Generic Application Programming Interface (API) for Sliding Window FEC Codes

Vincent Roca, Jonathan Detchart, Cédric Adjih, Morten Videbæk Pedersen

NWCRG
Internet-Draft
Intended status: Informational
Expires: May 8, 2019

V. Roca (Ed.)
INRIA
J. Detchart
ISAE - Supaero
C. Adjih
INRIA
M. Pedersen
Steinwurf ApS

November 4, 2018

Generic Application Programming Interface (API) for Sliding Window FEC Codes
draft-roca-nwcrg-generic-fec-api-04

Abstract

This document introduces a generic Application Programming Interface (API) for sliding window FEC codes. This API is meant to be compatible with any sliding window FEC code. It defines the core procedures and functions meant to control the codec (i.e., implementation of the FEC code). However, it leaves out all upper layer aspects that are the responsibility of the application or protocol making use of the codec. As a consequence, this is not an API for a FEC Scheme since certain mechanisms that must be defined by any FEC Scheme (e.g., signalling and FEC Payload IDs) are the responsibility of the caller instead of being addressed by the codec. A first goal of this document is to pave the way for a future open-source implementation of such codes, another goal is to simplify the development of content delivery protocols that rely on sliding window FEC codes for robust transmissions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 8, 2019.

Table of Contents

1. Introduction

Forward Erasure Correction (FEC) codes are a key element of communication systems, used to efficiently recover from packet losses during content delivery sessions. Among the FEC codes working at the network and higher layers, one can broadly distinguish block codes and sliding window codes. Block FEC codes require the data flow coming from the application to be segmented into blocks of a predefined maximum size, before generating a certain number of repair packets. With the second type of FEC codes, an encoding window continuously slides over the set of source data and repair packets are generated at any time by computing for instance a linear combination of data present in the encoding window. This fundamental

difference seriously impacts the way they can be used by a content delivery protocol or application.

This document introduces a generic Application Programming Interface (API) for sliding window FEC codes. This API is meant to be usable by any sliding window FEC code and FEC Scheme independently of the protocol that may rely on it. This API defines the core procedures and functions meant to control the codec (i.e., implementation of the FEC code), but leaves out all upper layer aspects that are the responsibility of the application making use of the codec.

This API is meant to be usable by any sliding window FEC code. independently of the FEC Scheme or network coding protocol that may rely on it This API defines the core procedures and functions meant to control the codec (i.e., implementation of the FEC code), but leaves out all upper layer aspects that are the responsibility of the application making use of the codec. For instance, those restricted to end-to-end use-cases as well as those compatible with in-network re-encoding use-cases. Additionally, this API is not impacted by the intra-flow versus inter-flow nature of the use-case, nor is it impacted by the single-path versus multi-paths nature of the use-case, since those are usage considerations under the responsibility of the caller.

A goal of this document is to pave the way for a future open-source implementation of such codes.

2. Definitions and Abbreviations

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

This document uses the following definitions and abbreviations:

XXX

3. AL-FEC Codes and Mechanisms Considered by the Generic API

This generic FEC API is meant to be used with:

- sliding window codes, that manage an encoding window (of fixed or variable size) that slides over the set of source symbols at the sender. On the opposite, block codes (e.g., Reed-Solomon, LDPC, Raptor(Q)) are out of scope;
- codes that are restricted to use-cases that involve a single encoding point and a single decoding point (i.e., FEC operations are carried out either within the end-hosts or middle-boxes), as

well as codes that can be used with use-cases that involve in-network re-coding operations;

- use-cases that are limited to an intra-flow coding (simple case), as well as use-cases that involve inter-flow coding. This second case is more complex to address (e.g., with questions such as how to identify a packet of a flow?) however this is the responsibility of the application or protocol using this codec and not the codec itself. This aspect is therefore transparent to the API;
- use-cases that are limited to single-path communications and use-cases that consider multi-path communications. Here also this is a usage consideration that is transparent to the API;
- use-cases that involve a dynamic adaptation of the codec parameters (e.g., its code rate because the communication path losses is known thanks to feedbacks and an appropriate strategy can be defined);
- fixed code rate or not FEC codes, including rateless codes where the number of repair symbols that can be generated is huge (in theory unlimited);
- ideal (MDS) or non-ideal (non-MDS) codes. However most of the time, sliding window codes are non-ideal codes, meaning that slightly more than l repair symbols may be required to recover all the l lost source symbols;

A key question is to determine what mechanisms are included in the codec and what mechanisms are left to the responsibility of the caller (i.e., an application or a protocol making use of this codec) (Figure 1). More precisely, an FEC Scheme (such as the RLC FEC Scheme [RLC] in case of FECFRAME [fecframe-ext]) defines all the internal code details in order to enable interoperable implementations, but also signaling considerations that are essential to use them in a specific context.

