Document Release Status	
Release Date 2013-05-02	Version

Versatile Binary Format Specification 3.0

	Document Name **VERSATILE BINARY FORMAT SPECIFICATION 3.0**			
	Document Type			
	Owner Domain: Document Prefix			
	Document No **00.06.15.004**	Revision **007**	Volume No **01**	Page No (in this doc.) **1 (28)**

Revision History

Previous Version	Current Version	Version Description	Authors (dept, name, email)	Date
X05	001	1st Release	94230 Olof Hansson, ohansson@volvocars.com	2002-12-03
001.1	002	2nd Release	94230 Olof Hansson, ohansson@volvocars.com	2004-02-06
002.2	003	3rd Release	94230 Olof Hansson, ohansson@volvocars.com	2004-08-17
003	004	4th Release	Vijaya Pinnamaneni vpinnama@ford.com	2006-01-09
004	005	5th Release	94230 Olof Hansson, ohansson@volvocars.com	2006-08-23
005	006	6th Release	94232 Magnus Persson mperss54@volvocars.com and Vijaya Pinnamaneni vpinnama@ford.com	2008-02-22
006	006.1	Draft Release	Jason Miller jmille72@ford.com	2012-05-10
006.1	006.2	Draft Release	Jason Miller jmille72@ford.com	2013-04-22
006.2	007	Official Release	Jason Miller jmille72@ford.com	2013-05-02

Change log

Release	Section	Change Description
006.1	All	vbf_version updated to 3.0
006.1	Change Log	Removed old change log from previous version.
006.1	All	Removed any references to PAG / non-PAG and kept non-PAG requirements
006.1	All	Removed any references to Aston Martin, Jaguar, Land Rover, Mazda, and Volvo.
006.1	4.2, 4.2.2	Updated the 2nd software part number value to be more generic (i.e., dependent upon OEM and not just limited to old Volvo KDP).
006.1	4.2.7	Clarified erase identifier usage.
006.1	4.2.8	Added requirement that if omit identifier is present, it must align with at least one address in the data section or erase identifier values.
006.1	All	Removed network identifier. This identifier caused confusion as it is independent of the software file, and was capable of changing from one vehicle to another (even if the ECU speed did not change), due to transparent CAN diagnostic gateways.
006.1	Table 2	Updated definitions to align with other core specifications.
006.1	All	Removed checksum_table identifier as this was Volvo specific.
006.1	3.1	Clarified usage of the parentheses and | separator
006.1	2.5	Deleted any reference to specific generation tools from this spec.
006.2	4.3	Clarified that length is always length of data being transmitted and therefore represents compressed and/or encrypted length.
006.2	4.3.1	Clarified that block checksums are always over data in ECU memory and therefore are calculated over the uncompressed and/or unencrypted data.
007	4.2.7	Added a recommendation for the address and length pairs to align with each physical flash sector instead of grouping flash sectors together.
007	1.2	Removed reference to NetCom eRoom.

Contents

List of Figures

List of Tables

1. Introduction

Software download involves the transfer of information from a tester to an ECU using data normally stored in a file. The data transferred may contain everything from configuration parameters of the ECU to a completely new ECU application.

1.1 Purpose/Scope

The purpose of this document is to define the format of the file used for software download for officially released Ford software parts.

1.2 Applicable Documents/References

Table 1: Applicable Documents/References

Reference Num	Source	Title	Version or date	Document Number
[1]	FMC	Software Download Specification	Latest Available	00.06.15.002
[2]	FMC	Data Compression and Encryption Specification	Latest Available	00.06.15.005

1.3 Abbreviations/Acronyms

CM:	CoMments
ECU:	Electronic Control Unit with a computer/microcontroller
GBL:	Gateway BootLoader
PBL:	Primary BootLoader
SBL:	Secondary BootLoader
VBF:	Versatile Binary Format
WERS:	Worldwide Engineering Release System
WS:	White Space

1.4 Definitions

Table 2: Definitions

Term	Definition
Main network	A network connected to the vehicle diagnostic connector which is partly or entirely used for diagnostic communications
Sub network	A network connected to the main network via a diagnostic gateway which is partly or entirely used for diagnostic communications.
Public network	Main or sub network
Private network	A communication network that is not connected to the vehicle diagnostic connector (either directly or through a diagnostic gateway). Note that this definition applies to this specification and is not necessarily applicable to any other specification.
Main node	An ECU that is connected to a main network. The ECU may or may not include a diagnostic gateway.
Sub node	An ECU that is connected to a sub network.
Transparent CAN Diagnostic Gateway	An ECU which connects a public CAN network to a CAN sub network in order to allow a tester to perform diagnostics on the sub-network ECUs. This is achieved by the gateway always relaying raw diagnostic request and response CAN frames in a manner that does not require any knowledge of the sub network existence to the tester. The sub network protocol is limited to CAN.
Explicit Diagnostic Gateway	An ECU which connects a public CAN network to a sub network in order to allow a tester to perform diagnostics on the sub-network ECUs. This approach requires explicit tester knowledge of the sub network in order to perform most diagnostic tasks. The sub network protocol is not limited to CAN.

2. Versatile Binary Format (VBF)

2.1 VBF version

This document specifies VBF version 3.0 of the Versatile Binary Format.

2.2 Introduction

The VBF file contains three parts: VBF version section, a header section and a data section. VBF version section specifies the version number of the VBF specification that the file is compliant with. VBF version uses ASCII representation. The header section contains general information relating to file usage, setup, etc. The header section uses ASCII representation. The data section contains the actual data that is to be transferred to the target ECU. The data section uses binary representation.

