit this, the stopping rule is evaluated on the alternate half-iterations (i.e., using $\lambda_{j,1}^{i}$), and when satisfied, decoding is terminated after the following half-iteration. The difference in algorithms can be approximately compensated for by adjusting θ appropriately.

With a stopping rule, any two of the four parameters of frame-error rate (FER), decoder speed, SNR, and threshold θ can be determined from the remaining two. In Fig. 6, frame-error rate is plotted parametrically against normalized decoder speed (measured as the reciprocal of the average number of iterations performed), as the SNR and θ are varied. For a fixed SNR, we see that, as θ is reduced from infinity, a marked increase in speed is realized with virtually no penalty in FER; then there is a "knee" in each curve, after which the FER increases rapidly with modest increases in speed. By choosing $\theta = 100$ (in its essentially arbitrary units), the decoder operates near these knees, achieving the best speed consistent with a minimal FER penalty.

In Fig. 7, the normalized speed is plotted parametrically against the SNR for four of the CCSDS codes as θ is varied, with the FER fixed at 10^{-4}. The prototype decoder's speed is slightly above 3000 kbit-iterations/second, so, for example, it can decode the (1784,1/6) turbo code with $E_b/N_0 = 0.4$ dB at $0.25 \times 3000 = 750$ kbits/second.

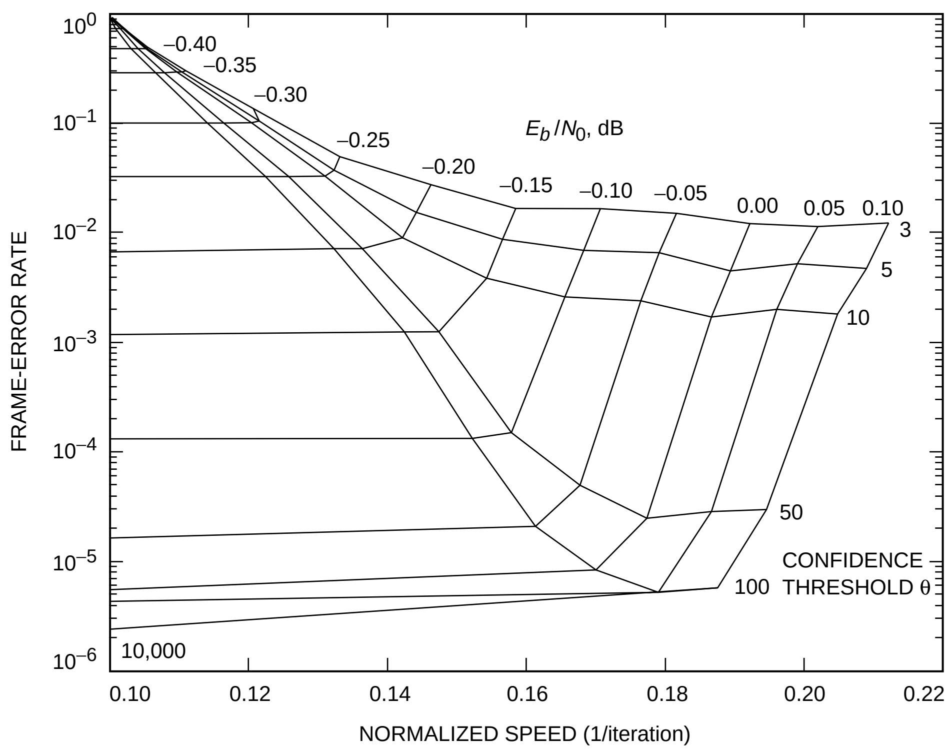

Fig. 6. Decoder speed and FER, as functions of SNR and stopping rule threshold, for the (*N* = 8920, *R* = 1/6) turbo code.

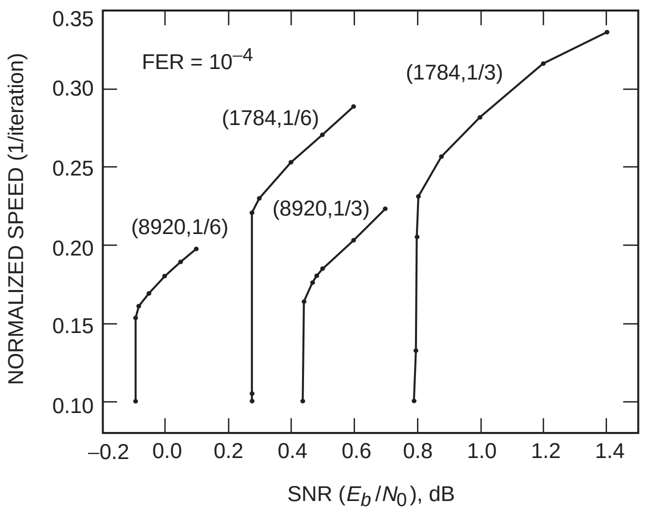

Fig. 7. Decoder speed achievable at a given SNR for four turbo codes.

