Honeywell

# Total Freedom 3.0

## Plug-In Developer's Guide

## For Xenon Ultra 196X Series Only

# User 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

# Trademarks

MIPS is a trademark of MIPS Inc.

IBM is a registered trademark of IBM in the United States.

Microsoft® Windows® and the Windows logo are trademarks or registered trademarks of Microsoft Corporation.

Apple is a trademark of Apple Inc., registered in the U.S. and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other product names or marks mentioned in this document may be trademarks or registered trademarks of other companies and are the property of their respective owners.

# Patents

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

# TABLE OF CONTENTS

# Customer Support

## Technical Assistance

To search our knowledge base for a solution or to log in to the Technical Support portal and report a problem, go to honeywell.com/PSStechnicalsupport.

For our latest contact information, see sps.honeywell.com.

CHAPTER

# 1 CONTENTS AND ENVIRONMENT

## Package Contents

### MIPS™ ELF Toolchain for Linux® PC

The MIPS ELF cross-compiling toolchain package for an IBM-compatible Linux PC.

### Plug-in Samples

Folder containing sample plug-in development projects. The Sample folder is created when you install the toolchain.

***Note:*** *Before downloading a plug-in to the scanner, you must first compile the plug-in on an IBM-compatible Linux PC using the MIPS ELF Toolchain for Linux PC.*

### EZConfig for Scanning

Used to download and debug.

## System Requirements

TotalFreedom GNU Toolchain is supported under the following system requirements:

- Processor: Minimum 2GHz
- Memory: Minimum 4 GB RAM
- Hard Drives: Minimum 50 GB free disk space
- Operating system: Utuntu 20.04 LTS 64-bit, may run on Windows® 10 64-bit + VMware 16.1.2 or later

- Software dependencies: GNU Make 4.2.1 or later, Binutils 2.34 or later, gcc 9.4.0 or later

The Toolchain has not been tested on other operating systems besides those mentioned above.

# Plug-in Development Environment

## Installing MIPS Toolchain for Linux PC

1. Log into an IBM-compatible Linux PC.
2. Copy the tarball file: PluginToolbin_Linux.tar.bz2 in the package to your development directory.
3. Untar the PluginToolbin_Linux.tar.bz2 file:

```
tar -xvf PluginToolbin_Linux.tar.bz2
```

CHAPTER

# 2 PLUG-IN DEVELOPMENT

The plug-in is a relocatable ELF file. Total Freedom plug-ins are different than a normal program running on a common operating system such as Windows and Linux.

The plug-in has no main function but does have an initial function instead of the main entry function.

The following function plug-in types can be defined: decode plug-ins, format plug-ins, and offline plug-ins. Another special kind of helper plug-in provides functions that can be called by other kinds of plug-ins. You can divide a plug-in into a functional plug-in and helper plug-in(s), which allows the sharing and changing of plug-in code and data in a modular fashion. (See 'Declare Plug-in" on page 6).

The installation package contains a format plug-in sample.

## Header Files

Use standard C library functions to develop plug-ins. To use the standard C library functions, include the standard C Library header files as you would when developing a standalone program. You also need Honeywell-defined header files to properly create plug-ins.

The following header files must be included:

```
#include <hsm_plugin/matrix_plugin.h>
```

The matrix_plugin.h header file contains the basic defines and data structures of the plug-in. It also contains hon_resolved_symbols.h which includes a list of API's that are external symbols to be resolved by the scanner application during the loading of the plugin. All plug-ins must include this header file.

```
#include <hsm_plugin/matrix_format_plugin.h>
```

The format plug-in header file contains relevant defines, data structures and API declarations. You must include this header file when creating a format plug-in.

```
#include <hsm_plugin/matrix_beep_led.h>
```

This file contains the beeper and LED control system call defines and declarations.

```
#include <hsm_plugin/matrix_bar codeid.h>
```

This header file contains Honeywell Symbology ID definitions. You must include this header file in plug-ins that work with the symbology IDs in the processed result of the scanner.

```
#include <hsm_plugin/matrix_decode_plugin.h>
```

This header file contains the decode plug-in definitions, data structures and API declarations.

```
#include <hsm_plugin/matrix_offline_plugin.h>
```

This header file contains the offline plug-in definitions, data structures and API declarations.

***Note:*** *These header files are built into the toolchain. Include them as follows:*

```
#include <hsm_plugin/HEADER_FILE.h>
```

# Plug-in Chain

Honeywell scanners provide a chain function that allows you to compact multiple plug-ins into a plug-in chain MOCF file. The data can be handled by this plug-in chain. The output data from the previous plug-in is passed to the next plug-in as the input data. The system routine should be a special plug-in and enabled by default. Call plug-ins and the system routine in the order of their appearance in the XML configuration file.

## Build Bin Files for Every Plug-in

Build every plug-in used for the plug-in chain. The "*.plugin" files generated are used to create a plug-in chain MOC file.

## Create a Plug-in Chain Configuration File

The configuration file determines whether the barcode data should be sent to plug-in routines or a system routine, and the order of the data. If there is no system routine configuration entry in the XML file, by default the scanner will call the system routine after all plug-ins have been called.

The following is an example of Plug-in Chain configuration file:

```
<?xml version="1.0"?>
<Format_PlugIn>

   < FormatPlugIn_1>
      <! Configurations for FormatPlugin_1!>
      ……
      <entrydatastate> MODIFIED </entrydatastate>
      <chainonexit> CHAINALWAYS </chainonexit>
      ……
   </ FormatPlugIn_1>

   <SystemRoutine>
      <entrydatastate> MODIFIED </entrydatastate>
      <chainonexit> CHAINIFSUCCESS  </chainonexit>
   </SystemRoutine>

   < FormatPlugIn_2>
      <! Configurations for FormatPlugin_2!>
      ……
      <entrydatastate> ORIGINAL </entrydatastate>
      <chainonexit> CHAINALWAYS </chainonexit>

   </ FormatPlugIn_2>
   ……
</Format_PlugIn>
```

In this configuration, first call FormatPlugin_1. Then, if FormatPlugin_1 parsed the input data successfully, the system routine is called and the output data from FormatPlugin_1 is passed to the system routine as input. If FormatPlugin_1 failed to parse the data, the system routine won't be called. After the system routine is treated, FormatPlugin_2 is treated according to its settings (whether it is called, what kind of input data should be passed, etc.). All the plug-ins are treated by calling the logic component in the firmware in the order of their appearance in the XML file.

## Create MOCF File that Contains Multiple Plug-ins

To create a plug-in chain MOCF file with multiple “*.plugin” files, create a MOCF file with one of the files first and then use the AppendToMocf tool to add plug-in files to the MOCF file.

```
AppendToMocf -m $(OutputFile) -t CompatProd -f
$(CompatProdRecFile) -d
AppendToMocf -m $(OutputFile) -t user -f FormatPlugIn_1.plugin
AppendToMocf -m $(OutputFile) -t user -f FormatPlugIn_2.plugin
AppendToMocf -m $(OutputFile) -t user -f ChainConf
AppendToMocf -m $(OutputFile) -t CustomDefaults -f
ChainCustomDefaults.txt
```

# Define Plug-in Information

Certain information must be built into the plug-in for it to load it properly. Define this information in the same source file with the plug-in declaration. (See 'Plug-in Chain" on page 4):

```
#define PLUGIN_NAME         SamplePlugin
#define COMPANY_NAME        Plug-In Developer, Inc.
#define MAJOR_VERSION       5
#define MONOR_VERSION       3
#define BUILD_NUMBER        37
#define CERTIFICATE         102148
#define CERTIFICATE_TIME    2010/02/02 15:00:05
#define PLUGIN_GUID         abcd1234
#define FILE_NAME           Sample.plugin
```

Note that the value of the definition should be ASCII character strings without double quotes.  Spaces and commas are permitted in the string.