Table 3: An Overview of the Versatile Binary Format

Name	Description	Type	Data
Version Section	VBF Version Number	Text	vbf_version=3.0;
Header section	Header	Text	n bytes, ASCII
Data section	Block 1	Start address	4 bytes, binary
		Length	4 bytes, binary
		Data	n bytes, binary
		Checksum	2 byte, binary
	Block 2	Start address	4 bytes, binary
		Length	4 bytes, binary
		Data	n bytes, binary
		Checksum	2 byte, binary
	:	:	:
	Block n	Start address	4 bytes, binary
		Length	4 bytes, binary
		Data	n bytes, binary
		Checksum	2 byte, binary

2.3 File Naming

All delivered or released VBF files shall be named the same as the corresponding released software part number (i.e., sw_part_number as described in section 4.2.2) with an extension of ".VBF".

Example:

1. sw_part_number contains WERS part number

   A VBF file with a header entry of

   sw_part_number = "YW4T-13B525-AB";

   shall utilize the file name of YW4T-13B525-AB.VBF

3. Notation and lexical elements

The VBF file header section uses some basic lexical elements, which are described, in this section. Also, the notation for describing the syntax is given here.

3.1 Notation

Words in bold indicate identifiers, which are special tokens.

Italicized words indicate a lexical item, and are described in text format.

The grouping symbols [ ], brackets, indicate that the contents are optional.

The grouping symbols { }, braces, group a set of items.

The | symbol separates a list of items where one and only one item must be present. Each item in the list must be enclosed by parentheses ( ).

3.2 General structure

The header consists of sequences of expressions. An expression is made up of identifiers, reserved words, constants, integers, reals and is always terminated with a semicolon (;). Expressions containing multiple values are grouped into nested sections using matching brace characters "{" and "}". Data shall be contained in braces only if there is more than one set of data with the exception of **description** [see section 4.2.1], **erase** [see section 4.2.7], and **omit** [see section 4.2.8] identifiers. In these cases, a single set of data shall also be contained in braces. Refer to section 3.3 for general format requirements for expressions and to section 4 for detailed format requirements for each identifier.

3.3 Identifiers, reserved words, and expressions

Identifier is the name of a variable and IdentifierValue is the data assigned to the variable. Allowed identifiers are **vbf_version**, **header**, **description**, **sw_part_number**, **sw_part_type**, **data_format_identifier**, **ecu_address**, **frame_format**, **erase**, **omit**, **call**, and **file_checksum**. Identifiers other than those explicitly allowed by this specification are not permitted. Duplicates of identifiers are not allowed. Each element within an IdentifierValue shall either be an integer, real, string embedded within quotes, or a reserved IdentifierValue (reserved word). Each identifier has a specific format allowed for its identifier values as defined in section 4. The lexical rules for identifiers and reserved words are identical to those for "C" programming language with upper- and lower-case always distinguished.

Unless otherwise specified in this document, a valid expression shall always consist of the following format (where WS/CM is defined as any combination of white space characters and complete comments).

```
[WS/CM]identifier[WS/CM]=[WS/CM]IdentifierValue[WS/CM];[WS/CM]
```

When the IdentifierValue is enclosed within matching brace characters "{" and "}" as allowed by this specification, the format of the IdentifierValue (including braces) shall be as follows:

```
{[WS/CM]IdentifierValue[WS/CM][,[WS/CM]IdentifierValue[WS/CM]]};
```

where the optional field of [,[WS/CM]IdentifierValue[WS/CM]] may occur multiple times.

Example:
sw_part_type = CARCFG;

In this example, **sw_part_type** is an identifier, and CARCFG is the reserved word value of the identifier **sw_part_type**.

3.4 Integers

An integer is a sequence of digits, optionally preceded by an indication of the base. The prefix “0x” specifies that the number is in hexadecimal, and the following characters must be chosen from the string “0123456789abcdefABCDEF”, with the conventional meaning. The prefix “0b” specifies that the number is in binary, and following digits must be “0” or “1”. When no prefix is given, the number is assumed to be in decimal. Thus the strings:

`255`, `0xFF` and `0b11111111`

all represent the same number. If an integer number contains invalid character(s), the VBF parser shall report this as an error.

3.5 Reals

A real is a sequence of digits, followed by a full stop character “.”, and another sequence of digits. As with an integer, prefixes can be given to specify hexadecimal or binary, but these representations should be used with care. If a real number contains invalid character(s), the VBF parser shall report this as an error.

3.6 Comments

Comments are allowed in the header section only and may appear within an expression as described in section 3.3. Valid comments shall always be ignored by a VBF parser. Comments can be grouped together by “/*” and “*/” and stretch over several lines (C-style), or they can start with “//” and stretch to the end of the line (C++ style). Neither bracket syntax nests. It is recommended that C++-style comments be used wherever possible, since it is much more difficult to accidentally comment out more than was intended (as compared to using the C-style syntax).

3.7 White Space Characters

Blank Space, Horizontal Tab (or Tab), Line Feed, Carriage Return, Form Feed, and Vertical Tab characters are defined as white space characters for the purpose of this specification. No other white space characters are allowed. A VBF parser shall always ignore valid white space characters unless otherwise specified in this document.

Table 4: ASCII Representation of White Space Characters

	Description	Symbol	Hex Value	Binary Value
1	Horizontal Tab (or Tab)	HT	09	00001001
2	Line Feed	LF	0A	00001010
3	Vertical Tab	VT	0B	00001011
4	Form Feed	FF	0C	00001100
5	Carriage Return	CR	0D	00001101
6	Blank Space	SPACE	20	00100000

3.8 Non-printing Characters

0x00 to 0x1F represents non-printing characters. All Non-printing characters, except white space characters, are not allowed in VBF file header. For white space non-printing (0x09 to 0x0D) character handling, refer to section 3.7.

