Honeywell

Decode Filter Script

For Honeywell Mobile Computers Powered by Android

Command Reference Guide

Disclaimer

Honeywell International Inc. (“HII”) reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult HII to determine whether any such changes have been made. HII makes no representation or warranties regarding the information provided in this publication.

HII shall not be liable for technical or editorial errors or omissions contained herein; nor for incidental or consequential damages resulting from the furnishing, performance, or use of this material. HII disclaims all responsibility for the selection and use of software and/or hardware to achieve intended results.

Web Address: sps.honeywell.com

For patent information, refer to www.hsmpats.com.

TABLE OF CONTENTS

CHAPTER

1 GETTING STARTED

Introduction

This document describes how to configure a filter for decode results during scanning. The Decode Filter feature provides configurability for modifying or rejecting data strings as they emerge from the barcode decoder.

This feature applies to Honeywell Android devices with integrated scanners.

The Decode Filter is specified in a script form. The script is applied through a configuration property of the internal scanner. The property is a multi-line string value, containing the filter script. The filter script can call built-in functions to test and extract parts of the decoded data, compose modified output, and save information for subsequent scans.

Note: *The content here assumes familiarity with configuring scanning properties and with general programming.*

About the Filter Script

The filter runs at the start of each scan, and after every decode result during the scan.

The filter script resembles some common languages but is far more limited. It supports if-blocks, statements, expressions, variables, constant strings, a few functions and a few operators.

- The only data type is a string.
- All variables share a common namespace.
- The if-block is the only type of control structure.
- Everything is case-sensitive.

Variables

The Decode Filter can read and store values in variables.

Variables may be named anything using alpha-numeric characters and underscore, not starting with a number.

All variables exist in a common namespace.

All variables persist between calls to the filter.

All variables are set to empty when a script is compiled into a filter.

Input/Output

The Decode Filter interacts with decoding through a few variables. These variables are set before the script is invoked, and examined after the script completes.

These variable names, and any future input/output variable names, start with the '_' character.

Variable Name	Initial Value	Effect if value is modified by the script
_event	"start" – The start of a scan cycle. Use this event to initialize any persisted variables as needed. "decode" – Scanning decoded a data string. "timeout" - Use this event to signal the elapsed time.	No effect
_data	The decoded data	If set to the empty string value, scanning continues as if the decode did not occur. Otherwise, this value is used as the decoded data.
_aimid	The AIM identifier for a barcode	The 2nd and 3rd characters are used as the altered AIM identifier.
_time	Elapsed time since the start of scan cycle.	No effect

Named Functions

Function	Description

and(...)	Returns a non-empty string value if all the parameters are non-empty. Otherwise, returns empty.
	
	Example
	`if( and(_match1,match2) ) {`
	`  _data=concat(_match1,":",_match2);`
	`}`
	
	The assignment inside the if block is only executed if _match1 and _match2 are non-empty.

concat(...)	Returns the concatenation of all parameters. **Example** `_data=concat(_data, "<SCAN");` The _data variable is assigned the value of _data appended by the string "<SCAN".
find(subject, substr)	Searches for the first occurrence of substr in subject. Returns the 0-based character position of substr as a base-10 string. Returns empty if the search fails. **Example** `mypos=find("Hello World", "World");` The variable mypos will return "6" as this is the zero-based position of "World" inside the string "Hello World".
in(subject, ...)	Returns a non-empty value if subject is equal to any of the other parameters. Otherwise, returns empty. **Example** `haseuro=in("Dollar", "Euro", "Yen");` The variable haseuro will have the empty value as "Dollar" is not in the listed parameters.
or(...)	Returns the value of the first non-empty parameter, or else the empty value. **Example** `firstmatch=or("", "second", "third");` The variable firstmatch will have the value "second" as this is the first non-empty parameter.

Function	Description

regex(subject, pattern)	Applies the regular expression pattern to the subject string. Returns empty if no match occurs, otherwise returns the full match value.
	
	Match groups are written to variables _match0, _match1, _match2, and so on.
	
