5 RFID Commands

This section will cover different emulations that support RFID within the Printronix RFID thermal products. For a full listing of supported commands, please refer to the *Programmers Reference Manuals* available on our website at www.PrintronixAutoID.com:

- Incorporate RFID commands into new or existing Printronix PGL® programs.
- Incorporate RFID commands into new or existing ZPL™ programs. By selecting the Printronix ZGL emulation you can seamlessly upgrade from Zebra™ printers.
- Incorporate RFID commands into new or existing SATO® printer language programs. By selecting the Printronix STGL emulation you can seamlessly upgrade from SATO printers. PGL RFID Commands.

IMPORTANT **With all examples make sure *MEDIA > Image > Label Length* matches the physical length of the installed media.**

RFWTAG

Purpose The RFWTAG command is used to program an RFID tag (embedded in a smart label) using structured data format. The data structure of an RFID tag can consist of one or more bit fields. Each bit field specifies its own field length, the data format, the field type plus additional options if the type is incremental, and finally the field value.

Mode CREATE

Format RFWTAG[;LOCKn*[;format]*];*size*[;offset][*;mem* bank]

(Bit Field) +

STOP

RFWTAG Specifies the RFWTAG command, enter **RFWTAG;**

LOCK*n[;format] or* PERMALOCK*n[;format]*
Optional parameter to lock the data block to prevent it from being overwritten. By default, the data are not locked initially. n is the passcode. The acceptable values for n are 1 to FFFFFFFF in hex, a 4 bytes data. When the LOCKn option is used to lock any memory bank, which at the same time is programmed with the write data, the same passcode will be written on ACS memory bank. The ACS memory bank will also be locked if ACS is not locked at the time of the operation. If ACS is already locked at the time of the operation, the passcode needs to match the current content of ACS so that the memory bank lock takes effect. The passcode (n) can also be in dynamic format. For dynamic format, enter LOCK<DFn>, where DFn is the dynamic field defined in EXECUTE mode. Both LOCK and PERMALOCK share the same syntax. For differences in functionality, see Note 12 on page **Error! Bookmark not defined.**.

format An optional parameter to specify the format for the passcode data. Enter B for binary, D for decimal, S for alphanumeric, and H for hexadecimal.

The default is decimal if *format* is not specified.

size — A decimal number specifying the overall bit length of the memory bank that will be written starting at the *offset* position (not necessarily the total bank size). Regardless of the overall bank size, any given write segment cannot exceed 256 bytes (2048 bits).

offset — This optional parameter of starting position to do the write relative to the start of the mem bank. The position is a word value (16 bits).

mem bank — Specifies which tag logical memory area that this command will be applied. If omitted, it defaults to the EPC memory area. Other areas include Identification, User Data, Access area and Kill area. Enter one of the following values:

'EPC' – EPC 12 bytes data area (default)

'TID' – Tag identification 8 bytes area (currently N/A)

'USR' – User 32 bytes area

'ACS' – 4 bytes access code area

'KIL' – 4 bytes kill code area

'PC' – 2 bytes PC code area (Gen 2 tags only)

New tags, such as RSI IN47 Crkscr, support 240 bits of EPC memory and 512 bits of USR memory.

Bit Field — A line description of a bit field and must have one of the following syntax formats:

1. For non-incremental data (both static and dynamic):

   length;[**DF***n*;]*format*;(D)*datafield*(D)

2. For incremental fixed data:

   length;**I;***format*;**STEP**[*idir*]*step*;[**RPT***n*;][**RST***n*;](D)*startdata*(D)

3. For dynamic incremental data:

   length;**IDF***n*;*format*;

length — A decimal number specifying the bit length of a field within a tag. The maximum length for each DFn field of NON-HEX format is 64 bits. For hexadecimal format, the bit length can be up to the maximum bit length specified for the corresponding memory bank.

DF*n* — Optional parameter to indicate this field has dynamic data. Replace *n* with a number ranging from 1 to 512 to identify the field number of this particular field. If this option is used, *datafield* is ignored, and dynamic data must be entered via the DF command in the EXECUTE mode.

