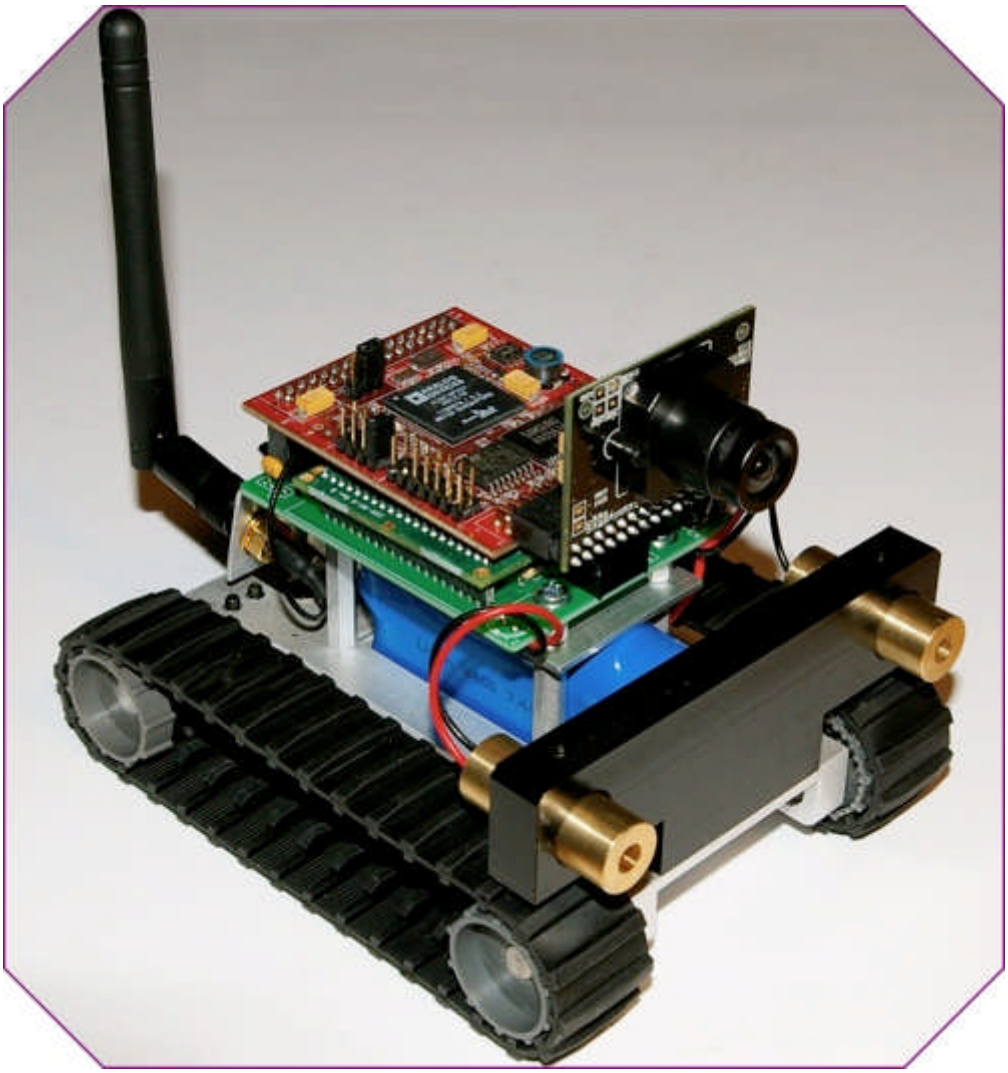
Surveyor SRV-1 Blackfin Robot Mannual

Open Source Wireless Mobile Robot with Video for Telepresence, Autonomous and Swarm Operation



Introduction:

Designed for research, education, and exploration, Surveyor's SRV-1 internet-controlled robot employs theSRV-1 Blackfin Camera Board with 1000MIPS 500MHz Analog Devices Blackfin BF537 processor, a digital video camera with resolution from 160x28 to 1280x1024 pixels, laser pointer ranging, and WLAN 802.11b/g networking on a dual-motor tracked mobile robotic base. Operating as a remotely-controlled webcam or a self-navigating autonomous robot, the SRV-1 can run onboard interpreted C programs or user-modified firmware, or be remotely managed from a Windows, Mac OS/X or Linux base station with Python or Java-based console software. The Java-based console software includes a built-in web server to monitor and control the SRV-1 via a web browser from anywhere in the world, as well as archive video feeds on demand or on a scheduled basis. Additional software support for the SRV-1 is also available by way of RoboRealm machine vision software, Microsoft Robotics Studio, and Cyberbotic's Webots.