# Declare Plug-in

The macro ‘DECLARE_PLUGIN(init_plugin, cleanup_plugin, plugin_type, MenuID)’ (defined in “matrix_plugin.h”) is used to declare the plug-in so the scanner can obtain information from the plug-in when it is loaded:

```
DECLARE_PLUGIN(init_plugin, cleanup_plugin, HON_PLUGIN_FORMAT,
0x01);
```

The init_plugin and cleanup_plugin correspond to the addresses of the plug-in initial function and plug-in cleanup function respectively. (See 'Decode Plug-In API" on page 33).

The plug-in class type is defined as:

```
enum HONPluginClassType
{
   HON_PLUGIN_TYPE_UNKNOWN = 0,
   HON_PLUGIN_FORMAT,
   HON_PLUGIN_DECODE,
   HON_PLUGIN_OFFLINE
};
```

MenuID is the identifier that scanners use to identify different plug-ins when they pass menu barcodes to plug-ins. (See 'ProcessingBarcode" on page 27).

# Export Symbols for Other Plug-ins

You can reference symbols that are defined in other functions. This helps you divide plug-ins into parts so that you can upgrade specific parts of the plug-in while keeping the rest of the plug-in unchanged.

To export a symbol to other plug-ins, use the macro 'EXPORT_SYMBOL(symbol)':

```
int HelloWorld(void)
{
   printf("Hello World Symbol\r\n");
   return 0;
}
EXPORT_SYMBOL(HelloWorld);
```

***Note:*** *You must define the plug-in that exports symbols for other plug-ins to call before defining any other plug-ins that will call the exported symbols in the plug-in configuration file. Otherwise, the loading of plug-ins will fail.*

# Define Plug-in Entry Function and Exit Function

Since the plug-in is not an executable program binary, it does not have "main" function. Instead, it contains an entry function and an exit function. The entry function is called when the plug-in is loaded to the initial plug-in and register plug-in APIs. You may need the exit function to clean up the plug-in contents when removing it.

Define your plug-in entry and exit functions by using the following prototypes:

```
int init_plugin(HONPluginRawInfo *plugin);
void cleanup_plugin(void);
```

The definitions of the entry function and exit function in HelloWorld.c are:

```
int init_plugin(HONPluginRawInfo *plugin)
{
/* This is a Hello World plug-in sample and you can add what
you want here */
   printf("/*****************************/\r\n");
   printf("      Hello World Plug-in \r\n");
   printf("/*****************************/\r\n");

   return 0;
}

void cleanup_plugin(void)
{
   return;// Do nothing
}
```

The entry function registers plug-in APIs. (See 'Register APIs" on page 29).

# Makefile

The sample plug-in projects provide a frame structure of the Makefile for creating your own plug-in(s). You can easily generate a Makefile by modifying the sample Makefile. The Makefile template is updated to support generating the MOCF file.

```
PREFIX = /home/honeywell/sandbox/my_sdk/PluginToolbin

CFLAGS = -march=mips32r2 -mabi=32 -mfp64 -Wall -Werror -Wundef -
fomit-frame-pointer -Wstrict-prototypes -Wno-trigraphs -fno-
strict-aliasing -fno-common -I${PREFIX}/mipsel-buildroot-linux-
gnu/include

LDFLAGS = -L${PREFIX}/mipsel-buildroot-linux-gnu/lib -L${PREFIX}/
lib/gcc/mipsel-buildroot-linux-gnu/9.3.0

ASFLAGS = -march=mips32r2

# Build Tools
AS = $(PREFIX)/bin/mipsel-buildroot-linux-gnu-as
CC = ${PREFIX}/bin/mipsel-buildroot-linux-gnu-gcc
LD = ${PREFIX}/bin/mipsel-buildroot-linux-gnu-ld
STRIP = ${PREFIX}/bin/mipsel-buildroot-linux-gnu-strip
APPENDMOC = ${PREFIX}/Tools/AppendToMocf
COMPATPODFILE = ${PREFIX}/Tools/AppCompatProd.txt
COMPATPODFILERF = ${PREFIX}/Tools/AppCompatProdRF.txt

#
# User defined fields
# Modify 'BINNAME' to define the name of the plug-in output
# Modify 'OBJS' to define list of object file names
#

#
BINNAME = Format
OBJS = format_plugin_sample.o

# Targets
all: moc

.PHONY: moc

moc:$(BINNAME).moc $(BINNAME)_RF.moc

$(BINNAME).moc: $(BINNAME).plugin $(BINNAME)Conf
   $(APPENDMOC) -m $@ -t CompatProd -f $(COMPATPODFILE) -d
   $(APPENDMOC) -m $@ -t user -f $(BINNAME).plugin
   $(APPENDMOC) -m $@ -t user -f $(BINNAME)Conf
```

```
$(BINNAME)_RF.moc: $(BINNAME).plugin $(BINNAME)Conf
   $(APPENDMOC) -m $@ -t CompatProd -f $(COMPATPODFILERF) -d
   $(APPENDMOC) -m $@ -t user -f $(BINNAME).plugin
   $(APPENDMOC) -m $@ -t user -f $(BINNAME)Conf

$(OBJS): %.o: %.c

$(BINNAME).plugin: $(OBJS)
   $(LD) -r $(LDFLAGS) -o foo.bin $(OBJS) -lc -lmatrix -lgcc
   $(STRIP) -g -o $@ foo.bin
   rm -rf foo.bin

clean:
   -rm -f *.o *~ $(BINNAME).plugin *.moc
```

You can use the Makefile template to build a MOCF file of a single plug-in along with a configuration file. To put multiple plug-ins in one MOCF file, use the tool “AppendToMocf” and the product-compatible file “AppCompatProd.txt”, located in the Tools folder where the toolkits were installed. Put the plug-ins that you want to add to the MOCF file and the plug-in configuration file in the same folder and use following commands:

```
$(ToolKitsInstallDir)/Tools/AppendToMocf -m Plugin.moc -t
CompatProd -f ${ ToolKitsInstallDir }/Tools/AppCompatProd.txt
-d
$(ToolKitsInstallDir)/Tools/AppendToMocf -m Plugin.moc -t user
-f a.plugin
$(ToolKitsInstallDir)/Tools/AppendToMocf -m Plugin.moc -t user
-f b.plugin
$(ToolKitsInstallDir)/Tools/AppendToMocf -m Plugin.moc -t user
-f c.plugin
……
$(ToolKitsInstallDir)/Tools/AppendToMocf -m Plugin.moc -t user
-f PluginConf
```

In this example, “Plugin.moc” is the MOCF file that contains all plug-ins and a configuration file. The files “a.plugin”, “b.plugin” and “...plugin” are plug-ins you want to put in the MOCF file. “PluginConf” is the plug-in configuration file.

# Build Plug-in

To build a plug-in, go to source directory for the plug-in and type:

```
make
```

To clean the project and remove all earlier compiled objects, go to the source directory and type:

```
make clean
```

In a new build environment, the plug-in configuration file must be in the same folder with the source code and must be named "$(BINNAME)Conf" so that Makefile can invoke tools to generate a MOCF file containing plug-in and configuration files. You can modify your configuration file and type "make moc" to generate a new MOCF file without re-compiling the plug-in:

```
make moc
```

# Plug-in Configuration File

The plug-in configuration file helps plug-ins load properly. A plug-in configuration file controls the operation of each plug-in class. You must assign a single configuration file to a plug-in class for the system to execute plug-ins in that class.

