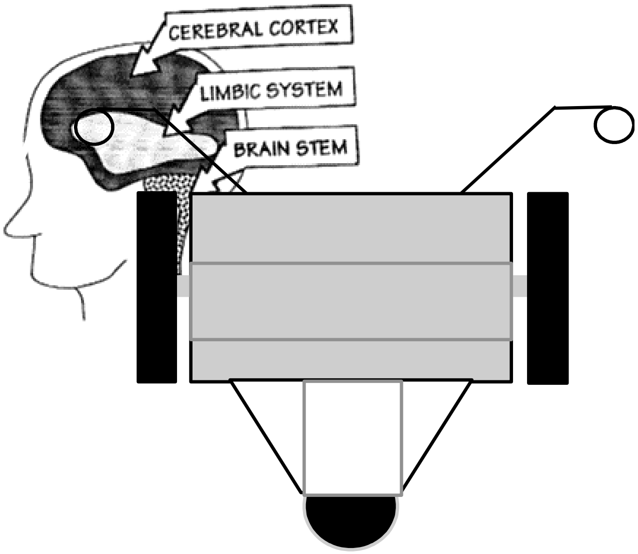
Whiskers

The Artificial Intelligence Robot
Technical Manual

Version 1.42
August 6,1997



Angelus Research Corp,
11801 Cardinal Circle #J
Garden Grove, California (714) 590-7877
Last Revision: August 6, 1999

Table of Contents

Preface

I must say, I had a lot of fun designing Whiskers. I started this project in the fall of 1991. As the Technical Vice President of the Robotics Society of California, I saw a need for an intelligent sophisticated robot that anyone could use. From the Techie type (like myself) to persons with very little technical knowledge, Whiskers scratches the itch of those who love robots. If you have no programming skills, or hardware experience; don't worry, Whiskers is designed to teach. Use his interactive control to learn his High Level Language first. You can do amazing things with it and also get an introduction to programming techniques. On this level you can easily teach him new songs to sing, wander around the room avoiding things, search for sounds, and perform neat tricks to amuse yourself and your friends. Kids really go crazy with him. Pull his whiskers, and see how he reacts. My two daughters, Amy and Sarah, have a ball chasing him around the house.

I would like to thank first my family for putting up with the long hours and endless ramblings on designing intelligence into a robot. I also would like to thank my friends George Ronnquist and Bill Chessell for being sounding boards for my ideas. Finally, this project would have remained a dream if Dr. Kenneth Butterfield, who bought one of my first robots, hadn't caught the Whisker's bug. He was crucial in implementing the many ideas that I had as well as contributing ones of his own.

For the more technically inclined, Whiskers is a very advanced robot. You can use any combination of: the high level language, Forth, or even assembler to program him. His software architecture is very state-of-the-art. He simulates living creatures by having an instinct level process that runs in the background. Just set up the instinctive responses to sensor hits, and control his behavior in a very biological way. Using the behavior level, add your own rules (behaviors) to add to his intelligence. Experiment with sound recognition, speech recognition, navigation, and in the future even neural networks. His capabilities are almost endless. Use your imagination to explore new ideas and share with other Whisker Owners what your have discovered.

Don Golding
Whisker's Creator

Whiskers the Robot

Angelus Research has developed a new and innovative tool for educators to use...Whiskers the Robot. With educational funds being scarce in this current economy, he is very affordable as well. With the current emphasis on giving students marketable job skills, Whiskers is an important tool to have.

Robotics will be the most important emerging technology of the nineteen nineties and beyond. Industry has been rushing to install automation at a feverish pace. While employment has been stagnant, capital expenditures by companies worldwide has been brisk. Graduates who understand robotics and automation are in great demand.

Whiskers the robot was designed to introduce and teach students about this exciting technology. Much like a personal computer, Whiskers is being used by middle schools through advanced mobile robotics research at Universities. If you can speak English, you can program Whiskers. English commands can be typed in interactively to control the robot or new commands can be created easily using a standard word processor and sent to Whiskers over his serial cable. Whiskers is completely self contained. No additional software is required on the personal computer (IBM or Macintosh) other than a terminal program.

Whiskers is very easy to program, anyone can add new commands in minutes. Even people who never have programmed before, can program this personable robot. Collision avoidance is handled automatically by his animal emulation software. Just like a real animal, Whiskers has instincts, behaviors, and goals. Instincts and behaviors are handled automatically in the background. Users can very easily add their own behaviors using just the English like High level language. More advanced users can also use a combination of the High Level Language, Forth, or even assembler. Using other languages like C, Pascal, ands Basic are also possible using a wireless modem and any type of computer.

Overview

WHISKER'S is easy to use and fun to program. You can learn a great deal about important issues in robotics. WHISKERS is a three-wheeled, battery-powered, free roaming, obstacle-avoiding robot. WHISKERS intelligence is derived from a single 68HC11micro controller. His propulsion is provided by two 12 volt geared DC motors both driven by dual H Bridge integrated circuit driver chip.

Whiskers has a computer onboard which allows you to control and program him. Using simple commands like FORWARD, STOP, FIND-SOUND, and many others, you can control him interactively (just typing them at the terminal) or program him by extending his language. Add your own words to perform tasks that interest you. He can avoid obstacles using his four light sensors, two on the side and two on the front, two whiskers; left and right, and the drag on each motor. The motor drag allows you to detect when Whiskers has run into something that his other sensors don't detect. It is a sensor of last resort. If he runs into a table leg between his forward sensors, the motor stall will detect and avoid it.

To program WHISKERS, you will need a personal computer running a terminal emulation program. An IBM compatible, Apple, or any other computer that has a serial port and can run a terminal program. Using the supplied serial cable connected to a serial port and communication software such as: Procomm, Crosstalk, Qmodem, etc. your can communicate with Whiskers. Many of these programs are free (shareware) and easy to get. Set your parameters to 9600 baud, no parity, 1 stop bit and you are ready to go. After connecting the cable supplied to whiskers and your serial port, turn Whiskers on and press any key. This will put you into the interactive mode to give him commands. If you don't hit a key, he will run the auto-start task. When you first get him, this task is a word called WANDER. It will demonstrate many of his features.

WHISKERS, in its basic configuration, can be controlled simply by typing the commands at the keyboard that Whiskers understands. Add to his capabilities by creating new words interactively or by editing a text file on your computer and downloading it to the robot. You have enough on board memory to add thousands of new commands.

An onboard battery backup circuit insures your code will not be lost when turned off. The battery circuit will keep your words safe for about ten days with a full charge. A partial charge would be proportionally less.

Note: the battery charger should be plugged in when Whiskers is not in use.

Chapter I
Technical Insights

Motor Drive

The drive motors used in "WHISKERS" are of a 12 volt DC gear motor design. The wheels are directly mounted on the motor shafts through an adapter. The robot is steered by either reversing the direction of one motor in regards to the other, or setting the motor speeds so they are different from one another. This is called a differential direct drive system.

Pulse Width Modulation

DC motors are the mainstay of robotics design. Controlling the speed of a robot must be done in the most efficient design possible. This is because of the finite amount of power stored in the battery. Saving energy means our robots can run for longer periods of time.

The most intelligent way to control a robots speed is called Pulse Width Modulation(PWM). This technique operates the drive integrated circuit in a full on or a full off mode. Semiconductor devices usually dissipate very minimal power when operated in this mode.

To begin to understand PWM, lets imagine a pulse train from the CPU that consists of on and off periods of equal time interval (50% duty cycle). This pulse train from the computer is then used to drive the motor. The pulses are applied to the motor so rapidly, that the mechanical inertia of the robot completely smoothes out these pulses to give an average speed proportionally to the duty cycle of the pulses. i.e.; on time verses off time. If the computer program wants additional speed, it increases the duty cycle by increasing the on period and reducing the off portion accordingly. This raises the average electrical power applied to the motor.

In WHISKERS the PWM pulse trains are generated by the instinct level through output lines: (PA3, PA4, PA5, and PA6) and fed to X17(UDH2993B chip) to interface the motors to the CPU. The WHISKERS design has implemented a simple feed back loop to the CPU. A motor current circuit is incorporated to monitor each motor. A series resistor (2 ohm) has been placed in each motor lead. R41monitors motor #1and R40 monitors current of motor #2. The circuit then filters or averages the voltage which represents current flow, before passing it to input ports PE5 and PE6. The A/D function allows the program to read a representation of the motor current. The current can then be used as an approximate motor load. A very high current will represent a motor stall condition.

Sensors

WHISKERS can not know where it is in space with out some type of sensors. Think about the problem that this little machine faces as it strolls about the premises. Try blindfolding yourself, stuffing cotton in your ears, and putting boxing gloves on your hands then try walking around the room. This is the task that WHISKERS tries every time the switch is turned on and he is allowed to roam.

Currently, we have provided WHISKERS with four fixed LED light transmitters and four optical sensors and one semi-directional (its located on the front) acoustical sensor and two discretely switched mechanical whisker detectors. It is the intention of this section to discuss a different type of sensor and the pro's and con's of each.

Optical Sensors(Light)

Sight on WHISKERS is an important sense to provide. Robotic vision systems are very expensive and complex. Elementary vision systems are used for nothing more than detecting obstacles. The more advanced vision systems determine patterns and shapes with the most advanced using two cameras and processing 3D images.

Hardware for making a single element robot eye is rather simple, so Whisker uses four of these to see his environment.

A single light sensitive photo detector (photodarlington) is all that is needed to sense the presence of light. The photo detector acts as a variable resistor. The resistance varies with increasing or decreasing light. With no light present, the resistance is at maximum. Apply light and the resistance is reduced.

The photo detector is connected in series with a resistor to operate as a voltage divider. The output tap is between the photo detector and the resistor. At no light the output swings to its highest value. This voltage value present at the center tap is connected to an A/D input so it's voltage value can be determined by the program.

WHISKERS was provided with four independent optical input channels of Left/front, Left/side, Right/ front, and Right/side each connected to PE0, PE1, PE2, and PE3 chip input pins respectively.

Optical Sensor Pairs (Collision)

The four optical sensor units supplied with the robot unit each consist of two elements, a high intensity LED and a photo detector as discussed previously.

The LED selected is a high intensity model transmitting 2000mcd of light when rated current is applied to the LED. The LED, when pulsed, projects a beam of light of high intensity. The LED transmits a non-coherent red light of 660nm wavelength.

The LED's as provided with WHISKERS are pulsed simultaneously and are triggered by the PA7 output of the CPU chip.

The detectors provided has a rather broad band spectral response. It was selected because of its robust (rugged) electrical characteristics.

The wide band response of the photo sensor makes it sensitive to room lighting, flash bulbs, flash lights and other stray light etc. This was chosen because it's output can be utilized for more than just collusion detection.

A selective red filter is installed over the optical detector to reduce the light input to frequencies that was transmitted by the LED.

The program provided with WHISKERS, first reads the digital value of the ambient room light. It then will pulse the LED on and then read the returned sensor value. It then takes a difference reading between the two and compares it to a threshold value (trigger level). If the level is above the threshold value, then the program considers the return a valid obstacle hit.

The program repeatedly pulses the LEDS on and off at the rate of about 30 times per second. The sensors are located as follows: two in the front, and one on each side. Each detector is read by one channel of the eight channel analog to digital converter. The light values are read and stored in memory by the instinct level. The values are compared to individual trigger levels to determine if there is an obstacle detected. This threshold is a software selectable term which increases as the room darkens. The trigger level can be used as a crude range selection device.

If there was a return detected by the optical sensors, then the instinct level overrides the desired direction, replacing it with the specific direction chosen for this sensor. When the obstacle is passed, the desired direction is automatically resumed. Without programming, you can change any of the sensor's responses with stop, left pivot, right pivot, forward, backup, and pivot about either wheel.

Turn Whiskers on and watch the optical detector system illuminate a barrier in its way and effectively acknowledge the obstacle. You can hold your hand in the light and notice that the detection range has changed. This is as expected as the reflectivity of the target or obstacle influences the amount of light returned.

Whiskers Detectors

Your robot is furnished with a set of mechanical whisker detectors located on each forward front corner of the robot. As you might have guessed this is where your robot got its name.

The whiskers are constructed of spring steel piano wire. They are securely connected to terminals on the electronics board. When the robot runs into something close that wasn't detected by the optical detectors, the whiskers are deflected or depressed. This in turn presses the steel wire against a metal post contacts on the board and indicates an obstacle to the CPU. The contact posts on the board are designed so that the spring wire detectors can be activated by either of two motions, a sideways pull or a push to the wire whisker.

The responsive actions are determined by the instinct level.

The Battery

A few words about the selection, care and feeding of the primary power source of WHISKERS.

Whiskers uses a gel cell battery. They are hermetically sealed so they don't leak. They also hold allot of energy for their weight. *Whiskers should be left on the battery charger whenever not in use.* He has an integral battery charging circuit so the battery won't be overcharged.

The charger supplied with WHISKERS is selected to provide the proper charge rate for the battery supplied. It will keep the battery up to maximum performance. The battery is charged using the float charge method at a regulated 13.6 volts. The charger may be left on continuously. Whiskers can operate for about 4 hours or more when fully charged.

The instinct level monitors the battery voltage. When it drops below a level set by the user, the robot will execute a battery low behavior; i.e., stop and cry for food.

Microphone

One of the most important senses the human beings and other animals use is hearing. Sound detection is simpler to implement on a robot than the sight process. A microphone lets WHISKERS listen to the world around him. The sound can be digitized at rates beyond human hearing or lower. Words are included to average the sound, find peaks, and so forth. This capability can be used to search out the specific sounds within a room.

The most sensitive type of microphone is the electret condenser element. This is the type used on Whiskers. It is mounted on the front so the robot can find sounds and move toward them. The robot does generate noise itself by moving. For critical sound analysis, stop the robot before sound sampling.

The output of the microphone is amplified and passed to analog to digital converter input PE7, where it is digitized and then is available for reading by the program.

Programming determines how often the sound is sampled. The minimum frequency requirement for discrete sampling of an analog signal is as follows:

Fs > 2fmax
Fs = sampling frequency
fmax= max frequency to be sampled

The sampling frequency must be greater than twice the frequency that is being sampled. Remember, determine the frequency of the information your interested in and then sample the information at least twice this rate.

Speaker Output

WHISKERS can create sounds through this speaker. WHISKERS can beep, pop, play music and reproduce any other monophonic sound that one can program.

The speaker is a piezo transducer speaker. It is connected to a digital output SS\ from the CPU through an amplifier device Q2. It can be driven from the program by executing a **CYCLES** command. By choosing the correct parameters, the frequency of the speaker output can be changed. The WHISKERS robot can even be made to sing your favorite song with simple programming.

The speaker can be used for outputs of warnings, musical notes, program debugging information, alerts and any other aural or acoustic data that your program requires.

Chapter II
Software Architecture

The robot control software included with this robot was created to best emulate animal behavior. This is the direction most mobile robot researchers are currently taking today. Behaviors are created with various levels of complexity and abstraction. Simpler but more critical behaviors such as collision avoidance, can take control from higher level functions, automatically. A high level function may be looking for a sound, for example. An eminent collision with an obstacle would interrupt (subsume) the high level function until the obstacle was avoided.

Programming a mobile robot is a completely different type of problem than programming a typical computer. Take, for example, programming a database. The program asks the user for a name to search for, then take considerable time, searching the data for a match. A screen is displayed saying the computer is busy, please wait. The computer will not respond to any external events (keystrokes) until it has completed this task. Now consider a robot wandering around a room. There are many obstacles in the room which the robot must avoid instantly. You commanded the robot to sing a song and wander about the room. While the robot is singing, he must respond to outside events in real time! This means that to control the robot, you must use some form of multitasking, or interrupt based sensors.

Whiskers uses a software architecture which conceptually consists of three layers:

LAYER	*PRIORITY*	*INTELLIGENCE REQ'D*	*OPERATES IN:*
Instincts	Highest	Least	Background
Tasks & Behaviors	Medium	Medium	Background
Outer Layer	Least	Highest	Foreground

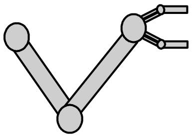
Task processing is tied to one of the processor's counter/timers which initiates an interrupt several hundred times a second, pre-emptively. Each instinct has an input value from one of the sensors, a trigger level to compare, a motor mask (direction override) , and a fired flag. The basic instincts the robot has are each processed and if the sensor value is greater than the trigger level, the instinct overrides the desired motor direction with it's motor mask, and sets the flag to true. The instincts toward the end of the list are of greater priority as they can override a previous instincts motor direction. This occurs as long as the sensor sees the obstacle. When the obstacle disappears, the original motor direction resumes. By changing each instincts motor mask and trigger level, the operator can completely change the robots response to the outside world. The instinct level also updates several system monitoring variables such as: battery voltage, compass, ambient light levels, etc.

Tasks and Behaviors execute only after all instincts have been processed. This processor goes down the task list, executing all tasks and behaviors until it comes to the end. Control then returns to the outer layer. A special note is in order, while a behavior is being performed, all instincts are still performed in the background. This simply means that the instincts for collision avoidance still operate even though a behavior has taken control. The programmer can create special background tasks such as timers, event counters, or any other special processes he/she may need. Behaviors are different from tasks in that they must first check to see if conditions (sensor combinations?) are met before firing (executing). Tasks are always executed. It is allowable for these tasks and behaviors to modify the task list in anyway they wish. They can delete tasks and behaviors, add them, or change variables that other processes may use. This is a very powerful concept. Using this layered approach, behaviors can be added to the robots' task list in a prioritized fashion.

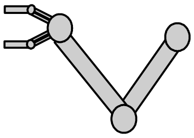
The Outer Layer can perform tasks such as get commands interactively, mapping functions, or monitor and change the Instinct and Task/Behavior levels to accomplish a specific goal. By using this layered abstraction technique, programming becomes very simple. Even non programmers can create very intelligent behaviors for the robot. Very advanced behaviors can emerge from numerous simpler ones.

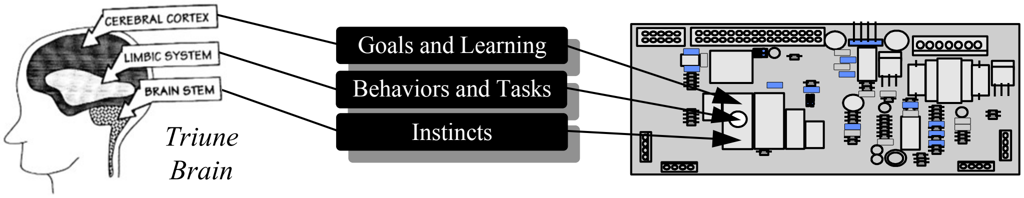
***From Angelus Research Corp*.**

Artificial Intelligence Breakthrough!



New Intelligent Controllers based on the Triune Brain





Triune Brain

Goals and Learning

Behaviors and Tasks

Instincts

Applications Include:

- Industrial Automation
- Automated Guided Vehicles
- Autonomous Mobile Robots
- Intelligent Arms/End Effectors
- Closed Loop Servo Control
- Intelligent DC Motor Control

Features and Benefits

- Program in English
- Real-time Sensor/Motor Fusion
- Multitasking and Networkable
- Simultaneous Multi-Axis Control
- Narrow Beam Intelligent Sonar
- Low Power and Low Cost

Instinct Level Modification

The instinct level performs time critical functions such as collision avoidance and system monitoring. Each of his four optical collision sensors and three summed sensors, has a trigger level, motor mask, and flag registers. By setting these to different values, completely different responses can be had. Their are nine motor masks that can used with each sensor. Set the mask register and trigger register for each sensors. Whiskers will automatically override the desired direction for as long as the sensors is triggered. The corresponding flag will also be set. You could override the instincts by setting all sensor motor masks to the desired direction. For example, FWD ALL-INSTINCTS and a FORWARD command would effectively disable collision avoidance. You should use rCOLLISION ON/OFF for this however.

Trigger levels:

A trigger level is the value required to *trigger* an override action such as the left front optical sensor seeing something. Higher values make the sensor less sensitive and also decreases the range proportionally.

rLF-TRIGGER	Left front light obstacle detector trigger
rLSF-TRIGGER	Left side front light obstacle detector trigger
rRF-TRIGGER	Right front light obstacle detector trigger
rRSF-TRIGGER	Right side front light obstacle detector trigger
rL-TRIGGER	Left summed light obstacle detector trigger
rR-TRIGGER	Right summed light obstacle detector trigger
rF-TRIGGER	Front summed light obstacle detector trigger
rLM-TRIGGER	Left motor stall trigger
rRM-TRIGGER	Right motor stall trigger

Used as: 30 LF-TRIGGER SET Sets left front trigger to 30

Motor masks

A motor mask is a abbreviation of the robot's various movements (motor directions). You must store this value is a motor mask register to have an effect on the robot.

Used as: REV rF-MASK SET

Robot will backup whenever the front sensors see and obstacle.

LF	01100000	Left motor forward
LR	01000000	Left motor reverse
RF	00011000	Right motor forward
RR	00010000	Right motor reverse
FWD	01111000	Both motors forward
REV	01010000	Both motors reverse
ST	00000000	Both motors stop
PVR	01110000	Right Pivot
PVL	01011000	Left Pivot

Instinctive motor response mailboxes:

These are used to hold the motor response as described previously. They determine the robots response to a sensor seeing and obstacle.

rLF-MASK	Left front light obstacle motor mask
rLSF-MASK	Left side front light obstacle motor mask
rRF-MASK	Right front light obstacle motor mask
rRSF-MASK	Right side front light obstacle motor mask
rL-MASK	Left summed light obstacle motor mask
rR-MASK	Right front light obstacle motor mask
rF-MASK	Front summed light obstacle motor mask

Used as:

PVR	rLF-MASK	SET	Robot will pivot right on obstacle detection
PVR	rLSF-MASK	SET	Robot will pivot right on obstacle detection
PVL	rRF-MASK	SET	Robot will pivot left on obstacle detection
PVL	rRSF-MASK	SET	Robot will pivot left on obstacle detection
REV	rF-MASK	SET	Robot will backup right on obstacle detection
RR	rL-MASK	SET	Robot will reverse right motor on obstacle detection
LR	rR-MASK	SET	Robot will reverse left motor on obstacle detection

DEFAULT-INSTINCTS	Set all instinct mask registers to factory defaults
ALL-INSTINCTS	Set all instinct mask registers to the same value
SAVE-INSTINCTS	Save all current instinct motor masks
RESTORE-INSTINCTS	Restore all instinct motor masks to there previous values

Used as:

SAVE-INSTINCTS	Save current instinct motor masks
ST ALL-INSTINCTS	Set all instincts to stop
DEFAULT-INSTINCTS	Set all instincts to factory defaults
RESTORE-INSTINCTS	Restore instinct motor masks

Light Detectors Return Values (range 0-255)

These are registers that hold the actual amount of light reflected back to the sensors. In the case of the motor drag registers (rLM-VALUE and rRM-VALUE), the amount of drag.

rLF-VALUE	Actual return value of left front obstacle sensor
rLSF-VALUE	Actual return value of left side front obstacle sensor
rRF-VALUE	Actual return value of right front obstacle sensor
rRSF-VALUE	Actual return value of right side front obstacle sensor
rL-VALUE	Actual return value of left summed obstacle sensor
rR-VALUE	Actual return value of right summed obstacle sensor
rF-VALUE	Actual return value of front summed obstacle sensor
rLM-VALUE	Actual return value of left motor drag sensor
rRM-VALUE	Actual return value of right motor drag sensor

Used as:

rLF-VALUE GET DISPLAY	Shows value of left front obstacle detector

Collision

Each bit position in vCOLLIDED represents a certain collision sensor being triggered. Can use as a general *any sensor triggered* flag.

vCOLLIDED	Any collision = true

Used as:

```
vCOLLIDED VALUE

  IF
    STOP
  ELSE
    FORWARD
THEN
```

Flag mailboxes

Used to check each individual optical sensor for triggering. A sixteen bit number is returned with the appropriate bit set for the sensor. Uses vCOLLIDED above. Should normally be used in conjunction with SENSOR below.