Features

- Open Source design with full access to source code (GPL) and schematics
- Robot is fully programmable for autonomous operation
- Extensive software support through 3rd party applications
- Teleoperate mode to drive robot around via console software or remotely via web browser
- Host software has built-in web server and video archiving
- Robot can run programs written in interpreted C and stored in onboard Flash
- Wireless remote control or viewing up to 100m indoors and 1000m outdoors (line of sight)
- Robot can be controlled from a terminal/console for easy testing
- Linux 2.6 support as well as "bare metal" programming with GNU bfin-elf-gcc

Hardware

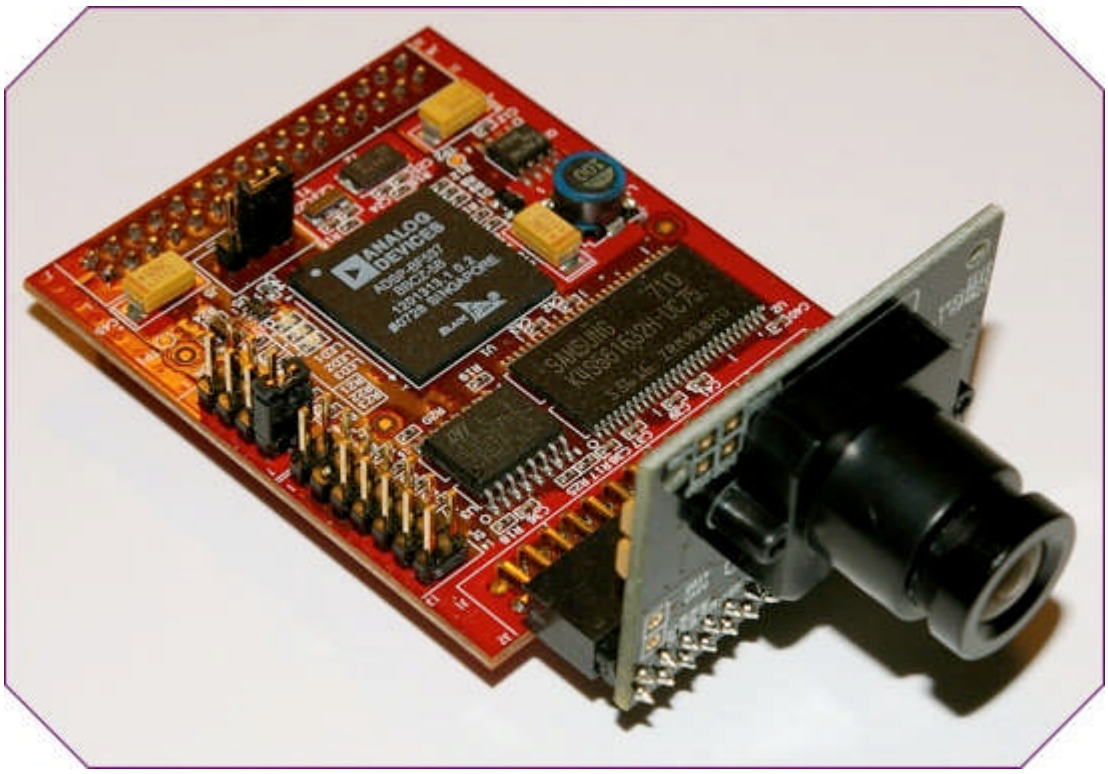
- Processor: 1000mips 500MHz Analog Devices Blackfin BF537, 32MB SDRAM, 4MB Flash, JTAG
- Camera: Omnivision OV9655 1.3 megapixel 160x128 to 1280x1024 resolution
- Robot Radio: Lantronix Matchport 802.11b/g WiFi
- Range: 100m indoors, 1000m line-of-site
- Sensors: 2 laser pointers for ranging
- Drive: Tank-style treads with differential drive via two precision DC gearmotors (100:1 gear reduction)
- Speed: 20cm - 40cm per second (approx 1 foot/sec or .5 mile/hour)
- Chassis: Machined Aluminum
- Dimensions: 120mm long x 100mm wide x 110mm tall (5" x 4" x 4.5")
- Weight: 460gm (16oz)
- Power: 7.2V 2AH Li-ion battery pack - 4+ hours per charge
- Charger: 100-240VAC 50/60Hz (US plug)