IV. The Frame-Synchronization DSP

The attached synchronization markers (ASMs) interposed between turbo-code data blocks are optimally detected by correlation. The marker $\mathcal{A}$ is a sequence of F symbols $a_i \in \{+1, -1\}$, $0 \leq i < F$. If detection must be performed from a single marker, the optimum estimate of its location is at the peak correlation,

$$l = \underset{j \in [0, B-1]}{\operatorname{argmax}} \; c_j \tag{2a}$$

where

$$c_j = \sum_{i=0}^{F-1} a_i y_{i+j} \tag{2b}$$

and B is the length of the turbo codeword plus that of the sync marker. If the autocorrelation function of the marker is approximately random except for the delta function at zero offset, it is straightforward to show that in the presence of additive white Gaussian noise the probability of correctly identifying the marker's location by this method is

$$P(\text{correct}) = \int_{-\infty}^{\infty} \left[1 - \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2F(1+\sigma^2)}} \right) \right]^{B-1} \frac{1}{\sigma\sqrt{2\pi F}} e^{-(x-F)^2/2F\sigma^2} dx$$

where

$$\sigma = \frac{1}{\sqrt{2RE_b/N_0}}$$

and R is the rate of the code. With $P(\text{error}) = 1 - P(\text{correct})$, an upper bound on the probability of error is

$$P(\text{error}) \leq \frac{B-1}{2} \operatorname{erfc}\left(\frac{\sqrt{F}}{\sqrt{2(1+2\sigma^2)}} \right)$$

This union bound approximation is within 10 percent of the actual value when the probability of error is below 10^{-5} and gradually deteriorates for higher error probabilities. If symbol slips are rare, the synchronization markers repeat once every B symbols. At the expense of some computation, latency, and sensitivity to symbol slips, we can match several sequential markers to improve the probability of correct identification. The performance equations must be modified only by replacing F with nF, where n is the number of patterns matched. For the turbo code with $N = 1784$ information bits, $F = 96$ synchronization symbols, and $B = 5460$ output symbols (so $R \approx 1/3$), the probability of incorrect detection is plotted against bit SNR in Fig. 8 for matching one, two, three, and four markers. We see that if only one pattern is matched, the errors introduced by synchronization failures can overwhelm the turbo-decoding errors.

We may match four sync patterns by defining

$$c'_j = c_{j-B} + c_j + c_{j+B} + c_{j+2B}$$

and performing the argmax in Eq. (2) over c'_j instead of c_j. This is equivalent to correlating the data stream with a pattern consisting of four ASMs, spaced apart by runs of $B - F$ zeros, as shown in Fig. 9. Taking the block of turbo-encoded data as y_{l+F} through y_{l+B-1}, there are two markers preceding the identified block and two following it. A peak is detected when the correlation pattern is shifted relative to the data stream as shown, so the block of symbols denoted by the brace is extracted for turbo decoding. Because only three of the ASMs in the correlation pattern coincide with valid data in this example

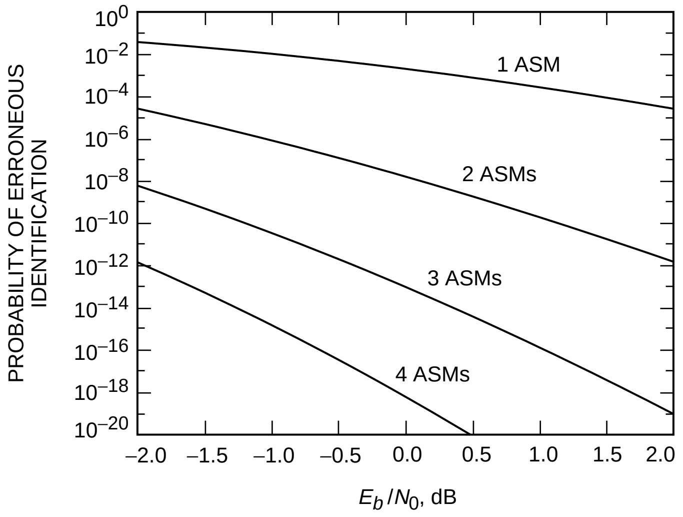

Fig. 8. Probability of erroneous synchronization for the (1784,1/3) turbo code.