LF-OBSTACLE	Left front hit mask
LSF-OBSTACLE	Left side front hit mask
RF-OBSTACLE	Right front hit mask
RSF-OBSTACLE	Right side front hit mask
L-OBSTACLE	Left summed hit mask
R-OBSTACLE	Right summed hit mask
F-OBSTACLE	Front summed hit mask
LW-OBSTACLE	Left whisker hit mask
RW-OBSTACLE	Right whisker hit mask
LM-OBSTACLE	Left motor stall hit mask
RM-OBSTACLE	Right motor stall hit mask
SENSOR	Used with flag mailboxes to convert mask to TRUE or FALSE so boolean operators like: AND, OR, NOT can be used.

Used as:

```
LF-OBSTACLE SENSOR

 IF
   RIGHT PIVOT
 THEN
```

Miscellaneous

vPWM-CYC	Frequency of instinct level, larger number equals greater period, default=16667
rMAX-SPEED	Maximum motor speed. Set to 100 to use percentages.
rCORRECTION	Modifies right motor speed to tune robot to go straight
rCOLLISION	Enable or disable collisions avoidance
rNO-STALLS	Disable stall detection
rSTALL-DELAY	Set stall delay
rLIGHTS	Disable/Enable light sensor avoidance
rWHISKERS	Disable/Enable whisker avoidance
rSENSE	Disable/Enable Analog to digital converter. Needed for lights and stalls
rSTALLS	Disable/Enable stalls detection
rLIGHTS	Disable/Enable light collision sensors
vCOMPASS	A differential counter for the high level tasks to monitor relative direction
vMAX-COMPASS	Maximum value for compass to be reset

Used as:

```
rWHISKERS ON
rLIGHTS ON
rSENSE OFF
16 rSTALL-DELAY SET
rSTALLS OFF
455 vMAX-COMPASS NOW          set to 455
1000 vCOMPASS NOW             set to 1000
vCOMPASS VALUE                get current reading
ST ALL-INSTINCTS              Set all instincts to the same mask
```

Task and Behavior Level Modification

The task processor works in the background. The execution of the task processor is based on the number of *instinct levels* that have been processed, since the last time *it* was processed. You can change the frequency by changing the value of rINSTINCTS. For example, 16 rINSTINCTS SET will cause the task processor to be serviced every 16 instincts. As this value gets lower, the tasks are processed more frequently and processing time is taken away from the *outer level*. As this value gets higher, (255 maximum) the tasks are processed less frequently and more time is available for the *outer level*. The maximum number of tasks is 500 in this version. You can think of behaviors as rules which Whiskers must follow in a prioritized fashion for him to survive.

ADD-TASK:	Add new task to the task list. This must be done interactively or in a text file download in the current version.
DEL-TASK:	Delete a task from the task list. This must be done interactively or in a text file download in the current version.
CLEAR-TASKS	Delete all tasks from the list. Sets each task cell to zero.
PRIORITY	Used with ADD-TASK: to set the priority of task when added to the list.
TASKS	Location of task list in memory.
SHOW-TASKS	Display all tasks in list. Used as: TASKS SHOW-TASKS
MULTITASKING	Enable task list processing
NORMAL	Disable task list processing
rINSTINCTS	Number of instinct levels processed between task processing

Samples

Command	**Task list**
1 PRIORITY ADD-TASK: LOW-BAT	LOW-BAT
1 PRIORITY ADD-TASK: F-HIT	F-HIT
1 PRIORITY ADD-TASK: BW-HIT	BW-HIT
DEL-TASK: F-HIT	LOW-BAT BW-HIT
1 PRIORITY ADD-TASK: F-HIT	F-HIT LOW-BAT BW-HIT
16 rINSTINCTS SET MULTITASKING	

Notes:

1) Make sure you have MULTI-TASKING in your AUTO-START word. The default on start up is NORMAL.

2) Do not print to the screen or use any other terminal IO words in this version. Examples are: ." , EMIT, KEY etc.

3) Test you tasks carefully. Make sure they don't leave items on the stack. Do this by executing your new task then typing **.S**. You should get an **EMPTY** message.

Such as: CLEAR-STACK F-HIT .S [Enter]

EMPTY is displayed

Sample Task Source Code

```
VARIABLE vFHIT                    Create a memory location called vFHIT
0 vFHIT NOW                       Initialize to zero

: F-ECOUNTER                      Front summed sensor event counter

F-OBSTACLE SENSOR                 Both front sensors hit?

  IF           If true...
    vFHIT INCREMENT               Increase by 1
  THEN
;

VARIABLE vBWHIT                   Create a memory location called vFH
0 vBWHIT NOW                      Initialize to zero

: BW-ECOUNTER                     Both whiskers touched simultaneous event counter

  RW-OBSTACLE SENSOR              Right whisker hit?
  LW-OBSTACLE SENSOR              Left whisker hit?
    AND
      IF                          If true...
          vBW-HIT INCREMENT       Increase by 1
      THEN
;

VARIABLE vMY-COUNTER
0 vMY-COUNTER NOW

: INC-MY-COUNTER
  vMY-COUNTER INCREMENT ;

: DEC-MY-COUNTER
  vMY-COUNTER DECREMENT ;

1 PRIORITY     ADD-TASK: INC-MY-COUNTER
1 PRIORITY     ADD-TASK: F-ECOUNTER
1 PRIORITY     ADD-TASK: BW-ECOUNTER
```

Outer level word to see tasks updating variables

```
: SHOWME
    CR                                  Newline
    BEGIN                               Begin looping

      vMY-COUNTER VALUE                 Get value
      DISPLAY                           Display it
      SPACE                             Add space between numbers

      vBWHIT VALUE                      Get value
      DISPLAY                           Display it
      SPACE                             Add space between numbers

      vFHIT VALUE                       Get value
      DISPLAY                           Display it
      SPACE                             Add space between numbers

    ?TERMINAL UNTIL ;                   Break out of loop on any key
```

Sample Behavior Source Code

```
VARIABLE vFRUSTRATION                       Create a two byte memory cell, max value = 65535.
0 vFRUSTRATION NOW                          Make it a value of zero.

100 vWAIL NOW                               Adjust wail sound.

VARIABLE vFRUST-TRIGGER                     Create a two byte memory cell.
8 vFRUST-TRIGGER NOW                        Make it a value of 8.

: FRUSTRATED                                Define a word called FRUSTRATED
  vFRUSTRATION VALUE                        Get vFRUSTRATION value
  vFRUST-TRIGGER VALUE                      Get vFRUST-TRIGGER  value

    >                                       Is vFRUSTRATED greater than vFRUST-TRIGGER?
      IF                                    If true...
          SAVE-DIR                          Save current direction

            5 0 DO                          Do this five times
              40 LEFT SPEED                 Set left speed to 40
              40 RIGHT SPEED                Set right speed to 40
              LEFT PIVOT                    Do a left pivot
              30 1 WAIL                     Make a wailing sound
              RIGHT PIVOT                   Do a right pivot
              30 1 WAIL                     Make a wailing sound
            LOOP                            Do again if less than five

          RESTORE-DIR                       Restore direction
          0 vFRUSTRATION NOW                Make it a value of 0
          EXIT                              Terminate this process

      ELSE                                  If false...

          vFRUSTRATION VALUE                Get vFRUSTRATION value

            0<                              Less than zero?
              IF                        If true...
                1 vFRUSTRATION NOW          Make it a value of 1
              ELSE                          If false...
                vFRUSTRATION DECREMENT      Subtract 1
              THEN                          End of IF statement

      THEN                              End of IF statement
;                                           End of definition
```

```
REG: rBAT-TRIGGER                               Create a two byte memory cell, max value = 65535.
80 rBAT-TRIGGER SET                             make it a value of 80

: LOW-BAT                                       Define a word called LOW-BAT
  rBAT-TRIGGER GET                              Get rBAT-TRIGGER value
  rADR4-B GET                                   Get rADR4-B value
  DUP                                           Duplicate value
    <                                           Is it less than?
      IF        If true...
        DISABLE                                 Turn off Instincts and Task Processor
        DECIMAL                                 Display value in decimal
        rADR4-B GET DISPLAY                     Get rADR4-B value
        SPACE
        DISPLAY" Battery low!" CR               Display Battery low!= on terminal, number, and newline
        DISPLAY" Please get me to"              Display Please get me to
        DISPLAY"  the charger...FAST!" CR       Display the charger...FAST!
        STOP                                    Stop robot

        BEGIN                                   Begin endless loop
        500 50 WARBLE                           Make warble sound
        AGAIN                                   Go back to BEGIN and do again...

      THEN      End of IF statement
;                                               End of definition

: RM-HIT                                        Define a word called RM-HIT
  RM-OBSTACLE SENSOR                            right motor stalled?

  IF            If true...
    4 VFRUSTRATION INCREASE                     Increase by 4
    SAVE-DIR    Save current direction
    SAVE-SPEEDS                                 Save current speeds
    BEST-PIVOT                                  Look at side sensors for clear route
    RAMP-UP                                     Ramp speed up to 80 at factor of 5
    1000 8 BIRD-CALL                            Make bird call sound
    RESTORE-DIR                                 Restore direction
    RESTORE-SPEEDS                              Restore direction
    CLEAR-SENSORS                               Clear all sensors flags
  THEN          End of IF statement
;                                               End of definition
```

```
: LM-HIT                                    Define a word called LM-HIT
  LM-OBSTACLE SENSOR                        Left motor stalled?

    IF          If true...
      4 vFRUSTRATION INCREASE               Increase by 4
      SAVE-DIRSave current direction
      SAVE-SPEEDS                           Save current speeds
      BEST-PIVOT                            Look at side sensors for clear route
      5 80 RAMP-UP                          Ramp up to speed 80 by factor of 5

      400 9 BIRD-CALL                       Make bird call sound
      RESTORE-DIR                           Restore direction
      RESTORE-SPEEDS                        Restore direction
      CLEAR-SENSORS                         Clear all sensors flags
    THEN        End of IF statement
;                                           End of definition
```

```
: F-HIT                                 Define a word called F-HIT
  F-OBSTACLE SENSOR                     Both front sensors hit?

    IF          If true...
      4 vFRUSTRATION INCREASE           Increase by 4

      LM-OBSTACLE SENSOR                Left motor flag
      RM-OBSTACLE SENSOR OR             Right motor flag
        NOT   IFIf not equal
                SAVE-DIR                Save current direction
                SAVE-SPEEDS             Save current speeds
                BACKUP                  Robot backs up
              2 90 RAMP-UP              Ramp speed up to 90 at factor of 2
                300 LASER               Make laser sound effect
                BEST-PIVOT              Look at side sensors for clear route
                60 1 WAIL               Make wail sound effect
                RESTORE-DIR             Restore direction
                RESTORE-SPEEDS          Restore direction
                CLEAR-SENSORS           Clear all sensors flags
              THEN                      End of IF statement

      THEN      End of IF statement
;                                       End of definition

: BS-HIT                                Define a word called BS-HIT
  LSF-OBSTACLE SENSOR                   Left front hit flag
  RSF-OBSTACLE SENSOR                   Right front hit flag
    AND                                 Both side sensors hit?

      IF        If true...

        SAVE-DIR                        Save current direction
        SAVE-SPEEDS                     Save current speeds
        0 vFRUSTRATION NOW              Make it a value of 0, turn off FRUSTRATION
        50 LEFT SPEED                   50 percent left speed
        50 RIGHT SPEED                  50 percent right speed

        LSF-GREATER?                    Left side sensor greater than right flag
          IF    If true...
            FWD ALL-INSTINCTS           Set all instincts to forward direction
            10 RIGHT %SPEED             Drop right speed by 10 percent
            300 1 WARBLE                Make warble sound effect
          ELSE                          If false...
            10 LEFT %SPEED              Drop left speed by 10 percent
            300 1 WARBLE                Make warble sound effect
          THENEnd of IF statement

        RESTORE-DIR                     Restore direction
        DEFAULT-INSTINCTS
      THEN      End of IF statement
;                                       End of definition
```

Add tasks to list

```
1 PRIORITY  ADD-TASK: LOW-BAT
2 PRIORITY  ADD-TASK: FRUSTRATED
3 PRIORITY  ADD-TASK: F-HIT
4 PRIORITY  ADD-TASK: RM-HIT
5 PRIORITY  ADD-TASK: LM-HIT
6 PRIORITY  ADD-TASK: BW-HIT
7 PRIORITY  ADD-TASK: BS-HIT
```

Create auto start word

```
: DEMO
   CALIBRATE
   MULITASKING
   FORWARD
   RIDE_OF_THE_VALKERIES
   BLOW_THE_MAN_DOWN!

   BEGIN
      FORWARD
   ?TERMINAL UNTIL ;

REMEMBER
AUTO-START: DEMO
```

Chapter III- WCL Whiskers Control Language

System Level

VARIABLE (--)	Allocates two bytes of memory. Creates new word in the dictionary that returns the address of the memory location. (65535 maximum value)

Used as:

VARIABLE COUNTER	Define location called: EVENT-COUNTER
1000 COUNTER NOW	set contents to a value of one thousand
COUNTER VALUE	get contents of COUNTER

REG:	Allocates one byte of RAM. Creates new word in the dictionary that returns the address of 1 byte memory location.

Used as:

REG: EVENT-COUNTER	Define location called: EVENT-COUNTER
1 EVENT-COUNTER SET	set contents to a value of one
EVENT-COUNTER GET	get contents of EVENT-COUNTER

SET (n addr --)	Set value of memory to number specified
GET (addr --)	Used with memory locations created by REG: Get the current value of memory
NOW (n --)	Used with memory locations created by VARIABLE Set value of memory to number specified
VALUE (n --)	Used with memory locations created by VARIABLE Get the current value of memory
ARRAY: (n --)	Allocates n bytes of RAM. Creates new word in the dictionary that returns the address of memory location.

Used as:

100 ARRAY: SOUNDS	create a 100 byte array named SOUNDS
0 SOUNDS SET	Store byte at offset 0
1 SOUNDS 1 + SET	Store byte at offset 1
2 SOUNDS 2 + SET	Store byte at offset 2

AUTO-START:

Cause a word to be automatically executed on power up.

Used as: AUTO-START: DEMO

Useful Words

BINARY	Change interactive number mode to binary. Useful in creating binary mask words to manipulate port address lines for example.
&H	Change base to HEX even in colon definitions. (Immediate word)
&D	Change base to DECIMAL even in colon definitions. (Immediate word)
&B	Change base to BINARY even in colon definitions. (Immediate word)

Used as:

```
BINARY 00010100          CONSTANT FWD DECIMAL
&B 00010100              CONSTANT FWD &D
&H AFFF                  CONSTANT END-MEMORY &D
&D 10                    CONSTANT MAX-HITS

: MOTORS-FWD ( -- )              ( turn both motors on; forward
  &B 00101000 &D  PORTA SET ;
```

ON (a --)	set flag on (-1), use only with words created by REG:
OFF (a --)	set flag off (0), use only with words created by REG:

Used as:

```
REG: COLLISION
COLLISON ON              ( turn collison flag on

: CHECK-COLLISION ( -- )

  COLLISION GET
     IF
        ENABLE
     ELSE
        DISABLE
     THEN
;

COLLISION OFF            ( turn collison flag off
```

DECREMENT (addr --) use only with words created by VARIABLE

Used as: FHIT-COUNTER DECREMENT

INCREMENT (addr --) use only with words created by VARIABLE

Used as: FHIT-COUNTER INCREMENT

% (n percent -- number) get percentage of number

Used as: 100 5 %
DISPLAY <cr>

Motor masks

A moto mask is a abbreviation of the robot's various movements (motor directions). You must store this value is a moto mask register to have an effect on the robot.

Used as: REV rMASK SET Robot will backup (instincts must be ENABLED).

LF	01100000	Left motor forward
LR	01000000	Left motor reverse
RF	00011000	Right motor forward
RR	00010000	Right motor reverse
FWD	01111000	Both motors forward
REV	01010000	Both motors reverse
ST	00000000	Both motors stop
PVR	01110000	Right Pivot
PVL	01011000	Left Pivot

LW	8 bit register for Left Whiskers hit
RW	8 bit register for Left Whiskers hit

Motor mask constants

01000000	LWMASK	Left Whiskers mask
00000010	RWMASK	Right Whisker mask
00000001	WMASK	Both Whiskers mask
00000011	L-OFF	Left motor off mask
10111111	L-ON	Left motor on mask
11101111	R-OFF	Right motor off mask
00010000	R-ON	Right motor on mask

Registers (used as REG GET | puts register value on stack)

rSPEED	Speed register
rLSPEED	Left speed register
rDIR	Direction
rF/B	Forward/backward register
rCHOICE	Choice register (used with LEFT, RIGHT words)
rMOT-MASK	Motor mask
SAV-rLSPEED	Save left speed
SAV-rRSPEED	Save right speed

Used as:
SAV-rLSPEED GET
20 LEFT SPEED

Motor variables (used as: vDEGREES VALUE)

vDEGREES	Degrees to turn
vLTON	Left motor on time
vLTOFF	Left motor off time
vRTON	Right motor on time
vRTOFF	Right motor off time

Motor control constants

cRIGHT	Returns 1
cLEFT	Returns 2
cFWD	Returns 3
cREV	Returns 4
cHIGH	Returns 5
cLOW	Retuurns 6

SENSORS	Show motor control registers, sensors, instinct level parameters

RIGHT	Sets choice register to cRIGHT
LEFT	Sets choice register to cLEFT
FORWARD	Sets rF/B register to cFWD and MASK register to FWD mask
BACKUP	Sets rF/B register to cREV and MASK register to REV mask
MAX-SPEED	Constant, returns maximum speed (100)
SPEED (speed --)	Sets motor speed

Used as:

```
40 LEFT SPEED
60 RIGHT SPEED
```

ADJUST-DEGREES (n --)	Adjusts delay in DEGREES command for tuning
DEGREES (degrees --)	Delaying word to convert degrees to time delay for turning commands

Used as:

```
LEFT PIVOT
34 DEGREES
STOP
```

STOP (--) Robot stops moving

FORWARD (--) Robot moves forward

STRAIGHT (--) Uses rCORRECTION value to make sure robot goes straight forward

BACKUP (--) Robot backs up

PIVOT (--) Pivot robot about its center, use LEFT or RIGHT first

TURN (--) Robot turns about left or right wheel, use LEFT or RIGHT first

REVERSE-DIR Reverse the direction of the motors FORWARD = REVERSE LEFT PIVOT = RIGHT PIVOT, etc

Used as:

```
FORWARD
LEFT TURN
50 DEGREES
BACKUP
REVERSE-DIR
RIGHT PIVOT
98 DEGREES
STOP
3 rCORRECTION SET
STRAIGHT FORWARD
```

SAVE-SPEEDS (--)	Save current speeds
RESTORE-SPEEDS (--)	Restore old speeds

Used as:

```
SAVE-SPEEDS
40 LEFT SPEED
40 RIGHT SPEED
BACKUP
2 SECS
LEFT PIVOT
90 DEGREES
RESTORE-SPEEDS
```

RAMP-UP (rate speed --)	Ramp both motors up to specified speed by rate
RAMP-DOWN (rate --)	Ramp both motors down by rate

Used as:

```
FORWARD
2 80 RAMP-UP
3 RAMP-DOWN
```

BEST-PIVOT (--)	Pivot away from closest side obstacles
CIRCLE (size --)	Make circle about a relative size
SQUARE (size --)	Make a square

Used as:

```
FORWARD
8 CIRCLE
2 MINUTES
6 SQUARE
```

SPIN (speed --)	Spin on center at specified speed

Used as:

```
30 %HIGHER-SPEED
DEFAULT-INSTINCTS
1 LEFT SPIN
30 SECS
STOP-INSTINCTS
FORWARD
```

BEARING (-- bearing)	Current direction using the software compass. This is a basic and relative number based on the difference between the pulse width modulation on times of the motors.

Analog to Digital Converter

ACQUIRE-A (--)	Acquire PE0 thru PE3 and place values in ADR1 thru ADR3.
ACQUIRE-B (--)	Acquire PE4 thru PE7 and place values in ADR1 thru ADR3.
BEYES-ON	Turn all LED's off, DISABLE first
BEYES-OFF	Turn all LED's on, DISABLE first

ADC MASKS

BPC-ON	All LED's on mask
BPC-OFF	All LED's off mask

Used as:

```
PORTA GET          Turn all on
BPC-ON OR
PORTA SET

PORTA GET          Turn all off
BPC-OFF AND
PORTA SET

DISABLE
BEYES-ON
BEYES-OFF
ENABLE
```

Light Sensors

BEYES-ON (--)	Turn all LED's on (note: must use DISABLE first)
BEYES-OFF (--)	Turn all LED's off (note: must use DISABLE first)
AMBIENT-LIGHT (-- off-light-level)	Returns ambient light level by averaging all light sensors (Range 0-255)

Microphone

1K Returns 1024

Commands

1K-BUFFER	A 1024 byte general purpose array.
DATA (addr # --)	Get value from memory array, takes address of array and offset (Range 0-255 for data value...8 bit)

Used as:

1K-BUFFER 3 DATA GET	Gets third value
2 1K-BUFFER 3 DATA SET	Sets third value to 2

vSAMPLES	Variable which holds number of samples to obtain
SAMPLES (samples --)	Number of samples to process

Used as:

```
10 SAMPLES
DIGITIZE
```

vRATE	Variable which holds the digitizing rate
RATE (rate --)	Sets the digitizing rate.

Used as:

10 RATE	Set rate to one tenth maximum
DIGITIZE	Aquire data

DIGITIZE (--)	Start digitizing process. Set RATE and samples first

Used as:

1 RATE	Set rate to maximum
256 SAMPLES	Number of samples
DIGITIZE	Acquire data
1K-BUFFER 256 DUMP	Display data

AVERAGE (-- average)	Return the average of samples stored in 1K-BUFFER

Used as:

```
500 SAMPLES
DIGITIZE
AVERAGE DISPLAY  <cr>
```

SOUND-LEVEL (-- level)	Returns the current ambient sound level

Used as:

```
SOUND-LEVEL  DISPLAY <cr>
```

STEP (step --)	Degrees to step for FIND-SOUND
FIND-SOUND (--)	Pivot by degrees set by step,
MAX-LEVEL (adr positions -- index)	Return index of maximum level in the array, address and number of positions are required
PIVOT-SOUND (--)	Pivot robot to maximum sound level

Instinct Level

DISABLE	Turn instincts (background task) off
ENABLE	Turn instincts (background task) off
vCOUNTER	Background counter, incremented each time instinct level is serviced. 16 bit.
PERIODS (delay --)	Delay processing for spefied periods. One period is the time between
DEFAULT-INSTINCTS	Set instincts to default masks

Default Masks:

rRF-MASK = PVL
rRSF-MASK = PVL
rRW-MASK = PVL
rLF-MASK = PVR
rLSF-MASK = PVR
rLW-MASK = PVR

STOP-INSTINCTS (--)	Stop on collision with any sensor

rRF-MASK = ST
rRSF-MASK = ST
rRW-MASK = ST
rLF-MASK = ST
rLSF-MASK = ST
rLW-MASK = ST

ALL-INSTINCTS (mask --)	Set all intincts to the same value

Used as:

```
ST ALL-INSTINCTS
FORWARD
BEGIN
  COLLIDED VALUE
    IF
      BEST-PIVOT
      CLEAR-SENSORS
```

```
            FORWARD
         THEN
    AGAIN
```

Mail Boxes

Note: use with SENSOR to get flag

LF-OBSTACLE	Left front obstacle flag
LSF-OBSTACLE	Left side front obstacle flag
RF-OBSTACLE	Right front obstacle flag
RSF-OBSTACLE	Right side front obstacle flag
F-OBSTACLE	Front summed obstacle flag
R-OBSTACLE	Right side/front summed obstacle flag
L-OBSTACLE	Left side front obstacle flag

rRF-MASK	Right Front Light Collision sensor motor mask
rRSF-MASK	Right Side Front Light Collision motor mask
rLF-MASK	Left Front Light Collision motor mask
rLSF-MASK	Left Side Front Light Collision motor mask
rLW-MASK	Left Whisker motor mask
rRW-MASK	Right Whisker motor mask

Obstacle Control

CALIBRATE (-)	Calibrate light sensors
TRIGGER-FACTOR (factor --)	Scales the calibrated values of the light sensors up and down, use before CALIBRATE.

