Informal Specification of `Bitwalker`

Jens Gerlach
Fraunhofer FOKUS, Berlin, Germany

Starting from the original high-level description of the `Bitwalker` software this document provides a more detailed, but still informal, specification. This specification attempts to be precise to the extend that

1. testers are able to write unit tests for `Bitwalker` without the need to look at its implementation
2. it can serve as a base for a *formal* specification in the ACSL specification language of Frama-C.

Contents

List of Corrections

1 Introduction

The original, high-level description of the `Bitwalker` software is given here:

> Task of the Bitwalker
>
> The Bitwalker shall sequentially read a bit field and convert it into a natural number. Conversely, the Bitwalker shall read a natural number and write it to a bit field.
>
> Description of the task
>
> A bit oriented field in memory is given.
>
> With a byte oriented representation in memory, the most significant bit would have the index 0 in Big-endian notation. The bit index increases monotonously by 1. Natural numbers that are given in Big-endian notation shall be read or written. The range of natural numbers is limited by the width of the bit field.
>
> Two access methods shall be provided:
>
> - Random read/write access to the bit field.
> - Sequential read/write access to the bit field starting at a given index.
>
> Each read/write operation shall "consume" the bits that have been accessed. The access functions shall be written in plain C and shall be reentrant.
>
> Context of the Bitwalker:
>
> ETCS uses telegrams in Big-endian notation to communicate between vehicle and track. In the scope of OpenETCS a generator shall be developed that automatically converts the telegram specification into an encoder/decoder. The Bitwalker will be used here as an auxiliary function.

While this description provides basic information about the context of `Bitwalker` and about the intentions of the designer there are many open questions:

- What are the exact names and signatures of the functions?
- Are there any error conditions for these functions?
- What does "consumption of bits" mean?
- What does *reentrant* exactly mean?

The `Bitwalker` software can be considered to consist of a *public* part (sequential access functions) and a *private* part (random access functions).

- The public part consists of the C data type `T_Bitwalkear_Incremental_Locals` and several C functions that can be used to manipulate objects this type.

- The private part consists of the functions `Bitwalker_Peek` and `Bitwalker_Poke` that implement the core functionality of `Bitwalker`.

Before we informally specify `Bitwalker` we introduce some auxiliary concepts and formulate general assumptions. We would also like to point out the following: When we speak in this document of *integers*, then we refer to the infinite set of mathematical integers $\{\ldots,-1,0,1,\ldots\}$ and not to one of the many finite representation provided by the type system of C.

This distinction is important because it usually makes more sense to describe the functionality of a piece of software in a more abstract way. In a later step the realities of specific the C type system have to be taken into account.

2 Basic concepts

- A *bit stream* is an array containing elements of type `uint8_t`.

  A bit stream of length n contains $8n$ bits.

- A bit stream is *valid* if the array is valid.

- A bit stream can be indexed both by its array indices and its *bit indices*.

  Figure 2.1 shows the difference between array indices and bit indices in a bit stream. The two bit indices, 0 and 14, mark bit positions in the first and second array element, respectively.

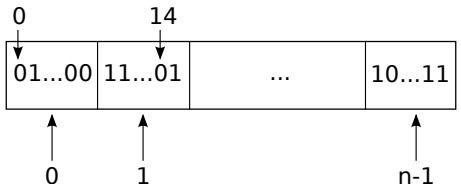

Figure 2.1: Array indices and bit indices in a bit stream

- The C programming language neither provides a type *bit* nor does it support random access to the bits of a bit stream. In order to access the i-th bit of a bit sequence one typically has to first access the byte with index $j = i/8$ and then access the bit $k = i \pmod 8$ within this byte. Note that in Figure 2.1 bytes and bits are indexed in increasing order from the *left*. On the byte level, however, bits are often indexed from the *right*. For example, to access the k-th bit of a byte `a` one can shift this bit to the right by $7 - k$ and extracts then the now rightmost bit by performing a bit-wise *and* with the value 1

```
(a >> (7-k)) & 1
```

- A *bit sequence* is a consecutive sequence of bits within a bit stream as represented in Figure 2.2.

