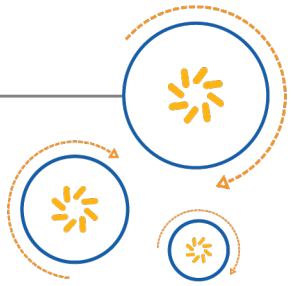
Qualcomm Technologies International, Ltd.



Bluetooth SDK Command Line Tools

User Guide

80-CT430-1 Rev. AN

October 18, 2017

Qualcomm Technologies International, Ltd. (formerly known as Cambridge Silicon Radio Limited) is a company registered in England and Wales with a registered office at: Churchill House, Cambridge Business Park, Cowley Road, Cambridge, CB4 0WZ, United Kingdom.
Registered Number: 3665875 | VAT number: GB787433096

Revision history

Revision	Date	Description
a	SEP 2005	Original publication of this document. (CSR™ reference: blab-ug-009Pa). Alternative document number CS-00101506-UG.
b	SEP 2006	Updated to include ledparse application. (CSR reference: blab-ug-009b)
3	NOV 2006	Updated to include functionality added in Qualcomm® BlueLab™ 3.6 (CSR reference: CS-101506-UG3) Not published
4	DEC 2006	Updated to include functionality added in Qualcomm® BlueLab™ 4.0 (CSR reference: CS-101506-UG4)
5	OCT 2009	Updated to latest style guidelines.
6	APR 2010	Updated for 2010 SDKs and BlueLab references removed.
7	MAR 2011	Updated for kalasm3. Updated to latest style guidelines.
8	JAN 2012	Updated to latest style
9	APR 2014	Updated to latest CSR style
10	MAY 2015	Updated for ADK 4
11	NOV 2015	BlueFlashCmd removed and replaced by NvsCmd
12	SEP 2016	Table in .parse file literals updated. Updated to conform to QTI standards.
AN	OCT 2017	Added to the Content Management System. No technical content was changed in this document revision.

Contents

Figures

1 Overview of the build process in SDKs

Figure 1-1 gives an overview of the utility applications and their use during the build and download processes as implemented by Bluetooth SDKs.