IDF*n* — Enter **IDF** to indicate this field is a bit field with dynamical assignment of increment (or decrement) data. The *step* and *startdata* parameters will be supplied by the IDF command in the EXECUTE mode. Replace *n* with a number ranging from 1 to 512 to identify the field number of this bit field. Dynamically enter the *step* and *startdata* parameters using the IDF command in the EXECUTE mode.

1. The same field number cannot be used in both DFn and IDFn.

2. If a field is defined as IDFn, it must be referenced as IDFn later for consistency. The same

applies for DFn.

3. If <IDFn> syntax is used for merging data into AFn or BFn, neither DFn, AFn, or BFn will be incremented. The increment only takes place in the ~DFn command where the STEP is specified.

format	A letter specifying the format of the data field. **B** – binary, **D** – decimal, **S** – string, **H** – hexadecimal
(D)	Delimiter designating the start and end of static data for this bit field. Replace (D) with any printable character, except the SFCC and the slash character (/).
datafield	The static data of this static field. It is a mandatory parameter of bit field with static data.
I	Identifies this field is an incremental bit field.
STEP	Specifies that the incremental data field will use the step method. Enter **STEP;**. The STEP option replaces the STEPMASK option that is used in Alpha and Barcode.
idir	Enter a plus sign (+) or leave the field blank to increment (default). Enter a minus sign (–) to decrement.
step	A decimal number specifies the amount to increment/decrement each time the form is executed. The increment is at bit level and will automatically wrap based on the field size.
RPT*n*	The optional incremental repeat count parameters to specify the number of times a particular field value is repeated before it is incremented. The default repeat count parameter *n* is 1, which will increment the field value each time it prints. The repeat count can range from 1 to 65535.
RST*n*	The optional incremental reset count parameter to specify the number of times an incremented field is printed before it is reset to the starting value. By default, there is no reset count. The reset count parameter *n* can range from 1 to 65535.
startdata	Defines the value of the field or the starting value of the incremented field. If the field is dynamic, the value will be specified later in the EXECUTE mode. The data must be specified within a pair of delimiters (D). The delimiter (D) cannot be a “/” or SFCC character since the “/” will comment out the rest of the line and SFCC is reserved for PGL commands. If “R” or “S” is used as delimiters, the data pattern must not comprise of the keywords in the incrementing options. Since the delimiters could be different from one value to another, proper care must be taken to avoid one of the letters mentioned above.

1. The RFWTAG command cannot be mixed with RFWRITE in the same form.

2. Each field structure must be specified in a single line and in the order it appears in the RFID tag from MSB bits to LSB bits (left to right). The sum of all the field lengths must match the size of the tag.

3. The host data are read in as ASCII characters. They would be converted to binary representation for the base field on the field format. Therefore, if the converted value is larger than the maximum value that a field can hold, an error will be reported. If the data value is smaller than the specified field length, on the other hand, the field will be padded to the left with zero bits.

4. Unlike the Alpha and Barcode command which use STEPMASK for incremental data, RFWTAG uses the STEP which will increment or decrement at bit level.

5. 432 IGP dots in the ~CREATE line specifies a 6 inch label. 6 inches = 432 (IGP dots)/72 (dpi)

   Use 144 for 2 inch labels and 288 for 4 inch labels.

6. ACS and KIL are similar to other memory banks. ACS contains the passcode which is used for LOCK and UNLOCK operations. KIL contains the killcode which is used to kill a tag. The user can write to or read from KIL memory bank, but the functionality of killing a tag is not currently applicable. Also, once ACS and KIL are locked, both cannot be written to or read from. For other memory banks, EPC, USR, and TID, once locked, they can be read from but not written to.

7. There are two ways to program the ACS memory area. One is to write to the ACS memory area directly with RFWTAG. The other is to use the LOCK option while writing to other memory banks. If ACS is not previously locked, then LOCK option will lock the memory bank and also write the passcode to ACS and lock ACS. When write to ACS with RFWTAG, ACS is not automatically locked. To lock ACS, use LOCKn with RFWTAG, where the passcode (n) should be the same as the write data to ASC.

8. There is only one passcode, the content of ACS memory bank, for each tag. The same passcode is used to lock or unlock any memory bank in that tag.