The configuration file format conforms to XML version 1.0. The parser on the device will parse the format as described below and will not necessarily be fully XML compliant. Note that inserting comments is optional. When this configuration file is reported to the host, the comments will remain. The items (not including comments) in the configuration file are as follows, with each item defined in the order of appearance.

1. <?xml version="1.0"?>
2. One of the following must appear, depending on the plug-in class. Each file must contain only one plug-in class configuration: <Decode_PlugIn>, <Format_PlugIn>, or <Offline_Plugin>.

***Note:*** *Each plug-in within a class requires a unique identifier. Immediately following this identifier are all the definitions associated with that particular plug-in. The identifier does not need to match plug-in file names or the name you assigned the plug-in. This identifier is used solely for reference within this document, both to demarcate all definitions associated with the plug-in and to allow plug-in definitions within this file to reference one another, such as for passing control from one plug-in to another. When you create this file, ensure that all plug-in identifiers are unique. The plug-in identifier must appear in the form <plug-in identifier>, using a unique identifier as described above.*

*If two or more plug-ins are defined with the same tag name, the scanner will load the one that is defined first in the configuration file and ignore the other(s) without reporting an error.*

a. You can include the name you assigned to the plug-in within the configuration file. The name is overridden by any value obtained from the plug-in when it is loaded. This name is always output in reports to the host. The format is as follows:

```
<name>
Plug-in name string
</name>
```

b. You can include the company name of the plug-in within the configuration file. It is overridden by any value obtained from the plug-in when it is

loaded. This company name is always output in reports to the host. The format is as follows:

```
<companyname>
Company name string
</companyname >
```

c. You can also include the license status of the plug-in within the configuration file. License status is reported with a value of YES or NO. This field is ignored as an input in the configuration file, as the plug-in itself is responsible for determining license status. This license status is always output in reports to the host. The format is as follows:

```
<licensed>
YES or NO
</licensed>
```

d. The following definition determines whether or not the plug-in is to be used. It contains a value of YES or NO. If not specified, the default value is YES. You can use NO in those instances when a plug-in resides on the barcode reader for future or alternate use but is not to be used in the present configuration. The format is as follows:

```
<active>
YES or NO
</active>
```

***Note:*** *To deactivate a plug-in, set the field of "active" in the configuration file to "NO". The inactive plug-ins are not loaded. There is no error beep to indicate that the inactive plug-ins were ignored.*

e. The plug-in major revision string, which you assign, can be included in the configuration file. It is overridden by any value obtained from the plug-in when it is loaded. The major revision string is always output in reports to the host. The format is as follows:

```
<majorrevision>
Major revision string
</majorrevision>
```

f. The plug-in minor revision string that you assign can also be included in the configuration file. It is overridden by any value obtained from the plug-in when it is loaded. The minor revision string is always output in reports to the host. The format is as follows:

```
<minorrevision>
Minor revision string
</minorrevision>
```

g. The plug-in build number string that you assign can be included in the configuration file. It is overridden by any value obtained from the plug-in when it is loaded. The build number string is always output in reports to the host. The format is as follows:

```
<build>
Build number string
</build>
```

h. The plug-in certificate number string supplied by Honeywell can be included in the configuration file. It is overridden by any value obtained

from the plug-in when it is loaded. The certificate number string is always output in reports to the host. The format is as follows:

```
<certificate>
Certificate number string
</certificate>
```

i. The plug-in certificate time stamp string supplied by Honeywell can be included in the configuration file. The format of the string is "YYYY/MM/DD HH:MM:SS". The string is overridden by any value obtained from the plug-in when it is loaded. The certificate time stamp string is always output in reports to the host. The format is as follows:

```
<certificatetime>
Certificate time stamp string
</certificatetime>
```

j. The GUID string supplied by Honeywell can be included in the configuration file. It is overridden by any value obtained from the plug-in when it is loaded. The GUID string is always output in reports to the host. The format is as follows:

```
<guid>
Certificate number string
</guid>
```

k. The following value defines the name of the plug-in binary file. This field is mandatory. The format is as follows:

```
<filename>
File name string
</filename>
```

***Note:*** *The PlugInFileName must be only the file name without any path.*

l. The following optional field defines whether the plug-in's main processing function (including the main process function since it will treated as a special plug-in) should receive original data (ORIGINAL), or data as it was modified by the last plug-in in the plug-in chain (MODIFIED). If not specified, the default is ORIGINAL. The format is as follows:

```
<entrydatastate>
ORIGINAL or MODIFIED (BOTH)
</entrydatastate>
```

m. The following optional field defines how to chain the plug-in based on the exit criteria from this plug-in's main processing function. Parameter values are as follows:

| | |
|---|---|
| CHAINALWAYS | Always chain to the next plug-in, regardless of exit criteria. |
| CHAINIFSUCCESS | Chain only if the plug-in exit state indicates success. |
| CHAINIFFAILURE | Chain only if the plug-in exit state indicates failure. |
| CHAINNEVER | Don't chain at all, regardless of the plug-in exit state. |

If not specified, the default is CHAINALWAYS. The format is as follows:

```
<chainonexit>
CHAINALWAYS or CHAINIFSUCCESS or CHAINIFFAILURE or
CHAINNEVER
</chainonexit>
```

n. A matching terminator for each plug-in identifier must follow all the definitions for that plug-in. The plug-in terminator must appear in the form </plug-in identifier>, where the plug-in identifier is the same used at the start of the plug-in definition.

3. One of the following must appear, depending on the plug-in class: </Decode_PlugIn>, </Format_PlugIn>, or </Offline_Plugin>.
4. The following is a sample format plug-in configuration file, assuming the following criteria:
   - Configuration file name = FormatPlugIn_conf.
   - Menu setting PLGFON is set to "FormatPlugIn_conf".
   - Identifier = SampleFormatPlugIn_1
   - Developer assigned name = SampleFormatPlugIn
   - Developer assigned company name = Plug-In Developer, Inc.
   - Major revision = 5
   - Minor revision = 3
   - Build number = 37
   - Certificate number = 102148 dated 2009/08/10 15:00:05
   - No GUID defined
   - Binary file name = FormatPlugIn.bin
   - This plug-in takes modified data, rather than original data, as its input.

- Chain to the next plug-in if this plug-in fails:

```
<?xml version="1.0"?>
< ! --- Should be Format_Plugin since currently we only
support format plug-ins --- !>
<Format_PlugIn>
   < ! --- Plug-in ID name. Should conforms to plug-in
   filename currently --- !>
   < SampleFormatPlugIn_1>
      <name> SampleFormatPlugIn </name>
      <company> Plug-In Developer, Inc. </company>
      <licensed> YES </licensed>
      <active> YES </active>
      <majorrevision> 5 </majorrevision>
      <minorrevision> 3 </minorrevision>
      <build> 37 </build>
      <certificate> 102148 </certificate>
      <certificatetime> 2009/08/10 15:00:05 </
      certificatetime>
      <guid></guid>
      <filename> FormatPlugIn.bin </filename>
      <entrydatastate> MODIFIED </entrydatastate>
      <chainonexit> CHAINIFFAILURE </chainonexit>
   </ SampleFormatPlugIn_1>
</Format_PlugIn>
```

# Configurations of System Routines

Each class of plug-ins has routines to provide functions. The scanner also has routines that provide functions, called system routines. System routines are enabled and called after all plug-in routines have been called.

You can disable/enable system routines by editing the plug-in configuration file. To do so, add a special plug-in entry in the XML configuration file. The entry name **MUST** be “SystemRoutine”. There are two sub-entries available in this entry: “entrydatastate” and “chainonexit”.

