Federal
Public Service
FINANCE

MD 29/04/2024

REGISTERED CASH REGISTER SYSTEM – TECHNICAL SPECIFICATIONS –

DETAILED DESCRIPTION OF THE FUNCTIONING OF THE FISCAL DATA MODULE AND THE COMMUNICATION WITH THE CASH REGISTER SYSTEM AND THE CLOUD SERVICE OF THE FPS FINANCE

VERSION 1.4 – 17/04/2026

CHANGELOG

Version	Date	Description
1.0	05/08/2024	Original text
1.1	06/09/2024	**Clarification**: - Physical connection POS - FDM
1.2	20/09/2024	**Correction**: - Full replacement table in chapter 3, section 3.3.
1.3	20/05/2025	**Clarification:** - Chapter 4: buffer should be accessible for reading/copying via backoffice
1.4	07/04/2026	**Addition** - 2.4 Digital signature: moved here from detailed description POS **Correction** - 6.1 verificationUrl: to higher level

INTRODUCTION ... 1
USED ABBREVIATIONS ... 1
CHAPTER 1 – TECHNICAL REQUIREMENTS FOR THE FISCAL DATA MODULE ... 2
1.1. GENERAL REQUIREMENTS ... 2
1.1.1. COUNTERS ... 2
1.1.2. VAT CALCULATION AND TAX BASE ... 2
1.1.3. BUFFERING AND ADJUSTABILITY OF SETTINGS ... 3
1.1.4. CE MARKING ... 3
1.2. HARDWARE REQUIREMENTS ... 3
1.2.1 STORAGE ... 3
1.3. USER INTERFACE ... 3
1.4. Firmware ... 4
CHAPTER 2 – COMMUNICATION WITH THE CONNECTED POS SYSTEM ... 5
2.1. PHYSICAL CONNECTION ... 5
2.2 GrapHQL service ... 5
2.2.1. GENERAL ... 6
2.2.2. MANDATORY MUTATIONS ... 6
2.2.3. CONTENT OF THE COMMUNICATION POS ➔ FDM ... 7
2.2.4. CONTENT OF THE COMMUNICATION FDM ➔ POS ... 7
2.2.5. FDM ➔ POS errorhandling ... 7
2.3. GRAPHQL SCHEMA FOR COMMUNICATION between POS and FDM ... 8
2.4. DIGITAL SIGNATURE ... 8
2.4.1. Canonical JSON for the Reproducible Hash ... 8
2.4.2. The 'enriched JSON' whose content is digitally signed ... 8
2.4.3. Certificate to be used for the signature ... 9
CHAPTER 3 – VERIFICATION URL – QR CODE ... 10
3.1. FIXED LENGTH ... 10
3.2. Base-62 character set To use ... 11
3.3. Bit stream ... 11
3.4. XOR mask ... 12
3.5. digitalSignature ... 12
3.6. SIGNATURE BIT COUNTER AND SIGNATURE BITS ... 12
3.7. fdmId ... 13
3.8. totalCounter ... 13
3.9. Padding ... 13
3.10. REFERENCE VALUES ... 13
3.11. Encoder ... 14
3.12. Decoder ... 16
CHAPTER 4 – TEMPORARY STORAGE OF DATA ... 21
CHAPTER 5 – COMMUNICATION FDM ↔ FPSFIN CLOUDSERVICE ... 22

5.1. CONNECTION AND SECURITY 22
5.1.1. CONNECTION 22
5.1..2. SECURITY 22
5.2. FIRST TIME EVER 22
5.3. API AND PROTOCOL FOR COMMUNICATION WITH FPS FINANCe CLOUD SERVICE 23
CHAPTER 6 - ONLINE FORWARDING OF TRANSACTION DATA 24
6.1. JSON MESSAGE TO FPS FINANCE 24

INTRODUCTION

The detailed descriptions provide a more technical and detailed view of the technical provisions of the Ministerial Decree of 29/04/2024 regarding the technical aspects and certification of the Fiscal Data Module (FDM).

This document is the detailed technical description concerning the functioning of the FDM, the communication with a certified cash register system, and the communication with the cloud service of the FPS Finance.

For the functioning of a certified cash register system, refer to the relevant detailed description.

This document may undergo minor changes due to possible regulatory decisions or to correct any errors.

Additional information can be obtained from the competent service, NCI division RCRS via secr.gksce@minfin.fed.be.

USED ABBREVIATIONS

AES	Advanced Encryption Standard
CBC	Cipher Block Chain
ECDSA	Elliptic Curve Digital Signature Algorithm
FDM	Fiscal Data Module
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON web token
LAN	Local Area Network
(m)TLS	(mutual) Transport Layer Security
NOP	No Operation
NTP	Network Time Protocol
PEM	Privacy Enhanced Mail
PKCS	Public Key Cryptography Standards
POS	Point Of Sale
QR code	Quick Response code
RCRS	Registered Cash Register System
RTC	Real Time Clock
SECP256r1	Specific Elliptic curve based on prime fields
TCP/IP	Transmission Control Protocol/Internet Protocol
UTC	Coordinated Universal Time
UTF	Unicode Transformation Format

CHAPTER 1 – TECHNICAL REQUIREMENTS FOR THE FISCAL DATA MODULE

Reference MD: Title II, Chapter 2

1.1. GENERAL REQUIREMENTS

In principle, an FDM must not contain any functionalities other than those described in the MD. The same applies to communication between POS and FDM and between FDM and POS.