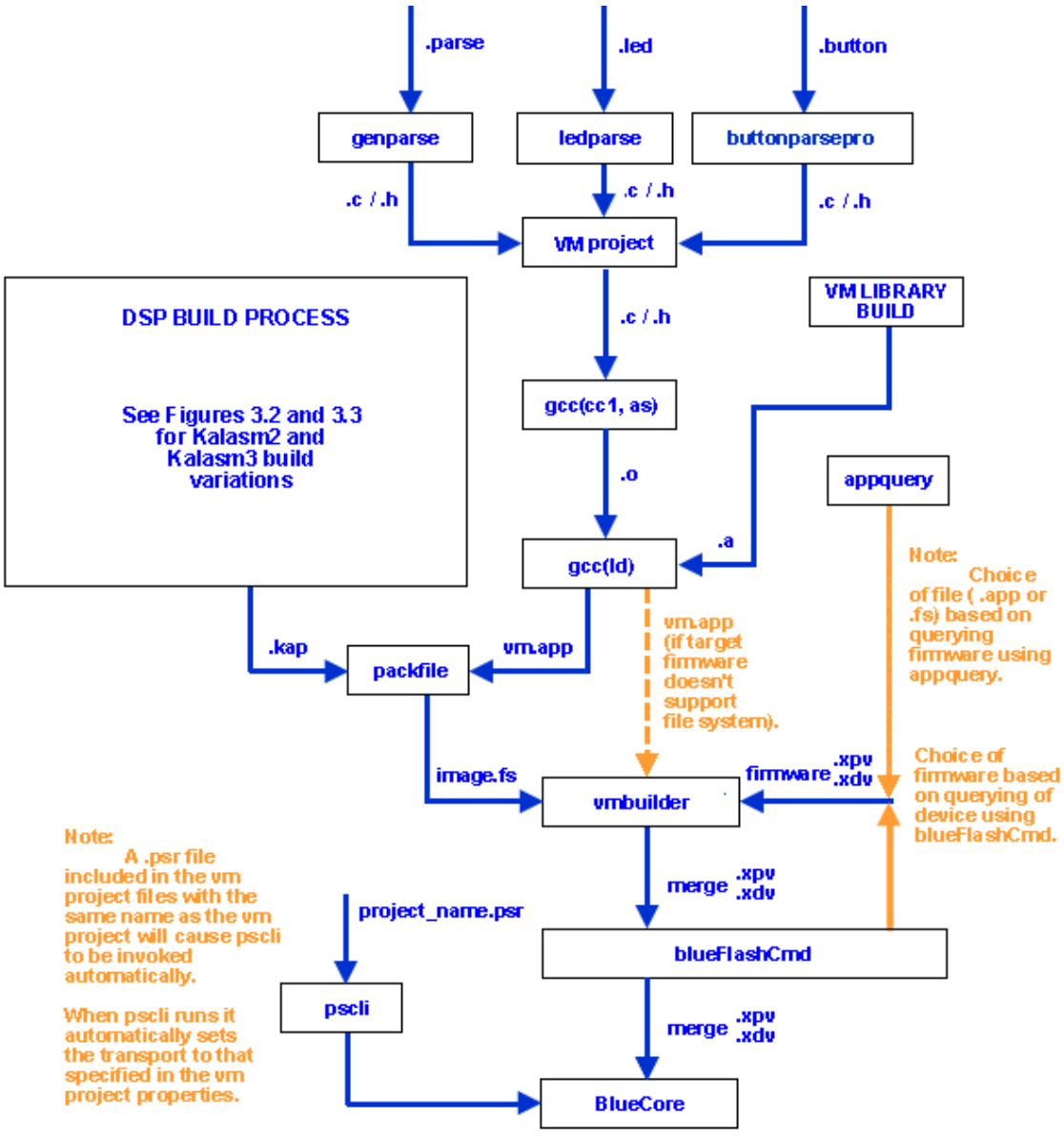


Figure 1-1 Overview of the utility applications

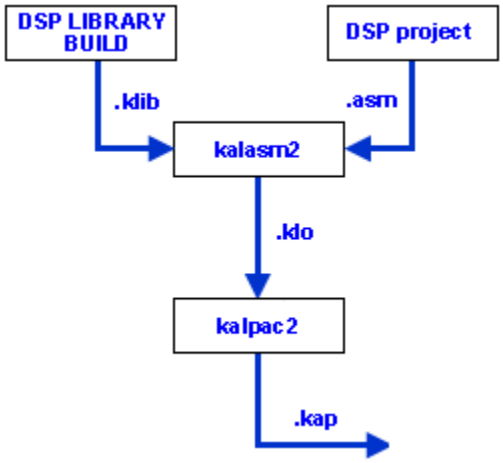


Figure 1-2 DSP build process for kalasm2

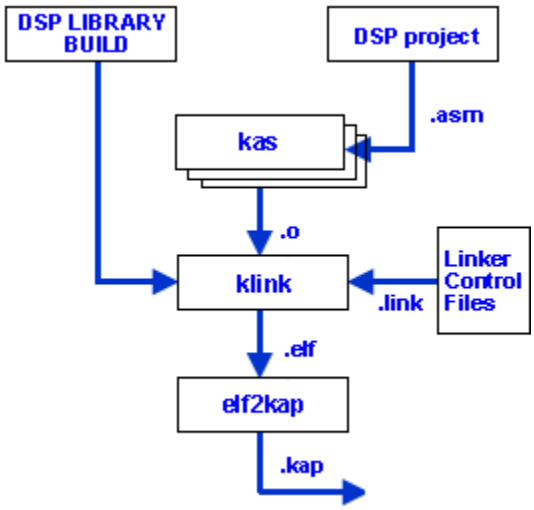


Figure 1-3 DSP build process for kalasm3

2 buttonparsepro

The buttonparsepro utility provides a quick and easy method of generating the code to bind button names to PIO events and to define the message sent to the application task when the button event occurs.

