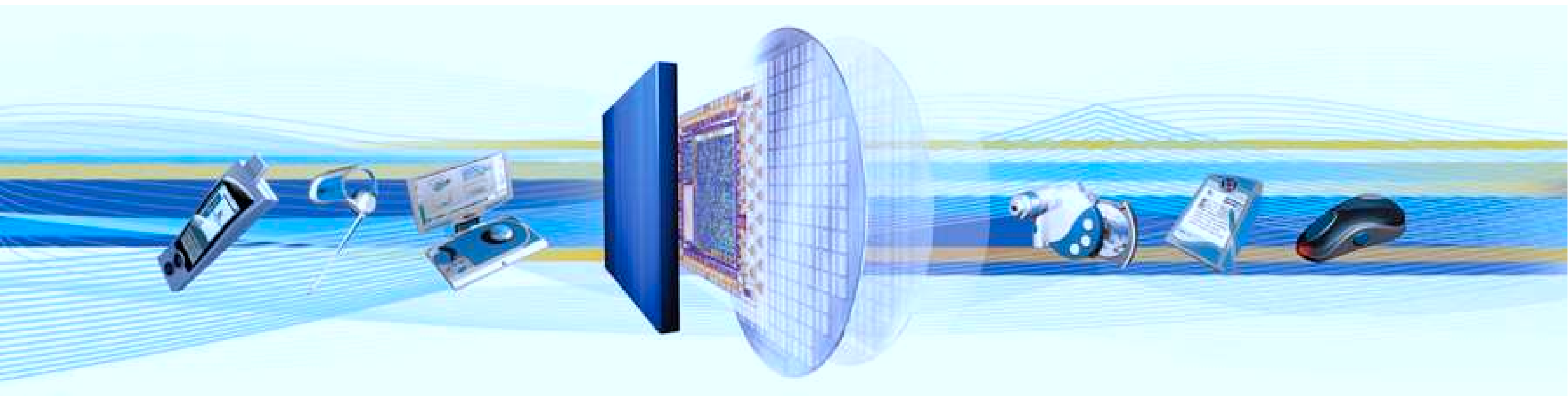
BlueLab™

a guide to

BlueLab Command Line Tools

January 2006



CSR

Cambridge Science Park
Milton Road
Cambridge CB4 0WH
United Kingdom

Registered in England 4187346

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001
www.csr.com

Contents

1 Introduction

This document describes the command line applications that are shipped with the BlueLab v3.4 Software Development Kit. It does not list all the individual options and commands accepted by each application, where appropriate these are included within the application and are listed when the application is run.

2 Overview of BlueLab Command Line Applications

The applications covered in this document can be divided into sub-groups based on their area of functionality:

2.1 Tools that generate code

- **buttonparse**

  The buttonparse application is used to bind button names to PIO pins and to define a message that will be sent when a button is pressed. It allows long, short, repeat and multiple button presses to be specified and assigned messages.

  From information input as a `.button` file, `.buttonparse.exe` generates the C files (`.c`) and header files (`.h`) required by the application to send the messages to a task, typically the AppTask, in response to a button event occurring.

  Note: It is the developer's responsibility to write the necessary code to handle these messages when they are received by the task. Buttonparse just generates the code to convert PIO transitions into messages.

- **genparse**

  This utility generates the code required to parse command strings (e.g. AT commands) received by an application in order to identify the command.

  This application generates the necessary C files (`.c`) and header files (`.h`) from text string statements in a `.parse` file.

  The files are generated in the same location as the `.parse` file and can be in either a library, as in the stereo headset reference application code which generates the files in the hfp library, or in the BlueLab Project files.

  Note: The best location for the files will largely depend on the application and is at the discretion of the developer.

2.2 Command line equivalents to tools in BlueSuite

These applications are command line equivalents to the BlueFlash and PSTool applications that are shipped with BlueLab and are part of BlueSuite.

- **BlueFlashCmd**

  The BlueFlashCmd application can be used to download and upload code to and from the flash memory on **BlueCore™** chips.

- **pscli**

  pscli is a command line interface to the BlueCore Persistent Store. It can be used to read, modify and write the PSKeys that configure the BlueCore device.

2.3 Tools to orchestrate the build process

- **appquery**

  When invoked, this utility application generates a list of the API dependencies for the code passed to it.