If a manufacturer wishes to add additional functionality and/or communication, it must:

- Contribute to the proper (and consequently improved) operation of the device;
- Be extensively documented in the certification application.

Any additional functionality and/or communication must always be subordinate to the functionalities and communication both with the POS and the cloud service of the FPS Finance.

The receipt and transmission of messages and their handling may never be interrupted for this additional functionality.

1.1.1. COUNTERS

The counters mentioned in Article 64 start again at 1 if they have reached their maximum value of 999999999.

1.1.2. VAT CALCULATION AND TAX BASE

Example of the calculation of the tax base and the VAT amount (as described in Article 66):

quantity	description	amount	VAT label
2	Food	41,50	B
2	Drinks	10,00	A
1	Food	12,75	B
1	Take away	13,00	C
5	Caution	2,00	X
1	Tabac	15,00	D
		94,25	

VAT code	total	non rounded VAT amount	rounded VAT amount	tax base
A	10,00	1,73553719	1,74	8,26
B	54,25	5,8125	5,81	48,44
C	13,00	0,735849057	0,74	12,26
D	15,00	0	0,00	15,00
X	2,00	0	0,00	2,00
	94,25		**8,28**	

Important note: VAT rates may change. Through the FPSFIN cloud service, the FDM will always receive the new rates and their validity period well in advance. If a bookingPeriod runs over two calendar days (e.g. from 30 June 2024 16:00 to 1 July 2024 12:00), the FDM will always apply the VAT rates that are/were in force on the bookingDate provided by the POS. In the above example, if the VAT rate of code B were to change to 9% on 1 July 2024, the FDM should still perform the calculation using the rate in force on 30 June 2024 (bookingDate).

1.1.3. BUFFERING AND ADJUSTABILITY OF SETTINGS

Reference is made to the communication protocol in Chapter 2.

1.1.4. CE MARKING

The manufacturer must ensure that their products comply with the requirements of all applicable legislation. They perform a conformity assessment and involve a notified body if required by EU legislation.

Furthermore, the manufacturer maintains a technical file of their products and shows it to the market authorities of the country where the products are traded upon request. In a declaration of conformity, the manufacturer declares that their products comply with all applicable legislation. Finally, the manufacturer affixes the CE mark to the product.

More info at: Frequently Asked Questions about CE Marking | FPS Economy

1.2. HARDWARE REQUIREMENTS

Reference MD: Title II, Chapter 2, section 2.

1.2.1 STORAGE

The storage mentioned in Article 76 and Article 81 are not the same. There must be at least 2 GB of capacity available to store (buffer) transaction data (enriched JSON). This is entirely separate from the internal memory, as described in Article 76.

1.3. USER INTERFACE

Reference MD: Title II, Chapter 2, section 3.

A minimum number of health indicators must be present on the device itself. These are described in Article 83 of the MD. The manufacturer can choose between a display or LEDs.

A detailed user manual/troubleshooting sheet must be available for the end user, installer, and the competent service at the FPS Finance.

Access to the FDM back office must be sufficiently protected. The producer documents this extensively in their certification application.

Through this user interface, it must be possible to fully configure the network connection with the POS and with the FPS Finance cloud service.

Article 85 of the MD allows the producer to choose between:

- User interface on the device itself (display);

- Through a connected device (laptop, tablet, …);
- Through the connected POS (screen);
- Or a combination of the above.

The user and/or installation manual describes the procedures in detail. This manual is handed over during the certification procedure.

1.4. FIRMWARE

The firmware must always carry a version number. The certificate of conformity always mentions this version number.

The firmware must be capable of supporting **all** functionalities required in the MD and in this detailed description.

Consequently, the firmware will:

- check the correctness (according to the JSON and GraphQL schema) of the received messages from the POS; incorrect messages will not be processed (digital signature, calculations, incrementing counters, …). The POS will be notified via error codes in the response;
- for correct messages, perform the necessary calculations, increment counters, conversions, and place a digital signature;
- store the content of the signed messages and prepare them for transmission to the FPS Finance cloud service;
- transmit the transaction data to the FPS Finance cloud service according to the procedures described further;
- be able to receive messages from the FPS Finance cloud service, interpret them correctly, and act accordingly, as provided in the further described protocol;

FDM producers are free to incorporate more functionalities. However, these must not impair the proper fiscal operation of the device and should contribute to the good operation/user-friendliness of the device. They must be described in detail in the accompanying documentation.

The firmware may receive necessary or mandated upgrades/patches. After approval by the competent service at FPS Finance, the installation on the devices is carried out by the producer or their delegate, in a secure manner. This is described in detail in the documentation provided with the certification application and forms part of the certification process.

CHAPTER 2 – COMMUNICATION WITH THE CONNECTED POS SYSTEM

2.1. PHYSICAL CONNECTION

The POS and FDM are connected to each other via a cable (LAN) or wirelessly (WiFi). Configuration of this connection is done in the back office of both the POS and FDM.

IMPORTANT: In theory, the wireless connection can also be established via the internet. If the internet connection fails, the POS-FDM connection will also automatically be lost, causing the POS to stop registering transactions.