General

buttonparsepro takes a `.button` file containing a description of the required buttons and outputs the `.c` and `.h` files required to implement them, that is:

```
buttonparsepro.exe <input file.button> <output file name>
```

Description of .button files

`.button` files initially define button names for the PIO pins the engineer is interested in, for example:

```
pio 7 UP
pio 6 DOWN
pio 5 OFF
```

It is also possible to specify debounce when reading these PIOs by adding a statement in the form:

```
debounce n m
```

Where `n` is the number of times to read the PIO before deciding the value is correct and m is the period in milliseconds to wait between reads.

NOTE If no debounce is specified, the default values `n` = `1` and `m` = `0` are used.

Next, the messages for the buttons and when they will be sent can be defined. The following code extracts describe the options available, with a brief description of the resulting functionality.

To send a message (`VOLUME_UP`) when PIO [7] is pressed and a repeat message every 500 ms if it is held down:

```
message VOLUME_UP
UP enter
UP repeat 500
```

To send a message (`VOLUME_DOWN`) when PIO [6] is pressed but no repeat message if it is held down:

```
message VOLUME_DOWN
DOWN enter
```

To send a message (`RESET`) when two PIOs ([6] and [7]) are pressed simultaneously for a given period (1 second), with no repeat message being sent:

```
message RESET
UP DOWN held 1000
```

To send a message (POWER_OFF) when a PIO [5] is held down for a specified period (2 seconds), that is, a long press, with a repeat message sent at a different repeat period (every 1 second) after the initial message:

```
message POWER_OFF
OFF held 2000
OFF repeat 1000
```

To send a message (SHORT_PRESS) when a PIO [5] is released:

```
message SHORT_PRESS
OFF release
```

If a held event (that is, a long press, if defined) is detected and a message is sent as a result, it cancels any release messages. This allows short and long press messages to be specified.

To send a message (DOUBLE_PRESS) when PIO [7] is pressed twice within a specified period (within a second):

```
message DOUBLE_PRESS
UP double 1000
```

An additional label held_release enables short, long and very long press to be defined.

For example, to send a message (LONG_RELEASE) when a button is released after being pressed for a specified period (2 seconds):

```
message LONG_RELEASE
OFF held_release 2000
```

NOTE If a release event is defined, the SHORT_PRESS message will be suppressed if either a held_release or held message has been sent.

Changes in the PIO state can be sent to the application task in the form of a PIO_RAW message.

NOTE This is generally used when monitoring the state of PIOs not used to monitor button events.

To receive the PIO message in this form, statements are added to the top of the .button file, for example:

```
pio_raw 13
pio_raw 15
```

The PIO_RAW message received by the task contains the state of both PIOs and can be read for each by looking at the relevant bit position for the required PIO, for example:

```
((PIO_RAW_T*)message)->pio & (1<<13)   to read PIO 13
((PIO_RAW_T*)message)->pio & (1<<15)   to read PIO 15
```

NOTE A case statement to handle PIO_RAW messages needs to be added to the handler code for the AppTask.

Developers should not attempt to specify a given PIO as both a button and non-button event. If this is done the code will not implement the button instance.

Additional functionality

The buttonparsepro utility:

- Supports up to 32 PIO lines.
- Can interpret VREG_EN and CHG_EN in the same way as PIO lines.

3 ledparse

The ledparse utility provides a quick and easy method of generating the code to define LED patterns.

ledparse includes support for:

- Multiple LEDs (including tricolor)
- Start/stop LED patterns
- Pattern configuration
- Single play patterns
- Repeated patterns
- Restarting a repeated pattern interrupted by a single play pattern

ledparse is compatible with all BlueCore variants with or without special LED drive pins.