4. Overview of the VBF file

4.1 Version Section

The first line of the VBF file shall indicate the version of the VBF file. No comments or non-white space characters are allowed before, within, or after the expression. No white space characters are allowed before the expression. Note that this means the first byte of the VBF file shall always be 0x76 (i.e., ASCII 'v'). This specification is for version 3.0 of the VBF file, and so the first line shall always read:

vbf_version = 3.0;

Examples:

Valid Format:

```
1.  vbf_version=3.0;

2.  vbf_version  =

                3.0  ;
```

Invalid Format:

```
1.       vbf_version=3.0;

2.  /*Comment*/vbf_version/* Comment*/=/*Comment*/3.0/*Comment*/;

3.  text vbf_version text= text 3.0 text; text
```

vbf_version is immediately followed by the **header** identifier with the exception of white space as described above. The header section as explained below contains all information needed to identify the VBF file and also information for the tester to perform a software download operation.

4.2 Header section

All the header information is contained in braces { }. The information between the braces in the header section shall consist entirely of expressions as defined in section 3.3. No comments or non-white space characters are allowed before **header** identifier or in between the **header** identifier and beginning braces ({) .

Examples:

Valid Format:

```
1.  vbf_version = 3.0; header {

2.  vbf_version = 3.0;
    header{

3.  vbf_version=3.0;
        header
                               {
```

Invalid Format:

```
1.  vbf_version=3.0;
    /*Comment*/
    header{

2.  header /*Comment*/ {
```

The format of the header section is defined as follows:

```
header {

          [description =  { "text string row 1" ,
                            "text string row 2" ,
                            "text string row 3" ,
                                    :
                            "text string row n"
                          } ; ]


          sw_part_number = ("WERS number") | ({"WERS number", "Other OEM number"});

          sw_part_type = type ;

          [data_format_identifier = data compression and encryption method;]

          ecu_address = (main node address) |
                        ({ main node address, sub network address, sub node address }) ;

          frame_format = network_frame_format;

          [erase =  {   { start address 1, length 1 },
                        { start address 2, length 2 },
                                    :
                        { start address n, length n }
                    } ; ]

          [omit =   {   { start address 1, length 1 },
                        { start address 2, length 2 },
                                    :
                        { start address n, length n }
                    } ; ]

          [call = start address ; ]

          file_checksum = checksum;
}
```

4.2.1 description

The **description** identifier is optional and can be used for a short description of the file. The **description** IdentifierValue may contain a maximum of 16 rows, with each row containing a maximum of 80 characters (bytes). Each row is contained in quotes (") and is terminated by a comma (,) or by right brace (}) followed by semicolon (;) if the row is a last row. White space within quotes is not ignored. All of the description rows shall be contained within braces whether there is one row or multiple rows. No quotes (") are allowed within a row.

Examples:

Valid Format:

```
1.  description =    {"Created: 2002-03-14"};

2.  description =    { "Software for U38X " ,
                       "Created: 2012-03-14"
                     };
```

Invalid Format:

```
1.  description =      " Created: 2012-03-14" ;          // Missing braces

2.  description =      "Software for U38X AWD" ,         // Missing braces
                       "Created: 2012-03-14";

3.  description =    { Software for U38X AWD,            // Missing quotes
                        Created: 2012-03-14
                     };
4.  description =    { "Software for "U38X" AWD"};       // Extra quotes
```

4.2.2 sw_part_number

The **sw_part_number** IdentifierValue contains the vehicle manufacturer part number for identification of the software component (VBF file). The **sw_part_number** IdentifierValue shall be contained in quotes and shall consist of a maximum of 24 characters (bytes). No white space characters or comments are allowed within these quotes. **sw_part_number** IdentifierValue is not case sensitive and may contain either upper- or lower-case characters.

The **sw_part_number** identifier can have either one or two identifier values assigned. If there is only one identifier value assigned, it shall be the WERS part number for Ford released parts. If there are two identifier values assigned, the first value shall always be WERS part number, and the second value shall always be the other OEM's part number. The second value is intended to provide linkage to software parts which are shared with other OEMs. If **sw_part_number** identifier contains more than one IdentifierValue, the values shall be separated by comma and contained in braces.

The software part number shall reside somewhere in the data section in the VBF file that is being programmed into the ECU and this downloaded part number shall be readable using service 0x22 (ReadDataByIdentifier) via one of the defined part number dataIdentifiers (e.g., 0xF188, 0xF120-0xF128, 0xF108, 0xF10A, 0xF16B-0xF16E, 0xF17D).

Examples:

Valid Format:

```
1.  sw_part_number = "YW4T-13B525-AB";                          // WERS format

2.  sw_part_number = {"YW4T-13B525-AB","31808832AB"};      // Both WERS & other OEM numbers
```

Invalid Format:

```
1.  sw_part_number = YW4T-13B525-AB;                                        // Missing quotes

2.  sw_part_number = "12YW4T-13B52578-ABCDEFGHI";                 // Maximum characters > 24

3.  sw_part_number =  "   YW4T-13B525-AB   ";                      // White Space within quotes

4.  sw_part_number =  "/*Comment*/YW4T-13B525-AB/*Comment*/";      // Comments within quotes

5.  sw_part_number = "YW4T-13B525-AB","31808832AB";               // Missing braces

6.  sw_part_number = {"31808832AB","YW4T-13B525-AB"};             // Incorrect Order
```

4.2.3 sw_part_type

The **sw_part_type** identifier indicates different types of a software parts in an ECU. Only the reserved word IdentifierValues defined in Table 5 shall be considered valid for the **sw_part_type** identifier.

Table 5: Software Part Types and Allowed Reserved Words

sw_part_type IdentifierValue Reserved Words	Description
CARCFG	Car configuration
CUSTOM	Customer parameters
DATA	Data or parameters (i.e., calibrations)
EXE	Executable (i.e., strategy)
GBL	Gateway Bootloader
SBL	Secondary Bootloader
SIGCFG	Signal configuration
TEST	Test program, (i.e. production test, diagnostics)

The identifier **sw_part_type** in a single VBF file can only be assigned one of the reserved word values in the table above.

A VBF file with **sw_part_type = SBL or GBL** is not allowed to use the identifier **erase**. These software types are downloaded to RAM and no erase operation shall be performed in this case.