9. For LOCKn and UNLOCKn, the passcode (n) (which includes the dynamic format <DFn>) does not accept incremental data. This also applies to the ACS and KIL memory banks. The write data to the ACS and KIL memory banks do not accept incremental data because the ACS memory bank contains passcodes for LOCK and UNLOCK operations, and the KIL memory bank contains a killcode to kill a tag. Incremental data do not apply to passcodes or killcodes.

10. When LOCK<DFn> and UNLOCK<DFn> are used in the same form with the same dynamic data (the passcode), the dynamic format <DFn> needs to be a different dynamic number for LOCK and UNLOCK since it is designed with a unique dynamic number that can be linked to only one object type. In this case, LOCK is linked to RFWTAG object and UNLOCK is linked to RFRTAG object. Although both options use the same passcode, the dynamic format needs to be in a different dynamic number in the same form.

11. Because PC field is related to EPC field, when PC RFWTAG is used in the form, it must be followed immediately by EPC RFWTAG, or else an error will be reported. Also, by specification, the first 5 bits of PC data need to comply with the length of EPC data, or else an error will be reported. For example, for 96 bits EPC, the 5 bits of PC data is 00110. For 64 bits EPC, the first 5 bits of PC data is 00100. Also, LOCK option is not and will not be supported for PC field, since PC field works with EPC field (which already supports LOCK option).

12. Both LOCK and PERMALOCK requires the user to enter the password. Once the tag is permanently locked with the PERMALOCK command, it cannot be unlocked again; the tag can only be read from and never be written to once it is permanently locked. On the other hand, after the tag is locked with the LOCK command, it can be unlocked again with the same password.

For PERMALOCK (ex, EPC), the password must match the current content of ACS bank for PERMALOCK to work. If the current content of ACS bank is null (0x0) which could be the case for the brand new tag, the password for PERMALOCK EPC will be 0x0. If you use a different password for PERMALOCK, you need to write (RFWTAG) the new content (password) to ACS first, and then use this new password to PERMALOCK EPC.

For LOCK (ex, EPC), the password may be different from the current content of ACS. When a new password is used to lock EPC where ACS is not locked, this new password is written to ACS and locks ACS at the same time while locking EPC. For new tags where ACS is not locked and has all null data, you can lock EPC with a new password directly without writing to ACS first.

Example 1

The following example programs an SGTIN–64 value into the RFID tag that is embedded in a 4x6 smart label. Assume that the SGTIN–64 value is provided as a single number.

```
~CREATE;SGTIN-64;432
RFWTAG;64
64;H;*87D0034567ABCDEF*        /EPC number
STOP
END
~EXECUTE;SGTIN-64;1
~NORMAL
```

Example 2

Same as Example 1, except the EPC number is broken into its component parts. Assume that the SGTIN–64 value has the Header = 2d, Filter Value = 5d, EPC Manager Index = 15383d, Object Class = 703710d or 0xABCDE, and the Serial Number = 0123456d.

```
~CREATE;SGTIN-64;432
RFWTAG;64
2;B;*10*              /Header
3;D;*5*               /Filter Value
14;D;*15383*          /EPC Manager Index
20;H;*ABCDE*          /Object Class
25;D;*0000123456*     /Serial Number
STOP
END
~EXECUTE;SGTIN-64;1
~NORMAL
```

Example 3

Same as Example 2, except it uses a dynamic method. This example also shows how to program another RFID tag without redefining the data structure of the SGTIN–64.

```
~CREATE;SGTIN-64;432
RFWTAG;64
2;DF1;B               /Header
3;DF2;D               /Filter Value
14;DF3;D              /EPC Manager Index
20;DF4;H              /Object Class
25;DF5;D              /Serial Number
STOP
ALPHA
```

```
AF1;18;10;5;3;3
STOP
END
~EXECUTE;SGTIN–64
~DF1;*10*              /Header
~DF2;*5*               /Filter Value
~DF3;*15383*           /EPC Manager Index
~DF4;*ABCDE*           /Object Class
~DF5;*0000123456*      /Serial Number
~AF1;<DF5>             /Print serial number on label
~NORMAL
~EXECUTE;SGTIN–64
~DF1;*10*              /Header
~DF2;*5*               /Filter Value
~DF3;*15383*           /EPC Manager Index
~DF4;*ABCDE*           /Object Class
~DF5;*0000123456*      /Serial Number
~AF1;<DF5>             /Print serial number on label
~NORMAL
```