Configuration files without any explicit system routine definitions are also supported. If there is no system routine configuration entry in the XML file, by default the scanner will call the system routine after all plug-ins have been called.

If you do not want the system routine to parse the input data, set the tag “chainonexit” to CHAINNEVER, which means the system routine is not called in the calling sequence.

If there is no system routine configuration in the configuration file, the default settings are used. The default settings for system routine are “entrydatastate” – MODIFIED and “chainonexit” – CHAINALWAYS. Then, regardless of whether or not the plug-in parsed the decode result data, the system routine is always called and will receive the output data from the plug-in and treat the received data as input.

The following is an example of disable/enable system routines:

```
<?xml version="1.0"?>
<Format_PlugIn>
<SystemRoutine>
      <entrydatastate> MODIFIED </entrydatastate>
<chainonexit> CHAINALWAYS/ </chainonexit>
</SystemRoutine>

   < FormatPlugIn_1>
      <! Configurations for FormatPlugin_1!>
      ……
</ FormatPlugIn_1>

< FormatPlugIn_2>
      <! Configurations for FormatPlugin_2!>
      ……
</ FormatPlugIn_2>

……

</Format_PlugIn>
```

# Memory and Storage for Plug-ins

## File Size of Plug-in

The file size of plug-ins is limited to 2 MB. If the total size of your plug-ins and all the files they generate during runtime reaches 2 MB, you cannot download any more plug-ins to the scanner.

***Note:*** *The total file size for the 8680iB is 54 MB and for the 8690i is 188 MB.*

The size of a single plug-in is limited to 2 MB. If you try to download a plug-in larger than 2 MB, the download will fail. In addition, if the plug-in debug setting is turned on (by sending menu command “PLGDBG1.” to the scanner), you will receive download failure information from the scanner.

## Stack Size

The size of stack for plug-ins is limited to 200K bytes. Therefore, you cannot define local variables larger than 200K bytes.

## Global Variables

If any global variable is not initialized in the plug-in, the memory for the global variable is allocated dynamically during loading time. The size for global variables in your plug-in is limited to 1 MB. Therefore, do not define global variables with initialization larger than 1 MB.

## Heap

The heap size for plug-ins is 1 MB. Standard library functions such as malloc, free, calloc and realloc are supported. If you try to allocate memory larger than 1 MB, the memory allocate functions (malloc, calloc and realloc) will fail.

# Create an MOCF file with Plug-ins

Convert or merge the plug-ins and plug-in configuration files to the "MOCF" file container before downloading the plug-in to a scanner. A scanner will not accept a binary plug-in file. Use the "AppendToMocf" tool to create an MOCF file. This tool is located in the folder $PluginDevToolInstallDir/Tools. In the same folder there are two compatible product record files (AppCompatProd.txt and AppCompatProdRF.txt), which you can use to generate MOCF files for corded and cordless scanners.

***Note:*** *There are examples for creating an MOCF file in the Makefile of the sample code. You can use the example Makefile in the sample code of the toolchain as reference to create your own Makefile.*

## Create an MOCF File that Contains a Single File

To create an MOCF file that only contains one file, use the shell commands:

```
AppendToMocf –m $(OutputFile) –t CompatProd –f
$(CompatProdRecFile) -d
AppendToMocf –m $(OutputFile) –t user –f $(PluginFile)
```

***Note:*** *Create the MOCF file name $(OutputFile). $(CompatProdRecFile) is the compatible product record file name. If you want the plug-in to be applied to corded scanners, set $(CompatProdRecFile) to AppCompatProd.txt, otherwise, use AppCompatProdRF.txt. $(PluginFile) is the plug-in binary file or plug-in configuration file that you must add to the MOCF.*

## Create an MOCF File that Contains Multiple Files

You can create an MOCF file with multiple files. Once an MOCF file is created, use the AppendToMocf tool to add more files to the MOCF file.

```
AppendToMocf –m $(OutputFile) –t CompatProd –f
$(CompatProdRecFile) -d
AppendToMocf –m $(OutputFile) –t user –f $(PluginFile1)
AppendToMocf –m $(OutputFile) –t user –f $(PluginFile2)
AppendToMocf –m $(OutputFile) –t user –f $(PluginFile3)
……
AppendToMocf –m $(OutputFile) –t user –f $(PluginFilen)
```

## Add Custom Defaults File to Plug-in MOCF file

Custom defaults files can be downloaded to a scanner for special uses. To add a custom defaults file to an MOCF file, use the following shell command:

```
AppendToMocf –m $(OutputFile) –t CustomDefaults –f
$(DefaultsFile)
```

# Download Plug-in and Configuration File

You can download the plug-in and configuration file to a scanner using EasyConfig software. Connect the scanner to EZConfig for Scanning. Click on the **Download** selection. Under **Firmware Download**, use the … button to browse to the MOCF file name. Click on **Download to Device**.

You may also use the **Scan Data** selection to send the command “PLGDIR” to verify that your files have saved to the scanner correctly.

CHAPTER

# 3 CONFIGURATION FILE SAMPLES

## FormatPlugin_1 and FormatPlugin_2

In the following configuration, FormatPlugin_1 is called and then, if FormatPlugin_1 parsed the input data successfully, the system routine is called and the output data from FormatPlugin_1 is passed to the system routine as input. If FormatPlugin_1 failed to parse the data, the system routine is not called. After system routine is treated, FormatPlugin_2 is treated according to its settings (to be called or not, what kind of input data should be passed, etc.). All the plug-ins are treated by the calling logic component in the firmware in the order of their appearance in the XML file.

```
<?xml version="1.0"?>
<Format_PlugIn>
   < FormatPlugIn_1>
      <! Configurations for FormatPlugin_1!>
      ……
</ FormatPlugIn_1>

<SystemRoutine>
      <entrydatastate> MODIFIED </entrydatastate>
<chainonexit> CHAINIFSUCCEED </chainonexit>
</SystemRoutine>

< FormatPlugIn_2>
      <! Configurations for FormatPlugin_2!>
      ……
</ FormatPlugIn_2>

< FormatPlugIn_3>
      <! Configurations for FormatPlugin_3!>
      ……
</ FormatPlugIn_3>

</Format_PlugIn>
```

# Call the System Routine

In the next example, the system routine is called whether or not FormatPlugIn_1 parsed the input data. The system routine will always use the original data as input (the data which was not treated by FormatPlugIn_1).

```
<?xml version="1.0"?>
<Format_PlugIn>
   < FormatPlugIn_1>
      <! Configurations for FormatPlugin_1!>
      ……
</ FormatPlugIn_1>

<SystemRoutine>
      <entrydatastate> ORIGINAL </entrydatastate>
<chainonexit> CHAINALWAYS </chainonexit>
</SystemRoutine>

< FormatPlugIn_2>
      <! Configurations for FormatPlugin_2!>
      ……
</ FormatPlugIn_2>

< FormatPlugIn_3>
      <! Configurations for FormatPlugin_3!>
      ……
</ FormatPlugIn_3>

</Format_PlugIn>
```

# Disable Calling the System Routine

In the next example, the system routine is not called at all. This case disables calling the system routine.

```
<?xml version="1.0"?>
<Format_PlugIn>
   < FormatPlugIn_1>
      <! Configurations for FormatPlugin_1!>
      ……
</ FormatPlugIn_1>

<SystemRoutine>
      <entrydatastate> MODIFIED </entrydatastate>
<chainonexit> CHAINNEVER </chainonexit>
</SystemRoutine>

< FormatPlugIn_2>
      <! Configurations for FormatPlugin_2!>
      ……
</ FormatPlugIn_2>

< FormatPlugIn_3>
      <! Configurations for FormatPlugin_3!>
      ……
</ FormatPlugIn_3>

</Format_PlugIn>
```