  During the Build process in BlueLab `make` invokes `appquery` and passes it the firmware selected by `make` as a result of a query of the BlueCore device performed using `BlueFlashCmd`.

  The output is then used to determine whether the firmware supports the read-only file system. If it does `make` will evoke `packfile` to produce a `.fs` file to pass to `vmbuilder`.

  Alternatively, if the firmware does not support the read-only file system the code will be passed to `vmbuilder` as an `.app` file, rather than as a `.fs` file.

- **make**

  This is the standard GNU `make` utility. The application executes a list of shell commands associated with each target to create or update a file of the same name. It uses the `makefiles` (shipped with BlueLab) to determine how the target is updated with regard to its dependencies.

  See http://www.gnu.org/software/make/manual/make.html for details concerning `make`.

- **recordflags**

  This application records options passed to other tools so that when the options are changed the record can be used to force code to be recompiled.

The following are small helper applications that are called, as required, by `make` during the build process and whose purpose is largely self-evident.

- **copyfile**
- **createpath**
- **remove**
- **workingdirectory**

2.4 Tools to package code

These utilities are automatically invoked, as required, by `make` during the build process:

- **kalpac2**

  This application generates the `.kap` files from output generated by kalasm2.

- **packfile**

  This application generates a `.fs` file from a directory tree.

- **unpackfile**

  This application generates a directory tree from a `.fs` file. (unpackfile is not used by makefiles supplied with BlueLab.)

- **vmBuilder**

  This application combines the `.fs` or `.app` file with the BlueCore firmware. The output is passed to BlueFlashCmd to download the final code to a BlueCore device.

2.5 Tools to compile, assemble and link BlueLab code

These utilities are automatically invoked,as required, by `make` during the build process.

- **xap-local-xap-ar**

  This is a cut-down version of the standard `ar` utility that is used to build Library files. See http://www.computerhope.com/unix/uar.htm#01 for a description of the standard `ar` utility.

  The limited number of command line options supported in BlueLab's implementation of `ar` are listed in Chapter 9.

- **xap-local-xap-gcc-3.3.3**

  This is a slightly modified version of the standard GNU compiler, although some minor adaptations have been made to optimise the performance for use with BlueLab applications. The standard implementation is described at http://gcc.gnu.org/onlinedocs/gcc-3.3.5/gcc/