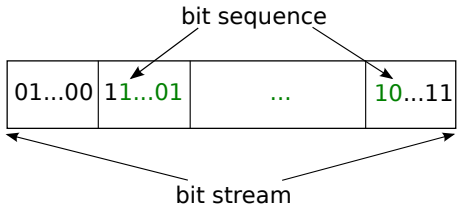

Figure 2.2: A bit sequence within a bit stream

A bit sequence is given by the position of its first bit (a bit index in the bit stream) and its *length*, that is, the number of bits it contains.

- A bit sequence that starts at bit index p and that consists of length $l \geq 0$ is refered to *valid* (with respect to a bit stream of length n) if the following conditions are satisfied

$$0 \leq p < 8n$$
$$0 \leq p + l \leq 8n$$

  Not that only the bits with indices $p \leq i < p + l$ are to be accessed but not the bit with index $p + l$.

We assume that the C-types `unsigned int` and `int`, which are used in the implementation to represent indices, counting and error codes, have a width of 32 bits. We point this out here because we conducted the verification on a platform with these characteristics.

As an aside, MISRA-C discourages the use of "generic" integer types such as `int` and `unsigned int` and recommends the use of integer types whose names contain the exact width.

3 The `Bitwalker` functions in more detail

3.1 The public interface of `Bitwalker`

This section describes the public interface of `Bitwalker`.

3.1.1 The structure `T_Bitwalkear_Incremental_Locals`

The type `T_Bitwalkear_Incremental_Locals` is defined as follows

```
struct  T_Bitwalker_Incremental_Locals
{
    uint8_t        *Bitstream;
    unsigned int   Length;
    unsigned int   CurrentBitposition;
};
```

Description

- `Bitstream` is the start address of a valid bit stream.
- `Length` is the length of the bit stream, that starts at `Bitstream`.
- `CurrentBitposition` is a valid bit index in the bit stream given by `Bitstream` and `Length`

Remark The field `Length` will be passed to `Bitwalker_Peek` and `Bitwalker_Poke` as `BitstreamSize`. This might be confusing because those functions also have an argument named `Length`.

3.1.2 `Bitwalker_IncrementalWalker_Init`

In this section we specify the function `Bitwalker_IcrementalWalker_Init`. The function initializes object of the type `T_Bitwalkear_Incremental_Locals`. The function signature reads:

```
void Bitwalker_IncrementalWalker_Init(
        T_Bitwalker_Incremental_Locals *Locals,
        uint8_t Bitstream[],
        unsigned int Size,
        unsigned int FirstBitposition);
```

Preconditions

- `Locals` is a dereferenceable pointer.
- `Bitstream` is the start address of a valid bit stream.
- `Size` is the length of the bit stream, that starts at `Bitstream`.
- `FirstBitposition` is a valid bit index in the bit stream given by `Bitstream` and `Size`

Description

The function `Bitwalker_IcrementalWalker_Init` assigns

- `Bitstream` to `Locals->Bitstream`.
- `Size` to `Locals->Length`
- `FirstBitposition` to `Locals->CurrentBitposition`

3.1.3 `Bitwalker_IncrementalWalker_Peek_Next`

The function `Bitwalker_IcrementalWalker_Peek_Next` reads a sequence from a bit stream and increments the current position in the bit stream by the length of the read bit sequence. Its function signature reads as follows:

```
uint64_t Bitwalker_IncrementalWalker_Peek_Next(
    T_Bitwalker_Incremental_Locals *Locals,
    unsigned int Length);
```

Preconditions

- `Locals` must be dereferenceable
- `Length` is the length of the bit sequence and shall be less or equal 64
- `Locals->CurrentBitposition` $\leq$ `UINT_MAX` $-$ `Length`

Description

`Bitwalker_IcrementalWalker_Peek_Next` reads a bit sequence from a bit stream and returns it as 64 bit integer.

FiXme Fatal: describe more precisely, see `Bitwalker_Peek`

If the bit sequence is not valid the function shall return `0`.

The function increments the value `Locals->CurrentBitposition` by `Length`.

FiXme Fatal: does it make sense to increment in the case of an invalid bit sequence?