# System Routine at End of Plug-In Sequence

In the example above, there is no system routine configuration entry in the XML file. So the system routine is put at the end of the plug-in calling sequence. In other words, the system routine is called by default after all the plug-ins have been processed, whether or not the last plug-in parsed data, and will take the data parsed by all the plug-ins as input data.

```
<?xml version="1.0"?>
<Format_PlugIn>
   < FormatPlugIn_1>
      <! Configurations for FormatPlugin_1!>
      ……
</ FormatPlugIn_1>

< FormatPlugIn_2>
      <! Configurations for FormatPlugin_2!>
      ……
</ FormatPlugIn_2>

< FormatPlugIn_3>
      <! Configurations for FormatPlugin_3!>
      ……
</ FormatPlugIn_3>

</Format_PlugIn>
```

CHAPTER

# 4 GENERATE MENU BARCODES

Each plug-in has a unique ID assigned to it. The ID is used to generate a menu barcode for the plug-in. Plug-in menu codes are generated using either a Normal, or a Lock-Mode method.

## Normal

Using the normal method, conform to the following format when you generate menu barcodes:

990XYYYYYDATA

"990" is a fixed prefix for plug-ins, "X" stands for plug-in types (0 for decode, 2 for format), "YYYYY" stands for a five-digit hexadecimal ID number, and DATA is the menu data that is sent to the plug-in. The "990XYYYYY" prefix is stripped off before the menu code is sent to the plug-in.

The programming barcode data can also be sent as a menu command to the scanner so that the plug-in can be configured that way. This only applies to the Normal method.

## Lock-Mode

Using the Lock-Mode method, you can scan an Enter barcode to lock the plug-in when you want to configure the plug-in via menu barcodes.

The format of the enter code is:

990XEntYYYYY

"990" is a fixed prefix for plug-ins, "X" stands for plug-in types (0 for decode, 2 for format), Ent indicates this is a lock-mode enter code, and "YYYYY" stands for a five-digit hexadecimal ID number.

Only one plug-in can be locked at a time. Once the plug-in is locked, all menu codes scanned are passed to the plug-in directly by calling the BarcodeProcessing API. Scanning data codes will cause the device to issue an error when a plug-in is locked. To exit the lock-mode, scan an Exit menu barcode.

If the scanner is in lock-mode, generate menu barcodes that conform to the format 990XDATA ("990" is a fixed prefix for the plug-in menu and "X" stands for plug-in types). When one of these codes is scanned, the prefix 990X is stripped off and DATA is passed to the locked plug-in if the locked plug-in is the type indicated by X.

The format of an Exit code is:

99Exit

There is an exception for scanning menu codes when the scanner is in lock-mode. If you scanned a menu code conforming to the format of the specific menu codes used in the normal way (990XYYYYYDATA), the scanner will strip off the "990XYYYYY" header and then pass the "DATA" to the plug-in.

***Note:*** *The helper plug-ins do not provide any API to the device, and they do not need any identifier. You must define the Macro "MenuID" to "-1", which is ignored.*

CHAPTER

# 5 FORMAT PLUG-IN API

## DataEdit

DataEdit is the main routine for formatting plug-ins. This API is called when the output string must be formatted before being sent out.

**Function prototype:**

```
int                                 /* Return zero on success, -1 if an error
                                    occurred  */

(*DataEdit)(DataEditParam           /* Input: Format parameters structure */
*format_param);
```

The parameter type “DataEditParam” is defined as:

```
typedef struct {
   // Revision number
   int RevisionNumber;
   // Input and Output Data. Note that input data could be byte
   wide or word wide. depends on the value of CharSize.
   #ifdef MATRIXPLUGIN_DATAEDITPARAM_MESSAGE_8BIT
      unsigned char *message;
   #else
      short *message;
   #endif
   // Number of Data Characters
   int length;
   //Character size (1 for byte, 2 for word)
   int CharSize;
   // Hand Held Products internal Code (Symbology) ID
   char HHPcodeID;
   // AIM/FACT/ISO "Symbology Identifier"
   char AIMcodeLetter;
   // ... and "Modifier" character
   char AIMcodeModifier;
} DataEditParam;
```

***Note:*** *The “message” member field of structure DataEditParam contains the passed-in data string. You must put the formatted data string back to “message” buffer. The length of the formatted data string must not exceed the length of the original string by more than 500 bytes, otherwise it will cause an overflow.*

The function returns -1 if an error occurred or formatting failed. If the format processing is successful, the function returns zero to indicate success, and restores the processed string to the “message” field in the input structure.

Below is an example of DataEdit API. This API of the plug-in simply adds the prefix “Code128*” and applies it to all the Code 128 barcodes.

```
/** This API is called to perform a data format.
 *  The plug-in developer should implement this
 *  routine by himself and set address of this
 *  function to the "DataEdit" field of the
 *  "DataEditApi" structure.
 */
int MatrixPluginDataEdit(DataEditParam *pFormatParam)
{
   // Add your Format code here and copy the result back to
   pFormatParam->message.
   unsigned char *buffer = NULL;
   unsigned short WidePrefix = {‘C’, ‘o’, ‘d’, ‘e’, ‘1’, ‘2’, ‘8’,
   ‘*’};

   // if not Code 128, just return -1
   if(pFormatParam->HHPcodeID != WA_CODELETTER_CODE128){
      return -1;
   }else{
      printf("This is Code128\r\n");
   }
   buffer = malloc((pFormatParam->length + 100 )*(pFormatParam-
   >CharSize));
   if(!buffer)
      return -1;
   if(pFormatParam->CharSize == 1){
      memcpy(buffer, “Code128*”, 8);
      memcpy(buffer+8, pFormatParam->message, (pFormatParam-
      >length)*(pFormatParam->CharSize));
      }else if(pFormatParam->CharSize == 2){
      memcpy(buffer, WidePrefix, 16);
      memcpy(buffer+16, pFormatParam->message, (pFormatParam
>length)*(pFormatParam->CharSize));
   }
   // Set length after data format
   pFormatParam->length += 8;
   free(buffer);
   return 0;
}
```

# ProcessingBarcode

This function is used to process specific user-defined programming barcodes.

**Function prototype:**

```
int                          /* Return zero on success, -1 if an
                             error occurred  */
(*ProcessingBarcode)(
char *pMenuData,             /* Input: Pointer of menu code data
                             */
int DataLength);             /* Input: Data length */
```

The function returns -1 if an error occurred or processing failed. If the programming barcode is processed successfully, the function returns zero to indicate success.

# CheckLicense

This function is used to validate the license of the plug-in.

**Function prototype:**

```
int                          /* Return zero on success, -1 if an
                             error occurred  */
(*CheckLicense)(
char *SN);                   /* Product serial number */
```

The product serial number is passed to the function as a null-terminated string of characters. The function must return 0 if the license is valid or -1 if not.

***Note:*** *This function is called after the plug-in is loaded and initialized. Plug-ins should keep the result of the CheckLicense function during the runtime of the plug-in, and should use the result to determine if the other APIs (for example, ProcessingBarcode) can be called or not (by returning 0 or -1 when the API is called).*

Setting up a license check mechanism requires two parts: license check and license file generation.

