Dietmar Schütz
Siemens AG, CT SE 2

Otto-Hahn-Ring 6
81739 München
Germany

eMail: dietmar.schuetz@siemens.com
Phone: +49 (89) 636-57380
Fax: +49 (89) 636-45450

BITSTREAM X-CODER

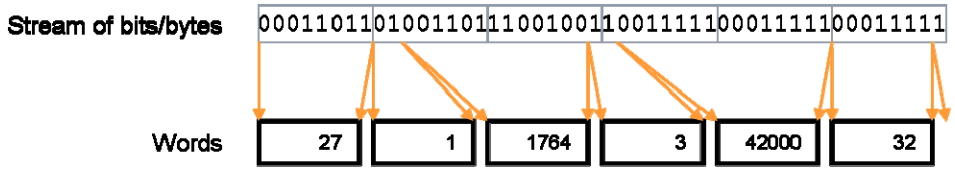
Version 1.0, EuroPLoP2008 Final

Summary This pattern provides an efficient solution for decoding densely packed bit-streams into properly aligned data structures. This conversion problem is typical for communication scenarios, where limited bandwidth and processing speed require strong optimization, and hence motivate different structures for both tasks. The proposed solution operates on bigger chunks instead of single bits, and prevalent shift operations are replaced by a finite state machine and lookup tables, thus yielding formidable throughput.

Context Software applications dealing with dense coding of information, for example communication protocols of embedded systems.

Example Consider an application scenario where wireless communication and transponders are used to exchange information with objects passing at significant speed. The contact duration is very short, and due to transmission reliability, the bandwidth is rather low too.
In order to pass as much information as possible, the telegrams transmitted are packed densely, not wasting any bits.
For efficient processing in the target environment, the telegrams need to be "unpacked" into word-aligned data structures.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.



Inflating a dense stream of bits into a word aligned data structure

The figure below is a concrete example taken from the European Train Control System (ETCS) "language" [UNISIG]. The left part shows the structure of a "static speed profile" telegram, containing two nested iterative subsections, each controlled by a counter value N_ITER. The right part shows the corresponding word-aligned data structure.

Variable	*Length [bit]*
NID_PACKET	8
Q_DIR	2
L_PACKET	13
Q_SCALE	2
D_STATIC(k)	15
V_STATIC(k)	7
Q_FRONT(k)	1
N_ITER(k)	5
NC_DIFF(k,m)	4
V_DIFF(k,m)	7
N_ITER	5

Telegram structure definition (with two nested iterative subsections)

```
struct sPACKET_27 {
  int packetId;           //NID_PACKET
  int direction;          //Q_DIR
  int packetLength;       //L_PACKET
  int scaleFactor;        //Q_SCALE;

  int nSpeedData;         //N_ITER +1
  struct sSpeedData {
    int staticDistance;   //D_STATIC
    int staticSpeed;      //V_STATIC
    int frontIndicator;   //Q_FRONT

    int nCategory;        //N_ITER(k)
    struct sCategory {
      int categoryId;     //NC_DIFF
      int cateorySpeed;   //V_DIFF
    } aCategory[33];
  } aSpeedData[33];
};
```

Corresponding target data structure definition (in C)

Another (simpler) example is base64 [Base64], a popular binary-to-text-encoding scheme used to transmit binary data across media that only support ASCII-printable characters (who's most significant bit equals zero). In this scheme, three in-bytes are inflated to four out-bytes, where every out-byte holds six of the 24 in-bits.

Problem

Processing units usually operate on aligned data (such as bytes and words) much faster compared to addressing and working on single bits, thereby dealing space for time. On the other hand, from the communication perspective, time is directly related to space. Hence, data structures are packed densely for faster transmission, and unpacked again for quicker processing. This artifice relieves the operational code from handling single bits, but nevertheless the non-productive packing routines burn CPU

resources by shifting bits back and forth. How can the packing/unpacking be done in an efficient way?

The following **forces** influence the solution:

- *Sequence of bytes*. Low level drivers usually operate on a sequence (stream or array) of bytes when transferring binary data, e.g. via a serial interface (UART). Hence, this is an appropriate representation of the "raw" bit stream.
- *Performance and Granularity*. Operating on a fine granularity (such as bits) automatically raises the number of objects to handle, with negative consequences for the performance of the overall operation. In addition, extracting bits from words requires additional effort.
- *Telegram format*. A telegram is a (nested) sequence of single elements, sometimes containing control content (e.g. length, number of consecutive elements, etc.) that have to be taken into account during processing the telegram. The single elements often adhere to a "processable" format (e.g. two's-complement), but might offend them too, especially in heterogeneous environments.
- *Telegram types*. Typically, there are various types of telegrams. Each of them comes with its own syntax, and needs to be handled accordingly.
- *Ease of code*. The telegram structure and corresponding data structures should be clearly visible in the code, not obscured by incomprehensive activities.

