Definition of the RCM-BFin GCC v2.0 Control Protocol (Blackfin Version) - as of December 2017

All commands from the host to the RCM-BFin robot are comprised of ASCII characters or ASCII followed by 8-binary or ASCII decimal characters. All commands receive an acknowledgment from the robot to the host, which is either a '#' character followed by the command, or '##' for variable length responses. Variable length commands which don't specify a return size append a CR + newline ('\r\n') to their response.

Note that many of these commands can be executed via a terminal program with TCP / telnet capability. Those commands that require binary arguments will not work this way unles your telnet program can generate arbitrary binary bytes. For example, you can connect using the 'netcat' command via

```
nc robot-ip 10001
```

When the robot first powers up, it will dump the 'V' Version command results ("##Version ...\r\n") so you can see what version of firmware is running. There is typically a 2 second delay after startup before the robot can accept any commands while the camera and other sensors are initialized - you should see the yellow LED's flash when the processor reboots. After startup, just to test that there is 2-way communication, send an 'V' to access the firmware version string. The only command that will produce strange results is the 'I' IMJ command, which grabs a JPEG frame - this will flood the screen with binary characters. However, other commands may return a limited number of binary (non-printable ASCII) bytes.

The firmware has a built-in C interpreter called "picoC". The 'Q' command will execute the C program that has been stored in the robot's flash buffer and the '!' command will run C interactively, terminated with an ESC. The flash buffer can be set by the 'zr' command which transfers the contents of the user flash segment to the flash buffer, or via the 'X' command which transfers a file from the host via XMODEM protocol. Before executing a program, the contents of the flash buffer can be examined using 'zd'. When the program finishes (assuming the C program isn't running an infinite loop), control returns to the C interpreter, unless the exit() function is called in the C program, which similar to ESC returns control to the regular RCM-BFin command processing loop. If the autorun(x) function is placed at the beginning of a program which is stored in flash sector #4, the robot will start running the stored C program unless ESC character is received within 'x' seconds of startup.

Command	Response	Description
Core Robot Commands		
'7'	'#7'	note that keypad commands ('1' - '9') don't become active until an 'Mxxx' motor control command has been received robot drift left
'8'	'#8'	robot drive forward
'9'	'#9'	robot drift right
'4'	'#4'	robot drive left
'5'	'#5'	robot stop
'6'	'#6'	robot drive right
'1'	'#1'	robot back left
'2'	'#2'	robot drive back
'3'	'#3'	robot back right
'0'	'#0'	robot rotate left 20-deg
'.'	'#.'	robot rotate right 20-deg
'+'	'#+'	increase motor/servo level
'-'	'#-'	decrease motor/servo level
'<'	'#<'	trim motor balance toward left
'>'	'#>'	trim motor balance toward right
'a'	'#a'	set capture resolution to 160x120
'b'	'#b'	set capture resolution to 320x240

'c'	'#c'	set capture resolution to 640x480
'd' or 'A'	'#A'	set capture resolution to 1280x1024
'D'	'##D'	checks MN13812 battery level detect circuit on SVS returns: ##D - battery voltage okay ##D - low battery voltage detected
'E'		launches flash buffer line editor - (T)op (B)ottom (P)reviouse (N)ext line (L)ist (I)nsert until ESC (D)elete (H)elp (X)exit
'Fab'	'#F'	Enables Failsafe mode for motor control 'ab' parameters sent as 8-bit binary a = left motor/servo failsafe level, b = right motor/servo failsafe level. sets motor/servo levels for 'M' and 'S' commands in case no command is received via the radio link within 2 seconds.
'f'	'#f'	disables Failsafe mode
'G'	HTML stream	HTTP GET command - parses HTTP request, e.g. GET /index.html HTTP/1.1 - recognizes files /00.html .. /09.html, stored respectively in flash sectors 10/11, 12/13 .. 28/29 - /index.html == /00.html - will replace "$$camera$$" with base64 live captured JPEG - use 'zBxx' to store in sector pair - zb10 stores 128kB from flash buffer to sectors 10/11
'I'	'##IMJxs0s1s2s3....'	grab JPEG compressed video frame x = frame size in pixels: 1 = 80x64, 3 = 160x120, 5 = 320x240, 7 = 640x480, 9 = 1280x1024 s0s1s2s3=frame size in bytes (s0 * 256^0 + s1 * 256^1 + s2 * 256^2 + s3 * 256^3) .... = full JPEG frame Note that sometimes the 'I' command returns nothing if the robot camera is busy, so the 'I' command should be called as many times as needed until a frame is returned
'irab'	'##irhh cc'	I2C register read ('ab' parameters sent as 8-bit binary) 'a' is device id/address (one binary byte), 'b' is register (one binary byte), 'hh' is the ASCII HEX representation of 'a' (always two characters) 'cc' is 8-bit return value from register displayed as decimal value, which could be one, two or three characters long, and there is a pair of bytes at the end. The can be used to know that the variable length 'cc' is complete.
'&irddab'	'##&irddhh cc'	I2C register read ('a' and 'b' parameters sent as 8-bit binary, 'dd' sent as 16-bit binary) 'a' is device id/address (one binary byte), 'b' is register (one binary byte), 'dd' is a 16-bit number that the BFin will remember, and send back in the reply, 'hh' is the ASCII HEX representation of 'a' (always two characters) 'cc' is 8-bit return value from register displayed as decimal value, which could be one, two or three characters long, and there is a pair of bytes at the end. The can be used to know that the variable length 'cc' is complete.
'iRab'	'##iRhh cc'	I2C register read ('ab' parameters sent as 8-bit binary) 'a' is device id, 'b' is register, 'hh' is the ASCII HEX represnetation of 'a' (always two characters), 'cc' is 16-bit return value from register displayed as decimal value, which could be from one to five characters long, and there is a