To generate a license file, create a data string from the serial number of the scanner using your own encryption algorithm. You could make a programming barcode (the programming barcode should conform to the plug-in programming barcode format “990XYYYYYDATA”) based on this data string. Add code in the API “ProcessingBarcode” so that after the license programming barcode is scanned, a license file can be generated in the scanner. Typically in “ProcessingBarcode”, to support licensing you must:

1. Distinguish a programming barcode for licensing.
2. Decrypt the passed in data of the programming barcode.
3. Extract the serial number from the Decrypt data and compare it with the serial number of the scanner.

4. If the serial number from decrypted data is the same as the serial number of scanner, create a license file in the scanner to contain the license information of the plug-in.

You can use a group ID method to implement your license check mechanism so that you do not need to generate programming barcodes for every scanner. A group ID is the identifier assigned to scanner groups, and it resides in the scanner. If the value of the group ID is 0, then the scanner does not have a group ID. For scanners with the same group ID, you can create a programming barcode to generate the license file. The CheckLicense function may be called twice if the scanner has a group ID. The plug-in must remember both the passed-in serial number and group ID during the runtime.

# Register APIs

The register API function is called to register APIs of the plug-in so they can be called by scanner applications. It returns zero for success and -1 for error.

**Function prototype:**

```
int                         /* Return zero on success, -1 if an
                               error occurred  */
register_apis(
void *Plugin,               /* Plugin object */
void *APIS);                /* API structure pointer */
```

The plug-in object structure type is defined as:

```
typedef struct{
char PluginRawName[PLUGIN_ID_LEN];      /* Raw name in plugin
                                           binary */
enum HONPluginClassType                 /* Raw Class Type in
PluginRawClassType;                        plguin binary */
int (*PluginInitRoutine)(void *Info);   /* Startup function.
                                           */
void (*PluginExitRoutine)(void);        /* Destruction
                                           function. */
void *PluginApis;                       /* Plugin APIs. This
                                           should be in this
                                           structure in order
                                           that the plugin could
                                           assign APIs' address
                                           here */
```

```
int MenuIdentifier;                    /* This field is the
                                       identifier assigned
                                       from Hoenywell. Menu
                                       codes with the
                                       identifier prefix are
                                       passed to  * the
                                       corresponding plug-in
                                       */
/* Other plugin infos */
char CompanyName[PLUGIN_STRING_LEN];
char
MajorVersionNumber[PLUGIN_STRING_LEN];
char
MinorVersionNumber[PLUGIN_STRING_LEN];
char BuildNumber[PLUGIN_STRING_LEN];
char
CertificateNumber[PLUGIN_STRING_LEN];
char
CertificateTime[PLUGIN_STRING_LEN];
char GUID[PLUGIN_GUID_LEN];
char FileName[PLUGIN_STRING_LEN];
} HONPluginRawInfo;
```

The API structure type is defined as:

```
typedef struct
{
   // Revision Number. It is used for Plug-in API forward
   compablity
   int RevisionNumber;
   // Format API callback
   int (*DataEdit)(DataEditParam *pFormatParam);
   // Plug-in Menuing API callback
   int (*ProcessingBarcode)(char *pMenuData, int DataLength);
   // Check license API callback
   int (*CheckLicense)(char *SN);
   // Get version API callback
   int (*GetVersion)(VersionInfo *pInfo);
} DataEditApi;
```

# Control the Scanner's Beeper and LED

This function is a system call to control the scanner's beeper and LED. Control of the LED is bound with the beeper, and the plug-in can control the beeper and LED by calling one system call:

**Function prototype:**

```
int /* return -1 for failure and return 0 for success */
beep_led_io(
unsigned int const* pBeepSeq, /* Input: the beeper/LED control
entry sequence */
unsigned int SeqLen);/* Input: length of the control sequence */
```

The beeper/LED control entry sequence is an array of integers starting with an audible LED sync (defined in "matrix_beep_led.h"). Three types of LEDs are defined: good read flash, error flash, and no LED. The following integers stand for entries of the sequence. The rules for the entries are:

- Each sequence should start with an audible LED sync (LED_DEFINE) and end with a terminator (0x00).
- The odd entries of the sequence are duration in heartbeats; the even entries are the frequency (0 is a rest).
- Frequency 100 or above is an audible sound (provided the beeper can create the sound).
- Frequency 100 or above will use the LED specified at the first char of the sequence.
- For each silent pause, use one of the LED defines as the frequency.

**Example:** This sequence can be read as 10mS sound at 200Hz with no led, then 10mS silence with no LED:

```
unsigned int ExampleSeq[] = {audible LED sync (LED_DEFINE),
duration of next freq (mS), audible freq (Hz), duration (mS),
silent freq (LED_DEFINE), end of string (0x00)};


unsigned int StandardClickSeq[] = {NO_LED, 10, 200, 10, NO_LED,
0x00};
```

**Example:** The beeper duration has a resolution of 10ms.

```
unsigned int StandardBeepSeq[]={LED Synchronized?,mS
(duration),frequency hz,end of string (0x00)};
```

***Note:*** *For reference, read the header file "matrix_beep_led.h"*

# Play Audio Files

This function is a system call to play audio files. The Makefile needs to be updated to include the .wav files. An example is shown below:

```
$(BINNAME).moc $(BINNAME)_RF.moc
$(BINNAME).moc: $(BINNAME).plugin $(BINNAME)Conf
$(APPENDMOC) -m $@ -t CompatProd -f $(COMPATPODFILE) -d
$(APPENDMOC) -m $@ -t user -f $(BINNAME).plugin
$(APPENDMOC) -m $@ -t user -f userid
$(APPENDMOC) -m $@ -u user -f <Audio File Name>.wav
```

After the audio file is included the audio file can be played through the plug-in using MATRIX_AUDIO system command. The parameter is the audio filename. An example is shown below:

```
struct matrix_syscall_param pParam;
pParam.syscall_id = MATRIX_AUDIO;
pParam.params[0] = (int *) <Audio character array pointer>;
matrix_syscall(&pParam);
```

CHAPTER

# 6 DECODE PLUG-IN API

## Logic of Calling Decode Plug-ins

The order of calling decode plug-ins is controlled by the configuration file but should also conform to the scanner's internal logic. The captured image is sent to the system decode routine first to detect programming barcodes. If it is a programming barcode, the barcode decode is processed directly and the image is not passed to any plug-ins. If the image is not a programming barcode, plug-ins and the system decode routine is called to decode the image according to the order in the configuration file.

Once the image is recognized and decoded, the decoding process is stopped. The plug-ins configured to be invoked after the current plug-in are ignored even if they are configured as CHAINALWAYS or CHAINIFSUCCEED. Therefore, you must determine the order of calling plug-ins and define the proper configuration file to ensure the plug-ins can be called. For instance, if there are two plug-ins in the plug-in chain and both of them can recognize and decode the same type of barcodes, you must define the plug-in that you want to use to decode that type of barcodes before defining the second plug-in.

## Decode Plug-in APIs

### Decode

This API, which is the main routine for a decode plug-in, is called to decode the image captured by a scanner.

**Function prototype:**

```
int                          /* Return zero on success, -1 if an
                             error occurred  */
  (*Decode)(
unsigned char *pBuffer,      /* Input: Pointer to image buffer */
```

```
int width,                  /* Input: image width */
int height);                /* Input: image height */
```

When the decode processing succeeds, the function returns zero. The function returns -1 if an error occurs or decode failed.

## ProcessingBarcode

The usage of this function is the same as the format plug-in. (See "ProcessingBarcode" on page 27).

**Function prototype:**

```
int                         /* Return zero on success, -1 if an
                            error occurred  */
 (*ProcessingBarcode)(
char *pMenuData,            /* Input: Pointer of menu code data
                            */
int DataLength);            /* Input: Data length */
```

When the programming barcode is processed successfully, the function returns zero. The function returns -1 if an error occurs or processing failed.