Example 4

This example shows how to program a roll of 1500 smart labels with SGTIN–64 values, where the Header = 2d, Filter Value = 5d, EPC Manager Index = 15383d, Object Class = 703710d or 0xABCDE, and the Serial Number starting from 0000000 to 0001499d.

```
~CREATE;SGTIN–64;432
RFWTAG;64
2;B;*10*               /Header
3;D;*5*                /Filter Value
14;D;*15383*           /EPC Manager Index
20;H;*ABCDE*           /Object Class
25;I;D;STEP1;*0*       /Serial Number
STOP
END
~EXECUTE;SGTIN–64;ICNT1500
~NORMAL
```

Example 5

This example shows how to program a 96 bit RFID tag. A SGTIN–96 format is used and the EPC number is broken into its component parts. Assume that the SGTIN–96 value has the Header = 48, Filter Value = 5d, EPC Manager Index = 123456d, Object Class = 777777d or 0xBDE31, and the Serial Number = 123456d.

96 bit tags must be broken up as in Examples 2, 3, and 4, and no field can be more than 64 bits in length if the format is binary or decimal. There is no restriction on the bit length if the format is hexadecimal.

```
~CREATE;SGTIN–96;432
RFWTAG;96
8;B;*00110000*         /Header
3;D;*5*                /Filter Value
3;D;*6*                /Partition
20;D;*123456*          /EPC Manager Index
```

```
24;D;*777777*          /Object Class
38;D;*123456*          /Serial Number
STOP
END
~EXECUTE;SGTIN-96;1
~NORMAL
```

Example 6

This example shows memory bank usage, where multiple RFWTAG and RFRTAG can be used.

```
~CREATE;SGTIN;216
SCALE;DOT;203;203
RFWTAG;96;EPC
96;IDF1;H
STOP
RFRTAG;96;EPC
96;DF3;H
STOP
RFWTAG;256;USR
256;IDF2;H
STOP
RFRTAG;256;USR
256;DF4;H
STOP

ALPHA
IAF1;24;POINT;90;60;16;6
IAF2;64;POINT;130;60;16;4
STOP

BARCODE
C3/9;X1;IBF1;64;170;60
PDF
STOP

VERIFY;DF1;H;*EPC W= *;*\r\n*
VERIFY;DF3;H;*EPC R= *;*\r\n*
VERIFY;DF2;H;*USR W= *;*\r\n*
VERIFY;DF4;H;*USR    R= *;*\r\n*

END
~EXECUTE;SGTIN;ICNT4
~IDF1;STEP+1;*31323334353637383941424344454647484 94A
4B4C4D4E4F*
~IAF1;<DF3>
~IAF2;<DF4>
~IBF1;<DF3>
~NORMAL
```

Example 7

This example shows memory bank usage with LOCK and UNLOCK option, where multiple RFWTAG and RFRTAG can be used, and the passcode for lock and unlock can be in dynamic format.

```
~CREATE;SGTIN;432
SCALE;DOT;203;203
RFWTAG;LOCK<DF6>;D;96;EPC
96;DF1;H
STOP
RFRTAG;UNLOCK<DF7>;D;96;EPC
96;DF2;H
STOP
RFWTAG;LOCKA1B2C3;H;32;KIL
32;DF3;H
STOP
RFRTAG;UNLOCKA1B2C3;H;32;KIL
32;DF4;H
STOP
RFWTAG;LOCK<DF8>;H;32;ACS
32;DF6;D
STOP
RFRTAG;UNLOCK<DF9>;H;32;ACS
32;DF10;H
STOP
ALPHA
AF1;24;POINT;400;60;16;6
AF2;8;POINT;600;60;16;6
AF3;6;POINT;800;60;16;6
AF4;8;POINT;1000;60;16;6
STOP
VERIFY;DF1;H;*DF1 = *;*\r\n*
VERIFY;DF2;H;*DF2 = *;*\r\n*
VERIFY;DF4;H;*DF4 = *;*\r\n*
VERIFY;DF6;H;*DF6 = *;*\r\n*
VERIFY;DF7;H;*DF7 = *;*\r\n*
VERIFY;DF8;H;*DF8 = *;*\r\n*
VERIFY;DF9;H;*DF9 = *;*\r\n*
VERIFY;DF10;H;*DF10 = *;*\r\n*
END
~EXECUTE;SGTIN;FCNT3
~DF1;*31323334353637383941424 3*
~DF3;*44454647*
~DF6;*10597059*
~DF7;*10597059*
~DF8;*A1B2C3*
~DF9;*A1B2C3*
~AF1;<DF2>
~AF2;<DF6>
~AF3;<DF8>
~AF4;<DF10>
~NORMAL
```