Examples:

Valid Format:

```
1.  sw_part_type = DATA ;

2.  sw_part_type = EXE ;                       // or any other valid type but "SBL" or "GBL"
    erase = {  { 0x00004200, 0x00003CBF },
               { 0x00008010, 0x00002FF3 }
            };
```

Invalid Format:

```
1.  sw_part_type = exe;                        //  Invalid case

2.  sw_part_type = SBL;                        // Invalid combination of sw_part_type and erase
    erase = {  { 0x00004200, 0x00003CBF },
               { 0x00008010, 0x00002FF3 }
            } ;

3.  sw_part_type = MYOWNTYPE;                  // Unknown IdentifierValue
```

4.2.4 data_format_identifier

The **data_format_identifier** is optional and when present with a non-zero value in the header section it represents that a data compression/encryption method is used. Valid values for the **data_format_identifier** are one byte long with a range of 0x00 to 0xFF with upper nibble representing "Data Compression Method" and lower nibble representing "Data Encryption Method". For the values, refer to methods defined in "Data Compression and Encryption Specification", see ref [2]. If the identifier is not present in the header section, it represents that neither compression nor encryption is used.

Examples:

Valid Format:

1. data_format_identifier = 0x00;	// 0x00 represents no compression or no encryption
2. data_format_identifier = 0x10;	// "1" represents that Compression Method #1, see ref [2] // "0" represents that no encryption is used
3. data_format_identifier = 23;	// 0x17 (Compressed data with Compression Method #1 // and encryption method #7)

Invalid Format:

1. data_format_identifier = 0x123;	// Invalid format (must be from $00 - $FF)
2. data_format_identifier = {0x10};	// Extra braces

4.2.5 ecu_address

The **ecu_address** identifier indicates the physical ECU address of a node on a network (and also the sub network address and sub node address if applicable). The sub network address and sub node address parameters shall only be used when the VBF file is for a sub node that communicates to a tester through an explicit diagnostic gateway. If **ecu_address** identifier contains more than one value, the values shall be contained in braces.

If the frame_format (see section 4.2.6) is equal to CAN_EXTENDED, then the following restrictions apply:

- Valid values for the main node address are one byte long with a range of 0x00 to 0xFF
    - Note that if the VBF file is for a sub node, the main node address shall be 0x00 since it is not used in addressing the sub node with 29-bit CAN IDs
- Valid values for the sub network address are one byte long with a range of 0x00 to 0x07
- Valid values for the sub node address are one byte long and range from 0x00 to 0xFF

If the frame_format (see section 4.2.6) is equal to CAN_STANDARD, then the following restrictions apply:

- Valid values for the main node address are 11 bits long with a range of 0x000 to 0x7FF
- Valid values for the sub network address are one byte long with a range of 0x00 to 0xFF
- Valid values for the sub node address are one byte long with a range of 0x00 to 0xFF

Examples:

Valid Format:

```
1.  frame_format = CAN_EXTENDED;              // Main node with address 0x51
    ecu_address = 0x51;

2.  frame_format = CAN_EXTENDED;              // Sub network address 0x06, sub node address 0x65
    ecu_address = {0x00, 0x06, 0x65};

3.  frame_format = CAN_STANDARD;              // Main node with ECU diagnostic reception ID 0x723
    ecu_address = 0x723;

4.  frame_format = CAN_STANDARD;              // Main node (gateway) ECU diagnostic reception ID 0x723
    ecu_address = {0x723, 0x00, 0x65};        // sub network address 0x00, sub node address 0x65
```

Invalid Format:

```
1.  frame_format = CAN_EXTENDED;
    ecu_address = {0x723, 0x00, 0x65};        // Invalid main node address

2.  frame_format = CAN_STANDARD;
    ecu_address = 0x10;                       // Invalid main node address

3.  ecu_address = {0x723};                    // Extra braces, Missing fields
```

4.2.6 frame_format

The **frame_format** identifier indicates if the standard frame format (11-bit CAN identifiers) or the extended frame format (29-bit CAN identifiers) shall be used for requests and responses to an ECU on a main network. Only the reserved word identifier values defined in Table 6 shall be considered valid for the **frame_format** identifier.

Table 6: Frame Format Types and Allowed Reserved Values

frame_format IdentifierValue Reserved Words	Description
CAN_STANDARD	standard frame format, 11-bit CAN identifiers
CAN_EXTENDED	extended frame format, 29-bit CAN identifiers

Examples:

Valid Format:
```
1. frame_format = CAN_EXTENDED;                // 29-bit CAN identifiers
```

Invalid Format:
```
1. frame_format = {CAN_STANDARD};              // Extra braces
```

4.2.7 erase

The **erase** identifier is used to indicate to the tester which range(s) of memory need to be sent in an erase request before a software download operation of the file occurs. The erase start address and length will be sent by the tester when downloading the file. The erase start address and length do not need to be matched to the start address and length of a physical flash sector and they do not need to align with the data section in the file. When present, the erase length can span over more than one flash sector, or can be within a single flash sector. However, to better support the ability of a tester to resume an aborted download (e.g., due to cable disconnect) it is strongly recommended to have a separate erase pair for each physical flash sector instead of combining flash sectors into a single address and length pair. The typical erase start address and length can automatically be derived from a Motorola S-record file or an Intel Hex file during generation of a VBF file, but there is no guarantee that this will align with acceptable flash erase segment boundaries for a given SBL. The erase start address and length are four bytes long with a range of 0x00000000 to 0xFFFFFFFF. If sw_part_type = SBL or GBL, the VBF file shall not contain the **erase** identifier. **erase** identifier values shall always be contained in two sets of braces whether there is one or more ranges present.