3.1.4 `Bitwalker_IncrementalWalker_Peek_Finish`

The function signature reads:

```
int Bitwalker_IncrementalWalker_Peek_Finish(
        T_Bitwalker_Incremental_Locals *Locals);
```

Preconditions

- `Locals` must be dereferenceable

Description

The function returns `Locals->CurrentBitposition`. It does not change any variables.

Remark The return value of this function is `int` despite the fact that it returns a value of type `unsigned int`. This does not make much sense and should best be avoided.

3.1.5 `Bitwalker_IncrementalWalker_Poke_Next`

The function `Bitwalker_IcrementalWalker_Poke_Next` writes a bit sequence into a bit stream and increments the current position in the bit stream by the length of the read bit sequence.

```
int Bitwalker_IncrementalWalker_Poke_Next(
        T_Bitwalker_Incremental_Locals *Locals,
        unsigned int Length,
        uint64_t Value);
```

Preconditions

- `Locals` must be dereferenceable
- `Locals->CurrentBitposition` + `Length` is less than or equal `UINT_MAX`

Description

We specify `Bitwalker_IcrementalWalker_Poke_Next` as follows: The function `Bitwalker_Poke` converts a 64-bit unsigned integer to a bit sequence and writes it into a bit stream.

For $0 \leq x$ exists a shortest sequence of 0 and 1 $(b_0, b_1, \ldots, b_{n-1})$ such that

$$\sum_{i=0}^{n-1} b_i \cdot 2^{(n-1)-i} = x. \tag{3.1}$$

The function `Bitwalker_IcrementalWalker_Poke_Next` tries to store the sequence $(b_0, b_1, \ldots, b_{n-1})$ in the bit sequence of `Length` bits that starts at bit index `Locals->CurrentBitposition`.

The return value depends on the following cases:

- If the bit sequence is not valid `Bitwalker_IcrementalWalker_Peek_Next` returns -1.
- If the bit sequence is valid, then there are two cases:
  - If x is greater or equal than $2^{\texttt{Length}}$, then x cannot be represented as bit sequence $(b_0, b_1, \ldots, b_{\texttt{Length}-1})$. `Bitwalker_IcrementalWalker_Poke_Next` returns then -2.
  - If x is less the $2^{\texttt{Length}}$, then the sequence $(\overbrace{0, \ldots, 0}^{\texttt{Length}-n}, b_0, b_1, \ldots, b_{n-1})$ is stored in the bit stream starting at `Locals->CurrentBitposition`. The return value of the function `Bitwalker_IcrementalWalker_Poke_Next` is 0.

Regardless of whether the poke was successful `Bitwalker_IcrementalWalker_Poke_Next` sets the value `Locals->CurrentBitposition` to the first position behind the sequence that it tired to poke. All other components of the record `Locals` remain unaltered.

FiXme Fatal: does it make sense to increment in the case of an invalid bit sequence?

3.1.6 `Bitwalker_IncrementalWalker_Poke_Finish`

The function signature reads:

```
int Bitwalker_IncrementalWalker_Poke_Finish(
        T_Bitwalker_Incremental_Locals *Locals);
```

Preconditions

- `Locals` must be dereferenceable

Description

The function returns `Locals->CurrentBitposition`.

Remark* The functions `Bitwalker_IcrementalWalker_Peek_Finish` and `Bitwalker_IcrementalWalker_Poke_Finish` have the same specification. Moreover, both function return an object of type `unsigned int` as `int` for which no convincing reason is available.

3.2 Specification of `Bitwalker_Peek` and `Bitwalker_Poke`

This section describes the functions `Bitwalker_Peek` and `Bitwalker_Poke` that are used for the implementation of some functions in Section 3.1.

3.2.1 Informal specification of `Bitwalker_Peek`

Now we specify `Bitwalker_Peek` with the introduced auxiliary concepts. The function `Bitwalker_Peek` reads a bit sequence from a bit stream and converts it to an integer.