- **kalasm2**

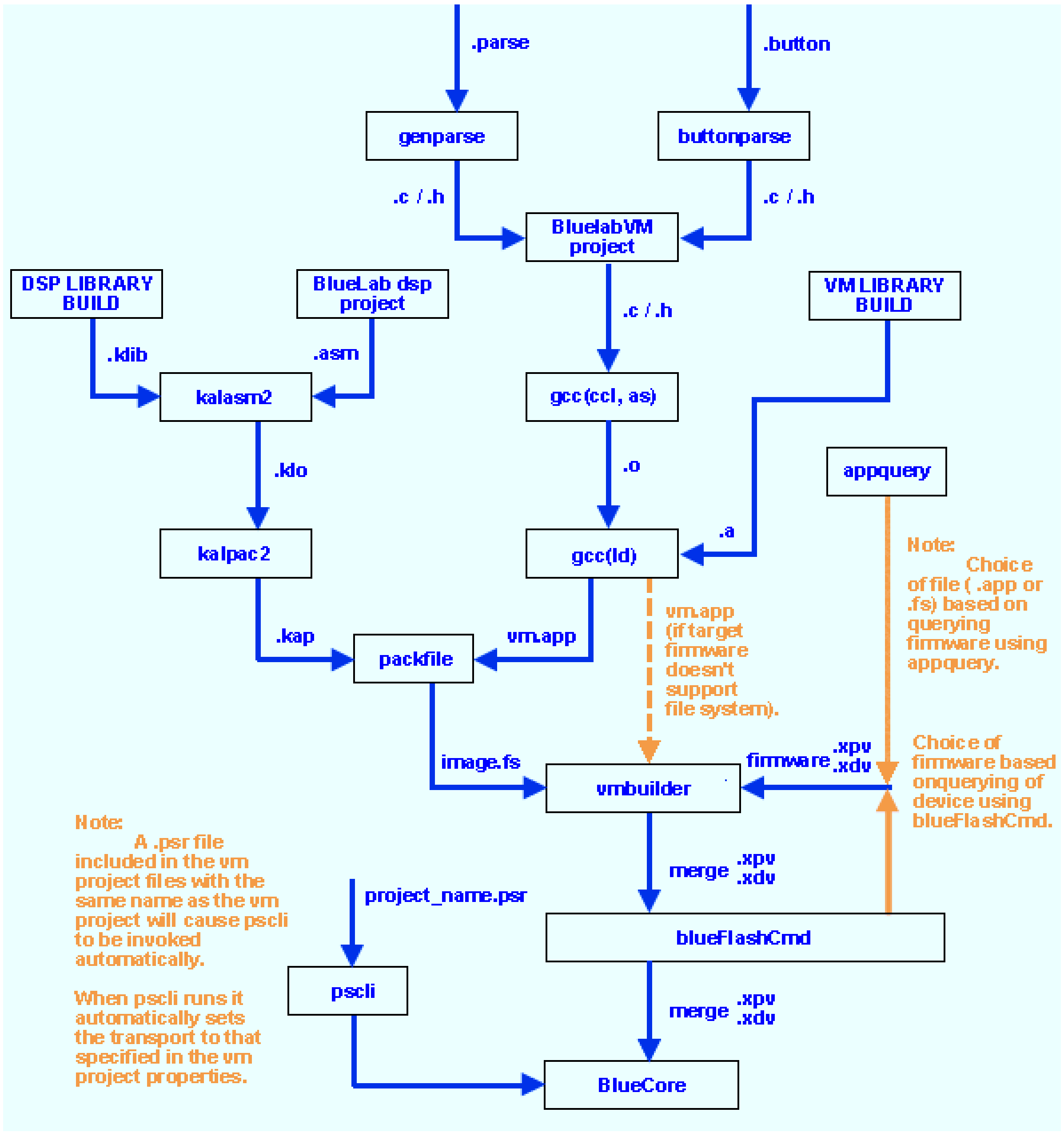
  kalasm2 is the Kalimba compiler. It accepts `.asm`, `.kobj` and `.klib` input files and produces `.klo` output files for the DSP on BlueCore Multimedia chips.

  Running `kalasm2.exe` in a command window will display the application help, which details the application options, and command availability.

  Kalasm2 is described in more detail in the Kalimba DSP Assembler User Guide (CSR reference: blab-ug-002P).

3 Build Process Overview

The following figure gives an overview of the utility applications and their use during the build and download processes as implemented by BlueLab.


.parse
.button
genparse
buttonparse
.c / .h
.c / .h
BluelabVM project
DSP LIBRARY BUILD
BlueLab dsp project
VM LIBRARY BUILD
.c / .h
.klib
.asm
kalasm2
gcc (ccl, as)
.klo
.o
appquery
kalpac2
gcc(ld)
.a
Note: Choice of file (.app or .fs) based on querying firmware using appquery.
.kap
packfile
vm.app
vm.app (if target firmware doesn't support file system).
image.fs
firmware
.xpv
.xdv
vmbuilder
Choice of firmware based onquerying of device using blueFlashCmd.
Note: A .psr file included in the vm project files with the same name as the vm project will cause pscli to be invoked automatically.
project_name.psr
merge
.xpv
.xdv
blueFlashCmd
pscli
When pscli runs it automatically sets the transport to that specified in the vm project properties.
merge
.xpv
.xdv
BlueCore

4 buttonparse

The buttonparse 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.

4.1 General

`Buttonparse` takes a `.button` file containing a description of the required buttons and outputs the `.c` and `.h` files required to implement them i.e:

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

4.2 Description of .button files

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

```
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:

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

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

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

```
message VOLUME_DOWN
DOWN enter
```

3. 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
```

4. To send a message (OFF) when a PIO (5) is held down for a specified period (2 seconds) i.e. 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
```

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

```
message SHORT_PRESS
OFF release
```

Note: If a HELD event (i.e. a long press, if defined) is detected and a message is sent as a result, it will cancel any release messages. This allows short and long press messages to be specified.

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