## CheckLicense

The usage of this function is the same as the format plug-in. See "CheckLicense" on page 28.

**Function prototype:**

```
int                         /* Return zero on success, -1 if an
                            error occurred  */
 (*CheckLicense)(
char *SN);                  /* Product serial number */
```

The product serial number is passed to the function as a null-terminated string of characters. The function returns 0 if the license is valid or -1 if not.

## CheckVersion

**Function prototype:**

```
int                         /* Return zero on success, -1 if an
                            error occurred  */
 (*CheckVersion)(
```

```
VersionInfo *Info);        /* Plug-in version info structure */
```

The version information type “VersionInfo” is defined as:

```
typedef struct {
 int RevisionNumber;       /* Revision number */
 char *GUID;               /* GUID of the plug-in */
 char *PluginName;         /* Plug-in name */
 char *CompanyName;        /* Company name of the plug-in */
 int MajorVersion;         /* Major version number */
 int MinorVersion;         /* Minor version number */
 int BuildNumber;          /* Build number of the plug-in
                           version */
char *CertificateNumber;   /* Certificate number of the plug-in
                           version */
char *CertificateTime;     /* Certificate time (yyyy/mm/dd
                           hh:mm:ss) */
} VersionInfo;
```

The plug-in information is filled into the input parameter structure when the function returns 0, which indicates that the information was obtained successfully. The function returns -1 if an error occurs.

## Register APIs

The register API function is a system call function for a plug-in to register its APIs. It returns zero for success and -1 for error.

**Function prototype:**

```
int                        /* Return zero on success, -1 if an
                           error occurred  */
 register_apis(
void *Plugin,              /* Plugin object */
void *APIS);               /* API structure pointer */
```

The plug-in object structure type is defined as:

```
typedef struct{
char PluginRawName[PLUGIN_ID_LEN];      /* Raw name in plugin
                                        binary */
enum HONPluginClassType                 /* Raw Class Type in
PluginRawClassType;                     plguin binary */
```

```
int (*PluginInitRoutine)(void *Info);    /* Startup function. */
void (*PluginExitRoutine)(void);         /* Destruction function. */
void *PluginApis;                        /* Plugin APIs. This should be in this structure in order that the plugin could assign APIs' address here */
int MenuIdentifier;                      /* This field is the identifier assigned from Honeywell. Menu codes with the identifier prefix are passed to * the corresponding plug-in */
/* Other plugin infos */
char CompanyName[PLUGIN_STRING_LEN];
char MajorVersionNumber[PLUGIN_STRING_LEN];
char MinorVersionNumber[PLUGIN_STRING_LEN];
char BuildNumber[PLUGIN_STRING_LEN];
char CertificateNumber[PLUGIN_STRING_LEN];
char CertificateTime[PLUGIN_STRING_LEN];
char GUID[PLUGIN_GUID_LEN];
char FileName[PLUGIN_STRING_LEN];
} HONPluginRawInfo;
```

The decode API structure type is defined as:

```
typedef struct
{
   /// Revision Number
   int RevisionNumber;
   /// Decode API callback
   int (*Decode)(unsigned char *pBuffer, int width, int height);
   /// Set Decoder Menu
   int (*SetDecoderMenu)(void *DecoderSetting);
   /// Plug-in Menuing API callback
   int (*ProcessingBarcode)(char *pMenuData, int DataLength);
   /// Check license API callback
   int (*CheckLicense)(char *SN);
   /// Get version API callback
   int (*GetVersion)(VersionInfo *pInfo);
   void (*GPIO_Plugins)(void);
} DecodeApi;
```

CHAPTER

# 7 OFFLINE PLUG-IN API

## Logic of Calling Offline Plug-ins

The order of calling offline plug-ins is controlled by the configuration file but should also conform to the scanner's internal logic. The captured image is sent to the system decode routine first to detect programming barcodes. If it is a programming barcode, the barcode is processed directly and the data is not passed to any plug-ins. If the image is not a programming barcode, after the system decode routine is called to decode the image, the decoded data will be sent to offline plug-ins if offline mode is enabled.

## Offline Plug-in APIs

### ProcessingEvent

This API is called to process system events passed to the offline plug-in.

**Function prototype:**

```
int                                 /* Return zero on success, non-zero
                                    if an error occurred */
(*ProcessEvent)(uint32_t
id);                                /* Input: Event Identification */
```

When the event processing succeeds, the function returns zero. The function returns non-zero if an error occurs.

### ProcessingBarcode

This API is used to process scanned barcodes.

**Function prototype:**

```
int                                 /* Return zero on success, non-zero if an
                                    error occurred */
(*ProcessingBarcode)(               /* Input: Pointer of the OfflineParam
OfflineParam                        structure */
*offline_param );
```

When the programming barcode is processed successfully, the function returns zero. The function returns non-zero if an error occurs or processing failed.

## CheckLicense

The usage of this function is the same as the format plug-in. See also CheckLicense on page 32.

**Function prototype:**

```
int                                 /* Return zero on success, -1 if an
                                    error occurred */
(*CheckLicense)(
char *SN);                          /* Product serial number */
```

The product serial number is passed to the function as a null-terminated string of characters. The function returns 0 if the license is valid or -1 if not.

## CheckVersion

The usage of this function is the same as the format plug-in. See also CheckVersion on page 32.

**Function prototype:**

```
int /* Return zero on success, -1 if an error occurred */
(*CheckVersion)(
VersionInfo *Info); /* Plug-in version info structure */
The version information type "VersionInfo" is defined as:
typedef struct {
int RevisionNumber; /* Revision number */
char *GUID; /* GUID of the plug-in */
char *PluginName; /* Plug-in name */
char *CompanyName; /* Company name of the plug-in */
int MajorVersion; /* Major version number */
int MinorVersion; /* Minor version number */
int BuildNumber; /* Build number of the plug-in version */
char *CertificateNumber; /* Certificate number of the plug-in
version */
char *CertificateTime; /* Certificate time (yyyy/mm/dd hh:mm:ss)
*/
} VersionInfo;
```

The plug-in information is filled into the input parameter structure when the function returns 0, which indicates that the information was obtained successfully. The function returns -1 if an error occurs.

## Process Manifest Plug-in APIs

This API is used to receive data from the remote server over Bluetooth or WiFi. This data can be store in the plug-in to process at a later time when the device is in offline mode.

**Function prototype:**

```
int (*PluginData)(unit8_t*data)
```

## Register APIs

The register API function is a system call function for a plug-in to register its APIs. It returns zero for success and -1 for error.

