1 Introduction

This document describes the MIDI features of Behringer’s ‘B-Control’ devices BCF2000 and BCR2000 under firmware version 1.10. However, it goes way beyond what you normally expect from a MIDI implementation document: there are explanations of all global and preset parameters, tables and descriptions of special functions, etc. It also provides a nearly complete description of BCL, Behringer’s MIDI-embedded language that controls the BCF/BCR2000.

Regrettably, Behringer have never released details of the MIDI implementation of the BCF/BCR2000, so everything in this document is based on third-party investigations, a.k.a. reverse engineering. Consequently, certain details may be wrong, and I accept no responsibility for any damage caused by any errors in this document.

This document owes a lot to Michael Kukat’s B-Control-Konfiguration\(^1\) and B-Control-Tokenreferenz\(^2\) web pages about BCL, and also to the work by Royce Craven in the Yahoo BC2000 users group. I also thank Royce for providing feedback to version 0.1 of this document, and apologize for taking so long to process his suggestions.

In this document, a sequence of three asterisks (***) indicates a passage that is still incomplete or inaccurate. Any comments, suggestions or corrections are welcome.

All references to Behringer’s official B-Control manual concern version 1.1, October 2004.\(^3\)

\(^1\) https://www.sequencer.de/synth/index.php/B-Control-Konfiguration

\(^2\) https://www.sequencer.de/synth/index.php/B-Control-Tokenreferenz

\(^3\) I have used both the German and English versions, BCF2000_BCR2000_GER_Rev_C.pdf and BCF2000_BCR2000_ENG_Rev_C.pdf respectively. In a few places there are important differences between the two, as discussed.
## Contents

1. Introduction ................................................................. 1
2. Contents ................................................................ 2
3. Terminology .................................................................. 5
4. Notational conventions .................................................... 6
5. Document version history .................................................. 7
6. MIDI System Exclusive messages ......................................... 9
   6.1 Commands .................................................................. 10
7. BCL messages ................................................................ 13
   7.1 MIDI format .............................................................. 13
8. BCL text ........................................................................ 14
   8.1 Case-sensitivity .......................................................... 14
   8.2 Spaces ...................................................................... 14
   8.3 Empty lines ............................................................... 14
   8.4 Comments ................................................................. 14
   8.5 Numbers .................................................................... 14
9. BCL blocks ...................................................................... 15
   9.1 Block Start statement .................................................. 16
   9.2 Block End statement .................................................... 17
10. BCL sections ................................................................. 18
11. Side-effects of BCL section selector statements ....................... 19
   11.1 Reinitialization of settings ............................................ 19
   11.2 Multiple occurrences of the same section selector statement.................................................. 20
   11.3 Invalid element selector statements ................................ 21
12. Global setup ................................................................... 22
   12.1 MIDI mode .............................................................. 23
   12.2 Startup preset ........................................................... 24
   12.3 Foot switch ................................................................. 25
   12.4 Receive channel .......................................................... 26
   12.5 Device ID ................................................................. 27
   12.6 Transmission interval .................................................. 28
   12.7 Dead time ................................................................. 29
   12.8 Factory defaults .......................................................... 30
13. Presets ......................................................................... 31
   13.1 Name ...................................................................... 32
   13.2 Snapshot ................................................................. 33
   13.3 Request .................................................................... 34
   13.4 Encoder groups ......................................................... 35
   13.5 Function keys ............................................................ 36
   13.6 Lock ........................................................................ 37
   13.7 LEARN output .......................................................... 38
   13.8 Initialization of all elements ........................................... 39
14. Control elements ......................................................... 42
   14.1 Standard output ........................................................ 43
      14.1.1 GS/XG Main Control parameters ............................... 43
   14.2 Show value ............................................................... 44
   14.3 Default value ............................................................ 45
   14.4 Current value ............................................................ 46
   14.5 Physical mapping ....................................................... 47
14.6 Custom output
  14.6.1 Data Specifier
  14.6.2 Change Definition
  14.6.3 Checksum Definition
  14.6.4 Direction Specifier
  14.6.5 Repeat
  14.6.6 Length of custom MIDI output definitions
14.7 Local
14.8 Standard vs. custom MIDI output
14.9 Value synchronization
  14.9.1 Button increment mode
  14.9.2 Encoder resolutions
  14.9.3 Frankenstein faders (a.k.a. fader calibration test)
15 Buttons
  15.1 Standard output
  15.2 Program Change
  15.3 Control Change
  15.4 NRPN (Non-Registered Parameter Number)
  15.5 Note
  15.6 Aftertouch
  15.7 MMC (MIDI Machine Control)
  15.8 GS/XG
  15.9 Mode
  15.10 Increment mode
16 Continuous elements (encoders/faders)
  16.1 Standard output
  16.2 Program Change
  16.3 Control Change
  16.4 NRPN (Non-Registered Parameter Number)
  16.5 Pitch Bend
  16.6 Aftertouch
  16.7 GS/XG
17 Encoders
  17.1 Standard output
  17.2 Mode
  17.3 Resolution
18 Faders
  18.1 Standard output
  18.2 Motor
  18.3 Override
  18.4 Key-override
19 Memory presets
  19.1 Recall
  19.2 Store
  19.3 Preset selection
20 Unknown dot statements
21 BCL Reply messages
  21.1 MIDI format
  21.2 Error codes
    21.2.1 Error 0
    21.2.2 Error 1
3 Terminology

BCF:
Abbreviation of the BCF2000 device.

BCR:
Abbreviation of the BCR2000 device.

BC:
Abbreviation used for a ‘generic’ device, i.e. when something applies to both the BCF and the BCR. (But beware: most of my testing was only performed on a BCR, so it is not always certain that the use of the word ‘BC’ (rather than ‘BCR’) is warranted.)

B-Control Language (BCL):
The language used by the BCF2000 and BCR2000 in SysEx messages using command $20. Note: ‘BCL’ is probably indeed Behringer’s ‘official’ name, as demonstrated by the file BCL.class in the B-Edit Java package: this file contains the identifiers for this language. However, outside the Behringer company nobody knows what BCL stands for; so that it stands for ‘B-Control Language’ is only an educated guess on my part.

BCL line:
A line of text in BCL format.

BCL statement:
A BCL line that actually does something (rather than being empty or containing just a comment).

BCL dollar statement:
A BCL statement starting with ‘$’. Semantically there are three subcategories:
1. Block statements: $rev and $end.
2. Section selectors: $global, $preset, $button, $encoder and $fader.

BCL dot statement:
A BCL statement starting with ‘.’. Each dot statement type can only occur after the appropriate section selector type(s). A dot statement affects the state of the current section (i.e. global setup, preset, button, encoder or fader).

BCL message:
A MIDI message (using command $20) containing a BCL line. (Technically the BCL line is embedded in the MIDI message.)

BCL message chain:
A sequence of BCL messages, where the first message’s index is zero and each message’s index is 1 higher than that of the previous message.
4 Notational conventions

- BCL text is written in Courier **bold** typeface.
- Variables are written in *italics*.
- In syntactic definitions, optional elements are written between square brackets, e.g. [Comment]. The brackets themselves should never be included in actual BCL text.
- Hexadecimal numbers (usually bytes) are written with the prefix ‘$’, except in a few cases where hexadecimal byte sequences are simply shown in Courier typeface.
- Arithmetic and bit operators are written in **bold** typeface: div, mod, and, or, shr.
5 Document version history

Version 1.2.9 (2017-10-17):
- The spurious output generated by certain .tx statements is discussed in much more detail in §13.7. §14.6.3 has been cleaned up in this regard too.
- §14.6.3 mentions Fractal Audio’s use of cks-3.
- The response by a button using .easypar NOTE to incoming MIDI messages is described.
- The description of the sign-and-magnitude method used by relative-3 and relative-3/14 has been corrected.
- It is mentioned that inc/dec for an encoder or fader using .easypar CC triggers value synchronization.
- Discussion of the fact that a .resolution statement with a parameter higher than 32673 causes the encoder to behave erratically.
- Quite a few minor stylistic improvements.

Version 1.2.8 (2016-02-08):
- Extra information (provided by Royce Craven) on the various SysEx checksum methods has been added to §14.6.3.
- All instances of ‘q.v.’ have been replaced with references to the pertinent sections.
- Minor stylistic improvements.

Version 1.2.7 (2015-03-27):
- The introduction has been expanded.
- The ‘Current value’ section mentions that a BCF fader only jumps to a new value automatically if .motor is on.
- The ‘Data Specifier’ section explicitly states the BC’s use of LSB bit numbering, and gives an example of a pair of Control Change messages together sending a 14-bit value.
- The Aftertouch sections mention that in Note Aftertouch messages the element’s Scope parameter functions as the note number.
- Minor stylistic improvements.

Version 1.2.6 (2012-08-23):
- The BC’s behavior upon reception of a .deviceid statement is specified.
- Minor stylistic improvements.

Version 1.2.5 (2011-10-21):
A few typos were corrected.

Version 1.2.4 (2011-03-07):
Minor cosmetic edits.

Version 1.2.3 (2010-11-05):
- Waldorf’s use of cks-2 is mentioned in §14.6.3.
- A few stylistic and terminological improvements.

Version 1.2.2 (2010-07-17):
- Description of the Send Text command.
- The internal data size of Active Sensing (SFE) in LEARN/custom output is specified.
- The sections of the BCL Reply error codes have been relegated to §21.2.
A few small cosmetic improvements.

Version 1.2.1 (2009-08-06):
- Description of the BC’s handling of BCL message chains containing more than 16384 messages.
- Description of the BC’s bug concerning the LEDs of the LEARN, EDIT and EXIT buttons after a transition from \texttt{.fkeys off} to \texttt{.fkeys on}.
- A typo has been corrected, and a few stylistic changes have been made.

Version 1.2 (2009-04-30):
- The \texttt{.easy-par CC, NRPN, AT} and \texttt{GS/XG} sections for buttons now take account of the fact that the Value2 parameter can be \texttt{off}.

Version 1.1:
- A few conceptual errors concerning \texttt{ntrimes} have been corrected.
- The interaction between \texttt{.mode incval} and \texttt{.minmax} is now described correctly.
- Some information on the BCF’s emulation mode identity messages has been added.
- The pdf file includes bookmarks for the document sections. In view of this, some additional section headings have been defined, and all references to pages have been replaced with references to sections.
- A few minor changes in terminology.
- Some typos have been corrected.

Version 1.0:
- The \texttt{.txinterval} and \texttt{.deadtime} sections have been extended. This includes the specification of the factory settings.
- The section on side-effects of section selector statements has been largely rewritten.
- The section on the Default setting has been largely rewritten, and a related section on Value has been added.
- The incorrect claims about the use of apostrophes in preset names (cf. \texttt{.name}) have been corrected.
- The facts concerning bare element selector statements are described.
- The \texttt{.tx} section for control elements has been worked out.
- The newly discovered \texttt{incval} parameter to the button \texttt{.mode} statement is described.
- The description of the button \texttt{increment} algorithm has been improved by taking the Default setting into account.
- Several new topics are discussed in depth: value synchronization, startup functions and functions in standard B-Control mode (in particular Panic Reset).
- Some typos have been corrected.
- The layout of a few tables has been improved.
- Numerous other corrections and additions have been made.

Version 0.1: First published version.
6 MIDI System Exclusive messages

When functioning in standard B-Control mode (cf. §22), the BCF2000 and BCR2000 use the following format for MIDI System Exclusive messages:

<table>
<thead>
<tr>
<th>Item description</th>
<th>MIDI byte(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Exclusive</td>
<td>$F0</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>$00 $20 $32 (=Behringer)</td>
</tr>
<tr>
<td>Device ID</td>
<td>$0x (0..15) (=actual device’s ID – 1) or $7F (=any)</td>
</tr>
<tr>
<td>Model</td>
<td>$14 (=BCF2000) or $15 (=BCR2000) or $7F (=any)</td>
</tr>
<tr>
<td>Command</td>
<td>0bbbbbbb (see table on following pages)</td>
</tr>
<tr>
<td>Data</td>
<td>sequence of zero or more bytes (0bbbbbbb); depends on Command</td>
</tr>
<tr>
<td>End-Of-Exclusive</td>
<td>$F7</td>
</tr>
</tbody>
</table>

Any BCF2000 or BCR2000 responds to the $7F-wildcards for Device ID and Model, but only sends its own Device ID and Model. Each device’s Device ID can be changed via its Global Setup.
### 6.1 Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Data</th>
<th>To BC</th>
<th>From BC</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$01</td>
<td>Request Identity</td>
<td>–</td>
<td>⬤</td>
<td>–</td>
<td>BC-EDIT uses Device ID $7F for this command, twice per MIDI output device: first Model $14, then Model $15. I have verified that using Device ID $7F and/or Model $7F works too. Reply: Send Identity.</td>
</tr>
<tr>
<td>$02</td>
<td>Send Identity</td>
<td>Identity string (e.g. ‘BCF2000 1.10’ or ‘BCR2000 1.10’)</td>
<td>–</td>
<td>⬤</td>
<td>Reply to Request Identity.</td>
</tr>
<tr>
<td>$20</td>
<td>Send BCL Message</td>
<td>Text line (0 to abt. 512 characters)</td>
<td>⬤</td>
<td>⬤</td>
<td>See §7. Reply: BCL Reply.</td>
</tr>
</tbody>
</table>
| $21     | BCL Reply | 1. Message index (2 bytes, i.e. 14 bits, MSB first)  
|         | Send Preset Name | 1. Zero (1 byte)  
2. Preset index (1 byte): 0..31/$7F  
3. Preset name (exactly 24 characters; *not* between quotes) | – | ⬤ | This uses the same command byte ($21) as BCL Reply, but the message *length* is different. |
<p>| $22     | Select Preset | 0..31: single memory preset (−1) | ⬤ | – | BC responds by selecting the memory preset (but does <em>not</em> send a MIDI reply message). If the preset value is out of range, absolutely nothing happens. |
| $34     | Send Firmware | 296 bytes (encrypted) | ⬤ | – | Cf. bcf2000_1-10.syx etc. |</p>
<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Data</th>
<th>To BC</th>
<th>From BC</th>
<th>Comments</th>
</tr>
</thead>
</table>
| $35     | Firmware Reply | 1. 2 bytes (i.e. 14 bits, MSB first), indicating memory address (divided by $100) of firmware in 16th Send Firmware message  
2. Error code (1 byte): 0 = no error; 1= error; other values never encountered yet | - | ● | Reply to every 16th Send Firmware message. As far as I’ve seen, whenever the error code is 1, the BC’s display briefly shows ‘Err5’. This makes sense, considering that BCL Reply error 5 means “wrong revision” (see §21). |
| $40     | Request Data | 0..31: single memory preset (MemoryPreset - 1)  
$7E: global setup + all filled memory presets  
$7F: temporary preset | ● | - | Reply: BCL message chain (not including $store MemoryPreset immediately before $send).  
Reply: BCL message chain.  
Reply: BCL message chain. |
| $41     | Request Global Setup | - | ● | - | Reply: BCL message chain. |
| $42     | Request Preset Name | 0..31: single memory preset (-1)  
$7E: all memory presets (not just the filled ones!)  
$7F: temporary preset | ● | - | Reply: Send Preset Name (same preset index).  
Reply: 32 separate Send Preset Name (preset indexes 0..31).  
Reply: Send Preset Name (preset index $7F). |
| $43     | Request Snapshot | - | ● | - | Requests current element values.  
Reply: snapshot (see §23.4). |
<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Data</th>
<th>To BC</th>
<th>From BC</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$78</td>
<td>Send Text</td>
<td>1. index of first character (0..111(?))</td>
<td></td>
<td>●</td>
<td>Passed-on text message from controlling device in any BCF’s emulation mode except Baby HUI.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. characters</td>
<td></td>
<td></td>
<td>(The BCFView utility interprets this as two lines of 56 characters each.)</td>
</tr>
</tbody>
</table>
7 BCL messages

7.1 MIDI format

<table>
<thead>
<tr>
<th>Item description</th>
<th>MIDI byte(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Exclusive</td>
<td>$F0</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>$00 $20 $32 (=Behringer)</td>
</tr>
<tr>
<td>Device ID</td>
<td>$0x (0..15) (=actual device’s ID – 1) or $7F (=any)</td>
</tr>
<tr>
<td>Model</td>
<td>$14 (=BCF2000) or $15 (=BCR2000) or $7F (=any)</td>
</tr>
<tr>
<td>Command</td>
<td>$20 (=BCL message)</td>
</tr>
<tr>
<td>Index MSB</td>
<td>0bbbbbbb</td>
</tr>
<tr>
<td>Index LSB</td>
<td>0bbbbbbb</td>
</tr>
<tr>
<td>BCL text line</td>
<td>sequence of 0 or more* characters (#32..#127)</td>
</tr>
<tr>
<td>End-Of-Exclusive</td>
<td>$F7</td>
</tr>
</tbody>
</table>

Index MSB and Index LSB together constitute a 14-bit index from 0 to 16383. The first BCL message in a message chain (normally a $rev statement) should have index 0, and each message’s index should be 1 higher than that of the previous message. (The last message in a chain is normally ‘$end’.)

The BC’s handling of message chains consisting of more than 16384 messages is inconsistent:

- The only case in which the BC may need to send more than 16384 BCL messages is in response to a data request for its global setup and filled memory presets (Request Data (command $40, subfunction $7E). In this case the BC wraps the index back to 0 after it has reached 16383.
- However, when receiving data, the BC does not accept a BCL chain consisting of more than 16384 messages. Instead, whenever the BC receives a message containing index 0, it interprets this as the start of a new chain. Consequently, the BC then expects a $rev statement, and the statements that continue the intended ‘wrapped’ chain tend to lead to BCL Reply errors 8 and 6 (in that order). Thus, the BC cannot read back a wrapped chain that it has produced itself upon a data request for its global setup and filled memory presets!

* There is (roughly) a 512-character limit for BCL text lines, so probably the BC uses a 512-byte buffer for BCL text lines.
8 BCL text

8.1 Case-sensitivity

All BCL identifiers are case-sensitive. The BC rejects any BCL identifier containing incorrect case, and
returns a BCL reply message specifying an error related to the offending identifier.

The only exception to this constraint are the hexadecimal digits ‘A’ .. ‘F’, which may be sent to
the BC in either upper or lower case. The BC itself always outputs these in upper case.

8.2 Spaces

Extra spaces may be inserted anywhere between identifiers in a BCL statement.

The BC itself always indents any dot statement by means of two spaces. However, these spaces
may be left out when you send any dot statement to the BC.

8.3 Empty lines

Empty lines may be inserted anywhere in a BCL chain. The BC itself never sends empty lines.

8.4 Comments

Syntax: \[BCL statement]\;[Comment]

BCL lines sent to the BC can include any comments after a semicolon (‘; ’). (Other characters like hash
(‘#’) and double slash (//), which also often function as comment initiators in computer languages (and
indeed in several BC utilities), are not allowed in this respect.)

For instance:

$store 1; this stores the current preset
; this is a self-referential comment

A semicolon inside .name’s PresetName argument (e.g. ‘.name \ abc; xyz ‘) is interpreted
as a normal character, i.e. not as a comment initiator.

The BC itself never sends comments. (So comments in BCL lines sent to the BC are not retained
when the BC sends the ‘same’ BCL lines back to the computer, e.g. via a preset dump.)

8.5 Numbers

Positive numbers cannot be preceded by ‘+’.
9 BCL blocks

A BCL block is a sequence of BCL lines, normally starting with a \$rev\ statement and ending with a \$end\ statement.

A standard BCL block has the following structure:

Block Start statement
[Embedded lines]
Block End statement

A BCL chain sent \textit{from} the BC consists of exactly one standard block that contains at least one section. However, BCL chains sent \textit{to} the BC may deviate from this in several ways:

1. A chain may consist of a sequence of blocks.
2. The Block End statement may be absent from a block.
3. A block may consist of only a Block End statement. (This is of course totally meaningless, but the BC reports no error.)
9.1 Block Start statement

Syntax:
To the BCF2000: $\texttt{rev F}[\textit{Revision}]$
To the BCR2000: $\texttt{rev R}[\textit{Revision}]$
From the BCF2000: $\texttt{rev F1}$
From the BCR2000: $\texttt{rev R1}$

$\texttt{rev}$ requires a single argument. Normally this argument consists of two parts (the second is actually optional):

1. \textit{Model}: a single character, indicating the type of the hardware device (BCF2000 or BCR2000) that the subsequent BCL lines address. ‘F’ stands for the BCF2000, ‘R’ for the BCR2000. Any other character than the one expected by the receiving hardware device causes BCL error 4. In accordance with the general BCL case-sensitivity rule, the BCF and BCR do not accept lower case for ‘F’ and ‘R’ respectively.

2. \textit{Revision}: the version of BCL that is going to be used in the subsequent BCL lines.
   Normally \textit{Revision} is a single ‘1’: the BCF and BCR themselves always send this. However, the BCF and BCR accept any sequence of characters, provided that the first character is \textit{not} ‘0’ (this results in BCL error 5). So a BCR accepts nonsense like ‘$\texttt{rev Rh38f!hP*d0gf084t}$’, but not ‘$\texttt{rev R0}$’ or ‘$\texttt{rev R01}$’! (The reason for the exclusion of the starting ‘0’ may be that the earliest BCF/BCR firmware version(s) (at least before 1.07) had ‘0’ here, although this is pure speculation.)
   \textit{Revision} may also be left out altogether, so you can send simply ‘$\texttt{rev F}$’ and ‘$\texttt{rev R}$’ to the BCF and the BCR respectively.
9.2 Block End statement

Syntax: $end

Nitpicker’s note: as indicated above, $end should have no arguments. However, the BC responds inconsistently to any ‘garbage’ after $end, in ways similar to its response to garbage in $preset statements (see §13).
10 BCL sections

A BCL section is a sequence of BCL lines pertaining to a particular section of the BC’s memory (either the global setup area or a subsection of the temporary preset). Any BCL section must be embedded in a BCL block (basically this means that there must be a previous $rev statement).