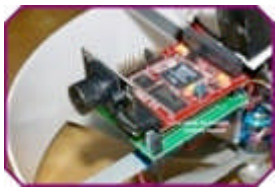
Software

- Robot Firmware: easily updated, written in C language under GPL Open Source, compiled with GNU bfin-elf-gcc and bfin-uclinux-gcc toolchains

- Onboard User Programming: interpreter for "small C" language with special robot-specific commands are provided for running user programs from onboard Flash memory
- Development Tools: GNU toolchains via http://blackfin.uclinux.org
- Console Software: Java based application, runs on Windows, MAC, Linux. WebcamSat web server module built into console software, allows multiple simultaneous remote viewers via Internet
- Robot Control Protocol: Published here. Easily used from other applications
- Third-party Software Support:
    - **RoboRealm** - http://www.roborealm.com/help/Surveyor_SRV1.php - The SRV-1 can now be directly controlled from RoboRealm, a very popular Windows-based machine vision software package for robots. The RoboRealm extensions for SRV-1 allow creation of scripts that combine image processing on live video feeds from the robot, e.g. color filtering, blob detection/tracking, edge detection/outlining and feature extraction, with decision processing and robot motion control, making it easy to create behaviors such as object location and tracking, obstacle avoidance, motion detection, notification, etc, with a web interface, and control can be scripted from C/C++, Python, Java, C#, Lisp, Visual Basic, WScript and COM through the RoboRealm API.
    - **Microsoft Robotics Studio** - http://www.surveyor.com/MSRS.html - Drivers for the SRV-1 in Microsoft Robotics Studio are now available. MSRS is a Windows-based environment for academic, hobbyist and commercial developers to create robotics applications across a wide variety of hardware. Key features and benefits include: end-to-end robotics development platform, lightweight services-oriented runtime, and a scalable / extensible platform.
    - **Webots** - http://www.cyberbotics.com - SRV-1 support is now included in Webots mobile robotics simulation software. Webots provides a rapid prototyping environment for modelling, programming and simulating mobile robots under Windows, Mac OS/X and Linux. The 3D modeling and physics are outstanding.

Surveyor SRV-1 Blackfin Camera





This page contains technical specs and links to the drivers, firmware and applications used with the SRV-1 Blackfin Camera. If you don't find what you are looking for on this page, try Surveyor Robotics Journal orSurveyor Robotics Forum

Specifications

- 500MHz Analog Devices Blackfin BF537 Processor (1000 integer MIPS)
- 32MB SDRAM, 4MB SPI Flash
- JTAG (tested with section5 ICEbear USB-JTAG)
- SPI Flash and UART boot mode select
- External I/O Header (32-pin - 16 x 2 x 0.1")
    - 3.3V Input - 145mA total draw at 500MHz, including camera
    - Board dimensions - 50 mm x 60 mm (2.0" x 2.6"), 36g (1.25 oz) including camera module
    - 2 UARTS - tested at up to 2.5Mbps with CTS/RTS flow control
    - 4 Timers (2 share pins with UART1)
    - SPI - 2 slave select, 1 master select
    - I2C
    - 16 GPIO
    - "S-32 expansion bus" header will support stacking of expansion boards
    - RoHS compliant