General

ledparse takes an `.led` file containing a description of the required LED patterns and outputs the `.c` and `.h` files required to implement them, that is:

```
ledparse.exe <input file.led> <output file name>
```

Description of .led files

`.led` files initially define the PIO/LED pins to be used in the patterns, for example:

```
pio 8 GREEN
led 0 RED
led 1 BLUE
```

Then go on to define the required patterns. Each pattern is defined using the following syntax:

```
pattern NAME_OF_PATTERN [RPT]
LED [LED2 LED3…] ACTION DURATION
```

Examples

To define a pattern where both the red and blue LEDs are on for 1 second:

```
pattern RED_BLUE_ONE_SEC
RED BLUE ON 1000
RED BLUE OFF 0
```

To define a repeating pattern where the red and blue LEDs alternate with a 200 ms duration.

```
pattern RED_BLUE_ALT RPT
RED ON 200
RED OFF 0
BLUE ON 200
BLUE OFF 0
```

Green permanently on:

```
pattern GREEN_ON RPT
GREEN ON 0
```

3.1 Using autogenerated LED patterns in the VM application

When ledparse has been used to generate the LED patterns, they can be implemented in the VM application code by including the `<output file name>.h` and calling `ledsPlay()` when the pattern is to be played:

```
#include <ledpatterns.h>
Bool ledsPlay (LedPattern_t pNewPattern)
```

That is, to play the repeating pattern `RED_BLUE_ALT as` defined in the second example in ledparse:

```
bool ledsPlay (RED_BLUE_ALT)
```

ledsPlay functionality

The effect of calling ledsPlay depends on whether an LED pattern is already playing, the type of pattern playing, and the type of the new pattern called:

When a repeating pattern is already playing:

If the new pattern is a repeating pattern, the new pattern will replace the original pattern.

If the new pattern is a non-repeating pattern, the original pattern is interrupted, the new pattern is played then the original pattern is resumed.

When a non-repeating pattern is playing:

If the new pattern is a repeating pattern, the new pattern will play when the original pattern completes.

If the new pattern is a non-repeating pattern, the new pattern is not played and `ledsPlay` returns false.

NOTE It is up to the caller to queue non-repeating patterns if required.

Further insight into use of ledparse in VM applications can be gained by examining the tutorial and example applications available with the Bluetooth SDKs.

4 genparse

The genparse utility provides a quick and easy method of generating code required by an application to identify incoming command strings, for example, to parse and identify AT commands in applications implementing audio gateway, headset or hands-free applications.

General

genparse takes one or more `.parse` files containing a description of command strings as input, and outputs the generated code as a `.c` and an `.h` file.

The output files can then be compiled and linked into the application. In Bluetooth SDKs `make` identifies a `.parse` file when included in a Project and calls `genparse.exe` before compiling and linking the application code.

4.1 .parse files

`.parse` files use the syntax:

`{pattern} : handler` or `<literal> : handler`

The following code extract shows examples taken from the `.parse` file included in the `hfp` library (found in `C:/<SDK-Name>/src/lib/hfp/hfp_parse.parse`):

```
# Set speaker gain indication (allowed values: 0-15)
{ \r \n + VGS =  %d:gain \r \n } : hfpHandleSpeakerGain
# Incoming call indication (literal)
<\r\nRING\r\n> : hfpHandleRing
```

The `hfp_parse.parse` file contains all the AT commands that are mandatory to the headset and hands-free specification. However, if a developer wants to support additional commands these can be added as required.

4.1.1 .parse file patterns

Each pattern is processed by the generated code from left to right. Fields in the pattern are handled as follows:

Field	Parser Handling
'\r' or '\n'	Written in the pattern as '\' followed by 'r' or 'n'. Requires that a carriage-return (\r) or newline (\n) is present in the input. Skips past it if this is so, otherwise this pattern fails and the next one is considered.
'%'	Indicates the start of a string or numeric parameter for the handler.
space, or tab character	Skips past one or more occurrences of either of these characters in the input.
',' or ';'	Expects but ignores a single occurrence of either a comma or semi-colon in the input. Does not differentiate between them, that is, considers either to be acceptable. Skips past it if this is so, otherwise this pattern fails and the next one is considered.
'?'	Expects but ignores one or more occurrences of a question mark in the input. Skips past it if this is so, otherwise this pattern fails and the next one is considered.
'-'	Expects but ignores one or more occurrences of a hyphen in the input. Skips past it if this is so, otherwise this pattern fails and the next one is considered.
'(' or ')'	Expects but ignores one or more occurrences of either an opening or closing bracket in the input. Does not differentiate between them, that is, considers either to be acceptable. Skips past it if this is so, otherwise this pattern fails and the next one is considered.
'=' or ':'	Expects but ignores an occurrence of either an equals sign or a colon in the input. Does not differentiate between them that is considers either to be acceptable. Skips past it if this is so, otherwise this pattern fails and the next one is considered.
'*' '^' or alphanumeric character	Requires that the specified character is next in the input (considered case-insensitively for alphabetic characters.) Skips past it if this is so, otherwise this pattern fails and the next one is considered.
//	Used to comment out a line.
Any other character	No other characters are allowed in patterns. genparse will fail with an error message if they are found.

4.1.2 .parse file parameters

A % in a pattern should be followed by either s:name, or d:name indicating string or numeric parameter respectively. name is a non-empty sequence of alphanumeric characters, which the handler can use to refer to that parameter.

A string parameter matches an (optionally) quoted, non-empty sequence of alphanumeric characters or spaces in the input. If it succeeds, the contents of the string (without enclosing quote marks) are passed to the handler, otherwise the pattern fails and the next one is considered.

The following alphanumeric characters are considered valid for a string parameter.

[SP] 0x20			# 0x23		% 0x25		‘ 0x27	(0x28	) 0x29	* 0x2a	+ 0x2b				
0 0x30	1 0x31	2 0x32	3 0x33	4 0x34	5 0x35	6 0x36	7 0x37	8 0x38	9 0x39						
	A 0x41	B 0x42	C 0x43	D 0x44	E 0x45	F 0x46	G 0x47	H 0x48	I 0x49	J 0x4a	K 0x4b	L 0x4c	M 0x4d	N 0x4e	O 0x4f
P 0x50	Q 0x51	R 0x52	S 0x53	T 0x54	U 0x55	V 0x56	W 0x57	X 0x58	Y 0x59	Z 0x5a		\ 0x5c		^ 0x5e	_ 0x5f
	a 0x61	b 0x62	c 0x63	d 0x64	e 0x65	f 0x66	g 0x67	h 0x68	i 0x69	j 0x6a	k 0x6b	l 0x6c	m 0x6d	n 0x6e	o 0x6f
p 0x70	q 0x71	r 0x72	s 0x73	t 0x74	u 0x75	v 0x76	w 0x77	x 0x78	y 0x79	z 0x7a	Legend:			[character] [hex value]	

If a non-valid character is encountered the input is terminated at that character.

A numeric parameter matches a non-empty sequence of decimal digits in the input. If it succeeds, the decimal value of the sequence is passed to the handler, otherwise the pattern fails and the next one is considered.

Any parameters passed to the handler are assembled in a structure whose name is derived from that of the handler. The name of the values in the structure is as specified in the pattern by name.

Complex patterns

In some cases, the pattern of a command may be too complex to define the message in genparse. In this case the parameter `%*` can be used. This parameter passes the full string making up the command from `%*` on.

4.1.3 .parse file literals

Each literal is compared exactly against a complete packet, including case and spaces. If it matches, then the corresponding handler is called. Handlers for literals do not receive a pointer to any parameters (other than the `BD_ADDR`) since literals cannot specify any parameters.

Literals should only be used to provide special-case handling for particular instances of long, complex AT commands, which would otherwise parse very slowly.

NOTE A parser must handle countless variations on a given AT command with changes in spacing and capitalization. It is not possible to cover all those variations with literals and a corresponding pattern must exist.

In other words, any parser should be developed so that it functions correctly with no literals present. The literals should be considered as an optimization, which can be added later.