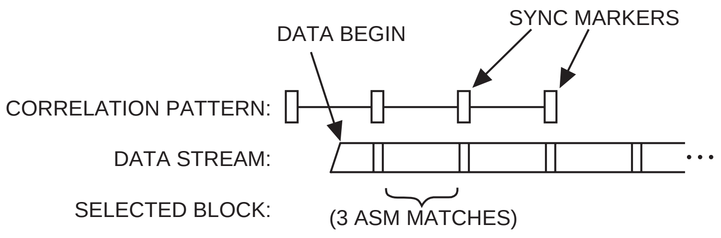

Fig. 9. Correlation performed by the frame synchronizer.

(and the fourth is presumably matched against random noise samples), the probability of erroneously identifying the location of this block of turbo-coded data is given approximately by the "3 ASMs" curve in Fig. 8.

As noted, two ASMs precede the identified block of symbols and two follow; this symmetry is desirable for synchronization recovery when symbols are inserted or omitted from the data stream. Figure 10 illustrates the behavior of the frame synchronizer at the beginning of the received data stream and when symbols are inserted. From the moment the turbo decoder is started, time is divided into search intervals as shown; one block of symbols beginning in each interval is always selected for a decoding attempt. Prior to interval 1, the synchronization decisions are not based on valid data, so the blocks of symbols selected for decoding are located randomly. In interval 1, the correlation pattern "reaches forward" far enough to find the first valid ASM in the data stream. With the probability of error determined by one ASM match, the indicated set of symbols will be extracted for decoding, though it contains no valid data. In interval 2, two ASMs are matched, the selected block of symbols is partially valid, and successful decoding might be possible. In intervals 3, 4, and 5, valid data are selected for decoding with the confidence provided by matching three, and then all four, ASMs.

The figure shows one or more inserted symbols in search interval 7, caused perhaps by a timing recovery error in the symbol tracking loop. This increases the probability of incorrect frame synchronization in search intervals 6 and 8 (for the identification is based on three ASMs rather than four),[5] and causes

[5] Performance is substantially worse than that shown by the "3 ASMs" curve in Fig. 8 because the misaligned symbols effectively increase σ.

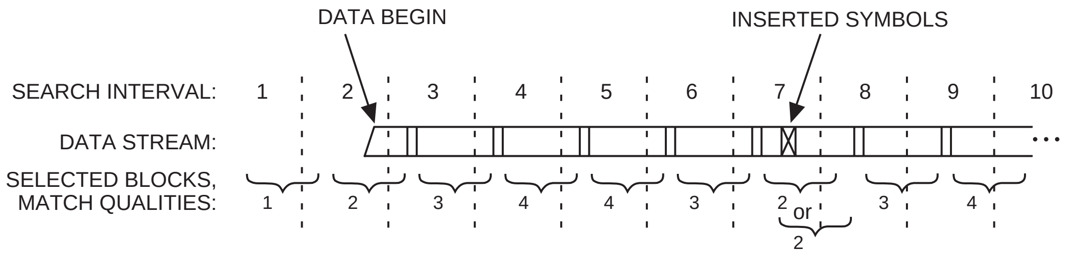

Fig. 10. The frame synchronizer's response to a data stream.

ambiguity in interval 7. Two ASMs can be matched by the correlator by selecting either a left-justified or a right-justified block; whichever yields the higher correlation will be selected.

In the remainder of this section, we delve into the assembly language of the TMS320C6200 DSP as we consider efficient ways of implementing the correlation computation. While these details tell little about the decoder's operation, they illustrate the types of optimizations that are available to the assembly language programmer and that have been made throughout this turbo-decoder implementation.

Returning to the single marker problem, the computation of each c_j, using Eq. (2) as written, requires F multiplications and $F-1$ additions. Using the fact that $a_i \in \{+1,-1\}$, the following manipulations permit reducing the total computation to about 3/8 F additions. Let

$$M = \{0 \le i < F : a_i = -1\}$$

be the set of indices where the sync pattern has value -1. Because its value is $+1$ at the other locations,

$$c_j = \sum_{i=0}^{F-1} y_{j+i} - 2\sum_{i\in M} y_{j+i}$$

Naming the first summation t_j and the second s_j, we have,

$$c_j = t_j - 2s_j$$

In this form, t_j can be computed inductively by

$$t_{j+1} = t_j + y_{j+F} - y_j$$

If $\mathcal{A}$ appears "random" (i.e., Bernoulli-1/2), the computation of s_j requires $|M| = F/2$ additions, and computation of t_j requires two more, assuming t_{j-1} is known.