- Omnivision OV9655 1.3 Megapixel Sensor
    - AA Format Module (32-pin header - 16 x 2 x 0.1")
    - Interchangeable Lens - Standard is 3.6mm f2.0 (90-deg), optional 2.2mm f2.5 (120-deg)
    - Camera header on production board is 90-degree female (vs straight female header shown in top left photo) - 90-degree expansion connector available to place camera parallel with processor board
    - RoHS compliant

- Radio/Motor Control Module
    - WiFi communication via Lantronix Matchport WLAN 802.11g radio
    - u.fl connector to external antenna
    - On-board 3.3V high efficiency switching regulator (Recom R-783.3-1.0) for battery input (4.75 - 18.0 VDC)
    - Dual H-bridge motor driver (Fairchild FAN8200) with 1000mA capacity per motor
    - 2 switching transistor drivers with 100mA capacity for laser range pointers
    - Board dimensions - 50 mm x 65 mm (2.0" x 2.6"), 10g (0.4 oz)
    - Mounts to SRV-1 Blackfin Camera via 32-pin external i/o header ("S-32 expansion bus")
    - RoHS compliant

- Default Firmware
    - Full-speed frame capture direct to SDRAM at 1280x1024, 640x480, 320x256 or 160x128 pixel resolutions
    - JPEG (Motion JPEG) compression
    - Basic image processing - histogram, pixel sampling, mean, frame difference, blob, scan, count, find
    - Motor control for PWM (H-bridge) and PPM (servo) interfaces
    - Built-in interpreters for Small C or Lisp languages for autonomous operation
    - Real-Time-Clock (milliseconds since reset), internal timer resolution to 10 nanoseconds
    - Direct control of I2C devices
    - In-Application-Programming (IAP) of flash memory

- XMODEM protocol for reliable file transfer
- Direct support for up to 4 Maxbotics ultrasound rangers
- Compiled with GNU Blackfin Toolchain (bfin-elf-gcc) found below
- Can be replaced by u-boot / uClinux found below

Schematics and Drawings

- SRV-1 Blackfin Camera Schematic (204kB)
- OV9655 Camera Module Schematic (136kB)
- Combo Matchport / Motor Driver EAGLE PCB Files: bfin-radiomotor-v2.sch bfin-radiomotor-v2.brd
- **S-32** expansion bus pin assignments
- Legacy Boards
  - Matchport/XBee Radio Module EAGLE PCB Files: bfin-radio-v2.sch bfin-radio-v2.brd
  - Motor Control EAGLE PCB Files: bfin-motor-v2.sch bfin-motor-v2.brd
  - Wiport Radio Module EAGLE PCB Files: bfin-wiport-v1.sch bfin-wiport-v1.brd

---

Reference Documents

- Analog Devices Blackfin BF537 Processor Data Sheet (2.4MB)
- Blackfin Hardware Reference Manual (4.7MB)
- Blackfin Programmer Reference Manual (3.4MB)
- Omnivision OV9655 Data Sheet (664kB)
- Lantronix Matchport User Guide (1.4MB)

SRV-1 Blackfin Camera Firmware (GPL Open Source)

- latest SRV-1 Blackfin firmware snapshot - srv-blackfin-050608.zip .... release notes
  check Surveyor Robotics Forum: SRV-1 Software for updates
- SVN archive for SRV-1 firmware
  From the command-line,
  svn co http://srv.transterpreter.org/

will check out the entire repository. To get only the trunk,
svn co http://srv.transterpreter.org/srv-blackfin/trunk

- SRV_protocol for Blackfin (command set for latest firmware)
- u-boot.ldr - 115kbps boot loader and console
- srv1 uclinux test image with Python (3MB) - 56kbps u-boot + console .... release notes

---

Blackfin Compiler and Debugger Toolchains (GPL Open Source)

- Linux GNU bfin-elf Blackfin toolchain 2008R1_RC8 - install as root from / (creates full path /opt/uClinux...)
  - blackfin-toolchain-08r1-8.i386.tar.gz
  - blackfin-toolchain-elf-gcc-3.4-addon-08r1-8.i386.tar.gz
  - blackfin-toolchain-elf-gcc-4.1-08r1-8.i386.tar.gz
  - blackfin-toolchain-gcc-3.4-addon-08r1-8.i386.tar.gz