A BCL section starts with a section selector statement, which can be one of the following:

$global
$preset
$button Button
$encoder Encoder
$fader Fader

A BCL section ends immediately before the next dollar statement (in other words, BCL does not have a dedicated section-ending statement comparable to $end for BCL blocks). A BCL section contains zero or more dot statements.

The order of sections in a preset dump from a BC is as follows:
1. The preset section.
2. The encoder sections (from low to high) for those encoders that contain one or more non-default settings.
3. The button sections (similarly).
4. BCF only: the fader sections (similarly).
11 Side-effects of BCL section selector statements

11.1 Reinitialization of settings

All section selector statements except $global (i.e. $preset, $button Button, $encoder Encoder and $fader Fader) reset all the indicated section’s settings.

A $global section without any dot statements does absolutely nothing:

```
$rev R1
$global ; doesn't change any global settings
$end
```

And here the only global settings affected are TransmissionInterval and DeadTime:

```
$rev R1
$global
   .txinterval 10
   .deadtime 50
$end
```

The values of the other global setup settings (MidiMode, StartupPreset etc.) are not affected here.

On the other hand, all preset settings are reset after

```
$rev R1
$preset ; resets all preset settings
$end
```

In particular, Name is set to 24 spaces, EncoderGroups to 4, Snapshot, Request and Lock to off, FunctionKeys to on, and any .minmax or .tx settings are removed.

However, $preset does not clear the preset’s elements (buttons, encoders and faders): that can only be achieved by a .init statement in the $preset section:

```
$rev R1
$preset ; resets all the preset’s settings (Name etc.)
   .init ; clears all the preset’s elements (buttons/encoders/faders)
$end
```
11.2 Multiple occurrences of the same section selector statement

It is syntactically legal for a BCL chain to contain the same section selector statement more than once. However, the fact that preset and element section selector statements clear their respective settings means that only the dot statements after the last copy of a particular section selector statement actually stick. Any values set in previous sections introduced by the same selector statement are reset to their defaults. So in effect you cannot send a sequence of `partial` sections for presets, buttons, encoders or faders.

This constraint is particularly relevant to the related pair of `.request` and `.tx`. Consider the following BCL block:

```
$rev R1
$preset
  .request on
$preset
  .tx $F0 $F7
$store 32
$end
```

The problem here is that the second `$preset` statement causes the BCR to reset `Request` to off. Consequently, contrary to what you might expect (cf. §19.3), the BCR will not send the MIDI bytes specified in the `.tx` statement upon selection of preset 32.

On the other hand, since a global setup selector statement does not reset any global settings, you can safely send a sequence of partial `$global` sections. For instance:

```
$rev R1
$global
  .txinterval 10
$global
  .deadtime 10
$end
```

This block has exactly the same effect as the example in §11.1 where the `.txinterval` and `.deadtime` statements occur under a single global setup selector statement.
11.3 Invalid element selector statements

As described earlier, any element selector statement (\$button, \$encoder or \$fader) needs exactly one argument: Button, Encoder or Fader, respectively. This should be the number of an existing element of the specified type. Thus, there are two ways in which an element selector statement can be invalid:

1. The argument is missing altogether. In this case the BC replies with BCL error 14: ‘Invalid number of arguments’.

2. The number indicated by the argument is out of range. In this case the BC replies with BCL error 9: ‘Element number out of range’.

However, in both cases there is still a very noticeable effect: the most recently selected element of the specified type gets reselected, and any further statements in the same BCL block are processed accordingly. (Interestingly, the reselected element’s settings are not initialized.)

This reselection occurs even if the previous selection occurred via an element selector statement in a previous BCL chain. In other words, the BC maintains Button, Encoder and Fader permanently.

In this respect it is also noteworthy that a .init statement (in a $preset section) has the side-effect of setting Button, Encoder and Fader to their respective highest existing elements: so on the BCF Button becomes 64, Encoder 32 and Fader 9, and on the BCR Button becomes 64 and Encoder 56. The same values are selected after the BC has been switched on: this is understandable, since the BC automatically copies a memory preset to the temporary preset.
12 Global setup

A global setup section in a BCL block is introduced by a global setup selector statement:

BCL syntax: $global

Nitpicker’s note: as indicated above, a global setup selector statement should have no arguments. However, the BC responds inconsistently to any ‘garbage’ after $global, in ways similar to its response to garbage in preset selector statements (see §13).

A global setup section sent by a BC contains the following dot statements (in this order):

```markdown
.midimode MidiMode
.startup StartupPreset
.footsw FootSwitch
.rxch ReceiveChannel
.deviceid DeviceID
.txinterval TransmissionInterval
.deadtime DeadTime
```

Details on these dot statements follow on the next pages.
12.1 MIDI mode

**BCL syntax:** `.midimode MidiMode`  
`MidiMode ∈ {U-1, U-2, U-3, U-4, S-1, S-2, S-3, S-4}`

This determines the BC’s MIDI signal flow to and from its MIDI and USB sockets. Refer to the B-Control manual for details.

You should be very careful when changing this setting (either via the BC’s Global Setup edit mode (via STORE + EDIT) or from a computer program), because of the software ‘rewiring’ this causes. In particular, the running USB driver may be closed and reopened, so that any running audio programs may run into trouble (because they remain connected to the now defunct BC’s USB-based MIDI devices). It is always best to close and reopen such programs immediately.
12.2 Startup preset

BCL syntax: `.startup StartupPreset

`StartupPreset ∈ {last, 1..32}`

This determines which memory preset becomes active (as the ‘temporary’ preset) when the BC gets switched on. `last` (the default) means that the memory preset that was active when the BC was last switched off becomes active again upon startup.

What actually happens when a memory preset becomes active is that all its settings (except its elements’ `Value` settings) are copied to the temporary preset. When the BC is turned off, any edits made to the temporary preset that haven’t been saved to a memory preset are lost.
12.3 Foot switch

BCL syntax: \texttt{.footsw FootSwitch}

FootSwitch ∈ \{\texttt{norm}, \texttt{inv}, \texttt{auto}\}

In terms of wiring, foot switches come in two versions: those whose electrical circuit is closed (i.e. has zero resistance) when the pedal is up (released), and those where the circuit is closed when the pedal is down (pressed).

You can connect one foot switch to the BCF and two foot switches to the BCR. On both devices the FootSwitch setting determines how the electrical signals coming from these foot switches are interpreted: The BCF and the BCR call a foot switch that has a closed circuit in the down position ‘normal’, and you should set FootSwitch to \texttt{norm} for such a switch. A foot switch having a closed circuit in the up position is considered ‘inverted’, and you should select \texttt{inv} for it. (Note: there is no industrial norm for what is ‘normal’ for foot switches: there are also audio hardware devices which consider a foot switch with a closed circuit in the up position ‘normal’!)

Alternatively you can set FootSwitch to \texttt{auto}. This causes the BC to autodetect the wiring of any foot switch when the BC is turned on (or when FootSwitch has just been set to \texttt{auto}).

In principle \texttt{auto} is just a way to relieve the user from the obligation to know the wiring of the pedals, but in one situation it is necessary to select \texttt{auto}: If you connect one normal and one inverted pedal to a BCR, and then set FootSwitch to \texttt{norm} or \texttt{inv}, one of the pedals gets interpreted incorrectly, since the single FootSwitch setting of \texttt{norm} or \texttt{inv} applies similarly to both pedals (this is definitely a design flaw of the BCR). However, if you select \texttt{auto} in this situation, the BCR autodetects each of the two different pedals correctly — so internally the BCR does maintain two settings then!

In fact, \texttt{auto} is the default for FootSwitch, and given the above considerations there is no reason to change this setting, unless for some strange reason autodetection fails for your foot switch(es).
12.4 Receive channel

BCL syntax: .rxch ReceiveChannel
ReceiveChannel ∈ {off, 1..16}

If a MIDI Program Change message is sent to the BC via MIDI input channel ReceiveChannel, the BC responds by selecting the preset corresponding with the message’s program number. (See §19.3 for more information.) If ReceiveChannel is off, the BC doesn’t respond to any Program Change messages.
12.5 Device ID

BCL syntax: `.deviceid DeviceID`

`DeviceID ∈ {1 .. 16}`

`DeviceID` is used in two ways:

1. In MIDI System Exclusive messages for BCs (see §6):
   a. Each BC always uses its own `DeviceID` in the MIDI SysEx messages it sends.
   b. Any message sent to a BC must specify that BC’s `DeviceID` or $7F$, otherwise that BC doesn’t react.

   When a BC receives a `.deviceid` message, it returns a BCL Reply message still using its old `DeviceID`. However, after this the BC switches to the new `DeviceID` immediately, so any further SysEx messages to the BC (even BCL messages in the same chain!) must use the new `DeviceID` (or $7F$, of course).

2. If a BC is in a USB-based MIDI mode (i.e. `MidiMode ∈ {U-1 .. U-4}`), the pre-Windows 7 Behringer USB drivers (in any case versions 1.1.1.0, 1.1.1.1 and 1.2.1.3) generate MIDI input and output device names containing the BC’s `DeviceID` between square brackets.

   Beware: in this situation you must be very careful: changing `DeviceID` causes the USB driver to close and reopen the BC’s USB-based MIDI input and output devices. Also see the remarks in §12.1.
12.6 Transmission interval

BCL syntax: .txinterval  TransmissionInterval
TransmissionInterval ∈ {2, 5, 10, 20, 50, 100}

Note that only the stated values for TransmissionInterval are allowed. So no rounding-down takes place: e.g. sending a value of 3 to the BC causes BCL error 11.

TransmissionInterval determines the minimum time interval in milliseconds that must elapse before the BC sends an element’s MIDI output again after the user has physically changed that element’s value for the second time in rapid succession. Note though, that if an element’s MIDI output consists of multiple messages, these are always sent in one go, with no reference to TransmissionInterval. Furthermore, TransmissionInterval does not restrict the intervals between the messages of different elements: for instance, when you move two encoders or faders simultaneously, their respective MIDI messages are sent virtually simultaneously too.

Notes:

- The English version of the BCF/BCR manual (version 1.1, section 4.5) claims that TransmissionInterval ‘only has an effect on MIDI Data packs such as SysEx dumps and not on controlling of MIDI commands (they are carried out in real time anyway).’ The rather shaky English here seems caused by clumsy translation from the original German version. However, the German version basically makes the same claims, and these are almost totally wrong. In reality TransmissionInterval works as follows:
  - TransmissionInterval does not affect the timing of SysEx dumps (e.g. of presets).
  - TransmissionInterval does affect the timing of any element’s MIDI output. It doesn’t matter whether this output has been defined via a standard output statement or via custom output statements, nor does it matter of what type this output is: MIDI channel messages, SysEx messages, or other.

- Behringer’s most recent B-Control manual, i.e. version 1.1 (available from the Behringer website), explains that TransmissionInterval (‘MIDI Data Interval’) can be changed manually on the BCF/BCR via push encoder 8 in Global Setup mode (which is accessed via EDIT + STORE).
  Beware: the manual that came with your BCF/BCR may well be version 1.0: this version does not mention TransmissionInterval.

- The B-Control-Tokenreferenz web page suggests that TransmissionInterval only exists as of firmware version 1.10, but this isn’t so. In any case it already existed in 1.07; in line with this, all Behringer’s published syx preset files contain .txinterval statements (this contrasts with the situation for .deadtime).
12.7 Dead time

BCL syntax: `.deadtime DeadTime

DeadTime \in \{0 .. 1000\}

`DeadTime` indicates the time in milliseconds during which an encoder or fader (including the BCF’s Foot Controller) remains impervious to MIDI messages received by the BC after being physically manipulated by the user. An encoder or fader ignores any incoming MIDI messages that arrive within the `DeadTime` window after you last moved the encoder or fader. These interfering MIDI messages may be data from a running sequencer; they could also be the very messages triggered by your own manipulation of the encoder or fader if there is a feedback loop, which would be particularly uncomfortable if the feedback loop has a relatively long delay.

Perhaps most importantly, a high value for `DeadTime` can prevent a fader’s motor from being triggered by an interfering incoming MIDI message and thereby jerking the fader away from under your fingers while you are moving it manually. Preventing this should be good for both your nerves and the fader’s motor!

Note that buttons (including the Foot Switch(es)) do not adhere to `DeadTime`: they always react to incoming MIDI messages.

The BC only sends `DeadTime` in multiples of 10.

You can send any whole number in the range from 0 to 1000 to the BC, but the BC rounds down any value to a multiple of 10, i.e. to `DeadTime \mod 10`. (Note: the upper limit of 1000 is strictly enforced: even 1001 causes BCL error 11; in other words, the range check takes place before the rounding down.)

`DeadTime` is an obscure setting: it isn’t mentioned in Behringer’s B-Control manual (in whatever version), and the syx preset files available from Behringer’s website don’t contain `.deadtime` statements. (These facts suggest that `DeadTime` didn’t yet exist in the earliest firmware versions, but in any case it was already present in version 1.07.)

Note: although the B-Control manual doesn’t mention `DeadTime` in its global setup section, `DeadTime` can be set manually on the BCF/BCR via push encoder 7 in Global Setup mode (which is accessed via EDIT + STORE).
12.8 Factory defaults

The table below contains the factory defaults for the global settings. For convenience, it also indicates the push encoders via which you can edit the settings manually on the BCF/BCR in Global Setup mode (accessed via EDIT + STORE).

<table>
<thead>
<tr>
<th>Setting</th>
<th>Factory default</th>
<th>Push encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BCF2000</td>
<td>BCR2000</td>
</tr>
<tr>
<td>MidiMode</td>
<td>U-3</td>
<td>1</td>
</tr>
<tr>
<td>StartupPreset</td>
<td>last</td>
<td>4</td>
</tr>
<tr>
<td>FootSwitch</td>
<td>auto</td>
<td>3</td>
</tr>
<tr>
<td>ReceiveChannel</td>
<td>off</td>
<td>2</td>
</tr>
<tr>
<td>DeviceID</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>TransmissionInterval</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>DeadTime</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

As this table shows, the BCF2000’s factory defaults for TransmissionInterval and DeadTime are higher than those of the BCR2000. The philosophy behind this is probably that the BCF2000’s motorized faders are more vulnerable to conflicts between values entered via physical manipulation and values sent to the device via MIDI triggering the motors.

Note: most of the factory defaults shown in the above table also occur in the files bcf_FACTORY_PRESETS.syx and bcr_FACTORY_PRESETS.syx, as available from Behringer’s website (they are included in Factory_Presets_BCF.zip and Factory_Presets_BCR.zip respectively).

Beware: Propellerhead’s Reason (in any case versions 3.0 and 4.0) sets TransmissionInterval to 2 and DeadTime to 400 for both the BCF and BCR, and does not restore the original values after it has run. Totally reprehensible!
13 Presets

A preset section in a BCL block is introduced by a preset selector statement:

BCL syntax: $preset

Nitpicker’s note: as specified above, a preset selector statement should have no arguments. However, in fact the BC is very messy concerning its checking of this: basically any sequence of arguments after $preset is wrongly accepted and correctly ignored, e.g. ‘$preset nonsense 1234567890 !@#$%^&*()_+’. However, if any unrecognized dot statement identifier occurs (e.g. ‘$preset .boo’), then the BC replies with error 1. On the other hand, if all dot statement identifiers are recognized (e.g. ‘$preset .init .easypar’), the BC reports no error, but of course the BC does not actually try to execute any dot statement!

A preset selector statement sets all preset settings to the following values:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PresetName</td>
<td>'</td>
</tr>
<tr>
<td>Snapshot</td>
<td>off</td>
</tr>
<tr>
<td>Request</td>
<td>off</td>
</tr>
<tr>
<td>EncoderGroups</td>
<td>4</td>
</tr>
<tr>
<td>FunctionKeys</td>
<td>on</td>
</tr>
<tr>
<td>Lock</td>
<td>off</td>
</tr>
<tr>
<td>LearnOutput</td>
<td>cleared</td>
</tr>
</tbody>
</table>

LearnOutput is a sequence of zero or more LEARN output statements. A LEARN output statement is a .tx statement followed by one or more bytes in decimal or hexadecimal format.

A preset section sent by a BC contains the following dot statements (in this order):

`; name PresetName
`; snapshot Snapshot
`; request Request
`; egroups EncoderGroups
`; fkeys FunctionKeys
`; lock Lock

LearnOutput
`.init`

Details on these dot statements follow on the next pages.
13.1 Name

BCL syntax: \texttt{.name 'PresetName'}

If this statement is sent to the BC, \textit{PresetName} can have any length from 0 to 24 characters. The BC itself always sends all 24 characters (typically with a lot of spaces at the end). \textit{PresetName} cannot be longer than 24 characters: the BC then returns error 12.

The allowed character range is \texttt{#32 .. #127}. If you send a character in the range \texttt{#0 .. #31} to the BC, the BC still replies with ‘no error’ (i.e. error 0), but does convert the character to a dot (\texttt{#46}), except \texttt{#0}, which cuts off the name at the point it occurs.

Character \texttt{#39}, the apostrophe (‘), functions as a meta-character in the \texttt{.name} statement, so unsurprisingly the way the BC scans the \texttt{.name} statement causes several peculiarities concerning apostrophes:

- If you forget the terminating apostrophe, the final character of \textit{PresetName} is interpreted as the terminator, causing the BC to remove it from the actual name. So for instance if you send

  \texttt{.name 'sleepy}

  to the BC, a subsequent preset dump from the BC returns this as

  \texttt{.name 'sleep                   '}

- The inclusion of one or more apostrophes \textit{within \textit{PresetName}} often causes the BC to return BCL error 14 (‘Invalid number of arguments’). The BC has then wrongly interpreted an apostrophe as \textit{PresetName}’s terminator, so that the rest of \textit{PresetName} has been interpreted as an unexpected extra parameter.

  But even when the BC returns \textit{no error}, the resulting \textit{PresetName} in the BC may differ from the one that was sent to the BC, and in many cases this altered \textit{PresetName} will then trigger error 14 after all when you retrieve it from the BC and send it back to the BC!

  So to avoid problems, it is advisable not to include \textit{any} apostrophes in \textit{PresetName}. 


13.2 Snapshot

BCL syntax: `.snapshot Snapshot
Snapshot ∈ {off, on}`

If `Snapshot` is `on`, the BC automatically outputs a ‘snapshot’ upon selection of a preset. That is: the BC sends any defined standard and/or custom output of the elements in the active memory preset, excluding elements whose `Value` setting is `off`. See §23.4 for further discussion.

Note that even the `Value` settings sent in this ‘initial’ snapshot are indeed the *latest* values: *not* (generally speaking) the *default* values, as defined in the stored state of the memory preset. This is because the BC maintains the *latest* values of each of its 32 memory presets even *across* changes from one preset to another, as can be demonstrated easily:

1. Start the BC, and select preset 1 (if it isn’t already selected).
2. Change an encoder’s default value (let’s call this value `D`) by turning its knob, and memorize the new value (we’ll call this `E`).
3. Select preset 2 (via ‘PRESET •’). The encoder now indicates its value for preset 2 (which may of course be totally different).
4. Reselect preset 1 (via ‘• PRESET’). The encoder gets restored to the *latest* value for preset 1, i.e. `E`, not `D`. If `Snapshot` is `on`, this reselection of preset 1 also causes the BC to send `E` to the computer, not `D`.

Irrespective of the value of `Snapshot`, you can always trigger a snapshot manually by pressing EDIT + ‘• PRESET’, cf. §23.4.
13.3 Request

BCL syntax: \texttt{.request Request}\nRequest \(\in\) \{\texttt{off, on}\}

If \textit{Request} is \texttt{on}, the MIDI bytes defined by \textit{LearnOutput} (cf. §13.7) are sent upon selection of the preset. If \textit{Snapshot} is \texttt{on} too, the \textit{LearnOutput} MIDI bytes are sent \textit{before} the snapshot.

Irrespective of the value of \textit{Request}, you can always make the BC output the MIDI bytes defined by \textit{LearnOutput} manually by pressing \texttt{EDIT + LEARN}, cf. §23.2.
13.4 Encoder groups

BCL syntax: `.egroups EncoderGroups
EncoderGroups ∈ {1, 2, 3, 4}

There are 8 push encoders, each of which can operate as 4 separate (virtual) buttons and 4 separate (virtual) encoders. Group 1 contains buttons/encoders 1-8, group 2 9-16, group 3 17-24 and group 4 25-32.

By default, all 4 push encoder groups are available, and group selection occurs by means of the 4 buttons in the ‘ENCODER GROUPS’ block. However, if fewer than 4 push encoder groups are needed, some or all of the ‘ENCODER GROUPS’ buttons can be used as independent buttons.