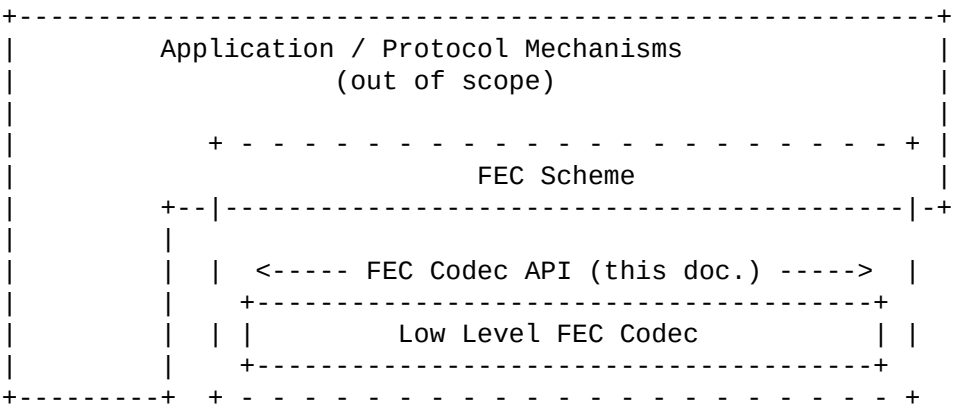

Figure 1: Position of the FEC Codec API with respect to the low level FEC Codec, the FEC Scheme, the protocol and other caller services.

3.1. Mechanisms Considered or Ignored by the API

Applying FEC coding, through an FEC Scheme, in a given protocol to improve transmission robustness involves many mechanisms. However, these mechanisms are not all the responsibility of the codec and can be implemented within the application or within the protocol that uses this FEC codec. For instance, the following mechanisms are considered **out of scope of the API**, being implemented by the caller, without any impact on the codec:

- memory management;
- packet transmission and reception;
- signaling header creation / parsing;
- ADU to source symbol mapping;
- code rate adjustment, for instance thanks to the knowledge of losses at a receiver via feedbacks;
- selective ACK creation and parsing;
- congestion control.

The following mechanisms are **within scope of the API**:

- session management (sender and receiver);
- encoding window management (sender and receiver);
- set/get/generate coding coefficients (sender and receiver);
- build coded symbol (sender only);
- decode with newly received source or repair symbol (receiver only);

4. Generic API for Sliding Window FEC Codes

The following sections describe the generic API, following a C-language formalism. This API tries to adhere to C99 version of C, although it may not strictly be guaranteed. Everything is prefixed by "swif" (sliding window FEC).

4.1. General Definitions Common to the Encoder and Decoder

This section gathers general definitions that are used by both an encoder and decoder.

About FEC Codepoints:

An application first needs to negotiate with its remote side the right FEC Scheme to use. This negotiation usually relies on the FEC Encoding ID associated to this FEC Scheme for this application. The FEC Encoding ID space, associated to an IANA registry, is protocol specific and the same value are usually associated to different FEC Schemes depending on the protocol. The FEC Encoding ID, from the Generic FEC API point of view, cannot be used to uniquely identify the codec.
The use of a codepoint to identify locally the right FEC codec requires that the application knows a mapping between the FEC Encoding ID it uses and the local FEC Codepoints corresponding to available codecs. This can be done at development time, after including the Generic FEC API header file, which gives access to the swif_codepoint_t enumeration.

```
<CODE BEGINS>
/**
 * Return value of any function.
 *
 * SWIF_STATUS_OK = 0   Success
 * SWIF_STATUS_FAILURE  Failure. The function called did not succeed to
 *                      perform its task, however this is not an error
 *                      (e.g., it happens when decoding fails).
 * SWIF_STATUS_ERROR    Generic error type. The detailed error type is
 *                      stored in the errno variable of swif_encoder_t and
 *                      swif_decoder_t structures.
 */
typedef enum {
        SWIF_STATUS_OK = 0,
        SWIF_STATUS_FAILURE,
        SWIF_STATUS_ERROR
} swif_status_t;
```

```
/**
 * Potential errors.
 */
typedef enum {
        SWIF_ERRNO_NULL = 0,            /* everything is fine */
        SWIF_ERRNO_UNSUPPORTED_CODEPOINT,
        /* and many more... */
} swif_errno_t;


/**
 * FEC Codepoints.
 * These identifiers are opaque identifiers that fully identify an FEC
 * code locally, including certain parameters like its Galois Field, or
 * the coding coefficient generator (if several exist).
 * These codepoints are codec specific and only have a local meaning.
 * They should not be transmitted as different implementations may use
 * them inconsistently.
 * Note that the same FEC code may be used by several FEC Encoding IDs
 * and therefore share the same codepoint. On the opposite multiple
 * implementations of a given FEC code may exist locally, for instance
 * with different optimizations, and then several codepoints, one per
 * codec, will exist for the same FEC code. The following names are
 * therefore only provided as examples.
 */
typedef enum {
        SWIF_CODEPOINT_NULL = 0,        /* codepoint 0 is reserved */

        /* codepoint for XXX sliding window code. */
        SWIF_CODEPOINT_XXX_CODEC,

        /* codepoint for YYY sliding window code. */
        SWIF_CODEPOINT_YYY_CODEC,

        /* list here other identifiers for any codec of interest... */
} swif_codepoint_t;


/**
 * Encoding Symbol Identifier (ESI) generic type.
 * With Sliding Window FEC codes, an ESI is in fact a source symbol
 * identifier, unlike block FEC codes.
 */
typedef uint32_t        esi_t;


/**
 * Throughout the API, a pointer to this structure is used as an
```

```
 * identifier of the encoder instance (or "enc").
 *
 * This generic structure is meant to be extended by each codec with
 * new pieces of information that are specific to each codec.
 */
typedef struct swif_encoder {
        swif_codepoint_t       codepoint;

        /* when a function returns with SWIF_STATUS_ERROR, the errno
         * variable contains a more detailed error type. This variable
         * is set by the codec and accessible to the application in
         * READ ONLY mode. Otherwise its value is undefined. */
        swif_errno_t            swif_errno;
} swif_encoder_t;


/**
 * Decoder structure that contains whatever is needed for decoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 */
typedef struct swif_decoder {
        swif_codepoint_t       codepoint;

        /* when a function returns with SWIF_STATUS_ERROR, the errno
         * variable contains a more detailed error type. This variable
         * is set by the codec and accessible to the application in
         * READ ONLY mode. Otherwise its value is undefined. */
        swif_errno_t           errno;
} swif_decoder_t;
<CODE ENDS>
```

General definitions.