Solution

Although your granularity is bits, operate on words: read and convert information in chunks as big as possible, at least bytes.
To this end, transform the bit reading functionality into a finite state machine (FSM), where each state represents distinct values for the bits not processed yet in the "head" byte of the input stream. A transition from on "state" to another one processes as many bits as possible at once, and accumulate these chunks into the aspired result. Embed this state and its handling into a function that is responsible to read a single element of the telegram (unpack a short sequence of bits from the input into a word). Use this function to assemble the structure of the whole telegram.

Structure

The *Input Stream* is a densely packed sequence of bits, typically stored as a sequence of bytes or words. The first bit of the input stream is not necessarily the most significant bit (MSB) of the first word or byte. Instead, there might by some leading junk bits that must be ignored[1].

[1] This situation can arise for example when receiving data from a transponder. Typically, such messages are repeated in a cyclic manner, and the continuous data stream must be inspected first to detect a position that fits the start condition of the telegram.

The *target structure* is a memory location where the decoded telegram should be stored into. Typically for embedded environments, this is a “flat”, memory section with fixed size, and can be addressed via a structure definition. In other, less restrictive environments, tree structures build from several nodes and leafs might be an option too. All elements in the target structure are available in an optimal format: they can be used “as is” for further processing.

The core element of this pattern is the *head buffer*, a small buffer that is responsible to store the next bits of the stream for further processing. It contains an aligned snippet from the stream, as well as the information how many bits of that are remaining for processing. Both pieces are coded together in a *State* variable.

A *finite state machine* (FSM) specifies the behaviour when decoding single elements out of the stream. Input values are the actual state, and the number of bits missing to complete the current telegram element. A transition “consumes” as much bits as possible from the state, yielding the new state, the remaining length, and the decoded data, properly aligned. The output values are taken from *transition tables*, that look-up the result using indexed access based on the inputs.

Since decoding an element might require more that one transition of the FSM, an *accumulator* is necessary to store and combine the partial results until the whole element is completed.

The structural parts listed above are managed and orchestrated together into a readBits *function* that is responsible to decode an Integer (a word) by consuming a given number of bits from the input stream.

Last but not least, a structure parser is responsible to process the sequence of single elements in the input stream and extract the corresponding telegram structure in the aspired output format. This task incorporates parsing optional, iterative, and nested structures based on control data embedded in the stream. In addition, some adaptations on the data might become necessary, such as scaling of values or conversions to the “local” representation (e.g. big-endian number formats).

Dynamics

There are two important dynamic scenarios that characterize this pattern, *initialization* and *decoder loop*.

For i*nitialization*, the state as well as the input stream must be set into a defined condition. Usually, the input stream can be used as provided by the corresponding environment (either as a quasi-infinite source of bytes/words that can be read consecutively, or as a continuous memory region with fixed length). More care might be necessary on initializing the state. If the first bit of the input stream is not the MSB of the first word, the valid bits must be mapped correctly into the state variable.

The *decoder loop* covers the scenario if a single element spreads across two or more consecutive bytes/words in the input stream. Since the head buffer

can only contain data from one byte/word, it will be emptied on the first pass, and hence must be reloaded until the element is complete.

Implementation

The implementation of this pattern incorporates at least the five steps described below.

1 *Decide on the chunk size.* The chunk size should correspond to the size of an operatable unit (byte, word) of the CPU. Bigger chunk sizes yield better performance (linear ratio), but dramatically rise the size of the lookup tables. For most environments, one byte is an appropriate choice. Two bytes are an option if space is nothing to worry about. Four bytes are too much to be handled with lookup tables, thus only applicable for the calculation approach (see step 3 for details).

☛ In the example implementation, the chunk size is a byte. □

2 *HeadBuffer, coding of state.* The state must reflect the remaining valid bits of the chunk, and is stored in the head buffer.

☛ Given that the first 3 bits have been processed previously, there are 5 remaining bits in the head buffer, hence 2^5 = 32 different states are necessary to reflect the possible variants. This sums up to 511 states for zero to eight remaining bits with distinct values. □

A simple approach uses a byte variable to store the bit values, and an integer variable indicating the number of valid bits (the rightmost bits in the byte). Using both values as lookup index, the table contains n* 2^n entries, with n as the number of bits in the “chunk” sizeof(chunk)*8. Unfortunately, this wastes a lot of space, since for small numbers of remaining bits the leading bits in data are irrelevant.

A more effective approach is to combine both variables into one by using a simple coding scheme:
Bits consumed already (in the example below indicated by a dash -) are set to “0”, a single “**1**” indicates the start of the “valid” part, and is followed by the values of the valid bits.

☛ State is stored in the head buffer as an integer, coded as concatenation of '0'* + '**1**' + ”remaining bits”. The state “values” range from 1 to 511. The following table shows examples for concrete state codings.

Scenario	Logical state	Coded state (content of head buffer)
five remaining bits, values 0 1 0 1 1	`---0 1011`	`0000 0000 001`**`1`**`0 1011` = 43_{10}
no remaining bits	`---- ----`	`0000 0000 0000 000`**`1`** = 1_{10}
eight remaining bits, all 1	`1111 1111`	`0000 000`**`1`**` 1111 1111` = 511_{10}

□

3 *Transition function and lookup tables*. Consuming a specific number of bits from the head buffer causes a transition from one state to another. The maximum number of bits that can be consumed at once is limited by the number of remaining bits in the head buffer. If the buffer runs empty, the missing bits will be read later after reloading the buffer.