`EncoderGroups` indicates the last push encoder group that is actually available:

<table>
<thead>
<tr>
<th><code>EncoderGroups</code></th>
<th>Available buttons/encoders via push encoders</th>
<th>Functions of buttons in ENCODER GROUPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top left</td>
<td>Top right</td>
</tr>
<tr>
<td>1</td>
<td>1 .. 8</td>
<td>Button 57</td>
</tr>
<tr>
<td>2</td>
<td>1 .. 16</td>
<td>Group 1</td>
</tr>
<tr>
<td>3</td>
<td>1 .. 24</td>
<td>Group 1</td>
</tr>
<tr>
<td>4</td>
<td>1 .. 32</td>
<td>Group 1</td>
</tr>
</tbody>
</table>

As the above table shows, if only one push encoder group is available, no group swapping is needed and all ‘ENCODER GROUPS’ buttons are available as independent buttons.
13.5 Function keys

BCL syntax: `.fkeys FunctionKeys
FunctionKeys ∈ {off, on}

Together with the Lock setting, FunctionKeys determines the function of the ‘function keys’, i.e. the BC buttons labeled STORE, LEARN, EDIT and EXIT:

If FunctionKeys is on and Lock is off, these buttons indeed perform the STORE, LEARN, EDIT and EXIT functions, and EDIT + STORE brings up Global Setup mode, etc.

If FunctionKeys is off or Lock is on, these buttons function as buttons 53-56, to which you can assign MIDI output messages just as you can for the other buttons on the BC.

Note that if Lock is on, the value of FunctionKeys is irrelevant and the function keys always function as buttons 53-56.

Note: the LEDs of the LEARN, EDIT and EXIT buttons remain lit indefinitely after a transition from .fkeys off to .fkeys on if they happen to be lit at the moment of transition: even switching to a different preset then doesn’t clear these LEDs! This problem is caused by a bug in the BC’s firmware (in any case vs. 1.10).
13.6 Lock

BCL syntax: `.lock  Lock
Lock ∈ \{off, on\}

This determines the function of the PRESET buttons (\* and \*):

If `Lock` is `off`, these buttons select the previous and next preset, respectively.

If `Lock` is `on`, these buttons function as buttons 63 and 64, to which you can assign MIDI output messages just as you can for the other buttons on the BC.

Moreover, if `Lock` is on, the value of `FunctionKeys` is irrelevant and the ‘function keys’ function as buttons 53-56.
13.7 LEARN output

LEARN output statement:
BCL syntax: \texttt{.tx \{b_1 \ldots b_N\}}
Each argument \(b_i\) must constitute a byte, defined decimally (0 .. 255) or hexadecimally ($00 .. $FF).
The BC itself always outputs a LEARN output statement using \textit{hexadecimal} definitions. So e.g. if you send the statement \texttt{.tx 240 247} to the BC, a preset dump from the BC will return this as \texttt{.tx $F0 $F7}.'

A preset’s \textit{LearnOutput} is a sequence of zero or more LEARN output statements. The BC outputs the sequence of bytes specified in these LEARN output statements as MIDI data in two situations:
1. Upon selection of the preset, provided that \textit{Request} is \texttt{on}.
2. When the user executes a manual Data Request by pressing \texttt{EDIT + LEARN}. See §23.2.

At least one byte must be specified after \texttt{.tx} (in other words: \(b_1\) is obligatory), otherwise the BC replies with error 3. This means that you can only undefine a previous \textit{LearnOutput} by sending a \texttt{$\text{preset}$} section without any LEARN output statements. In accordance with this constraint, the BC itself never sends bare LEARN output statements in preset dumps.

For each preset, the BC stores \textit{LearnOutput} (the sequence of all LEARN output statements) in an internal data buffer of 127 bytes. When the BC receives a LEARN output statement that overflows this internal buffer, it replies with error 15.

Each \texttt{.tx} identifier takes up \textit{two} bytes in the internal data buffer. Each byte \(b_i\) takes up \textit{one} byte, except if it is \texttt{SFE} (Active Sensing), in which case it takes up \textit{two} bytes. So if you define only one LEARN output statement, that statement can at most define 125 bytes; if you define two LEARN output statements, these statements may only define 123 bytes (divided in whatever way between the two statements).

Notes:

- In preset dumps \textit{from} the BC (e.g. via \texttt{EDIT + ‘PRESET +’}) the BCL message containing a LEARN output statement specifying \textit{exactly} 125 MIDI bytes is incorrect. That is, the format of such a BCL MIDI message itself is invalid, and Windows’ low-level MIDI input routine balks at it. It seems that the BC uses a buffer of 512 bytes for sending BCL messages, which turns out to be \textit{just} too short for a \texttt{.tx} statement specifying 125 bytes (124 bytes is no problem). Note that the BC does send the 125 bytes \textit{themselves} (i.e. as MIDI data) correctly upon selection of the preset and when you press \texttt{EDIT + LEARN}.

- The B-Control-Tokenreferenz web page claims that the maximum number of definable MIDI bytes is 123. (‘Die maximale Länge der übertragenen Daten scheint auf 123 Bytes begrenzt zu sein,...’) Now if we are talking about \textit{one} SysEx message, the maximum number of MIDI \textit{data} bytes is indeed 123, since each SysEx message requires two obligatory MIDI status bytes: one initial \texttt{SF0} and one final \texttt{SF7}. However, the bytes specified after \texttt{.tx} may constitute more than \textit{one} message, and these messages don’t have to be \textit{SysEx} messages; so the B-Control-Tokenreferenz web page’s claim is incorrect. (Moreover, as stated above, there can be more than one \texttt{.tx} statement!)

\textbf{Spurious output from a .tx statement containing a non-final System Exclusive message}
There are no ‘semantic’ restrictions to the MIDI bytes specified in a single \texttt{.tx} statement: a single \texttt{.tx} statement may contain multiple MIDI messages: channel messages, system exclusive messages, or whatever.

38
So for economy’s sake it would seem best to put all MIDI messages in a single LEARN output statement. However, there is a big caveat: the BC may output one to three additional, spurious MIDI bytes when a `.tx` statement containing a SysEx message in non-final position is output via the BC’s USB connection. (The BC’s standard MIDI outputs (OUT A and OUT B/THRU) are not affected.)

For instance, the statement

```
.tx $F0 $7D $01 $02 $F7
 $F0 $7D $11 $12 $F7
```

leads to the following output:

F0 7D 01 02 F7 (first SysEx message)
02 F7 (spurious repetition of the end of the first SysEx message)
F0 7D 11 12 F7 (second SysEx message)

To avoid this type of spurious output, you must put every System Exclusive message at the end of a separate `.tx` statement. So the above example should be rewritten as:

```
.tx $F0 $7D $11 $12 $F7
.tx $F0 $7D $21 $22 $F7
```

In fact, for clarity you may consider putting all MIDI messages (so even the non-SysEx ones) in separate `.tx` statements. On the other hand, this takes up a bit more preset memory than multiple messages in one `.tx` statement.

The exact content of the spurious output after a non-final SysEx message depends on the length of the SysEx message, and on Windows also on the running status byte:

The table below specifies the additional, spurious output after non-final SysEx messages of various lengths, as seen on OS X 10.9.

<table>
<thead>
<tr>
<th>Non-final SysEx message</th>
<th>Spurious output (OS X 10.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0 F7</td>
<td>F0 F7</td>
</tr>
<tr>
<td>F0 7D F7</td>
<td>F0 7D F7</td>
</tr>
<tr>
<td>F0 7D 01 F7</td>
<td>F7</td>
</tr>
<tr>
<td>F0 7D 01 02 F7</td>
<td>02 F7</td>
</tr>
<tr>
<td>F0 7D 01 02 03 F7</td>
<td>02 03 F7</td>
</tr>
<tr>
<td>F0 7D 01 02 03 04 F7</td>
<td>F7</td>
</tr>
<tr>
<td>F0 7D 01 02 03 04 05 F7</td>
<td>05 F7</td>
</tr>
</tbody>
</table>

We here see the repetition of the one to three last bytes of the non-final SysEx message, where the number of spurious bytes follows a cyclical pattern: 2, 3, 1, 2, 3, 1, 2... So we may surmise that the BC’s USB controller applies inappropriate MIDI running status logic: it seems to wrongly think that there is a MIDI channel message without status byte and try to ‘correct’ this by completing that message.

It depends on the receiving MIDI device how the spurious output from the BC is processed. Probably the output seen on OS X 10.9 is what the BC’s USB actually outputs, but for instance Windows 7’s standard MIDI driver applies further processing to what it receives from the BC:

When the running status byte being maintained by Windows’ MIDI driver is *undefined*, this driver removes any spurious output from the BC starting with a *data* byte ($00-$7F), in effect passing only spurious output starting with a status byte ($F0 or $F7):
This reduction in spurious output may seem positive. However, when the Windows driver’s running status byte has been defined, the driver actually makes things worse, namely by converting any spurious output starting with a data byte to a complete channel message starting with that running status byte.
For instance, if we have set the driver’s running status byte to $B0 by sending a Control Change message on channel 1 before triggering the `.tx` statement, we get:

<table>
<thead>
<tr>
<th>Non-final SysEx message</th>
<th>Spurious output (Windows 7; running status byte = $B0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0 F7</td>
<td>F0 F7</td>
</tr>
<tr>
<td>F0 7D F7</td>
<td>F0 7D F7</td>
</tr>
<tr>
<td>F0 7D 01 F7</td>
<td>F7</td>
</tr>
<tr>
<td>F0 7D 01 02 F7</td>
<td>none</td>
</tr>
<tr>
<td>F0 7D 01 02 03 F7</td>
<td>none</td>
</tr>
<tr>
<td>F0 7D 01 02 03 04 F7</td>
<td>F7</td>
</tr>
<tr>
<td>F0 7D 01 02 03 04 05 F7</td>
<td>none</td>
</tr>
</tbody>
</table>

So for instance the Windows MIDI driver converts the BC’s spurious 02 F7 to B0 02 F7. Note that the resulting final ‘data’ byte (which is supposed to lie in the $00-$7F range) is actually a status byte ($F7) here, so MIDI applications unprepared for this out-of-range value may face data corruption or even crash. Also note that the BC’s spurious 02 03 F7 becomes B0 02 03: the spurious F7 gets swallowed here — small consolation...
13.8 Initialization of all elements

BCL syntax: `.init`

This statement clears all the temporary preset’s elements. That is: all buttons, encoders and faders are initialized.

In fact, the effect of a `.init` statement is identical to that of a sequence of empty element sections for all the temporary preset’s elements. So a simple BCL block like

```
$rev F1
$preset
  .init
$end
```

has the same result as

```
$rev F1
$encoder 1
$encoder 2
etc.
$encoder 32
$button 1
$button 2
etc.
$button 64
$fader 1
$fader 2
etc.
$fader 9
$end
```

It is not obligatory to include `.init` in a preset section sent to the BC: if you leave `.init` out, any temporary preset elements not included in the rest of the BCL chain simply remain at their current settings.

However, the BC itself always includes `.init` in a preset dump, even if you have previously omitted `.init` from a preset section you have sent to it! This is done for economy: a preset dump from the BC always contains a complete definition of the temporary preset — the `.init` statement allows the BC to achieve this completeness while leaving out any ‘empty’ elements.

Finally, note that `.init` changes the BC’s global variables `Button`, `Encoder` and `Fader`. See the section on invalid element selector statements for discussion.
14 Control elements

The settings described in this section of this document are common to all BC elements (buttons, encoders and faders).
14.1 Standard output

BCL syntax: `.easypar StandardOutput`

*StandardOutput* is a sequence of arguments. The first of these is *Type*, which determines the number and nature of the rest of the arguments.

For any control element (button, encoder or fader) you can select one ‘standard’ output definition. This usually causes the BC to output a single MIDI message whenever you physically manipulate (i.e. press, turn or slide) the control element. Which types of standard output definition are available, depends on the type of control element. For more complicated (‘custom’) output, the `.tx` statement must be used instead.

Every `.easypar` statement has several side-effects:
- *Default* is set to a value that depends on the actual `.easypar` statement.
- *CustomOutput* is cleared.

Further side-effects are specific to specific element types (buttons, encoders and faders) and `.easypar` statement types. These side-effects are described in the pertinent sections.

This is not the place for a complete discussion of the MIDI protocol. Many books and web documents exist on this subject; even Behringer’s BC Manual contains some useful information. However, the following table specifies the general MIDI message types and the `.easypar` statement *Type(s)* generating them. The Button and Encoder/Fader columns indicate whether a particular `.easypar` *Type* is available for buttons and encoders/faders respectively.

<table>
<thead>
<tr>
<th>Basic type</th>
<th>MIDI message</th>
<th>.easypar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note Off</td>
<td>$8c Note Velocity</td>
</tr>
<tr>
<td></td>
<td>Note On</td>
<td>$9c Note Velocity</td>
</tr>
<tr>
<td></td>
<td>Note Aftertouch</td>
<td>$Ac Note Aftertouch</td>
</tr>
<tr>
<td></td>
<td>Control Change</td>
<td>$Bc Controller Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program Change</td>
<td>$Cc Program</td>
</tr>
<tr>
<td></td>
<td>Channel Aftertouch</td>
<td>$Dc Aftertouch</td>
</tr>
<tr>
<td></td>
<td>Pitch Bend</td>
<td>$Ec ValueLSB ValueMSB</td>
</tr>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exclusive</td>
<td>$F0 ... $F7</td>
</tr>
<tr>
<td></td>
<td>Common</td>
<td>misc.</td>
</tr>
<tr>
<td></td>
<td>Real-time</td>
<td>misc.</td>
</tr>
</tbody>
</table>
Notes:

- Every MIDI message starts with a status byte (in the range $80-$FF) and is followed by zero or more data bytes (in the range $00-$7F), as dictated by the status byte. The only exception to this rule is the System Exclusive message, which has a variable number of data bytes and must therefore be terminated by the end-of-exclusive marker $F7.

- Every MIDI Channel Message specifies a channel $c$ from $0$ to $F$ as the lower nibble (four bits) of the status byte. Beware: ‘external’ descriptions of this MIDI channel usually add 1 to $c$, leading to a range of 1-16.

- If you study the precise definitions of the `.easypar` statement types (as specified in subsequent sections of this document), you will see that for `NOTE`, `AT`, `CC`, `PC` and `PB` there is a more or less one-to-one correspondence between the MIDI message type and the `.easypar` Type. (I say ‘more or less’ because of certain small differences: for instance, `PC` allows you to not only set a program number, but also MSB and LSB bank numbers.)

  By contrast, the `NRPN`, `GS/XG` and `MMC` types correspond with small, rather esoteric subsets of the MIDI message types to which they belong. (In fact, `NRPN` defines output that consists of more than one MIDI message, and in many cases `GS/XG` does so too.)

- The MIDI Note Off message has no corresponding `.easypar` Type. This is in line with the common practice of using a MIDI Note On message with `Velocity` 0 instead of a MIDI Note Off message: MIDI Note Off `Velocity` is seldom relevant, and the use of MIDI Note On allows for a shorter MIDI stream due to the use of the ‘Running Status’ protocol, which stipulates that a Channel Message’s status byte does not have to be included in the stream if it is identical to the previous message’s status byte.

- Both Note and Channel Aftertouch are represented by the `AT` Type. This is possible because the additional `Scope` argument distinguishes between Note and Channel Aftertouch.

- The `GS/XG` Type merely offers ‘user-friendly’ shortcuts for the ‘Main Control’ parameters defined in the Roland GS and Yamaha XG MIDI specifications. Depending on the actual GS/XG parameter chosen, the resulting MIDI output follows either the `CC` or the `NRPN` scheme. See the tables on the next page.
### 14.1.1 GS/XG Main Control parameters

<table>
<thead>
<tr>
<th>.easypar Parameter</th>
<th>Display</th>
<th>GS/XG Parameter</th>
<th>Type</th>
<th>NRPN_MSB</th>
<th>NRPN_LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutoff</td>
<td>CUTF</td>
<td>Filter Cutoff Frequency</td>
<td>NRPN</td>
<td>$01</td>
<td>$20</td>
</tr>
<tr>
<td>resonance</td>
<td>RESO</td>
<td>Filter Resonance</td>
<td>NRPN</td>
<td>$01</td>
<td>$21</td>
</tr>
<tr>
<td>v-rate</td>
<td>RATE</td>
<td>Vibrato Rate</td>
<td>NRPN</td>
<td>$01</td>
<td>$08</td>
</tr>
<tr>
<td>v-depth</td>
<td>DEPT</td>
<td>Vibrato Depth</td>
<td>NRPN</td>
<td>$01</td>
<td>$09</td>
</tr>
<tr>
<td>v-delay</td>
<td>DLY</td>
<td>Vibrato Delay</td>
<td>NRPN</td>
<td>$01</td>
<td>$0A</td>
</tr>
<tr>
<td>eg-attack</td>
<td>ATC</td>
<td>EG Attack Time</td>
<td>CC</td>
<td>$01</td>
<td>$63</td>
</tr>
<tr>
<td>eg-decay</td>
<td>DCY</td>
<td>EG Decay Time</td>
<td>CC</td>
<td>$01</td>
<td>$64</td>
</tr>
<tr>
<td>eg-release</td>
<td>RELS</td>
<td>EG Release Time</td>
<td>CC</td>
<td>$01</td>
<td>$66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.easypar Parameter</th>
<th>Display</th>
<th>GS/XG Parameter</th>
<th>Type</th>
<th>$01</th>
</tr>
</thead>
<tbody>
<tr>
<td>modulation</td>
<td>MODU</td>
<td>Modulation</td>
<td>CC</td>
<td>$0A</td>
</tr>
<tr>
<td>p-time</td>
<td>PORT</td>
<td>Portamento Time</td>
<td>CC</td>
<td>$05</td>
</tr>
<tr>
<td>volume</td>
<td>VOL</td>
<td>Volume</td>
<td>CC</td>
<td>$07</td>
</tr>
<tr>
<td>panorama</td>
<td>PAN</td>
<td>Pan</td>
<td>CC</td>
<td>$0A</td>
</tr>
<tr>
<td>rev-send</td>
<td>REVB</td>
<td>Reverb Send</td>
<td></td>
<td>$5B</td>
</tr>
<tr>
<td>crs-send</td>
<td>CRS</td>
<td>Chorus Send</td>
<td></td>
<td>$5D</td>
</tr>
<tr>
<td>dly-send</td>
<td>VARS</td>
<td>Delay/Variation Send</td>
<td></td>
<td>$5E</td>
</tr>
</tbody>
</table>
14.2 Show value

BCL syntax: `.showvalue ShowValue
ShowValue ∈ {off, on}`

If `ShowValue` is on, manipulation of the element (button, encoder, fader) causes the BC’s display to briefly show the element’s new value.

Note: the BC’s display contains 4 characters; if the value to be shown is 10000 or higher (up to the absolute maximum of 16383), the most-significant digit (1) is not displayed; so e.g. 10000 is displayed as ‘0000’.

Any `.easypar` statement sets the element’s `ShowValue` to off, so any `.showvalue` statement (or at least `.showvalue on`) should occur after any `.easypar` statement.
14.3 Default value

BCL syntax: `.default Default`

To the BC: `Default ∈ {off, 0 .. 16383}`
From the BC: `Default ∈ {0 .. 16383}`

The `Default` setting provides initialization for the element’s `Value`, as described in the next section. `Default` does not have to lie within any range defined by `.easypar` or `.minmax`’s `Value1` and `Value2`.

Note that `Default` can be changed in other ways as well:

1. Any element section selector statement ($button, $encoder or $fader) clears `Default`. So it’s as if each section starts with an implicit `.default off` statement.

2. Any `.easypar` statement overwrites any previously set `Default` with an implicit value that is deemed appropriate for that particular `.easypar` statement.
   
   So in the following BCL section the `.default` statement is pointless, since the `.easypar` statement immediately overwrites the value set by `.default`:

   ```
   $encoder 1
   .default 10 ; this sets Default to 10
   .easypar 1 1 0 127 absolute ; this sets Default to 0
   ```
   
   So any `.default` statement should occur after `.easypar`:

   ```
   $encoder 1
   .easypar 1 1 0 127 absolute ; this sets Default to 0
   .default 10 ; this sets Default to 10
   ```

3. When the user manually stores a preset via the STORE button, the target memory preset’s `Default` is set to the current `Value` setting of the active memory preset. (See the next section for more information on `Value`.) Note that this does not occur when the BC receives the BCL `$store` statement!

A special case is `.default off`. The B-Control-Tokenreferenz web page doesn’t mention this, but it can indeed be sent to the BC. The BC itself never sends `.default off` in preset dumps: instead, if no default has been defined, the BC simply sends no `.default` statement.

Since the section selector statement has already cleared `Default` (i.e. set it to `off`), `.default off` is only meaningful to clear a value for `Default` set via a `.default` or `.easypar` statement in the same section. For instance:

```
$encoder 1
.easypar 1 1 0 127 absolute ; this sets Default to 0
.default off
```
14.4 Current value

Value $\in \{\text{off}, 0 \ldots 16383\}$

Every element in a memory preset has a Value setting. Value constitutes the element’s current value, i.e. the element’s ‘core’ value, which is used in standard and custom MIDI output. Note that Value isn’t stored in the memory preset as such: it is only maintained from the moment the BC is powered on until it is powered off; after that, its value is lost.

A Value setting can be changed in the following ways:

1. When the BC gets powered on, Value is initialized to the element’s Default setting.

2. When the Default setting of an element in the temporary preset gets changed via any BCL statement as described in the previous section (i.e. $\$\text{button}, \$\text{encoder}, \$\text{fader}, \text{.easypar}$ or $.\text{default}$), the corresponding Value in the active memory preset is updated to this new Default setting.

   Note that Value is not reset to Default upon preset selection (via any of the methods described in §19.3)! The BC only maintains Value for all the elements of the 32 memory presets, not for the temporary preset. Upon preset selection, a memory preset’s settings are copied to the temporary preset and the selected memory preset becomes the active memory preset, but the Value settings of this memory preset don’t change! This feature allows you to use the 32 memory presets as one virtual ‘super’-preset: you can switch between memory presets without ever destroying any memory preset’s Value settings.