4.2. Encoder

```
<CODE BEGINS>
/**
 * Create and initialize an encoder, providing only key parameters.
 *
 * @param codepoint     opaque identifier that fully identifies the FEC
 *                      code to use.
 * @param verbosity     print information on the codec processing.
 *                      0 is the minimum verbosity, the maximum verbosity
 *                      level being implementation specific.
 * @param symbol_size   source and repair symbol size in bytes. Cannot
 *                      change during the codec instance lifetime.
 * @param max_encoding_window_size
```

```
 * @return              pointer to a swif_encoder_t structure if okay, or
 *                      NULL in case of error.
 **/
swif_encoder_t* swif_encoder_create (
                                swif_codepoint_t codepoint,
                                uint32_t        verbosity,
                                uint32_t        symbol_size,
                                uint32_t        max_coding_window_size);


/**
 * Release an encoder and its associated ressources.
 **/
swif_status_t   swif_encoder_release (swif_encoder_t*        enc);


/**
 * Set the various callback functions for this encoder.
 * All the callback functions require an opaque context parameter, that
 * must be initialized accordingly by the application, since it is
 * application specific.
 *
 * @param enc
 * @param source_symbol_removed_from_coding_window_callback
 *              (IN) Pointer to the function, within the application,
 *              that needs to be called each time a source symbol is
 *              removed from the left side of the coding window.
 *              This callback is called each time the encoding window
 *              slides to the right and an old source symbol needs to
 *              be removed on the left. The application therefore knows
 *              this source symbol will no longer be used by the codec
 *              and can free the associated buffer if need be. This
 *              function does not return anything.
 * @param context_4_callback
 *              (IN) Pointer to the application-specific context that
 *              will be passed to the callback function (if any). This
 *              context is not interpreted by this function.
 * @return
 */
swif_status_t   swif_encoder_set_callback_functions (
                swif_encoder_t*        enc,
                void (*source_symbol_removed_from_coding_window_callback) (
                                        void*   context,
                                        esi_t   old_symbol_esi),
                void* context_4_callback);

/**
 * This function sets one or more FEC codec specific parameters,
```

```
 * using a type/length/value approach for maximum flexibility.
 *
 * @param enc
 * @param type          (IN) Type of parameter.
 * @param length        (IN) length of the pointed value.
 * @param value         (IN) Pointer to the value. The exact type of
 *                      the object pointed is FEC codec specific.
 * @return
 */
swif_status_t   swif_encoder_set_parameters  (
                                swif_encoder_t* enc,
                                uint32_t        type,
                                uint32_t        length,
                                void*           value);

/**
 * This function gets one or more FEC codec specific parameters,
 * using a type/length/value approach for maximum flexibility.
 *
 * @param enc
 * @param type          (IN) Type of parameter.
 * @param length        (IN) length of the pointed value.
 * @param value         (IN/OUT) Pointer to the value. The exact type of
 *                      the object pointed is FEC codec specific.
 *                      This function updates the value object
 *                      accordingly. The caller, who knows the FEC codec,
 *                      is responsible to allocate the appropriate
 *                      object buffer.
 * @return
 */
swif_status_t   swif_encoder_get_parameters  (
                                swif_encoder_t* enc,
                                uint32_t        type,
                                uint32_t        length,
                                void*           value);

/**
 * List here the FEC codec specific control parameters.
 */
enum {
        swif_ENCODER_GET_PARAM_ENCODER_STATISTICS = 1,
        swif_ENCODER_SET_PARAM_RLC_DENSITY_THRESHOLD
};


/**
 * Create a single repair symbol (i.e. perform an encoding).
 *
```

```
 * @param new_buf       (IN) The pointer to the buffer for the repair
 *                      symbol to build can either point to a buffer
 *                      allocated by the application, or let to NULL
 *                      meaning that this function will allocate memory.
 * @return
 */
swif_status_t   swif_build_repair_symbol (
                                swif_encoder_t* enc,
                                void*           new_buf);
<CODE ENDS>
```

Encoder API proposal

```
<CODE BEGINS>
/**
 * Encoder structure that contains whatever is needed for encoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 * However it MUST be aligned with swif_encoder_t (same first items) in
 * order to be able to cast a pointer to one of the two structures,
 * depending on the context.
 */
typedef struct swif_encoder_internal {
        swif_codepoint_t       codepoint;

        /* when a function returns with SWIF_STATUS_ERROR, the errno
         * variable contains a more detailed error type. */
        swif_errno_t    swif_errno;

        /* desired verbosity: 0 is the minimum verbosity, the maximum
         * level being implementation specific. */
        uint32_t        verbosity;

        /* maximum number of source symbols used for any repair symbol */
        uint32_t        max_coding_window_size;

        /* exact size (in bytes) of any source or repair symbol */
        uint32_t        symbol_size;

        /* add whatever may be needed hereafter... */
} swif_encoder_internal_t;
```

Non normative example of internal structure used by an encoder.