		pair of bytes at the end. The can be used to know that the variable length 'cc' is complete.
'iMabc'	'##iMaa xx xx ... xx'	I2C multiple register read ('abc' parameters sent as 8-bit binary) a is device id, b is register, c is count, xx is 8-bit return values from register displayed as decimal value
'iwabc'	'##iwaa'	I2C register write ('abc' parameters sent as 8-bit binary) a is device id, b is register, c is value written to register
'iWabcd'	'##iWaa'	multi-byte I2C register write ('abcd' parameters sent as 8-bit binary) a is device id, b is first byte, c is second byte, d is third byte
'idabcef'	'##idaa'	I2C dual register write ('abcde' parameters sent as 8-bit binary) a is device id, b is register 1, c is value written to register 1, d is register 2, e is value written to register 2
'l'	'#l'	turn on lasers
'L'	'#L'	turn off lasers
'Mabc'	'#M'	direct motor control 'abc' parameters sent as 8-bit binary a=left speed, b=right speed, c=duration*10milliseconds speeds are 2's complement 8-bit binary values - 0x00 through 0x7F is forward, 0xFF through 0x81 is reverse, e.g. the decimal equivalent of the 4-byte sequence 0x4D 0x32 0xCE 0x14 = 'M' 50 -50 20 (rotate right for 200ms) duration of 00 is infinite, e.g. the 4-byte sequence 0x4D 0x32 0x32 0x00 = M 50 50 00 (drive forward at 50% indefinitely)
'mabc'	'#m'	to employ direct PWM motor control of 2nd bank of timers (TMR6 and TMR7) - same format as 'M' command
'o'	'#o'	enable caption overlay
'O'	'#O'	disable caption overlay
'p'	'##ping xxxx xxxx xxxx xxxx\r\n'	ping ultrasonic ranging modules attached to pins 27, 28, 29, 30 with trigger on pin 18 - tested with Maxbotics EZ0 and EZ1 modules. xxxx return value is range in inches * 100 (2500 = 25 inches)
'qx'	'##quality x\r\n'	sets JPEG quality between 1-8 ('x' is an ASCII decimal character). 1 is highest, 8 is lowest
'R'	'##Range(cm) = xxx'	measure range to nearest obstacle using laser pointers
'r'	'##Range(cm)'	same as 'R', but with lots of diagnostic output
'Sab'	'#S'	direct servo control (TMR2 and TMR3) 'ab' parameters sent as 8-bit binary a=left servo setting (0x00-0x64), b=right servo setting (0x00-0x64) servo settings are 8-bit binary values, representing timing pulse widths ranging from 1ms to 2ms. 0x00 corresponds to a 1ms pulse, 0x64 corresponds to a 2ms pulse, and 0x32 is midrange with a 1.5ms pulse
'sab'	'#s'	direct servo control of 2nd bank of servos (TMR6 and TMR7) 'ab' parameters sent as 8-bit binary a=left servo setting (0x00-0x64), b=right servo setting (0x00-0x64) servo settings are 8-bit binary values, representing timing pulse widths ranging from 1ms to 2ms. 0x00 corresponds to a 1ms pulse, 0x64 corresponds to a 2ms pulse, and 0x32 is midrange with a 1.5ms pulse
't'	'##time - millisecs: xxxx\r\n'	outputs time in milliseconds since reset

'Tx'	'#T'	changes threshold in 'g2' edge detection - T4 is default, range is T1 to T9
'V'	'##Version ...\r\n'	read firmware version info response is terminated by newline character
0x1F (binary byte)	'##Version ...\r\n'	Returns version string, but does so without a surrounding packet, even if packet mode is turned on.
'X'	'#Xmodem transfer count: bytes'	Xmodem-1K file transfer - receive file via xmodem protocol - store in flash buffer
'y'	'#y'	flip video capture (for use with upside-down camera)
'Y'	'#Y'	restore video capture to normal orientation
'zAxx'	'##zA - read 131072 bytes\r\n'	flash memory large read - read 128kb from specified flash sector pair xx 02/03 ... 62/63 to flash buffer (e.g. read C program from flash sector before running C interpreter). Sectors 00 and 01 are off-limits.
'zBxx'	'##zB - wrote 131072 bytes\r\n'	flash memory large write - write 128kb from flash buffer to specified flash sector buffer pair 02/03 ... 62/63 (sectors 00 and 01 are off-limits)
'zc'	'##zclear\r\n'	clear contents of flash buffer
'zC'	'##zCRC xxxx\r\n'	compute crc16_ccitt for flash buffer
'zd'	'##zd...\r\n'	flash buffer dump - dump contents of flash memory buffer to console
'zr'	'##zr\r\n'	flash memory read - read 65kb from user flash sector to flash buffer (e.g. read C program from flash sector before running C interpreter)
'zRxx'	'##zRead\r\n'	flash memory read - read 65kb from specified flash sector xx (02 - 63) to flash buffer (e.g. read C program from flash sector before running C interpreter). Sectors 00 and 01 are off-limits.
'zw'	'##zw\r\n'	flash memory write - write 65kb from flash buffer to user flash sector
'zWxx'	'##zWxx\r\n'	flash memory write - write 65kb from flash buffer to specified flash sector 02-63 (sectors 00 and 01 are off-limits)
'zZ'	'##zZ\r\n'	flash memory boot sector update - writes contents of flash buffer to boot sectors of flash memory - used to replace u-boot.ldr or RCM-BFin.ldr - checks first that a valid LDR format image is in the flash buffer
Special Commands		
'$!'		reset Blackfin
'$E'		read optional wheel encoders on GPIO-H14 and H15
'$ex'		x = motor # (1-4); read cummulative pulse count from wheel encoder on SRV-4WD
'$g'		parse GPS input
'$R'		SVS command - configures slave Blackfin to receive SPI transfer to flash buffer
'$X'		SVS command - configures master Blackfin to transfer contents of flash buffer via SPI
'$Axx'	'##$Axx yyyy\r\n'	read AD7998 A/D channel xx (01-08, 11-18 or 21-28)
'$C'	'##$C xxx\r\n'	read HMC6352 compass
'$c'	'##c heading=344 x=-505 y=-110 z=447 xmin=-1032 xmax=-228 ymin=-970 ymax=-89\r\n'	read HMC5843 compass
'$Ta'	'##$Tx yyyy\r\n'	read LIS3LV02DQ tilt sensor channel a (1 = x axis, 2 = y axis, 3 = z axis)
Vision Commands		