Examples:

Valid Format:

```
1.  erase =   {  { /*start address*/ 0x00004200, /*Length*/ 0x00003CBF }
              };

2.  erase =   {  { 0x00004200, 0x00003CBF },
                 { 0x00008010, 0x00002FF3 }
               };                          // Two blocks: 0x00004200-0x00007EBE, 0x00008010-0x0000B002
```

Invalid Format:

```
1.  erase =   { 0x00004200, 0x00003CBF };                   // Missing outer braces

2.  erase =   { { 0xFFFFFFFE, 0x00000002 }                  // Address Overflow
              };

3.  erase =   { 0x00004200, 0x00003CBF },                   // Missing outer braces
              { 0x00008010, 0x00002FF3 };

4.  erase =   { 0x00004200, 0x00003CBF,                     // Missing inner braces
                0x00008010, 0x00002FF3
              };
```

4.2.8 omit

The **omit** entry is optional and shall only be used when explicit approval is granted by the responsible authority appointed by the operating company. The **omit** entry indicates the range(s) of memory that is to be ignored by any standard VBF parser and is mandatory to be supported by all VBF parsers. All memory ranges included in the **omit** entry shall be manually deleted by the VBF parser from both the **erase** entry identifier (if present) in the header section as well as the data section. The effective net result of this deletion are that the address ranges within **omit** are not requested to be erased nor programmed by tools responsible for programming ECUs. Note that the presence of the **omit** entry shall have no effect on the calculation of the checksums within the data section of the VBF file. The omit start address and length are four bytes long with a range of 0x00000000 to 0xFFFFFFFF. The **omit** identifier values shall always be contained in two sets of braces, whether there is one or more ranges present.

When the **omit** entry is utilized, the values used for **<u>each</u>** pair of start address and length within the **omit** entry shall meet the following requirements. A VBF parser shall deem the file as invalid if any of these requirements are not satisfied.

- The memory area to omit shall align with at least one memory area from either the **erase** header field or the memory area defined by a single block within the data section.
- If the memory area to omit (based upon start address and length) overlaps with any byte of a memory area defined by a single start address / length pair with the **erase** header field, the values contained within start address and length for that memory area within the **erase** entry shall exactly match the values contained within the start address / length pair of the **omit** entry. This means each start address / length pair within an omit entry is only ever allowed to omit complete memory areas from the **erase** entry and can never omit partial memory areas or span multiple **erase** entry pairs. Note that this may sometimes require that a single large contiguous memory area is split into smaller contiguous memory areas within these entries.
- If the memory area to omit (based upon start address and length) overlaps with any byte of a memory area defined by a single block within the data section, the values contained within start address and length for that block within the data section shall exactly match the values contained within the start address and length pair of the **omit** entry. A block within the data section is defined as a group of four items (start address, length, data, checksum) as detailed within section 4.3. This means each start address / length pair within an omit entry is only ever allowed to omit complete memory blocks from the data section and can never omit partial memory blocks or span multiple memory blocks from the data section. Note that this may sometimes require that a single large contiguous memory block is split into smaller contiguous memory blocks.

Where approved by the responsible authority appointed by the operating company, specialty VBF parsers may ignore the **omit** identifier in order to fully erase and/or program all memory ranges within the file (i.e., parse the file as if the **omit** identifier were not present). An example use of **omit** is when the same VBF file is to be used by both module manufacturing and service for cases where all bytes in the data section are relevant for module manufacturing, but are not relevant for service. In this example, approved module manufacturing VBF tools would ignore **omit** entries while service and other standard VBF tools would process and act upon **omit** entries.

The following example demonstrates the end effect of a standard VBF parser deleting the memory ranges based upon an **omit** entry. Assume a VBF file with the following **erase** entry and data section (note that data and checksum fields are purposely excluded from data section to simplify the example):

```
    erase = { { 0x00000000, 0x00002000 },
              { 0x00008000, 0x00001000 }
            };
```

Data section	Block 1	Start address	0x00000000
		Length	0x000007FF
	Block 2	Start address	0x00008000
		Length	0x00001000
	Block 3	Start address	0x00100000
		Length	0x00004000

Assume this VBF file now has an **omit** entry as shown below:

```
omit =    { { 0x00008000, 0x00001000 } };
```

The net effect on the parsed erase address ranges and data section ranges as perceived by a standard VBF parser after deleting the **omit** address ranges would then be as follows:

```
erase =  { { 0x00000000, 0x00002000 } };
```

Data section	Block 1	Start address	0x00000000
		Length	0x000007FF
	Block 2	Start address	0x00100000
		Length	0x00004000

Examples:

Valid Format:

```
1.  omit =    { { 0x00000000, 0x00000700 }
              };

2.  omit =    { { 0x00000000, 0x00000400 },
              { 0x00000700, 0x00000800 }
               };                     // Two blocks: 0x00000000-0x000003FF, 0x00000700-0x00000EFF
```

Invalid Format:

```
1.  omit =    { 0x00004200, 0x00003CBF };                   // Missing outer braces

2.  omit =    { 0x00004200, 0x00003CBF },                   // Missing outer braces
              { 0x00008010, 0x00002FF3 };

3.  omit =    { 0x00004200, 0x00003CBF,                     // Missing inner braces
                0x00008010, 0x00002FF3
              };

4.  erase =  { { 0x00000000, 0x00002000 },
               { 0x00008000, 0x00001000 } };

    omit =    { { 0x00008500, 0x00000500 } };
    // Omit memory area overlaps with erase memory area, but only contains a subset.

5.  omit =    { { 0x00008000, 0x00002000 } };        // Omit memory area spans multiple data section blocks
```

Data section	Block 1	Start address	0x00008000
		Length	0x00001000
	Block 2	Start address	0x00009000
		Length	0x00001000

4.2.9 call

The **call** entry is mandatory in VBF files with **sw_part_type** identifier values equal to SBL or GBL and optional in VBF files with **sw_part_type** equal to TEST. For all other identifier values of **sw_part_type**, the **call** entry is not allowed. The **call** entry indicates the start address to a pre-defined executable function in ECU memory. The call identifier start address is four bytes long with a range of 0x00000000 to 0xFFFFFFFF.