4.3. Decoder

```
<CODE BEGINS>
/**
 * Create and initialize a decoder, providing only key parameters.
 *
 * @param codepoint     opaque identifier that fully identifies the FEC
 *                      code to use.
 * @param verbosity     print information on the codec processing.
 *                      0 is the minimum verbosity, the maximum verbosity
 *                      level being implementation specific.
 * @param symbol_size   source and repair symbol size in bytes. Cannot
 *                      change during the codec instance lifetime.
 * @param max_coding_window_size
 * @param max_linear_system_size
 * @return              pointer to a swif_decoder_t structure if okay, or
 *                      NULL in case of error.
 **/
swif_decoder_t* swif_decoder_create (
                                swif_codepoint_t codepoint,
                                uint32_t        verbosity,
                                uint32_t        symbol_size,
                                uint32_t        max_coding_window_size,
                                uint32_t        max_linear_system_size);


/**
 * Release a decoder and its associated ressources.
 *
 * @param dec        context (i.e., pointer to decoder structure).
 **/
swif_status_t   swif_decoder_release (swif_decoder_t*        dec);


/**
 * Set the various callback functions for this decoder.
 * All the callback functions require an opaque context parameter, that
 * must be initialized accordingly by the application, since it is
 * application specific.
 *
 * @param dec   context (i.e., pointer to decoder structure).
 * @param source_symbol_removed_from_linear_system_callback
 *              (IN) Pointer to the function, within the application, that
 *              needs to be called each time a source symbol is removed from
 *              the left side of the linear system.
 *              This callback is called each time the linear system slides
 *              to the right and an old source symbol needs to be removed
 *              on the left. This function does not return anything.
```

```
 * @param decodable_source_symbol_callback
 *              (IN) Pointer to the function, within the application, that
 *              needs to be called each time a source symbol is decodable.
 *              What it does is application-dependent, but it MUST return
 *              either a pointer to a data buffer, left uninitialized, of
 *              the appropriate size, or NULL if the application prefers to
 *              let the codec allocate the buffer.
 *              In any case the codec is responsible for storing the actual
 *              symbol value within the data buffer. Also, no matter
 *              whether the data buffer is allocated by the application or
 *              the codec, it is the responsibility of the application to
 *              free this buffer when needed, once decoding is over (but
 *              not before since the codec does not keep any internal copy).
 * @param decoded_source_symbol_callback
 *              (IN) Pointer to the function, within the application, that
 *              needs to be called each time a source symbol is decodable and
 *              all computations performed (i.e., the buffer does contain the
 *              symbol value).
 *              This callback is called in a second time, when the newly
 *              decodable source symbol is actually decoded and ready,
 *              i.e., when all the computations (like XOR and GF(2**8)
 *              operations) have been performed. In any case, it is the
 *              responsibility of the application to free this buffer when
 *              needed, once decoding is over (but not before since the
 *              codec does not keep any internal copy). This function does
 *              not return anything.
 * @param context_4_callback
 *              (IN) Pointer to the application-specific context that will be
 *              passed to the callback function (if any). This context is not
 *              interpreted by this function.
 * @return
 */
swif_status_t   swif_decoder_set_callback_functions (
                swif_decoder_t*  dec,
                void (*source_symbol_removed_from_linear_system_callback) (
                                        void*   context,
                                        esi_t   old_symbol_esi),
                void* (*decodable_source_symbol_callback) (
                                        void    *context,
                                        esi_t   esi),
                void* (*decoded_source_symbol_callback) (
                                        void    *context,
                                        void    *new_symbol_buf,
                                        esi_t   esi),
                void*        context_4_callback);


/**
```

```
 * This function sets one or more FEC codec specific parameters,
 *        using a type/length/value approach for maximum flexibility.
 *
 * @param dec           context (i.e., pointer to decoder structure).
 * @param type          (IN) Type of parameter.
 * @param length        (IN) length of the pointed value.
 * @param value         (IN) Pointer to the value. The exact type of
 *                      the object pointed is FEC codec specific.
 * @return
 */
swif_status_t   swif_decoder_set_parameters  (
                                swif_decoder_t* dec,
                                uint32_t        type,
                                uint32_t        length,
                                void*           value);

/**
 * This function gets one or more FEC codec specific parameters,
 * using a type/length/value approach for maximum flexibility.
 *
 * @param dec           context (i.e., pointer to decoder structure).
 * @param type          (IN) Type of parameter.
 * @param length        (IN) length of the pointed value.
 * @param value         (IN/OUT) Pointer to the value. The exact type of
 *                      the object pointed is FEC codec specific.
 *                      This function updates the value object
 *                      accordingly. The caller, who knows the FEC codec,
 *                      is responsible to allocate the appropriate
 *                      object buffer.
 * @return
 */
swif_status_t   swif_decoder_get_parameters  (
                                swif_decoder_t* dec,
                                uint32_t        type,
                                uint32_t        length,
                                void*           value);

/**
 * List here the FEC codec specific control parameters.
 */
enum {
        swif_DECODER_GET_PARAM_DECODER_STATISTICS = 1,
        swif_DECODER_SET_PARAM_RLC_DENSITY_THRESHOLD
};


/**
 * Submit a received source symbol and try to progress in the decoding.
```

```
 * For each decoded source symbol (if any), the application is informed
 * through the dedicated callback functions.
 *
 * This function usually returns SWIF_STATUS_OK, regardless of whether
 * this new symbol enabled the decoding of one or several source symbols,
 * or SWIF_STATUS_ERROR. It cannot return SWIF_STATUS_FAILURE.
 *
 * @param dec   context (i.e., pointer to decoder structure).
 * @param new_symbol_buf
 *              (IN) Pointer to the new source symbol now available (i.e.
 *              a new symbol received by the application, or a decoded
 *              symbol in case of a recursive call if it makes sense).
 * @param new_symbol_esi
 *              (IN) encoding symbol ID of the new source symbol.
 * @return
 */
swif_status_t   swif_decoder_decode_with_new_source_symbol (
                                swif_decoder_t* dec,
                                void* const     new_symbol_buf,
                                esi_t           new_symbol_esi);


/**
 * Submit a received repair symbol and try to progress in the decoding.
 * For each decoded source symbol (if any), the application is informed
 * through the dedicated callback functions.
 *
 * This function requires that the application has previously initialized
 * the coding window and coding coefficients appropriately. The application
 * keeps a full control of the repair symbol buffer, i.e., the application
 * is in charge of freeing this buffer as soon as it believes appropriate
 * (a copy is kept by the codec). This is motivated by the fact that a
 * repair symbol may be part of a larger buffer (e.g., if there are
 * several repair symbols per packet, or because of a packet header): only
 * the application knows when the buffer can be safely freed.
 *
 * This function usually returns SWIF_STATUS_OK, regardless of whether
 * this new symbol enabled the decoding of one or several source symbols,
 * or SWIF_STATUS_ERROR. It cannot return SWIF_STATUS_FAILURE.
 *
 * @param dec   context (i.e., pointer to decoder structure).
 * @param new_symbol_buf
 *              (IN) Pointer to the new repair symbol now available (i.e.
 *              a new symbol received by the application or a decoded
 *              symbol in case of a recursive call if it makes sense).
 * @return
 */
swif_status_t   swif_decoder_decode_with_new_repair_symbol (
```

```
                                swif_decoder_t* dec,
                                void* const     new_symbol_buf);
<CODE ENDS>
```