Delaying Words

DELAY (time --)	A time delaying word using a null loop.
SECS (secs --)	Delay processing for given seconds
MINUTES (min --)	Delay processing for given minutes
PERIODS (time --)	A time delaying word that uses the instinct interupt for delaying execution of next word.

Used as:

```
2000 DELAY
1 MINUTES
35 SECS
500 PERIODS
```

Speaker Control

NOTE (Period duration --)	Output to speaker with frequency and duration as parameters

Used as:

18182 28 NOTE	A tone

Sound Effects

WAIL (range n --)

WARBLE (freg times --)

LASER (freg --)

TONES (freq times --)

UP-DOWN (freq steps --)

Used as:
40 10 WARBLE
40 LASER
60 2 TONES
40 20 UP-DOWN

Music

Give each note the period of the note as follows:

1/8 note
1/4 note
3/8 note
1/2 note
5/8 note
3/4 note
7/8 note
Whole note

Five octives of the musical scale have notes defined.

1st Scale: A A# B C C# D D# E F F# G G#

2nd Scale: 1A 1A# 1B 1C 1C# 1D 1D# 1E 1F 1F# 1G 1G#

3rd Scale: 2A 2A# 2B 2C 2C# 2D 2D# 2E 2F 2F# 2G 2G#

4th Scale: 3A 3A# 3B 3C 3C# 3D 3D# 3E 3F 3F# 3G 3G#

5th Scale: 4A 4A# 4B 4C 4C# 4D 4D# 4E 4F 4F# 4G 4G#

Note: Use the ~ (tilde) character for silent pauses

Sample Songs

```
: NOEL

1 2E  1 2D  2 2C  1 2D  1 2E  1 2F  4 2G  1 3A  1 3B  2
3C  2 3B 2 3A  4 2G  1 3A  1 3B  2 3C  2 3B  2 3A  2 2G  2 3A
2 3B 2 3C  2 2G  2 2F  4 2E  1 2E  1 2D  2 2C  1 2D  1 2E
1 2F 4 2G  1 3C 1 3B  4 3A  2 3A  6 2G  2 3C  2 3B  2 3A  2 2G
2 3A 2 3B  2 3C 2 2G  3 2F  4 2E  ;

: MUSIC-OF-THE-NIGHT

2 3A 2 2C 2 2G 2 2C 1 2F 1 2G 1 3A 1 3A# 2 2G 2 3C 2 3A 2 2C 2
2G 2 2C 1 2F 1 2G 1 3A 1 3A# 2 2G 2 3C 1 3D 1 3F 1 3F 1 3F 2 3G
1 3F 1 3E 1 3D 1 3F 1 3F 1 3F 2 3G 2 3F 1 3D 1 3F 1 3F 1 3F 1
3G 1 3F 1 3D 1 3A 4 3C 2 2G 2 3A

2 0 DO ( do this twice )

     3A 2 2C 2 2G 2 2C 1 2F 1 2G 1 3A 1 3A# 2 2G 2 3C 2 3A 2 2C 2
     2G 2 2C 1 2F 1 2G 1 3A 1 3A# 2 2G 2 3C 1 3D 1 3F 1 3F 1 3F 2 3G
     1 3F 1 3E 1 3D 1 3F 1 3F 1 3F 2 3G 1 3F 1 3E 1 3D 1 3F 1 3F 1
     3F 1 3G 1 3F 1 3D 1 3A 6 3C 2 3A 1 3A 1 2G 1 2G 1 3A 1 3A# 1 3C
     1 3A 1 2G 6 2F 1 3A 1 3C 2 3G 1 3F 1 3D#  1 3D 1 3C 1 3A#
     1 3C 2 3C 2 3A# 2 2G# 1 3C 1 3D#  2 3G# 1 3G 1 3F
     1 3F 1 3D#  1 3D#  1 3D 4 3D ~ ~ 1 3G 1 3G 1 3G 1
     3F 1 3E 1 3D 1 3C 1 3D 1 3C 6 4C 1 3E 1 3E 2 3E 1 3E 1 3E 1 3E
     1 3D 1 3E 1 3F 6 3E ~ ~
LOOP

2 3A  2 2C  2 2G  2 2C  1 2F  1 2G  1 3A  1 3A#  2 2G  2 3C
2 3A  2 2C  2 2G  2 2C  1 2F  1 2G  1 3A  1 3A#  2 2G  2 3C  1 3D  1 3F
13F  1 3F  1 3G  1 3F  1 3E  1 3D  1 3F  1 3F  1 3F  2 3G  1 3F  1 3E  1 3D
1 3F  1 3F  1 3F  1 3G  1 3F  1 3D  1 3A  6 3C  2 3A  1 3A  1 2G  1 2G
1 3A  1 3A#  1 3C  1 3A  1 2G  4 2F  2 2G  2 2C  1 2F  1 2G  1 3A  1 3A#
2 2G  2 3C  2 3A  2 2C  2 2G  2 2C  1 2F  1 2G  1 3A  1 3A#  2 2G  2 3C  1 3D  1 3F
13F  1 3F  1 3G  1 3F  1 3E  1 3D  1 3F  1 3F  1 3F  2 3G  1 3F  1 3E  1 3D
1 3F  1 3F  1 3F  1 3G  1 3F  1 3D  1 3A  6 3C  2 3A  1 3A  1 2G  1 2G
1 3A  1 3A#  1 3C  1 3A  1 2G  10 3F ;

: SAN-FRANCISCO

2 3A  2 3A#  2 3D  6 3C  2 3D  2 3E  2 3F  2 3D  4 2G  2 2G  2 2F#  2 2G
4 3D  2 3F  2 3E  1 3C  4 3A  2 3A  2 3A#  2 3B  1 3C  1 3A#  1 3A  1 3A#
6 3C  ~ ~  2 3D  1 3E  1 3D  1 3C  1 3D  4 3E  2 3E  3 3D  1 3E  4 3F  2 3G
3 3E  1 2G 5 3C 1 3D 1 3C 1 3A# 1 3A 1 3A# 2 3D 4 3C 2 3D 2 3E
2 3F 2 3D 4 2G 2 2G 2 2F# 2 2G 4 3E 2 3E 2 3F 2 3G 4 4A 2 4A
2 3G 2 4A 3 4B 2 4A 4 3F 2 3G 2 4A 2 3G 4 3D 2 3D 2 3C# 2 3D
3 4B 2 3D 3 4A 3 4A 4 3F ;
```

: CLEMENTINE

1 2G 2 2G 2 2D 1 3B 1 3B 2 3B 2 2G 1 2G 1 3B 2 3D 2 3D 1 3C
1 3B 3 3A 1 3A 1 3B 2 3C 2 3C 1 3B 1 3A 2 3B 2 2G 1 2G 1 3B
2 3A 2 2D 1 2F# 1 3A 3 2G 1 2G 1 2G 2 2G 2 2D 1 3B 1 3B 2 3B
2 2G 1 2G 1 3B 2 3D 2 3D 1 3C 1 3B 3 3A 1 3A 1 3B 2 3C 2 3C
1 3B 1 3A 2 3B 2 2G 1 2G 1 3B 2 3A 2 2D 1 2F# 1 3A 3 2G 1 2G
1 2G 2 2G 2 2D 1 3B 1 3B 2 3B 2 2G 1 2G 1 3B 2 3D 2 3D 1 3C
1 3B 3 3A 1 3A 1 3B 2 3C 2 3C 1 3B 1 3A 2 3B 2 2G 1 2G 1 3B
2 3A 2 2D 1 2F# 1 3A 3 2G 1 2G 1 2G 2 2G 2 2D 1 3B 1 3B 2 3B
2 2G 1 2G 1 3B 2 3D 2 3D 1 3C 1 3B 3 3A 1 3A 1 3B 2 3C 2 3C
1 3B 1 3A 2 3B 2 2G 1 2G 1 3B 2 3A 2 2D 1 2F# 1 3A 3 2G ;

Chapter III

Experiments

Note: Before starting any of the following experiments, turn your robot on, and immediately hit the [Enter] key several times. This will interupt the auto start sequence. Whiskers will respond with an *OK*.
Remember, Whiskers is case sensitive. You must type in the *words* exactly as shown in the manual. For example, rLF-OBSTACLE is correct, rlf-obstacle is not.

Lets start with Whiskers proped up by a book or something so his wheels don't touch the ground.

Instinct level:

Type the following commands and note Whiskers behavior.

FORWARD	Whiskers wheels turn going forward.
DISABLE	Whiskers instinct level is disabled. His lights stop Flashing, and words that depend on this level don't work, such as: FORWARD.
FORWARD	Nothing happens.
STOP	Nothing happens.
ENABLE	Whiskers instinct level is enabled. His lights start flashing, and words that depend on this level now work, such as: FORWARD.
FORWARD	Whiskers moves forward.
STOP	Whiskers stops.
LEFT PIVOT	Whiskers pivots about his center to the left.
RIGHT PIVOT	Whiskers pivots about his center to the right.
LEFT TURN	Whiskers turns about his left wheel.
RIGHT TURN	Whiskers turns about his right wheel.
BACKUP	Whiskers backs up.
STOP	Whiskers stops.
2 100 RAMP-UP	Whiskers ramps both motor speeds using rate 2 to speed 100
4 RAMP-DOWN	Whiskers ramps down from the current speed using rate 4.
50 LEFT SPEED	Whiskers changes his left motor speed to 50 percent
50 RIGHT SPEED	Whiskers changes his right motor speed to 50 percent

LEFT PIVOT 10 DEGREES STOP	Whiskers pivots 10 degrees and stops.

10 ADJUST-DEGREES	For tuning degrees command.
LEFT PIVOT 10 DEGREES STOP	Whiskers pivots 10 degrees and stops but goes further than before.

FORWARD	Go forward
17 LEFT SPEED	Change left speed
17 RIGHT SPEED	Change right speed
SAVE-SPEEDS	Save current speeds
100 LEFT SPEED	Faster speeds
100 RIGHT SPEED	Faster speeds
RESTORE-SPEEDS	Restore current speeds

WORDS

```
3CAC PROM-ENDS          3C72 END-PROM                3C5B REG:              3C47 VARIABLE
3BD3 BOOT-UP            3B69 NEW-SYSTEM              3A8D WANDER            3993 EXPLORE
397D FORGET             38EB CHECK-KEYBOARD          388C RECALL            3860 REMEMBER
F57C CLEAR-STACK        37F9 CHECK-SUM               37DF AUTO-START:       37B7 BOOT:
377B INIT-ROBOT         3753 RECALL-SYS              3740 LAST-PROM         371C SAVE-SYS
36DA R-SYS              364E SYS                     363C CANCEL            362B S-VARS
361A U-VARS             3609 vBOOT-WORD              35F7 CLD/WRM           35A0
RIDE_OF_THE_VALKERIES
3550 WHY_AM_I_BLUE      347D BLOW_THE_MAN_DOWN!      3450 ROCK_IT_AWAY!     30AE CLEMENTINE
2F26 SAN-FRANCISCO      2F04 ~                       2ED4 UP-DOWN           2EA6 TONES
2E7A LASER              2E48 WARBLE                  2DE9 CYCLES            2DD8 sTIME
2DC8 dTIME              29CA MUSIC                   29A1 INIT-RCL          296C SPIN
2958 STOP-INSTINCTS     28FF DEFAULT-INSTINCTS       28E1 WAIT-TIME         28AE %HIGHER-TRIGGER
2894 vWAIT-TIME         2859 SQUARE                  27FC CIRCLE            27CD TURN
2782 PIVOT-SOUND        273E MAX-LEVEL               26C5 FIND-SOUND        26AE STEP
269F vSTEP              268F LEVELS                  2677 ALL-INSTINCTS     264B BEST-PIVOT
2618 PIVOT              25DE STRAIGHT                1AAA SHOW-SENSORS      258D RAMP-DOWN
2551 RAMP-UP            252F RESTORE-SPEEDS          2508 SAVE-SPEEDS       24D2 START-ROBOT
23FE CALIBRATE          23D3 TRIGGER-TABLE           23AA BACK              23AA BACKUP
2391 FORWARD            2377 STOP                    2338 DEGREES           231C BEARING
2308 CLEAR-COMPASS      22EE ADJUST-DEGREES          22D7 360               22C6 MINUTES
22A1 SECS               2292 SAV-rRSPEED             227C SAV-rLSPEED       225C ACTION
2245 %SPEED             2231 %                       21DB SPEED             21CD RCL
2178 VEC-INIT           2167 VEC-TABLE-END           2117 VEC-TABLE         20A7 INSTINCT
2073 COMPASS,           2024 PW-MOD                  1FEB WHISKERS          1F7B STALLS
```

This is a sample of the many words that Whiskers knows. The numbers by the names are actual memory locations at which these words are defined. Notice a few words such as:

1 A	First octive musical note A, 1/8 note.
7 1B	Second octive musical note B, 7/8 note.
2 2C	Third octive musical note C, 1/4 note.
6 3D	Fourth octive musical note D, 3/4 note.
5 4E#	Fifth octive musical note E sharp, 5/8 note.

40 LASER	Make a laser sound. Used as: 70 LASER, 40 LASER,etc.

Put your hand in front of each light sensor. You will see Whiskers try to avoid you hand.
Try changing the value of the trigger level:

10 TRIGGER-FACTOR
CALIBRATE
SENSORS

Note that Whiskers becomes very nervious. You have set this level so low that the ambient light noise is detected. Try the hand experiment again.

50 TRIGGER-FACTOR
CALIBRATE
SENSORS

Note that Whiskers stops being nervous. You can keep increasing the trigger level and see how his range is reduced. Try the hand experiment each time.

Try each command below and conduct the hand experiment with the optical collision sensors.

FORWARD

BACKUP

LEFT PIVOT

RIGHT PIVOT

LEFT TURN

RIGHT TURN

STOP

SQUARE

FORWARD
100 LEFT SPEED
100 RIGHT SPEED

Try the hand experiment notice that Whiskers does not slow down to think about avioding you hand. The direction that you want to go is overidden until your hand is no longer a threat. This is his instinct level working.

14 RIGHT SPEED	Whiskers would go in a circle. This shows his pulse width modulation motor control. You can try different speeds between 1 and 100 with this software.
DISABLE	Disables his instinct level
FORWARD	Nothing happens, you have shut off his instinct level and pulse width modulator.
ENABLE	The robot goes forward. You have turned on his instinct level and pulse width modulator.
STOP-INSTINCTS	Stop on any obstacle
FORWARD	try the hand experiment, Whiskers now stops when he sees something

REV rRSF-MASK SET	Change Whiskers behavior on the right side sensors to backup. Put your hand in front of the right side sensor. His wheels backup as long as you hold your hand there.
ST rRSF-MASK SET	Change Whiskers behavior on the right side sensors to backup. Put your hand in front of the right side sensor. His wheels stop as long as you hold your hand there.
DEFAULT-INSTINCTS	Set behaviors back to default tug on each whisker, see him pivot away?
CLEAR-SENSORS	Clear all sensor flags
rLW-OBSTACLE GET DISPLAY	Get left side light sensor flag=0
	push on left whisker
rLW-OBSTACLE GET DISPLAY	Get left side light sensor flag=1
RIDE_OF_THE_VALKERIES	Sings the song with a raspy voice.
	you can hear the instinct level working
DISABLE	disable instincts
RIDE_OF_THE_VALKERIES	sings the song with a smoother voice.
1 2A 4 2B 4 2C	Sing a,b,c notes on the second octive
1 3C 8 3C	try using from 1 to 8 as parameters.
70 LASER	make a laser firing sound, try different
100 LASER	parameters
50 50 WARBLE	

```
    DISABLE                         Lets' try out his hearing
    10 RATE                         Sampling rate, 1 is the highest
    256 SAMPLES                     256 samples of sound
                                    before excuting the DIGITIZE  command, whistle or make a
                                    continous tone into the microphone at the front of the robot.
    DIGITIZE                        Stops instinct level and digitizes sound

    1K-BUFFER 256 DUMP              display 256 bytes of memory from 1K buffer
```

```
0  2  219  0  0  144  0  0  140  0  0  255  0  0  255  255  86  255  255  9
255  255  0  252  255  0  5  236  0  1  111  0  0  130  0  0  255  189  0  25
255  108  255  255  70  255  255  0  255  255  0  2  233  0  0  92  0  0  248
  0  255  146  0  255  221  42  255  255  105  255  255  1  255  255  0  160
 0  15  194  0  0  89  0  0  138  0  0  255  0  0  255  199  0  255  226  5
255  255  100  255  255  6  255  255  0  255  255  0  142  255  1  21  192  2
  172  0  0  113  0  0  255  0  0  255  92  0  255  203  10  255  223  65  25
255  81  255  234  40  255  255  0  204  255  1  110  230  4  0  242  0  0  5
0  0  93  0  0  255  0  0  255  38  0  255  196  0  255  179  101  255  255
?  255  255  28  255  245  2  255  255  0  255  255  1  244  250  1  2  255  1
0  212  1  0  159  0  0  187  0  0  255  0  0  255  0  0  255  196  90  255
135  255  230  1  255  255  0  255  255  1  106  231  14  0  174  0  0  159
  0  255  0  0  255  218  0  255  229  87  255  220  4  255  244  0  200  255
  0  160  0  OK
```

```
ENABLE
```

To see how to add your own words (commands) try the following:

```
: FEET
  DO
    1000 PERIODS       ( adjust as neccessary, speed dependent
  LOOP
  STOP
;

: ON-LEFT-WHISKER
  CLEAR-SENSORS
  BEGIN

    rLW-OBSTACLE SENSOR

      IF
        50 LASER
        CLEAR-SENSORS
      THEN
  AGAIN
;

: TEST
  FORWARD 2 FEET
  ON-LEFT-WHISKER
;
```

```
REMEMBER
AUTO-START: TEST
```

Put the robot on the floor. Turn the robot off then on again. He will automatically run the TEST word you just defined. After he goes one foot, try touching his left whisker. He should make a laser sound. There are many more words that Whiskers knows, look in the manual and the source code provided for more.

Chapter IV

The Forth Language

Table of Contents

Arithmetic

*	(w1 w2 --- w3) Multiplies w2 by w1 and leaves the product w3 onto stack.
*/	(n1 n2 n3 --- n4) Multiplies n1 by n2 and divides the product by n3. The quotient, n4 is placed on the stack.
*/MOD	(n1 n2 n3 --- n4 n5) n1 is multiplied by n2 producing a product which divided by n3. The remainder, n4 and the quotient, n5, are then placed on the stack.
+	(w1 w2 --- w3) Adds w2 and w1 then leaves the sum, w3 on the stack.
-	(w1 w2 --- w3) Subtracts w2 from w1 and leaves the result, w3 on the stack.
/	(n1 n2 --- n3) Divides n1 by n2 and leaves the quotient n3 on the stack.
/MOD	(n1 n2 --- n3 n4) Divides n1 by n2 then leaves on the stack the remainder n3 and the quotient n4.
1+	(w1 --- w2) Adds 1 to w1 then leaves the sum, w2 on the stack.
1-	(w1 --- w2) Subtract 1 from w1 then leaves the difference, w2 on the stack.
2*	(w1 --- w2) Multiplies w1 by 2 to give w2.
2+	(w1 --- w2) Adds two to w1 and leaves the sum, w2 on the stack.
2-	(w1 --- w2) Subtracts two from w1 and leaves the result, w2 on the stack.
2/	(n1 --- n2) Divides n1 by 2, giving n2 as the result.
ABS	(n --- u) Leaves on the stack the absolute value, u of n.
D+	(wd1 wd2 --- wd3) Adds wd1 and wd2 and leaves the result, wd3 on stack.
D-	(wd1 wd2 --- wd3) Subtracts wd2 from wd1 and returns the difference wd3.
D2/	(d1 --- d2) Divides d1 by 2 and gives quotient d2.
DABS	(d --- ud) Returns the absolute value of d as ud.
DMAX	(d1 d2 --- d3) Returns d3 as the greater of d1 or d2.
DMIN	(d1 d2 --- d3)

		Returns d3 as the lesser of d1 or d2.
DNEGATE	(d1 --- d2)	Leaves the two's complement d2 of d1.
MAX	(n1 n2 --- n3)	Leaves the greater of n1 and n2 as n3.
MIN	(n1 n2 --- n3)	Leaves the lesser of n1 and n2 as n3.
MOD	(n1 n2 --- n3)	Divides n1 by n2 and leaves the remainder n3.
NEGATE	(n1 --- n2)	Leaves the two's complement n2 of n1.
S->D	(n --- d)	Sign extend single number to double number.
UM*	(u1 u2 --- ud)	Multiplies u1 and u2 returning the double length product ud.
UM/MOD	(ud u1 --- u2 u3)	Divides the double length unsigned number ud by u1 and returns the single length remainder u2 and the single length quotient u3.

Comparison

0<	(n --- flag)	Leaves a true flag if n is less than zero.
0=	(w --- flag)	Leaves a true flag if w is equal to zero.
0>	(n --- flag)	Leaves a true flag if n is greater than zero.
<	(n1 n2 --- flag)	Leaves a true flag on stack if n1 is less than n2.
=	(w1 w2 --- flag)	Returns a true flag if w1 is equal to w2.
>	(n1 n2 --- flag)	Returns a true flag if n1 is greater than n2.
D0=	(wd --- flag)	Returns a true flag if wd is equal to zero.
D<	(d1 d2 --- flag)	Leaves a true flag if d1 is less than d2; other-wise leaves a false flag.
D=	(wd1 wd2 --- flag)	Returns a true flag if wd1 is equal to wd2.
DU<	(ud1 ud2 --- flag)	Returns a true flag if ud1 is less than ud2.
U<	(u1 u2 --- flag)	Returns a true flag if u1 is less then u2.

Compiler

(	(---)	Starts a comment input. Comment is ended by a).
>BODY	addr1 --- addr2)	Leaves on the stack the parameter field address, addr2 of a given field address, addr1.
COMPILE	(---)	Copies the compilation address of the next non-immediate word following COMPILE.
EXECUTE	(addr ---)	Executes the definition found at addr.
IMMEDIATE	(---)	Marks the most recently created dictionary entry as a word that will be executed immediately even if FORTH is in compile mode.
LITERAL	(16b ---)	Compile a system dependent operation so that when later executed, 16b will be left on the stack.
RECURSE	(---)	Compile the compilation address of definition currently being defined.
STATE	(--- addr)	Returns the address of the user variable that contains a value defining the compilation state.
[	(---)	Places the system into interpret state to execute non-immediate word/s during compilation.
[']	(--- addr), (---)	Returns and compiles the code field address of a in a colon-definition.