- Windows GNU bfin-elf Blackfin toolchain - click on blackfin-toolchain-win32-SVN.exe
- ldr-utils.zip (900kb) - tools to build and manage .LDR files
- ldr-for-windows.zip (773kb) - Windows version of .LDR tools
includes ldr.exe and cygwin1.dll - copy both into GNU Toolchain/SVN/elf/bfin-elf/bin
- TeraTerm terminal program for Windows - works well for firmware uploads
- section5.ch ICEbear JTAG debug tools (use 1.11beta)

SRV-1 Java Console (GPL Open Source)

- SRV1Console.zip (780kb) - last updated 20-Apr-2008 for SRV-1 Blackfin
- SRV1Console and SRV-1 robot Setup Instructions
- SRV1Console.java source code - compile with
"javac -classpath .:ImageButton:RXTXcomm.jar:wstreamd_embed.jar:
SRV1Console.java"
and note that this is **GPL open source**, so modifications and extensions are meant to be shared !

SRV-1 Python console (GPL Open Source)

- **download pySRV1Console-122907.zip**

  If you don't already have Python installed on your system, or if you are running a version prior to 2.4, you should download the Python installer from http://www.python.org/download/. On systems that already have Python installed, the only additional module you may need to install is pySerial, which is downloaded from http://pyserial.sourceforge.net.

  Once you have Python installed, the console is started with:

  python pySRV1Console.py -srv_host xxx.xxx.xxx.xxx -srv_port 10001
  (insert correct SRV-1 IP address)

  Once pySRV1Console is running, the main console page is accessed via your browser at:

  http://localhost:8888/view.html

  Please note that this initial release does not yet support the firmware upload, user flash editor, or image processing functions, but these are in development. Corrections and contributions to this codebase are most welcome !

3rd Party Software Support

- Roborealm - SRV-1 Transterpreter
- Swarthmore College E28/CPSC82 Course: Mobile Robotics
- Brooklyn College robotics.edu - CIS1.5: Introduction to programming using C++
- Lynxmotion Tri-Track
- Coaxial Rotor UAV (almost flies)
- AscTec X-3D Quad Rotor UAV
- SRV-1 Blackfin Wireless Mobile Robot

Common Configuration / Troubleshooting Issues

- Configuring Matchport module via serial
- Restoring SRV-1 Blackfin flash firmware
- Compiling SRV-1 Blackfin firmware on Windows
- Executing C-programs using onboard interpreter
- uClinux Install Basics
- Updating Matchport firmware
- 2.5Mbps Matchport / Blackfin Configuration
- Adding ultrasonic ranging modules
- Adding I2C compass

Definition of the SRV-1 Control Protocol (Blackfin Version) - as of 6 May 2008

All commands from the host to the SRV-1 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 newline ('\n') to their response.

Note that all of these commands can be executed via a terminal program with TCP / telnet capability. 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 ...\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.

There are now 2 versions of firmware - one with a prototype Lisp interpreter and the other with the Little C interpreter. The 'Q' command will execute the C program that has been stored in the robot's flash buffer. The 'P' command will run a Lisp program from the flash buffer, and the '!' command will run Lisp 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 regular SRV-1 command processing loop.

If there are any questions about this protocol, send email to support@surveyor.com or check the Surveyor Robotics Forum.