The amount of arithmetic required can be reduced further by computing the c_j's in pairs. To do so, we define three new sets, each with cardinality about $F/4$:

$$M_0 = \{i : a_i = -1, a_{i+1} = -1\}$$

$$M_1 = \{i : a_i = -1, a_{i+1} = +1\}$$

$$M_2 = \{i : a_i = +1, a_{i+1} = -1\}$$

Letting

$$s_{k,j} = \sum_{i \in M_k} y_{j+i}, \quad k = 0, 1, 2 \tag{3}$$

we have,

$$c_j = t_j - 2(s_{0,j} + s_{1,j})$$

and

$$c_{j+1} = t_{j+1} - 2(s_{0,j} + s_{2,j})$$

Assuming t_{j-1} is known, these equations compute two correlations in $3/4\ F + 3$ additions and two multiplications by two, or about $3/8\ F$ additions apiece.

This procedure can be extended to simultaneously compute more than two c_j's. However, the benefit becomes smaller, the number of intermediate values that must be computed grows rapidly, and the number of registers available in the 'C6201 becomes limiting.

To extend the above computation to detect four synchronization markers, little additional computation is necessary. With two more additions, the correlations c'_j can be computed inductively as

$$c'_j = c'_{j-B} - c_{j-2B} + c_{j+2B}$$

The key step in the correlation computation, Eq. (3), serves as an attractive example of the 'C6201 assembly language. A single step in the computation begins with fetching a pair, y_{j+i} and y_{j+i+1}, of 16-bit values using one 32-bit load instruction (which also post-increments the address pointer to the next values to be fetched):

```
        ldw     *ad++[2],reg
```

The pair ends up "stacked" in the destination register `reg` with y_{j+i} in the 16 less significant bits and y_{j+i+1} in the 16 more significant bits, and so they must be separated. This can be done with two tricks. First, by taking all final results modulo 2^{16}, we can use the stacked pair in place of y_{j+i}, and y_{j+i+1} in the upper half of the register will be ignored and eventually discarded. Second, the 'C6201 has an instruction that multiplies the high halves of two registers together (two 16-bit quantities) and puts the resulting 32-bit quantity in a destination register. By high-half-multiplying the stacked y values with a register (named `one`) containing 1 in its high half, the result will be y_{j+i+1}, and it will be placed alone in the destination register (named `high`):

```
        ldw     *ad++[2],low
        mpyh    low,one,high
```

The resulting values are then added to the appropriate running sums $s_{k,j}$. Perhaps `low` should be added to $s_{1,j}$ and `high` to $s_{2,j}$. The two instructions are

```
        add     s1,low,s1
||      add     s2,high,s2
```

and they can be executed simultaneously, as denoted by the `||` marks in the margin. This sequence of four instructions forms the basic building block of the correlation computation.

The result of each instruction is not available until one or more clock cycles after it is executed, so (with the exception of the two `add` instructions) this particular set cannot be executed simultaneously. We can solve this by "pipelining" the operation. By starting one such computation in every clock cycle, the DSP can simultaneously execute one representative of each type of instruction, although they will belong to several different computations taking place concurrently. Taking the delays into account, the load instruction that fetches y_{j+i} and y_{j+i+1} takes place simultaneously with the multiply instruction for y_{j+i-19}, the add-low instruction for y_{j+i-18}, and the add-high instruction for y_{j+i-27}.