**Function prototype:**

```
int                     /* Return zero on success, -1 if an
                        error occurred */
register_apis(
void *Plugin,           /* Plugin object */
void *APIS);            /* API structure pointer */
```

The plug-in object structure type is defined as:

```
typedef struct{
char PluginRawName[PLUGIN_ID_LEN];                  /* Raw name in plugin binary */
enum HONPluginClassType PluginRawClassType;         /* Raw Class Type in plguin binary */
int (*PluginInitRoutine)(void *Info);               /* Startup function. */
void (*PluginExitRoutine)(void);                    /* Destruction function. */
void *PluginApis;                                   /* Plugin APIs. This should be in this structure in order that the plugin could assign APIs' address here */
int MenuIdentifier;                                 /* This field is the identifier assigned from Honeywell. Menu codes with the identifier prefix are passed to * the corresponding plug-in */
/* Other plugin infos */
char CompanyName[PLUGIN_STRING_LEN];
char MajorVersionNumber[PLUGIN_STRING_LEN];
char MinorVersionNumber[PLUGIN_STRING_LEN];
char BuildNumber[PLUGIN_STRING_LEN];
char CertificateNumber[PLUGIN_STRING_LEN];
char CertificateTime[PLUGIN_STRING_LEN];
char GUID[PLUGIN_GUID_LEN];
char FileName[PLUGIN_STRING_LEN];
} HONPluginRawInfo;
```

The plugin API structure type is defined as:

```
typedef struct
{
    /// Revision Number
    int RevisionNumber;
    /// Event processing API callback
```

```
    int (*ProcessingEvent)(uint32_t id);
    /// Plug-in barcode processing API callback
    int (*ProcessingBarcode)( OfflineParam *offline_param);
    /// Check license API callback
    int (*CheckLicense)(char *SN);
    /// Get version API callback
    int (*GetVersion)(VersionInfo *pInfo);
    void (*GPIO_Plugins)(void);
} OfflineApi;
```

# Offline Timer

This is a system call to set a timer parameter in milliseconds. When the timer expires the response is provided as an event OFFLINE_EVENT_TIMER_EXPIRE.

**Example:** Define the timer parameters.

```
struct matrix_syscall_param pParam;
pParam.syscall_id = MATRIX_TIMER;
pParam.params[0] = (int *) <timer value>;
matrix_syscall(&pParam);
```

# Play Audio in Plug-in

See Play Audio Files on page 32 to add audio files to the devices.

**Example:** Define audio file.

```
struct matrix_syscall_param pParam;
pParam.syscall_id = MATRIX_AUDIO;
pParam.params[0] = (int *) <Audio Filename without file
extension>;
matrix_syscall(&pParam);
```

CHAPTER

# 8 DIAGNOSTICS

## Boot Mode to Disable Loading Plug-in

If the scanner interface becomes locked due to corrupt plug-ins, you may boot the scanner without loading plug-ins. The following steps force the scanner to boot in boot mode:

1. Run EZConfig for Scanning and use **Configure-Communications** to set the Baud Rate to **115200**, and Word Format to **N 8 1**.
2. From the menu, select **Device-Force Reader to Boot Mode**.
3. Power the scanner and press any key.
4. From the Application Explorer pane, select **Scan** Data.
5. From the menu, select **View-Serial Command Window**. Enter **232** in the text box of the Command Center window and click the **Send Non Menu Command** button.
6. The scanner loads the application without loading plug-ins.

In this mode, you can scan programming barcodes or send menu commands to disable the plug-in. After power-cycling the scanner, the new configuration files or modified plug-ins can be downloaded.

# View Plug-in Configuration

The menu command “PLGINF” is used to show the plug-in configurations and load status of plug-ins. Send menu command “PLGINF” in the Serial Command Window in EZConfig for Scanning. A sample of the output is shown:

```
Plugin Configurations:
[Format Plugin Configuration]
   <HelloWorld.plugin>
      [name]:                  HelloWorld
      [company]:               Plug-In Developer, Inc.
      [licensed]:              YES
      [active]:                YES
      [majorrevision]:         5
      [minorrevision]:         3
      [build]:                 37
      [certificate]:           102148
      [certificatetime]:       2009/08/10 15:00:05
      [guid]:                  abcd1234
      [filename]:              HelloWorld.plugin
      [mainroutineorder]:      BEFORE
      [bar codeinterceptmode]: YES
      [entrydatastate]:        MODIFIED
      [chainonexit]:           CHAINIFFAILURE
      [loadstatus]:            SUCCESS

   <sample.plugin>
      [name]:                  FormatPlugin
      [company]:               Plug-In Developer, Inc.
      [licensed]:              YES
      [active]:                YES
      [majorrevision]:         5
      [minorrevision]:         3
      [build]:                 37
      [certificate]:           102148
      [certificatetime]:       2009/08/10 15:00:05
      [guid]:                  abcd1234
      [filename]:              sample.plugin
      [mainroutineorder]:      BEFORE
      [bar codeinterceptmode]: YES
      [entrydatastate]:        MODIFIED
      [chainonexit]:           CHAINIFFAILURE
      [loadstatus]:            SUCCESS
```

Some of the fields in the configuration file may be updated to conform to scanner settings the first time the plug-in is loaded.

# Load Status of Plug-ins

The “loadstatus” field in the configuration file is updated every time after a plug-in is loaded. It indicates success or the reason for failure if the plug-in cannot be loaded. This field may display:

| | |
|---|---|
| SUCCESS | The plug-in is loaded successfully |
| INACTIVE | The plug-in is inactive |
| UNLICENSED | The plug-in is unlicensed |
| NORESOURCE | 1. Short of resources to load the plug-in<br>2. Cannot open plug-in file (file not found)<br>3. Not enough memory<br>4. File operation error when loading plug-in<br>5. Main routine not found in the plug-in<br>6. Helper not found in the plug-in |
| CORRUPT | 1. Plug-in is corrupt<br>2. Unknown symbol<br>3. Bad relocation<br>4. Relocation out of range<br>5. Unknown relocation |
| CORRUPTCONFIGENTRY | Configuration file is corrupt |
| NOPLUGINDEFINED | No definition in configuration entry |
| PLUGINTERMINATE | Error occurred during plug-in initialization |

# Plug-in Relevant Menu Settings

Plug-in relevant menu settings are used to help develop and debug plug-ins:

| | |
|---|---|
| PLGIPE | Fully visible boolean setting to enable / disable image processing class plug-ins (1 for enable, 0 for disable). |
| PLGDCE | Fully visible boolean setting to enable / disable decode class plug-ins (1 for enable, 0 for disable). |
| PLGFOE | Fully visible boolean setting to enable / disable format class plug-ins (1 for enable, 0 for disable). |
| PLGOLE | Fully visible boolean setting to enable / disable offline class plug-ins (1 for enable, 0 for disable). |
| PLGDBG | Fully visible boolean setting to enable / disable plug-ins to output debug information (1 for enable, 0 for disable). |

| | |
|---|---|
| PLGIPN | Fully visible string setting containing the name of the image processing class configuration file. Default is null (no configuration file). |
| PLGDCN | Fully visible string setting containing the name of the decode class configuration file. Default is null (no configuration file). |
| PLGFON | Fully visible string setting containing the name of the format class configuration file. Default is null (no configuration file). To turn on FormatConf, enter the menu command: "PLGFONFormatConf" and hard reboot the scanner. The configuration file name should be a string consisting of ASCII characters except '.', '?', ';' and '!' (these characters are reserved for menu commands). |
| PLGOLN | Fully visible string setting containing the name of the offline class configuration file. Default is null (no configuration file). |
| PLGINF | The plug-in configuration files may be reported to the host via the hidden PLGINF menu command. |
| PLGDEL | Delete the plug-in file or configuration file from the scanner. |
| PLGDIR | List all the plug-in files and configuration files in the scanner. |
| PLGREA | Read the content of a configuration file. (Do not use this menu command to output a plug-in file.) |
| PLGREN | Rename a plug-in file or configuration file:<br>PLGRENOldFileName:NewFileName<br>The old name and new name are separated by a colon. |
| PLGCPY | Copy a plug-in file or a configuration file:<br>PLGCPYOrgFileName:DstFileName<br>OrgFileName is the original file name and DstFileName is the destination file name. The original name and destination name are separated by a colon. |
| PLGDLA | Delete all plug-in files and configuration files in the scanner. |

Honeywell
855 S. Mint St.
Charlotte, NC 28202

sps.honeywell.com

TF3-PD-EN-UG-01 Rev A
4/23