	Example
	`mymatch=_data.regex('^00(.*)');`
	-or-
	`mymatch=regex(_data, '^00(.*)');`
	
	regex is very powerful. The syntax is similar to PCRE. See, for example, regex101.com for regex evaluation.
	
	The above example will set mymatch to non-empty if the _data variable starts with "00" followed by anything. The group match (.*) will be returned in the automatic variable _match1. Here everything except the starting "00" will be returned as _match1.

Function	Description

sub(subject, start, end)	Extract a substring from subject. Returns the extracted substring.
	
	subject: The main string.
	
	start: The character position of the start of the substring.
	"nnn": Any base-10 integer specifies the starting character position in the subject string. The first character in subject is position "0".
	"-nnn": Start from the number of characters from the end of subject.
	"@*delimiter*": Find the first occurrence of *delimiter* in *subject*, and start immediately after it.
	
	end:
	"nnn": Any base-10 integer specifies the position of the character immediately after the substring.
	"-nnn": End at the specified number of characters before the end of *subject*.
	"+nnn": End at the specified number of characters after the start position.
	"@*delimiter*": Beginning from the start position, find the first occurrence of *delimiter* in *subject*, and end immediately before it.
	
	Example
	
	`mysub=sub("Hello World", "2");`
	Will return all characters of _data starting with zero-based position 2, so mysub will be "llo World".
	
	`mysub=sub("Hello World", "0", "5");`
	Will return the substring starting with 0 until zero-based position 5. This will give "Hello"
	
	`mysub=sub("Hello World", "1", "5");`
	Will return the substring starting with 1 until zero-based position 5. This will give "ello"
	
	`mysub=sub("1234567890", "-4");`
	Will return the last 4 characters: "7890"
	
	`mysub=sub("1234567890","5","+2");`
	Will return "67", a substring of 2 characters starting at zero-based position 5. If +nnn as end parameter is greater than the length of the string, the remaining characters are returned.
	
	`mysub=sub("1234567890", "@6");`
	Returns everything after the first "6" character: "7890". If the character to look for is not found in the string, nothing is returned.
	
	`mysub=sub("Hello World", "@ ", "-2");`
	Returns the substring starting after the first space character until two characters before the end of the string. If the end parameter -nnn is larger than the string, nothing is returned.

Function	Description

replace (subject, start, end, old_value, new_value)	Extract a substring from subject and replace the substring's old_value with a new_value.
	
	subject: The main string.
	
	start: similar definition as presented in sub function.
	
	end: similar definition as presented in sub function.
	
	old_value: character or substring to replace
	
	new_Value: character or substring to use as replacement
	
	Example 1
	`mysub=replace("Hello World","1","9","l","L")`
	Will return the substring starting with 1 until zero-based position 9 replacing all occurrences of "l" with "L", so mysub will be "eLLo Wor".
	
	Example 2
	`mysub=replace("Hello World","0","-4","l","L")`
	Will return the substring starting with 0 until zero-based position until 4 characters from the end, replacing all occurrences of "l" with "L", so mysub will be "HeLLo W".
	
	Example 3
	`mysub=replace("Hello World","0","","l","L")`
	Will return the substring starting with 0 until zero-based position until the end of the string, replacing all occurrences of "l" with "L", so mysub will be " HeLLo WorLd".

Constants

Represent a constant value by enclosing it in single-quotes, double-quotes or forward slashes.

Escape sequences

Sequence	Replacement
\xhh	A single character having the code unit value *hh*, where *h* is a hexadecimal digit.
\character	Character, assuming character isn't part of a sequence listed above.

In constants enclosed by single- and double-quotes, the escape sequence is replaced per the table.

Examples

```
If(_event=='decode'){
  _data=_data.concat('\x40');
}
```

This will add the character with hex value 0x40 (decimal 64, char is the at symbol '@') at the end of the data.