This collection of four instructions has been selected to use exactly half the resources available in the 'C6201 in a clock cycle. This means that with a second set of registers allocated to the task (suffixed by `a` and `b`), two sets of these instructions can run simultaneously. When the assembly language is dressed up with comments and with the names of the eight functional units spelled out explicitly (`.d1`, `.d2`, etc.), the code for one clock cycle is as follows:[6]

```
;Clock $i$
        ldw.d1     *ada++[2],lowa      ;fetch $y_{j+i}$ and $y_{j+i+1}$
||      ldw.d2     *adb++[2],lowb      ;fetch $y_{j+i+2}$ and $y_{j+i+3}$
||      mpyh.m1    lowa,onea,higha     ;separate $y_{j+i-19}$ from $y_{j+i-18}$
||      mpyh.m2    lowb,oneb,highb     ;separate $y_{j+i-17}$ from $y_{j+i-16}$
||      add.s1     s1a,lowa,s1a        ;add $y_{j+i-18}$ to $s_{1,j}$
||      add.s2     s0b,lowb,s0b        ;add $y_{j+i-16}$ to $s_{0,j}$
||      add.12x    s2b,higha,s2b       ;add $y_{j+i-27}$ to $s_{2,j}$
||      add.11x    s0a,highb,s0a       ;add $y_{j+i-25}$ to $s_{0,j}$
```

Occasionally y_i is not in one of the sets M_0, M_1, M_2, and some of these instructions aren't needed. It can be shown that if $\mathcal{A}$ has perfect Bernoulli-1/2 statistics, this reduces the number of clock cycles required from 1/4 F to 7/32 F and leaves a few unused instruction "holes" in those that remain. To compute correlations over four frame-sync markers, two additional clock cycles are required for the memory accesses to load and store the various c_j and c'_j values. The remaining computations can be fitted into the holes and require no additional clock cycles. Thus, the $F = 96$ bit synchronization marker requires $7/32 \times 96 + 2 = 23$ clocks per pair of correlations, and $F = 192$ requires 45, given a one cycle penalty due to slightly unfavorable statistics of the longer marker.

V. The Control DSP

The control DSP is responsible for coordination between the data source, the frame-synchronizer DSP, the turbo-decoder DSPs, and the data sink, and for managing data buffers between each of these. The input symbols are provided by the Block V receiver during normal operation or by various hardware sources during simulation and testing, but in every case they are collected and stored in memory by a direct memory access (DMA) channel within the control DSP. Similarly, the output symbols are delivered to the telemetry processor (TLP) during normal operation or to other destinations for testing. Whatever the specific situation, there is software somewhere that is responsible for delivery of each decoded block of data when it becomes available.

[6] By using the addressing mode `*ada++[2]`, the pointer `ada` is automatically incremented to fetch every other pair of y values, while `adb` fetches the alternate ones. This alternation avoids memory bank hits, and when combined with the 'C6201's circular addressing mode, the pointers also automatically wrap around when reaching the end of their circular buffer.

The data handling is considered as a collection of four processes and four buffers, as sketched in Fig. 11. Each of the processes (data receipt, frame synchronization, turbo decoding, and data delivery) is asynchronous with the others and runs whenever its input buffer(s) have data and there is space for the result in its output buffer(s); otherwise, the process waits:

```
When a process finishes:
   Mark the process as idle
   Increment the fullness of its input buffers (if any)
   Decrement the fullness of its output buffers (if any)
Start a process, and mark it as busy, whenever:
   the process is idle,
   and its input buffers (if any) aren't empty,
   and its output buffers (if any) aren't too full.
```

The frame-synchronization and turbo-decoding processes run on different DSPs, data receipt is performed by the control DSP's DMA unit, and some of the data delivery is handled by external hardware. The control DSP starts each process by setting a semaphore or sending an interrupt, and each process interrupts the control DSP when it finishes.

In implementation, each of the processes and buffers has complications, some of which warrant discussion. The frame synchronizer does not consume the stream of received symbols, but merely breaks them into frames. Thus, its input buffer consists only of a pointer to the received symbol stream and a count of the number of symbols waiting for synchronization. The frame synchronizer's output buffer is implemented as a circular queue of pointers, each of which points to a frame of turbo-encoded symbols. As input, the turbo-decoder process uses both these pointers in the `Coded Frames` buffer and the symbols they refer to in the `Received Symbols` buffer.

Because turbo decoding is performed by several DSPs in parallel, the implementation of their shared output buffer is interesting. It consists of a set of memory blocks, each of which can hold one frame of decoded data, and a queue of pointers to those blocks. When a decoder is assigned a block of data to decode, it takes a pointer from the head of the queue. It decodes the block, stores the result in the memory indicated by the pointer, and then returns the pointer to the tail of the queue. The data-delivery process monitors the tail of the queue, and when each pointer is returned, it delivers the corresponding data to the output of the system. The turbo decoders may require varying amounts of time to complete their decoding tasks, so decoders running simultaneously may finish in a different order from that in which they were started. When this happens, the pointers will be returned in a different sequence, and the decoded blocks will be saved out of order. The prototype decoder does not attempt to correct this; instead, a sequence number is included in the auxiliary header information delivered with each block, and the reordering must be handled downstream.