all parameters are sent as ASCII decimal characters ('0' - '9')		
'g0'	'##g0'	grab reference frame and enable frame differencing
'g1'	'##g1'	enable color segmentation
'g2'	'##g2'	enable edge detection (threshold changed with 'T' command)
'g3'	'##g3'	enable horizon detection (threshold changed with 'T' command)
'g4'	'##g4'	enable obstacle detection (threshold changed with 'T' command)
'g5'	'##g5'	enable stereo processing (only works with SVS that has GPIO-H8 connection between processors)
'g6x'	'##g6 bin# x'	graphically overlay blob search results for color bin# x (e.g. 'g63' displays blobs matching color bin 3)
'g_'	'#g_'	disable frame differencing / color segmentation / edge detection. the '_' in 'g_' could be any character other than 0, 1, 2, 3, 4
'vax'	'##vax\r\n'	the 'va' command enables/disables automatic gain, white balance and exposure camera functions (default x=7) x=4 -> AGC enable x=2 -> AWB enable x=1 -> AEC enable x=7 -> AGC+AWB+AEC on x=0 -> AGC+AWB+AEC off
'vbc'	'##vbc dd\r\n ssss x1 x2 y1 y2\r\n ....'	the 'vb' command searches for blobs matching the colors in color bin #c, indicates the number of blobs found as dd, and returns a count of matching pixels in the blob, along with coordinates of an x1, x2, y1, y2 rectangular region containing the matching pixels. up to 16 blobs can be returned, and the blobs are sent in order of pixel count, though blobs smaller than MIN_BLOB_SIZE (currently set to 5 pixels) aren't shown.
'vccy1y2u1u2v1v2'	'##vcc\r\n'	the 'vc' command directly sets the contents of color bin #c. this command will return string with 'vc' followed by the color bin number. for example, we could save a set of colors to color bin #3 corresponding to measurements taken at another time, such as the above mentioned orange golf ball color measurement, using 'vc3127176086111154200'. we could then confirm that the colors were properly stored by issuing the command 'vr3' to retrieve the contents of color bin #3.
'vfxxx1xxx2yyy1yyy2'	'##vf xxxx\r\n'	the 'vf' counts the number of pixels matching color bin #c in the range of x1, x2, y1, y2 e.g. 'vf10100020001500220' searches for color bin #1 pixels in the x range from 100-200 and y range from 150-220
'vh'	'##vhist y u v\r\n'	computes and lists the distribution of Y, U and V pixels over the entire range of possible values, divided into bins of 0-3, 4-7, 8-11, ... 248-251, 252-255
'vm'	'##vmean yy uu vv\r\n'	computes mean values for Y, U and V over the entire image.
'vpxxxxyyyy'	'##vp yyy uuu vvv\r\n'	the 'vp' command samples a single pixel defined by coordinates xxxx (column 0000-0159, 0000-0319, 0000-0639, 0000-1279 depending on resolution) and yy (row 0000-0127, 0000-0255, 0000-0511, 0000-1023, where 0000 is top of image). 'vp01600128' will sample a pixel in the middle of the image at 320x256 resolution, 'vp01600000' will sample a pixel in the middle of the top row, etc...
'vrc'	'##vrc y1 y2 u1 u2 v1 v2\r\n'	the 'vr' command retrieves the stored color info from color bin #c. this command will return string with 'vr' followed by the color bin number, followed by y1=Ymin, y2=Ymax, u1=Umin, u2=Umax, v1=Vmin, v2=Vmax. in the above example where colors for an orange golf ball were captured using the 'vg' command for color bin #0, issuing a 'vr0' command will return the colors stored in color bin #0 - e.g. '##vr0 127 176 86 111 154 200\r\n'.
'vsx'	'##vscan = pix xxxx xxxx xxxx xxxx ...	vs scans for edge pixels in x (1-9) columns using edge_thresh set by 'T' or 'vt' command. displays total number of edge pixels found and distance from

	xxxx\r\n'	bottom to first edge pixel in each column
'vtxxx'	'##vthresh xxxx\r\n'	vt sets the edge_thresh global variable from 0000-9999 for edge detection (default is 3200). equivalent to 'T' console function but with more precision
'vzx'	'##vzero\r\n'	vz0 zeros out all of the color bins, vz1 / vz2 / vz3 / vz4 segments colors into various color spaces which can be used by enabling 'g1' color segmentation function ('G' turns it off)
Neural Network Commands		
all parameters are sent as ASCII hex characters ('0' - 'f')		
'np'		store a new pattern
'nd'		display a stored pattern
'ni'		initialize the network with random weights
'nt'		train the network from stored patterns
'nx'		test the network with sample pattern
'ng'		grab a pattern using blob located by "vb"
'nb'		match pattern against specific blob from "vb"
New commands for GCC v2.0 RCM-BFin firmware version		
'ibxxyyz'	'##ib xx yy z v\r\n'	**Read One Bit From I2C Register** **Command:**'ibxxyyz' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register z is a one character ASCII hexadecimal value from 0 to 7 representing the bit to be read example: "ib327F3" would request a read of the 3rd bit of register 0x7F from I2C address 0x32 **Response:**'##ib xx yy z v\r\n' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register z is a one character ASCII hexadecimal value from 0 to 7 representing the bit to be read v is a one character ASCII '0' or '1' representing the value of the bit that just got read example: "##ib 32 7F 3 0\n\r" would be indicating a 0 from bit 3 of register 0x7F of address 0x32
'icxxyyz'	'##ic xx yy z v\r\n'	**Read One Bit From I2C Register** Identical to 'ibxxyyz' except that this command uses a repeated start I2C transaction : https://www.i2c-bus.org/repeated-start-condition/ **Command:**'icxxyyz' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register z is a one character ASCII hexadecimal value from 0 to 7 representing the bit to be read