3. When the temporary preset is copied to a memory preset (via the $\$\text{store}$ statement or the manual ‘store preset’ function involving the STORE button), the target memory preset’s Value settings are set to those of the old active memory preset, and then the target becomes the new active memory preset.

   So the target memory preset’s Value settings are not set to the temporary preset’s Default settings! In fact, as already mentioned in the previous section, in the case of the manual ‘store preset’ function (but not the $\$\text{store}$ statement), the target memory’s Default settings are set to the Value settings!

4. When the user physically manipulates a button, encoder or fader, the corresponding Value in the active memory preset is updated accordingly.

5. When the BC receives a MIDI message corresponding with the standard output (i.e. $.\text{easypar}$) definition of an element in the active memory preset, that element’s Value is updated to the received value. Note that this does not work for elements for which only custom output is defined.

Note that a BCF fader only jumps to a new value automatically physically if its .motor setting is on. This is particularly relevant if you want the fader to automatically move to its Default setting upon first selection of the preset.

If an element’s Value is off, the Snapshot Send function (see §23.4) doesn’t send any standard or custom output for that element. Value can only become off as a result of Default being off, either directly (methods 1 and 2 above) or indirectly (method 3 above).
14.5 Physical mapping

BCL syntax: .minmax Value1 Value2
Value1, Value2 ∈ \{0 .. 16383\}

This statement is primarily intended for setting the element’s mapping from physical position to Value for custom output messages (as defined via .tx statements). However, it also affects any standard output defined via a preceding .easypar statement, overriding any settings for Value1 and Value2 occurring in that .easypar statement.

The way in which Value1 and Value2 are used, depends on the element type:

Button:
Value1 indicates Value’s first state, Value2 indicates Value’s second state. When a button changes from the first to the second state and vice versa depends on the Mode setting. For instance, if Mode is updown, Value1 is associated with the ‘up’ position, and Value2 with the ‘down’ position.

Encoder:
Value1 indicates the ultimate value that Value can take if you turn the knob in the counterclockwise direction, Value2 the ultimate value in the clockwise direction.

Fader:
Value1 indicates the value Value takes if you move the fader to its bottom position, Value2 the value associated with the top position.

In principle Value1 does not have to be lower than Value2: any combination is legal, since Value1 and Value2 merely determine the mapping from physical position to Value. (As Royce Craven has pointed out to me, for a pair of inversely related parameters it can be handy to reverse the direction of one of the elements.)

However, if you use the .mode incval statement for a button, Value1 does have to be lower than Value2. See the discussion in §15.10.

Also note that Value1 and Value2 don’t prevent Value from being set to any value by ‘external’ means, i.e. initialization of the Default setting or (provided standard output has been defined for the element) a corresponding incoming MIDI channel message.
14.6 Custom output

The definition of the custom MIDI output of elements (buttons, encoders en faders) basically follows the same format as the LEARN output for a preset: for each element you can define a sequence of custom output statements, each specifying a sequence of MIDI output bytes. All the syntactic constraints specified for LEARN output in §13.7 apply to custom output as well. However, there is one difference: whereas a LEARN output statement can only contain byte definitions, a custom output statement can also contain ‘special’ identifiers. There are five types, which are described on the next pages. (See Royce Craven’s BCSecrets.pdf document for a less formalistic approach, more directly aimed at concrete problem-solving.)

Standard vs. custom output

Nearly(?) all standard output (.easypar) definitions could be programmed as custom output (.tx): the custom output format is much wider. On the other hand, the BC interprets incoming parameter feedback via MIDI only according to the element’s standard output definition; the BC does not use any custom output definitions to synchronize the element’s current Value. In other words, custom output definitions are ‘deaf’. So the best strategy is to only use a custom output definition if you can’t achieve the same result via a standard output definition.
14.6.1 Data Specifier

The BC outputs the bytes specified in the custom output statement(s) as MIDI data whenever you change the physical position of a control element. You order the BC to include the actual position or change of position by including one or more data specifiers in the custom output statement(s). This causes the BC to output Data, or rather (as we will see) portions of Data.

By default, Data is simply Value, i.e. the actual value (position) of the control element. However, after a Change Definition in a custom output statement (as described in §14.6.2), Data indicates the relative change in the control element’s value.

Data has 14 bits. The BC can be ordered to transmit various portions of Data as a MIDI byte:

<table>
<thead>
<tr>
<th>DataSpecifier</th>
<th>Bits</th>
<th>Transmitted MIDI byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>0 .. 6</td>
<td>Data and $007F</td>
</tr>
<tr>
<td>va10.6</td>
<td>0 .. 6</td>
<td>Data and $007F</td>
</tr>
<tr>
<td>va10</td>
<td>0</td>
<td>Data and $0001</td>
</tr>
<tr>
<td>va10.3</td>
<td>0 .. 3</td>
<td>Data and $000F</td>
</tr>
<tr>
<td>va14.7</td>
<td>4 .. 7</td>
<td>(Data and $00F0) shr 4</td>
</tr>
<tr>
<td>va18.11</td>
<td>8 .. 11</td>
<td>(Data and $0F00) shr 8</td>
</tr>
<tr>
<td>va12.13</td>
<td>12 .. 13</td>
<td>(Data and $3000) shr 12</td>
</tr>
<tr>
<td>va7.13</td>
<td>7 .. 13</td>
<td>(Data and $3F80) shr 7</td>
</tr>
<tr>
<td>va1.7</td>
<td>1 .. 7</td>
<td>(Data and $00FE) shr 1</td>
</tr>
</tbody>
</table>

All these specifiers use ‘LSB 0 bit numbering’, i.e. bit 0 is the least significant bit, bit 13 the most significant bit. (See [https://en.wikipedia.org/wiki/Bit_numbering](https://en.wikipedia.org/wiki/Bit_numbering) for further discussion.)

Apart from val, these portions are clearly intended to be transmitted in combinations of subsequent MIDI bytes that cover the whole Data, i.e. all its 14 bits. In fact, there are only three such combinations:

1. **va17.13 & va10.6**
   Probably the most widely usable combination. This splits Data into two groups of seven bits. For instance, to make an encoder output its 14-bit value in a pair of Control Change messages for the Modulation controller of MIDI channel 1, you would use (in this order):
   ```
   .tx $B0 1 va17.13 ; controller 1 = Modulation MSB
   .tx $B0 33 va10.6 ; controller 33 = Modulation LSB
   ```

2. **va12.13 & va18.11 & va14.7 & va10.3**
   Amounts to a segmentation into four groups of four bits (‘nibbles’), where bits 14 and 15 are zero.

3. **va12.13 & va18.11 & va11.7 & va10**
   A rather curious segmentation. (I don’t know if any MIDI device requires this; alternatively, va11.7 and va10 might be intended for together capturing an 8-bit Data.)
Note that `val` and `val0.6` are semantically identical: they refer to bits 0 .. 6, i.e. the 7 lowest bits of `Data`. You can use the one or the other in any context. However, the BC itself never sends `val0.6`, so if you send `val0.6` to the BC, it returns it as `val`. For clarity in your own definitions you would normally use `val` if `Data`’s range (as defined via `.minmax`) lies in 0 .. 127, and `val0.6` otherwise, i.e. in contexts where you also use other portions of `Data`. It’s just that a combination of `val7.13` & `val` looks a bit confusing.
14.6.2 Change Definition

These definitions do not generate any MIDI output themselves, but turn the output for any further Data Specifiers (val etc.) into measurements of relative change.

Three cases:

1. **rel2s**
   
   $Data = \text{Change}$
   
   In this case $Data$ is simply $\text{Change}$, formatted as a standard two’s complement value. Basically **rel2s** is the same as **easypar**’s Mode argument **relative-1**. 
   
   Note that **rel2s** is equivalent to **reloffs 0**.

2. **reloffs** $Offset$

   $Offset \in \{0 .. 16383\}$
   
   $Data = Offset + \text{Change}$
   
   Basically **reloffs** is a generalized form of **easypar**’s Mode argument **relative-2**.

3. **relsign** $Offset$

   $Offset \in \{0 .. 16383\}$
   
   If $\text{Change} > 0$, then $Data = \text{Change}$
   
   If $\text{Change} < 0$, then $Data = Offset - \text{Change}$
   
   Note that if $\text{Change}$ is negative here, $Data$ is greater than $Offset$ (unless wrapping occurs, of course, e.g. if $Offset = 16383$). So e.g. if $Offset$ is $40$ and $\text{Change}$ is $-1$, then $Data$ is $41$. (So if you define $Offset$ as $00$, the bizarre result is that there is no difference in output between turning the knob to the left and turning it to the right!)
   
   Basically **relsign** is a generalized form of **easypar**’s Mode argument **relative-3**.

$Offset$ can be sent to the BC in decimal or hexadecimal notation. The BC sends $Offset$ in four-digit hexadecimal notation (Shhhh), so e.g. **reloffs 2** is returned as **reloffs $0002$**.

The following example clarifies the various options:

```
$encoder 1
  .showvalue on
  .mode 1dot
  .resolution 96
  .default 0
  .minmax 0 127
  .tx $F0 $7D  val  rel2s val  reloffs $40 val  relsign $40 val  $F7
```

(The extra spaces in the last line are simply meant to make the grouping a bit clearer. By the way, $7D$ indicates the ‘non-commercial’ manufacturer; thus, any actual MIDI hardware device from a ‘real’ manufacturer that happens to receive this SysEx test message should simply ignore it.)

If you now turn push encoder 1 a bit in the clockwise direction, then back, the BC produces a sequence of MIDI System Exclusive messages like this:
The first `val` in the `.tx` statement has produced the third byte of each SysEx message: this is simply the actual knob value (as simultaneously shown by the BC in its display). The fourth, fifth and sixth bytes are the relative values produced by `rel2s`, `reloffs $40` and `relsign $40` respectively; in each of these cases the meaning of the output value (i.e. `Change`) is ‘+1’ for the first three messages and ‘–1’ for the last three messages — obviously the receiving MIDI device must be able to interpret these data bytes correctly.
14.6.3 Checksum Definition

ChecksumDefinition = Method  StartByteIndex
Method ∈ {cks-1, cks-2, cks-3}
StartByteIndex ∈ {0 .. 127}

At the position of ChecksumDefinition, the MIDI output from the BC generates one MIDI data byte (i.e. in the range 0 .. 127) that represents the checksum of a sub-sequence of MIDI output bytes.

The first MIDI byte taken into account in the calculation of this checksum is the byte whose position in the current custom output statement is indicated by StartByteIndex; the first MIDI byte defined after the .tx identifier occupies position 0. The last MIDI byte taken into account is the byte immediately before the Checksum Definition itself.

cks-1: The sum of all the MIDI output bytes and the checksum itself is a value of which bits 0 to 6 are zero. (Or, to say the same thing differently, the checksum is bits 0 to 6 of the negative sum of the MIDI output bytes.)

The Roland company uses this method (in particular for RQ1 and DT1 messages). According to Royce Craven, it has also been used by E-MU Systems and Yamaha.

cks-2: The checksum is bits 0 to 6 of the sum of all the MIDI output bytes.

The Waldorf company uses this method (e.g. for the MicroWave and Pulse), and (according to Royce Craven) so does Kurzweil.

cks-3: First, the byte at StartByteIndex and its successor are xor’ed; then, each following MIDI output byte is xor’ed with the result of the previous xor.

As discovered by Royce Craven, this method is recommended by the MMA (MIDI Manufacturers Association). It is used by the Fractal Audio company.

If you know any other manufacturer implementing any of these methods, let me know so that I can add this information.

Like the MIDI bytes of a .tx statement, StartByteIndex can be sent to the BC in decimal or hexadecimal notation. The BC sends StartByteIndex in decimal notation. (So StartByteIndex is the only decimal argument in a .tx statement sent by the BC: any MIDI bytes and arguments to Change Definitions are sent as hexadecimals!)

As Royce Craven discovered, the BC erroneously doesn’t send back the Method identifier; obviously this completely ruins the intended definition.4

Note that the BC doesn’t balk at any StartByteIndex that refers to a forward byte, or indeed a position beyond the end of the actual message. (In fact, since any custom output statement can only define 125 bytes at best, a value for StartByteIndex in the range of 125 .. 127 always refers to a non-existent position.) I haven’t checked the actual MIDI output in any of these anomalous cases.

A custom output statement can contain multiple checksum definitions, so in theory you can include multiple SysEx messages in a single statement. However, this is unadvisable for two reasons. In the first place, as discussed in §13.7, a SysEx message in a non-final position in a .tx statement may cause up to three spurious output bytes. Secondly, StartByteIndex counts from the start of the custom output

---

4 Actually the BC does return the space character that is always sent before any identifier occurring after another identifier. So for instance '.tx $F0 $7D $01 cks-1 2 $F7' comes back as '.tx $F0 $7D $01 2 $F7', i.e. with two spaces between $01 and 2, the first space being "in front of" the phantom Method identifier, the second one in front of 2.
Consider the following statement:

```
.tx $F0 $7D $01 cks-2 2 $F7 $F0 $7D $01 cks-2 2 $F7
```

This statement contains two identical definitions for SysEx messages, but the BC’s MIDI output resulting from this statement includes different checksum bytes, because the second StartByteIndex value of 2 refers to the byte with the index of 2 in the whole statement, which is the third byte of the first SysEx message. Consequently, in calculating the second checksum, the BC sums the bytes with indexes 2 to 7 ($01 + $01 + $F7 + $F0 + $7D + $01 = $267), instead of using only the byte at index 7. So the output from the BC is:

F0 7D 01 01 F7 (first SysEx message)
01 F7 (spurious repetition of the end of the first SysEx message)
F0 7D 01 67 F7 (second SysEx message: unintended checksum)

To correct the checksum of the second SysEx message, you would have to change the second StartByteIndex to 7:

```
.tx $F0 $7D $01 cks-2 2 $F7 $F0 $7D $01 cks-2 7 $F7
```

This would produce:

F0 7D 01 01 F7
01 F7
F0 7D 01 01 F7

The second checksum is as intended here.

However, even here the BC generates spurious output between the two SysEx messages. So to avoid any spurious output and StartByteIndex confusion, you should define each SysEx message via a separate custom output statement. For instance:

```
.tx $F0 $7D $01 cks-2 2 $F7
.tx $F0 $7D $01 cks-2 2 $F7
```

This yields the intended MIDI output from the BC:

F0 7D 01 01 F7
F0 7D 01 01 F7
14.6.4 Direction Specifier

\[ \text{DirectionSpecifier} \in \{\text{ifp}, \text{ifn}\} \]

Any definitions (MIDI bytes or special identifiers) after \textit{ifp} only lead to MIDI output when \textit{Change} is positive. Any definitions after \textit{ifn} only leads to MIDI output when \textit{Change} is negative.

It is possible to have any number of Direction identifiers in the same custom output statement. Normally you have only one \textit{ifp} and one \textit{ifn}, but multiple occurrences are accepted too. E.g. \texttt{.tx ifp $C0 \$01 ifn $C0 \$02 ifp $C0 \$03 ifn $C0 \$04} leads to \texttt{C0 01 C0 03} for positive changes and to \texttt{C0 02 C0 04} for negative changes.

Note that the problem concerning System Exclusive messages mentioned under ‘Checksum Definition’ applies to direction specifiers as well: the rule seems to be that a System Exclusive message in an \textit{ifp} or \textit{ifn} clause produces spurious output if more bytes are output afterwards. Note the precise definition of this rule: for instance, if a System Exclusive message occurs in an \textit{ifp} clause, any output byte definitions in subsequent \textit{ifn} clauses are allowed (since these don’t produce any output if the direction is positive), but any output byte definition in the same or any subsequent \textit{ifp} clause leads to spurious output.

For instance:

\texttt{
$\text{encoder 1}$  
  \texttt{.showvalue on}  
  \texttt{.mode 1dot}  
  \texttt{.resolution 100}  
  \texttt{.minmax 0 100}  
  \texttt{.tx $C0 \$01 ifp $C1 \$01 $F0 $7D $F7 $C2 \$01}}

In case of a positive change, the MIDI output is \texttt{C0 01 C1 01 F0 7D F7 F0 7D F7 C2 01}: i.e. the SysEx message is repeated!
14.6.5 Repeat

Repeat = ntimes

The MIDI output resulting from any definitions (MIDI bytes or special identifiers) after ntimes is repeated a number of times, based on the amount of change. This makes it possible to send the change in the control element’s value as a series of identical messages, instead of the ‘normal’ single message specifying the actual (‘absolute’) new value (Value) or the relative change (Change). Typically you would only use this if the receiving device cannot handle absolute values or amounts of change.

It is syntactically valid to include multiple instances of ntimes on the same line, but only the last instance works: any previous ones are simply ignored.

The following example should clarify how ntimes actually works:

$encoder 1
   .showvalue on
   .mode 1dot
   .resolution 200
   .minmax 0 127
   .tx $B0 $00 val rel2s $B0 $01 val ntimes $B0 $02 $00

This yields output like the following sequence:

B0 00 01 (new absolute value)
B0 01 01 (change of +1 via rel2s)
B0 02 00 (the ntimes clause "fires" once, in line with the change)

B0 00 03 (new absolute value)
B0 01 02 (change of +2 via rel2s)
B0 02 00 (ntimes fires once)
B0 02 00 (ntimes fires once more, to match the change of +2)

B0 00 06 (new absolute value)
B0 01 03 (change of +3 via rel2s)
B0 02 00 (ntimes fires once)
B0 02 00 (ntimes fires once more)
B0 02 00 (ntimes fires once more, to match the change of +3)

So in this example the ntimes clause is output exactly as many times as the size of the change indicated via rel2s.

Notes:

- ntimes always applies, even if it occurs after ifp or ifn. Consider the pattern ‘ifp ntimes A ifn B’: the BC of course repeats A (e.g. a sequence of byte values) upon a positive change, but it also repeats B upon a negative change, even though ntimes itself appears to lie within the ifp clause.
- Checksum definitions (cks-1, cks-2, cks-3) occurring after ntimes calculate their checksums only once; so the checksum is the same in each repeat.
14.6.6 Length of custom MIDI output definitions

The BC’s internal data buffer for a control element’s custom output statement(s) contains 127 bytes. The various constituents take up the following number of bytes:

<table>
<thead>
<tr>
<th>Constituent</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.tx (i.e. the statement identifier itself)</td>
<td>2</td>
</tr>
<tr>
<td>MIDI byte $00 .. $FD, $FF</td>
<td>1</td>
</tr>
<tr>
<td>MIDI byte $FE (Active Sensing)</td>
<td>2</td>
</tr>
<tr>
<td>Special identifier (val, releffs, cks-1 etc.)</td>
<td>2</td>
</tr>
<tr>
<td>Offset (the 14-bit argument to releffs and relsign)</td>
<td>2</td>
</tr>
<tr>
<td>StartByteIndex (the 7-bit Checksum Definition argument)</td>
<td>1</td>
</tr>
</tbody>
</table>

So e.g.:
- If a custom output statement contains one val, you have only 123 bytes available for other MIDI bytes, special identifiers and arguments, because the .tx identifier itself takes two bytes and the val another two.
- ‘releffs $0040’ takes up four bytes: two for releffs, two for $0040.
- ‘cks-1 6’ takes up three bytes: two for cks-1, one for 6.
14.7 Local

BCL syntax: `.local Local
Local ∈ {off, on}

A `.local` statement is legal (i.e. accepted by the BCF and BCR) in `$button` and `$encoder` sections (but not under `$fader`). Its function is unknown. The BC never includes it when it sends a preset dump, so maybe it's not actually stored in the button or encoder.

One might think that `.local off` prevents the BC from sending MIDI messages when a particular button or encoder is being moved (similar to the EXIT button’s local-off procedure described in section 4.6 of the B-Control manual). However, this appears not to be the case. So...? ***

It also seems that neither `.local off` nor `.local on` blocks parameter feedback from the computer to the BC.
14.8 Standard vs. custom MIDI output

Contrary to what the B-Control-Tokenreferenz web page claims, the BC can output a control element’s standard MIDI output and its custom MIDI output together. All you have to do is include .easypar and .minmax/.tx in the same BCL section for that control element’s definition, where .easypar must occur before .minmax and .tx. (The order of .minmax versus .tx doesn’t matter.)

So for example:

$encoder 1
  .easypar CC 1 0 127 absolute
  .showvalue on
  .mode 1dot
  .resolution 96 96 96 96
  .default 0
  .minmax 127 0
  .tx $F0 $7D val cks-1 1 $F7

An element definition like this has 2 effects:

1. When the BC sends a preset dump (which occurs for instance when you press EDIT + ‘PRESET *’), only the .easypar definition is included, not the .tx/.minmax definitions. (In other words, the BC is buggy in this respect!)

2. Whenever you physically move the control element, the BC first sends the standard MIDI output (as defined via .easypar), then the custom MIDI output (as defined via .tx).

   Of course the standard output and the custom output always use the same control value (or change). In fact, this value is determined exclusively by the .minmax arguments Value1 and Value2: .easypar’s Value1 and Value2 parameters are completely irrelevant, even though they are misleadingly included in the preset dump!

   So in the example above, the .minmax statement causes reverse direction, as demonstrated via the encoder’s dot, the BC’s display, the CC message and the SysEx message alike.

In fact, if you leave out the .tx line from the above BCL script, .minmax still affects the control element’s behavior and the MIDI message(s) defined via .easypar, although the .minmax definition is of course still not sent back in preset dumps.