In tests, the maximum speed of the system is determined by the turbo-decoding process for all but the longest turbo-code frame lengths. For the length-8920 codes, the frame synchronizer's speed becomes

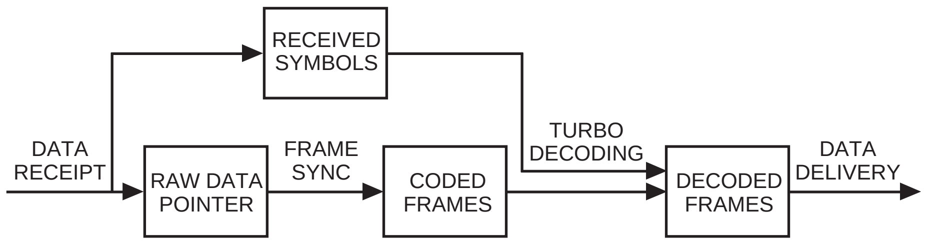

Fig. 11. Decoder structure as a collection of buffers connected by processes.

limiting. If this proves to be a problem, a less computationally complex synchronization algorithm could readily be implemented with a small penalty in error rate.

VI. Conclusion

Turbo codes provide outstanding performance with low encoder complexity and moderate decoder complexity. This makes them attractive replacements for the far more complex codes in use today, and CCSDS standardization is facilitating the use of turbo codes. Several projects, including the Spacecraft Transponder Modem (STM), the System Interface Assembly (SIA), and the turbo-decoder development described here, will introduce turbo codes to deep-space communications by 2003.

Current microprocessor clock speeds are high enough that the decoder can be implemented in software. The current implementation can perform turbo decoding at 300 kbits/second using 10 iterations per frame. With a stopping rule, the speed can be doubled or tripled with some penalty in E_b/N_0. While the software development is painstaking by most programming standards, it is an attractive alternative to custom hardware development. The resulting decoder can be modified as needed to handle variants of the standard codes, to offer additional capabilities when necessary, and to incorporate the newest research results. It is also portable to new hardware as faster DSPs and more capable support circuitry become available.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes," *IEEE International Conference on Communications*, Geneva, Switzerland, pp. 1064–1070, May 1993.

[2] D. Divsalar and F. Pollara, "Turbo Codes for Deep-Space Communications," *The Telecommunications and Data Acquisition Progress Report 42-120, October–December 1994*, Jet Propulsion Laboratory, Pasadena, California, pp. 29–39, February 15, 1995.
http://tmo.jpl.nasa.gov/tmo/progress_report/42-120/120D.pdf

[3] J. B. Berner, S. Kayalar, and J. D. Perret, "The NASA Spacecraft Transponding Modem," *2000 IEEE Aerospace Conference Proceedings*, Big Sky, Montana, vol. 7, pp. 195–209, March 18–25, 2000.

[4] Consultative Committee for Deep Space Data Systems, *Telemetry Channel Coding (101.0-B-4 Blue Book)*, CCSDS: Newport Beach, California, May 1999.

[5] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate," *IEEE Transactions on Information Theory*, vol. IT-20, no. 2, pp. 284–287, March 1974.

[6] A. Matache, S. Dolinar, and F. Pollara, "Stopping Rules for Turbo Decoders," *The Telecommunications and Mission Operations Progress Report 42-142, April–June 2000*, Jet Propulsion Laboratory, Pasadena, California, pp. 1–22, August 15, 2000.
http://tmo.jpl.nasa.gov/tmo/progress_report/42-142/142J.pdf

[7] C. Heegard and S. Wicker, *Turbo Coding*, Boston, Massachusetts: Kluwer, 1998.

[8] J. Eyre, "The Digital Signal Processor Derby," *IEEE Spectrum*, vol. 38, no. 6, pp. 62–68, June 2001.

Appendix
The Turbo Decoding Algorithm

A concise description of the standard turbo-decoding algorithm is surprisingly difficult to find in the literature, so it is presented here using notation similar to [7]. A brief summary of that notation follows; see [7] for a more detailed explanation. A binary convolutional encoder has an internal state s_{j-1} immediately prior to time j when it is given input bit m_j to encode. The state and input bit specify the branch of the trellis, which we denote $b_j = b(s_{j-1}, m_j)$. The inverse functions are denoted $s_{j-1} = \sigma^-(b_j)$ and $m_j = l_{\text{in}}(b_j)$. Each branch has an output label of several symbols (indexed by k), $c_j^k = l_{\text{out}}^k(b_j)$ and a following state, $s_j = \sigma^+(b_j)$. Because the first constituent convolutional encoder is systematic, there is some k for which $c_j^k = m_j$. We put tildes over the quantities and functions for the second constituent convolutional encoder, and leave those for the first convolutional encoder unadorned.