4.1.4 .parse file handlers

The genparse output places declarations for the handlers used in the `.parse` file in the output `.h` file. The client application must provide definitions for all handlers included in the `.parse` file.

The first parameter to all handlers is a pointer to the Task passed in to `parseData`. Handlers that have parameters in their pattern are also passed a pointer to a structure containing those parameters.

You can use the same handler for multiple patterns, but the names and types of the parameters must be identical for all of the patterns using that handler.

A default handler:

```
void handleUnrecognised(const uint8 *data, uint16 length, const Task
*Task);
```

is declared in the output `.h` file to handle data that does not match any of the patterns or literals in the `.parse` file. This function must always be defined by the application.

4.2 genparse output files

The genparse utility outputs two files: a `.c` file and an accompanying `.h` file.

The files contain an implementation of:

```
const uint8 parseData (const uint8 *s, const uint8 *e, Task task);
```

This function scans the input between `s` and `e`. For each complete packet in that range, it will call the appropriate handler if a pattern or literal match is found. If no match is found `handleUnrecognised` is called. It returns a pointer to the first unprocessed character in the input.

The output files also contain an implementation of:

```
uint16 *parseSource (Source rfcDataIncoming, Task task);
```

This function calls `parseData` on the contents of a source. It then drops from the source any characters that have been dealt with. It returns non-zero if any characters have been dropped, zero otherwise.

Applications, for example the `stereo_headset`, typically call the `parseSource` on incoming RFCOMM data although it can be called to parse data from any source.

4.3 Genparse packet boundaries

The code generated by the genparse utility only tries to match patterns on complete packets in the input source. It considers a complete packet to be either a string terminated with a carriage-return, or a string surrounded by carriage-return line-feed pairs. This covers the AT command syntax allowed by ETS 300 916 (GSM 07.07).

4.4 Genparse examples

This section takes examples from the `hfp_parse.parse` and `hfp_parse.h` files to further clarify how genparse handles `.parse` files.

The simplest example is a literal that by definition does not pass any parameters:

```
for example {\r \n OK \r \n} : hfphandleOk
```

In this case genparse declares `void hfphandleOk(Task);` in the `.h` file.

Next consider a pattern that passes a numeric parameter as a named component:

```
for example { \r \n + VGS =  %d:gain \r \n } : hfpHandleSpeakerGain
```

This statement causes genparse to generate the following code:

```
struct hfpHandleSpeakerGain
{
uint16 gain;
};
void hfpHandleSpeakerGain (Task, const struct hfpHandleSpeakerGain *);
```

String parameters are passed as elements of `type struct` or `struct` sequence that contain a pointer to the start of the data, and its length.

Thus a statement:

```
for example   { \r \n + BINP = %s:num \r \n } : hfpHandleDataResponse
```

This statement causes genparse to generate the following code:

```
struct sequence
{
  const uint8 *data;
  uint16 length;
};
struct hfpHandleDataResponse
{
  struct sequence num;
        };
void hfpHandleDataResponse(Task, const struct hfpHandleDataResponse*);
```

When a repeated number of parameters are expected, you can express this using a loop:

```
for example .{ \r \n + CHLD = ( [p,1,99][%s:n][ , ] ) \r \n}
:hfpHandleCallHoldInfo
```

This statement causes genparse to generate the following code:

```
struct region_hfpHandleCallHoldInfo_p
{
  uint16 count;
  const uint8 *s;
  const uint8 *e;
  uint16 next;
  const uint8 *next_s;
};
struct hfpHandleCallHoldInfo
{
  struct region_hfpHandleCallHoldInfo_p p;
};
```

NOTE This code defines a structure that accommodates up to 99 comma (,) separated strings, which are passed a 'p' to the helper function.

```
struct value_hfpHandleCallHoldInfo_p
{
  struct sequence n;
};
```

This code defines a structure that holds the string values within the pattern.

```
struct value_hfpHandleCallHoldInfo_p
get_hfpHandleCallHoldInfo_p(const       struct
region_hfpHandleCallHoldInfo_p*, uint16);
```