		example: "ic327F3" would request a read of the 3rd bit of register 0x7F from I2C address 0x32 **Response:**'##ic xx yy z v\r\n' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register z is a one character ASCII hexadecimal value from 0 to 7 representing the bit to be read v is a one character ASCII '0' or '1' representing the value of the bit that just got read example: "##ic 32 7F 3 0\n\r" would be indicating a 0 from bit 3 of register 0x7F of address 0x32
'ilxxyy'	'##il xx yy vv\r\n'	**Read One Byte From I2C Register** **Command:**'ilxxyy' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register example: "il327F" would request a read of register 0x7F from I2C address 0x32 **Response:**'##il xx yy vv\r\n' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register vv is a two character ASCII hexadecimal value from 00 to FF representing the value just read example: "##il 32 7F F2\n\r" would be indicating a value of 0xF2 from register 0x7F of address 0x32
'inxxyy'	'##in xx yy vv\r\n'	**Read One Byte From I2C Register** Identical to 'ilxxyy' except that this command uses a repeated start I2C transaction : https://www.i2c-bus.org/repeated-start-condition/ **Command:**'inxxyy' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register example: "in327F" would request a read of register 0x7F from I2C address 0x32 **Response:**'##in xx yy vv\r\n' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register vv is a two character ASCII hexadecimal value from 00 to FF representing the value just read example: "##in 32 7F F2\n\r" would be indicating a value of 0xF2 from register 0x7F of address 0x32
'&ildddd xxyy'	'##&il dddd xx yy vv\r\n'	**Read One Byte From I2C Register** **Command:**'&ildddd xxyy' dddd is a four character ASCII hexadecimal value from 0000 to FFFF. It is not used by the BFin firmware, but is returned in any responses generated from this command. xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register example: "ilABCD327F" would request a read of register 0x7F from I2C address 0x32 **Response:**'##&il dddd xx yy vv\r\n'

		dddd is the same four character ASCII hexadecimal value from 0000 to FFFF as was sent in the command. xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register vv is a two character ASCII hexadecimal value from 00 to FF representing the value just read example: "##&il ABCD 32 7F F2\n\r" would be indicating a value of 0xF2 from register 0x7F of address 0x32
'iLxxyy'	'##iL xx yy vvvv\r\n'	**Read Two Byte From I2C Register** **Command:**'iLxxyy' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register example: "iL327F" would request a read of register 0x7F from I2C address 0x32 **Response:**'##iL xx yy vv\r\n' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register vvvv is a four character ASCII hexadecimal value from 0000 to FFFF representing the value just read example: "##iL 32 7F EF7D\n\r" would be indicating a value of 0xEF7D from register 0x7F of address 0x32
'iMxxyy'	'##iM xx yy vvvv\r\n'	**Read Two Byte From I2C Register** Identical to 'iLxxyy' except that this command uses a repeated start I2C transaction : https://www.i2c-bus.org/repeated-start-condition/ **Command:**'iMxxyy' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register example: "iM327F" would request a read of register 0x7F from I2C address 0x32 **Response:**'##iM xx yy vv\r\n' xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register vvvv is a four character ASCII hexadecimal value from 0000 to FFFF representing the value just read example: "##iM 32 7F EF7D\n\r" would be indicating a value of 0xEF7D from register 0x7F of address 0x32
'&iLddddxxyy'	'##&iL dddd xx yy vvvv\r\n'	**Read Two Byte From I2C Register** **Command:**'&iLddddxxyy' dddd is a four character ASCII hexadecimal value from 0000 to FFFF. It is not used by the BFin firmware, but is returned in any responses generated from this command. xx is a two character ASCII hexadecimal value from 00 to FF representing the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register example: "&iLABCD327F" would request a read of register 0x7F from I2C address 0x32 **Response:**'##&iL dddd xx yy vv\r\n' dddd is the same four character ASCII hexadecimal value from 0000 to FFFF as was sent in the command. xx is a two character ASCII hexadecimal value from 00 to FF representing