I. Encoding Equations

A turbo encoder initializes $s_0 = \tilde{s}_0 = 0$, is given message m_j, $1 \leq j \leq N$, and performs the following operations:

$$\tilde{m}_{\pi(j)} = m_j$$

$$\left.\begin{array}{ll} b_j = b(s_{j-1}, m_j) & \tilde{b}_j = \tilde{b}(\tilde{s}_{j-1}, \tilde{m}_j) \\ s_j = \sigma^+(b_j) = \sigma^+\big(b(s_{j-1}, m_j)\big) & \tilde{s}_j = \tilde{\sigma}^+(\tilde{b}_j) = \tilde{\sigma}^+\big(\tilde{b}(\tilde{s}_{j-1}, \tilde{m}_j)\big) \\ c_j^k = l_{\text{out}}^k(b_j) = l_{\text{out}}^k\big(b(s_{j-1}, m_j)\big) & \tilde{c}_j^k = \tilde{l}_{\text{out}}^k(\tilde{b}_j) = \tilde{l}_{\text{out}}^k\big(\tilde{b}(\tilde{s}_{j-1}, \tilde{m}_j)\big) \\ \multicolumn{2}{c}{\text{for } 1 \leq j \leq N} \end{array}\right\} \tag{A-1}$$

The functions $\sigma^+\big(b(s,m)\big)$, $\tilde{\sigma}^+\big(\tilde{b}(\tilde{s},\tilde{m})\big)$, $l_{\text{out}}^k\big(b(s,m)\big)$, and $\tilde{l}_{\text{out}}^k\big(\tilde{b}(\tilde{s},\tilde{m})\big)$ can be determined from Fig. 1, and the function $\pi(j)$ is specified in the CCSDS standard [4]; all can be precomputed and tabulated for an efficient encoder.

CCSDS trellis termination is achieved by using Eq. (A-1) for $N+1 \leq j \leq N+4$, with m_{N+1} through m_{N+4} chosen as the unique solutions such that $s_{N+4} = 0$, and similarly for $\tilde{m}_{N+1}$ through $\tilde{m}_{N+4}$. It turns out that this unique solution is described by a function $m_j = m_{\text{term}}(s_{j-1})$, and this function can also be tabulated in advance.

II. Channel Equations

For BPSK transmission over an AWGN channel, each binary output c from the encoder is mapped to a real value x, and Gaussian random noise is added to each by the channel to form the received symbol y:

$$x_j^k = \begin{cases} +1 & \text{if } c_j^k = 0 \\ -1 & \text{if } c_j^k = 1 \end{cases}$$

$$y_j^k = x_j^k + n_j^k$$

where $n_j^k \sim \mathcal{N}(0, \sigma^2)$, and

$$\sigma = \frac{1}{\sqrt{2RE_b/N_0}}$$

III. Decoding Equations

The turbo-decoding equations are given here in logarithmic form, because this appears to be less sensitive to the rounding errors introduced by fixed-point arithmetic. We define the $\widehat{+}$ operation as

$$a \widehat{+} b = \log(e^a + e^b) = \max(a, b) + \log\left(1 + e^{-|a-b|}\right)$$

Note that this operation is associative, so

$$\widehat{\sum_{i=1}^{n}} a_i = a_1 \widehat{+} \cdots \widehat{+} a_n$$

is unambiguous.

When using fixed-point arithmetic, necessary for efficient implementation on fixed-point DSPs, one must choose the amount of precision to maintain by multiplying all values by a constant and rounding to integers. If all values were multiplied by 8, for example, this would preserve 3 bits of fraction following the binary point. For historical reasons, a scale factor of 10 is used, preserving 1 decimal digit, or 3.32 bits, of fractional information. The received noisy symbol data must be correspondingly scaled and quantized, and after that, the quantization and scale factor appear in the equations in only two places: the $\widehat{+}$ operation and the clipping level κ (and also in a stopping rule threshold θ if used). Suppose $c = a \widehat{+} b$; let $S = 10$ be the scale factor, and $A = Sa$, $B = Sb$, and $C = Sc$ be the scaled versions of a, b, and c. Then,

$$\begin{aligned} C &= S(a \widehat{+} b) \\ &= \max(A, B) + S \log\left(1 + e^{-|A-B|/S}\right) \end{aligned}$$

In implementation, the function $S \log(1 + e^{-|A-B|/S})$ is rounded to the nearest integer and tabulated for integer values of $|A - B|$.