This function extracts the string values present in the string represented by `[%s:n ]` in the matched pattern which are passed to the handler function:

```
void hfpHandleCallHoldInfo(Task, const struct
hfpHandleCallHoldInfo*);
```

genparse simply declares this prototype. It is the developer's responsibility to define the necessary implementation for the function. Parameters received can be extracted from the structure passed into the function as or if required. For example, an application might print all the strings matched using:

```
void hfpHandleCallHoldInfo (Task t, const struct
hfpHandleCallInfo* x)
{
   int i;
   for (i=0; i< x->p.count; ++i)
{
   value_hfpHandleCallInfo y = get_hfpHandleCallInfo_p (&x->p,
i);
   print (y.n.data, y.n.length);
}
        }
void print (const uint8 *s, uint16 n)
{
uint16 i; for (i=o, i< n; ++i)putchar (s[i]);
putchar ('\n')
}
```

4.5 Genparse user defined parameters

The genparse utility has been extended to allow users to define their own parameters in the `.parse` file.

User parameters are defined using the syntax:

```
[regex] : parameter_name
```

User defined parameters can be used in patterns to restrict the characters that are matched for a particular value in the handler.

4.5.1 Genparse qualifiers

User-defined parameters can also make use of regular expression qualifiers `+` and `*`. When neither of these qualifiers are used, the value in the handler is matched to only a single character. `+` extends this to one or more and `*` extends this to zero or more.

Examples:

To define a parameter to match one of characters `a`, `b`, `c`, `d`, `s` or `r`:

```
[a-dsr] : new_param
{ \r \n NEW = %new_param : name \r \n} : newHandler
```

To define a parameter to match characters other than `n`, `m`, `o`, `p`, `q` and `r`:

```
[^n-r] : sec_new_parameter
{ \r \n SEC = %sec_new_parameter : name \r \n} : secHandler
```

To define a parameter to match zero or more of characters `b`, `e`, and `g`:

```
[beg] : third_new_param
{ \r \n THIRD = %new_third_param : name \r \n} : thirdHandler
```

5 NvsCmd

NvsCmd is a command line tool that can be used to download or dump code to and from the BlueCore IC as well as identify chips and their characteristics.

In the context of this document the main use of NvsCmd is during Run in xIDE, where `make` invokes it to query the device to determine the correct choice of firmware and to finally download the merge `.xuv` file onto the BlueCore IC.

The NvsCmd line options and commands are documented in the application and are displayed when the application is run with the `--help` command line option.

To use the facilities provided by this application during the development process, QTIL recommends using BlueFlash, which provides the same functions with a more user-friendly interface.

NOTE NvsCmd is primarily intended for use by xiDE, for example, when downloading code to a development platform. QTIL supply a separate SDK library TrueTest which can be used to write applications better suited to downloading code in a production environment. Contact QTIL for further details.

6 pscli

pscli is a command line implementation of the PSTool application and can be used to read and write values to the Persistent Store of BlueCore ICs in much the same way.

In the context of this document the main use of pscli is during Run in xIDE, where `make` automatically invokes pscli when a `.psr` file is present in the project.

NOTE During **Run**, `pscli` automatically sets the IC's transport to that specified in the **Project Properties** of the Bluetooth SDK project, that is, if the engineer has manually changed the IC's transport using PSTool the transport will be overwritten when `pscli` is evoked, unless the transport is also changed in **Project Properties**.

The `pscli` command line options and commands are clearly documented in the application and displayed when `pscli.exe` is run in a command window.

QTIL recommends that PSTool is used when manually manipulating PS Keys during the development process and that pscli is only used by the SDK build process.

The use of PSTool including the syntax for `.psr` and `.psq` files is described in the *PSTool User Guide*.

NOTE `pscli` is primarily intended for use by xIDE, for example, when downloading PS Key values to a development platform. QTIL supplies a separate application TrueTest that is better suited to downloading code in a production environment. Contact QTIL for further details.