   So if you send

$button 52
  .easypar CC 1 1 0 10 toggleon
  .minmax 20 30

to the BC, a preset dump back from the BC returns this as

$button 52
  .easypar CC 1 1 0 10 toggleon

but pressing the button itself repeatedly causes the BC to output a sequence of 30, 20, 30, 20, ...! This is very curious behavior.

---

5 I also think that the value for Default was set to .minmax’s Value2 (so 30 in the example) when I stored this to a memory preset, then recalled that memory preset, but I haven’t examined this phenomenon any further.
14.9 Value synchronization

If two or more elements (buttons, encoders or faders) refer to the same ‘MIDI entity’ (Channel and Controller, NRPN, Scope etc.), their behavior is affected by a special feature of the BC: the BC synchronizes the Value of all elements which refer to the same MIDI entity. That is: when you physically move one element and thereby change its Value, the Value of any other element referring to the same MIDI entity is updated to the moving element’s Value.

Notes:

- Value synchronization only applies to elements having standard output (defined via .easypar), not to elements having only custom output (defined via .tx statements).
- Value synchronization works for any combination of elements. That is: you can mix buttons, encoders and faders.
- Different elements referring to the same MIDI entity can actually have different values for Default (although it’s probably unwise to define them this way), but once you have moved one of them, their Value settings remain synchronized.

Value synchronization is useful in various circumstances. On the following pages some possibilities are discussed:
14.9.1 Button increment mode

As described in §15.10, the increment mode causes a button to step through its value range (from Value2 to Value1). But how do you go backwards in the same range? If the range is very small (up to — say — five values),6 it may be feasible to simply keep pressing the button until the value wraps back. However, if the range is larger, this is impractical. What you then need is a second button, one that steps through the range in the opposite direction. Thus, you get a pair of buttons, where one has a positive increment and the other a negative increment. The following pair of button definitions exemplifies this setup for the Main Volume controller (CC#7):

```
$button 49
  .easypar CC 1 7 127 0 increment 1
  .showvalue on
  .default 64
$button 51
  .easypar CC 1 7 127 0 increment -1
  .showvalue on
  .default 64
```

Note that button combinations like this are only viable because the BC applies value synchronization: if you press one button (thereby changing its Value), the Value of the other button is updated to the pressed button’s Value. So if you press button 49 in the above example, the Value of both buttons becomes 65. Then, if you press button 51, the Value of both buttons becomes 64 (if the BC didn’t apply value synchronization, the output value would be 63 at this point).

It is also possible to combine buttons with different increments in this way. This can give you great flexibility. For instance:

```
$button 49
  .easypar CC 1 7 100 0 increment 1
  .showvalue on
  .default 50
$button 50
  .easypar CC 1 7 100 0 increment 10
  .showvalue on
  .default 50
$button 51
  .easypar CC 1 7 100 0 increment -1
  .showvalue on
  .default 50
$button 52
  .easypar CC 1 7 100 0 increment -10
  .showvalue on
  .default 50
```

---

6 As Royce Craven has pointed out to me, this is often the case with program banks.
14.9.2 Encoder resolutions

You can have one encoder provide ‘coarse’ editing and another ‘fine’ editing. For instance:

```plaintext
$encoder 1 ; coarse
   .easypar CC 1 7 0 100 absolute
   .showvalue on
   .mode bar
   .resolution 100
   .default 50
$encoder 2 ; fine
   .easypar CC 1 7 0 100 absolute
   .showvalue on
   .mode bar
   .resolution 10
   .default 50
```
14.9.3 Frankenstein faders (a.k.a. fader calibration test)

For the BCF’s faders it’s a bit harder to come up with any really useful application of value synchronization, but you can try the setup below for the ultimate spooky effect. It’s also somewhat useful for checking whether your faders and their motors are calibrated.

```
$fader 1
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 2
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 3
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 4
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 5
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 6
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 7
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
$fader 8
  .easypar CC 1 7 0 100 absolute
  .showvalue on
  .motor on
  .default 50
```

You should also try this in combination with the button and encoder examples mentioned above! It’s also fun to set the eight faders to different ranges, e.g. 0-30, 10-40, 20-50, etc.
15 Buttons

A button section in a BCL block is introduced by a button selector statement:

BCL syntax: `$button Button$

Button $\in \{1 .. 64\}$

<table>
<thead>
<tr>
<th>Button</th>
<th>Group/Row</th>
<th>Activity condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 .. 8</td>
<td>Push Encoders (Group 1)</td>
<td></td>
</tr>
<tr>
<td>9 .. 16</td>
<td>Push Encoders (Group 2) $EncoderGroups \geq 2$</td>
<td></td>
</tr>
<tr>
<td>17 .. 24</td>
<td>Push Encoders (Group 3) $EncoderGroups \geq 3$</td>
<td></td>
</tr>
<tr>
<td>25 .. 32</td>
<td>Push Encoders (Group 4) $EncoderGroups = 4$</td>
<td></td>
</tr>
<tr>
<td>33 .. 40</td>
<td>Keys (Upper Row)</td>
<td></td>
</tr>
<tr>
<td>41 .. 48</td>
<td>Keys (Lower Row)</td>
<td></td>
</tr>
<tr>
<td>49 .. 52</td>
<td>User Keys (i.e. at bottom right of BC)</td>
<td></td>
</tr>
<tr>
<td>53 .. 56</td>
<td>Function Keys (STORE etc.) $FunctionKeys = \text{off}$</td>
<td></td>
</tr>
<tr>
<td>57 .. 60</td>
<td>ENCODER GROUPS</td>
<td>see §13.4</td>
</tr>
<tr>
<td>61</td>
<td>BCF2000: Foot Switch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCR2000: Foot Switch 1</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>BCR2000: Foot Switch 2 $Device is BCR2000$</td>
<td></td>
</tr>
<tr>
<td>63 .. 64</td>
<td>PRESET (• and ◦)</td>
<td>$Lock = \text{on}$</td>
</tr>
</tbody>
</table>

If the pertinent activity condition is not met, the button (or foot switch) doesn’t react when you press it: no new value is shown in the BC’s display, and no MIDI data is sent.

However, the BC maintains (i.e. accepts and returns) any button’s BCL definition at all times, even when its activity condition is not being met. (The BCF2000 even maintains ‘button’ 62, although the BCF2000 doesn’t even have Foot Switch 2!)
A button selector statement sets all the selected button’s settings to the following values:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardOutput</td>
<td>cleared</td>
</tr>
<tr>
<td>ShowValue</td>
<td>off</td>
</tr>
<tr>
<td>Default</td>
<td>off</td>
</tr>
<tr>
<td>Mode</td>
<td>down</td>
</tr>
<tr>
<td>Value1</td>
<td>0</td>
</tr>
<tr>
<td>Value2</td>
<td>0</td>
</tr>
<tr>
<td>CustomOutput</td>
<td>cleared</td>
</tr>
<tr>
<td>Local</td>
<td>unknown (probably off)</td>
</tr>
</tbody>
</table>

A button section sent by a BC contains a subset of the following dot statements (in this order); individual statements are only sent in certain situations, as indicated:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.easypar</td>
<td>StandardOutput is defined</td>
</tr>
<tr>
<td>.showvalue</td>
<td>ShowValue always</td>
</tr>
<tr>
<td>.default</td>
<td>Default $\neq$ off</td>
</tr>
<tr>
<td>.mode</td>
<td>StandardOutput is not defined</td>
</tr>
<tr>
<td>.minmax</td>
<td>StandardOutput is not defined</td>
</tr>
<tr>
<td>.tx</td>
<td>both these conditions are met:</td>
</tr>
<tr>
<td></td>
<td>• StandardOutput is not defined</td>
</tr>
<tr>
<td></td>
<td>• CustomOutput is defined</td>
</tr>
</tbody>
</table>

Note: the BC never sends any .local statement.

Details on these dot statements follow on the next pages.
15.1 Standard output

The *Type* argument of a `.easypar` statement for a button must be one of the following identifiers:

- PC
- CC
- NRPN
- NOTE
- AT
- MMC
- GS/XG

The individual cases are described on the following pages.

A `.easypar` statement in a button section has the following side-effects:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowValue</td>
<td>off</td>
</tr>
<tr>
<td>Default</td>
<td>depends on actual <code>.easypar</code> statement</td>
</tr>
<tr>
<td>Mode</td>
<td></td>
</tr>
<tr>
<td>Value1</td>
<td></td>
</tr>
<tr>
<td>Value2</td>
<td></td>
</tr>
<tr>
<td>CustomOutput</td>
<td>cleared</td>
</tr>
<tr>
<td>Local</td>
<td>unknown</td>
</tr>
</tbody>
</table>

‘Side-effect’ here means that these settings aren’t included as arguments in the `.easypar` statement itself, but are changed by the BC ‘behind your back’. However, many `.easypar` statements do have *Mode*, *Value1* and *Value2* as arguments, so strictly speaking their new values are then ‘main’ effects, rather than ‘side’ effects.
15.2 Program Change

BCL syntax: `.easypar PC Channel BankMSB BankLSB Program`

Side-effects:
If `Program = off`: Default, `Value1` and `Value2` become 0
If `Program * off`: Default, `Value1` and `Value2` become `Program`
`Mode` becomes `down`

MIDI output:
1. If `BankMSB * off`: $Bc $00 BankMSB
2. If `BankLSB * off`: $Bc $20 BankLSB
3. If `Program * off`: $Cc Program

Definitions:
`Channel` ∈ {1 .. 16}
c = `Channel` - 1
`BankMSB, BankLSB, Program` ∈ {off, 0 .. 127}
15.3 Control Change

BCL syntax:

If $Mode \in \{\text{toggleoff, toggleon}\}$:
  \[.easypar CC \text{ Channel} \text{ Controller Value1 Value2 Mode}\]

If $Mode = \text{increment}$:
  \[.easypar CC \text{ Channel} \text{ Controller Value1 Value2 Mode Increment}\]

Side-effects:
If $Value2$ is \text{off}, \text{Default} becomes 0, else \text{Default} becomes $Value2$

---

MIDI output:

If $Mode = \text{toggleoff}$:
  \[\text{On: } \$\text{bc Controller Value1_LSB}\]
  \[\text{Off: if } Value2 \neq \text{off} \Rightarrow \$\text{bc Controller Value2_LSB}\]

If $Mode = \text{toggleon}$:
  \[\text{On: } \$\text{bc Controller Value1_LSB}\]
  \[\text{Off: } \$\text{bc Controller Value2_LSB}\]

If $Mode = \text{increment}$:
  \[\$\text{bc Controller Value_LSB}\]

---

Definitions:
Channel $\in \{1 .. 16\}$
$c = \text{Channel} - 1$
Controller $\in \{0 .. 127\}$
Value1 $\in \{0 .. 16383\}$
Value2 $\in \{\text{off}, 0 .. 16383\}$
  (if $Mode$ is \text{toggleon}, the BC converts a received \text{off} to 0; if $Mode$ is \text{increment}, \text{off} is retained but acts as 0 in the MIDI output algorithm)
Mode $\in \{\text{toggleoff, toggleon, increment}\}$
Increment $\in \{-127 .. -1, 1 .. 127\}$ (0 is not allowed: causes BCL error 11)
Value1_LSB = Value1 and $\$7F$
Value2_LSB = Value2 and $\$7F$
Value_LSB = Value and $\$7F$
Value: see §15.10

The 14-bit range (0 .. 16383) of Value1 and Value2 is anomalous, since MIDI Control Change messages can only send values in the range of 0 .. 127:
- The BC accepts any received 14-bit value: the BCL reply message reports ‘no error’.
- In preset dumps, the BC retains any received 14-bit value, and Default (i.e. the argument to the .default statement) is set to Value2 unconditionally, i.e. even if Value2 is higher than 127.
- If you edit Value1 or Value2 on the BC itself (by pressing the EDIT button, then manipulating push encoder 4 or 5 respectively), the BC displays any existing 14-bit value. However, you are not
allowed to raise the parameter to a value above 127; so for instance you can lower 10000 to 9999, but when you try to raise 10000 to 10001, the display jumps to 127.

- The algorithm for calculating Value (see §15.10) uses the full Value1 and Value2 (not just their lowest 7 bits, i.e. Value1_LSB and Value2_LSB).
- In actual MIDI Control Change messages the BC sends the value’s lowest 7 bits.
15.4 NRPN (Non-Registered Parameter Number)

BCL syntax:

If \( \text{Mode} \in \{\text{toggleoff}, \text{toggleon}\} \):
\[
\text{.easypar NRPN Channel NRPN Value1 Value2 Mode}
\]

If \( \text{Mode} = \text{increment} \):
\[
\text{.easypar NRPN Channel NRPN Value1 Value2 Mode Increment}
\]

Side-effects:
If Value2 is \text{off}, Default becomes 0, else Default becomes Value2

MIDI output:

If \( \text{Mode} = \text{toggleoff} \):
- On:
  1. \$Bc $63 NRPN_MSB
  2. \$Bc $62 NRPN_LSB
  3. \$Bc $06 Value1_LSB
- Off: if Value2 * \text{off}:
  1. \$Bc $63 NRPN_MSB
  2. \$Bc $62 NRPN_LSB
  3. \$Bc $06 Value2_LSB

If \( \text{Mode} = \text{toggleon} \):
- On:
  1. \$Bc $63 NRPN_MSB
  2. \$Bc $62 NRPN_LSB
  3. \$Bc $06 Value1_LSB
- Off:
  1. \$Bc $63 NRPN_MSB
  2. \$Bc $62 NRPN_LSB
  3. \$Bc $06 Value2_LSB

If \( \text{Mode} = \text{increment} \):
  1. \$Bc $63 NRPN_MSB
  2. \$Bc $62 NRPN_LSB
  3. \$Bc $06 Value_LSB

One might think that the BC could use the 14-bit range of \( \text{Value1} \) and \( \text{Value2} \) to send 14-bit NRPN data, but unfortunately this is not the case: the BC only sends NRPN data entry \( \text{MSB} \) messages (never data entry \( \text{LSB} \) messages), with \( \text{Value1} \) or \( \text{Value2}'s \) lowest 7 bits.

Definitions:
\( \text{Channel} \in \{1 .. 16\} \)
\( c = \text{Channel} - 1 \)
\( \text{NRPN, Value1} \in \{0 .. 16383\} \)
\( \text{Value2} \in \{\text{off}, 0 .. 16383\} \)
(if \textit{Mode} is \texttt{toggleon}, the BC converts a received \texttt{off} to 0; if \textit{Mode} is \texttt{increment}, \texttt{off} is retained but \texttt{acts} as 0 in the MIDI output algorithm)

\begin{align*}
\text{NRPN\_MSB} &= \text{NRPN} \downarrow \text{7 (i.e. the 7 highest bits of \text{NRPN})} \\
\text{NRPN\_LSB} &= \text{NRPN} \, \& \, \text{$7F$ (i.e. the 7 lowest bits of \text{NRPN})} \\
\text{Value1\_LSB} &= \text{Value1} \, \& \, \text{$7F$} \\
\text{Value2\_LSB} &= \text{Value2} \, \& \, \text{$7F$} \\
\text{Mode} &\in \{\texttt{toggleoff, toggleon, increment}\} \\
\text{Increment} &\in \{-127 .. -1, 1 .. 127\} \text{ (0 is not allowed: causes BCL error 11)} \\
\text{ValueLSB} &= \text{Value} \, \& \, \text{$7F$} \\
\text{Value: see §15.10}
\end{align*}

As for \texttt{.easypar CC}, the 14-bit range of \textit{Value1} and \textit{Value2} is anomalous.
15.5 Note

BCL syntax: .easypar NOTE Channel Note Velocity Mode

Side-effects:
Default becomes 0
Value1 becomes 0
Value2 becomes Velocity

MIDI output:
On: $9c Note Velocity
Off: $9c Note 0

In any co-occurring custom output messages (as defined via .tx statements), Value (cf. val etc. in the .tx statement) is set to Velocity (not to Note!) for the ‘On’ event, and to 0 for the ‘Off’ event.

MIDI input:
The button responds to any incoming MIDI Note On message on the same channel (i.e. with status byte $9c) specifying the same note number. The button’s LED switches on (or stays on) if the incoming message’s velocity byte is greater than the button’s Velocity div 2; otherwise the LED switches off (or stays off).

Note that the button does not respond to incoming MIDI Note Off messages (with status byte $8c).

Definitions:
Channel ∈ {1 .. 16}
c = Channel - 1
Note ∈ {0 .. 127}

To the BC: Velocity ∈ {0 .. 127}
From the BC: Velocity ∈ {1 .. 127}
Mode ∈ {toggleoff, toggleon}

Velocity can be 0 when sent to the BC, but the BC converts this to 1. (In fact, you can even enter 0 via the manual edit mode on the BC, but this value is converted to 1 too as soon as you leave edit mode!)
15.6 Aftertouch

BCL syntax:

If $Mode \in \{\text{toggleoff, toggleon}\}$:

$$\text{.easypar AT} \text{ Channel Scope Value1 Value2 Mode}$$

If $Mode = \text{increment}$:

$$\text{.easypar AT} \text{ Channel Scope Value1 Value2 Mode Increment}$$

Side-effects:
If Value2 is off, Default becomes 0, else Default becomes Value2

MIDI output:

If $Mode = \text{toggleoff}$:

If $Scope = \text{all}$:

On: $Dc \text{ Value1}$
Off: if $Value2 \text{ off}$: $Dc \text{ Value2}$
These are Channel Aftertouch messages.

If $Scope \in \{0 .. 127\}$:

On: $Ac \text{ Scope Value1}$
Off: if $Value2 \text{ off}$: $Ac \text{ Scope Value2}$
These are Note Aftertouch messages, with Scope indicating the note number.

If $Mode = \text{toggleon}$:

If $Scope = \text{all}$:

On: $Dc \text{ Value1}$
Off: $Dc \text{ Value2}$
These are Channel Aftertouch messages.

If $Scope \in \{0 .. 127\}$:

On: $Ac \text{ Scope Value1}$
Off: $Ac \text{ Scope Value2}$
These are Note Aftertouch messages.

If $Mode = \text{increment}$:

If $Scope = \text{all}$:

On: $Dc \text{ Value}$
This is a Channel Aftertouch message.

If $Scope \in \{0 .. 127\}$:

On: $Ac \text{ Scope Value}$
This is a Note Aftertouch message.

Definitions:

$Channel \in \{1 .. 16\}$
$c = Channel - 1$
$Scope \in \{\text{all, 0 .. 127}\}$
Value1 ∈ \{0 .. 127\}
Value2 ∈ \{off, 0 .. 127\}

(if Mode is toggleon, the BC converts a received off to 0; if Mode is increment, off is retained but acts as 0 in the MIDI output algorithm)

Mode ∈ \{toggleoff, toggleon, increment\}
Increment ∈ \{-127 .. -1, 1 .. 127\} (0 is not allowed: causes BCL error 11)

Value: see §15.10
15.7 MMC (MIDI Machine Control)

BCL syntax: `.easypar MMC Device Command Location FrameRate`

Side-effects:
- `Default` becomes a value associated with `FrameRate`, i.e. 0, 24, 25, 30 and 30 for `noloc, 24f, 25f, 30df` and `30f` respectively. (I have no idea what the point of this is, but never mind.)
- `Mode` becomes `down`
- `Value1` and `Value2` become $60 \times H$ + $M$

MIDI output:

1. If `FrameRate` ≠ `noloc`:
   - `$F0 \ 7F \ DeviceByte \ 06 \ 44 \ 06 \ 01 \ FrameRate_H \ M \ S \ F \ 00 \ 7F`
   - This is a ‘locate’ message (cf. `06 \ 01`). (*** But what does `06 \ 44` mean here? ***)

2. If `Command` ≠ `locate`:
   - `$F0 \ 7F \ DeviceByte \ 06 \ CommandByte \ 7F$

As the above definitions imply, there is no MIDI output if `Command` = `locate` and `FrameRate` = `noloc`.

In any co-occurring custom output messages (as defined via `.tx` statements), `Value` (cf. `val` etc. in the `.tx` statement) is set to $60 \times H + M$.

Definitions:

- `Device \in \{all, 0 .. 126\}`
  - `Device = all`: `DeviceByte = 7F`
  - `Device \in \{0 .. 126\}`: `DeviceByte = Device`

- `Command \in \{play, pause, stop, fwd, rew, locate, punch-in, punch-out\}`
<table>
<thead>
<tr>
<th>Command</th>
<th>CommandByte</th>
</tr>
</thead>
<tbody>
<tr>
<td>play</td>
<td>$02</td>
</tr>
<tr>
<td>pause</td>
<td>$09</td>
</tr>
<tr>
<td>stop</td>
<td>$01</td>
</tr>
<tr>
<td>fwd</td>
<td>$04</td>
</tr>
<tr>
<td>rew</td>
<td>$05</td>
</tr>
<tr>
<td>locate</td>
<td>$01 (N/A)</td>
</tr>
<tr>
<td>punch-in</td>
<td>$06</td>
</tr>
<tr>
<td>punch-out</td>
<td>$07</td>
</tr>
</tbody>
</table>

**Location:**

FrameRate = noloc:

- Location is not used, but it is still syntactically required as an argument. You can send it to the BC as any sequence of characters except spaces and semicolons; the BC always sends 00:00:00.00 in preset dumps.