Examples:

Valid Format:

1. sw_part_type = SBL;
   call = 0x00000100; // Start address for SBL

2. sw_part_type = TEST;
   call = 0x00000100; // Start address for GBL

Invalid Format:

1. sw_part_type = SBL;
   With no **call** identifier

2. sw_part_type = GBL;
   With no **call** identifier

3. sw_part_type = EXE; // or any other IdentifierValue but SBL or GBL or TEST
   call = 0x00000100;

4.2.10 file_checksum

A four-byte file checksum shall be provided in the header section with a range of 0x00000000 to 0xFFFFFFFF. The purpose of this checksum is to validate the integrity of the file itself prior to attempting a download. The checksum shall be a CRC32 of all the data contained in the data section of the file, including start address, length and checksum for all data blocks. This means that if the data bytes to download are compressed (i.e., data_format_identifier contains a non-zero value), then the file checksum is performed using the compressed data. The tester shall check this checksum against the file checksum it has calculated before the software download operation is performed with the VBF file.

The file checksum shall meet the following criteria:

- 32 bit CRC, polynomial = 0x04C11DB7
  (x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1)
- Initial value = 0xFFFFFFFF
- Bit reversed

For a fast CRC32 calculation a look-up table implementation is the preferred solution.
See Appendix B for "C-code" example.

Example:
file_checksum = 0x29058c73 ;

Note: This is the same CRC32 algorithm as used by "WinZip" (among many others).

4.2.11 Header section example #1

The following example is a VBF-file with **sw_part_type** = EXE, supporting standard frame format (11-bit CAN identifiers)

```
vbf_version = 3.0;
header {
               // This is a comment
               /* This is also a comment */
               // Short description of the VBF file
               description = { "Application SW for 5-cyl diesel engine",
               "Created 2011-11-10"
               };
               // Software part number
               sw_part_number = "YW4T-13B527-BC";
               // Software type
               sw_part_type = EXE;
               // Compression and Encryption Method
               data_format_identifier = 0x10
               // ECU address
               ecu_address = 0x720;
               // 11-bit CAN-identifers
               frame_format = CAN_STANDARD;
               // Erase information
               // start length
               erase = { { 0x00008000, 0x00007EF0},
                          { 0x00010000, 0x0002FE05}
                          };
               // File checksum
               file_checksum = 0x1A47EDA3;
} <binary data section follows>
```

Figure 4-1: Header section example #1

4.2.12 Header section example #2

The second example is a VBF-file with **sw_part_type** = SBL, supporting extended frame format (29-bit CAN-identifiers).

```
vbf_version = 3.0;
header {
               // This is a comment
               /* This is also a comment */
               // Short description of the VBF file
               description = {"SBL for DIM",
                              "Created 2011-11-10"
                                  };
               // Software part number
               sw_part_number = "YW4T-13B526-AE";
               // Software type
               sw_part_type = SBL;
               // Compression and Encryption Method
               data_format_identifier = 0x10
               // ECU address
               ecu_address = 0x51;
               // 29-bit CAN-identifiers
               frame_format = CAN_EXTENDED;
               // Start address for the SBL
               call = 0x00000100;
               // File checksum
               file_checksum = 0xA1DC5A45;
} <binary data section follows>
```

Figure 4-2: Header section example #2

4.3 Data section

After the header section ends, the data section immediately follows. The header section ends with a } character (ASCII 0x7D). The binary data section begins with the next byte immediately following this ASCII } character.

Table 7: Data Section

Version Section	VBF Version Number	Text	vbf_version=3.0;
Header section	Header	Text	n bytes, ASCII
Data section	Block 1	Start address	4 bytes, binary
		Length	4 bytes, binary
		Data	n bytes, binary
		Checksum	2 byte, binary
	Block 2	Start address	4 bytes, binary
		Length	4 bytes, binary
		Data	n bytes, binary
		Checksum	2 byte, binary
	:	:	:
	Block n	Start address	4 bytes, binary
		Length	4 bytes, binary
		Data	n bytes, binary
		Checksum	2 byte, binary

The data section may contain more than one block of data, and every block includes four parts.

- 4-byte start address, physical address in ECU memory, with a range of 0x00000000 to 0xFFFFFFFF.
- 4-byte length, number of data bytes in block (excluding Start address, Length and Checksum), with a range of 0x00000001 to 0xFFFFFFFF. If data compression and/or encryption is used (refer to 4.2.4), this length represents the compressed and/or encrypted length.
- Data
- 2-byte checksum for the data block (excluding Start address and Length) with a range of 0x0000 to 0xFFFF.

The Start address, Length, and Checksum fields shall always be stored with the most significant byte (msb) first.

4.3.1 Checksum calculation

The VBF file shall include a 2-byte checksum for every data block. The 2-byte checksum shall be calculated including all data bytes (excluding Start address, Length, and Checksum) in the data block. If data compression and/or encryption is used (refer to 4.2.4), the data block checksum shall be calculated using uncompressed and/or unencrypted data (i.e., data written to ECU memory). The test tool shall not use these embedded checksums to validate the integrity of the file before the software download operation is performed with the VBF file. However, the test tool shall use these checksums to validate the ECU's positive response for TransferExit (0x37) service request.

The checksum algorithm to be used shall be the CRC16-CITT:

- Polynomial = 0x1021 (x^16+x^12+x^5+1)
- Initial value = 0xFFFF

For a fast CRC16-CITT calculation a look-up table implementation is the preferred solution.
See Appendix A for "C-code" example.