The FDM manufacturer provides sufficient options to ensure a secure connection between the POS and FDM. Typical examples are: HTTP with token, HTTPS with TLS using self-signed certificates, HTTPS with mTLS using a self-signed certificate based on the intermediate certificate of the FDM manufacturer.

The final responsibility for correct and secure installation and configuration lies with the distributor/installer.

The POS and FDM use the TCP/IP protocol to exchange their messages. These messages are further described below. Other messages are only allowed as long as they do not affect the mandatory features and contribute to the proper functioning of the system. The FDM makes its GraphQL service available to the POS via HTTP(S).

The FDM is connected to the FPSFIN cloud API via the internet and regularly receives instructions through this connection. In certain cases, as described in the relevant detailed documentation, such instructions may result in displaying a message on the POS and/or manual intervention via the POS.

The FDM may not be connected with third party applications. Access may only be granted to the POS applications, the FPSFIN applications and the application of the FDM manufacturer.

2.2 GRAPHQL SERVICE

Message exchange between POS and FDM takes place via the GraphQL service of the FDM.

The GraphQL standard of October 27, 2021, is fully followed.

More information can be found at: Release October 2021 · graphql/graphql-spec · GitHub.

Requests are sent by the POS to the GraphQL service of the FDM. JSON data is used as the payload for sending event data. This payload is sent to the path **"/graphql"** on the FDM with the HTTP verb "POST" and Content-Type: application/json.

All JSON objects follow the RFC 8259 standard.

String fields **never** contain leading or trailing whitespace.

The sent data includes, among others:

- transaction data (events P and N);
- financial data (event F);
- social data (event S);
- copies (event C);
- training events (event T);
- invoices (event I);
- reports (event R).

WWW.FIN.BELGIUM.BE
FEDERAL PUBLIC SERVICE FINANCE
.be

These objects and their conditions are further described in the complete GraphQL schema.

2.2.1. GENERAL

The following events were determined in the ministerial decree:

- NORMAL (label N);
- PRO FORMA (label P);
- TRAINING (label T);
- COPY (label C);
- FINANCIAL (label F);
- SOCIAL (label S);
- INVOICE (label I);
- REPORT (label R).

All these events are sent to the FDM to be digitally signed as a security measure against data manipulation. Various mutations are made available by the GraphQL service of the FDM for this purpose. These mutations can be used to correctly populate the JSON structure of the registrations on the FDM. The JSON structure of the FDM can contain any type of event; through the various mutations, the FDM accepts only those fields and objects that are allowed for each type of event.

2.2.2. MANDATORY MUTATIONS

The following mutations are defined:

<u>For the event S:</u>

- signWorkIn
- signWorkOut

<u>For the event I:</u>

- signInvoice

<u>For the event N:</u>

- signSale
- signSaleRefund

<u>For the event P:</u>

- signCostCenterChange
- signOrder
- signPreBill

<u>For the event F:</u>

- signMoneyInOut
- signDrawerOpen
- signPaymentCorrection

<u>For the event R:</u>

- signReportTurnoverX
- signReportTurnoverZ

- signReportUserX
- signReportUserZ

For the event C:

- signCopy

2.2.3. CONTENT OF THE COMMUNICATION POS ➔ FDM

We refer to the content of the DETAILED DESCRIPTION POS, published on our website www.geregistreerdkassasystqeem.be/nl/gks-2-0 or www.systemedecaisseenregistreuse.be/fr/sce-2-2 .

Communication rules

To ensure the correct handling of the mutations and uniquely distinguish them, the following key fields are defined:

- **posId**
- **posDateTime**
- **terminalId**
- **eventLabel**
- **posFiscalTicketNo**

When a POS sends a mutation to the FDM with these 5 key fields identical to those sent in a previous message, the FDM must verify whether this is a mutation to which a response has already been given (including updating the relevant counters on the FDM).

The following situations can occur:

1. The combination of key fields is different: the FDM handles this mutation in the normal way.
2. The content of the mutation is completely identical: the FDM handles this mutation as a "duplicate" submission and sends back the same data in the response as in the original mutation; the FDM does not update its counters; this duplicate transaction is not recorded on the FDM.
3. The combination of key fields is identical, but the other values in the mutation are different from those in the previously processed mutation: the FDM refuses to process this mutation and sends an error code in its response.

2.2.4. CONTENT OF THE COMMUNICATION FDM ➔ POS

We refer to the content of the DETAILED DESCRIPTION POS, published on our website www.geregistreerdkassasystqeem.be/nl/gks-2-0 or www.systemedecaisseenregistreuse.be/fr/sce-2-2 .

2.2.5. FDM ➔ POS ERRORHANDLING

We refer to the content of the DETAILED DESCRIPTION POS, published on our website www.geregistreerdkassasystqeem.be/nl/gks-2-0 or www.systemedecaisseenregistreuse.be/fr/sce-2-2 .

General remark about error handling

Certain errors and warnings are shown to the user via the POS user interface. In addition to the provisions above, the FDM manufacturer provides the necessary additional information for the user so that they can act appropriately to resolve the error.

This is described in detail in the documentation accompanying the certification application.

2.3. GRAPHQL SCHEMA FOR COMMUNICATION BETWEEN POS AND FDM

We refer to the content of the DETAILED DESCRIPTION POS, published on our website www.geregistreerdkassasystqeem.be/nl/gks-2-0 or www.systemedecaisseenregistreuse.be/fr/sce-2-2 .