Command	Response	Description
Core Robot Commands		
except for 'q' command, all parameters are sent as 8-bit binary characters (0x00 - 0xFF)		
'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 160x128
'b'	'#b'	set capture resolution to 320x256
'c'	'#c'	set capture resolution to 640x512
'A'	'#A'	set capture resolution to 1280x1024
'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
'g0'	'##g0'	grab reference frame and enable frame differencing
'g1'	'##g1'	enable color segmentation
'G'	'#G'	disable frame differencing and color segmentation
'I'	'##IMJxs0s1s2s3....'	grab JPEG compressed video frame x = frame size in pixels: 1 = 80x64, 3 = 160x128, 5 = 320x256, 7 = 640x512, 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'	'##ir cc'	I2C register read ('ab' parameters sent as 8-bit binary) a is device id, b is register, cc is 8-bit return value from register displayed as decimal value
'iRab'	'##iR cc'	I2C register read ('ab' parameters sent as 8-bit binary) a is device id, b is register, cc is 16-bit return value from register displayed as decimal value
'iwabc'	'##iw'	I2C register write ('abc' parameters sent as 8-bit binary) a is device id, b is register, c is value written to register
'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)
'o'	'#o'	enable caption overlay
'O'	'#O'	disable caption overlay
'p'	'##ping xxxx xxxx xxxx xxxx\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\n'	sets JPEG quality between 1-8 ('x' is an ASCII decimal character). 1 is highest, 8 is lowest
'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\n'	outputs time in milliseconds since reset
'V'	'##Version ...\n'	read firmware version info response is terminated by newline character
'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
'zc'	'##zclear\n'	clear contents of flash buffer
'zd'	'##zd...\n'	flash buffer dump - dump contents of flash memory buffer to console
'zr'	'##zr\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)
'zw'	'##zw\n'	flash memory write - write 65kb from flash buffer to user flash sector
'zZ'	'##zZ\n'	flash memory boot sector update - writes contents of flash buffer to boot sectors of flash memory - used to replace u-boot.ldr or srv1.ldr - checks first that a valid LDR format image is in the flash buffer
Vision Commands		
all parameters are sent as ASCII		

decimal characters ('0' - '9')		
'vbc'	'##vbc x1 x2 y1 y2 ssss...\n'	the 'vb' command searches for blobs matching the colors in color bin #c, and returns coordinates of an x1, x2, y1, y2 rectangular region containing the matching pixels, along with a count of matching pixels in the blob. 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. an easy way to test this function is to use the 'm' command, which is hardwired to grab color samples from columns 20-59 in rows 0-5 and store them in color bin #0, then try the 'vb0' command to see what blobs show up. if an object is large enough to cover that hardwired area of the image, you should get pretty good tracking, and can try moving the robot away from the object to see how the blob is tracked.
'vccy1y2u1u2v1v2'	'##vcc\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.
'vfc'	'##vfc 01 02 03...\n'	similar to the 'vs' scan command, except that returned values are distance to first pixel matching the target color in each pixel column. 01, 02, 03, etc represents hex value from 0-63 for each pixel column (80 columns total), so a low value indicates the matching color nearby, higher value indicates matching pixels further away, and FF represents no match. the value is roughly proportional to distance from robot to matching pixel.
'vgcx1x2y1y2'	'##vgc y1 y2 u1 u2 v1 v2\n'	the 'vg' command grabs and samples the range of YUV colors in a rectangular region defined by x1, x2, y1, y2, and save info to color bin #c. there are 16 possible color bins, ranging from 0x0 to 0xF. e.g. 'vg030500515' will sample colors for color bin #0, ranging from column 30 through column 50 at heights ranging from line 5 to line 15, where line 0 is the lowest line in the image. the robot will return a string with 'vg' followed by the color bin number, then y1=Ymin, y2=Ymax, u1=Umin, u2=Umax, v1=Vmin, v2=Vmax. as an example, when sampling an area that contained an orange golf ball using the 'vg' command for color bin #0, the robot returned '##vg0 127 176 86 111 154 200\n'. a graphical