		the I2C address yy is a two character ASCII hexadecimal value from 00 to FF representing the I2C register vvvv is a four character ASCII hexadecimal value from 0000 to FFFF representing the value just read example: "##&iL ABCD 32 7F EF7D\n\r" would be indicating a value of 0xEF7D from register 0x7F of address 0x32
'$Hs'		Start hard-iron calibration for compass
'$He'		End hard-iron calibration for compass
'$HdXXXXX'		Set compass deviation, X is in 10ths of a degree (-1800 to 1800)
'$HvXXXXX'		Set compass variation, X is in 10ths of a degree (-1800 to 1800)
'$Hp'		Print entire contents of IMU EEPROM
'$Hh'		Print just compass heading
'$Hc'		Clear deviation, variation, and hard-iron cal from compass EEPROM
'$P1'	'#$P1'	Enable PicoC Background Mode (allows processing of RCM-BFin commands while PicoC program is running.)
'$P2'	'#$P2'	Disable PicoC Background Mode
'$v1'	'#$v1'	Turn video streaming on (sends video frames without "I" command.)
'$v2'	'#$v2'	Turn video straeming off
'$Op1'	'#$Op1'	Turn Packet Mode on (see other documents for details)
'$Op2'	'#$Op2'	Turn Packet Mode off
'$Ov1'	'#$Ov1'	Turn video test mode #1 on (sends all 0xFFs for video frame)
'$Ov2'	'#$Ov2'	Turn video test mode #1 off
'$Ov3abcd'	'#$Ov3'	Set time between streaming video frames in ms (abcd is four-byte unsigned int sent as binary, LSB first.)
'NActt'	'#NRA'	Stream analog input from RCM/RCM2. 'c' is a one byte binary analog channel number, and 'tt' is a two byte binary (LSB first) value which represents the time in milliseconds between sensor packets. Use 'tt' = 0x0000 to turn off streaming for analog channel 'c'. This command turns on streaming for analog channel 'c' every 'tt' milliseconds. The sensor data sent every 'tt' ms will consist of '#NAcaa' where 'c' is one byte binary analog channel and 'aa' is a two byte binary value (LSB first) representing the 12-bit analog value of channel 'c'.
'NDarbtt'	'#NRD'	Stream one bit from an I2C register. 'a' is a one byte binary I2C address to read. 'r' is a one byte binary I2C register to read (at address 'a'). 'b' is a one byte binary number from 0 to 7 indicating the bit to be read (from within register 'r') 'tt' is a two byte binary (LSB first) number, indicating the time in milliseconds between sensor streaming packet sends. If tt = 0 then streaming from this combination of I2C address and register is turned off. The sensor data sent every 'tt' ms will consist of '#NDarbv', where 'a', 'r' and 'b' are as above, and 'v' is the binary value 0 or 1 representing the state of the digital input. This command is typically used to stream a single digital input's value back to the PC, although it can be used to read the state of any single bit in any I2C register at any I2C address.
'Ndarbtt'	'#NRd'	Identical to 'NDarbtt' except that this command uses a repeated start I2C transaction : https://www.i2c-bus.org/repeated-start-condition/ Stream one bit from an I2C register. 'a' is a one byte binary I2C address to read. 'r' is a one byte binary I2C register to read (at address 'a'). 'b' is a one byte binary number from 0 to 7 indicating the bit to be read (from within register 'r') 'tt' is a two byte binary (LSB first) number, indicating the time in milliseconds between sensor streaming packet sends. If tt = 0 then streaming from this combination of I2C address and register is turned off.

		The sensor data sent every 'tt' ms will consist of '#Ndarbv', where 'a', 'r' and 'b' are as above, and 'v' is the binary value 0 or 1 representing the state of the digital input. This command is typically used to stream a single digital input's value back to the PC, although it can be used to read the state of any single bit in any I2C register at any I2C address.
'NBarbtt'	'#NRB'	Stream data from an I2C register. 'a' is a one byte binary I2C address to read. 'r' is a one byte binary I2C register to read (at address 'a'). 'b' is a one byte binary number indicating the number of bytes to be read from 1 to 255. 'tt' is a two byte binary (LSB first) number, indicating the time in milliseconds between sensor streaming packet sends. If tt = 0 then streaming from this combination of I2C address and register is turned off. The sensor data sent every 'tt' ms will consist of '#NBarv1v2v3...', where 'a', and 'r' are as above, and 'v1' is the first byte returned from the I2C device, 'v2' is the second byte, 'v3' is the third, etc. For example if 'b' = 0x01, then there will only be one return value in the stream reply and it will look like '#NBarv'. This command is typically used to stream data when reading a single byte ('b' = 1) or multiple bytes from a single sensor ('b' > 1). For example, you could stream compass data from a HMC6352 compass at address 0x22, which requires 2 bytes of data to be read out, by setting up the steam as follows (raw hex bytes) : 0x4E 0x42 0x22 0x41 0x02 0x64 0x00. This would ask for a 2-byte read from address 0x22, register 0x41, every 100ms. If the current heading is 180 degrees, the resulting stream reply would look like this : 0x23 0x4E 0x42 0x22 0x41 0x07 0x08.
'Nbarbtt'	'#NRb'	Identical to 'NBarbtt' except that this command uses a repeated start I2C transaction : https://www.i2c-bus.org/repeated-start-condition/ Stream data from an I2C register. 'a' is a one byte binary I2C address to read. 'r' is a one byte binary I2C register to read (at address 'a'). 'b' is a one byte binary number indicating the number of bytes to be read from 1 to 255. 'tt' is a two byte binary (LSB first) number, indicating the time in milliseconds between sensor streaming packet sends. If tt = 0 then streaming from this combination of I2C address and register is turned off. The sensor data sent every 'tt' ms will consist of '#Nbarv1v2v3...', where 'a', and 'r' are as above, and 'v1' is the first byte returned from the I2C device, 'v2' is the second byte, 'v3' is the third, etc. For example if 'b' = 0x01, then there will only be one return value in the stream reply and it will look like '#NBarv'. This command is typically used to stream data when reading a single byte ('b' = 1) or multiple bytes from a single sensor ('b' > 1). For example, you could stream compass data from a HMC6352 compass at address 0x22, which requires 2 bytes of data to be read out, by setting up the steam as follows (raw hex bytes) : 0x4E 0x42 0x22 0x41 0x02 0x64 0x00. This would ask for a 2-byte read from address 0x22, register 0x41, every 100ms. If the current heading is 180 degrees, the resulting stream reply would look like this : 0x23 0x4E 0x42 0x22 0x41 0x07 0x08.
'&NBddarbtt'	'#&NRBdd'	Stream data from an I2C register. 'dd' is a 16-bit (2 byte) value that is not used by the BFin firmware, but will be returned in any responses that are generated because of this command. 'a' is a one byte binary I2C address to read. 'r' is a one byte binary I2C register to read (at address 'a'). 'b' is a one byte binary number indicating the number of bytes to be read from 1 to 255. 'tt' is a two byte binary (LSB first) number, indicating the time in milliseconds between sensor streaming packet sends. If tt = 0 then streaming from this combination of I2C address and register is turned off. The sensor data sent every 'tt' ms will consist of '#&NBddarv1v2v3...', where 'dd', 'a', and 'r' are as above, and 'v1' is the first byte returned from the I2C device, 'v2' is the second byte, 'v3' is the third, etc. For example if 'b' = 0x01,