2.4. DIGITAL SIGNATURE

The signature is applied to the “enriched JSON” after the JSON re-encoding has been performed. This ensures that the signature can be reproduced consistently.

2.4.1. Canonical JSON for the Reproducible Hash

The flexibility that JSON offers allows the same information to be encoded in different ways. To always obtain the same signature for the same source data, a number of rules are applied. These rules lead to a canonical JSON.

The following rules should be strictly applied by the FDM to the fields from the JSON mentioned further in section 2.4.2.

- **Numbers** may only contain the characters 0 to 9, the minus sign (-), and a decimal point. This means, among other things, that values in scientific notation must be re-encoded to their most basic form. Trailing zeros after a decimal point are not allowed.
- All **white spaces**, as described in the JSON specifications, are removed, and no white space is added.
- Characters with a **code point > U+001F and < U+007F** are **never** escaped, with the exception of code points **U+0022** (double quotation mark) and **U+005C** (backslash), which must always be escaped according to the JSON specifications. These two must be escaped in the shortest possible notation (respectively \" and \\).
- Characters with a **code point < U+0020 or > U+007E** are **always** escaped and always in their shortest possible notation. Backspace, form feed, line feed, carriage return, and horizontal tab have such short notations (respectively \b, \f, \n, \r, and \t).
- When escaping a code point as one or more sets of 4 hexadecimal characters, the hexadecimal characters are always expressed in **uppercase** (for example: \uAAFF).
- The JSON decoder must adhere to the **strict** interpretation of the JSON specifications. For example: the literals true, false, and null are always in lowercase and without quotation marks, the names of the name/value pairs in the objects must always be enclosed in quotation marks, etc.
- The **name/value pairs** of each object in JSON are **sorted** in ascending order based on the numerical value of the code points in the name. It is assumed, as a best practice, that no duplicate names are used, as no JSON standard exists in this regard.

2.4.2. The 'enriched JSON' whose content is digitally signed

The 'enriched' JSON object (**enrichedEventData**) consists of the values and objects below, depending on whether they were used in the message or not.

<u>Optional</u> fields that contain a null value or <u>optional</u> array’s that are empty, should not be included in the object and therefore not included in the signature as well.

```
{
 language
 posId
 vatNo
 estNo
 terminalId
 deviceId
 posDateTime
 posFiscalTicketNo
 ticketMedium
 eventOperation
 copyOfEvent
 employeeId
 customerVatNo
 invoiceNo
 fdmRefs
 bookingPeriodId
 bookingDate
 costCenter
 transfer
 transaction
 vatCalc
 financials
 drawer
 reportNo
 reportBookingDate
 posDevices
 fdmDevices
 turnover
 users
 fdmSwVersion
 posSwVersion
 bufferCapacityUsed
 fdmId
 fdmDateTime
 eventLabel
 eventCounter
 totalCounter
}
```

The value copyOfEvent will be determined and added by the FDM

copyOfEvent (enum, mandatory)

> The eventLabel as detected by the FDM upon receival of the message from the POS. Values to be used as mentioned in the enum EventLabel (see above).

2.4.3. Certificate to be used for the signature

The signature must be placed using the hardware certificate, the details of which are described in detailed description of the operation of and communication between the fiscal data module and the FPS cloud service.

CHAPTER 3 – VERIFICATION URL – QR CODE

The POS must print a QR code on its VAT receipt. The FDM generates the URL for this purpose. This URL consists of two main parts:

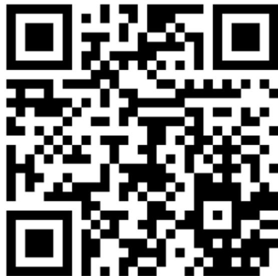
- The fixed prefix: https://www.gs2.be/
- The encoded value.

The prefix is fixed and determined by the FPS Finance.

The encoded value is fully calculated by the FDM and is unique for each event N. This value contains:

- An indication of the number of bits of the digitalSignature included in the calculation;
- The bits of the digitalSignature (with a maximum of 5 bytes);
- The fdmId;
- The total counter of the events;
- None or several padding bits.

Example:



Prefix: https://www.gs2.be/

encoded value: viXnmc1vvqGaMAS8MJV

3.1. FIXED LENGTH

The encoded value is **always** 19 base-62 characters long.

Since base-62 characters can be five or six bits, padding may be necessary. Conversely, it is also possible that the encoded value would exceed 19 characters. In such a case, the amount of data in the encoded value must be reduced.

By default, the encoder will include 40 bits of the digitalSignature; if necessary, this can be reduced to 36, 32, or 28 bits. The encoder must always include the maximum possible number of bits of the digitalSignature..