		interface for defining the "region of interest" to be sampled would be especially helpful in using this command.
'vh'	'##vhist y u v\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\n'	computes mean values for Y, U and V over the entire image.
'vnc'	'##vnc 01 02 03 ...\n'	similar to the 'vb' scan command, except that the returned values are the number of pixels matching the target color in each pixel column.
'vpxxyy'	'##vp yyy uuu vvv\n'	the 'vp' command samples a single pixel defined by coordinates xx (column 00-79) and yy (row 00-63, where 00 is bottom of image). This is functionally equivalent to using the 'vg' command to sample of the range of 1 pixel, but somewhat less complicated to call. 'vp4032' will sample a pixel in the middle of the image, 'vp4000' will sample a pixel in the middle of the bottom row, etc...
'vrc'	'##vrc y1 y2 u1 u2 v1 v2\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\n'.
'vsc'	'##vsc 01 02 03...\n'	same function as the "Scan" command above for viewing raw "pixel column vector" data , except that we can specify which color bin to use when scanning. 01, 02, 03, etc represents a value 0-63 for each pixel column (80 columns total), so a low value indicates blockage nearby, high value indicates open column (vector). the value is roughly proportional to distance from robot to blockage.
'vx0' or 'vx1'	'##vx 80x64 scaling disabled\n' or '##vx 80x64 scaling enabled\n'	disables or enables 80x64 scaling for most vision commands, i.e. vision results will be based on native resolution or scaled to 80x64. default state is scaling enabled.
'vz'	'##vzero\n'	zeros out all of the color bins
Lisp Commands		
'P'	execute Lisp from flash buffer	runs Lisp program stored in flash buffer, which got there via the 'zr' command (which reads user flash sector into flash buffer) or 'X' command (XMODEM

		file transfer). Lisp interpreter description and sample code found here - http://www.umcs.maine.edu/~chaitin/rov.html Current version of embedded Lisp code includes neither garbage collection nor any control functions for the robot, but these extensions are in development.
'!'	execute Lisp interactively	runs Lisp interactively from console; ESC character (0x1B) terminates interpreter and returns to main firmware control loop
C Language Commands		
'Q'	execute C program	runs C program stored in flash buffer, which got there via the 'E' line editor, the 'zr' command (which reads user flash sector into flash buffer) or 'X' command (XMODEM file transfer)
C primitives		
	assignments	a = b / 20; a = a - 0x55;
	operators	+ - * / % & | ^ = ( ) > >= < <= == !=
	int, char	'int' is 32-bit signed integer, 'char' is 8 bit unsigned chars. variables declared locally or globally - variable names can be up to 31 characters in length
	/* comment */	no processing of characters, e.g. /* this is a comment */ /* this is a two line comment */

	for()	for() loop, can be nested 15 deep, e.g.
		
		main()
		{
		int i, j, k;
		for(i = 0; i < 5; i = i + 1) {
		for(j = 0; j < 3; j = j + 1) {
		for(k = 3; k ; k = k - 1) {
		print(i);
		print(j);
		print(k);
		}
		}
		}
		}

	int|char|void function() ... return	up to 25 functions can be defined, function name length up to 25 characters, and functions can be called recursively e.g.

		factr(int i)
		{
		  if (i < 2) {
		  return 1;
		  }
		  else {
		  return i * factr(i-1);
		  }
		}

	if () ... else	int x;
		x = sonar(1); /* read sonar range */
		if (x < 500) {
		  motors(50, -50); /* if obstacle detected, turn right */
		} else {
		  motors(50, 50);
		}

	char input()	checks the serial port (radio channel) for a single incoming character. ch = input() /* if no data is found, ch = 0 */

	void print()	prints a set of strings and variables, followed by a newline ('\n'), e.g.
		  print("hello");
		  print("test" x y z);

	while()
	while() ... do
	do ... while()
		count = a;
		  do {
		  print(count);
		  } while(count=count-1);

C - robot control/sense		
	void delay(milliseconds)	program delay in milliseconds, e.g. delay(500);

	int get(location)	pointer operation - gets int data from memory location
		
		  int S; S = get(0x00100000); /* sets S from memory location 0x00100000 */
		  int S; int T = 0x00100000; S = get(T); /* sets S from memory location pointed to by T */

	void laser(on/off)	laser(1); turns on laser pointers, laser(0); turns them off

	void motors(left, right)	sets left and right motor output, e.g.
		