```
message DOUBLE_PRESS
UP double 1000
```

4.2.1 Functionality added in BlueLab 3.3

The buttonparse utility has been extended in BlueLab v3.3 to include the following additional functionality:

1. An additional label `held_release` has been added to enable 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 `held` event is defined and a message has been sent the `held_release` message is suppressed (if a `release` event is defined the SHORT_PRESS message will be suppressed if either a `held_release` or `held` message has been sent).

2. 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.

   For example, it is used in the Stereo Headset application shipped with BlueLab v3.3 to monitor PIOs 13 and 15 to determine the battery charging status.

   To receive the PIO message in this form statements are added to the top of the `.button` file e.g:

   ```
   pio_raw 13
   pio_raw 15
   ```

   The `PIO_RAW` message received by the task will contain the state of both PIOs and can be read for each by looking at the relevant bit position for the required PIO e.g:

   `((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.

4.2.2 Functionality added in BlueLab 3.4

The buttonparse utility has been extended in BlueLab 3.4 to include support for:

- Active-low PIOs
- External event recognition

Active-low PIOs

Button presses are supported by the buttonparse utility as active-low when the PIO line concerned is specified in the PIO_WAKEUP_STATE PS Key.

Note: PIO_WAKEUP_STATE is only supported on BlueCore3 variants or later.

The PIO_WAKEUP_STATE PS Key holds an uint16 which acts as a bitmask mapping to the available PIO lines. When a bit is set to 1 the corresponding PIO is treated as active-low.

Example

To implement PIO 2 as active-low PS Key PIO_WAKEUP_STATE would be set to 0x4 (binary 0000 0000 0000 0100).

External event recognition

In BlueLab 3.4 external events can be passed from the application into the buttonparse state machine. This allows button presses for buttons not connected to a PIO to be utilised by an application.

Example:

To generate a message (CONNECT) which will be sent to the application when an external button is released after a specified period (2 seconds) the following code is required.

In the .button file:

```
message CONNECT
external 0 held_release 2000
```

In the VM application code:

```
if (button_pressed)
     pioExternal (piostate, (1<<0), (1<<0)); /* Button pressed
     */
else
     pioExternal (pioState, (1<<0), 0); /* Button released */
```

Description of pioState function parameters

The `pioExternal` function takes the following parameters:

1. A pointer to the pioState structure, this is the same as the first parameter used when calling `pioInit()`.
2. External events to clear, passed as a bitmap.
3. External events to toggle, passed as a bitmap.

Thus to:

1. **Set an event:** `pioExternal (pioState, (1<<Event), (1<< Event))`
2. **Clear an event:** `pioExternal (pioState, (1<<Event), 0)`
3. **Toggle an event:** `pioExternal (pioState, 0, (1<<Event))`

5 genparse

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

5.1 General

`genparse` takes one or more `.parse` file(s) 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 BlueLab `make` will identify a `.parse` file when included in a BlueLab Project and call `genparse.exe` before compiling and linking the application code.

5.2 Description of .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 (which can be found in `C:/BlueLab_3.x/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 wishes to support additional commands these can be added as required.

5.2.1 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 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. See section 5.2.2.
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 i.e. 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 of an opening or closing bracket in the input. Does not differentiate between them i.e. 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 of an equals sign or a colon in the input. Does not differentiate between them i.e. 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.
`#` (Bluelab 3.2 and earlier) or `//` (blueLab 3.3 and later)	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.

5.2.2 Parameters

A % in a pattern should be followed by either `s:name`, or `d:name` indicating a 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) will be passed to the handler, otherwise the pattern will fail and the next one is considered.

A numeric parameter matches a non-empty sequence of decimal digits in the input. If it succeeds, the decimal value of the sequence will be passed to the handler, otherwise the pattern will fail 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 within 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 genpars. In this case the parameter %* can be used. This parameter will pass the full string making up the command from %* on.

5.2.3 Literals

Each literal is compared exactly against a complete packet, including case and spaces. If it matches then the corresponding handler will be 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 capitalisation. 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 optimisation, which can be added later.

5.2.4 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 which 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.

5.3 Description of 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.

5.4 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 surround by carriage-return line-feed pairs. This covers the AT command syntax allowed by ETS 300 916 (GSM 07.07).

5.5 Examples

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

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

   e.g. `{\r \n OK \r \n} : hfphandleOk`

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

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

   e.g. `{ \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 *);
```

3. 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:

   e.g. `{ \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 *);
```

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

   e.g.`{ \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 structures that accommodate up to 99 comma (,) separated strings, which are passed a 'p' to the helper function.

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

Note: 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);
```

Note: 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 *);
```

Note: 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 const; ++i)

   {

      value_hfpHandleCallInfo y = get_hfpHandleCallInfo_p (&x->p, i);

      print (y.n.data, y.n.length);

   }

   void print (const uint8 *s, uint16 n)

   while

   {

     uint16 i; for (i=o, i< n; ++i)putchar (s[i]);

     putchar ('\n')

   }

}
```

BlueLab™ a guide to BlueLab Command Line Tools

5.6 Functionality added in BlueLab v3.4

Support for multiple `.parse` files was added in BlueLab v3.4.