3.2. BASE-62 CHARACTER SET TO USE

Decimal	Binary	Base62
0	00000	0
1	11111	1
2	000010	2
3	000011	3
4	000100	4
5	000101	5
6	000110	6
7	000111	7
8	001000	8
9	001001	9
10	001010	A
11	001011	B
12	001100	C
13	001101	D
14	001110	E
15	001111	F

Decimal	Binary	Base62
16	010000	G
17	010001	H
18	010010	I
19	010011	J
20	010100	K
21	010101	L
22	010110	M
23	010111	N
24	011000	O
25	011001	P
26	011010	Q
27	011011	R
28	011100	S
29	011101	T
30	011110	U
31	011111	V

Decimal	Binary	Base62
32	100000	W
33	100001	X
34	100010	Y
35	100011	Z
36	100100	a
37	100101	b
38	100110	c
39	100111	d
40	101000	e
41	101001	f
42	101010	g
43	101011	h
44	101100	i
45	101101	j
46	101110	k
47	101111	l

Decimal	Binary	Base62
48	110000	m
49	110001	n
50	110010	o
51	110011	p
52	110100	q
53	110101	r
54	110110	s
55	110111	t
56	111000	u
57	111001	v
58	111010	w
59	111011	x
60	111100	y
61	111101	z

3.3. BIT STREAM

Both the encoder and decoder routines use a 64-bit buffer that acts as a bit stream. Bits are pushed into the stream from least significant to most significant and processed by the stream from most significant to least significant..

The longest value that needs to be pushed into the stream is 41 bits long. The bitstream processing starts only after each field has been pushed into it (to avoid overflow). The processed bits are immediately removed from the stream.

A counter continuously tracks how many bits have been pushed into the stream. The number of processed bits decreases the counter each time.

As soon as the five oldest bits represent 00000 or 11111, they are processed as a five-bit unit; all other values are processed as six-bit units.

he encoder pushes the bit representation of the original data into the bit stream and uses the table from section 3.3. to determine the correct base-62 character for the bits it processes.

The decoder uses the base-62 character to determine how many bits and in what pattern to push them into the stream, thereby reconstructing the originally encoded data from the bits it processes.

3.4. XOR MASK

To achieve a good distribution of the encoded values, an XOR mask is used on the values of fdmId and totalCounter.

Without this mask, the recurring values of fdmId and the increasing totalCounter could form a recognizable pattern in the encoded values.

The mask is generated based on the first two bytes of the digitalSignature (hereafter referred to as B1 and B2).

```
set mask equal to B2
shift the bits in the mask to the left by 8 positions
add B1 to mask
shift the bits in the mask to the left by 8 positions
add the inverse of B1 to mask
shift the bits in the mask to the left by 8 positions
add the inverse of B2 to mask
shift the bits in the mask to the left by 8 positions
add B1 to mask
shift the bits in the mask to the left by 8 positions
add B2 to mask
```

Of the resulting 48-bit mask, the least significant 41 bits are used as a mask for the fdmId, and the inverse of this 48-bit mask uses the least significant 30 bits as a mask for the totalCounter..

3.5. DIGITALSIGNATURE

Up to five bytes of the digitalSignature are included in the encoded value. Bytes one, two, and three of the digitalSignature are always included. Bytes four and five can be partially or entirely omitted if the encoded value would exceed 19 base-62 characters. They are included or omitted as half bytes, based on the sequence 40, 36, 32, 28 as described below.

B1	B2	B3	B4		B5	
Always	Always	Always	Always (28)	>= 32	>= 36	= 40
0x00 – 0xFF	0x00 – 0xFF	0x00 – 0xFF	0x0 – 0xF	0x0 – 0xF	0x0 – 0xF	0x0 – 0xF

3.6. SIGNATURE BIT COUNTER AND SIGNATURE BITS

The signature bit counter is the first field pushed into the bit stream. This counter contains the number of bits that the digital signature consists of. This field always contains two bits.

1	1	40 bits
1	0	36 bits
0	1	32 bits
0	0	28 bits

The signature bit counter field is immediately followed by the corresponding number of signature bits (second field).

3.7. FDMID

The third field that must be pushed into the bit stream is the fdmId. This is a string consisting of three characters from the range A to Z, followed by eight digits from the range 0 to 9.

The three characters are converted to integer values (c1, c2, c3) from the range 0 to 25, where A = 0, B = 1, C = 2, and so on.

The eight digits from the string are transformed into an integer value (ms). These four values are then combined into one 41-bit integer value using the following formula:

v = ((((((ms * 26) + c3) * 26) + c2) * 26) + c1)

Before this value is pushed into the bit stream, the 41-bit XOR mask is applied.

3.8. TOTALCOUNTER

The **fourth field** pushed into the bit stream is the totalCounter. This is a 30-bit integer value. Before this value is pushed into the bit stream, the 41-bit XOR mask is applied.

3.9. PADDING

The padding to be applied serves two main purposes:

- It ensures the proper functioning of the bit stream and that all bits are encoded as base-62 values;
- It ensures that the final result is 19 base-62 characters long..

Padding bits always have the value 0. Non-zero padding is not allowed, and encoded values containing such values must be considered invalid.

3.10. REFERENCE VALUES

The following values can be used as a reference to verify the encoding and decoding implementations.