[COMPILE]	(---) Causes an immediate word to be compiled.
]	(---) Places the system into compilation state.] places a non-zero value into the user variable STATE.

Control

+LOOP	(n ---), (sys ---) (compiling) Increments the DO LOOP index by n.
<MARK	--- addr) Leaves current dictionary location to be resolved by <RESOLVE .
<RESOLVE	(addr ---) Compiles branch offset to location previously left by <MARK .
>MARK	--- addr) Compiles zero in place of forward branch offset marks it for future resolve.
>RESOLVE	(addr ---) Corrects branch offset previously compiled by mark to current dictionary location.
AGAIN	(---), (sys ---) (compiling) Ends BEGIN loop.
BEGIN	(---), (--- sys) (compiling) Marks the start of a loop.
DO	(w1 w2 ---), (--- sys) (compiling) Repeats execution of words between DO LOOPs and DO +LOOPs, the number of times is specified by the limit from w2 to w1.
ELSE	(---), (sys1 --- sys2) (compiling) Allows execution of words between IF and ELSE if the flag is true, otherwise, it forces execution of words after ELSE.
END	(sys ---) Performs the same function as UNTIL. See UNTIL .
I	(--- w) Places the loop index onto the stack.
IF	(flag ---), (--- sys) (compiling) Allows a program to branch on condition.
J	(--- w) Returns the index of the next outer loop.
K	(--- w) Returns the index of the second outer loop in nested do loops.
LEAVE	(---) Forces termination of a DO LOOP.
LOOP	(----),(sys ---) (compiling) Defines the end point of a do-loop.
REPEAT	(---), (sys ---) (compiling) Terminates a BEGIN...WHILE...REPEAT loop.
THEN	(---), (sys ---) (compiling)

	Marks the end of a conditional branch or marks where execution will continue relative to a corresponding IF or ELSE .
UNTIL	(flag ---), (sys ---) (compiling) Marks the end of an indefinite loop.
WHILE	(flag ---), (sys1 --- sys2) (compiling) Decides the continuation or termination of a ...WHILE...REPEAT loop.

Definition

2CONSTANT	(32b ---) Creates a double length constant for a <name>. When <name> is executed, 32b is left on the stack.
2VARIABLE	(---) Creates double-length variable for <name>. When <name> is executed, its parameter field address is placed on the stack.
:	(--- sys) Starts the definition of a word. Definition is terminated by a ; .
:CASE	(n ---), (--- sys) (compiling) Creates a dictionary entry for <name> in current and sets the compile mode.
;	(sys ---) Terminates a colon-definition.
;CODE	(---), (sys1 --- sys2) (compiling) Terminates a defining-word. May only be used in mode.
<BUILDS	(---) Creates a new dictionary entry for <name>
CODE	(--- sys) Creates an assembler definition.
CODE-SUB	(--- sys) Creates an assembler definition subroutine.
CONSTANT	(16b ---) Creates a dictionary entry for <name>.
CREATE	(---) Creates a dictionary entry for <name>.

DOES>	(--- addr), (---) (compiling)	Marks the termination of the defining part of the defining word <name> and begins the definition of run-time action for words that will later be by <name>.
END-CODE	(sys ---)	Terminates an assembler definition.
USER	(n ---)	Create a user variable.
VARIABLE	(---)	Creates a single length variable.

Dictionary

'	(--- addr)	Returns <name>'s compilation address, addr
,	(16b ---)	Reserves 16b of space in the dictionary.
ALLOT	(w ---)	Reserves w bytes of dictionary space.
AUTOSTART	(addr ---)	Prepare autostart vector at addr which will cause name> to be executed upon reset. Note: addr must be on a 1K byte boundary.
C,	(16b ---)	Stores the least significant byte of 16b into a byte at the next available dictionary location.
CFA	(pfa --- cfa)	Alter parameter field pointer address to code field address.
DP	(--- addr)	Put Dictionary Pointer address on stack.
FORGET	(---)	Deletes <name> from the dictionary.
H/C	(addr ---)	Separate heads and codes portions of definition to different place in memory.
HERE	(--- addr)	Leaves the address of the next variable dictionary location
HWORD	(---)	Moves codes portion of last defined word from the codes memory to the heads memory.
LATEST	(--- nfa)	Leaves name field address (nfa) of top word in CURRENT.

LFA	(pfaptr --- lfa)	Alter parameter field pointer address to link field address.
NFA	(pfaptr - nfa)	Alter parameter field pointer address to name field address.
PAD	(--- addr)	Puts onto stack the starting address in memory of scratchpad.
PFAPTR	(nfa --- pfaptr)	Alter name field address to parameter field pointer address.
TASK	(---)	A dictionary marker null word.
TRAVERSE	(addr n --- addr)	Adjust addr positively or negatively until contents of addr is greater then $7F.
UNDO	(---)	Forget the latest definition regardless of smudge condition.

Format

#	(+d1 --- +d2)	+d1 is divided by BASE and the quotient +d2 is placed onto the stack. The remainder is converted to an ASCII character and appended to the output string toward lower memory addresses.
#>	(32b --- addr +n)	Terminates formatted (or pictured) output string (ready for TYPE).
#S	(+d --- 0 0)	Converts all digits of an entire number into string.
-TRAILING	(addr +n1 --- addr +n2)	Counts +n1 characters starting at addr and subtracts 1 from the count when a blank is encountered. Leaves on the stack the final string count, n2 and addr.
<#	(---)	Starts a formatted (pictured) numeric output. Terminated by #> .
COUNT	(addr1 --- addr2 +n)	Leaves the address, addr2 and the character count +n of text beginning at addr1.
DPL	(--- addr)	Returns the address of the user variable containing the number of places after the fractional point for input conversion.
FLD	(--- addr)	Returns the address of the user variable which contains the value of the field length reserved for a number during output conversion.
HOLD	(char ---)	Inserts character into a pictured numeric output string.
NUMBER	(addr --- d)	Converts the counted string at addr to d according to the value of BASE .
SIGN	(n ---)	Appends an ASCII " - " (minus sign) to the start of a pictured numeric output string if n is negative.

Input/Output

#TIB	(--- addr) Returns the address of the user variable that holds the number of characters input.
."	(---) Displays the characters following it up to the delimiter " .
.(	(---) Displays string following .(delimited by) .
.R	(n +n ----) Displays the value n right justified in a field +n characters wide according to the value of BASE.
.S	(---) Display stack contents without modifying the stack.
.	(n ---) Removes n from the top of stack and displays it.
?	(addr ---) Displays the contents of addr.
?TERMINAL	(--- flag) True if any key is depressed.
ABORT"	(----) If flag is true, message that follows " is displayed and the ABORT function is performed. If flag is false, the flag is dropped and execution continues.
BL	(--- n) Puts the ASCII code for a space (decimal 32) on the stack.
C/L	(--- n) Maximum number of characters per line
CR	(---) Generates a carriage return and line feed.
D.	(d ---) Displays the value of d.
D.R	(d +n ---) Displays the value of d right justified in a field +n characters wide.

DUMP	(addr u ---)	Displays u bytes of memory starting at addr.
EMIT	(16b ---)	Displays the ASCII equivalent of 16b onto the screen.
EXPECT	(addr +n ---)	Stores up to +n characters into memory beginning at addr.
ID.	(nfa ---)	Print <name> given name field address (NFA).
KEY	(--- 16b)	Pauses to wait for a key to be pressed and then places the ASCII value of the key (n) on the stack.
QUERY	(---)	Stores input characters into text input buffer.
SPACE	(---)	Sends a space (blank) to the current output device.
SPACES	(+n ---)	Sends +n spaces (blanks) to the current output device.
SPAN	(--- addr)	Returns the address of the user variable that contains the count of characters received and stored by the most recent execution of EXPECT .
TIB	(--- addr)	Returns the address of the start of the text-input buffer.
TYPE	(addr +n ---)	Displays a string of +n characters starting with the character at addr.
U.	(u ---)	Displays the unsigned value of u followed by a space.
U.R	(u +n ---)	Displays the value of u right justified in a field +n characters wide according to the value of BASE.

Interpreter

;S	(---)	Stop interpretation.
>IN	(--- addr)	Leaves the address of the user variable >IN which contains the number of bytes from the beginning of the input stream at any particular moment during interpretation.
EXIT	(---)	Causes execution to leave the current word and go back to where the word was called from.
FIND	(addr1 --- addr2 n)	Obtains an address of counted strings, addr1 from the stack. Searches the dictionary for the string.

INTERPRET	(---)	Begins text interpretation at the character indexed by the contents of >IN relative to the block number contained in BLK, continuing until the input stream is exhausted.
QUIT	(---)	Clears the return stack, stops compilation and returns control to current input device.
WORD	(char --- addr)	Generates a counted string until an ASCII code, char is encountered or the input stream is exhausted. Returns addr which is the beginning address of where the counted string are stored.

Logic

AND	(16b1 16b2 --- 16b3)	Leaves the bitwise logical AND of 16b1 and 16b2 as 16b3.
NOT	(16b1 --- 16b2)	Leaves the one's complement 16b2 of 16b1.
OR	(16b1 16b2 --- 16b3)	Leaves the exclusive-or 16b3 of 16b1 an 16b2.
XOR	(16b1 16b2 --- 16b3)	Performs a bit-by-bit exclusive or of 16b1with 16b2 to give 16b3.

System

(LINE)	(n1 n2 --- addr count)	Virtual storage line primitive.
-->	(---)	Immediately transfers interpretation to the start of the next sequential block.
.LINE	(n1 n2 ---)	Display line of text from mass storage.
>L	(n ---)	Place the <text> on line n of the current screen as designated by SCR.
B/BUF	(--- 1024)	Returns a constant 1024 which is the number of bytes per block buffer.
BLK	(--- addr)	Leaves the address of the user variable containing the number of the block that is currently being interpreted.
BLOCK	(u --- addr)	Leaves the address of the block buffer containing block u.
BUFFER	(u --- addr)	Assigns the block buffer and its address to block u. Leaves the address of the block buffer on the stack.
EMPTY-BUFFERS	(---)	Removes the update status and removes assignments of all blocks to buffers.
FIRST	(--- n)	Leaves address of first block buffer.
FLUSH	(---)	

Performs the functions of the SAVE-BUFFERS then unassign all block buffers.

INDEX	(u1 u2 ---) Displays the top line from each block in a range of blocks starting at block u1 to u2.
LIMIT	(--- n) Top of memory
LIST	(u ---) Lists the block whose number is u. SCR is set to u.
LOAD	(u ---) Begins interpretation of block u.
OFFSET	(--- addr) Returns the address of the user variable that contains a block offset to mass storage.
SAVE-BUFFERS	(---) Copies the contents of all block buffers marked as UPDATEd to their corresponding mass-storage blocks.
SCR	(--- addr) Returns the address of the user variable containing the number of block most recently listed.
THRU 1	(u1 u2 ---) Load blocks from u1 through u2.
TRIAD	(u16 ---) Lists three screens to the output device. One of screens listed is specified by the user.
UPDATE	(---) Marks the block buffer as having been modified and as ready to be saved to mass storage.

Memory

!	(16b addr ---) Stores 16 at addr.
+!	(w1 addr ---) Adds w1 to the value at addr then stores the sum addr replacing its previous value.
1+!	(addr ---) Adds one to the value at addr and stores the at addr.
1-!	(addr ---) Subtracts one from the value at addr and stores result at addr.
2!	(32b addr ---) Stores 32b at addr.
2@	(addr --- 32b) Returns 32b from addr.
@	(addr --- 16b) Replaces addr with its 16b contents on top of the stack.
BLANKS	(addr u ---) Sets u bytes of memory beginning at addr to the ASCII code for space (decimal 32).
C!	(16b addr ---) Stores the least significant byte of 16b into addr.
C@	(addr --- 8b) Fetches the 8b contents from addr.
CMOVE	(addr1 addr2 u ---) Moves towards high memory the u bytes at addresses addr1 and addr2.
CMOVE>	(addr1 addr2 u ---) Moves u bytes beginning at addr1 to addr2.
EEC!	(16b addr ---) Stores the least significant byte of 16b into addr in EEPROM.
EEMOVE	(addr1 addr2 u ---) Moves the u bytes at addresses addr1 and addr2.
EEWORD	(---) Moves code of last defined word from the codes memory of the EEPROM memory.
ERASE	(addr u ---) Sets u bytes of memory to zero, beginning at addr.
FILL	(addr u 8b ---) Fills u bytes, beginning at addr, with byte pattern 8b.

Numeric

BASE	(--- addr)	Leaves the address of the user variable containing the numeric conversion radix.
CONVERT	(+d1 addr1 --- +d2 addr2)	Converts an input string into a number.
DECIMAL	(---)	Sets the input-output numeric conversion base to ten.
HEX	(---)	Sets the numeric input-output conversion to base 16.

Operating System

ABORT	(flag ---)	Clears the data stack and performs the function of QUIT .
COLD	(---)	Cold starts FORTH.
ATO4	(--- N)	Returns address of subroutine call to high level word as indicated in D register.

Primitive

?BRANCH	(flag ---)	Compiles a conditional branch operation.
BRANCH	(---)	Compiles an unconditional branch operation.

Stack

-ROLL	(n ---)	Removes the value on the top of stack and inserts it to the nth place from the top of stack.
2DROP	(32b ---)	Removes 32b from the stack.
2DUP	(32b --- 32b 32b)	Duplicates 32b.
2OVER	(32b1 32b2 --- 32b1 32b2 32b3)	32b3 is a copy of 32b1
2ROT	(32b1 32b2 32b3 --- 32b2 32b3 32b1)	Rotates 32b1 to the top of the stack.
2SWAP	(32b1 32b2 --- 32b2 32b1)	Swaps 32b1 and 32b2 on the stack.
><	(8b1/8b2 --- 8b2/8b1)	Swaps the upper and lower bytes of the value on the stack.
>R	(16b ---)	Removes 16b from user stack and place it onto return stack.
?DUP	(16b --- 16b 16b), (0 --- 0)	

		Duplicates 16b if it is a non-zero.
DEPTH	(--- +n)	Returns count +n of numbers on the data stack.
DROP	(16b ---)	Removes 16b from the data stack.
DUP	(16b --- 16b 16b)	Duplicates 16b.
OVER	(16b1 16b2 --- 16b1 16b2 16b3)	16b3 is a copy of 16b1.
PICK	(+n --- 16b)	Copies the data stack's +nth item onto the top.
R>	(--- 16b)	16b is removed from the return stack and placed onto the data stack.
R@	(--- 16b)	16b is a copy of the top of the return stack.
ROLL	(+n ---)	Removes the stack's nth item and places it onto the top of stack.
ROT	(16b1 16b2 16b3 --- 16b2 16b3 16b1)	Rotates 16b1 to the top of the stack.
S0	(--- addr)	Returns the address of the variable containing the initial value of the bottom of the stack.
SP@	(--- addr)	addr is the address of the top of the parameter stack just before SP@ was executed.
SWAP	(16b1 16b2 --- 16b2 16b1)	Exchanges positions of the top two items of the stack.

Vocabulary

ASSEMBLER	(---)	Replaces the first vocabulary in the search order with the assembler vocabulary.
CONTEXT	(--- addr)	Returns the address of a user variable that deter mines the vocabulary to be searched first in the dictionary.
CURRENT	(--- addr)	Returns the address of the user variable specifying the vocabulary into which new word definitions will be entered.
DEFINITIONS	(---)	Specify the vocabulary into which new definitions are to be added.
FORTH	(---)	Replaces the first vocabulary in the search order with FORTH.
FORTH-83	(---)	Initializes FORTH-83 into the system.
VOCABULARY	(---)	Creates a vocabulary word.

WORDS	(---) Lists all the words in the CURRENT vocabulary.

Chapter V

Hardware and Software Details

FORTH 68HC11 MEMORY MAP

DEC	HEX	NAME	VALUE	
	$0000	W	0	
	$0001	"		
	$0002	IP	0	WORD PTR AFTER BOOT PAT.
	$0003	"		
	$0004	UP	$0006	UAREA
	$0005	"		
	$0006	DNLINK	0	
	$0007	"		
	$0008	UPLINK	0	
	$0009	"		
	$000A	PRIORITY	0	
	$000B	"		
	$000C	RPSAVE	0	
	$000D	"		
	$000E	R0	STACKINIT	
	$000F	"		
	$0010	S0	BOS	
	$0011	"		
	$0012	KEY-BC-PTR	DEFKEY	
	$0013	"		
	$0014	EMIT-BC-PTR	DEFOUT	
	$0015	"		
	$0016	UKEY	KEYSUB+2	
	$0017	"		
	$0018	UEMIT	EMITSUB+2	
	$0019	"		
	$001A	U?TERMINAL	QTSUB+2	
	$001B	"		
	$001C	TIB	TIBX	
	$001D	"		
	$001E	UC/L		

System Memory Map

B000	PORTA

PA0 -	Left Whisker
PA1 -	Right Whisker
PA2	not used
PA3 -	Left motor direction
PA4 -	Left motor enable
PA5	Right motor direction
PA6 -	Right motor enable
PA7 -	Led control

B002	PIOC
B003	PORTC (not available; used to address memory)
B004	PORTB (not available; used to address memory)
B005	PORLCL(not used)
B007	DDRC (not used)
B008	PORTD (serial port)

PD0 -	Rxd; recieve data RS232
PD1 -	TxD; transmit data RS232
PD2 -	MISO; network output
PD3 -	MOSI; network input
PD4 -	SCK; network clock
PD5 -	/SS; network select device, speaker output
PD6	AS
PD7 -	R/W; read/write

B009	DDRD (data direction register for port D)
B00A	PORTE (ADC inputs)

PE0 -	Left front photo detector
PE1 -	Left side photo detector
PE2 -	Right front photo detector
PE3 -	Right side photo detector
PE4 -	Battery
PE5 -	Left motor monitor
PE6 -	Right motor monitor
PE7 -	Audio microphone

B00B	CFORC	(oc1 oc2 oc3 oc4 oc5 - - - force output compare 8 bit)
B00C	OC1M	(oc1 oc2 oc3 oc4 oc5 - - - enable port A pins 8 bit)
B00D	OC1D	(oc1 oc2 oc3 oc4 oc5 - - - pin level high = 1 8 bit)
B00E	TCNT	(main timer read only 16 bit)
B010	TIC1	(input compare 1 timer 16 bit)
B012	TIC2	(input compare 2 timer 16 bit)
B014	TIC3	(input compare 3 timer 16 bit)
B016	OC1	(output compare 1 timer 16 bit)
B018	OC2	(output compare 2 timer 16 bit)
B01A	OC3	(output compare 3 timer 16 bit)
B01C	OC4	(output compare 4 timer 16 bit)
B01E	OC5	(output compare 5 timer 16 bit)

B020	TCTL1	(om2 ol2 om3 ol3 om4 ol4 om5 ol5 timer contro 8 bit)
B021	TCTL2	(- - edg1b edg1a edg2b edg2a edg3b edg3a 8 bit)
B022	TMSK1	(oc1I oc2I oc3I oc4I oc5I ic1I ic2I ic3I IRQ enable 8 bit)
B023	TFLG1	(oc1f oc2f oc3f oc4f oc5f ic1f ic2f ic3f IRQ f 1=clr 8 bit)
B024	TMSK2	(toI rtiI paovI raiI - - pr1 pr0 overflow IRQ enable 8 bit)
B025	TFLG2	(tof rtif paovf paif - - - - timer ovrflow=1 tof=1 set 8 bit)

B026	PACTL (ddrA7 paen pamod pedge - - rtr1 rtr0 pulse acc ctrl bit)
B027	PACNT
B028	SPCR
B029	SPSR
B02A	SPDR
B02B	BAUD
B02C	SCCR1
B02D	SCCR2
B02E	SCSR
B02F	SCDR
B030	ADCTL
B031	ADR1
B032	ADR2
B033	ADR3
B034	ADR4

Robot Control Board

The WHISKERS control board is the very heart of this little robot. It has been specially designed to provide the required I/O to allow almost unlimited programing capability. You use the WHISKERS to carry out your own programing ideas and it becomes the very extension of your intellect, and his.

Lets review where all major plugs and components are located.

J1 36 pin expansion connector

Pin	Signal
J1-1	VBATSW
J1-2	VBATSW
J1-3	GND
J1-4	GND
J1-5	VCC
J1-6	VCC
J1-7	/RESET
J1-8	ECLOCK
J1-9	NC
J1-10	/XIRQ
J1-11	AS
J1-12	/R/W
J1-13	A14
J1-14	A15
J1-15	A12
J1-16	A13
J1-17	A10
J1-18	A11
J1-19	A8
J1-20	A9
J1-21	SCK
J1-22	/SS
J1-23	MISO
J1-24	MOSI
J1-25	D1
J1-26	D0
J1-27	D3
J1-28	D2
J1-29	D5
J1-30	D4
J1-31	D7
J1-32	D6
J1-33	NC
J1-34	/MEMDIS
J1-35	NC
J1-36	/IRQ

RS232 Serial connector

J2-1	GND
J2-2	RDY\
J2-3	RXD\
J2-4	TXD\

J4 two pins to the microphone

J5 a 10 pin plug - network bus to connect multiple processors.