		  motors(50, -50); /* spin right */
		  motors(-50, 50); /* spin left */
		  motors(-50, -50); /* go backwards */
		  motors(50, 50); /* go forwards */

	int rand()	returns a pseudo-random number between 0 and 65535

	int readi2c(channel, register)	same as 'ir' command - returns 8-bit value
	int readi2c2(channel, register)	same as 'iR' command - returns 16-bit value
	void servo(left, right)	sets left and right servo output, e.g. servo(50, 50); /* stop */ servo(25, 75); /* spin left */ servo(25, 25); /* go backwards */ servo(75, 75); /* go forward */
	void servo2(left, right)	same as servo() command, but supports a 2nd set of servos on timer ports 6 & 7
	int sonar(channel)	if sonar transducers are installed, returns measured range in channels 1-4. return value of 0 means there is no active transducer on that channel
	void set(location, value)	pointer operation - write int data to memory location set(0x00100000, 555); /* writes 555 to memory location 0x00100000 */ int S = 0x00100000; set(S, 555); /* writes 555 to memory location pointed to by S */ int T = 555; set(0x00100000, T); /* writes value of T to memory 0x00100000 */ int S=0x00100000; int T = 555; set(S, T); /* writes value of T to memory location pointed to by S */
	int time()	get current time (milliseconds since startup) - int t1; t1 = time();
	int writei2c(channel, register, value)	same as 'iw' command
C - robot vision		

	int blob(color, num)	size = blob(colorbin, blobnum); - similar to the "vb" Blob function above, but only returns the selected blob[blobnum] matching the specified color (0 - 9), using reserved variables x, y and z, where x = centroid x coordinate, y = centroid y coordinate, z = blob width. Note that calling blob() with blobnum=0 will compute a new blob search, but calling blob() with blobnum>0 will only retrieve previous located blobs rather than triggering a new search
		e.g.
		
		main() {
		int x, y, z;
		color(2, 100, 140, 90, 110, 130, 160);
		imgcap();
		if (blob(2, 0))

		{
		print("blob 0" x y z); }
		if (blob(2, 1))
		{
		print("blob 1" x y z); }
		}
		
		which might output
		blob 0 40 30 15
		blob 1 60 25 8
		for one blob centered at (40,30) with a width of 15 pixels and another blob centered at (60,25) with a width of 8 pixels
		
		a returned size of 0 would indicate that no matching blob was found

	void color(bin, y0, y1, u0, u1, v0, v1)	sets the y, u, v color range for a color bin, e.g. color(0, 100, 140, 90, 110, 130, 160); would set color bin #0 with y ranging 100-140, u 90-110, v 130-160. functionally equivalent to 'vc' color set function, and results can be confirmed with 'vr' command

	void count(color)	similar to the "vn" color count function above, but divides the image into just 3 non-overlapping regions, with reserved variables x, y and z corresponding to the count of matching pixels in the left, center and right regions, e.g.
		
		main() {
		int x, y, z;
		color(0, 100, 140, 90, 110, 130, 160);
		imgcap();
		count(0);
		print(x); /* left region */
		print(y); /* center region */
		print(z); /* right region */
		}
		
		might output
		15 85 40
		that the center area has the largest number of matching pixels

	void imgcap()	captures image at default resolution
	void imgrcap()	captures and stores reference frame at default resolution
	void imgdiff(flag)	enables image differencing on captured images at default resolution. imgdiff(1) causes imgcap() to compute difference between latest captured frame and reference frame, imgdiff(0) turns off frame differencing
	void resolution('abcA')	changes capture resolution 'a' = 160x128, 'b' = 320x256, 'c' = 640x512, 'A' = 1280x1024
	void scaling(flag)	scaling(0) disables / scaling(1) enables - 80x64 image scaling in vision processing functions. If scaling is enabled, coordinates returned by vision

		are scaled to 80x64 pixel resolution

	void scan(color)	similar to the "vs" Scan function above, but uses matches specified color (0 - 9) against surroundings to look for unblocked areas. the image is divided into 3 overlapping regions, with reserved global variables x, y and z corresponding to left, center and right. if a region is blocked, the value will be zero. if it is not blocked, the value will be positive, with a higher number indicating more open space in that region, e.g.
		
		main() {
		  int x, y, z;
		  color(0, 100, 140, 90, 110, 130, 160);
		  imgcap();
		  scan(0);
		  print(x); /* left region */
		  print(y); /* center region */
		  print(z); /* right region */
		}
		
		might output
		  7 12 0
		indicating that the right is blocked, but forward and left are both open