		then there will only be one return value in the stream reply and it will look like '#NBarv'. This command is typically used to stream data when reading a single byte ('b' = 1) or multiple bytes from a single sensor ('b' > 1). For example, you could stream compass data from a HMC6352 compass at address 0x22, which requires 2 bytes of data to be read out, by setting up the steam as follows (raw hex bytes) : 0x4E 0x42 0x22 0x41 0x02 0x64 0x00. This would ask for a 2-byte read from address 0x22, register 0x41, every 100ms. If the current heading is 180 degrees, the resulting stream reply would look like this : 0x23 0x4E 0x42 0x22 0x41 0x07 0x08.
'|'		If PicoC Background Mode is on, this command will interrupt a running PicoC program
'B'		Execute a mini-PicoC program - can interrupt already running PicoC program. Send PicoC program, then terminate with ESC or Ctrl-C.
'H'		Send Hokuyo data as ASCII
'h'		Send Hokuyo data as binary
'/'		Test command - send any bytes, terminate with ESC or ctrl-c, then RCM-BFin will echo those bytes back
'$OKaaa'	'#$OK'	Turn on or off RCM-BFin streaming shutdown. 'aaa' is a three byte ASCII decimal number from 000 to 255, and represents the number of seconds of 'quiet' (i.e. no data from PC to RCM-BFin) before shutting down all streaming data commands. Use 'aaa' of '000' to turn this feature off. Upon boot, the stream shutdown feature will be off.
C interpreter		
Q	##Leaving PicoC[CR][LF]	Execute C Program Runs the PicoC program stored in flash buffer, which got there via the 'E' line editor, the 'zr' or 'zRxx' command (which reads a flash sector into flash buffer) or 'X' command (XMODEM file transfer). PicoC program must have an exit(1); call to return control to the main firmware. When PicoC Background Mode mode is active (simultaneous PicoC and Firmware commands), you must send a '|' character to get PicoC to exit before the exit(); call in the code. When PicoC Background Mode is not active (only PicoC running, Firmware commands ignored), you have to send an 'ESC' to terminate the PicoC program before the exit(); call runs. When PicoC exits, the RCM-BFin will send the string "##Leaving PicoC\r\n", which indicates to the client that the PicoC program exited normally.
'|'		Break out of running PicoC program If a PicoC program is running and PicoC Background Mode is active, you can send the '|' character to the firmware and at the next statement, the PicoC program will exit and return "##Leaving PicoC\r\n".
'!'		Run PicoC interactively Send ESC to exit.
'B'		Send down short PicoC Program and run it After sending the 'B', send the text of the PicoC program. End the send with the ESC character (0x1B) or ctrl-c (0x03). The program will be immediately executed. Note that if another PicoC program is already running, this new PicoC program will run to completion, interrupting whatever the original PicoC program was doing. So make sure it doesn't take too long to execute. PicoC Background Mode must be turned on for this to work.
'$P?'	'#P?0' or '#P?1'	Report back how many PicoC programs are currently running. If none are running then this command will return #P?0. If one or more are running, then this command will return #P?1.
'$P1'	'#$P1'	Enable parsing of firmware commands (these commands) during PicoC

		program execution. This is called PicoC Background Mode. This command turns it on. By default on bootup this mode is ON (starting with RCM-BFin firmware test28). When enabled, in between each PicoC command, the firmware will check to see if any new commands have been received by the PC and will execute them if so. If you run another PicoC program while executing an existing PicoC program, only two levels of PicoC are allowed. If you try to run a third level (using 'B' or 'Q') it will fail and report an error message. Note that any data that would normally go to the PicoC program from the PC will instead be diverted to the firmware command parser, and any data sent from the running PicoC program will get interspersed with any replies coming from the firmware in response to commands from the PC.
'$P2'	'#$P2'	Disable parsing of firmware commands (these commands) during PicoC program execution. This sets the RCM-BFin to operate as it did previous to version test28 - when running PicoC, firmware commands are ignored.
'$x'		Send PicoC program to alternate buffer If PicoC Background Mode is on, and you want to send down a second PicoC program to run, you need to send it to a different buffer than the normal 'flash buffer' since that is already being used by the already running PicoC program. So this command uses the XMODEM protocol (just like the 'X' command does) to send a file (PicoC program) to the alternate PicoC buffer. When you then execute a 'Q' command, the alternate buffer is used as the source of the second-level PicoC program.
$P3a[CR]	#$P3	Set buffered PicoC printf() mode Where 'a' is an ASCII '1' (for ENABLED) or ASCII '0' (for DISABLED) On boot, the buffered printf() mode is ENABLED ('a' = '1'). When enabled, this option causes all printf() and output() calls to send their output to a buffered serial stream in the RCM-BFin's RAM. The contents of this buffer can be read out with the *R command (see below). The reason for this command is so that output from the PicoC program can be properly structured so that PC clients can understand the difference between PicoC output and command responses for other commands, which may come back from the RCM-BFin interleaved with one another. When DISABLED, printf() and output() return to their normal functionality. Up to 64KB may be stored at once in this buffer before data is thrown out.
*B[CR]	#B,[length], [buffer_data]	Read the contents of the PicoC printf() buffer and clear it This command causes the contents of the PicoC buffered serial stream (from printf() and output() calls in PicoC, when the buffered mode is enabled) to be send in the reply. Note that the reply can be up to 64KB long. After the entire contents of the buffer are sent in the reply, the buffer is cleared. [length] is an ASCII value from 0 to 65525 and it indicates the number of bytes in [buffer_data]. [buffer_data] is the entire contents of the serial stream buffer, and can include any data bytes.