BUZ1 two lead connection to speaker

P1 MODB - Vbb or Gnd
P2 MODA - Vcc or Gnd
P3 IRQ\ - Vcc or Gnd
P4 XIRQ\- Vcc or Gnd

X 9 - LED connection
X10 - sensor connection
X11 - sensor connection
X12 - LED connection
X13 - sensor connection
X14 - LED connection
X15 - LED connection
X16 - sensor connection

Power connector

X18 - p1 - VCHARGE	Connect to center (+) lead on charger plug
X18 - p2 - VBATSW1	Connect to plus lead on battery through power switch
X18 - p3 - VBAT	Connect directly to the plus lead on battery
X18 - p4 - MOT2+	Right motor plus lead
X18 - p5 - MOT2-	Right motor negative lead
X18 - p6 - MOT1+	Right motor plus lead
X18 - p7 - MOT1-	Right motor negative lead
X18 - p8 - no connect	
X18 - p9 - GND	Negative on charger connector
X18 - p10 - GND	Negative on battery
X18 - p11 - GND	Reset pushbutton switch
X18 - p12 - RESET	Reset pushbutton switc

Components

XTAL - 8.00 Megahertz

U1 74HC00 - quad NAN Gate

U2 74HC00 - quad NAN Gate

U3 74HC138 - Octal Latch

U4 MS62256L - 32K RAM

U5 ROM - 27C128

U6 74HC138 - 3 to 8 MUX

U7 TL084CN

X1 MAX232 - RS232 Line Driver

X2 F68HC11FH microcontroller

X17 UDN2993B motor control

VR1 7805 5v regulator

VR2 7805 5v regulator

VR3 7805 8v regulator

Chapter VI

High Level Source Code

```
FORTH DEFINITIONS
HEX

: START-PROM ;

2FE CONSTANT LAST-PROM ( DON 6-19-93 )
2FB CONSTANT AUTO-VEC

: VERSION
  CR
  ." Whiskers KB.2.3" CR
  ." Multitasker 1.0" CR ;

: WARN ( Redefine warning to work with FORTH.ASP PROCOM protocal )
   CR HERE COUNT TYPE ."  ? MS" 47 EMIT 20 EMIT DUP .
   1000 0 DO LOOP
   HERE COUNT TYPE ."  forth error message " . CR ABORT ;

 ( ' WARN CFA 04C ! -1 054 !  DON

: NAME ( CFA -- )  (  usage   CFA NAME NEWNAME
   CREATE -2 ALLOT 2+ HERE 2- ! ;

( Hidden -headerless- MaxFORTH words )
( F415 NAME INNUMBER
F65A NAME 0
FDD6 NAME 0!
EBB7 NAME -FIND
E6E5 NAME DOCOL
F57C NAME SP!
F573 NAME RP!
F38C NAME ?STATE
EA69 NAME ?ERROR
EA81 NAME ?STACK
F656 NAME 1
F50C NAME HEADERLESS
FE07 NAME @@
FE13 NAME @!
( FE80 NAME DOCON
( EE50 NAME DO2CON
( FEB7 NAME DOUSE
EA51 NAME ?EXEC
F2EA NAME ?NOTEND
F52D NAME WARNING
F521 NAME UABORT
FC87 NAME (DO)
FC9E NAME (LOOP)
EA9F NAME DLIT
FE2C NAME CLIT
FE39 NAME LIT
FCFC NAME NZBRANCH

( WORD TIMER CODE                                   Feb 13,1991 )

HEX

: TM      ( use: TM WORD to time WORD in microseconds )
          ' CFA
          B00E @ >R EXECUTE
          B00E @ R> - AA - 0 2 UM/MOD . DROP ."  microseconds"
          ;

HEX
```

```
: >NAME ( CFA -- NFA | 0 )
   LATEST
   BEGIN
           DUP >R 1 TRAVERSE 1+
           2DUP 2+ @ 2- = IF 2DROP R> EXIT ELSE R> DROP THEN
           @ DUP 0=
   UNTIL SWAP DROP
   ;


( System Words)

( FORGET SYSTEM )

: SYSTEM ;

: COPYRIGHT
 CR VERSION
 CR ." (c) 1992 ANGELUS RESEARCH"
 CR ." This software cannot be sold or incorporated into"
 CR ." another product without express permission from"
 CR ." Angelus Research, 6344 Sugar Pine Circle, #98"
 CR ." Angelus Oaks, California 92305 (909) 794-8325"
 CR ." Purchasing this product, gains the purchaser the"
 CR ." right to modify and use the source code provided for"
 CR ." their personal use only, with exception to the source"
 CR ." in the public.fth file which may be freely distributed."
 CR ." All rights reserved"
 CR
 CR ." to bypass autostarting, press space bar..."
 CR ." to restore system to factory configuration, press Q."
 CR
 CR ." May the Forth be with you..."
;

HEX


: ARRAY: ( bytes -- )
  VARIABLE
  VP +! ;


: REG: ( n -- )
  ( fix me for users! )
  VARIABLE -1 VP +! ;

: BINARY
  2 BASE ! ;

: &H HEX ;     IMMEDIATE
: &D DECIMAL ; IMMEDIATE
: &B BINARY ;  IMMEDIATE

DECIMAL

: ON ( a -- )
  -1 SWAP C! ;

: OFF ( a -- )
  0 SWAP C! ;

REG: rBUSY       ( DON flag for behavior level
```

```
REG: rINSTINCTS

: WAIT-BUSY ( -- )
  BEGIN                              ( wait until task completed
    rBUSY C@
  NOT UNTIL ;

: WORDS

  WAIT-BUSY

  rBUSY ON
  WORDS
  rBUSY OFF ;



( Register Locations )
HEX
B000 CONSTANT PORTA
B002 CONSTANT PIOC
B003 CONSTANT PORTC
B004 CONSTANT PORTB
B005 CONSTANT PORLCL
B007 CONSTANT DDRC
B008 CONSTANT PORTD
B009 CONSTANT DDRD
B00A CONSTANT PORTE
B00B CONSTANT CFORC ( oc1 oc2 oc3 oc4 oc5 - - - force output compare   8 )
B00C CONSTANT OC1M  ( oc1 oc2 oc3 oc4 oc5 - - - enable port A pins     8 )
B00D CONSTANT OC1D  ( oc1 oc2 oc3 oc4 oc5 - - - pin level high = 1     8 )
B00E CONSTANT TCNT  ( main timer read only       16)
B010 CONSTANT TIC1  ( input compare 1 timer        16)
B012 CONSTANT TIC2  ( input compare 2 timer        16)
B014 CONSTANT TIC3  ( input compare 3 timer        16)
B016 CONSTANT OC1   ( output compare 1 timer       16)
B018 CONSTANT OC2   ( output compare 2 timer       16)
B01A CONSTANT OC3   ( output compare 3 timer       16)
B01C CONSTANT OC4   ( output compare 4 timer       16)
B01E CONSTANT OC5   ( output compare 5 timer       16)
B020 CONSTANT TCTL1 ( om2 ol2 om3 ol3 om4 ol4 om5 ol5 timer contro   8 )
B021 CONSTANT TCTL2 (  - - edg1b edg1a edg2b edg2a edg3b edg3a                 8 )
B022 CONSTANT TMSK1 ( oc1I oc2I oc3I oc4I oc5I ic1I ic2I ic3I IRQ enable   8 )
B023 CONSTANT TFLG1 ( oc1f oc2f oc3f oc4f oc5f ic1f ic2f ic3f IRQ f 1=clr  8 )
B024 CONSTANT TMSK2 ( toI rtiI paovI raiI - - pr1 pr0  overflow IRQ enable 8 )
B025 CONSTANT TFLG2 ( tof rtif paovf paif - - - - timer ovrflow=1 tof=1 set8 )
B026 CONSTANT PACTL ( ddrA7 paen pamod pedge - - rtr1 rtr0 pulse acc ctrl  8 )
B027 CONSTANT PACNT
B028 CONSTANT SPCR
B029 CONSTANT SPSR
B02A CONSTANT SPDR
B02B CONSTANT BAUD
B02C CONSTANT SCCR1
B02D CONSTANT SCCR2
B02E CONSTANT SCSR
B02F CONSTANT SCDR
B030 CONSTANT ADCTL
B031 CONSTANT ADR1
B032 CONSTANT ADR2
B033 CONSTANT ADR3
B034 CONSTANT ADR4

&B
: INIT ( -- )
```

```
 10000000 PACTL C!
 00000000 PORTA C!
 00100000 DDRD C! ;
&D

: DISABLE ( -- )
 0 PORTA C!
 0 TMSK1 C!
 ;

: ENABLE ( -- )
 &B 00010000 &D TMSK1 C!
 ;

&B
01100000 CONSTANT LF
01000000 CONSTANT LR
00011000 CONSTANT RF
00010000 CONSTANT RR
01111000 CONSTANT FWD
01010000 CONSTANT REV
00000000 CONSTANT ST
01110000 CONSTANT PVR
01011000 CONSTANT PVL

&H

REG: rMASK

&D

: M ( n -- )
 DEPTH
 IF
        DUP PORTA C!
        rMASK C!
 ELSE
        CR ." Nothing on stack "
 THEN
 CR ." Use one of these: LF LR RF RR FWD REV ST PVL PVR"
;

: S ( -- )
 ST M ;


DECIMAL

( Task Words )

ALIAS +!    INCREASE   ( n ADDR -- )  ( DON 6-24-93
ALIAS -!    DECREASE   ( n ADDR -- )
ALIAS 1+!   INCREMENT  ( ADDR -- )    ( DON 6-24-93
ALIAS 1-!   DECREMENT  ( ADDR -- )
ALIAS !     NOW        ( ADDR -- )
ALIAS @              VALUE      ( ADDR -- )
ALIAS C!    SET        ( ADDR -- )    ( DON 7-7-93
ALIAS C@    GET                 ( ADDR --)
ALIAS SP! CLEAR-STACK  ( -- )
ALIAS DISPLAY" ."                          ( DON 7-14-93

: INDEX ( addr offset -- n )
 DEPTH
 1 >
```

```
    IF
      1- +
    ELSE
      CLEAR-STACK
      CR ." need array name and index!"
    THEN
;

: THEN
  DONE ; IMMEDIATE   ( DON 7-14-93 )

: SYSTEM-INIT
  INIT
  &H FF6C 26 ! &D   ( KHZ FOR writing to EE rom )
  0 rMASK C! ;

.( load TIMER.FTH next...)

( Timer Routines   DRG 3-7-92 )

HEX
( SENSOR 203-374-1411 EXT 127)
( FORTH DEFINITIONS )
( ASSEMBLER )

CODE-SUB CLI
  0E C, ( CLI )
  39 C, ( RTS )
END-CODE

CODE-SUB SEI
  0F C, ( SEI )
  39 C, ( RTS )
END-CODE

( FORGET ADC1 )
: ADC1 ;

&B
    01111111 CONSTANT BPC-OFF
    10000000 CONSTANT BPC-ON
&H

: BEYES-ON ( -- )
  PORTA C@  BPC-ON OR  PORTA C! ;

: BEYES-OFF ( -- )
  PORTA C@  BPC-OFF AND          PORTA C! ;

: DELAY ( time -- )
  0 DO LOOP ;


DECIMAL

( REG: addresses are consecutive and can be treated as a table )

( storage table for the first bank of the adc registers )
REG: rADR1-OFF
REG: rADR2-OFF
REG: rADR3-OFF
REG: rADR4-OFF
REG: rADR1-ON
REG: rADR2-ON
```

```
REG: rADR3-ON
REG: rADR4-ON
( storage table for the second bank of the adc registers )
REG: rADR1-B
REG: rADR2-B
REG: rADR3-B
REG: rADR4-B



1024  CONSTANT 1K

1K ARRAY: 1K-BUFFER

VARIABLE vSAMPLES
: SAMPLES ( samples -- )
        DUP 1K >
          IF
            ." too many...1024 max!"
            DROP
          ELSE
            vSAMPLES !
          THEN ;

VARIABLE vRATE
: RATE ( rate -- )
  vRATE ! ;

: DIGITIZE ( -- )

        DISABLE         ( interupts )
        &B 00110100 &D ADCTL C!

        vSAMPLES @
        1K-BUFFER + 1K-BUFFER
        DO
                vRATE @ 0 DO LOOP
                ADR4 C@
                I C!
        LOOP

  &B 00010000 &D ADCTL C!

;


: AVERAGE ( -- average )
        ( Loop contents optimized for speed )
        0.0                     ( SUM )
        1K-BUFFER vSAMPLES @ +        ( HI )
        1K-BUFFER               ( LO )
        DO I C@ 0 D+ LOOP
        vSAMPLES @ UM/MOD
        SWAP 2* vSAMPLES @ > -        ( round result )
        ;

: SOUND-LEVEL ( -- level )
   DISABLE
   2000 DELAY
   10 RATE
   1K SAMPLES
   DIGITIZE
   AVERAGE ;
```

```
: INIT-ADC1 ( -- )
  1K-BUFFER 1K ERASE
  rADR1-OFF 12 ERASE

  256 vSAMPLES                  !
  100 RATE
;

DECIMAL

1 CONSTANT cRIGHT    ( RCL Direction controls )
2 CONSTANT cLEFT
3 CONSTANT cFWD
4 CONSTANT cREV

REG: rCHOICE                ( RCL direction choice )

REG: rLSPEED
REG: rRSPEED

( left motor pulse width modulation registers )
REG: rLEFT-CTR
REG: rLTON
REG: rLTOFF

( right motor pulse width modulation registers )
REG: rRIGHT-CTR
REG: rRTON
REG: rRTOFF

VARIABLE vCOUNTER  ( Counts instinct interrupts)

VP @ CONSTANT cVAR-START ( start of the system variables )


( ************* start of system variables *****************)


REG: rMAX-SPEED

REG: rCORRECTION                  ( Modifies right speed to match left, can be + or - )

REG: rLF-TRIGGER                  ( discriminator values )
REG: rLSF-TRIGGER
REG: rRF-TRIGGER
REG: rRSF-TRIGGER
REG: rL-TRIGGER
REG: rR-TRIGGER
REG: rF-TRIGGER
REG: rLM-TRIGGER
REG: rRM-TRIGGER

REG: rFACTOR                      ( trigger calibration factor )

REG: rLF-MASK                       ( collision response masks )
REG: rLSF-MASK
REG: rRF-MASK
REG: rRSF-MASK
REG: rLW-MASK
REG: rRW-MASK
REG: rL-MASK
REG: rR-MASK
REG: rF-MASK
REG: rLM-MASK
```

```
REG: rRM-MASK

REG: rCOLLISION                  ( disable collision detection flag )

VARIABLE vSENSOR-CTR             ( sensor timing values--could be REG: )
VARIABLE vEYES-ON-PER
VARIABLE vEYES-OFF-PER
VARIABLE vSCAN-OFF-DELAY
VARIABLE vSCAN-ON-DELAY

VARIABLE vPWM-CYC

REG: rSTALL-DELAY                ( disable stall detection VALUE )
REG: rLIGHTS                     ( disable light collisions )
REG: rSTALLS                     ( disable stall collisions )
REG: rWHISKERS                   ( disable whiskers collisions )
REG: rSENSE              ( disable ADC readout -required for LIGHTS,STALLS)
REG: rSUM-FACTOR         ( DON used to adjust sum trigger for calibration

VARIABLE vMAX-COMPASS

REG: rMAX-INSTINCTS

2VARIABLE vPACE                   ( sets pace of song in units of 1/4 notes/min)
( example   60 vPACE ! will slow a song to 60 1/4 notes/min)
( 120 vPACE ! is 'normal' )

VARIABLE vINT-CFA        ( cfa of hi level hook in interrupt routine )

( ************* end of system variables *****************)

VP @ cVAR-START -
CONSTANT cVAR-LENGTH                     ( length of the system variable table )

VARIABLE vCOMPASS
REG: rNO-STALLS                          ( disable stall detection flag )

VARIABLE vWAIL

( initialize the system variables )

: TRIGGER-FACTOR ( n -- )
 DUP 255 <
  IF
   rFACTOR C!
  ELSE
   CR ." too high...must be less than 255!"
  THEN
;

: SUM-FACTOR ( n -- )
 DUP 255 <
  IF
   rSUM-FACTOR C!
  ELSE
   CR ." too high...must be less than 255!"
  THEN
 ;

: INIT-VARS

   vSENSOR-CTR           0!
 3  vEYES-ON-PER         !
 3 vEYES-OFF-PER         !
```

```
  1 vSCAN-OFF-DELAY  !
  1 vSCAN-ON-DELAY   !
 16700 vPWM-CYC      !
;
: INIT-OPTIONS
 50 rLSPEED          C!
 50 rRSPEED          C!
 DECIMAL
 80 rSUM-FACTOR      C!
 50  rFACTOR         C!
 100 rMAX-SPEED      C!
    rCORRECTION      OFF
 455 vMAX-COMPASS    !
 0 vCOMPASS          !
    vINT-CFA         0!

  50 rNO-STALLS      C!
  255 rSTALL-DELAY   C!
  23  rLM-TRIGGER    C!
  23  rRM-TRIGGER    C!
  160 vWAIL           !
  &H
  10 rMAX-INSTINCTS  C!
  -1 rCOLLISIONC!    ( Enable collision detection )
  -1 rSENSE          C!      ( Enable flashing lights and ADC reads )
  -1 rWHISKERS       C!      ( Enable whisker collisions )
  -1 rLIGHTS         C!      ( Enable light collisions )
  -1 rSTALLS         C!      ( Enable stall collisions )
  ( rLF-TRIGGER 7 FF FILL    ( DON set trigger levels )

  &D
  120 120 vPACE      2!      ( set full rational fraction )
;

INIT-OPTIONS
INIT-VARS

REG: rLF-VALUE               ( on - off values )
REG: rLSF-VALUE
REG: rRF-VALUE
REG: rRSF-VALUE
REG: rL-VALUE
REG: rR-VALUE
REG: rF-VALUE
REG: rLM-VALUE
REG: rRM-VALUE

&B
1              CONSTANT LF-OBSTACLE           ( specific collision flags )
10             CONSTANT LSF-OBSTACLE
100             CONSTANT RF-OBSTACLE
1000           CONSTANT RSF-OBSTACLE
10000          CONSTANT L-OBSTACLE
100000         CONSTANT R-OBSTACLE
1000000        CONSTANT F-OBSTACLE
                                              ( Start new byte for whiskers and stalls )
100000000      CONSTANT LW-OBSTACLE
1000000000     CONSTANT RW-OBSTACLE
10000000000    CONSTANT LM-OBSTACLE
100000000000   CONSTANT RM-OBSTACLE

VARIABLE vCOLLIDED                            ( any collision flag - or of obstacles )
VARIABLE vCOL-FLAG                            ( any collision signal )
```

```
VARIABLE vSENSOR-CYC                                ( sensor cycle toggle )

&B 00010000 &H CONSTANT TFLG1-MASK                  ( OC4 int flag )
&B 00010000 &H CONSTANT SCAN-A                      ( 1st ADC bank )
&B 00010100 &H CONSTANT SCAN-B                      ( 2nd ADC bank )
&B 10000000 &H CONSTANT SENSOR-MASK

( DON
( &B 00000001 &H CONSTANT RW-MASK                   ( whisker masks )
( &B 00000010 &H CONSTANT LW-MASK
  &B 00000010 &H CONSTANT RW-MASK                   ( whisker masks )
  &B 00000001 &H CONSTANT LW-MASK

REG: rCOL-MASK                                      ( motor mask from collision )



: RIGHT ( -- )
  cRIGHT rCHOICE C! ;

: LEFT ( -- )
  cLEFT rCHOICE C! ;

  &B
    10111111 CONSTANT L-OFF
    01000000 CONSTANT L-ON
  &D


  &B 11101111 CONSTANT R-OFF
    00010000 CONSTANT R-ON
  &D

: PERIODS ( ms -- | ~ 2 ms typical )
  vCOUNTER 0!
  BEGIN
    vCOUNTER @ OVER >
  UNTIL
  DROP ;


HEX

: OBSTACLE ( MASK ... T/F )
  vCOLLIDED @ AND ;

: CLEAR-SENSORS ( -- )                              ( erase memory of hit )
  vCOLLIDED 0!
  vCOL-FLAG 0!
  ;

DECIMAL

: PRINT                                             ( print the name of the following word inside : definition )
  R@ 2+ @ 2- NFA ID. SPACE
;

: 10.R 9 .R SPACE ;
: 7.R 6 .R SPACE ;
: 5.R 4 .R SPACE ;
: 10_ 10 SPACES ;
: 17.R 10_ 7.R ;

: SCANS
```

```
            CR  PRINT vPWM-CYC @ 7.R
            CR  PRINT vSENSOR-CTR @ 7.R
               PRINT vSENSOR-CYC @ 7.R
            CR  PRINT vEYES-ON-PER @ 7.R
               PRINT vEYES-OFF-PER @ 7.R
            CR  PRINT vSCAN-OFF-DELAY @ 7.R
               PRINT vSCAN-ON-DELAY @ 7.R
            CR ;

: MOTORS
  CR PRINT rMAX-SPEED C@ 7.R
     PRINT rCORRECTION C@ 7.R
     PRINT vPWM-CYC @ 7.R
  CR PRINT rLSPEED C@ 5.R
     PRINT rRSPEED C@ 5.R
  CR PRINT rLTON  C@ 5.R
     PRINT rLTOFF C@ 5.R
     PRINT rLEFT-CTR C@ 5.R
  CR PRINT rRTON  C@ 5.R
     PRINT rRTOFF C@ 5.R
     PRINT rRIGHT-CTR C@ 5.R
  BASE @ BINARY
  CR PRINT rMASK C@ .
     PRINT PORTB C@ 10.R
     PRINT PORTA C@ 10.R
  BASE !
     ;

DECIMAL
: SENSORS ( -- )
  CR 21 SPACES
  ." ON  OFF  VALUE  OBSTACLE   TRIGGER   MASK"
  CR
  ." Left Front Sensor  "
    rADR1-ON C@ 5.R
    rADR1-OFF C@ 5.R
    rLF-VALUE  C@ 7.R
    LF-OBSTACLE OBSTACLE 10.R
    rLF-TRIGGER C@ 10.R
    rLF-MASK C@ 7.R
  CR
  ." Left Side Sensor   "
    rADR2-ON C@ 5.R
    rADR2-OFF C@ 5.R
    rLSF-VALUE   C@ 7.R
    LSF-OBSTACLE OBSTACLE 10.R
    rLSF-TRIGGER C@ 10.R
    rLSF-MASK C@ 7.R
  CR
  ." Right Front Sensor "
    rADR3-ON C@ 5.R
    rADR3-OFF C@ 5.R
    rRF-VALUE  C@ 7.R
    RF-OBSTACLE OBSTACLE 10.R
    rRF-TRIGGER C@ 10.R
    rRF-MASK C@ 7.R
  CR
  ." Right Side Sensor  "
    rADR4-ON C@ 5.R
    rADR4-OFF C@ 5.R
    rRSF-VALUE  C@ 7.R
    RSF-OBSTACLE OBSTACLE 10.R
    rRSF-TRIGGER C@ 10.R
    rRSF-MASK C@ 7.R
```

```
CR
." Left summed sensor "
  rL-VALUE       C@ 17.R
  L-OBSTACLE OBSTACLE 10.R
  rL-TRIGGER C@ 10.R
  rL-MASK C@ 7.R
CR
." Right summed sensor"
  rR-VALUE       C@ 17.R
  R-OBSTACLE OBSTACLE 10.R
  rR-TRIGGER C@ 10.R
  rR-MASK C@ 7.R
CR
." Front summed sensor"
  rF-VALUE       C@ 17.R
  F-OBSTACLE OBSTACLE 10.R
  rF-TRIGGER C@ 10.R
  rF-MASK C@ 7.R
CR
." Left whisker       "
  LW-OBSTACLE OBSTACLE 10_ 17.R
  rLW-MASK C@ 10_ 7.R
CR
." Right whisker      "
  RW-OBSTACLE OBSTACLE 10_ 17.R
  rRW-MASK C@ 10_ 7.R
CR
." Left motor current "
  rLM-VALUE   C@ 17.R
  LM-OBSTACLE OBSTACLE 10.R
  rLM-TRIGGER C@ 10.R
  rLM-MASK C@ 7.R
CR
." Right motor current"
  rRM-VALUE   C@ 17.R
  RM-OBSTACLE OBSTACLE 10.R
  rRM-TRIGGER C@ 10.R
  rRM-MASK C@ 7.R
CR
  PRINT rCOLLISION C@ 7.R
  PRINT vCOLLIDED @ 7.R
  PRINT vCOL-FLAG @ 7.R
  PRINT rFACTOR C@ 5.R
CR
  PRINT rWHISKERS C@ 5.R
  PRINT rSENSE C@ 5.R
  PRINT rLIGHTS C@ 5.R
  PRINT rSTALLS C@ 5.R
CR
  PRINT vCOMPASS @ 7.R
  PRINT vMAX-COMPASS @ 7.R
CR ." BATTERY =" rADR1-B    C@ 5.R
   ."    SOUND =" rADR4-B    C@ 5.R
CR
   PRINT rNO-STALLS C@ 5.R
   PRINT rSTALL-DELAY C@ 5.R
   PRINT rMASK C@ 5.R
   PRINT rCOL-MASK C@ 5.R
CR
   PRINT rBUSY C@ 5.R
   PRINT rMAX-INSTINCTS C@ 5.R
   PRINT vINT-CFA @ DUP IF 2- NFA SPACE ID. ELSE . THEN
CR
;
```

```
&D
REG: rSAV-DIR

: REVERSE-DIR ( -- )
  rMASK C@
   &B 00101000 XOR  &D
  rMASK C!
;

: SAVE-DIR ( -- )
  rMASK C@
  rSAV-DIR C! ;

: RESTORE-DIR ( -- )
  rSAV-DIR C@
  rMASK C! ;

HEX

CREATE VEC-TABLE

 7E C, FFFE @ ,    ( B7BF SCI SER SYS )
 7E C, FFFE @ ,    ( B7C2 SPI SER )
 7E C, FFFE @ ,    ( B7C5 PLS ACC EDGE )
 7E C, FFFE @ ,    ( B7C8 PLS ACC OVFL )
 7E C, FFFE @ ,    ( B7CB TMR OVERFLOW )
 7E C, FFFE @ ,    ( B7CE TMR OUT CMP 5 )
 7E C, ' INSTINCT @ , ( B7D1 TMR OUT CMP 4 )
 7E C, FFFE @ ,     ( B7D4 TMR OUT CMP 3 )
 7E C, FFFE @ ,     ( B7D7 TMR OUT CMP 2 )
 7E C, FFFE @ ,    ( B7DA TMR OUT CMP 1 )
 7E C, FFFE @ ,    ( B7DD TMR IN CAP 3 )
 7E C, FFFE @ ,    ( B7E0 TMR IN CAP 2 )
 7E C, FFFE @ ,    ( B7E3 TMR IN CAP 1 )
 7E C, FFFE @ ,    ( B7E6 REAL TIME )
 7E C, FFFE @ ,    ( B7E9 IRQ )
 7E C, FFFE @ ,    ( B7EC XIRQ )
 7E C, FFFE @ ,    ( B7EF SWI )
 7E C, FFFE @ ,    ( B7F2 OP-CODE TRAP )
 7E C, FFFE @ ,    ( B7F5 COP FAILURE )
 7E C, FFFE @ ,    ( B7F8 CLK MON )
 HERE CONSTANT VEC-TABLE-END

  DISABLE

( Robot Control Language Version 1.0  12-3-92 )

( Copyright Angelus Research 1992-1997                 )
( This software cannot be sold or incorporated into )
( another product without express permission from   )
( Angelus Research, 6344 Sugar Pine Circle,     )
( Angelus Oaks, California 92305 ( 909-794-8325        )
( All rights reserved                                  )

( FORGET RCL )
: RCL ;
DECIMAL

: SPEED ( speed -- )  ( motor = LEFT or RIGHT )
  rMAX-SPEED C@ MIN
  0 MAX
  rCHOICE C@
    cLEFT = IF
```

```
                DUP rLSPEED C!
                DUP rLTON C!
                rMAX-SPEED C@ SWAP - rLTOFF C!
              ELSE
                DUP rRSPEED C!
                DUP rRTON C!
                rMAX-SPEED C@ SWAP - rRTOFF C!
              THEN
  NO-STALLS
  ;

: % ( n percent -- )
  100 */ ;

: %SPEED ( n -- )  ( n is percent of full speed )
        rMAX-SPEED C@ %
        SPEED ;

: ACTION ( -- t/f ) ( sense both whiskers hit )
        RW-OBSTACLE LW-OBSTACLE +
        DUP OBSTACLE =
        ;

REG: SAV-rLSPEED
REG: SAV-rRSPEED

: SECS ( secs -- )
  2000.000 vPWM-CYC @ UM/MOD SWAP DROP  *
  PERIODS ;

: MINUTES ( min -- )
  60 SECS ;

360 CONSTANT 360

: ADJUST-DEGREES ( n -- )
  vMAX-COMPASS +! ;

: CLEAR-COMPASS
        vCOMPASS 0!
        ;

: BEARING ( -- bearing )
        vCOMPASS @
        360 vMAX-COMPASS @ */
        ;

: DEGREES  ( degrees -- )
        ABS
        rCHOICE C@
        cRIGHT =
        IF
                BEARING +
                BEGIN
                        DUP BEARING
                < UNTIL
        ELSE
                NEGATE BEARING +
                BEGIN
                        DUP BEARING
                > UNTIL
        THEN
        DROP ;
```

```
: STOP ( -- )
 ( CLEAR-SENSORS
 NO-STALLS
 ST rMASK C! ;

: FORWARD ( -- )
 ( CLEAR-SENSORS
 NO-STALLS
 FWD rMASK C! ;

: BACKUP ( -- )
 ( CLEAR-SENSORS
 NO-STALLS
 REV rMASK C! ;

ALIAS BACKUP BACK

CREATE TRIGGER-TABLE

 12 C, 13 C, 14 C, 15 C, 16 C, 17 C, 19 C, 21 C, 23 C, 26 C, 29 C,
 33 C, 37 C, 41 C, 46 C, 51 C, 56 C, 62 C, 69 C, 77 C, 86 C, 96 C,
 107 C, 119 C, 132 C, 156 C, 190 C,

DECIMAL
( DON


: CALIBRATE
    CR ." trigger set to   L    LS   R    RS   L+   R+   F+ "
        CR 15 SPACES
        4 0 DO                      ( Treat individual triggers as a table )
                TRIGGER-TABLE
                rADR1-OFF I + C@ ( get ambient light )
                rFACTOR C@ / +          ( calculate table offset )
                C@ DUP 5.R
                rLF-TRIGGER I + C!
        LOOP
        rLF-TRIGGER C@                  ( left summed trigger )
        rLSF-TRIGGER C@
        +  rSUM-FACTOR C@  % ( DON 2/ DUP 2/ +
        255 MIN
        DUP 5.R rL-TRIGGER C!

        rRF-TRIGGER C@                  ( right summed trigger )
        rRSF-TRIGGER C@
        +  rSUM-FACTOR C@  % ( DON 2/ DUP 2/ +
        255 MIN
        DUP 5.R rR-TRIGGER C!

        rLF-TRIGGER C@                  ( front summed trigger )
        rRF-TRIGGER C@
        +  rSUM-FACTOR C@  % ( DON 2/ DUP 2/ +
        255 MIN
        DUP 5.R rF-TRIGGER C!
        ;

: START-ROBOT
 INIT

 0 vPRIORITY !                          ( init BEHAVIORS 6-19-93

 &B 00010000 &H ADCTL C!
 SEI
 VEC-INIT
```

```
 VEC-INIT

 &B 00010000 &D TMSK1 C!
 900  OC4 !
 CLI
 CALIBRATE          ( DON 5-17-93 )
 ;

: SAVE-SPEEDS ( -- )
 rLSPEED C@ SAV-rLSPEED C!
 rRSPEED C@ SAV-rRSPEED C! ;

: RESTORE-SPEEDS ( -- )
 ( CLEAR-SENSORS
 SAV-rLSPEED C@  LEFT SPEED
 SAV-rRSPEED C@  RIGHT SPEED ;

: RAMP-UP ( rate speed -- )
 ( CLEAR-SENSORS
 SWAP 10 * SWAP                ( DON delay interval )
 100 MIN
 0 DO
  I LEFT SPEED                 ( DON changed to SPEED
  I RIGHT SPEED                ( "
  DUP DELAY
 LOOP
 DROP ;

: RAMP-DOWN ( rate -- )
 ( CLEAR-SENSORS
 100 *
 0 rLSPEED C@rRSPEED C@  MIN
 DO
  I LEFT SPEED
  I RIGHT SPEED
  DUP DELAY
 -1 +LOOP
 STOP
 ;

ALIAS SENSORS SHOW-SENSORS ( -- )

: STRAIGHT ( Set speeds to same average value -with correction )
 rCHOICE C@
 rCORRECTION C@
 rRSPEED C@ rLSPEED C@ + OVER + 2/
 ( Split speed difference between sides )
 RIGHT SPEED
 rRSPEED C@ SWAP -
 LEFT SPEED
 rCHOICE C!
 ;

: PIVOT ( -- ) ( use current average speed )
 ( CLEAR-SENSORS
 NO-STALLS
 rCHOICE C@            ( Get chosen direction )
  cLEFT = IF
          PVL rMASK C!
         ELSE
          PVR rMASK C!
         THEN
 ;
```

```
: BEST-PIVOT ( -- )
( CLEAR-SENSORS
 NO-STALLS
 RSF-OBSTACLE OBSTACLE
  IF
   LEFT PIVOT   ( DON
  ELSE
   RIGHT PIVOT              ( DON
  THEN  ;


: ALL-INSTINCTS ( mask -- )
 rLF-MASK  11 ROT FILL ; ( masks are stored consequtively in RAM )

100 ARRAY: LEVELS

VARIABLE vSTEP

: STEP ( step -- ) ( degrees to step )
        vSTEP ! ;

: FIND-SOUND ( -- )         ( DON
        STOP
        CLEAR-COMPASS
        75 LEFT SPEED
        75 RIGHT SPEED
        PVL ALL-INSTINCTS

        360  vSTEP @  /
        0 DO

                NO-STALLS
                ENABLE
                RIGHT PIVOT
                vSTEP @ I * BEARING -
                DEGREES
                STOP

                DISABLE
                10000 DELAY
                SOUND-LEVEL  DUP
                LEVELS I +  C!

                CR ." level=" . SPACE
                PRINT BEARING .

        LOOP ;

: MAX-LEVEL ( -- index )
        0 0           ( maximum index,level reading )
        360 vSTEP @ / 0
        DO
                DUP I LEVELS + C@
                < IF                                ( new maximum )
                        2DROP                       ( old index,level )
                        I DUP LEVELS + C@           ( new index, level )
                THEN

        LOOP
        ;

: PIVOT-SOUND ( -- )
        BEARING
```

```
        MAX-LEVEL DROP
        vSTEP @ * -
        CR ." going to " BEARING OVER - .  SPACE ." degrees" CR
        NO-STALLS
        ENABLE
        LEFT
        PIVOT
        DEGREES
        STOP ;

: TURN ( -- )
 NO-STALLS
 rCHOICE C@

   DUP cLEFT =
     IF
         RF rMASK C!
     ELSE
         LF rMASK C!
     THEN
;

: CIRCLE ( dia -- ) ( Set difference in speed of 'dia' )
 rMAX-SPEED C@ %    ( All speeds are treated as % of full speed )
 rCORRECTION C@ +   ( Split speed difference between sides )
 rRSPEED C@ rLSPEED C@ + OVER + 2/
 rCHOICE C@
   cRIGHT =
    IF
        LEFT SPEED              ( use range check in speed to set bounds )
        rLSPEED C@ SWAP -       ( ensure proper difference )
        RIGHT SPEED
    ELSE
        RIGHT SPEED
        rRSPEED C@ SWAP -
        LEFT SPEED
    THEN
 FORWARD ;

: SQUARE ( size -- )
 1000 *
 STRAIGHT FORWARD
 4 0 DO
        DUP DELAY
        RIGHT PIVOT  90 DEGREES
        FORWARD
 LOOP
 DROP
 STOP ;

VARIABLE vWAIT-TIME

: %HIGHER-TRIGGER ( percent -- )
        7 0 DO
                rLF-TRIGGER I + C@
                OVER %
                rLF-TRIGGER I + C!
        LOOP DROP ;

: WAIT-TIME ( time -- )
 vWAIT-TIME ! ;

: DEFAULT-INSTINCTS ( -- )
  PVL rRF-MASK            C!            ( intialize instincts )
```

```
   PVL rRSF-MASK C!
   PVR rLF-MASK          C!
   PVR rLSF-MASK C!
   PVR rLW-MASK          C!
   PVL rRW-MASK          C!
   PVR    rL-MASK        C!
   PVL    rR-MASK        C!
   REV    rF-MASK        C!
   RR  rLM-MASK          C!    ( DON 5-13-93 )
   LR  rRM-MASK          C!    ( DON 5-13-93 )
;
DEFAULT-INSTINCTS

: STOP-INSTINCTS ( -- )
        rLF-MASK 11 ERASE ( intialize instincts )
        ;

: SPIN ( %speed -- )
  rCHOICE C@ SWAP                        ( get current direction choice )
  DUP LEFT %SPEED RIGHT %SPEED
    cLEFT =
     IF
        LEFT PIVOT
     ELSE
        RIGHT PIVOT
     THEN
;

: INIT-RCL
  0 SAV-rLSPEED          C!
  0 SAV-rRSPEED          C!

  20 vSTEP          !
  1000 vWAIT-TIME !
;

                          ( Sound routines )

HEX

VOCABULARY MUSIC IMMEDIATE
MUSIC DEFINITIONS

HEX

DECIMAL

(  period    dur  )

  18182     28     NOTE:  A
  17161     29     NOTE:  A#
  16198     31     NOTE:  B
  15289     33     NOTE:  C
  14431     35     NOTE:  C#
  13621     37     NOTE:  D
  12856     39     NOTE:  D#
  12135     41     NOTE:  E
  11454     44     NOTE:  F
  10811     46     NOTE:  F#
  10204     49     NOTE:  G
   9631     52     NOTE:  G#
   9091     55     NOTE: 1A
   8581     58     NOTE: 1A#
   8099     62     NOTE: 1B
```

```
  7645    65    NOTE: 1C
  7215    69    NOTE: 1C#
  6810    73    NOTE: 1D
  6428    78    NOTE: 1D#
  6067    82    NOTE: 1E
  5727    87    NOTE: 1F
  5405    92    NOTE: 1F#
  5102    98    NOTE: 1G
  4816   104    NOTE: 1G#
  4545   110    NOTE: 2A
  4290   117    NOTE: 2A#
  4050   123    NOTE: 2B
  3822   131    NOTE: 2C
  3608   139    NOTE: 2C#
  3405   147    NOTE: 2D
  3214   156    NOTE: 2D#
  3034   165    NOTE: 2E
  2863   175    NOTE: 2F
  2703   185    NOTE: 2F#
  2551   196    NOTE: 2G
  2408   208    NOTE: 2G#
  2273   220    NOTE: 3A
  2145   233    NOTE: 3A#
  2025   247    NOTE: 3B
  1911   262    NOTE: 3C
  1804   277    NOTE: 3C#
  1703   294    NOTE: 3D
  1607   311    NOTE: 3D#
  1517   330    NOTE: 3E
  1432   349    NOTE: 3F
  1351   370    NOTE: 3F#
  1276   392    NOTE: 3G
  1204   415    NOTE: 3G#
  1136   440    NOTE: 4A
  1073   466    NOTE: 4A#
  1012   494    NOTE: 4B
   956   523    NOTE: 4C
   902   554    NOTE: 4C#
   851   587    NOTE: 4D
   804   622    NOTE: 4D#
   758   659    NOTE: 4E
   716   698    NOTE: 4F
   676   740    NOTE: 4F#
   638   784    NOTE: 4G
   602   831    NOTE: 4G#

DECIMAL

( START DON
( : CYCLES ( Period Duration --- )
(           MUSIC
(           SWAP 57 * SWAP 1 MAX NOTE ;  ( This is equivelent to old CYCLES )

FORTH DEFINITIONS
VARIABLE dTIME
VARIABLE sTIME
HEX

ASSEMBLER
HEX

FORTH

VARIABLE MIN-FREQ
```

```
VARIABLE LENGTH

: WAIL ( rate times  -- )
 MUSIC
  0 DO
        vWAIL @          DUP
        2* SWAP DO
                I I 2 PICK / NOTE
               LOOP
        LOOP
 DROP
;

: LSF-GREATER? ( -- )                  ( left side sensor greater than
 rLSF-VALUE C@                         ( right side sensor? T/F
 rRSF-VALUE C@
 > ;

: RSF-GREATER? ( -- )                  ( right side sensor greater than
 rRSF-VALUE C@                         ( left side sensor? T/F
 rLSF-VALUE C@
 > ;

: WARBLE ( freg times -- )             ( 400 10 WARBLE )

 MUSIC

 SWAP MIN-FREQ !
 0 DO
        MIN-FREQ @  60 DO
                      I MIN-FREQ @   1 /  I - NOTE
                 10 +LOOP
    LOOP ;

: LASER ( freg -- )                   ( 800 LASER )

 MUSIC

 MIN-FREQ !
 10 0 DO
        MIN-FREQ @  60 DO
                      I 10  NOTE
                10 +LOOP
    LOOP ;

: BIRD-CALL ( freq times -- )                ( 600 2 TONES )

 MUSIC

 SWAP MIN-FREQ !
 0 DO
        MIN-FREQ @  0 DO
                      I  MIN-FREQ @ I -  NOTE
                 100 +LOOP
    LOOP ;

: UP-DOWN ( freq steps -- )                  (        1000 20 BUP-DOWN )

 MUSIC

  DUP
  1 DO
        I OVER NOTE
```

```
   10 +LOOP

  DUP
  1 DO
       DUP I - OVER NOTE
   10 +LOOP
  2DROP
 ;

: TONES ( period times -- )                         ( 60 2 TONES )
 MUSIC
       0 DO
         DUP 0 DO
                I OVER I - NOTE
               LOOP
        LOOP DROP ;

: ~ 2000 0 DO LOOP ;

FORTH DEFINITIONS

: BLOW_THE_MAN_DOWN!
       MUSIC
       2 2G
       2 2G 1 3A 3 2G
       2 2E 2 2C 2 2E
       3 2G 1 3A 2 2G
       4 2E 1 2C 1 2E
       6 2G 6 3A
       3 2F 1 2E 2 2F
       4 2D 4 2D
       2 2D 2 2D 2 2D
       2 2F 2 2E 2 2D
       2 2F 2 2E 2 2D
       6 3A
       2 2G 2 2G 2 2G
       4 2G 2 2F
       3 2E 1 2D 2 2E
       6 2C ;

: RIDE_OF_THE_VALKERIES
       MUSIC
       2 2C 1 2C 3 2D# 3 2C
       2 2D# 1 2D# 3 2G 3 2D#
       2 2G 1 2G 3 3A# 3 2A#
       2 2D# 1 2D# 6 2G ;

       : INIT-ROBOT ( -- )
        SYSTEM-INIT
       INIT
        vCOUNTER            0!
        1  vSENSOR-CYC      !
          vCOLLIDED         0!
          rINSTINCTS        OFF
          vCOMPASS          0!
        50 LEFT SPEED
        50 RIGHT SPEED
        INIT-ADC1
        INIT-RCL
        DEFAULT-INSTINCTS ;

       HEX

       : CHECK-KEYBOARD ( -- )
```

```
 CR ." Press a key to stop autostart process " CR
 &D
 CR
 #TIB @  >IN !
 10000 0 DO
         ?TERMINAL
          IF
               KEY
                81 =     ( press Q key to reset system
                 IF
                  NEW-SYSTEM
                 THEN
               ABORT" key pressed, interactive mode now" ( DON
          THEN
        LOOP
;

HEX
: END-PROM ( end of eprom stub )
 45 C,    04E C, 44 C,  ( end of prom! )
 FF , FF , FF , FF , FF , ;
DECIMAL

CANCEL

: PROM-ENDS ;

HEX
( DON )
' PROM-ENDS 4A ! ( set fence to top of prom )
 END-PROM
```