Example 8

This example shows the usage of RFWTAG with PC field which needs to be followed immediately by RFWTAG with EPC field. There is not restriction for RFRTAG with PC filed.

```
~NORMAL
~CREATE;TEST1;432
RFWTAG;16;PC
16;H;*3000*
STOP
RFWTAG;96;EPC
96;H;*313233343536373839414243*
STOP
RFWTAG;256;USR
256;H;*313233343536373839414243444546474849 4A4B*
STOP
RFRTAG;16;PC
16;DF1;H
STOP
RFRTAG;96;EPC
96;DF2;H
STOP
VERIFY;DF1;H;*DF1 = *;*\r\n*
VERIFY;DF2;H;*DF2 = *;*\r\n*
END
~EXECUTE;TEST1
~NORMAL
```

Example 9

This example shows the usage of PERMALOCK.

```
~NORMAL
~CREATE;RFID;432
ALPHA
IAF1;24;POINT;4;5;10;10
STOP
RFWTAG;32;ACS
32;H;*ABC*
STOP
RFWTAG;PERMALOCKABC;H;96;EPC
96;IDF1;H
STOP
RFRTAG;96;EPC
96;DF2;H
STOP
VERIFY;DF2;H;* *
END

~EXECUTE;RFID;ICNT5
IDF1;STEP+1;*222222222222222220011*
IAF1;<DF2>
~NORMAL
```

Example 10

This example shows the access of 240 bits EPC and 512 bits USR.

```
~CREATE;TEST;X;NOMOTION
```

```
RFWTAG;LOCK0C0D0E0F;H;240;EPC
240;I;H;STEP+1;*010203040506070809101112131415
1516171819202122232425262728293031323334353
6373839404142434445464748495051525354555657
5758596061626364*
STOP
RFWTAG;LOCK0C0D0E0F;H;512;USR
512;I;H;STEP+1;*010203040506070809101112131415
1516171819202122232425262728293031323334353
6373839404142434445464748495051525354555657
5758596061626364*
STOP
RFWTAG;LOCK0C0D0E0F;H;32;KIL
32;H;*08090A0B*
STOP
RFRTAG;UNLOCK0C0D0E0F;H;32;ACS
32;DF31;H
STOP
VERIFY;DF31;H;*#ACS=*;“\r\n”
RFRTAG;UNLOCK0C0D0E0F;H;32;KIL
32;DF22;H
STOP
VERIFY;DF22;H;*KIL=*;“\r\n”
RFRTAG;UNLOCK0C0D0E0F;H;240;EPC
240;DF1;H
STOP
VERIFY;DF1;H;*EPC=*;*\r\n
RFRTAG;UNLOCK0C0D0E0F;H;512;USR
512;DF7;H
STOP
VERIFY;DF7;H;*USR=*;*\r\n*
END
~EXECUTE;TEST;10
~NORMAL
```

RFRTAG

Purpose To read the content of an RFID tag (embedded in a smart label) into a dynamic field. This command cannot be mixed with the RFREAD command.

Mode CREATE

Format RFRTAG[;UNLOCKn*[;format]*];*size*[;offset][*;mem bank*]

(Bit Field)+

STOP

RFRTAG	Specifies the RFRTAG command, enter **RFRTAG;**
size	A decimal number specifying the overall bit length of the RFID tag memory bank that will be read starting at the *offset* position (not necessarily the total bank size). Regardless of the overall bank size, any given read segment cannot exceed 256 bytes (2048 bits).
offset	This optional parameter of starting position to do the read relative to the start of the mem bank. The position is a word value (16 bits).