FDM serial	TotalCounter	Original signature bytes (hex)					Encoded	Signature bits
FDM01000001	1	9B	21	C7	09	BF	viXnmc1vvqGaMAS8MJV	40
FDM01000001	2	26	8D	D5	E5	F4	oQDrUNqaEVtV0cdZTbm	40
FDM01000001	3	B4	8B	E6	23	67	xIBvYDdswxqDH0rYqjt	40
FDM01000001	999999997	F8	4B	C9	FF	27	1XBoVy1c8b0cu0MZd98	36
FDM01000001	999999998	20	26	5B	00	3C	Y0JBO0CDvAB1ergy5YS	36
FDM01000001	999999999	75	E1	F9	FD	97	tNX1FxBt2UWgjADUqFX	40
AAA01000001	1	42	A6	98	AD	4A	qAccArA8jNY1mE3fhrO	40
AAA01000001	999999999	B1	5E	58	9A	4C	x5UM9fCMYqUX1s9n8TU	40
AAA99999999	1	5B	4C	86	12	4C	rjCXX9CuSrM70KSJAIo	40
AAA99999999	999999999	F7	40	CD	C4	62	1T0Pk8r1Fg2K60UrWe0	36
ZZZ01000001	1	96	E1	D0	9B	C5	vRXq915oB1WvMkNuMaV	40
ZZZ01000001	999999999	74	C6	C6	CC	49	tJ6nin9EyxIEyISk8MC	40
ZZZ99999999	1	45	94	D8	4D	B1	qMKs4snxi0TE9h6bBfg	40
ZZZ99999999	999999999	0D	F3	38	D8	B5	mtpEDYrIkG9rGPgqdVc	40
XYZ98123456	33751	99	F2	2F	B2	CB	vdoBxBBXFSOzRorvSkq	40
XYZ98123456	33766	D4	A0	48	B4	45	zIWIBH5eL2nqnMhGL5o	40
XYZ98123456	33788	35	13	34	9A	01	pKJD9e0gL2q4UOs4qaG	40

3.11. ENCODER

This implementation assumes that all inputs are valid values.

```
public static byte[] Encode(string fdmSerial, long totalCounter, byte[] digitalSignature)
{
	// State machine states:
	// 0. initialize
	// 1. push signature bitcount modifier
	// 2. push signature bits, apply bitcount modifier
	// 3. push FDM manufacturer, model, and serial number
	// 4. push TotalCounter
	// 5. flush the stream by adding padding if need be
	// 6. verify length of the output (return the result)

	byte[] bytes = new byte[19];
	long bitStream = 0;
	long mask = digitalSignature[1];
	mask = (mask << 8) + digitalSignature[0];
	mask = (mask << 8) + ~digitalSignature[0];
	mask = (mask << 8) + ~digitalSignature[1];
	mask = (mask << 8) + digitalSignature[0];
	mask = (mask << 8) + digitalSignature[1];
	for (int state = 0, signatureBitCountModifier = 4, byteCount = 0, streamLen = 0; state < 7 && signatureBitCountModifier > 0; state++)
	{
		switch (state)
		{
			case 0:
				bitStream = 0;
				streamLen = 0;
				byteCount = 0;
				break;
```

```
                case 1:
                        bitStream = (bitStream << 2) | (byte)(signatureBitCountModifier - 1);
                        streamLen += 2;
                        break;

                case 2:
                        bitStream = (bitStream << 8) | digitalSignature[0];
                        bitStream = (bitStream << 8) | digitalSignature[1];
                        bitStream = (bitStream << 8) | digitalSignature[2];
                        bitStream = (bitStream << 8) | digitalSignature[3];
                        bitStream = (bitStream << 8) | digitalSignature[4];
                        streamLen += 40;

                        int undo = (4 * (4 - signatureBitCountModifier));
                        bitStream = (bitStream >> undo);
                        streamLen -= undo;
                        break;

                case 3:
                        int c1 = fdmSerial[0] - 65;
                        int c2 = fdmSerial[1] - 65;
                        int c3 = fdmSerial[2] - 65;
                        long ms = int.Parse(fdmSerial.Substring(3, 8));
                        long mask41 = mask & (((long)1 << 41) - 1);
                        long v = ((((((ms * 26) + c3) * 26) + c2) * 26) + c1) ^ mask41;
                        bitStream = (bitStream << 41) | v;
                        streamLen += 41;
                        break;

                case 4:
                        long mask30 = (~mask) & (((long)1 << 30) - 1);
                        bitStream = (bitStream << 30) | (totalCounter ^ mask30);
                        streamLen += 30;
                        break;

                case 5:
                        if (0 != streamLen)
                        {
                                int n = 6 - streamLen;
                                bitStream = bitStream << n;
                                streamLen += n;
                        }
                        break;

                case 6:
                        while (bytes.Length > byteCount)
                        bytes[byteCount++] = 48; /* padding with "0" */
                        return bytes;
        }
        /* try to read from the bit stream */while (6 <= streamLen)
        {
```

```
		if (bytes.Length <= byteCount)
			{
				/* we are ending up with too many characters, drop half a signature byte and start
					over */
				signatureBitCountModifier--;
				state = -1;
				break;
			}

			int read = 6;
			int bits = (int)((bitStream >> (streamLen - read)) & 63);
			switch (bits)
			{
				case 0:
					read = 5;
					break;

				case 1:
					bits = 0;
					read = 5;
					break;

				case 62:
				case 63:
					bits = 1;
					read = 5;
					break;
			}
			bytes[byteCount++] = (byte)(bits + ((10 > bits) ? 48 : (35 < bits) ?
				61 : 55));
			streamLen -= read;
			bitStream &= (((long)1) << streamLen) - 1;
		}
}
/* if we got here the encoding failed */return null;
}
```