Chapter VI

The History of Robotics

As told by Brian Morris in the book:

The World of Robots

Published by Gallery books and W.H. Smith Publishers
Madison Ave.
New York City, NY 10016

The follow text is an excerpt out of the fine text about robots. The Author did some excellent research into this field and really seemed to have a good grasp of the issues facing robotics today.

"Ladies and gentlemen - Meet the future! - the portentous but telling words of the traveling salesman in *Butch Cassidy and the Sundance Kid* as he introduces the crowd to the safety bicycle. They are as appropriate now in a book on robots as they were then in a film about outlaws.

We in the western world live in an industrialized, stratified society facing the promise and the problems of a post-industrial future in which the robot seems to offer either an inexhaustible source of cheap uncomplaining labor or else one more insidious threat to industrial employment and working-class prosperity; the salesman's audience lived in the Old West just as the Frontier was closing, as pioneering gave way to production, and the factories of the East and Mid-West began to turn out cheap, mass-produced consumer goods that would irrevocably transform the isolated bucolic existence of the typical North Arnerican. The bicycle was the first mechanized personal transport, affordable by anyone in employment irrespective of status or class; it was the forerunner of the automobile, the washing machine, the television, the telephone, the computer - all the household technology which the affluent sections of the twentieth-century world take for granted but upon which our leisurely, peaceful ways of life depend utterly.

Naturally, that bicycle salesman was no altruistic missionary, despite his spiel; he was there, hundreds of miles from paved streets and tramcars, in some cow-town in the middle of nowhere, at the behest of exactly the force that has the robot knocking at our doors today - the profit motive. Bicycles could be made more cheaply than the horses that they could more or less replace, so entrepreneurs built factories to make bicycles, and sent their salespeople across the face of the world to sell them-never mind the future or the social implications - the business of business is business! Come forward in time a century, for bicycle read robot, for horse read worker, and both the title and the cautionary rubric of this chapter are explained. The fruits of mechanization are certainly plentiful, but they are also strongly flavored, with more than a hint of sourness.

Just as the first stages of industrialization which produced the bicycle brought personal freedom and prosperity to much of the world, so that same process brought mechanized warfare, assembly lines and totalitarian societies. William Blake's "dark satanic mills" were mighty engines of progress but the miserable lives of the men, women and children who slaved there in heat, fumes, danger and din, were the coin in which the future was bought. Cheap coal and rubber made the bicycle economically possible then; cheap electricity, silicon and plastic make the robot possible today.

There may seem to be nothing dark or satanic about the sterile fluorescent-lit quiet of an integrated-circuit production line, but most of its human workers are Third-World women. Still subject to unheathly working conditions, economic exploitation and repressive political regimes often the more or less overt puppets of the multi-national companies that own or fund the industries whose raw material those integrated circuits are. The unemployed manual and semi-skilled workers all over the world whose jobs have been taken over by automated, computerized and (increasingly) robotic machines may not starve in their state-supported idleness, but neither are they likely to see themselves as the citizens of a new Periclean Athens, leisured and cultivated members of an elite society, freed from daily toil by the uncomplaining drudgery of armies of robot slaves.

Two hundred years of industrial development may have made us wary, if not actually cynical, about the possible social costs of robots, but the myths and folk-tales of history should be enough to close the subject once and for all. The image of the created being turning up on its creator with dire results is thousands of years old, and common to most cultures.

The most significant machines are not always the most complicated, nor the most powerful. The mass-produced bicycle of the nineteenth century brought personal transport within the reach of most wage earners, the "slavery" of the factories stem brought forth the freedom of the roads. Humans have always been ready to make that kind of deal, and the robot seems to be the bargain of the century; do we really know the terms, and can we afford the payments?

The archetypal rogue robot is perhaps the golem of Judaic myth. The word is used in the Talmud (the collection of rabbinical writings on scriptural, civic and moral matters dating back to the Babylonian and Egyptian captivities during the first and second millenia BC) to describe Adam as the shapeless clay into which the Creator breathed life. Rabbi Low of Prague in 1580 is supposed to have raised such a golem and employed him as a servant in the synagogue until his developing sense of identity and rebellion forced the Rabbi to return him to the clay at the age of thirteen.

Sanskrit myth of similar antiquity tells of the creation of a female humanoid called Tilottama whose beauty is such that two of the gods are killed fighting over her. Later stories tell of a mysterious smith creating lamas and monks of gold, kings and courtiers of bronze, melodious choirs of silver, and soldiers of bronze. By the Middle Ages such robots are common figures in Indian myth as the products of human artifice - mechanical marvels of wood and metal.

Greek legend tells of Pygmalion, king of Cyprus, who falls in love with Galatea, a beautiful ivory statue. Aphrodite, goddess of beauty, brings Galatea to life so that the king may marry her a happy outcome in this original version but one cursed with jealousy and resentment in subsequent re-tellings by W.S. Gilbert (*Pygmalion and Galatea*), G.B. Shaw (*Pygrnalion*) and Lerner and Loewe (*My Fair Lady*).