In order to differentiate function names in two or more `.parse` files, engineers can specify a prefix to make the function calls unique when processed by genparse.

Example

Adding the statement:

```
prefix Label
```

will result in the function names in the `.parse` file to be treated in genparse as:

- `LabelParseData`
- `LabelParseSource`
- `LabelHandleUnexpected`

Using a unique prefix for each `.parse` file avoids conflict in function calls when multiple `.parse` files are being used in an application.

6 BlueFlashCmd

BlueFlashCmd is a command line implementation of the BlueFash application and can be used to download or dump code to / from the BlueCore chip.

In the context of this document the main use of BlueFlashCmd is during Run in BlueLab, where `make` invokes it to query the device to determine the correct choice of firmware and to finally download the `merge .xpv` and `.xdv` pair onto the BlueCore chip.

The BlueFlashCmd 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, CSR recommend using BlueFlash which provides the same functions with a more user-friendly interface.

Note: BlueFlashCmd is primarily intended for use by BlueLab e.g. when downloading code to a development platform. CSR supply a separate SDK library TrueTest which can be used to write applications better suited to downloading code in a production environment. Please contact CSR for further details.

7 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 chips in much the same way.

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

Important: During Run pscli automatically sets the chip's transport to that specified in the Project Properties of the BlueLab project. i.e. if the engineer has manually changed the chip's transport using PSTool the transport will be overwritten when pscli is invoked unless the transport is also changed in the Project Properties.

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

CSR recommend that PSTool is used when manually manipulating PSKeys during the development process and that pscli is only used as part of BlueLab.

The use of PSTool including the syntax for `.psr` and `.psq` files is described in the PSTool User Guide (CSR ref: blab-ug-008P).

Note: pscli is primarily intended for use by BlueLab e.g. when downloading PSKey values to a development platform. CSR supply a separate application TrueTest which is better suited to downloading code in a production environment. Please contact CSR for further details.

8 make

8.1 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 BlueLab 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 chip.

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

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

Note: BlueLab calls `make` when a Build (F7) or Run (F5) operation is performed in xIDE.

8.2 General

When `make` is invoked by BlueLab it normally orchestrates preprocessing, compilation, assembly and 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 BlueLab, the makefile generated by xIDE automatically passes the necessary command line options and file names to correctly compile and link BlueLab Projects and to download the resultant code to the BlueCore chip.

In general BlueLab 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: CSR has tested the combination of command line options used by Bluelab.– Users who want to generate their own makefiles should be aware that using options other than those used by BlueLab may expose subtle problems. In particular, invoking higher optimisation levels (-O2 and -O3) in gcc are known to cause some issues.

9 xap-local-xap-ar

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

- **Making a library**

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

  For example:

  `ar r library.a obj1.o obj2.o` adds the two 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 are reported as an error.

Document References

Document	Reference
PSTool User Guide	CSR reference blab-ug-008P
Kalimba DSP Assembler User Guide	CSR reference blab-ug-002Pd

Terms and Definitions

BlueCore™	Group term for CSR's range of Bluetooth wireless technology chips
Bluetooth®	Set of technologies providing audio and data transfer over short-range radio connections
Bluetooth SIG	Bluetooth Special Interest Group
CSR	Cambridge Silicon Radio
DSP	Digital Signal Processor
PIO	Parallel Input/Output
Rfcomm	Serial Cable Emulation Protocol

Document History

Revision	Date	History
a	30 SEP 05	Original publication of this document. (CSR reference: blab-ug-009Pa)
b	28 JAN 06	Buttonparse and genparse sections updated to reflect additional functionality provided in BlueLab v3.4

BlueLab™

A guide to Bluelab Command Line Tools

User Guide

blab-ug-009Pb

January 2006