7 make

Introduction

The `make` application determines which pieces of the program need to be compiled (or recompiled) and issues the necessary commands to compile (or recompile) them based on rules defined in `makefiles`. The project `makefile` generated by xIDE defines the relationship between the files making up the application (and the firmware image being used), and provides rules for building the final `merge.xpv` and `.xdv` pair that is programmed onto on the BlueCore IC.

Each time changes are made to the source, invoking `make` orchestrates all necessary recompilations and generation of the final machine code for the application program.

The `make` utility shipped with Bluetooth SDKs is a variant of the open source GNU `make` utility described at http://www.gnu.org/software/make/manual/make.html

NOTE Bluetooth SDKs call `make` when a Build (**F7**) or Run (**F5**) operation is performed in xIDE.

General

When `make` is invoked, it normally orchestrates:

- Preprocessing
- Compilation
- Assembly
- Linking

The `make` application invokes other utilities as required to carry out the compilation and passes command line options and file names as defined in the makefile.

When called in Bluetooth SDKs, the makefile generated by xIDE automatically passes the necessary command line options and file names to correctly compile and link Bluetooth SDK Projects and to download the resultant code to the BlueCore IC.

In general, SDK users need not be concerned with the dependencies and intricacies of how `make` is configured and implemented in xIDE. However, for those interested in learning more about `make` there are many commercial publications dedicated to all aspects of its use.

NOTE QTIL has tested the combination of command line options used by Bluetooth SDKs. Users who want to generate their own makefiles should be aware that using options other than those used by Bluetooth SDKs may expose subtle problems. In particular, evoking higher optimization levels (-O2 and -O3) in `gcc` are known to cause some issues.

8 xap-local-xap-ar

The implementation of xap-local-xap-ar supplied with Bluetooth SDKs support a limited number of the command line options supported by a full implementation. This chapter details the supported options.

- **Making a lbrary**

Files can be added to a library using `q`, `u` or `r` command line options. If the library does not exist it is created. If the files are already in the library, they are replaced.

For example:

`ar r library.a obj1.o obj2.o` adds the 2 object files to the library.

- **Unpacking a file**

All the files in a library can be extracted using:

```
ar x library.a
```

Selected files can be extracted using:

```
ar x library.a obj1.0 obj2.o
```

- **Other options**

`c`, `v`, `f`, `l`, `s`, options are silently ignored.

`d`, `m`, `p`, `t`, `a`, `b`, `i`, `o`, `P`, `V, and N` options are not implemented and reported as an error.

9 xipbuild

The xipbuild utility can be used to build xIDE projects from the command line rather than through the xIDE GUI. Each xIDE project's details are contained in a file with the `.xip` extension.

For xIDE workspaces that contain multiple projects (for example a single VM project and multiple DSP projects) xipbuild must be evoked for each project individually rather than once for the complete workspace.

- **Full option list**

```
xipbuild [-b | -f | -c] [-d | -a | -n configuration] project
```

- **Building a project**

To perform a standard rebuild on the project demo:

```
xipbuild demo
```

- **Build options**

```
-b : Standard build (the default)

-f : Rebuild All

-c : Clean
```

- **Configuration options**

```
-d : Build/clean default build configuration (the default)

-a : Build/clean all build configurations

-n : Build/clean a named build configuration
```

Document references

Document	Reference
Kalimba DSP Assembler User Guide	CS-00110560-UG
PSTool User Guide	80-CT424-1/CS-00101505-UG

Terms and definitions

Term	Definition
AT	Attention (modem command prefix)
BlueCore	Group term for the range of QTIL Bluetooth wireless technology ICs
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
Bluetooth SIG	Bluetooth Special Interest Group
DSP	Digital Signal Processor
GUI	Graphical User Interface
HFP	Handsfree Profile
IC	Integrated Circuit
LED	Light Emitting Diode
PIO	Programmable Input Output
QTIL	Qualcomm Technologies International, Ltd.
RFCOMM	Serial Cable Emulation Protocol
SDK	Software Development Kit
VM	Virtual Machine