5. Appendix A

5.1 CRC16-CITT C-code example

This example uses a look-up table with pre-calculated CRCs for fast calculation.

```
/* function declarations */
unsigned int CalcCRC(unsigned int size,unsigned char *data);

/* test data array */
unsigned char data[256] =
{           0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,
            0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,0x1d,0x1e,0x1f,
            0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2a,0x2b,0x2c,0x2d,0x2e,0x2f,
            0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3a,0x3b,0x3c,0x3d,0x3e,0x3f,
            0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x4b,0x4c,0x4d,0x4e,0x4f,
            0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0x5b,0x5c,0x5d,0x5e,0x5f,
            0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x6b,0x6c,0x6d,0x6e,0x6f,
            0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x7b,0x7c,0x7d,0x7e,0x7f,
            0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8a,0x8b,0x8c,0x8d,0x8e,0x8f,
            0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0x9b,0x9c,0x9d,0x9e,0x9f,
            0xa0,0xa1,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xab,0xac,0xad,0xae,0xaf,
            0xb0,0xb1,0xb2,0xb3,0xb4,0xb5,0xb6,0xb7,0xb8,0xb9,0xba,0xbb,0xbc,0xbd,0xbe,0xbf,
            0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xcb,0xcc,0xcd,0xce,0xcf,
            0xd0,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xdb,0xdc,0xdd,0xde,0xdf,
            0xe0,0xe1,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xeb,0xec,0xed,0xee,0xef,
            0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff
};

unsigned int crctab[256] =
{
            0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
            0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
            0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
            0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
            0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
            0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
            0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
            0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
            0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
            0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
            0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
            0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
            0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
            0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8D68, 0x9D49,
            0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
            0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
            0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
            0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
            0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
            0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
            0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
            0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
            0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
            0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
            0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
            0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
            0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
            0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
            0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
            0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
            0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
            0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0
};
```

```
/* driver */
void main(void)
{
            unsigned int crc;

            crc=CalcCRC(256,data);

            while(1); /* the result of the above calculation shall be: crc=0x3FBD */
}


/* Calculate CRC */
unsigned int CalcCRC(unsigned int size, unsigned char *data)
{
            unsigned int crc=0xffff; /* initial value */
            int tmp;
            int i;

            for(i=0;i<size;i++) {
                tmp=(crc>>8)^data[i];
                crc=(crc<<8)^crctab[tmp];
            }
            return crc;
}
```