Later Greek myths feature the first robot engineers, Hephaestus and Daedalus. The former created many mechanical humanoids, foremost among them being Talos, the giant bronze guadian of the beaches od Crete who hugged his enemies to death in his red-hot bosom, but died when his vital fluids drained away through his heel. Daedalus was the legendary Athenian inventor who built the Cretan Labyrinth, invented the saw, the axe and the gimlet, and created a wooden sculpture that he brought to life by pouring mercury into its veins. His genius had tragic reward when his son, Icarus, took flight on Daedalus's marvelous wings, and, flying too close to the sun in his unthinkng trust, fell to his death. If we are to admire the Greeks in their slave-supported ease, we should at least assume some of their suspicion of technology.

The classic robot tragedy of modern western myth is, of course, Frankenstein's monster in Mary Shelley's story. The eponynnous scientist creates his golem from human flesh and animates it by lightning, only to see it become a child-killing monster which ultimately turns upon Frankenstein himself. Virtually every robot story written since has followed this pattern of creation, rebellion, and disaster.

These powerful stories obviously bespeak a deeply felt human urge which engineers have long since labored to fulfil Despite the Greek myths described above, the engineers of antiquity had neither the materials nor the methods necessary for robot engineering, though Hero of Alexander, inventor of the steam turbine, built ingenious mechanical tableaux powered by air or water featuring moving human and animal figures. The first true automaton seems to have been the mechanical duck of Jacques de Vaucanson, presented to the Academie Royale des Sciences in Paris in 1738. The duck flagged its wings, quacked, ate and shat; Academiciens' reactions are not recorded.

Working at the same time as de Vaucanson was the Swiss inventor Pierre Jacquet-Droz, creator of puppets and mechanical marvels. His most celebrated automaton, The Writer, survives in a Neuchatel museum, and is a beautiiully constructed model of a young man seated at a writing desk. He dips his pen into the inkwell and in good clerk's hand writes "Cogito Ergo Sum" ("I thing therefore I am"). This choice of apothegm is a fitting homage to its author, Rene Descartes, the seventeenth-century philosopher, since he himself is supposed to have created a mechanical servant-woman called Francine, who was thrown into the sea by a superstitious sailor.

As the engineering skills of the nineteenth century expanded the mechanical possibilities, so the range of simulacra and automata became more diverse: from The Turk, the mechanical chess-player that actually contained a small man, to the steam-driven man invented by George Moore in Britain in 1892, capable of a claimed 8 mph walk over level ground, wind-assisted. None of them, however, had any practical use; real working robots had to wait on the essentially twentieth-century developments af electricity, alloy and plastic technology, and, most important computers.

The story of modern robots, the semi-or pseudo-intelligent autonomous automata that the word really means to most of us, is actually the story of computers in mobil form. Without the decision-making logical power of computers, a mechanical man is just a moving curiosity; contrariwise, the computer is fine as a simple information processing device, but starts to be significantly useful when housed in some mechanical muscle. Just as robots had a long pre-history in myth and model-making, so the development of computers, which seems to begin only in the 1940s, actually stretches back hundreds of years - to the invention of the abacus, or counting frame, in about 2000 BC at a pinch, but certainly to the seventeenth century when Blaise Pascal and Gottfried Leibniz both invented mechanical calculating machines.

Just as robots had a long pre-history in myth and model-making, so the development of computers, which seems to begin only in the 1940s, actually stretches back hundreds of years - to the invention of the abacus, or counting frame, in about 2000 BC at a pinch, but certainly to the seventeenth century when Blaise Pascal and Gottfried Leibniz both invented mechanical calculating machines.

In the early years of the nineteenth century, Charles Babbage began work on his Analytical Engine, a hand--cranked calculator which embodied in its mechanical designs the essential principles of computer architecture and operation as we understand them today. He was frustrated largely by being born before technology had developed to allow him to contruct his designs. His companion, Ada Countess Lovelace, was an equally gifted mathematician and the first computer programmer - with nothing to program. Apart from her place in the history of computing, her name lives on in the computing language Ada, developed in the 1980s by the US Department of Defense. Another contemporary similarly cursed and blessed was George Boole who developed in 1847 the algebra that underlies the logic of all computers.

At the same time as Babbage and Lovelace were struggling with the unrealizable future, Joseph Marie Jacquard was making it possible while making a profit - doing well while doing good - from his automatic weaving loom. This was controlled by a deck of punched cards containing a program of movements and operations. The same idea was used by Hermann Hollerith in 1890 for the machine which he built to analyze the US census returns, and infinitely greater commercial success followed. For that census of 56 million people he charged the government 65 cents per 1000 returns. In 1924 his Tabulating Recording Company became International Business Machines - IBM - the most important single body in the history of computing.

It took a world war to impel the next step in computing: the administrative needs of centralized states running huge armies and production forces spurred the development of data-processing techniques and information technology; the technical needs of the gunners and bombardiers demanded computing machinery and the vast research efforts that produced radar and the atomic bomb also created significant new industries making advanced electronic components. In 1943 a British code-breaking team led by the brilliant mathematician, Alan Turing, built Colossus, the first recognizable computer. One hundred miles from their laboratories, the Germans were bombarding London with V 1 s - pilotless aircraft powered by rocket engines, steering themselves to the target by following radio beams and cutting their engines after they had measured a programed flight path from launch. This was both the golem returning to its clay and a fearsome robotic dawn.

With peace in 1945 came the first true electronic computers, from teams in Pennsylvania and Manchester, England. Dependent on the bulky, fragile thermionic valves, they were made obsolete almost immediately by the invention of the silicon transistor in 1948. This marvelous machine, this motionless lump of sand and tin, whose moving parts are electrons and "holes" in space-time, is the key to the computing door. Once it came into mass-production, the whole world of computers and robots and space exploration became not just possible but certain.

The significant steps from then to this day are easily described IBM's first computer and FORTRAN, the first popular computer programing language, were launched in 1957, and computers became affordable to business and the universities. In the 1960s the space research effort led to the miracles of miniaturization which are integrated circuiits. Matchboxes could now hold computing power that the warehouse-size computers of the 1940s could not match. In the 1970s these siliicon chips enabled computing's Henry Ford to produce the electronic Model T: Apple Corporation, owned by a young Californian named Steve Wozniak, produced and sold millions of cheap, robust, admirable personal computers. As the computer took over the living-rooms and studies of the West, so robots marched into factories of the East, having become commercial reality almost by stealth: George C. Devol took out American patents on the robot hand in 1961, Joseph Engelberger's Unimatiion Inc. installed its first robot in 1961, and the Japanese Industrial Robot Association was founded in 1971.

A robot killed a Japanese engineer 1981. This tragic event,despite its human cost and wealth of fearsome symbolisms, was actually a prosaic industrial accident involving not some latter-day Talos but a commercial robot arm - nothing like the hulking steel androgyne of modern robot myth, but instead a small flexible electric crane powered by electricity and controlled by its own built-in computer. The arm, not the android, is the commonest form of today's and tomorrow's robots - 25,000 in Japan, l5,000 in the USA, 8,000 in West Germany in 1985. It is to be found paint-spraying, welding and assembling in factories across the world. The mechanical humans of popular imagination exist today but really only as curiosities and entertainments-as yet. So powerful seems to be the cultural need for the walking, talking golem-Galatea,however, that we can almost expect Milton's two-handed engine at our doors any day now. (Not that he would recognize it, since he was actually talking metaphorically about theChurch of England).

The letters "RUR" on Capt. Richards robot allude to Karel Capeks1917 play, Rossum's Universal Robots in which the word "robot" was coined. The work involved in building such a curiosity, and its success as an entertainment, both testify to the hold on the public imagination of Capek's and other robot myths.

Appendix A

After the Crash

Whiskers has a very powerful software system that allows you to create not only programs and new commands, but even your own compiling words. This power doesn't come without a cost, however. You also have the capability to crash the system. You might change a critical pointer in memory the causes the PROM(software) to be delinked from the system. By running a utility called WIPE on your disk, and downloading the initialization file, you can bring him back.

1) Remove the top cover
2) With the front of the robot facing you, look at the large square chip that has 68HC11on it
3) On the right side are two jumpers and two positions, they are in the NORMAL position
4) Put the jumpers in the bootstrap position, turn the robot off then on.
5) Type in WIPE at the command line.
6) Type 1 to erase EEPROM and change config registers.
7) Type enter at the CURRENT WIPE SETTINGS screen.
8) Move the two jumpers that are next to the processor from NORMAL to BOOT STRAP.
9) Press reset on Whiskers, this will take a few seconds.
10) Press Y to run wipe again.
11) Type 2 to DEFEAT F68HC11 AUTOSTART
12) Type enter at the CURRENT WIPE SETTINGS screen.
13) Press reset on Whiskers, this will take a few seconds.
14) Move the two jumpers that are next to the processor from BOOT STRAP to NORMAL.
15) Turn Whiskers off then on again.
16) When you see the copyright screen, immediately press the Q key.

Appendix B

Error Codes

Code	Message	Description
0	?	The word is not in the dictionary
1	STACK EMPTY	You need a number on the stack to perform this operation
2	DICTIONARY FULL	You donot have any more room to define new words
4	NOT UNIQUE	The name you are defining is already in the dictionary Note: this message is ok as you can redefine previously Defined words.
7	FULL STACK	The stack is full
17	COMPILATION ONLY	You can only use this word/command during compiling a new word, not while you are in the interpretive mode
18	EXECUTION ONLY	The word must be used in the interpretive mode.
19	CONDITIONALS NOT PAIRED	Omitted words or incorrect nesting of conditionals
20	DEFINITION NOT FINISHED	You forgot the ; at the end of your definition
21	IN PROTECTED DICTIONARY	The word in question is in the PROM or cannot be forgotten
22	USE ONLY WHEN LOADING	Incorrect use of word
23	NO NAME	Attempt to create a definition without appropriate name

Appendix C

ASSEMBLER LANGUAGE
for Whiskers

Angelus Research Corp.
6344 Sugar Pine Circle, #98
Angelus Oaks, California 92305
(909) 794-8325

Last Revision: January 19, 2015

Note: References to downloading the assembler in this manual are to be disregarded. The assembler is already on the PROM.

ASSEMBLER LANGUAGE for Whiskers

TABLE OF CONTENTS

Introduction

User Reference

Appendix A

Appendix B

Appendix C

Appendix D

YOUR NOTES

USER REFERENCE

Execution

1. (Your Macro definitions)
2. (Your program)

Command Syntax

Although HC11 Assembler mnemonics and annotation are used, the actual syntax is Reverse Polish Notation. This is required because the FORTH input interpreter is being used to read the source. Figure 1 demonstrates the differences between the two. All of the op-code mnemonics for this compiler are defined to end with a comma. This is done to differentiate between mnemonics and hex addresses. The list of recognized mnemonics is included in Appendix C.

CONVENTIONAL	*ASM6811*
LDAB # 25	25 # LDAB,

FIGURE 1
ADDRESSING MODES

Figure 2 displays the explicit addressing modes this compiler will recognize. Chapter 6 of the Motorola Document cited in the introduction to this manual contains an excellent detailed discussion of this idiosyncrasy of the Motorola HC11 CPU.

The operand in an immediate mode instruction is the same width as the referenced accumulator, resulting in either a single, or two byte operand. This gives a variable instruction length of two to four bytes.

Direct address mode uses a single byte offset, the high order address byte is assumed to be binary zeroes, thus limiting this mode to addresses contained in page 0 of memory. Only four commands using this mode are three bytes long, all of the rest are two.

Extended address mode uses an explicit address, thus requiring two bytes for the operand. Only four commands using this mode are four bytes long, all of the rest are three.

For indexed mode, the effective address is computed as the sum of the 8 bit operand and the contents of the indicated index register. Instructions using this addressing mode will be either two or three bytes in length.

#	Immediate Mode
DIR	Direct Address Mode, (Page 0 only)
EXT	Extended Address Mode
,X	Indexed by Register X
,Y	Indexed by Register Y

FIGURE 2

Language Extensions

Figure 3 lists the additional mnemonics recognized by the compiler. These are implemented primarily to aid program structure and are utilized in the same general manner as their FORTH counterparts. The continued use of the trailing comma as part of the Operation Code allows the compiler to keep these special clauses separate from the actual FORTH words, which are still available.

'THEN,' is equivalent to the ENDIF clause used in some dialects of FORTH. 'IF,' consumes and tests the top cell on the data stack at run time, requiring the test condition to precede the 'IF,' statement. In FORTH, unlike many other high level languages, this clause actually means "IF SO, ...". If there are ANY bits set in this cell, control transfers to the statement(s) immediately following 'IF,'. Otherwise the first group of statements is skipped, and control is transferred to the statement following the 'NEXT,' clause. (Hence, the ENDIF used by some older versions of FORTH.)

Inclusion of an 'ELSE,' clause makes the execution of this construct a little less obvious. Basically, the group of statements immediately after the 'IF,' and the group following the 'ELSE,' can be conceptualized as two mutually exclusive subroutines, where completion of either will result in a transfer of control to the point marked by the 'NEXT,'. The value of the top cell on the data stack controls which group of statements gets executed. Figure 5 contains an example of 'IF,', 'ELSE,', 'THEN,'.

'BEGIN', is the marker that indicates the beginning of a loop. It is followed by either 'UNTIL,' or 'AGAIN,', who do the actual transfer of control back to the location specified by 'BEGIN,'. 'UNTIL,' also consumes and tests the top cell on the data stack at run time like 'IF,'. Only if there are NO bits set in this cell, is control transferred to the location indicated by 'BEGIN,'. 'AGAIN,' is functionally equivalent to "0 UNTIL," and requires an explicit escape from within the loop.

Although described as a "single set" for discussion simplicity, both of the above code construct pairs can be nested (and intermixed) as deep as is required, as long as they are kept properly paired. It is considered good form to indent statements based on the level of nesting, making it easier to discern which "pairs" of clauses match each other when reading the source code.

'.CLR.IF,' and '.SET.IF,' are conditional clear and set respectively and are the single command implementation of 'IF,' and 'CLR,' or 'IF,' and 'SET,'.

BEGIN,
UNTIL,
AGAIN,
.CLR.IF,
IF,
ELSE,
THEN,
.SET.IF

FIGURE 3

Figure 4 lists the mnemonics pre-defined for use with these structured decision points. For the most part, these are the same codes as the assembler mnemonics. In practice, these precede the decision point to indicate the type of assembler test to be selected.

.GT.	signed	Greater Than
.GE.	signed	Greater Than/Equal To
.LE.	signed	Less Than/Equal To
.LT.	signed	Less Than
.HI.	unsigned	Higher
.HS.	unsigned	Higher/Same
.LS.	unsigned	Lower/Same
.LO.	unsigned	Lower
.CS.	simple	Carry Set
.CC.	simple	Carry Clear
.--.	simple	Negative
.++.	simple	Positive/Zero
.VS.	simple	Overflow Set
.VC.	simple	Overflow Clear
.EQ.	simple	Equal
.NE.	simple	Not Equal
.FL.	simple	False
.TR.	simple	True
.NOT.		Invert Logic

FIGURE 4

Figure 5 demonstrates the correct syntax for a compare followed by a conditional branch. Notice that the actual generated code is the opposite of that specified, in order to accommodate the logic of the If, ELSE, THEN source statement, and the single branch vector of the machine command. In this particular example, <stmt-2> is selected when the condition is satisfied, and <stmt-1> when it is not.

ASM6811

```
<var-1> CMPB,
.GT. IF,
        <stmt-1> ELSE,
        <stmt-2> THEN,
```

ASSEMBLER EQUIVALENT

```
CMPB <var-1>
BLE <stmt-1>
<stmt-2>
```

FIGURE 5

Figure 6 lists the pre-defined macros included within this compiler. These are all references to specific internal FORTH system variables, and their use allows named constants rather than absolute addresses or values.

TOP	Address of first item on User Data Stack
SEC	Address of second item on User Data Stack
PUSHD	Address of routine to replace "TOP" (dbl)item and execute next (FORTH) word
NEXTSD	Address of routine to add "TOP" (dbl)item and execute next (FORTH) word
PUT	Addr of routine to replace "TOP" (sing)item and execute next (FORTH) word
NEXT	Addr of routine to execute next (FORTH) word
NEXT3	Address within routine to execute next (FORTH) word (using current pointer +2)
NEXT1	Address within routine to execute next (FORTH) word (using current pointer)
NEXT2	Address within routine to jump direct to next (FORTH) word
POP	Address of routine to remove "TOP" item
POPTWO	Address of routine to remove "TOP" 2 items
W	Address of word pointer
IP	Address of instruction pointer
UP	Address of user area pointer

Figure 6

Macro Facility

The compiler simulates this ability by allowing the definition of new assembler commands, and linking them to the system using the internal facilities of FORTH. This accomplishes much the same intent as for a macro, but with less memory overhead, and a greater flexibility in choice of language. Since we are using the Forth compilers native capability, this "macro" can be written in "assembler", FORTH, or Whiskers own Robot Control Language.

To add a new "macro" to the language, it is necessary to invoke the assembler vocabulary with the following compiler directive:

```
ASSEMBLER DEFINITIONS
```

Then add the necessary code fragments, and switch back to the FORTH vocabulary with the following compiler directive:

```
FORTH DEFINITIONS
```

The code fragment contained in Figure 7 shows the correct method of adding two "macros". They are both useful in reducing the number of instructions required to be written by the programmer in the main line of the program. The first macro loads a value from the FORTH Data Stack into the HC11's D Register. The second macro moves a value to the FORTH Data Stack from the HC11's D Register. In both cases, the Y Register, which is used as the FORTH Stack Pointer, must be manipulated either before or after the transfer. Use of these macros relieves the programmer from having to remember the order in which the instructions must be executed to preclude having an unprotected stack during high level interrupts.

```
ASSEMBLER DEFINITIONS

:  POPD, TOP LDD, INY, INY, ;
:  PUTD, DEY, DEY, TOP STD, ;

FORTH DEFINITIONS
```

Figure 7

Carefully note that the "macros" are defined with a name that contains a trailing comma, this convention is strongly suggested for those "macros" that are physical commands as opposed to "macros" used to define dynamic memory locations, etc. Each of these "macros" contains reference to another "macro", 'TOP', which is simply a mnemonic reference to a dynamic memory location. After reading this code fragment into the compiler, 'POPD,' and 'PUTD,' may now be used as part of the assembly language, even to the extent of being referenced within other "macros".

APPENDIX A

This Appendix contains example code fragments demonstrating the use of ASM6811 in producing useful programs. Named examples ending with a single quote produce code representitive of that in the kernel with similar names. These should be useful for point of reference and comparison.

```
( ASM6811 VERSION 1.11 05/14/93 )

CODE SWAP'
  0 ,Y LDD,             ( GET TOP WORD INTO D )
  2 ,Y LDX,             ( GET SEC WORD INTO X )
  2 ,Y STD,             ( PUT D INTO SEC WORD )
  0 ,Y STX,             ( PUT X INTO TOP WORD )
  NEXT JMP,             ( RETURN CONTROL TO FORTH, GO TO NEXT WORD )
END-CODE

CODE SWAP"              ( SAME AS SWAP AND SWAP' BUT USING MACROS TOP & SEC )
  TOP LDD,              ( GET TOP WORD INTO D )
  SEC LDX,              ( GET SEC WORD INTO X )
  SEC STD,     ( PUT D INTO SEC WORD )
  TOP STX,     ( PUT X INTO TOP WORD )
  NEXT JMP,             ( RETURN CONTROL TO FORTH, GO TO NEXT WORD )
END-CODE

CODE NOT'
  TOP    COM,           ( COMPLIMENT BYTE AT TOP OF STACK   MSB )
  TOP 1+ COM,           ( COMPLEMENT BYTE AT TOP OF STACK+1 LSB )
  NEXT JMP,             ( RETURN CONTROL TO FORTH, GO TO NEXT WORD )
END-CODE

CODE-SUB 2*'
  TOP 1+ ASL,           ( ARITHMETIC SHIFT LEFT LSB )
  TOP    ROL,           ( ARITHMETIC SHIFT LEFT MBD WITH CARRY )
  RTS,                  ( RETURN FROM CODE-SUB TO ITS INTERPRETER BACK TO FORTH )
END-CODE

CODE @'
  TOP LDX,              ( GET ADDRESS OF VALUE )
  0 ,X LDD,             ( GET VALUE FROM ADDRESS )
  PUT JMP,     ( REPLACE ADDRESS ON STACK WITH VALUE, GO TO NEXT WORD )
END-CODE

CODE !'
  TOP LDX,              ( GET ADDRESS TO STORE VALUE AT )
  SEC LDD,              ( GET VALUE TO STORE )
  0 ,X STD,             ( PUT VALUE IN ADDRESS )
  POPTWO JMP,           ( TAKE ADDRESS & VALUE OFF STACK, GO TO NEXT WORD )
END-CODE
```

```
CODE C@'
  TOP LDX,              ( GET ADDRESS OF BYTE VALUE )
  0 ,X LDAB,            ( GET BYTE VALUE FROM ADDRESS )
  CLRA,                 ( CLEAR MSB )
  PUT JMP,      ( REPLACE ADDRESS ON STACK WITH VALUE, GO TO NEXT WORD )
END-CODE

CODE C!'
  TOP LDX,              ( GET ADDRESS TO STORE BYTE VALUE AT )
  SEC 1+ LDAB,          ( GET BYTE VALUE TO STORE )
  0 ,X STAB,    ( PUT BYTE VALUE IN ADDRESS )
  POPTWO JMP,           ( TAKE ADDRESS & VALUE OFF STACK, GO TO NEXT WORD )
END-CODE

CODE +'
  TOP LDD,              ( GET FIRST VALUE TO ADD )
  SEC ADDD,             ( ADD SECOND VALUE TO IT )
  SEC STD,      ( PUT VALUE IN SECOND )
  POP JMP,      ( TAKE FIRST VALUE OFF STACK, GO TO NEXT WORD )
END-CODE

CODE DUP'
  TOP LDD,              ( GET FIRST VALUE TO ADD )
  PUSHD JMP,            ( PUSH NEW VALUE ONTO STACK, GO TO NEXT WORD )
END-CODE

CODE MIN'
  TOP LDD,              ( GET FIRST VALUE )
  SEC CPD,              ( COMPARE TO SECOND VALUE
  .LE.                  ( BRANCH IF GREATER THAN )
  IF,
    SEC STD,    ( IF NOT GREATER THAN REPLACE )
  THEN,
  POP JMP,      ( REMOVE FIRST VALUE FROM STACK, GO TO NEXT WORD )
END-CODE

CODE 0='
  TOP LDD,              ( GET VALUE FROM STACK
  .NE.                  ( SEE IF IT WAS NOT EQUAL TO ZERO )
  IF,
    CLRA,               ( IF NOT EQUAL, LEAVE FALSE INDICATION, 0 )
    CLRB,
  ELSE,
    -1 # LDD,           ( IF EQUAL, LEAVE TRUE INDICATION, FFFF )
  THEN,
  PUT JMP,      ( REPLACE FIRST VALUE WITH BOOLEAN, GO TO NEXT WORD )
END-CODE

CODE I'
  TSX,                  ( GET RETURN STACK POINTER INTO X )
  0 ,X LDD,             ( GET COPY OF TOP OF RETURN STACK INTO D )
  PUSHD JMP,            ( SAVE VALUE ON DATA STACK, GO TO NEXT WORD )
END-CODE

CODE J'
  TSX,                  ( GET RETURN STACK POINTER INTO X )
  4 ,X LDD,             ( GET COPY OF TOP OF RETURN STACK INTO D )
```

```
  PUSHD JMP,          ( SAVE VALUE ON DATA STACK, GO TO NEXT WORD )
END-CODE

CODE K'
  TSX,                ( GET RETURN STACK POINTER INTO X )
  8 ,X LDD,           ( GET COPY OF TOP OF RETURN STACK INTO D )
  PUSHD JMP,          ( SAVE VALUE ON DATA STACK, GO TO NEXT WORD )
END-CODE

CODE EXECUTE'
  TOP LDX,            ( GET VALUE OF CFA OF WORD TO EXECUTE OFF STACK )
  INY,
  INY,                ( TAKE VALUE OFF STACK )
  NEXT2 JMP,          ( EXECUTE NEXT WORD
END-CODE

CODE KEY'
  UP LDX,             ( GET USER POINTER )
  10 ,X LDX,          ( GET UKEY FROM USER AREA )
  0 ,X JSR,           ( JUMP SUBROUTINE "INDIRECT" THROUGH UKEY )
  NEXT JMP,           ( GO TO NEXT WORD )
END-CODE

CODE EMIT'
  UP LDX,             ( GET USER POINTER )
  12 ,X LDX,          ( GET UEMIT FROM USER AREA )
  0 ,X JSR,           ( JUMP SUBROUTINE "INDIRECT" THROUGH UEMIT )
  NEXT JMP,           ( GO TO NEXT WORD )
END-CODE

CODE ?TERMINAL'
  UP LDX,             ( GET USER POINTER )
  14 ,X LDX,          ( GET U?TERMINAL FROM USER AREA )
  0 ,X JSR,           ( JUMP SUBROUTINE "INDIRECT" THROUGH U?TERMINAL )
  NEXT JMP,           ( GO TO NEXT WORD )
END-CODE
```

```
CODE-SUB KEYSUB'
  UP LDX,               ( GET USER POINTER )
  0C ,X LDX,            ( GET KBOFF FROM USER AREA )
  BEGIN,
    PSHX,               ( SAVE X FOR AWHILE )
    0 ,X LDX,           ( USE KBOFF TO GET "INDIRECT" ADDRESS OF STATUS REG. )
    0 ,X LDAB,          ( GET STATUS REGISTER )
    PULX,               ( RESTORE X )
    2 ,X ANDB,          ( GET PATTERN TO AND WITH STATUS VALUE )
    3 ,X EORB,          ( GET PATTERN TO XOR WITH STATUS VALUE )
    .NE.
  UNTIL,                ( CONTINUE LOOPING TIL NON ZERO RESULT )
  4 ,X LDX,             ( USE KBOFF TO GET "INDIRECT" ADDRESS OF DATA REG. )
  0 ,X LDAB,            ( GET DATA REGISTER )
  CLRA,                 ( CLEAR MSB )
  DEY,                  ( OPEN HOLE ON STACK )
  DEY,
  TOP STD,      ( PUT DATA "KEY" ON STACK )
  RTS,                  ( RETURN TO FORTH OR CALLING ROUTINE )
END-CODE

CODE-SUB EMITSUB'
  UP LDX,               ( GET USER POINTER )
  0E ,X LDX,            ( GET OUTOFF FROM USER AREA )
  BEGIN,
    PSHX,               ( SAVE X FOR AWHILE )
    0 ,X LDX,           ( USE KBOFF TO GET "INDIRECT" ADDRESS OF STATUS REG. )
    0 ,X LDAB,          ( GET STATUS REGISTER )
    PULX,               ( RESTORE X )
    2 ,X ANDB,          ( GET PATTERN TO AND WITH STATUS VALUE )
    3 ,X EORB,          ( GET PATTERN TO XOR WITH STATUS VALUE )
    .NE.
  UNTIL,                ( CONTINUE LOOPING TIL NON ZERO RESULT )
  TOP 1+ LDAB,          ( GET CHAR FROM STACK )
  4 ,X LDX,             ( USE OUTOFF TO GET "INDIRECT" ADDRESS OF DATA REG. )
  0 ,X STAB,    ( WRITE CHAR TO DATA REGISTER )
  INY,                  ( CLOSE HOLE ON STACK )
  INY,
  RTS,                  ( RETURN TO FORTH OR CALLING ROUTINE )
END-CODE
```

```
CODE-SUB QTSUB'
  UP LDX,               ( GET USER POINTER )
  0C ,X LDX,            ( GET KBOFF FROM USER AREA )
  PSHX,                 ( SAVE X FOR AWHILE )
  0 ,X LDX,             ( USE KBOFF TO GET "INDIRECT" ADDRESS OF STATUS REG. )
  0 ,X LDAB,            ( GET STATUS REGISTER )
  PULX,                 ( RESTORE X )
  2 ,X ANDB,            ( GET PATTERN TO AND WITH STATUS VALUE )
  3 ,X EORB,            ( GET PATTERN TO XOR WITH STATUS VALUE )
  .NE.
  IF,                   ( BRANCH IF EQUAL ZERO RESULT )
    FF # LDAB,          ( LOAD TRUE RESULT )
  THEN,                 ( CLEAR MSB )
  TBA,                  ( PUT BOOLEAN IN BOTH HALVES OF D )
  DEY,                  ( OPEN HOLE ON STACK )
  DEY,
  TOP STD,      ( PUT DATA "KEY" ON STACK )
  RTS,                  ( RETURN TO FORTH OR CALLING ROUTINE )
END-CODE

( TO INSTALL NEW KEY ROUTINES )
( ' KEYSUB' @  16 ! ( PUTS KEYSUB  PFA INTO VECTOR )
( ' EMITSUB' @ 18 ! ( PUTS EMITSUB PFA INTO VECTOR )
( ' QTSUB' @   1A ! ( PUTS QTSUB   PFA INTO VECTOR )
```