Decoder API proposal

```
<CODE BEGINS>
/**
 * Decoder structure that contains whatever is needed for decoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 * However it MUST be aligned with swif_decoder_t (same first items) in
 * order to be able to cast a pointer to one of the two structures,
 * depending on the context.
 */
typedef struct swif_decoder_internal {
        swif_codepoint_t       codepoint;

        /* when a function returns with SWIF_STATUS_ERROR, the errno
         * variable contains a more detailed error type. */
        swif_errno_t           errno;

        /* desired verbosity: 0 is the minimum verbosity, the maximum
         * level being implementation specific. */
        uint32_t               verbosity;

        /* maximum number of source symbols used for any repair symbol */
        uint32_t               max_coding_window_size;

        /* max. number of source symbols kepts in current linear system.
         * If the linear system grows above this limit, old source
         * symbols in excess are removed and the application callback
         * called. This value should be larger than the
         * max_coding_window_size. */
        uint32_t               max_linear_system_size;

        /* exact size (in bytes) of any source or repair symbol */
        uint32_t               symbol_size;

        /* add whatever may be needed hereafter... */
} swif_decoder_internal_t;
```

Non normative example (RLC) of internal structure used by a decoder.

4.4. Coding Window Functions at an Encoder and Decoder

This section gathers functions used to manage the coding window, both at an encoder and at a decoder. At an encoder a sliding (of fixed or elastic size) encoding window is managed. Whenever a repair symbol needs to be created, a linear combination (that is code specific) of source symbols currently in the encoding window is performed. This encoding window is managed with the functions below plus, potentially, internal mechanisms that are code specific.