FrameRate ≠ noloc:

- Location = $H:\{0 .. 23\}$
- $M, S \in \{0 .. 59\}$
- $F \in \{0 .. f - 1\}$, where $f$ is the actual frame rate (i.e. 24, 25 or 30, cf. FrameRate)

$H, M, S$ and $F$ must each be written as exactly 2 digits (i.e. if necessary with a leading zero)

FrameRate ∈ \{noloc, 24f, 25f, 30df, 30f\}

<table>
<thead>
<tr>
<th>FrameRate</th>
<th>FrameRateBits</th>
<th>Frames per second</th>
<th>Standard usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>noloc</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24f</td>
<td>$00$</td>
<td>24</td>
<td>Film</td>
</tr>
<tr>
<td>25f</td>
<td>$20$</td>
<td>25</td>
<td>TV: PAL</td>
</tr>
<tr>
<td>30df</td>
<td>$40$</td>
<td>29.97</td>
<td>TV: NTSC (‘drop frame’)</td>
</tr>
<tr>
<td>30f</td>
<td>$60$</td>
<td>30</td>
<td>Inaccurate simplification of NTSC</td>
</tr>
</tbody>
</table>

FrameRate_H = FrameRateBits or H

---

7 Royce Craven provided the info in this column.
15.8 GS/XG

BCL syntax: `.easypar GS/XG Channel Parameter Value1 Value2 Mode`

Side-effects:
If `Value2` is `off`, `Default` becomes 0, else `Default` becomes `Value2`.

MIDI output:

If `Mode = toggleoff`:
   If `Parameter`’s type is NRPN:
      On:
      1. $Bc $62 NRPN_MSB
      2. $Bc $63 NRPN_LSB
      3. $Bc $06 Value1
      Off: if `Value2` *off*:
         1. $Bc $62 NRPN_MSB
         2. $Bc $63 NRPN_LSB
         3. $Bc $06 Value2
   If `Parameter`’s type is CC:
      On: $Bc CC_Controller Value1
      Off: if `Value2` *off*: $Bc CC_Controller Value2

If `Mode = toggleon`:
   If `Parameter`’s type is NRPN:
      On:
      1. $Bc $62 NRPN_MSB
      2. $Bc $63 NRPN_LSB
      3. $Bc $06 Value1
      Off:
      1. $Bc $62 NRPN_MSB
      2. $Bc $63 NRPN_LSB
      3. $Bc $06 Value2
   If `Parameter`’s type is CC:
      On: $Bc CC_Controller Value1
      Off: $Bc CC_Controller Value2

Definitions:
`Channel ∈ {1 .. 16}`
`c = Channel – 1`
`Parameter, NRPN_MSB, NRPN_LSB` and `CC_Controller`: see the tables in §14.1.1
`Value1 ∈ {0 .. 127}`
`Value2 ∈ {off, 0 .. 127}` (if `Mode` is `toggleon`, the BC converts a received `off` to 0)
`Mode ∈ {toggleoff, toggleon}`
15.9 Mode

BCL syntax:

If \( \text{Mode} \in \{ \text{down, updown, toggle} \} \):
\[
\text{.mode} \ \text{Mode}
\]

If \( \text{Mode} = \text{incval} \):
\[
\text{.mode incval} \ \text{Increment}
\]
Where \( \text{Increment} \in \{-127..-1, 1..127\} \).

Beware: the BC incorrectly returns the \text{incval} mode as \text{down}.

The \text{.mode} statement is primarily intended to be used in conjunction with \text{.tx} statements, i.e. for custom output. However, a button has only one \text{Mode} setting; that is: any \text{.mode} statement also affects any standard output defined via a previous \text{.easypar} statement.

All in all there are three BCL statement types that affect \text{Mode}:

- \$button \text{ Button} (this sets \text{Mode} to \text{down})
- \text{.easypar} \text{ Type } \ldots \text{ Mode}
- \text{.mode} \text{ Mode}

Obviously, the most recent occurrence of these BCL statement types determines the current value of \text{Mode}. So if \text{.easypar} has come last, its \text{Type} and \text{Mode} arguments are the decisive factors; if a \text{.mode} statement has come last, its \text{Mode} argument decides.

The \text{Mode} setting affects the button’s behavior as follows:

Provided that \text{ShowValue} is \text{on}, the BC’s display briefly shows the following values at the moments when you push and release the button sequentially:

<table>
<thead>
<tr>
<th>.easypar</th>
<th>Type</th>
<th>PC, MMC</th>
<th>CC, NRPN, NOTE*, AT, GS/XG*</th>
</tr>
</thead>
<tbody>
<tr>
<td>.mode</td>
<td>Mode</td>
<td>down</td>
<td>updown</td>
</tr>
</tbody>
</table>

| Push 1st time | on | on | on | Default + Increment |
| Release 1st time | oFF | oFF | on | - |
| Push 2nd time | on | on | oFF | Default + 2×Increment |
| Release 2nd time | oFF | oFF | oFF | - |

* Actually, \text{NOTE} and \text{GS/XG} cannot use \text{increment}.

However, the protocols for \text{down} and \text{toggleon/toggle} specify that the BC doesn’t actually send the MIDI message(s) associated with the flashed value when you release the button. So if \( A \) stands for the actual MIDI message(s) associated with the button’s \text{on} state, and \( B \) for the one(s) associated with the \text{oFF} state, we find the following MIDI output pattern:
<table>
<thead>
<tr>
<th>.easypar Type</th>
<th>PC, MMC</th>
<th>CC, NRPN, NOTEx, AT, GS/XGx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>toggleoff</td>
<td>toggleon</td>
</tr>
<tr>
<td>1</td>
<td>down</td>
<td>updown</td>
</tr>
</tbody>
</table>

The table above shows the behavior of different modes:

<table>
<thead>
<tr>
<th>Push 1st time</th>
<th>Release 1st time</th>
<th>Push 2nd time</th>
<th>Release 2nd time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Default + Increment</td>
<td>–</td>
<td>Default + 2×Increment</td>
<td></td>
</tr>
</tbody>
</table>

Note that it is legal to send .easypar and .mode statements to the BC in the same $button section. For instance:

```
$button 33
  .easypar CC 1 1 127 0 Mode1 1
  .showvalue on
  .mode Mode2
```

However, this construct is inadvisable, because Mode2 in the .mode statement overwrites Mode1 set by the .easypar statement. This can mess up the button’s behavior:

As the above tables show, .easypar’s toggleoff and .mode’s updown cause identical behavior, and so do toggleon and toggle, and increment and incval. So it would be safe to follow the .easypar version by the .mode version in these cases (but why would you?). However, certain other combinations may cause the button to behave rather strangely. For instance, putting ‘.mode down’ after CC, NRPN, NOTE, AT or GS/XG causes the BC to only output the button’s down message (A), never the up message (B); note that you can achieve the same result by using toggleon/toggle and setting Value1 (hence A) to the same value as Value2 (hence B). Flatly disastrous is what happens to .easypar’s increment mode if you include any subsequent .mode statement other than .mode incval: the button then no longer functions incrementally!

And as if things weren’t confusing enough: when the BC sends a ‘double-mode’ definition back to the computer (via a subsequent preset dump), the original Mode argument of .easypar is retained, but the .mode statement is not included. In fact, the BC never outputs a .mode statement for a button that uses .easypar.
15.10 Increment mode

A button responds incrementally if the Mode argument of `.easypar CC, NRPN or AT is increment` or if the Mode argument of `.mode is incval`. In either case, the Increment argument determines the jump size for each successive time the button is pressed.

For `.easypar`, the following algorithm is used to calculate the sequence of values which the BC outputs in MIDI messages. (Actually, for `.easypar CC and NRPN` the BC only outputs the least significant 7 bits; for AT `Value` is only 7 bits anyway.)

- Before the button is pressed for the first time, `Value` is `Default`. (Obviously this value is not output!)
- Each time the button is pressed, the intermediate variable `N` is calculated as `Value + Increment`. (Remember that `Increment` can be negative, so in that case `N` is lower than `Value`.)
  - If `Increment > 0`: if `N` is higher than `Value1`, `Value` is set to `Value2`, otherwise to `N`.
  - If `Increment < 0`: if `N` is lower than `Value2`, `Value` is set to `Value1`, otherwise to `N`.

Finally, the new `Value` is incorporated in the appropriate MIDI message(s), and shown in the BC’s display if `ShowValue` is on.

Several aspects of this algorithm are worth highlighting:

- `Value1` must always be higher than `Value2`, otherwise `Value` will immediately stick at `Value1` if `Increment` is negative, and at `Value2` if `Increment` is positive. This occurs irrespective of the value of `Default`.
- The algorithm produces cyclical output: after reaching one end of the range, `Value` wraps around to the other end.
- Remember that a `.easypar` statement sets `Default` and `Value` to `Value2`. Therefore, if there is no `.default` statement after the `.easypar` statement, the following behavior occurs:
  - If `Increment > 0`, the first value sent is `Value2 + Increment`.
  - If `Increment < 0`, the first value sent is `Value1`. (This is because `Value2 + Increment` is lower than `Value2`, so wrapping takes place.)

For example:
```
.easypar CC 1 1 10 1 increment 2  =>  3, 5, 7, 9, 1, 3, ...
.easypar CC 1 1 10 1 increment -2 =>  10, 8, 6, 4, 2, 10, 8, ...
```

So in this situation there is effectively a mismatch between the output for a positive and for a negative `Increment`: `Value2` is initially skipped if `Increment` is positive, but `Value1` is not skipped if `Increment` is negative.

See the ‘Value synchronization’ section in §14.9 for examples of multiple `increment` buttons referring to the same ‘MIDI entity’.

The algorithm above applies in almost the same way for a custom output definition, i.e. one using `.mode incval, .minmax and .tx` instead of `.easypar`. The one crucial difference is that in terms of the above algorithm, the `Value1` and `Value2` arguments of `.minmax` are to be swapped; so whenever the
above algorithm talks about *Value1*, this actually refers to `.minmax`'s *Value2*, and vice versa. Consequently `.minmax`'s *Value1* must be smaller than *Value2*, etc.

So for instance the custom output versions of the `.easypar` examples discussed above are:

```plaintext
.mode incval 2
.minmax 1 10
.tx $B0 0 val
```

and

```plaintext
.mode incval -2
.minmax 1 10
.tx $B0 0 val
```

Note that in both cases `.minmax`'s *Value1* contains the minimum and *Value2* the maximum.
16 Continuous elements (encoders/faders)

16.1 Standard output

The `Type` argument of a `.easypar` statement for an encoder or fader must be one of the following identifiers:

- PC
- CC
- NRPN
- PB
- AT
- GS/XG

The individual cases are described on the following pages.

Note that a fader whose `Motor` setting is `off` does not physically reflect any of the changes to `Default` specified under ‘side-effects’.
16.2 Program Change

BCL syntax: `.easypar PC Channel BankMSB BankLSB`

Side-effects:
- `Default` becomes 0
- `Value1` becomes 0
- `Value2` becomes 127

MIDI output:
1. If `BankMSB` is off: $Bc$00 BankMSB
2. If `BankLSB` is off: $Bc$20 BankLSB
3. $Cc$ Value

Definitions:
- `Channel ∈ {1 .. 16}`
- `c = Channel − 1`
- `BankMSB, BankLSB ∈ {off, 0 .. 127}`
- `Value ∈ {0 .. 127}`
16.3 Control Change

BCL syntax: \texttt{.easypar CC Channel Controller Value1 Value2 Mode}

Side-effects:
\textit{Default} becomes \textit{Value1}

MIDI output:

If \textit{Mode} $\in \{\text{absolute, relative-1, relative-2, relative-3}\}$ or \textit{Controller} $\in \{32 .. 127\}$:
\texttt{SBe Controller Data}

If \textit{Mode} $\in \{\text{absolute/14, relative-1/14, relative-2/14, relative-3/14}\}$ and \textit{Controller} $\in \{0 .. 31\}$:
1. $\texttt{SBe Controller DataMSB}$
2. $\texttt{SBe Controller+32 DataLSB}$

Definitions:
\textit{Channel} $\in \{1 .. 16\}$
\texttt{c} = \textit{Channel} $-1$
\textit{Controller, Data, DataMSB, DataLSB} $\in \{0 .. 127\}$
\textit{Value1, Value2} $\in \{0 .. 16383\}$

This 14-bit range of \textit{Value1} and \textit{Value2} is anomalous for the 7-bit modes (\textit{absolute} .. \textit{relative-3}). The situation is similar to the one concerning the \textit{Value1} and \textit{Value2} arguments of \texttt{.easypar CC} for buttons (see §15.3).

\textit{Mode}:
- \texttt{absolute}
- \texttt{relative-1}
- \texttt{relative-2}
- \texttt{relative-3}
- \texttt{inc/dec}
- \texttt{absolute/14}
- \texttt{relative-1/14}
- \texttt{relative-2/14}
- \texttt{relative-3/14}

Note that \texttt{inc/dec} (which is primarily meant for the \textit{Mode} argument of \texttt{.easypar NRPN}) here results in a largely dysfunctional encoder or fader: it doesn’t send any MIDI messages, although it does trigger value synchronization (cf. §14.9) in any other elements using the same \textit{Channel} and \textit{Controller}.

Note that the MIDI output of a 14-bit mode is in fact that of the corresponding 7-bit mode if \textit{Controller} is 32 or higher, since the MIDI Control Change protocol defines only 32 controller numbers (0-31) for which a corresponding LSB controller number is available (namely 32-63).

How \textit{Data} and \textit{DataMSB/DataLSB} relate to the physical control element depends on the \textit{Mode} argument:
- For \texttt{absolute} and \texttt{absolute/14}, \textit{Data} and \textit{DataMSB/DataLSB} represent the actual position of the control element.
- For the relative modes, \textit{Data} and \textit{DataMSB/DataLSB} are encodings of \textit{Change}. \textit{Change} reflects the physical amount of movement, but in the case of encoders also depends on \texttt{.resolution}’s
arguments: these function as multipliers. If you move the control slowly, you usually trigger a sequence of messages with Change only being −1 or +1: you have to move the control rather quickly to trigger more extreme values for Change.

The table below explains the encoding methods of the 7-bit relative modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Method</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Change</td>
</tr>
<tr>
<td>relative-1</td>
<td>Two’s complement</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>−1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+2</td>
</tr>
<tr>
<td>relative-2</td>
<td>Binary offset ($40)</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>−1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+2</td>
</tr>
<tr>
<td>relative-3</td>
<td>Sign-and-magnitude: Data’s bit 6 indicates Change’s sign:</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>−1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+2</td>
</tr>
</tbody>
</table>

Here are the 14-bit relative encoding modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Method</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Change</td>
</tr>
<tr>
<td>relative-1/14</td>
<td>Two’s complement</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>−1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+2</td>
</tr>
<tr>
<td>relative-2/14</td>
<td>Binary offset ($40 $00)</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>−1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+2</td>
</tr>
<tr>
<td>relative-3/14</td>
<td>Sign-and-magnitude: DataMSB’s bit 6 indicates Change’s sign:</td>
<td>−2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>−1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+2</td>
</tr>
</tbody>
</table>

Of course the BC never actually sends Data or DataMSB/DataLSB if Change is 0; they are only included in the above tables for clarification of the encoding methods.
For the relative modes the BC does maintain the range defined by \textit{Value1} and \textit{Value2}, but only in one respect: the values shown by the BC’s display (provided that \textit{ShowValue} is \textbf{on}) still refuse to go beyond the range defined by \textit{Value1} and \textit{Value2}, no matter how hard you keep turning an encoder’s knob. However, MIDI messages indicating the amount of change are being sent as long as you keep turning the knob, without any regard for \textit{Value1} or \textit{Value2}! (Apparently the idea is that the receiving MIDI device is responsible for handling these ongoing messages correctly.)
16.4 NRPN (Non-Registered Parameter Number)

BCL syntax: .easypar NRPN Channel NRPN Value1 Value2 Mode

Side-effects:
Default becomes Value1

MIDI output:
1. $Bc$63 NRPN_MSB
2. $Bc$62 NRPN_LSB
3. If Mode $\in \{\text{absolute}, \text{relative-1}, \text{relative-2}, \text{relative-3}\}$:
   $Bc$06 Data
   - For an increase: a sequence of one or more $Bc$60 (=Data Increment) $01$
   - For a decrease: a sequence of one or more $Bc$61 (=Data Decrement) $01$
4. If Mode $\in \{\text{absolute/14}, \text{relative-1/14}, \text{relative-2/14}, \text{relative-3/14}\}$:
   1. $Bc$06 Data_MSB
   2. $Bc$26 Data_LSB

Definitions:
Channel $\in \{1 .. 16\}$
c = Channel - 1
NRPN, Value1, Value2 $\in \{0 .. 16383\}$
NRPN_MSB = NRPN shr 7 (i.e. the 7 highest bits of NRPN)
NRPN_LSB = NRPN and $7F$ (i.e. the 7 lowest bits of NRPN)
Mode:
- absolute
- relative-1
- relative-2
- relative-3
- inc/dec
- absolute/14
- relative-1/14
- relative-2/14
- relative-3/14
Data, Data_MSB, Data_LSB $\in \{0 .. 127\}$ (same calculation as under .easypar CC)

For the 7-bit Mode values (absolute, relative-1, relative-2, relative-3 and inc/dec),
the Value1 and Value2 arguments should actually be 7-bit too, i.e. restricted to the range of 0 .. 127.
The situation is similar to the one concerning the Value1 and Value2 arguments of .easypar CC for buttons (see §15.3).
16.5 Pitch Bend

BCL syntax: **.easypar PB Channel Range**

Side-effects:
*Default* becomes 64
*Value1* becomes 64 - *Range* \( \div \) 2
*Value2* becomes 64 + *Range* \( \div \) 2

MIDI output:
$Ec $00 Value

Note: the general MIDI Pitch Bend format is $Ec ValueLSB ValueMSB, so the fact that the BC always sends $00 for *ValueLSB* means that it doesn’t support 14-bit values: this is a bit strange, since the BC does support 14-bit values for several other message types.

Definitions:
*Channel* \( \in \{1..16\} \)
\( c = \text{Channel} - 1 \)
*Range* \( \in \{0 .. 127\} \)
*Value* \( \in \{64 - \text{Range} \div 2 .. 64 + \text{Range} \div 2\} \)

So for instance:
- If *Range* \( \in \{126, 127\} \), then *Value* \( \in \{1 .. 127\} \) (so *Value* is never 0!).
- If *Range* \( \in \{124, 125\} \), then *Value* \( \in \{2 .. 126\} \).
- If *Range* \( \in \{2, 3\} \), then *Value* \( \in \{63 .. 65\} \).
- If *Range* \( \in \{0, 1\} \), then *Value* = 64 (i.e. unchangeable).
16.6 Aftertouch

BCL syntax: `.easypar AT Channel Scope Value1 Value2`

Side-effects:
*Default becomes Value1*

---

MIDI output:

If `Scope = all`:
   
   `$Dc Value`
   
   This is a Channel Aftertouch message.

If `Scope ∈ \{0 .. 127\}`:
   
   `$Ac Scope Value`
   
   This is a Note Aftertouch message, with `Scope` indicating the note number.

---

Definitions:
*Channel ∈ \{1 .. 16\}*
*c = Channel − 1*

*Scope ∈ \{all, 0 .. 127\}*

*Value1, Value2, Value ∈ \{0 .. 127\}*
16.7 GS/XG

BCL syntax: \texttt{.easypar GS/XG Channel Parameter Value1 Value2}

Side-effects:
\texttt{Default} becomes \texttt{Value1}

\begin{itemize}
\item MIDI output:
\end{itemize}

\begin{enumerate}
\item If \texttt{Parameter}'s type is NRPN:
  \begin{enumerate}
  \item $Bc$ $62$ \texttt{NRPN_MSB}
  \item $Bc$ $63$ \texttt{NRPN_LSB}
  \item $Bc$ $06$ \texttt{Value}
  \end{enumerate}
\item If \texttt{Parameter}'s type is CC:
  \begin{enumerate}
  \item $Bc$ \texttt{CC_Controller} \texttt{Value}
  \end{enumerate}
\end{enumerate}

Definitions:
\begin{itemize}
\item \texttt{Channel} $\in \{1 .. 16\}$
\item $c = \text{Channel} - 1$
\item \texttt{Parameter}, \texttt{NRPN_MSB}, \texttt{NRPN_LSB} and \texttt{CC_Controller}: see the tables in §14.1.1
\item \texttt{Value1}, \texttt{Value2}, \texttt{Value} $\in \{0 .. 127\}$
\end{itemize}
17 Encoders

An encoder section in a BCL block is introduced by an encoder selector statement:

BCL syntax: \texttt{$encoder \ Encoder$}

BCF2000: \texttt{Encoder \in \{1 \ldots 32\}}

BCR2000: \texttt{Encoder \in \{1 \ldots 56\}}

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Group/Row</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \ldots 8</td>
<td>Push Encoders (Group 1)</td>
<td>BCF2000/BCR2000</td>
</tr>
<tr>
<td>9 \ldots 16</td>
<td>Push Encoders (Group 2)</td>
<td></td>
</tr>
<tr>
<td>17 \ldots 24</td>
<td>Push Encoders (Group 3)</td>
<td></td>
</tr>
<tr>
<td>25 \ldots 32</td>
<td>Push Encoders (Group 4)</td>
<td></td>
</tr>
<tr>
<td>33 \ldots 40</td>
<td>Encoders (Top Row)</td>
<td>BCR2000</td>
</tr>
<tr>
<td>41 \ldots 48</td>
<td>Encoders (Middle Row)</td>
<td></td>
</tr>
<tr>
<td>49 \ldots 56</td>
<td>Encoders (Bottom Row)</td>
<td></td>
</tr>
</tbody>
</table>

An encoder selector statement sets all that encoder’s settings to the following values:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{StandardOutput}</td>
<td>cleared</td>
</tr>
<tr>
<td>\textit{ShowValue}</td>
<td>\texttt{off}</td>
</tr>
<tr>
<td>\textit{Mode}</td>
<td>\texttt{off}</td>
</tr>
<tr>
<td>\textit{R}_1 \ \textit{R}_2 \ \textit{R}_3 \ \textit{R}_4</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>\textit{Default}</td>
<td>\texttt{off}</td>
</tr>
<tr>
<td>\textit{Value1}</td>
<td>0</td>
</tr>
<tr>
<td>\textit{Value2}</td>
<td>0</td>
</tr>
<tr>
<td>\textit{CustomOutput}</td>
<td>cleared</td>
</tr>
<tr>
<td>\textit{Local}</td>
<td>unknown (probably \texttt{off})</td>
</tr>
</tbody>
</table>
If at least one setting for a particular encoder deviates from its default, the BC includes a corresponding encoder section in a preset dump. This section contains a subset of the following dot statements, in this order; individual statements are only sent in certain situations, as indicated:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.easypar</td>
<td>StandardOutput is defined</td>
</tr>
<tr>
<td>.showvalue</td>
<td>ShowValue always</td>
</tr>
<tr>
<td>.mode</td>
<td>Mode always</td>
</tr>
<tr>
<td>.resolution</td>
<td>$R_1$, $R_2$, $R_3$, $R_4$ always</td>
</tr>
<tr>
<td>.default</td>
<td>Default * off</td>
</tr>
<tr>
<td>.minmax</td>
<td>StandardOutput is not defined</td>
</tr>
<tr>
<td>.tx</td>
<td>CustomOutput both these conditions are met:</td>
</tr>
<tr>
<td></td>
<td>- StandardOutput is not defined</td>
</tr>
<tr>
<td></td>
<td>- CustomOutput is defined</td>
</tr>
</tbody>
</table>

Note: the BC never sends any .local statement.