Its function signature reads as follows:

```
uint64_t  Bitwalker_Peek(unsigned int Startposition,
                         unsigned int Length,
                         uint8_t Bitstream[],
                         unsigned int BitstreamSizeInBytes);
```

Arguments and Preconditions

The arguments of `Bitwalker_Peek` have the following purpose:

- `Startposition` is the bit index in the bit stream where the bit sequence starts.
- `Length` is the length of the bit sequence.
- `Bitstream` is the array which provides the bit stream.
- `BitstreamSizeInBytes` is the length of the array containing the bit stream.

The following preconditions shall hold for the function arguments. Note that additional constraints are implicitly expressed by the use of *unsigned* integer types.

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Length` $\leq$ `64` and
- `Startposition` $\leq$ `UINT_MAX - Length`. This condition expresses that no arithmetic overflows shall occur when evaluating `Startposition + Length`.

Description

As mentioned, the function `Bitwalker_Peek` reads a bit sequence from a bit stream and converts it to a 64-bit unsigned integer.

For a bit sequence $(b_0, b_1, \ldots, b_{n-1})$ the function `Bitwalker_Peek` returns the sum

$$\sum_{i=0}^{n-1} b_i \cdot 2^{(n-1)-i} \tag{3.2}$$

Note that is a higher-level description than what is done in the source code. There is, in our opinion, not much point to reflect all of the low-level bit operations into the specification if a clearer description is at hand.

If the bit sequence is not valid, then `Bitwalker_Peek` shall return 0. We were wondering why the implementation maps an illegal input to a legitimate output. The code providers argued along the lines that this error condition was not considered important enough to be properly reported. One can interpret this design decision as an attempt to increase the robustness of the function against illegal values. In general, we recommend to explicitly describe all error conditions and to devise a consistent error detection and error recovery strategy.

3.2.2 Informal specification of `Bitwalker_Poke`

In this section we examine the function `Bitwalker_Poke` in the same manner as we did it for `Bitwalker_Peek`.

The function `Bitwalker_Poke` converts an integer to a bit sequence and writes it into a bit stream. Its function signature reads as follows:

```
int       Bitwalker_Poke(unsigned int Startposition,
                         unsigned int Length,
                         uint8_t Bitstream[],
                         unsigned int BitstreamSizeInBytes,
                         uint64_t Value);
```

Arguments and Preconditions

The arguments have the following purpose:

- `Startposition` is the bit index in the bit stream where the bit sequence starts.
- `Length` is the length of the bit sequence.

- `Bitstream` is the array which provides the bit stream.
- `BitstreamSizeInBytes` is the length of the array containing the bit stream.
- `Value` is the integer which shall be converted into a bit sequence.

The following conditions shall hold for the function arguments:

- `Bitstream` is a valid array of length `BitstreamSizeInBytes`
- `Startposition + Length` is less than `UINT_MAX`.

Note that additional constraints are implicitly expressed by the use of *unsigned* integer types.

Description

Now we can specify `Bitwalker_Poke` as follows: The function `Bitwalker_Poke` converts a 64-bit unsigned integer to a bit sequence and writes it into a bit stream.

For $0 \leq x$ exists a shortest sequence of 0 and 1 $(b_0, b_1, \ldots, b_{n-1})$ such that

$$\sum_{i=0}^{n-1} b_i \cdot 2^{(n-1)-i} = x. \tag{3.3}$$

The function `Bitwalker_Poke` tries to store the sequence $(b_0, b_1, \ldots, b_{n-1})$ in the bit sequence of `Length` bits that starts at bit index `Startposition`.

The return value of `Bitwalker_Poke` depends on the following three cases:

- If the bit sequence is not valid, then `Bitwalker_Poke` returns -1.
- If the bit sequence is valid, then there are two cases:
  - If x is greater or equal than $2^{\texttt{Length}}$, then x cannot be represented as bit sequence $(b_0, b_1, \ldots, b_{\texttt{Length}-1})$. `Bitwalker_Poke` returns then -2.
  - If x is less the $2^{\texttt{Length}}$, then the sequence $(\overbrace{0, \ldots, 0}^{\texttt{Length}-n}, b_0, b_1, \ldots, b_{n-1})$ is stored in the bit stream starting at `Startposition`. The return value of `Bitwalker_Poke` is 0.