In constants enclosed by forward slashes, no replacements are made. The \ character is used only to force a / character into the constant string, rather than marking the end.

```
If(_event=='decode'){
  sub2=_data.sub('@\x1d');
  _data=sub2;
}
```

Using the @ in a sub expression makes the sub look for the first occurrence of the following character. The \x1d is the non-printable character for the GS or FNC1 code. In the example, only the part after the first GS symbol is returned. If no GS is inside the data, nothing is returned.

```
If(_event=='decode'){
  sub2=_data.sub('\@'); # will not look for an @
  _data=sub2;
}
```

The above will not find the @ inside data. The \ does not remove the special meaning of the @ inside a sub expression. The \character sequence is only evaluated in regex expressions. To use an @ inside a sub expression, use \x40 instead:

```
sub2=_data.sub('@\x40');
```

Then the part after the @ is returned by the sub expression.

```
If(_event=='decode'){
  split=_data.regex(/(.*)\/(.*)/);
}
```

In the above regex enclosed in forward slashes, the \ removes the special meaning of the forward slash inside the regex. The regex will match any data with a forward slash inside and return the data part before the first slash and the part after this first slash in _match1 and _match2.

```
If(_event=='decode'){
  split=_data.regex("(.*)\[(.*)");
}
```

The left square bracket [will normally start a regex character list or class, for example, for all digits this would be [0-9] or [0123456789] or [[:digit:]]. The back slash before the [removes this special meaning and makes the [a literal search character. The above regex will match any data with a [inside.

Applying a Filter

The filter script is part of Scanning settings.

- In Android Settings > Honeywell Settings > Scanning > Internal Scanner > (any profile) > Decode Settings > Decode Filter > Decode filter script
- In DataCollectionService.xml, the property is DEC_DECODE_FILTER

The property takes a string value. The string contains the text of the filter script.

The script content must be encoded according to XML standards and use HTML entities like “&” for an ampersand character or “<” and” >”.

Include the script in the settings file DataCollectionService.xml.

When the filter string property is applied to the scanner, it is compiled and becomes the active filter.

As with all scanning properties, the property value is part of a single wedge profile, or a single application's configuration. Changing apps changes the filter script along with the other settings.

Disabling a Filter

The decode filter timeout option can be used to disable the logic inside the filter script after a given amount of timeout. An expiration of non-zero default value will cause the scanner to revert to normal scan mode without filter logic.

Debugging a Filter

When composing a filter, you will want to get feedback on how it is interpreted by the device.

Currently, only the logcat method is available to help diagnose a filter script. Use logcat with a filter of "Decode-Filter" to get only Decode Filter debug output, for example: "logcat Decode-Filter:V *:S" or "logcat |grep Decode-Filter".

Logcat debugging method

A configuration option is available to write information to logcat about the compilation and execution of the decoder filter script. This method requires Android Debug Bridge (ADB) and familiarity with using adb logcat.

By default, no filter information is shared in logcat. During experimentation, cause runtime diagnostic information to appear in logcat using the Debug Level property. Set to 4 to emit the most information. Set to 0 to emit no information.

- In Android settings > Honeywell settings > Scanning > Internal Scanner > (any profile) > Decode Settings > Decode Filter > Debug Level
- From an application, property DEC_DECODE_FILTER_DEBUG

Script Details

Whitespace

Whitespace separates tokens but otherwise is ignored, except inside of constants.

Line Endings

Single expressions must end with a semicolon, for example:

```
var1="Hello World";
```

Function Calls

Any number of parameters to a function call is allowed.

These are equivalent:

a.fname(b, c)

fname(a, b, c)

Structure

This section provides an informal but more complete description of the script. A script is a Block, which breaks down to tokens per the following:

Block: A sequence of zero or more of:

Statement

If-Block

;

Statement:

Function ;

If-Block: one of:

if (Expression) { Block }

if (Expression) { Block } else { Block }

if (Expression) { Block } else If-Block

Expression: one of:

Function-call

Name

Constant

Name = Expression

(Expression)

! Expression

Expression Binary-Operator Expression

Binary-Operator: One of: == != && ||<>

Function-call: one of

Name (Parameter-List)

Expression . Name (Parameter-List)

Name: A character sequence containing only characters A..Z, a..z, 0..9, and _. It may not begin with 0..9.