UNLOCK*n[;format]*

Optional parameter to unlock the data block so it can be overwritten later. n is the passcode. The acceptable values for n are 1 to FFFFFFFF in hex, a 4 bytes data. The value of n should be the same passcode used for the LOCK option to unlock the protected data block. When the UNLOCKn option is used to unlock any memory bank, which at the same is programmed to read the tag, the operation UNLOCKn will not unlock ACS memory area. The passcode (n) can also be in dynamic format. For dynamic format, enter LOCK<DFn>, where DFn is the dynamic field defined in EXECUTE mode.

format is the optional parameter to specify the format for the passcode data. Enter B for binary, D for decimal, S for alphanumeric, and H for hexadecimal. The default is decimal if *format* is not specified.

mem bank Specifies which tag logical memory area that this command will be applied. If omitted, it defaults to the EPC memory area. Other areas include Identification, User Data, Access area, and Kill area. Enter one of the following values:

'EPC' – EPC 12 bytes data area (default)

'TID' – Tag identification 8 bytes area

'USR' – User 32 bytes area

'ACS' – 4 bytes access code area

'KIL' – 4 bytes kill code area

'PC' – 2 bytes PC code area (Gen 2 tags only)

Bit Field A line description of a bit field; must have one of the syntax formats:

length;**DF***n***;***format*

length A decimal number specifying the bit length of a field within a tag. The maximum length is 64 bits for binary or decimal format. For hexadecimal format, the bit length can be up to the maximum bit length specified for the corresponding memory bank.

DF*n* Indicate dynamic data field to store the read result. Replace *n* with a number ranging from 1 to 512 to identify the field number of this field.

format A letter specifying the representation format of the field data.

B – binary, **D** – decimal, **S** - string
H – hexadecimal

1. Multiple RFRTAG commands are allowed in the same form but the same DFn field cannot be defined multiple times.
2. The DF field length is restricted to 64 bits for binary or decimal format and must be a multiple of 8 bits. The sum of all field lengths must be equal to the tag size.
3. The first field always start at the MSB bit. The bit length of a field dictates the start bit of the next field, etc. As a result, DF fields will not overlap each other.

4. RFRTAG does not allow incremental fields (with the “I” prefix).
5. 432 IGP dots in the ~CREATE line specifies a 6 inch label. 6 inches = 432 (IGP dots)/72 (dpi)

   Use 144 for 2 inch labels and 288 for 4 inch labels.

Example

Same as Example 4 on page **Error! Bookmark not defined.**, except the increment is dynamic and the result is merged into Alpha to print on the smart label.

```
~CREATE;SGTIN-64;432
RFWTAG;64
2;B;*10*              /Header
3;D;*5*               /Filter Value
14;D;*15383*          /EPC Manager Index
20;D;*123456*         /Object Class
25;IDF1;H             /Serial Number STOP
RFRTAG;64
64;DF2;H;
STOP
ALPHA
IAF1;16;3;12;0;0
STOP
END
~EXECUTE;SGTIN-64;ICNT1500
~IDF1;STEP+1;*0*
~IAF1;<DF2>

~NORMAL
```

1. The <IDF1> usage does not increment the DF1 field. It merges the DF1 content into the AF1 field, keeping the same representation previously defined for IDF1.
2. The use of IAF1 is to print alpha on every label. If AF1 is used instead, only the first label is printed. The AF1 field is not incremented either since it is using the result from the DF1 merge.

VERIFY

Purpose Request the printer to send to the host the ASCII representation of a dynamic field. The dynamic field could be one of AFn, BFn, or DFn, but cannot be RFn.

The Verify command is supported only on Thermal printers.

Mode CREATE

Format VERIFY**;field;***format***;**(D)*ASCIIheader*(D);(D)*ASCIITrailer*(D)

VERIFY	The command to verify data of a dynamic field, enter VERIFY;
field	The dynamic field AFn, BFn, or DFn that contains the data to be sent to the host.
format	A letter specifying the format of the outgoing data to be sent to the host. **B** – binary, **D** – decimal, **H** – hexadecimal, **S** – string Based on the incoming format of the data field, a format conversion may be performed if the outgoing format is not the same. The AFn and BFn format is always S type. The DFn format could be either B, D, or H. Due to the possible conversion the outgoing data stream could be longer than

the incoming one. The maximum length for the outgoing data is 512 bytes. If the format request will result in a data stream exceeding the maximum length, an error would be reported.