PicoC Shared Memory

There are three shared variable arrays that are accessible from within a running PicoC program. Each array is 255 elements long. There is a string array (where each string can be 255 bytes long), an integer array, and a floating point value array. These arrays are global, and are not cleared between invocations of PicoC. The following six commands allow the PC to read/write from any of these shared variables while the PicoC program is running, thus allowing communication between the PC client and the running PicoC program. All accesses to the shared variable arrays are atomic from both PicoC and the PC sides. All of these commands are completely human readable as they are in ASCII. They can be entered from a terminal emulator. [CR] represents the single byte of 0x0D which is Carriage Return, and [LF] represents the single byte 0x0A which is Line Feed.

*W,S,[index],[value] [CR]	#WS	Write string [value] into PicoC shared memory string number [index]. [value] is terminated by the [CR] but the [CR] will not be part of the string. Strings can be from zero to 255 bytes long. [index] can be from 0 to 255. For example, to write the value "Hi Mom!" into string 55, you would send

		'*WS55,Hi Mom![CR]'. This commands is in pure ASCII except that the string can by any series of bytes except [CR].
*W,I,[index],[value][CR]	#WI	write integer [value] into PicoC shared memory integer number [index]. [value] is a signed integer from -2147483648 to 2147483647 (four byte signed integer value). [index] can be from 0 to 255. For example, to write the value 1234 into integer location 43, you would send '*W,I,43,1234[CR]'. This commands is in pure ASCII and [value] is terminated by [CR].
*W,F,[index],[value][CR]	#WF	Write float [value] into PicoC shared memory float number [index]. [value] is any valid single precision float value. [index] can be from 0 to 255. For example, to write the value 3.14159 into float location 110, you would send '*W,F,110,3.14159[CR]'. This commands is in pure ASCII and [value] is terminated by the [CR].
*R,S,[index][CR]	##R,S,[index],[value][CR][LF]	Read a string from PicoC shared memory string number [index]. [index] can be from 0 to 255. The RCM-BFin will respond with a packet that includes [index] as well as [value] which is the string requested. For example, if string number 24 contained "abcd hello!" then the command '*R,S,24[CR]' would return '##R,S,24,abcd hello![CR][LF]'.
*R,I,[index][CR]	##R,I,[index],[value][CR][LF]	Read an integer from PicoC shared memory integer number [index]. [index] can be from 0 to 255. The RCM-BFin will respond with a packet that includes [index] as well as [value] which is the integer requested. For example, if integer number 88 contained -1234567 then the command '*R,I,88[CR]' would return '##R,I,88,-1234567[CR][LF]'.
*&R,I,[ddddd],[index][CR]	##&R,I,[ddddd],[index],[value][CR][LF]	Read an integer from PicoC shared memory integer number [index]. [ddddd] is a number from 0 to 65535 that the BFin firmware does not use, but simply returns with any responses generated by the command. [index] can be from 0 to 255. The RCM-BFin will respond with a packet that includes [index] as well as [value] which is the integer requested. For example, if integer number 88 contained -1234567 then the command '*R,I,12345,88[CR]' would return '##R,I,12345,88,-1234567[CR][LF]'.
*R,F,[index][CR]	##R,F,[index],[value][CR][LF]	Read a float from PicoC shared memory float number [index]. [index] can be from 0 to 255. The RCM-BFin will respond with a packet that includes [index] as well as [value] which is the float requested. For example, if float number 244 contained 66.77888989 then the command '*R,F,244[CR]' would return '##R,F,244,66.77888989[CR][LF]'.
*&R,F,[ddddd],[index][CR]	##&R,F,[ddddd],[index],[value][CR][LF]	Read a float from PicoC shared memory float number [index]. [ddddd] is a number from 0 to 65535 that the BFin firmware does not use, but simply returns with any responses generated by the command. [index] can be from 0 to 255. The RCM-BFin will respond with a packet that includes [index] as well as [value] which is the float requested. For example, if float number 244 contained 66.77888989 then the command '*&R,F,12345,244[CR]' would return '##&R,F,12345,244,66.77888989[CR][LF]'.
Using the above two concepts of PicoC shared variables, and streaming, combined together they can perform a very powerful function on the RCM-BFin. The commands below show how to perform PicoC shared variable streaming. The basic concept is to send a single command that tells the RCM-BFin to send back a value (or more than one value) from the PicoC shared memory array every 'tt' milliseconds. Once this streaming command is sent to the RCM-BFin, the value of that variable will be sent back to the PC at that interval. This allows the PC to have a 'view' into the current state of the PicoC variables without having to poll using a '*R' command. These commands follow the same comma delimited human readable/writable rules as the other PicoC shared memory commands so they are very easy to type and test from a terminal emulator.		
NP,[a],[tt],[b],[c][CR]	#NRP	[a] is one byte type:'F' (for float), 'I' (for integer) or 'S' (for string) [tt] is an ASCII number, from 0 to 65535, indicating the time in milliseconds between sensor streaming packet sends [b] is an ASCII start value from 0 to 255 indicating which variable to start reading from (index) [c] is an ASCII end value from 0 to 255 indicating which variable to end with (index) - this parameter is optional, and if not present, will just use [b] as the only variable streamed.