3.12. DECODER

This implementation assumes that the parameter bytes is an array with a length equal to or greater than 19.

```
public static bool Decode(byte[] bytes)
{
		// State machine states:
		// 0. get signatureBitCount
		// 1. get signature byte #1
		// 2. get signature byte #2
		// 3. get signature byte #3
		// 4. get signature byte #4 MSB (according to the signatureBitCount)
		// 5. get signature byte #4 LSB (according to the signatureBitCount)
```

```
// 6. get signature byte #5 MSB (according to the signatureBitCount)
// 7. get signature byte #5 LSB (according to the signatureBitCount)
// 8. get FDM manufacturer, model, and serial number
// 9. get TotalCounter
// 10. verify padding (must be zeroes)

int state = 0;
long bitStream = 0;
int streamLen = 0;
int nextByteIndex = 0;
long mask = 0;

/* output variables */
StringBuilder fdm = new StringBuilder();
long totalCounter = 0;
byte[] signatureBytes = new byte[5];
int signatureBitCount = 40;

while (state <= 10)
{
        if (nextByteIndex < bytes.Length)
        {
                int b = bytes[nextByteIndex];
bitStream &= (((long)1) << streamLen) - 1;

/* convert the byte to the range (-1, 63), where -1 indicates illegal character */
b -= (48 <= b && 57 >= b) ? 48 : (65 <= b && 90 >= b) ?
     55 : (97 <= b && 122 >= b) ? 61 : b + 1;

switch (b)
{
        case -1:
                return false; /* invalid character in input */

        case 0:
                bitStream = (bitStream << 5);
                streamLen += 5;
                break;

        case 1:
                bitStream = (bitStream << 5) | 31;
                streamLen += 5;
                break;

                default:
                        bitStream = (bitStream << 6) | (byte)b;
                        streamLen += 6;
                        break;
        }
        nextByteIndex++;
}
else if (nextByteIndex++ > bytes.Length)
```

```
        return false; /* we ran out of characters but the state machine is still waiting for more bits */

    switch (state)
    {
        case 0:
            if (2 <= streamLen)
            {
                int modifier = (int)(((bitStream >> (streamLen - 2)) & 3) + 1);
                signatureBitCount -= (4 * (4 - modifier));
                streamLen -= 2;
                state++;
            }

            break;
        case 1:
        case 2:
        case 3:
            if (8 <= streamLen)
            {
                signatureBytes[state - 1] = (byte)(bitStream >> (streamLen - 8));
                streamLen -= 8;
                state++;
            }
            break;

        case 4:
            if (4 <= streamLen)
            {
                if (28 <= signatureBitCount)
                {
                    signatureBytes[state - 1] |=
                        (byte)(((bitStream >> (streamLen - 4)) & 15) << 4);
                    streamLen -= 4;
                }
                state++;
            }
            break;

        case 5:
            if (4 <= streamLen)
            {
                if (32 <= signatureBitCount)
                {
                    signatureBytes[state - 2] |=
                        (byte)((bitStream >> (streamLen - 4)) & 15);
                    streamLen -= 4;
                }
                state++;
            }
            break;

        case 6:
```

```
		if (4 <= streamLen)
			{
				if (36 <= signatureBitCount)
				{
					signatureBytes[state - 2] |=
						(byte)(((bitStream >> (streamLen - 4)) & 15) << 4);
					streamLen -= 4;
				}
				state++;
			}
			break;

	case 7:
		if (4 <= streamLen)
		{
			if (40 <= signatureBitCount)
			{
				signatureBytes[state - 3] |=
					(byte)((bitStream >> (streamLen - 4)) & 15);
				streamLen -= 4;
			}
			state++;
		}
		break;

	case 8:
		if (41 <= streamLen)
		{
			mask = signatureBytes[1];
			mask = (mask << 8) + signatureBytes[0];
			mask = (mask << 8) + ~signatureBytes[0];
			mask = (mask << 8) + ~signatureBytes[1];
			mask = (mask << 8) + signatureBytes[0];
			mask = (mask << 8) + signatureBytes[1];

			long mask41 = mask & ((long)1 << 41) - 1;
			long v = ((bitStream >> (streamLen - 41)) & 0x1FFFFFFFFFF) ^ mask41;
			fdm.Append((char)((v % 26) + 65));
			fdm.Append((char)(((v / 26 % 26)) + 65));
			fdm.Append((char)((v / 676 % 26) + 65));
			fdm.Append((v / 17576).ToString("00000000"));
			streamLen -= 41;
			state++;
		}
		break;

	case 9:
		if (30 <= streamLen)
		{
			long mask30 = (~mask) & ((long)1 << 30) - 1;
			totalCounter =
```

```
                        ((long)((bitStream >> (streamLen - 30)) & 0x1FFFFFFFFF)) ^
                        mask30;
                    streamLen -= 30;
                    state++;
                }
                break;

            case 10:
                if (0 != (bitStream & ((long)1 << streamLen) - 1))
                    return false; /* non-zero padding encountered */
                return true; /* output variables are set */
            }
        }
        return false;
    }
```

CHAPTER 4 – TEMPORARY STORAGE OF DATA

The FDM has a data storage memory of at least 2 GB. This allows the FDM to (temporarily) buffer the enriched JSON messages before they are forwarded to the FPSFIN cloud service.

These enriched JSON messages, supplemented with some additional fields (see further), form the payload of the messages that will be sent to ws_reg.
The FDM ensures that these messages cannot be deleted or modified.