Details on the dot statements that are specific to encoders follow on the next pages.
17.1 Standard output

A `.easypar` statement in an encoder section has the following side-effects:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowValue</td>
<td><code>off</code></td>
</tr>
<tr>
<td>Mode</td>
<td><code>off</code></td>
</tr>
<tr>
<td>R₁, R₂, R₃, R₄</td>
<td>depends on actual <code>.easypar</code> statement</td>
</tr>
<tr>
<td>Default</td>
<td></td>
</tr>
<tr>
<td>Value1</td>
<td></td>
</tr>
<tr>
<td>Value2</td>
<td></td>
</tr>
<tr>
<td>CustomOutput</td>
<td>cleared</td>
</tr>
<tr>
<td>Local</td>
<td>unknown</td>
</tr>
</tbody>
</table>

‘Side-effect’ here means that these settings aren’t included as arguments in the `.easypar` statement itself, but are changed by the BC ‘behind your back’. However, many `.easypar` statements do have `Value1` and `Value2` as arguments, so strictly speaking their new values are then ‘main’ effects, rather than ‘side’ effects.
17.2 Mode

BCL syntax: `.mode Mode`

Encoders 1-32:

\[ Mode \in \{ \text{off, 1dot, 1dot/off, 12dot, 12dot/off, bar, bar/off, spread, pan, qual, cut, damp} \} \]

Encoders 33-56 (BCR2000 only):

\[ Mode \in \{ \text{off, 1dot, 1dot/off} \} \]

Note: in fact encoders 33-56 also accept (and return) the other values defined for encoders 1-32. However, the LEDs for encoders 33-56 don’t actually function according to these values.
17.3 Resolution

BCL syntax:

*To the BC:* `.resolution \( R_1 \) [\( R_2 \) [\( R_3 \) [\( R_4 \)]]]`
\( R_i \in \{1..65535\} \) (however, 32674 or higher leads to invalid behavior; see discussion below)

*From the BC:* `.resolution \( R_1 \) \( R_2 \) \( R_3 \) \( R_4 \)`
\( R_i \in \{0..65535\} \)

The B-Control-Token reference web page claims that only multiples of 96 (or actually 24!) are allowed, but this is not the case; *any* value in the range 1..65535 is allowed for any of the four arguments. That the BC itself often uses *defaults* that are multiples of 96 is irrelevant in this respect.

The four arguments indicate the amounts of value change per 360-degree rotation, in four rotational speed regions in increasing order: \( R_1 \) sets the amount for the lowest speed range, \( R_4 \) for the highest. (I haven’t tried to establish the exact speed thresholds; to do so would be rather tricky.)

If you specify fewer than 4 arguments, the BC sets the left-out argument(s) to the *last* value you do specify. So e.g. `.resolution 96 192` is interpreted as `.resolution 96 192 192 192`.

Note that the \( R \) arguments can be *higher* than the maximum value range of the controller. For instance:

```
$encoder 1
   .easypar CC 1 1 0 127 absolute
   .showvalue on
   .mode 1dot
   .resolution 1000
   .default 0
```

This simply causes the knob to traverse the value range (0 to 127) very quickly, namely in 127/1000 (about 1/8) of a full turn.

Obviously, if you specify the *same* value for all four \( R \) arguments, then the encoder value’s change per rotational *distance* is constant, no matter how *fast* you turn the knob.

So for instance `.resolution 1 1 1 1` (or the equivalent `.resolution 1`) will cause an excruciatingly sluggish response of 1 per rotation at any speed.

On the other hand, `.resolution 10 100 1000 10000` will cause enormous increases in change as you turn faster (or more precisely: whenever you pass a speed threshold).

It is also legal to use *decreasing* values, e.g. `.resolution 10000 1000 100 10`, but obviously this causes the knob to behave in extremely counter-intuitive ways.

As stated above, the *legal* maximum of the four \( R_i \) parameters is 65535. However, due to a bug in the BC’s firmware (in any case version 1.10) the *practical* maximum is 32673. This is because an \( R_i \) of 32674 or higher causes weird jumps in the encoder’s value when you turn the encoder at the speed associated with that \( R_i \), as can be seen from the BC’s display and the MIDI messages the BC outputs. For instance, turning such an encoder clockwise when it is already at its maximum may wrongly lead to a value below that maximum; of course the encoder should simply stay at its maximum. In other situations, turning such an encoder clockwise may fail to reach the maximum, etc.
Resolution defaults

An encoder section selector statement causes the BC to set all that encoder’s resolution values to zero (.resolution 0 0 0 0).

A situation in which these four defaults of 0 are not changed, is undesirable for two reasons:
1. The knob doesn’t do anything when you turn it.
2. If you subsequently have the BC perform a preset dump, the BC duly sends '.resolution 0 0 0 0' to the computer. However, you’d better not send this line back to the BC, because that would lead to a BCL reply with error 11 (‘Argument value out of range’). (So in this case the BC sends data which it doesn’t accept!)

Two dot statements modify an encoder’s resolution values: .easypar and .resolution. Given the initialization performed by the section selector statement, you should always include a valid .resolution statement in any $encoder section that doesn’t include .easypar. Note that a .minmax statement does not affect the resolution values at all.

A .easypar statement sets four ‘reasonable’ resolution values, as follows:

<table>
<thead>
<tr>
<th>PC</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R₁</td>
<td>R₂</td>
<td>R₃</td>
<td>R₄</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CC, NRPN, AT and GS/XG:</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs (Value₂ - Value₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolution</td>
<td>R₁</td>
<td>R₂</td>
<td>R₃</td>
<td>R₄</td>
</tr>
<tr>
<td>0 .. 20</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>21 .. 200</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>201 .. 2000</td>
<td>96</td>
<td>192</td>
<td>384</td>
<td>768</td>
</tr>
<tr>
<td>2001 .. 16383</td>
<td>96</td>
<td>192</td>
<td>768</td>
<td>2304</td>
</tr>
</tbody>
</table>

(Thus, the range of 201 .. 2000 uses 96 times 1, 2, 4 and 8, and the range of 2001 .. 16383 uses 96 times 1, 2, 8 and 24.)

<table>
<thead>
<tr>
<th>PB</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolution</td>
<td>R₁</td>
<td>R₂</td>
<td>R₃</td>
<td>R₄</td>
</tr>
<tr>
<td>0..21(!)</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>22(!) .. 127</td>
<td>96</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
</tbody>
</table>
Of course you can see all these default values by making the BC perform a preset dump. E.g. 
\`.easypar NRPN 1 1 0 16383 absolute\` causes the BC to define \`.resolution 96 192 768 2304\`. 
18 Faders

A fader section in a BCL block is introduced by a fader selector statement:

BCL syntax: $fader  Fader

BCF:
Fader $\in \{1 .. 9\}$ (note that 9 refers to the BCF’s Foot Controller input socket)

BCR:
Any fader selector statement causes the BCR to return BCL error 9: ‘Element number out of range’. However, subsequent fader dot statements are processed as if the fader does exist, in the sense that the BCL Reply error codes follow the same rules as on the BCF. So for instance, if a fader dot statement occurs in a fader section and is syntactically correct, error 0 (‘No error’) is returned. However, the BCR never returns any fader sections in preset dumps.

A fader selector statement assigns the following values to that fader:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardOutput</td>
<td>cleared</td>
</tr>
<tr>
<td>ShowValue</td>
<td>off</td>
</tr>
<tr>
<td>Motor</td>
<td>off</td>
</tr>
<tr>
<td>Override</td>
<td>move</td>
</tr>
<tr>
<td>OverrideButton</td>
<td>off</td>
</tr>
<tr>
<td>Default</td>
<td>off</td>
</tr>
<tr>
<td>Value1</td>
<td>0</td>
</tr>
<tr>
<td>Value2</td>
<td>0</td>
</tr>
<tr>
<td>CustomOutput</td>
<td>cleared</td>
</tr>
</tbody>
</table>
If at least one setting for a particular fader deviates from its default, the BCF includes a corresponding fader section in a preset dump. This section contains a subset of the following dot statements, in this order; individual statements are only sent in certain situations, as indicated:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.easypar StandardOutput</td>
<td>StandardOutput is defined</td>
</tr>
<tr>
<td>.showvalue ShowValue</td>
<td>always</td>
</tr>
<tr>
<td>.motor Motor</td>
<td>always</td>
</tr>
<tr>
<td>.override Override</td>
<td>always</td>
</tr>
<tr>
<td>.keyoverride OverrideButton</td>
<td>always</td>
</tr>
<tr>
<td>.default Default</td>
<td>Default ≠ 0</td>
</tr>
<tr>
<td>.minmax Value1 Value2</td>
<td>StandardOutput is not defined</td>
</tr>
<tr>
<td>.tx CustomOutput</td>
<td>both these conditions are met:</td>
</tr>
<tr>
<td></td>
<td>• StandardOutput is not defined</td>
</tr>
<tr>
<td></td>
<td>• CustomOutput is defined</td>
</tr>
</tbody>
</table>

Details on the dot statements that are specific to faders follow on the next pages.
18.1 Standard output

A `.easypar` statement in a fader section has the following side-effects:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowValue</td>
<td>off</td>
</tr>
<tr>
<td>Motor</td>
<td>off</td>
</tr>
<tr>
<td>Override</td>
<td>move</td>
</tr>
<tr>
<td>OverrideButton</td>
<td>off</td>
</tr>
<tr>
<td>Default</td>
<td></td>
</tr>
<tr>
<td>Value1</td>
<td>depends on actual <code>.easypar</code> statement</td>
</tr>
<tr>
<td>Value2</td>
<td></td>
</tr>
<tr>
<td>CustomOutput</td>
<td>cleared</td>
</tr>
</tbody>
</table>

‘Side-effect’ here means that these settings aren’t included as arguments in the `.easypar` statement itself, but are changed by the BC ‘behind your back’. However, many `.easypar` statements do have `Value1` and `Value2` as `arguments`, so strictly speaking their new values are then ‘main’ effects, rather than ‘side’ effects.
18.2 Motor

BCL syntax: `.motor Motor
Motor ∈ {off, on}`

If `Motor` is `on`, a physical fader (1-8) on the BCF automatically moves to the correct position if the fader’s `Value` changes; this may occur upon preset selection, value synchronization (see §14.9) or an incoming MIDI message. Note that you can always switch a fader off temporarily via the key-override feature (see §18.4).

Basically the motors are what you paid for when you bought the BCF, but these ‘magical’ moves may sometimes drive you crazy, so `.motor off` switches them off.

The BCF does maintain `Motor` for ‘fader’ 9 (the Foot Controller), but this setting is completely meaningless.
18.3 Override

BCL syntax: `.override Override

Override ∈ {move, pickup}`

For faders 1-8 Override is only meaningful if Motor is off. For ‘fader’ 9 (the Foot Controller), Override is always meaningful: the value of fader 9’s Motor is ignored.

If Override is move, the BCF’s display immediately outputs (and shows, if ShowValue is on) any new value corresponding with your manipulation of the physical fader (or foot controller), no matter which value has been sent to the BCF most recently via MIDI. So this setting may lead to jumps in value.

If Override is pickup, the behavior after the BC has received a value for the fader (or foot controller) via MIDI is different: the BCF then only starts outputting (and possibly showing) your new manually entered values after you have first moved the fader or foot controller to the position corresponding with the value received via MIDI. So if Override is pickup, it is highly advisable to set ShowValue to on: the BCF’s display then keeps showing the value to which you have to move until you have indeed reached that position: otherwise you may end up moving the fader or foot controller for ages without ever realizing that the BCF isn’t actually sending any corresponding MIDI messages because you haven’t passed the target position yet!
18.4 Key-override

BCL syntax: `.keyoverride OverrideButton`
To the BC: OverrideButton ∈ {off, 1 .. 64}
From the BC: OverrideButton ∈ {off, 1 .. 8, 33 .. 64}

If OverrideButton is off, no button affects the fader’s motor. Otherwise OverrideButton represents the number of the button that you can keep pressed to temporarily switch the motor off: this may prevent you and a sequencer program from getting into a tug-of-war. (Obviously this key-override method is only required if Motor has been set to on: after `.motor off` the motor is off anyway!)

The BC does accept any value representing a button in push encoder group 2, 3 or 4 (i.e. buttons 9 .. 32), but converts such a value to its corresponding button number in group 1, i.e. to ((OverrideButton − 1) mod 8) + 1.

Notes:

- One button can be selected for any number of faders, so one button could even temporarily switch off all motors simultaneously.

- Somewhat curiously, the BCF treats a `.keyoverride` statement for ‘fader’ 9 (the Foot Controller) in exactly the same way as for the actual faders 1-8. However, pressing the button associated with ‘fader’ 9 doesn’t achieve anything, because obviously external foot controllers don’t have BCF-controlled motors.
19 Memory presets

All settings made via $preset, $button, $encoder and $fader affect the settings of the current, ‘temporary’ preset: its settings are lost when the BC is switched off. However, there are also 32 memory presets, whose settings are retained during the time the BC is off.

The $recall and $store commands allow you to copy the temporary preset to a memory preset, and vice versa.
19.1 Recall

BCL syntax: $\texttt{recall MemoryPreset}$

$\texttt{MemoryPreset} \in \{1 .. 32\}$

$\texttt{recall}$ loads the temporary preset from the memory preset specified by $\texttt{MemoryPreset}$. There are also a number of side-effects: see §19.3.
19.2 Store

BCL syntax: $\texttt{store MemoryPreset}$

$\texttt{MemoryPreset} \in \{1 .. 32\}$

$\texttt{store}$ saves the temporary preset as the memory preset specified by $\texttt{MemoryPreset}$. 
19.3 Preset selection

There are four ways of selecting a memory preset:

1. Press ‘PRESET’ or ‘PRESET’ on the BC. See §23.5 for discussion.

2. Send a Program Change MIDI message to the BC. Specifically: $Cn MemoryPreset−1, where $n is equal to .rxch’s ReceiveChannel − 1.

3. Send Behringer’s SysEx command $22 to the BC, specifying the desired memory preset.

4. Send a BCL chain to the BC containing a recall statement: $recall MemoryPreset.

Any of these preset selection methods causes the BC to execute the following sequence of actions:

1. The BC copies the selected memory preset’s settings to the temporary preset. Note that this does not include the Value settings of the memory preset’s elements, since these Value settings aren’t actually part of the memory preset as such.

2. If the preset’s Request setting is on, the BC sends any MIDI bytes defined in the preset’s LEARN output (as defined via one or more .tx statements).

3. If the preset’s Snapshot setting is on, the BC sends any standard and/or custom output defined for the preset’s elements. See §23.4.
20 Unknown dot statements

Dot statements that do exist, but do not work in any of the known sections:

- `.rangeon` (argument specs unknown)
- `.xref` (argument specs unknown)

On both the BCF and the BCR these dot statements trigger error 13 (‘Setting not allowed in current section’) in any section ($\text{global}, \text{preset}, \text{button}, \text{encoder}, \text{fader}$). Note that these dot statements do not trigger error 1, so they are indeed defined. Very strange.
21 BCL Reply messages

Each BCL message you send to a BC causes this BC to reply with a message indicating the reception of the BCL message. If an error occurred, the reply message specifies the error type.

As far as I know, a BC cannot be instructed not to send these reply messages, although a computer program can of course ignore any or all of them.

21.1 MIDI format

<table>
<thead>
<tr>
<th>Item description</th>
<th>MIDI byte(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Exclusive</td>
<td>$F0</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>$00 $20 $32 (=Behringer)</td>
</tr>
<tr>
<td>Device ID</td>
<td>$0x 0..15 (=actual device’s ID – 1)</td>
</tr>
<tr>
<td>Model</td>
<td>$14 (=BCF2000) or $15 (=BCR2000)</td>
</tr>
<tr>
<td>Command</td>
<td>$21 (=BCL reply message)</td>
</tr>
<tr>
<td>Index_MSB</td>
<td>$0bbbbbb</td>
</tr>
<tr>
<td>Index_LSB</td>
<td>$0bbb</td>
</tr>
<tr>
<td>Error code</td>
<td>$0bbb (see §21.2)</td>
</tr>
<tr>
<td>End-Of-Exclusive</td>
<td>$F7</td>
</tr>
</tbody>
</table>

Index_MSB and Index_LSB together constitute the 14-bit index as it occurred in the BCL message to which the BC is responding.
21.2 Error codes

The following pages describe the error codes occurring in BCL Reply messages. (Actually error 7 is probably internal to the BC.)

Each error code is described in three ways:

1. BC-EDIT identifier:
The error code’s name occurring in BCLError.class, which is a file embedded in BC-EDIT’s bcedit.jar. In most cases ‘BC-EDIT identifier’ is simply a contracted version of ‘BC-EDIT message’ (see below), but in one case (error 7) it provides information ‘BC-EDIT message’ doesn’t.

2. BC-EDIT message:
The error code’s message occurring in BCLError.class.

3. Meaning:
The error code’s message in BC Manager. Hopefully these messages are more to-the-point than those used in BC-EDIT.
21.2.1 Error 0

BC-EDIT identifier: noerr
BC-EDIT message: OK

Meaning: No error.
21.2.2 Error 1

BC-EDIT identifier: unknowntoken
BC-EDIT message: unknown token

Meaning: Invalid identifier after ‘$’ or ‘.’.

Examples:
‘$’
‘.’
‘$haha’
‘.boo’
21.2.3 Error 2

BC-EDIT identifier: datawithouttoken
BC-EDIT message: data without token

Meaning: ‘$’ or ‘.’ expected.

Occurs for any non-empty line that doesn’t start with ‘$’, ‘.’ or ‘;’ (ignoring spaces, of course).

Example:
‘nonsense’

Note: this error number is also shown in the BC’s display upon the reception of faulty firmware data, as discussed in §6 under Firmware Reply.
21.2.4 Error 3

BC-EDIT identifier: argumentmissing
BC-EDIT message: argument missing

Meaning: MIDI output argument expected.

Examples:

‘.tx’ (without any MIDI bytes)
‘.tx $F0 cks-1’ (without argument for cks-1)
21.2.5 Error 4

BC-EDIT identifier: wrongdevice
BC-EDIT message: wrong device

Meaning: Invalid model.

This error causes termination of any current section. This error does not cause termination of any current block.

This error has precedence over error 5.

A note on terminology: BC-EDIT confusingly states that this error occurs if there is a ‘wrong device’, but in fact it concerns the Model specified in a $rev statement. Talking about a model here is also more consistent with the B-Control System Exclusive message format (described in §6), which includes both a ‘Device ID’ (which can be set manually for each BCF2000 and BCR2000) and a ‘Model’ (BCF2000 or BCR2000).

Examples:
BCR: ‘$rev r1’
BCR: ‘$rev F0’
BCR: ‘$rev F1’
21.2.6 Error 5

BC-EDIT identifier: wrongrevision
BC-EDIT message: wrong revision

Meaning: Unsupported revision.

This error causes termination of any current *section*.

Unlike error 4, this error also causes termination of any current *block* (i.e. it functions like $\texttt{end}$).

Examples:
BCR: `$\texttt{rev R0}$`
BCR: `$\texttt{rev R01}$`
21.2.7 Error 6

BC-EDIT identifier: missingrevision
BC-EDIT message: missing revision

Meaning: No block defined.

This error occurs for a section selector, command or dot statement outside a block (i.e. when no $rev has been sent in the same message chain, or when $end has been sent after the last $rev in the same chain).

This error has precedence over error 13.

This error also occurs if you send $rev R1 with a correct BCL message index of 0, followed by $preset with an incorrect BCL message index of 0 (should be 1).

Compare error 22.