The decoding algorithm is as follows.

(1) Scale the received symbols according to the SNR:

$$r_j^k = \begin{cases} y_j^k/\sigma^2 & \text{if } c_j^k \text{ is transmitted} \\ 0 & \text{if } c_j^k \text{ is punctured} \end{cases}$$

(2) Initialize the extrinsic log likelihood ratios:

$$\mu_j^0 = 0 \text{ for each } 1 \leq j \leq N$$

(3) Iterate for $i = 1, 2, \ldots$, until a stopping condition is satisfied or for a fixed number of iterations I:

(a) Tabulate metrics for each trellis label:

$$m_j(l_{\text{in}}, l_{\text{out}}) = l_{\text{in}}\mu_j^{i-1} + \sum_k l_{\text{out}}^k r_j^k$$

for all combinations of $l_{\text{in}}, l_{\text{out}}^k \in \{-1, +1\}$ and each $1 \leq j \leq N$.

(b) Initialize α_0's and β_N's according to known trellis termination conditions. For example, if the initial trellis state is known to be 0 and the final state is unknown, use

$$\alpha_0(s) = \begin{cases} 0 & \text{if } s = 0 \\ -\infty & \text{otherwise} \end{cases}$$

and

$$\beta_N(s) = 0$$

for each state $s \in \{0, \ldots, 2^\nu - 1\}$

(c) Compute α's and β's for each trellis state. Normalize as needed to prevent numeric overflow:

$$\alpha_j(s) = \widehat{\sum_{+}}_{\substack{b \text{ such that} \\ \sigma^-(b)=s}} \alpha_{j-1}\big(\sigma^-(b)\big) + m_j\big(l_{\text{in}}(b), l_{\text{out}}(b)\big)$$

$$\beta_j(s) = \widehat{\sum_{+}}_{\substack{b \text{ such that} \\ \sigma^+(b)=s}} \beta_{j+1}\big(\sigma^+(b)\big) + m_{j+1}\big(l_{\text{in}}(b), l_{\text{out}}(b)\big)$$

for each state $s \in \{0, \ldots, 2^\nu - 1\}, 1 \leq j \leq N - 1$.

(d) Compute new extrinsic information, and clip to $\pm\kappa$:

$$\lambda_j^{i-0.5}(a) = \widehat{\sum_{+}}_{\substack{b \text{ such that} \\ l_{\text{in}}(b)=a}} \alpha_{j-1}\big(\sigma^-(b)\big) + \beta_j\big(\sigma^+(b)\big) + m_j\big(l_{\text{in}}(b), l_{\text{out}}(b)\big)$$

for each $a \in \{-1, +1\}$ and $1 \le j \le N$, and

$$\mu_j^{i-0.5} = \text{ median}\left(-\kappa, \kappa, \frac{1}{2}\left(\lambda_j^{i-0.5}(+1) - \lambda_j^{i-0.5}(-1)\right) - \mu_j^{i-1}\right)$$

for each $1 \le j \le N$.

(e) Interleave the extrinsic information:

$$\tilde{\mu}_{\pi(j)}^{i-0.5} = \mu_j^{i-0.5}$$

for each $1 \le j \le N$.

(f) Repeat the BCJR algorithm in Steps (a) through (d) using $\tilde{\mu}_j^{i-0.5}$, $\tilde{r}_j^k$, and the tilde versions of the various functions, resulting in $\tilde{\lambda}_j^i(a)$ and $\tilde{\mu}_j^i$.

(g) Deinterleave the extrinsic information:

$$\mu_j^i = \tilde{\mu}_{\pi(j)}^i$$

for each $1 \le j \le N$.

(4) Deinterleave the final reliabilities and make hard decisions after the last iteration:

$$\hat{m}_j = \begin{cases} 0 & \text{if } \tilde{\lambda}_{\pi(j)}^I(+1) > \tilde{\lambda}_{\pi(j)}^I(-1) \\ 1 & \text{otherwise} \end{cases}$$

for each $1 \le j \le N$.