Parameter-List: A comma-separated list of zero or more Expression.

Constant: A string of characters surrounded by single-quotes, double-quotes, or forward-slashes. See section on Constants.

Add a Decode Filter Script to a Device

You can add a decode filter script to a device in three ways:

- Create an XML file and copy it to the device.
- Create a barcode for the script using EZConfig or Enterprise Provisioner and scan it.
- Enter the script in the Decode Filter window on the device in **Settings > Honeywell Settings > Scanning > Internal Scanner > Default Profile > Decode Settings > Decode Filter**.

Refer to the user guide for your device for more information on using these features.

CHAPTER

2 SAMPLE DECODE FILTER SCRIPTS

Introduction

This section provides examples of decode filter scripts.

Reject Barcodes

This script rejects barcodes that do not begin with “1Z”.

```
if (_event == 'decode') {
   prefix = _data.sub('0', '2');
   if (prefix != '1Z') {
      _data = '';
   }
}
```

Remove “00” from Beginning of Barcode

This script checks if a barcode begins with “00”. If so, it removes the first two zeroes and transmits the rest of the barcode.

```
if (_event == 'decode'){
   prefix = _data.sub('0', '2');
   body = _data.sub('2');
   if(_aimId == ']C1' && prefix == '00') [
      _data = body;
   }
}
```

Only Scan Barcodes Beginning with “02”

This script only scans barcodes if the first two characters are “02”.

```
if (_event == 'decode'){
   prefix = _data.sub('0', '2');
   if(prefix != '02'){
      _data = '';
   }
}
```

If Barcode is GS1 128 AIM, Transmit]C1

If the barcode is GS1 128, this script will transmit “]C1” and all other characters.

```
if (_event == 'decode'){
   _body = _data.sub('0');
   if(_aimId == ']C1'){
      _data = concat(_aimId,_body);
   }
}
```

Replace GS within Barcode with Two GS Codes

This script replaces two control codes inside data by the pipe character.

```
if (_event == 'decode'){
_data.regex('([[:alnum:]]+)\x1d([[:alnum:]]+)\x1d([[:alnum:]]+)'
);
if( _aimId == ']C1' && and(_match1,_match2,_match3) ) {
  _data=concat(_aimId, _match1, "|", _match2, "|", _match3);
}
```

Scan Multiple Codes with Timeout

To scan multiple codes with one scan button press, you will need to set up the filter script for the multi-code scan. For instance, to read the desired three codes within two seconds, enter the following script:

```
if (_event == 'start') {
    hm1 = '';
    hm2 = '';
    hm3 = '';
}

if (_event == 'decode') {
    if (_data.regex(/30........../)) {
        if(hm1 == '') {
            hm1 = _data;
        }
    }
    if (_data.regex(/4.......000./)) {
        if(hm2 == '') {
            hm2 = _data;
        }
    }
    if (_data.regex(/[57]......./)) {
        if(hm3 == '') {
            hm3 = _data;
        }
    }
    _data = '';
}

if (_event == 'timeout') {
    if(_time < '2000') {
        if (and(hm1, hm2, hm3)) {
            _data = concat(hm1, '|', hm2, '|', hm3);

        }
    }
}
```

When barcodes are scanned that match the above patterns, the data is transmitted only if all patterns and conditions have been fulfilled.

The above barcodes sheet can be scanned and will fulfill the Decode Filter when the scan window (the backlight or aiming) is moved over the different barcodes. As soon as all three patterns have been scanned within a desired time, the scanner will transmit the result as a combination of the three barcodes:

301234567890|412345670001|51234567

or

301234567890|412345670001|71234567

The barcode scanner may be moved to be able to decode all barcodes and find the patterns. The output sequence is defined by the script as the separator (here a pipe symbol, |).

Honeywell
855 S. Mint St.
Charlotte, NC 28202

sps.honeywell.com