Stored messages can only be copied/viewed via the back office. In the case no internet connection can be established, the viewing and copying of the messages should be available via de backoffice.

Forwarded enriched JSON messages may only be deleted after transmission according to the provisions of Chapter 5.

CHAPTER 5 – COMMUNICATION FDM ↔ FPSFIN CLOUDSERVICE

5.1. CONNECTION AND SECURITY

5.1.1. CONNECTION

The connection via the internet is configured through the back office of the FDM. This back office is specific to the manufacturer, and the user interface may vary. At a minimum, the back office provides the following configuration options:

- The URLs of the FPSFIN cloud service;
- The URL of the time server.

5.1..2. SECURITY

Every communication is secured through the use of:

- The authentication certificate;
- he mTLS protocol, which encrypts the transmissions.

5.2. FIRST TIME EVER

In accordance with the provisions of the ministerial decree of 29/04/2024, both the cash register system and the Fiscal Data Module are registered by the involved stakeholders in the various phases (production, delivery) in the online GKS e-service of the FPS Finance.

At the enterprise and establishment level, it is determined which FDM is linked to which cash register system.

When the GKS is put into service, the cash register and FDM are linked to each other. Upon a correct startup (the unique responsibility of the hospitality business and its supplier(s)), the FDM will contact the cloud service of the FPS Finance via a secure internet connection. The FPSFIN cloud service:

1. verifies the serial number of the FDM, assigns the unique and personalized authentication certificate, and sends it via the secure connection to the FDM, which stores it in the secure environment (art. 77 of this ministerial decree);
2. simultaneously sends the initialization package (overview of the applicable VAT rates, personalized frequency settings,…);

The full detailed description is included in the document FPSFIN_API_PROTOCOL_FDM_FINCLOUD which is also available on our website www.geregistreerdkassasystqeem.be/nl/gks-2-0 or www.systemedecaisseenregistreuse.be/fr/sce-2-2 .

5.3. API AND PROTOCOL FOR COMMUNICATION WITH FPS FINANCE CLOUD SERVICE

Reference MD: Title II, Chapter 2, Section 4.

The FDM communicates with the (cloud) servers of the FPSFIN using the protocol described in the document FPSFIN_API_PROTOCOL_FDM_FINCLOUD which is available on the website www.geregistreerdkassasystqeem.be/nl/gks-2-0 or www.systemedecaisseenregistreuse.be/fr/sce-2-2 .

No communication other than this is allowed.

CHAPTER 6 - ONLINE FORWARDING OF TRANSACTION DATA

Depending on the configured transmission frequency, the FDM sends packets of transaction data (JSON messages) from the FDM to the FPS Finance cloud service. Such packets contain at least the content of one complete event or a NOP (see Chapter 4 of this annex).

The FPS Finance cloud service confirms (or denies) the successful receipt of the packets. The FDM retains these packets for 10 days, after which they may be deleted from the buffer.

These packets are retained by the FPS Finance cloud service for at least 3 years.

This retention does not affect the legal tax retention and submission obligations of the original data on the cash register system (Articles 315bis and 315ter of the Income Tax Code 1992 and Articles 60 and 63 of the VAT Code).

Data other than those included in this ministerial decree cannot and must not be forwarded via this connection.

6.1. JSON MESSAGE TO FPS FINANCE

The JSON message forwarded to the FPS Finance servers via ws_reg contains the arrays, objects, and values as sent by the cash register system to the FDM and those created by the FDM itself. This is referred to as the enriched JSON.

{

unsentEventCount (numeric, mandatory)

Number of unsent messages in the buffer. Always 0 in case of a No Operation (NOP).

fdmId (string, mandatory)

Serial number of the FDM.

events (array, mandatory)

Array of enriched events. Mandatory, in case of a NOP the array is empty.

[

{

enrichedEventData (object, mandatory)

Object containing event data from the cash register, supplemented with calculations from the FDM. The fields included in this object are described in the Detailed POS Description.

digitalSignature (string, mandatory)

Base64 digital signature based on the canonical JSON re-encoding of `enrichedEventData`.

shortSignature (string, conditional)

SHA-1 checksum in hex of the digital signature in bytes. Only calculated for event N.

verificationUrl (string, conditioneel)

This URL is generated by the FDM based on a fixed prefix and an encoded value, so that it can be printed as a QR code on the VAT receipt. Only calculated for event N.

fdmLocalisation (object, mandatory)

}

In exceptional cases, when errors occur when reading a transaction, this object may be replaced by:

{

rawCounter (numeric, mandatory)

This counter is incremented each time a corrupt transaction is sent.

rawBytes (string, optional)

The raw bytes in base64 of the corrupt transaction. This is filled in when the transaction is (partly) readable.

rawInfo (string, mandatory)

Additional information from the FDM, such as error messages.

}

]

}

fdmLocalisation

Mandatory object related to the localization of the FDM at the time of communication. If this FDM includes an optional 4G/5G module, these coordinates must be sent.

{

geoLocation (object, optional)

Contains localization data of the optional GPS module of the FDM.

{

latitude (numeric, mandatory)

Contains the latitude data, in decimal format. Min value = -90, max value = 90. Format = 2,6. Example: 50.851415.

longitude (numeric, mandatory)

Contains the longitude data, in decimal format. Min value = -180, max value = 180. Format = 3,6. Example: 4.354365.

}

}