6. Appendix B

6.1 CRC32 C-code example

This example uses a look-up table with pre-calculated CRCs for fast calculation.

```
/* function declarations */
void generate_crc32_table(void);
unsigned long calc_crc32(unsigned long size, unsigned char *data);

/* variables */
unsigned long crc32;
static unsigned long crc32_table[256];

/* test data array */
unsigned char data[256] =
{           0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,
            0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,0x1d,0x1e,0x1f,
            0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2a,0x2b,0x2c,0x2d,0x2e,0x2f,
            0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3a,0x3b,0x3c,0x3d,0x3e,0x3f,
            0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0x4a,0x4b,0x4c,0x4d,0x4e,0x4f,
            0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5a,0x5b,0x5c,0x5d,0x5e,0x5f,
            0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x6a,0x6b,0x6c,0x6d,0x6e,0x6f,
            0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7a,0x7b,0x7c,0x7d,0x7e,0x7f,
            0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8a,0x8b,0x8c,0x8d,0x8e,0x8f,
            0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9a,0x9b,0x9c,0x9d,0x9e,0x9f,
            0xa0,0xa1,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xab,0xac,0xad,0xae,0xaf,
            0xb0,0xb1,0xb2,0xb3,0xb4,0xb5,0xb6,0xb7,0xb8,0xb9,0xba,0xbb,0xbc,0xbd,0xbe,0xbf,
            0xc0,0xc1,0xc2,0xc3,0xc4,0xc5,0xc6,0xc7,0xc8,0xc9,0xca,0xcb,0xcc,0xcd,0xce,0xcf,
            0xd0,0xd1,0xd2,0xd3,0xd4,0xd5,0xd6,0xd7,0xd8,0xd9,0xda,0xdb,0xdc,0xdd,0xde,0xdf,
            0xe0,0xe1,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xeb,0xec,0xed,0xee,0xef,
            0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfd,0xfe,0xff
};


/* driver */
void main(void)
{
            unsigned long crc32;

            generate_crc32_table();
            crc32=calc_crc32(256,data);

            while(1); /* the result of the above calculation shall be: crc32=0x29058c73 */
}


/* calculate the crc32 value */
unsigned long calc_crc32(unsigned long size, unsigned char *data)
{

            unsigned long i, crc;

            crc=0xffffffff;         // initial value

            for(i=0;i<size;i++) {
                crc = ((crc>>8)&0x00FFFFFF)^crc32_table[(int)(crc^data[i])&0xff];
            }

            return ~crc;
}
```

```
#define CRC32_POLY 0xedb88320 // 0x04c11db7 bit reversed

/* generate the lock-up table */
void generate_crc32_table(void)

{
            int i, j;
            unsigned long crc;

            for(i=0;i<256;i++) {
                crc=i;
                for(j=8;j>0;j--) {
                    if(crc&1) {
                        crc=(crc>>1)^CRC32_POLY;
                    }
                    else{
                        crc>>=1;
                    }
                }
                crc32_table[i]=crc;
            }
}

// As an alternative to the "generate_crc32_table()" function the following table can be used

/*
static unsigned long crc32_tab[256] =
{           0x00000000L, 0x77073096L, 0xee0e612cL, 0x990951baL, 0x076dc419L, 0x706af48fL,
            0xe963a535L, 0x9e6495a3L, 0x0edb8832L, 0x79dcb8a4L, 0xe0d5e91eL, 0x97d2d988L,
            0x09b64c2bL, 0x7eb17cbdL, 0xe7b82d07L, 0x90bf1d91L, 0x1db71064L, 0x6ab020f2L,
            0xf3b97148L, 0x84be41deL, 0x1adad47dL, 0x6ddde4ebL, 0xf4d4b551L, 0x83d385c7L,
            0x136c9856L, 0x646ba8c0L, 0xfd62f97aL, 0x8a65c9ecL, 0x14015c4fL, 0x63066cd9L,
            0xfa0f3d63L, 0x8d080df5L, 0x3b6e20c8L, 0x4c69105eL, 0xd56041e4L, 0xa2677172L,
            0x3c03e4d1L, 0x4b04d447L, 0xd20d85fdL, 0xa50ab56bL, 0x35b5a8faL, 0x42b2986cL,
            0xdbbbc9d6L, 0xacbcf940L, 0x32d86ce3L, 0x45df5c75L, 0xdcd60dcfL, 0xabd13d59L,
            0x26d930acL, 0x51de003aL, 0xc8d75180L, 0xbfd06116L, 0x21b4f4b5L, 0x56b3c423L,
            0xcfba9599L, 0xb8bda50fL, 0x2802b89eL, 0x5f058808L, 0xc60cd9b2L, 0xb10be924L,
            0x2f6f7c87L, 0x58684c11L, 0xc1611dabL, 0xb6662d3dL, 0x76dc4190L, 0x01db7106L,
            0x98d220bcL, 0xefd5102aL, 0x71b18589L, 0x06b6b51fL, 0x9fbfe4a5L, 0xe8b8d433L,
            0x7807c9a2L, 0x0f00f934L, 0x9609a88eL, 0xe10e9818L, 0x7f6a0dbbL, 0x086d3d2dL,
            0x91646c97L, 0xe6635c01L, 0x6b6b51f4L, 0x1c6c6162L, 0x856530d8L, 0xf262004eL,
            0x6c0695edL, 0x1b01a57bL, 0x8208f4c1L, 0xf50fc457L, 0x65b0d9c6L, 0x12b7e950L,
            0x8bbeb8eaL, 0xfcb9887cL, 0x62dd1ddfL, 0x15da2d49L, 0x8cd37cf3L, 0xfbd44c65L,
            0x4db26158L, 0x3ab551ceL, 0xa3bc0074L, 0xd4bb30e2L, 0x4adfa541L, 0x3dd895d7L,
            0xa4d1c46dL, 0xd3d6f4fbL, 0x4369e96aL, 0x346ed9fcL, 0xad678846L, 0xda60b8d0L,
            0x44042d73L, 0x33031de5L, 0xaa0a4c5fL, 0xdd0d7cc9L, 0x5005713cL, 0x270241aaL,
            0xbe0b1010L, 0xc90c2086L, 0x5768b525L, 0x206f85b3L, 0xb966d409L, 0xce61e49fL,
            0x5edef90eL, 0x29d9c998L, 0xb0d09822L, 0xc7d7a8b4L, 0x59b33d17L, 0x2eb40d81L,
            0xb7bd5c3bL, 0xc0ba6cadL, 0xedb88320L, 0x9abfb3b6L, 0x03b6e20cL, 0x74b1d29aL,
            0xead54739L, 0x9dd277afL, 0x04db2615L, 0x73dc1683L, 0xe3630b12L, 0x94643b84L,
            0x0d6d6a3eL, 0x7a6a5aa8L, 0xe40ecf0bL, 0x9309ff9dL, 0x0a00ae27L, 0x7d079eb1L,
            0xf00f9344L, 0x8708a3d2L, 0x1e01f268L, 0x6906c2feL, 0xf762575dL, 0x806567cbL,
            0x196c3671L, 0x6e6b06e7L, 0xfed41b76L, 0x89d32be0L, 0x10da7a5aL, 0x67dd4accL,
            0xf9b9df6fL, 0x8ebeeff9L, 0x17b7be43L, 0x60b08ed5L, 0xd6d6a3e8L, 0xa1d1937eL,
            0x38d8c2c4L, 0x4fdff252L, 0xd1bb67f1L, 0xa6bc5767L, 0x3fb506ddL, 0x48b2364bL,
            0xd80d2bdaL, 0xaf0a1b4cL, 0x36034af6L, 0x41047a60L, 0xdf60efc3L, 0xa867df55L,
            0x316e8eefL, 0x4669be79L, 0xcb61b38cL, 0xbc66831aL, 0x256fd2a0L, 0x5268e236L,
            0xcc0c7795L, 0xbb0b4703L, 0x220216b9L, 0x5505262fL, 0xc5ba3bbeL, 0xb2bd0b28L,
            0x2bb45a92L, 0x5cb36a04L, 0xc2d7ffa7L, 0xb5d0cf31L, 0x2cd99e8bL, 0x5bdeae1dL,
            0x9b64c2b0L, 0xec63f226L, 0x756aa39cL, 0x026d930aL, 0x9c0906a9L, 0xeb0e363fL,
            0x72076785L, 0x05005713L, 0x95bf4a82L, 0xe2b87a14L, 0x7bb12baeL, 0x0cb61b38L,
            0x92d28e9bL, 0xe5d5be0dL, 0x7cdcefb7L, 0x0bdbdf21L, 0x86d3d2d4L, 0xf1d4e242L,
            0x68ddb3f8L, 0x1fda836eL, 0x81be16cdL, 0xf6b9265bL, 0x6fb077e1L, 0x18b74777L,
            0x88085ae6L, 0xff0f6a70L, 0x66063bcaL, 0x11010b5cL, 0x8f659effL, 0xf862ae69L,
            0x616bffd3L, 0x166ccf45L, 0xa00ae278L, 0xd70dd2eeL, 0x4e048354L, 0x3903b3c2L,
            0xa7672661L, 0xd06016f7L, 0x4969474dL, 0x3e6e77dbL, 0xaed16a4aL, 0xd9d65adcL,
            0x40df0b66L, 0x37d83bf0L, 0xa9bcae53L, 0xdebb9ec5L, 0x47b2cf7fL, 0x30b5ffe9L,
            0xbdbdf21cL, 0xcabac28aL, 0x53b39330L, 0x24b4a3a6L, 0xbad03605L, 0xcdd70693L,
            0x54de5729L, 0x23d967bfL, 0xb3667a2eL, 0xc4614ab8L, 0x5d681b02L, 0x2a6f2b94L,
            0xb40bbe37L, 0xc30c8ea1L, 0x5a05df1bL, 0x2d02ef8dL
};
*/
```