		Note that the commas are required between parameters, and that [CR] terminates the command, and that the [c] parameter is optional. For example, this command "NP,F,2000,96[CR]" Would ask the RCM-BFin to generate a streaming packet every 2 seconds consisting of the float variable at index 96. If [tt] = 0 then streaming from this combination of [a], [b] and [c] is turned off.
&NP,[a],[dddd],[tt],[b],[c][CR]	#&NRP,[dddd]	[a] is one byte type:'F' (for float), 'I' (for integer) or 'S' (for string) [dddd] is an ASCII decimal number, from 0 to 65535, which the BFin firmware does not use, but simply sends back with any returned results. [tt] is an ASCII decimal number, from 0 to 65535, indicating the time in milliseconds between sensor streaming packet sends [b] is an ASCII decimal start value from 0 to 255 indicating which variable to start reading from (index) [c] is an ASCII decimal end value from 0 to 255 indicating which variable to end with (index) - this parameter is optional, and if not present, will just use [b] as the only variable streamed. Note that the commas are required between parameters, and that [CR] terminates the command, and that the [c] parameter is optional. For example, this command "&NP,F,12345,2000,96[CR]" Would ask the RCM-BFin to generate a streaming packet every 2 seconds consisting of the float variable at index 96. If [tt] = 0 then streaming from this combination of [a], [b] and [c] is turned off.

Replies

Streaming PicoC Shared Variable data reply formats

The packet sent every [tt] milliseconds will consist of the following:

##NP,F,[b],[f1],[f2],[f3],...[CR][LF]	For Float values: [b] is the starting index of [f1] [f1] is the first floating point value [f2] is the second floating point value (if present) [f3] through 'fn' are the rest of the floating point values (if present) So for a command "NP,F,2000,96,101[CR]" a streaming response packet might be: "##NP,F,96,2.15,-69291.2131,.6,10.0,.000012,99.99[CR][LF]"
##NP,I,[b],[i1],[i2],[i3],...[CR][LF]	For Integer values: [b] is the starting index of [i1] [i1] is the first integer value [i2] is the second integer value (if present) [i3] through [in] are the rest of the integer values (if present) So for a command "NP,I,1000,41[CR]" a streaming response packet might be: "##NP,I,41,1234[CR][LF]"
##NP,S,[b],[s1],[s2],[s3],...[CR][LF]	For String values:

		[b] is the starting index of [s1] [s1] is the first string [s2] is the second string(if present) [s3] through [sn] are the rest of the strings (if present) So for a command "NP,S,1000,15,16[CR]" a streaming response packet might be: "##NP,S,15,abc how are you? Fine!,Robot Has Died. Sensor 4 = 55.[CR][LF]" The main restriction here is that you may not use commas or [LF] or [CR] in your strings or the PC parser will get confused.
See https://github.com/Engineering-3/RCM-BFin/RCM-BFin-Firmware/RCM-BFin_gcc/RCM-BFin C Interpreter.html for full C interpreter documentation.		
Packet Mode		
'$Op1'	'#$Op1'	Turns packet mode on. See Proposal for RCM-BFin Packet Communications document for details on packet mode. As of test46, this is the default at boot.
'$Op2'	'#$Op2'	Turns packet mode off. As of test46, this is no longer the default mode on boot.
Z	##Z pkt mode off[CR][LF]	Turns packet mode off. This command can be sent no matter if the RCM-BFin is in packet mode or not (just like the V command). The RCM-BFin will drop out of packet mode when it receives a Z, and THEN send the response. If you need to communicate with a robot from a terminal emulator (or any application that doesn't handle packet mode) you'll need to send this Z command right after each robot boot, since packet mode is now the default behavior.
RCM-BFin Special I2C functions		
I2C address 0x55		Autoincriment The RCM-BFin will 'listen in' on certain I2C address. When a write or a read from these addresses happens, the RCM-BFin will take special action. The autoincriment address is 0x55. It only listens for writes to 0x55. Any I2C command (like "iw") can be used to write to these special I2C registers. The following resisters exist at this address, and can be written to:
	0x01	Autoincriment 1 Command Register: register 0x01, defaults to value 0x80 Writing to this register triggers the RCM-BFin to write new data to I2C address Autoincriment 1 Destination Address and register Autoincriment 1 Destination Register. These writes happen every 50ms. The data written will increase with every write if the value written to the Autoincriment 1 Command Register is greater than 0x80 (and at a rate proportional to how much beyond 0x80 the value is), and the data written will decrease if the value written to this register is less than 0x80 (and at a rate proportional to how much below 0x80 the value is). Once the data value gets to 0xFF or 0x00, it will no longer be written. To stop the data value from changing, write either 0x00 or 0x80 to this register. The initial value of the data is 0x00.
	0x02	Autoincriment 1 Destination Register: register 0x02, defaults to value 0x18
	0x03	Autoincriment 1 Destination Address: register 0x03, defaults to value 0x2C
	0x11	Autoincriment 2 Command Register: register 0x01, defaults to value 0x80 Writing to this register triggers the RCM-BFin to write new data to I2C address Autoincriment 2 Destination Address and register Autoincriment 2 Destination Register. These writes happen every 50ms. The data written will increase with every write if the value written to the Autoincriment 2 Command Register is greater than 0x80 (and at a rate proportional to how much beyond 0x80 the value is), and the data written will decrease if the value written to this register is less than 0x80 (and at a rate proportional to how much below 0x80 the value is). Once the data value gets to 0xFF or 0x00, it will no longer

		be written. To stop the data value from changing, write either 0x00 or 0x80 to this register. The initial value of the data is 0x00.
	0x12	Autoincriment 2 Destination Register: register 0x02, defaults to value 0x18
	0x13	Autoincriment 2 Destination Address: register 0x03, defaults to value 0x2D

Last updated 11 December 2017, valid for RCM-BFin firmware: RCM-BFin GCC built:xx:xx:xx - xxx xx 2012 v2.0test56

Tan colored cells indicate a change since the last firmware version