Examples:
(no previous $rev) ⇒ '$global'
(no previous $rev) + '$global' ⇒ '.init'
(no previous $rev) + '$preset' ⇒ '.init'
21.2.8 Error 7

BC-EDIT identifier: internal
BC-EDIT message: none

Meaning: Unknown.

So far I have never encountered this error, and indeed the BC-EDIT identifier and the lack of a corresponding BC-EDIT message suggest that this is a BC-internal error message that is never sent in a BCL Reply message.
21.2.9 Error 8

BC-EDIT identifier: modemissing
BC-EDIT message: mode missing

Meaning: No section defined.

This error occurs for a dot statement outside any section (compare error 13).

This error has precedence over error 6. (So the BC first checks the presence of a section, then the presence of a block, and then the identity of the section!)

Examples:

BCR: ‘$rev R1’ → ‘.init’
(no previous $rev) → ‘.init’
21.2.10 Error 9

BC-EDIT identifier: baditemindex
BC-EDIT message: bad item index

Meaning: Element number out of range.

This error does not occur for `$recall 33` or `$store 33`, which generate error 11 (see §21.2.12).

Examples:
`$encoder 99`
BCR(!): `$fader 1`
21.2.11 Error 10

BC-EDIT identifier: notanumber
BC-EDIT message: not a number

Meaning: Invalid numerical argument.

This error occurs when a number is expected but e.g. a string is found.

Examples:

```
$encoder bad
.default on' (note: '.default off' is legal; so the occurrence of error 10 for on probably indicates that .default primarily expects a numerical value)
.default $F0 F7' ('F7' is the offender here)
```
21.2.12 Error 11

BC-EDIT identifier: valoutofrange
BC-EDIT message: value out of range

Meaning: Argument value out of range.

This error occurs when a numerical argument exceeds its allowed range.

Examples:
```
 .egroups 0
 .egroups 5
 $#recall 33$
 $#store 33$
```
21.2.13 Error 12

BC-EDIT identifier: invalidargument
BC-EDIT message: invalid argument

Meaning: Invalid text argument.

This error occurs when a text argument is expected, but an invalid string (or even a number) is found.

Examples:
`.showvalue dumbo`
`.showvalue 1`
`.easypar CC 1 1 0 127 boo`
`.name ''`
`.name NoStartingQuote`
`.name '1234567890123456789012345'"
21.2.14 Error 13

BC-EDIT identifier: invalidcommand
BC-EDIT message: invalid command

Meaning: Setting not allowed in current section.

This error occurs for a dot statement in the wrong section (compare error 8).

This error has precedence over error 14.

Examples:

`$global`
`.init`

`$preset`
`.rxch off`
21.2.15 Error 14

BC-EDIT identifier: wrongnumberofargs
BC-EDIT message: wrong number of arguments

Meaning: Invalid number of arguments (too few or too many).

This error has precedence over errors 4, 5, 10, 11 and 12.

Examples:
BCR: `$rev R1 extra`
`.minmax 0`
`.minmax 0 100 1`
`.name`
`.name 'valid' invalid`
21.2.16 Error 15

BC-EDIT identifier: toomuchdata
BC-EDIT message: too much data

Meaning: Too much MIDI output data.

Context: a .tx statement that overflows the LEARN or custom output data area of 127 positions. (See § 13.7 and § 14.6 for details.) Note that this limit does not concern the length of the received BCL line (or message).
21.2.17 Error 16

BC-EDIT identifier: alreadydefined
BC-EDIT message: already defined

Meaning: unknown. (Never encountered yet.)
21.2.18 Error 17

BC-EDIT identifier: presetmissing
BC-EDIT message: preset missing

Meaning: unknown. (Never encountered yet.)
21.2.19 Error 18

BC-EDIT identifier: presettooocomplex
BC-EDIT message: preset too complex

Meaning: Preset too complex.

The BC displays this error when you manually try to store the temporary preset to a memory preset if the temporary preset contains too many definitions. (‘Manually’: i.e. by pressing STORE, optionally selecting a memory preset number, and pressing STORE again.) If this error occurs, the existing memory preset data is not affected in any way.

Curiously the BC does not return error 18 upon reception of a \$store command via MIDI if the temporary preset is too complicated; I think this is a bug. However, as in case of a manual invalid attempt, the existing memory preset data is not affected in any way.

The total maximum definition size is 4956 bytes for the BCF2000 and 4344 bytes for the BCR2000. The following items count toward this total:

- The preset’s LEARN output definition.
- The elements’ standard output definitions.
- The elements’ custom output definitions.
21.2.20 Error 19

BC-EDIT identifier: wrongpreset
BC-EDIT message: wrong preset

Meaning: unknown. (Never encountered yet.)
21.2.21 Error 20

BC-EDIT identifier: presettoonew
BC-EDIT message: preset to onew [sic — this may be a typo in BCLError.class: it may have been intended as ‘preset too new’, although that doesn’t really make sense either]

Meaning: unknown. (Never encountered yet.)
21.2.22 Error 21

BC-EDIT identifier: presetcheck
BC-EDIT message: preset check

Meaning: unknown. (Never encountered yet.)
21.2.23 Error 22

BC-EDIT identifier: sequence
BC-EDIT message: sequence error

Meaning: Invalid message index.

Compare error 6.
21.2.24 Error 23

BC-EDIT identifier: wrongcontext
BC-EDIT message: wrong context

Meaning: unknown. (Never encountered yet.)
22  Startup functions

By keeping one or two specific buttons pressed while switching the POWER button on, you can execute the following functions:

<table>
<thead>
<tr>
<th>Model</th>
<th>Button(s) held</th>
<th>Function</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCF2000/BCR2000</td>
<td>STORE + LEARN</td>
<td>Enter bootloader mode</td>
<td>LOAd</td>
</tr>
<tr>
<td></td>
<td>STORE + EXIT</td>
<td>Initialize temporary preset</td>
<td>Init</td>
</tr>
<tr>
<td>BCF2000</td>
<td>33 (top row, column 1)</td>
<td>Select standard B-Control mode</td>
<td>bC</td>
</tr>
<tr>
<td></td>
<td>34 (top row, column 2)</td>
<td>Select Mackie Control Mapping for Cubase SX and Nuendo</td>
<td>MC C</td>
</tr>
<tr>
<td></td>
<td>35 (top row, column 3)</td>
<td>Select Logic Control Mapping for Logic Audio</td>
<td>LC</td>
</tr>
<tr>
<td></td>
<td>36 (top row, column 4)</td>
<td>Select Mackie Control Mapping for Sonar 3</td>
<td>MCSo</td>
</tr>
<tr>
<td></td>
<td>37 (top row, column 5)</td>
<td>Select Mackie Baby HUI Mapping</td>
<td>bhuI</td>
</tr>
</tbody>
</table>

In all cases except bootloader mode, the BC’s display first briefly shows the firmware version number before the message specified in the Display column in the above table.

Note that initializing the temporary preset only works in standard B-Control mode. Obviously the BCR is always in standard B-Control mode, since it has no emulation modes. However, on the BCF this initialization doesn’t work if an emulation mode was active when the BC was last powered off.

Actually you can switch from a BCF emulation mode to standard B-Control mode and initialize the temporary preset in one go, by holding button 33 and STORE + EXIT while pressing the POWER button. Unless you’re an octopus, this requires some twister-like hand positioning: try pressing the POWER button with your right thumb, while holding button 33 with your left hand and STORE and EXIT with the little and ring fingers respectively of your right hand.

Further details on these startup functions follow on the following pages.
22.1 Bootloader mode

In bootloader mode the BC has very limited functionality. All buttons, encoders and faders are dead, and standard MIDI output B and the USB connection don’t work. The BC only responds to two types of SysEx messages (cf. §6) sent to its standard MIDI input:

1. Request Identity:
   The BC responds with a Send Identity message to its standard MIDI output A. The returned identity string is ‘BCF2000 BOOTLOADER 1.0’ and ‘BCR2000 BOOTLOADER 1.0’ for the BCF and BCR respectively.

2. Send Firmware:
   After every sixteenth Send Firmware message, the BC responds with a Firmware Reply message to its standard MIDI output A.

Note that Device ID is $7F (=‘any’) in any Send Identity or Firmware Reply message from a BC in bootloader mode. Bootloader mode doesn’t use the ‘actual’ Device ID (1-16) set via Global Setup, simply because it doesn’t have access to it.

To return from bootloader mode to standard B-Control mode (or to the previously selected BCF emulation mode), you can simply switch off the BC, then on again. Upon restart, you don’t have to hold any buttons, because unlike the BCF emulation modes, bootloader mode is not ‘permanent’.

Note that the BC enters bootloader mode automatically (i.e. without the user pressing any keys during power-on) if the BC’s current firmware is somehow invalid (e.g. because the previous firmware upgrade procedure was interrupted). The display then shows ‘noOS’, which stands for ‘no Operating System’. You can only remedy this situation by sending valid firmware to the BC’s standard MIDI input.
22.2 Initialization of temporary preset

Provided that the BC is in standard B-Control mode, the Init function starts up the BC normally, except that the temporary preset is ‘empty’, rather than loaded with the settings from the memory preset selected via .startup StartupPreset (cf. §12.2). It’s as if you send the following BCL chain to the BC:

```bcl
$rev F1 ; this is for the BCF: the BCR of course needs R1 here
$preset
  .name 'init
  .snapshot off
  .request off
  .egroups 4
  .fkeys on
  .lock off
  .init
$end
```

This initialization is useful if you want to set up a preset from scratch. It also avoids any undesired immediate LEARN output (as defined in the selected memory preset) to the receiving MIDI device(s).
22.3 The BCF2000 emulation modes

The BCF2000’s emulation modes (or ‘mappings’) provide partial or complete emulations of other MIDI control devices. See Behringer’s BCF2000_Emulation_modes.pdf document for details.

From among the System Exclusive commands available in standard B-Control mode, the emulation modes recognize only Request Identity and Send Firmware, and respond by Send Identity and Firmware Reply messages respectively. So in this respect they behave like bootloader mode. However, their actual response messages are exactly the same as in the standard B-Control mode, not as in bootloader mode: these messages contain the ‘actual’ Device ID (1-16), and the Send Identity message contains the standard identity string, i.e. ‘BCF2000 1.10’ or ‘BCR2000 1.10’.

Global Setup edit mode is available immediately after power-on if you keep the emulation mode’s button (34, 35, 36 or 37) pressed until ‘EG’ appears in the BC’s display. However, some of the global settings available in standard B-Control mode are unavailable now:

<table>
<thead>
<tr>
<th>Push encoder</th>
<th>Setting/function</th>
<th>Available in emulation mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MidiMode</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>ReceiveChannel</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>FootSwitch</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>StartupPreset</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>DeviceID</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>SysEx dump</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>DeadTime</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>TransmissionInterval</td>
<td>yes</td>
</tr>
</tbody>
</table>
22.3.1 Emulation mode identity SysEx messages

Six types of emulation mode identity message pertain to the BCF when it is in MC C, LC or MCSo mode (but not in standard B-Control or bhul mode). These are three types of Mackie identity request message (one ‘long’ one and two ‘short’ ones), each with a corresponding identity reply message.

Note: I haven’t been able to lay my hands on any Mackie SysEx specs documents, so I haven’t been able to establish the nature of the conceptual differences between these message types; hence, the descriptions below are rather tentative. Also note that there are other Mackie SysEx messages that can be sent to the BCF in certain emulation modes, but I haven’t fully investigated these yet.

1. Request Emulation Mode Long Identity

Format:
F0 00 00 66 m 00 F7

- Bytes two to four ($00 $00 $66) are the Mackie Manufacturer ID.
- The fifth byte (m) must be $10 if the targeted BCF is in LC mode and $14 if it’s in MC C or MCSo mode. (I have no idea why no distinction is made between MC C and MCSo modes.)
- The sixth byte ($00) is a command byte meaning ‘Request Emulation Mode Long Identity’.

When the BCF receives this message, it replies with a corresponding Send Emulation Mode Long Identity message (see below), provided that it is in the mode indicated by m, otherwise it does not respond in any way.

2. Send Emulation Mode Long Identity

Format:
F0 00 00 66 m 01 54 5A 42 47 2D n 2D 42 43 46 n F7

- Bytes two to four ($00 $00 $66) are the Mackie Manufacturer ID.
- Byte five (m) indicates the BCF’s emulation mode: $10 = LC, $14 = MC C or MCSo.
- The sixth byte ($01) is a command byte meaning ‘Send Emulation Mode Long Identity’.
- The rest of the message data is a text string following the pattern ‘TZBG-n-BCF’n’:
  - ‘TZ’ are the initials of the Behringer developer of the BCF2000 and BCR2000.
  - ‘BG’ probably stands for ‘Behringer’.
  - Each of the two ns is actually a character from ‘A’ ($41) to ‘P’ ($50), calculated as the BCF’s DeviceID + $40. (So ‘A’ means that DeviceID is 1 and ‘P’ means that DeviceID is 16.) As far as I know, the values of the two ns are always identical; I don’t know why the Device ID is thus represented twice in this text string.
  - ‘BCF’: very obvious...

The BCF sends this message in two situations:
- Upon power-on, provided it is in MC C, LC or MCSo mode.
- Upon reception of a Request Emulation Mode Long Identity message (see above), provided that the request message contains the correct value for m.

Note that the BCF only replies if the request message’s m matches the BCF’s actual emulation mode. So discovering the BCF’s current emulation mode via these messages is a complicated affair: you must send both the MC C/MCSo and the LC mode identity requests, and (since there are no emulation mode identity messages whatsoever in B-Control or ‘bhul’ mode)
you must also send a BC message (as described in §6) to which the BCF only responds in B-Control mode (e.g. a temporary preset name request).

3. Request Emulation Mode Short Identity

Format:
F0 00 00 66 m 02 F7

- Bytes two to four ($00 $00 $66) are the Mackie Manufacturer ID.
- Byte five indicates the BCF’s emulation mode: $10 = LC, $14 = MC C or MCSo.
- The sixth byte ($02) is a command byte meaning ‘Request Emulation Mode Short Identity’.

When the BCF receives this message, it replies with a corresponding Send Emulation Mode Short Identity message (see below), provided that it is in the mode indicated by m, otherwise it does not respond in any way.

4. Send Emulation Mode Short Identity

Format:
F0 00 00 66 m 03 54 5A 42 47 2D n 2D F7

- Bytes two to four ($00 $00 $66) are the Mackie Manufacturer ID.
- Byte five (m) indicates the BCF’s emulation mode: $10 = LC, $14 = MC C or MCSo.
- The sixth byte ($03) is a command byte meaning ‘Send Emulation Mode Short Identity’.
- The rest of the message data is a text string following the pattern ‘TZBG-n-’. This is a shortened version of the string in Send Emulation Mode Long Identity message (see above).

The BCF sends this message upon reception of a Request Emulation Mode Short Identity message (see above), provided that the request message contains the correct value for m.

5. Request Emulation Mode Short Identity (alternative)

Format:
F0 00 00 66 m 1A F7

- The sixth byte ($1A) is a command byte meaning ‘Request Emulation Mode Short Identity (alternative)’.
- All other specs are identical to those of the message type discussed under (3) above.

6. Send Emulation Mode Short Identity (alternative)

Format:
F0 00 00 66 m 1B 54 5A 42 47 2D n 2D F7

- The sixth byte ($1B) is a command byte meaning ‘Request Emulation Mode Short Identity (alternative)’.
- All other specs are identical to those of the message type discussed under (4) above.
The BCF sends this message upon reception of the message type discussed under (5) above, provided that the request message contains the correct value for $m$. 
23 Functions in standard B-Control mode

The table below specifies a number of functions that can be executed when the BC is in standard B-Control mode:

<table>
<thead>
<tr>
<th>Function</th>
<th>Action 1</th>
<th>Action 2</th>
<th>Section in B-Control manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy Encoder Group</td>
<td>Press STORE</td>
<td>Press encoder group</td>
<td>4.2.3</td>
</tr>
<tr>
<td>Store Preset</td>
<td></td>
<td>Select preset number</td>
<td>4.2.2</td>
</tr>
<tr>
<td>LEARN</td>
<td>Hold LEARN</td>
<td>Move element</td>
<td>4.3.1</td>
</tr>
<tr>
<td>Edit Global Setup</td>
<td></td>
<td>Press STORE</td>
<td>4.5</td>
</tr>
<tr>
<td>Data Request</td>
<td>Hold EDIT</td>
<td>Press LEARN</td>
<td>4.6 (3rd item)</td>
</tr>
<tr>
<td>Panic Reset</td>
<td></td>
<td>Press EXIT</td>
<td>4.6 (2nd item)</td>
</tr>
<tr>
<td>Snapshot Send</td>
<td></td>
<td>Press ‘PRESET’</td>
<td>4.6 (4th item)</td>
</tr>
<tr>
<td>Single Preset Dump</td>
<td></td>
<td>Press ‘PRESET •’</td>
<td>4.6 (5th item)</td>
</tr>
<tr>
<td>Edit Element</td>
<td></td>
<td>Move element</td>
<td>4.3.2 and 4.6 (6th item)*</td>
</tr>
<tr>
<td>Temporary Local Off</td>
<td>Hold EXIT</td>
<td>Move element</td>
<td>4.6 (1st item)</td>
</tr>
<tr>
<td>Select Preset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Press ‘• PRESET’</td>
<td>–</td>
<td>4.2.1</td>
</tr>
<tr>
<td></td>
<td>Hold ‘• PRESET’</td>
<td>Rotate encoder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Press ‘PRESET •’</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hold ‘PRESET •’</td>
<td>Rotate encoder</td>
<td></td>
</tr>
</tbody>
</table>

‘Press’ here means ‘push and release’, and of course ‘Hold’ means ‘push, but don’t release yet’.

Note that the actions mentioned in this table are just meant as quick reminders: in some cases subsequent actions are required to complete the operation. Refer to the pertinent sections in the B-Control manual for the full procedures.

The next pages look at some of these functions in more detail:

---

* Section 4.6 of the B-Control manual specifically discusses the edit procedure for ‘Motor Off Function’: you can select a key-override button by pressing that button. However, this editing facility actually takes place within the context of fader editing, which is why this table doesn’t mention ‘Motor Off Function’.
23.1 LEARN

The LEARN function temporarily puts the BC in learning mode. This enables you to define a particular element’s custom output: the BC sets the element’s CustomOutput to the first MIDI message you send to the BC while it is in learning mode. Upon reception of this MIDI message, the BC automatically terminates learning mode and informs you whether the received MIDI message is ‘GOOd’ or ‘bAd’.

The BC’s LEARN function is limited in several ways:

- Since only the first received MIDI message is entered into CustomOutput, you can’t create CustomOutput consisting of two or more messages (either in a single .tx statement or spread out across multiple .tx statements).
- You can’t define special identifiers.

So if you want to use any of these more advanced features, you cannot use the BC’s LEARN function; instead, you must send a BCL chain containing the element’s CustomOutput definition to the BC. See §14.6 for details.
23.2 Data Request

The Data Request function makes the BC send the MIDI bytes defined via the temporary preset’s LearnOutput. See §13.7.
23.3 Panic Reset

The B-Control manual is vague about what the Panic Reset function actually does:

The German version says ‘mit dieser Funktion werden die wichtigsten MIDI-Daten zurückgesetzt’. This literally means ‘with this function the most important MIDI data are reset’.

By contrast, the corresponding sentence in the English manual reads ‘this function resets the most important MIDI data to their factory settings’. The phrase ‘to their factory settings’ throws a rather different light on the matter: it suggests that Panic Reset affects the BC’s global setup (operating mode, receive channel etc.). However, this is definitely not the case: Panic Reset doesn’t affect the BC itself in any way — not even its temporary preset, let alone its global setup.

Actually, Panic Reset causes the BC to output the following sequence of MIDI messages:

1. For each MIDI channel $c$ from $0$ to $F$:
   1. $B_c$ $7B$ $00$ All Notes Off
   2. $B_c$ $78$ $00$ All Sounds Off
   3. $B_c$ $40$ $00$ Damper Pedal $\Rightarrow$ Off
   4. $B_c$ $01$ $00$ Modulation $\Rightarrow$ 0

2. For each MIDI channel $c$ from $0$ to $F$:
   $E_c$ $00$ $40$ Pitch Bend $\Rightarrow$ 0

So the BC first outputs four messages starting with $B0$, then four with $B1$ etc., and finally the sixteen $Ec$ messages.

Note that although Panic Reset sends messages to reset the receiving MIDI device’s Damper Pedal and Modulation controllers, any elements on the BC using these controllers are not reset. Obviously this can lead to mismatches between the BC and the receiving device.
23.4 Snapshot Send

The Snapshot Send function makes the BC send any standard and/or custom output defined for the elements of the temporary preset. However, if an element’s *Value* is **off**, no output whatsoever is sent for that element, not even any custom output messages that don’t even refer to *Value*.

The order used is as follows:
1. Only on the BCF: the active faders.
2. The active encoders.
3. The active buttons.

Note that the BC ‘stupidly’ sends multiple values for the same MIDI entity (e.g. a particular note number on a particular MIDI channel) if two or more elements use it. These values can even be *different*, namely if the *Default* settings of the elements differ and the user hasn’t physically manipulated one of the elements involved and no MIDI message mentioning the MIDI entity has been sent to the BC (cf. §14.9).
23.5 Select Preset

Pushing the ‘PRESET’ or ‘PRESET’ button down selects the previous, respectively next preset. If you then keep this button pushed down, you can select any preset quickly by rotating any encoder knob. See §19.3 for other methods of preset selection and discussion of its effects.