ASCIIheader A mandatory parameter to specify an ASCII string of characters, which is followed by the RFID data, to be sent by the printer to the host.

ASCIItrailer An optional parameter to specify an ASCII string of characters, which will follow the RFID data, to be sent by the printer to the host.

(D) Delimiter designating the start and end of a character string. Replace (D) with any printable character, except the SFCC and the slash character (/). The string could be empty, i.e. there are not headers preceding the field data.

1. The DFn field must be defined previously in the CREATE mode before it can be specified in the VERIFY command otherwise it will be considered as a syntax error and the VERIFY command will abort.
2. All RFID Read/Write commands are executed first in the order they appear in CREATE mode, followed by Alpha and Barcode commands, and finally VERIFY commands. The VERIFY commands are always executed last although they may appear before other commands in the CREATE mode. The reason for this is to make sure the data are sent back to the host only if other commands are completed and the form is not aborted.
3. If the data comes from a DFn field, the DFn format is the original format before any conversion. If the VERIFY command specifies a different format, the data would then be converted to the new format. If the data comes from an AFn or BFn, the original format is S format.
4. Below is the possible syntax for header and trailer string,

```
1, VERIFY;DF2;H;*Head = *                  //Header only
2, VERIFY;DF2;H;*Head = *; *Tail*          //Header & trailer
3, VERIFY;DF2;H;**;*Tail*                  //Trailer only
4, VERIFY;DF2;H;*Head = *;**               //Header only
```

To insert the CR/LF character, add “\r” and “\n” as CR/LF characters, such as

```
VERIFY;DF2;H;*Head=*; *Tail\r\n*          //“Head=<tag data>Tail<CR><LF>”
```

If the user wants to display “\r” or “\n” as normal text character, do the following:

```
VERIFY;DF2;H;*Header\\r\\n*    //this will display “Header\r\n” on the screen, where
                               double back slash “\\” (0x5C 0x5C) will be replaced with
                               one back slash ‘\’ (0x5C).
```

The characters \r and \n can be inserted anywhere in the header string and trailer string.

To summarize,

```
\r –> 0x0D                     //CR
\n –> 0x0A                     //LF
\\ –> \                        //one back slash
```

Example 1

This example requests the printer to send to the host the content of the RFID tag, in hexadecimal format, both before and after the RFWTAG command writes data to the tag. Also, the label is not moved.

```
~CREATE;VERIFY;432;NOMOTION
RFRTAG;64
```

```
64;DF1;H
STOP
VERIFY;DF1;H;*TagBefore=*
RFWTAG; 64
2;B;*01*
6;D;*29*
24;H;*466958*
17;H;*ABC*
15;D;*1234*
STOP
RFRTAG;64
64;DF2;H
STOP
VERIFY;DF2;H;*TagAfter=*
END
~EXECUTE;VERIFY;1
~NORMAL
TagBefore=A5A500005D055E04          <== Whatever data inside the tag before
TagAfter=5D466958055E04D2           <== Should match with RFWTAG command
```

Example 2

This example reads a roll of 1500 pre-programmed smart labels.

```
~CREATE;READONLY;432
RFRTAG;64
64;DF1;H
STOP
VERIFY;DF1;H;**
END
~EXECUTE;READONLY;1500
~NORMAL

A5A500005D055E04                    <== Data from first tag
   …                                <== Data from 1498 tags in middle
A5A50000000550D4                    <== Data from last tag
```

Example 3

This example requests the printer to program a roll of 2000 smart labels using the RFWTAG command with incremental field. Then, it sends the actual data from each of the 2000 tags to the host.

```
~CREATE;SIMPLE;432;NOMOTION
RFWTAG;64
2;B;*01*
6;D;*29*
24;H;*466958*
17;H;*ABC*
15;I;D;STEP+1;*0000*
STOP
RFRTAG; 64
64;DF1;H
STOP
VERIFY;DF1;H;*Data=*
```