For each pair of (state, bitsToConsume) the subsequent state can be easily computed. The same is true for the value of the extracted bits, and the still missing bits to be read later.

For optimal performance on most processors, these functions are not computed at runtime. Instead, the results are stored in appropriate lookup tables. These tables may consist of previously calculated constants, or can be initialised at runtime.

Note: The look-up approach might be suboptimal for modern processors with data caches. In such cases, there is a high probability for cache misses when accessing the lookup tables, causing a much slower access to (uncached) memory, which results in significant performance drops. In such environments, calculating the "lookup" value at runtime can be quicker than real lookup, providing more speed and less memory consumption.

4 *Implement the bit-reading function*. The following code describes the core of the read routine.

```
readBits( byte &buf;       // pointer to input stream
          unsigned n;      // number of bits to consume
          unsigned &result // resulting value
          ) {

  static unsigned z = 0;   // „empty" byte as start state
  unsigend accu = 0;       // accumulated result

  while( n > 0 ) {  // there is something to read
    if( z == 0 )
      z = *(buf++); // „refill" state from input stream

    z_save = z;
    accu |= value(z_save, n);  // look-up (part of) value
    z = z_new(z_save, n);      // look-up new state
    n -= used( z_save, n );    // look up number of missing bits
    }

  result = accu;
  }
```

This core routine might be extended with handling of error conditions (e.g. premature end of input stream). Another add-on might be an explicit initialization of the "state" to a value different than 0, in order to cope with bit streams that do not start at a byte boundary in the input buffer.

The following show the values for `z`, n, and accu for two examples. The first example consumes only a part (4 bits) of the input stream (buffered in `z`, containing 6 bits), processed in on step. The second example consumes 11 bits, and hence needs to iterate through the loop and to refill the buffer.

```
           z n      accu                  new z    n'
-------------------------------------------------------------------

Example 1:
    --011011 4   + 0000000000000110  ------11    0
                   0000000000000110
                 = 6₁₀

Example 2:
    --011011 11    0000001101100000  --------    5      // refill z
    10011110 5   + 0000000000010011  -----110    0
                   0000001101110011
                 = 883₁₀
```

5 *Implement the structure parser*. The structure parser might be implanted in two different ways: either hard-coded or by means on an Interpreter (see [GOF94] for details).

A hard-coded parser might be more efficient, but can (depending on the size and number of telegram structures) result in a huge amount of code.

The probably smarter way is an interpreter, with data structures that controls the behaviour: a sequence of read operations processes the input stream, and writes the resulting values through a "write pointer" (base plus offset) into the target structure, while a small stack automaton keeps track of nested telegram structures.

Example Resolved

The thousands of line of code of hard coded structure parsers are obsolete, replaced by a small read function and the structure interpreter, both fitting on a single page. They are complemented by declarative code that provides the parser "programs", reflecting the telegram structures, automatically generated from XML-based structure definitions. Execution times are significantly improved, and maintenance is eased up.

Consequences

BITSTREAM X-CODER provides the **benefits** depicted below:

- *Speed.* The method described above is ten to a hundred times faster compared to a bitwise reading and recombination of the encoded values.
- *Code reduction.* Especially in conjunction with an interpreting structure parser, the required active code is rather small. In addition, it does not depend on the number and complexity of different telegram structures.

On the other hand, the pattern carries the following **liabilities**:

- *Complexity*. Less obvious than the "straight forward" implantation, not easy to understand.
- *Memory Consumption.* The transition tables are getting quite big, especially when choosing bigger chunks for processing (exponential growth in size for linear increase in speed).

Variants

The pattern can be applied "the other way around" to implement the encoder. Accordingly, the tail buffer is extended until it is full, and then flushed to the output stream.

Some compression algorithms share many commonalities with encoding and decoding bitstreams. For example, zip uses Huffman codes (the length of the "code" relates to the relative frequency of the corresponding character.

Credits

Thanks to my shepherd Peter Sommerlad, for his patience and supportive suggestions. I also thank the participants of the writers workshop at EuroPLoP2008 (André L. Santos, Carsten Hentrich, Diethelm Bienhaus, Jürgen Salecker, Paul G. Austrem) for their valuable feedback. Thanks to my working student Omar Farooq, he implemented the core of this pattern in the context of the "ETCS language" defined in [UNISIG].

References

[GOF94]
E.Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software; Addison-Wesley 1994

[POSA96]
F. Buschmann, R. Meunier, H. Rohnert, M. Stal, P. Sommerlad: Pattern Oriented Software Architecture, A System of Patterns; Wiley 1997

[UNISIG]
UNISIG SUBSET-026
http://www.era.europa.eu/public/Documents/ERTMS%20Documentation/Mandatory%20Specifications/SRS%20230.zip

[Base64]
Wikipedia article on Base64
http://en.wikipedia.org/wiki/Base64