At a decoder, before submitting a new repair symbol to the codec, the application must specify the associated encoding window used at the source. This is done by the reset/add a single or set of symbols/ remove a symbol functions. Once this coding window is ready, as well as the coding coefficient list if applicable, the application calls the decode_with_new_repair_symbol() function. A coding window may be reused for several repair symbols as long as they are all built from the same set of source symbols. In that case resetting the coding window and setting it from scratch would be a waste of time. The coding window must be viewed as a temporary list used solely by the decode_with_new_repair_symbol() function and kept independent from the linear system managed by the codec.

```
<CODE BEGINS>
/**
 * This function resets the current coding window. We assume here that
 * this window is maintained by the FEC codec instance.
 * Encoder:     reset the encoding window for the encoding of future
 *              repair symbols.
 * Decoder:     reset the coding window under preparation associated to
 *              a repair symbol just received.
 *
 * @return
 */
swif_status_t   swif_encoder_reset_coding_window (swif_encoder_t*  enc);

swif_status_t   swif_decoder_reset_coding_window (swif_encoder_t*  dec);


/**
 * Add this source symbol to the coding window.
 * Encoder:     add a source symbol to the coding window.
 * Decoder:     add a source symbol to the coding window under preparation.
 *
 * @param new_src_symbol_buf    (encoder only) pointer to a buffer
 *              containing the source symbol. The application MUST NOT
 *              free nor modify this buffer as long as the source symbol
 *              is in the coding window.
```

```
 * @param new_src_symbol_esi    ESI of the source symbol to add.
 * @return
 */
swif_status_t   swif_encoder_add_source_symbol_to_coding_window (
                                swif_encoder_t* enc,
                                void*           new_src_symbol_buf,
                                esi_t           new_src_symbol_esi);

swif_status_t   swif_decoder_add_source_symbol_to_coding_window (
                                swif_decoder_t* dec,
                                esi_t           new_src_symbol_esi);


/**
 * Remove this source symbol from the coding window.
 *
 * Encoder:     remove a source symbol from the encoding window, e.g.
 *              because the application knows that a source symbol has
 *              been acknowledged by the peer (if applicable). Note that
 *              the left side of the sliding window is automatically
 *              managed by the codec and no action is needed from the
 *              application. If needed a callback is available to inform
 *              the application that a source symbol has been removed).
 * Decoder:     remove a source symbol from the coding window under
 *              preparation.
 *
 * @param old_src_symbol_esi    ESI of the source symbol to remove from
 *              the coding window.
 * @return
 */
swif_status_t   swif_encoder_remove_source_symbol_from_coding_window (
                                swif_encoder_t* enc,
                                esi_t           old_src_symbol_esi);

swif_status_t   swif_decoder_remove_source_symbol_from_coding_window (
                                swif_decoder_t* dec,
                                esi_t           old_src_symbol_esi);
<CODE ENDS>
```

Coding Window Functions at an Encoder and Decoder.

4.5. Coding Coefficients Functions at an Encoder and Decoder

This section gathers functions used to manage the coding coefficients, both at an encoder and at a decoder. Since different FEC codecs will have different requirements, it is important to keep these functions separate from the build_repair_symbol() and decode_with_new_repair_symbol() functions. Several situations exist:

- the application provides the list of coding coefficients to use for the next build_repair_symbol();
- the application provides a key (typically a PRNG seed) that the codec uses to produce the coding coefficients to use for the next build_repair_symbol();
- the choice of the coding coefficients is totally performed by the codec, in an autonomous manner (e.g., the codec includes an algorithm that produces an appropriate seed based on various criteria, or the codec selects a set of coding coefficients based on various criteria). In that case the application needs to retrieve the list of coding coefficients or the key selected by the codec;

```
<CODE BEGINS>
/**
 * The following functions enable an encoder (resp. decoder) to
 * initialize the set of coefficients to be used for encoding
 * or associated to a received repair symbol.
 *
 * Encoder: calling one of them MUST be done before calling
 *              build_repair_symbol().
 * Decoder: calling one of them MUST be done before calling
 *              decode_with_new_repair_symbol().
 */

/**
 * Encoder: this function specifies the coding coefficients chosen by
 *          the application if this is the way the codec works.
 * Decoder: communicate with this function the coding coefficients
 *          associated to a repair symbol and carried in the packet
 *          header.
 *
 * @param coding_coefs_tab
 *              (IN) table of coding coefficients associated to each of
 *              the source symbols currently in the encoding window.
 *              The size (number of bits) of each coefficient depends on
 *              the FEC Scheme. The allocation and release of this table
 *              is under the responsibility of the application.
 * @param nb_coefs_in_tab
 *              (IN) number of entries (i.e., coefficients) in the table.
 * @return
 */
swif_status_t   swif_encoder_set_coding_coefs_tab (
                                swif_encoder_t* enc,
                                void*           coding_coefs_tab,
                                uint32_t        nb_coefs_in_tab);

swif_status_t   swif_decoder_set_coding_coefs_tab (
```

```
                                swif_decoder_t* dec,
                                void*           coding_coefs_tab,
                                uint32_t        nb_coefs_in_tab);


/**
 * The coding coefficients may be generated in a deterministic manner,
 * for instance by a PRNG known by the codec and a seed (perhaps with
 * other parameters) provided by the application.
 * The codec may also choose in an autonomous manner these coefficients.
 * This function is used to trigger this process.
 * When the choice is made in an autonomous manner, the actual coding
 * coefficient or key used by the codec can be retrieved with
 * swif_encoder_get_coding_coefs_tab().
 *
 * @param key   (IN) Value that can be used as a seed in case of a PRNG
 *              for instance, or by a specific coding coefficients
 *              function. Set to 0 if not required by a codec.
 * @param add_param
 *              (IN) an opaque 32-bit integer that contains a codec
 *              specific parameter if needed. Set to 0 if not used.
 * @return
 */
swif_status_t   swif_encoder_generate_coding_coefs (
                                swif_encoder_t* enc,
                                uint32_t        key,
                                uint32_t        add_param);

swif_status_t   swif_decoder_generate_coding_coefs (
                                swif_decoder_t* dec,
                                uint32_t        key,
                                uint32_t        add_param);


/**
 * This function enables the application to retrieve the set of coding
 * coefficients generated and used by build_repair_symbol(). This is
 * useful when the choice of coefficients is performed by the codec in
 * an autonomous manner but needs to be sent in the repair packet header.
 * This function is only used by an encoder.
 *
 * @param coding_coefs_tab
 *              (OUT) pointer to a table of coding coefficients.
 *              The size (number of bits) of each coefficient depends on
 *              the FEC scheme. Upon return of this function, this table
 *              is allocated and filled with coefficient values. The
 *              release of this table is under the responsibility of the
 *              application.
```

```
 * @param nb_coefs_in_tab
 *              (IN/OUT) pointer to the number of entries (i.e.,
 *              coefficients) in the table.
 *              Upon calling this function, this number must be zero.
 *              Upon return of this function this variable is initialized
 *              with the actual number of entries in the coeffs_tab[].
 * @return
 */
swif_status_t   swif_encoder_get_coding_coefs_tab (
                                swif_encoder_t* enc,
                                void**          coding_coefs_tab,
                                uint32_t*       nb_coefs_in_tab);


/**
 * Get information on the current coding window at the encoder.
 * This function stores the ESI of the first source symbol and
 * last source symbol in the coding window, as well as the number
 * of symbols. In theory the application should be able to recover
 * the information (it knows when new symbols are added and old
 * symbols removed), but it's easier to let the SWiF Codec care
 * about it. The number of source symbols is also returned.
 * In situations where there's no gap (i.e., when
 * swif_encoder_remove_source_symbol_from_coding_window() has not
 * been used), nss can also be calculated with first/last. However
 * it is more convenient to use nss directly (in particular in case
 * of wraping to zero of either first or last).
 *
 * @param enc
 * @param first         (in/out) pointer to ESI of the first source
 *                      symbol in the coding window (inclusive)
 * @param last          (in/out) pointer to ESI of the last source
 *                      symbol in the coding window (inclusive)
 * @param nss           (in/out) pointer to number of source symbols
 *                      in the coding window
 * @return
 */
swif_status_t   swif_encoder_get_coding_window_information (
                                swif_encoder_t* enc,
                                esi_t*          first,
                                esi_t*          last,
                                uint32_t*       nss);
<CODE ENDS>
```

Coding Coefficients Functions at an Encoder and Decoder.

5. Security Considerations

TBD

6. IANA Considerations

This document has no IANA requirement.

7. Acknowledgments

The authors would like to thank TBD.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

8.2. Informative References

[fecframe-ext] Roca, V. and A. Begen, "Forward Error Correction (FEC) Framework Extension to Sliding Window Codes", Transport Area Working Group (TSVWG) draft-ietf-tsvwg-fecframe-ext (Work in Progress), June 2018, <https://tools.ietf.org/html/draft-ietf-tsvwg-fecframe-ext>.

[RLC] Roca, V. and B. Teibi, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Scheme for FECFRAME", Transport Area Working Group (TSVWG) draft-ietf-tsvwg-rlc-fec-scheme (Work in Progress), June 2018, <https://tools.ietf.org/html/draft-ietf-tsvwg-rlc-fec-scheme>.

Authors' Addresses

Vincent Roca
INRIA
Univ. Grenoble Alpes
France

EMail: vincent.roca@inria.fr

Jonathan Detchart
ISAE - Supaero
France

EMail: jonathan.detchart@isae-supaero.fr

Cedric Adjih
INRIA
France

EMail: cedric.adjih@inria.fr

Morten V. Pedersen
Steinwurf ApS
Denmark

EMail: morten@steinwurf.com