Following is an example taken from the Max-FORTH Users Manual under the section titled "CODE DEFINITIONS". First the machine coded example from the manual is repeated, followed by the translation in ASM6811 format and finally its equivalent in high level FORTH. The function is to read all four channels of the A/D and leave them on the Data Stack.

```
( EXAMPLE FROM MANUAL )

CODE-SUB READ-A/D-CH0-3
      CE C, B030 ,              ( LDX $B030 )
                                ( SET ADCTL FOR MULT READINGS, STRT CONV )
      86 C, 10 C,               ( LDAA # 10 )
      A7 C, 00 C,               ( STAA 0,X , $B030 )
                                ( WAIT UNTIL CCF SET )
                                ( SPIN )
 1F C, 00 C, 80 C, FC C,        ( BRCLR 0,80,SPIN )
      4F C,                     ( CLRA )
                                ( TAKE DATA, OPEN STACK, STORE DATA )
      E6 C, 01 C,               ( LDAB 1,X )
      18 C, 09 C,               ( DEY )
      18 C, 09 C,               ( DEY )
      18 C, ED C, 00 C,         ( STD 0,Y )
      E6 C, 02 C,               ( LDAB 2,X )
      18 C, 09 C,               ( DEY )
      18 C, 09 C,               ( DEY )
      18 C, ED C, 00 C,         ( STD 0,Y )
      E6 C, 03 C,               ( LDAB 3,X )
      18 C, 09 C,               ( DEY )
      18 C, 09 C,               ( DEY )
      18 C, ED C, 00 C,         ( STD 0,Y )
      E6 C, 04 C,               ( LDAB 4,X )
      18 C, 09 C,               ( DEY )
      18 C, 09 C,               ( DEY )
      18 C, ED C, 00 C,         ( STD 0,Y )
      39 C,                     ( RTS)
END-CODE
```

```
( EXAMPLE FOR ASM6811 WITH MACRO )
ASSEMBLER DEFINITIONS

: PUTD, DEY, DEY, TOP STD, ;

FORTH DEFINITIONS

CODE-SUB READ-A/D-CH0-3-LL
  B030 LDX,
                              ( SET ADCTL FOR MULT READINGS, STRT CONV )
  10 # LDAA,
  0 ,X STAA,
                              ( WAIT UNTIL CCF SET )
                              ( -4 80 0 ,X BITCLR, )
                              ( This ^ is one way of doing it )
                              ( The BEGIN, UNTIL, loop is shown below to be closer to high level way )
  BEGIN,
    B030 LDAA,
    80 # ANDA,
    .NE.
  UNTIL,
  CLRA,
                              ( TAKE DATA, OPEN STACK, STORE DATA )
  1 ,X LDAB,
  PUTD,
  2 ,X LDAB,
  PUTD,
  3 ,X LDAB,
  PUTD,
  4 ,X LDAB,
  PUTD,
  RTS,
END-CODE

( EXAMPLE IN HIGH LEVEL )
: READ-A/D-CH0-3-HL
  10 B030 C!
                              ( SET ADCTL FOR MULT READINGS, STRT CONV )
                              ( WAIT UNTIL CCF SET )
  BEGIN
    B030 C@ 80 AND
  UNTIL
                              ( TAKE DATA )
  B031 C@
  B032 C@
  B033 C@
  B034 C@
;
```

APPENDIX B

This Appendix contains a listing of the distributed source code

COMPILER

```
( ASM6811 )

( COLD


( FORGET TASK
( HEX

( C400 DP !
( C100 1C !
( 50 1E !

( VERSION 1.11  05/14/93 )
: CODE-SUB [COMPILE] CODE-SUB [COMPILE] ASSEMBLER ; IMMEDIATE

ASSEMBLER DEFINITIONS

( NOTE: VARIABLE MODE NEEDS TO BE ASSIGNED TO RAM IF THIS TO BE
EPROM/ROM'ed )
( IE: 8 CONSTANT MODE )
VARIABLE MODE
: #   00 MODE ! ;
: DIR 10 MODE ! ;
: ,X  20 MODE ! ;
: ,Y 120 MODE ! ;
: EXT 30 MODE ! ;
EXT ( INITIALIZE MODE VARIABLE )

: ?# MODE @ 0= ;
: ?DIR MODE @ 10 = ;
: ?,X MODE @  20 = ;
: ?,Y MODE @ 120 = ;
: ?EXT MODE @ 30 = ;
: MODE-LSB MODE @ FF AND ;
: ERROR EXT ' ID. CFA 4A + EXECUTE ;
: RANGE-C, DUP FF00 AND IF HERE 1+ - DUP ABS FF00 AND
                           IF 3 ERROR THEN THEN C, ;

: CPU <BUILDS C, DOES> C@ C, EXT ; ( SINGLE BYTE OP-CODE )
: PG-2-CPU <BUILDS C, DOES> 18 C, C@ C, EXT ; ( 18 AND SINGLE BYTE OP-CODE )

: HHLL/LL, ?EXT IF , ELSE C, THEN EXT ;
: 2HHLL/LL, ?EXT ?# OR IF , ELSE C, THEN EXT ;
: MODE-ADJ, ( a n --- a )
  ?EXT IF OVER FF00 AND 0= IF DIR THEN THEN MODE-LSB OR C, ;
: 18,Y ?,Y IF 18 C, THEN ;
: SOK? ?# IF 3 ERROR THEN ;

: xx12-CPU <BUILDS C, DOES> 18,Y ?# ?DIR OR IF 3 ERROR THEN
```

```
                    C@ MODE-LSB OR C, HHLL/LL, ;
: 1112-CPU <BUILDS C, DOES> 18,Y C@ MODE-ADJ, HHLL/LL, ;
: x112-CPU <BUILDS C, DOES> 18,Y SOK? C@ MODE-ADJ, HHLL/LL, ;
: 2112-CPU <BUILDS C, DOES> 18,Y C@ MODE-ADJ, 2HHLL/LL, ;

: (OP-DD-MM) 18,Y SOK? MODE-LSB 20 = IF C OR THEN C, C, C, ;
: OP-RR <BUILDS C, DOES> C@ C, RANGE-C, EXT ;
: OP-DD-MM <BUILDS C, DOES> C@ (OP-DD-MM) EXT ;
: OP-DD-MM-RR <BUILDS C, DOES> C@ (OP-DD-MM) RANGE-C, EXT ;

: CPX, ?,Y IF CD C, THEN 8C MODE-ADJ, 2HHLL/LL, ;
: LDX, ?,Y IF CD C, THEN CE MODE-ADJ, 2HHLL/LL, ;
: STX, ?,Y IF CD C, THEN SOK? CF MODE-ADJ,  HHLL/LL, ;
: CPY, ?,X IF 1A ELSE 18 THEN C, 8C MODE-ADJ, 2HHLL/LL, ;
: LDY, ?,X IF 1A ELSE 18 THEN C, CE MODE-ADJ, 2HHLL/LL, ;
: STY, ?,X IF 1A ELSE 18 THEN C, SOK? CF MODE-ADJ, HHLL/LL, ;
: CPD, ?,Y IF CD C, ELSE 1A C, THEN 83 MODE-ADJ, 2HHLL/LL, ;

00 CPU TEST,
01 CPU NOP,
02 CPU IDIV,
03 CPU FDIV,
04 CPU LSRD,
05 CPU ASLD, 05 CPU LSLD,
06 CPU TAP,
07 CPU TPA,
08 CPU INX,
09 CPU DEX,
0A CPU CLV,
0B CPU SEV,
0C CPU CLC,
0D CPU SEC,
0E CPU CLI,
0F CPU SEI,
10 CPU SBA,
11 CPU CBA,
12 OP-DD-MM-RR BRSET,
13 OP-DD-MM-RR BRCLR,
14 OP-DD-MM BSET,
15 OP-DD-MM BCLR,
16 CPU TAB,
17 CPU TBA,
( 18 PAGE 2 )
19 CPU DAA,
( 1A PAGE 3 )
1B CPU ABA,
( 1C BSET,
( 1D BCLR,
( 1E BRSET,
( 1F BRCLR,
20 OP-RR BRA,
21 OP-RR BRN,
22 OP-RR BHI,
23 OP-RR BLS,
24 OP-RR BCC,
24 OP-RR BHS,
25 OP-RR BCS,
25 OP-RR BLO,
```

```
26 OP-RR BNE,
27 OP-RR BEQ,
28 OP-RR BVC,
29 OP-RR BVS,
2A OP-RR BPL,
2B OP-RR BMI,
2C OP-RR BGE,
2D OP-RR BLT,
2E OP-RR BGT,
2F OP-RR BLE,
30 CPU TSX,
31 CPU INS,
32 CPU PULA,
33 CPU PULB,
34 CPU DES,
35 CPU TXS,
36 CPU PSHA,
37 CPU PSHB,
38 CPU PULX,
39 CPU RTS,
3A CPU ABX,
3B CPU RTI,
3C CPU PSHX,
3D CPU MUL,
3E CPU WAI,
3F CPU SWI,
40 CPU NEGA,
( 41 NC )
( 42 NC )
43 CPU COMA,
44 CPU LSRA,
( 45 NC )
46 CPU RORA,
47 CPU ASRA,
48 CPU ASLA,
49 CPU ROLA,
4A CPU DECA,
( 4B NC )
4C CPU INCA,
4D CPU TSTA,
( 4E NC )
4F CPU CLRA,
50 CPU NEGB,
( 51 NC )
( 52 NC )
53 CPU COMB,
54 CPU LSRB,
( 55 NC )
56 CPU RORB,
57 CPU ASRB,
58 CPU ASLB,
59 CPU ROLB,
5A CPU DECB,
( 5B NC )
5C CPU INCB,
5D CPU TSTB,
( 5E NC )
5F CPU CLRB,
( 60-7F )
```

```
40 xx12-CPU NEG,
43 xx12-CPU COM,
44 xx12-CPU LSR,
46 xx12-CPU ROR,
47 xx12-CPU ASR,
48 xx12-CPU ASL,
49 xx12-CPU ROL,
4A xx12-CPU DEC,
4C xx12-CPU INC,
4D xx12-CPU TST,
4E xx12-CPU JMP,
4F xx12-CPU CLR,
( 80-BF )
80 1112-CPU SUBA,
81 1112-CPU CMPA,
82 1112-CPU SBCA,
83 2112-CPU SUBD,
84 1112-CPU ANDA,
85 1112-CPU BITA,
86 1112-CPU LDAA,
87 x112-CPU STAA,
88 1112-CPU EORA,
89 1112-CPU ADCA,
8A 1112-CPU ORAA,
8B 1112-CPU ADDA,
( 8C CPX,
8D x112-CPU JSR,
8E 2112-CPU LDS,
8F x112-CPU STS,
8F CPU XGDX,
( C0-FF )
C0 1112-CPU SUBB,
C1 1112-CPU CMPB,
C2 1112-CPU SBCB,
C3 2112-CPU ADDD,
C4 1112-CPU ANDB,
C5 1112-CPU BITB,
C6 1112-CPU LDAB,
C7 x112-CPU STAB,
C8 1112-CPU EORB,
C9 1112-CPU ADCB,
CA 1112-CPU ORAB,
CB 1112-CPU ADDB,
CC 2112-CPU LDD,
CD x112-CPU STD,
( CE LDX, )
( CF STX, )
CF CPU STOP,

08 PG-2-CPU INY,
09 PG-2-CPU DEY,
30 PG-2-CPU TSY,
35 PG-2-CPU TYS,
38 PG-2-CPU PULY,
3A PG-2-CPU ABY,
3C PG-2-CPU PSHY,

8F PG-2-CPU XGDY,
```

```
8D OP-RR BSR,


: TOP ,Y    0  ;     (  ADDRESS THE BOTTOM OF THE STACK  *)
: SEC ,Y    2  ;         ( ADDRESS SECOND ITEM ON STACK  *)

: ?EXEC STATE @ IF 12 ERROR THEN ;
: ?PAIRS - IF 13 ERROR THEN ;

: BEGIN,   HERE  1  ;
: UNTIL,   ?EXEC >R 1 ?PAIRS R> C, HERE  1+ - C, ;
: AGAIN,   20 UNTIL, ;
: IF,      C, HERE 0 C, 2 ;
: THEN,    ?EXEC 2 ?PAIRS HERE OVER 1+ - SWAP C! ;
: ELSE,    2 ?PAIRS HERE 1+ 0 BRA,
           SWAP HERE OVER 1+ - SWAP C! 2 ;
: .NOT. 1 XOR ;                    ( REVERSE ASSEMBLY TEST )

20 CONSTANT .FL.
21 CONSTANT .TR.
22 CONSTANT .LS.
23 CONSTANT .HI.
24 CONSTANT .CS.
24 CONSTANT .LO.
25 CONSTANT .CC.
25 CONSTANT .HS.
26 CONSTANT .EQ.
27 CONSTANT .NE.
28 CONSTANT .VS.
29 CONSTANT .VC.
2A CONSTANT .--.
2B CONSTANT .++.
2C CONSTANT .LT.
2D CONSTANT .GE.
2E CONSTANT .LE.
2F CONSTANT .GT.

: BOK? ?# ?EXT OR IF 3 ERROR THEN ;
: BIT-BR <BUILDS C, DOES>
  C@ 18,Y BOK? MODE-LSB 10 - IF C OR THEN C, C,
  C, 2 C, 20 C, HERE 0 C, 2 ; IMMEDIATE

13 BIT-BR .CLR.IF,
12 BIT-BR .SET.IF,

' @  CFA FE43 FE22 - + CONSTANT PUSHD
' @  CFA FE47 FE22 - + CONSTANT NEXTSD
' @  CFA FE47 FE22 - + CONSTANT PUT
' @  CFA FE4A FE22 - + CONSTANT NEXT
' @  CFA FE4C FE22 - + CONSTANT NEXT3
' @  CFA FE50 FE22 - + CONSTANT NEXT1
' @  CFA FE52 FE22 - + CONSTANT NEXT2
' 1+ CFA FC97 FC7C - + CONSTANT POP
' 1+ CFA FC93 FC7C - + CONSTANT POPTWO

0 CONSTANT W
2 CONSTANT IP
```

4 CONSTANT UP

FORTH DEFINITIONS

APPENDIX C

This Appendix contains lists of the compiler recognized operation code mnemonics in various sort orders.

Ascending Order by Generated Op Code
Single Byte Codes

00	TEST,	Test Operation
01	NOP,	No Operation
02	IDIV,	Integer Divide
03	FDIV,	Fractional Divide
04	LSRD,	Logical Shift Right Double Accumulator
05	ASLD,	Arithmetic Shift Left Double Accumulator
05	LSLD,	Logical Shift Left Double
06	TAP,	Transfer from Accum A to Condition Code Reg.
07	TPA,	Transfer from Condition Code Reg. to Accum A
08	INX,	Increment Index Register X
09	DEX,	Decrement Index Register X
0A	CLV,	Clear Twos-Complement Overflow Bit
0B	SEV,	Set Twos-Complement Overflow Bit
0C	CLC,	Clear Carry
0D	SEC,	Set Carry
0E	CLI,	Clear Interrupt Mask
0F	SEI,	Set Interrupt Mask
10	SBA,	Subtract Accumulators
11	CBA,	Compare Accumulators
12	BRSET,	Branch if Bit(s) Set
13	BRCLR,	Branch if Bit(s) Clear
14	BSET,	Set Bit(s) in Memory
15	BCLR,	Clear Bit(s) in Memory
16	TAB,	Transfer from Accumulator A to Accumulator B
17	TBA,	Transfer from Accumulator B to Accumulator A
18		(PAGE 2)
19	DAA,	Decimal Adjust Accumulator A
1A		(PAGE 3)
1B	ABA,	Add Accumulator B to Accumulator A
1C	BSET,	Set Bit(s) in Memory (Indexed, X)
1D	BCLR,	Clear Bit(s) in Memory (Indexed, X)
1E	BRSET,	Branch if Bit(s) Set (Indexed, X)
1F	BRCLR,	Branch if Bit(s) Clear (Indexed, X)
20	BRA,	Branch Always
21	BRN,	Branch Never
22	BHI,	Branch if Higher
23	BLS,	Branch if Lower or Same
24	BCC,	Branch if Carry Clear
24	BHS,	Branch if Higher or Same
25	BCS,	Branch if Carry Set
25	BLO,	Branch if Lower
26	BNE,	Branch if Not Equal to Zero
27	BEQ,	Branch if Equal
28	BVC,	Branch if Overflow Clear
29	BVS,	Branch if Overflow Set
2A	BPL,	Branch if Plus

(Numeric, continued)

2B	BMI,	Branch if Minus
2C	BGE,	Branch if Greater than or Equal to Zero
2D	BLT,	Branch if Less than Zero
2E	BGT,	Branch if Greater than Zero
2F	BLE,	Branch if Less than or Equal to Zero
30	TSX,	Transfer from Stack Pointer to Index Reg X
31	INS,	Increment Stack Pointer
32	PULA,	Pull Data from Stack (into Accumulator A)
33	PULB,	Pull Data from Stack (into Accumulator B)
34	DES,	Decrement Stack Pointer
35	TXS,	Transfer from Index Reg X to Stack Pointer
36	PSHA,	Push Data onto Stack (from Accumulator A)
37	PSHB,	Push Data onto Stack (from Accumulator B)
38	PULX,	Pull Index Register X from Stack
39	RTS,	Return from Subroutine
3A	ABX,	Add Accumulator B to Index Register X
3B	RTI,	Return from Interrupt
3C	PSHX,	Push Index Register X onto Stack
3D	MUL,	Multiply Unsigned
3E	WAI,	Wait for Interrupt
3F	SWI,	Software Interrupt
40	NEGA,	Negate (Accumulator A)
41		(No Code)
42		(No Code)
43	COMA,	Complement (Accumulator A)
44	LSRA,	Logical Shift Right (Accumulator A)
		(45 No Code)
46	RORA,	Rotate Right (Accumulator A)
47	ASRA,	Arithmetic Shift Right (Accumulator A)
48	ASLA,	Arithmetic Shift Left (Accumulator A)
49	ROLA,	Rotate Left (Accumulator A)
4A	DECA,	Decrement (Accumulator A)
4B		(No Code)
4C	INCA,	Increment (Accumulator A)
4D	TSTA,	Test (Accumulator A)
4E		(No Code)
4F	CLRA,	Clear (Accumulator A)
50	NEGB,	Negate (Accumulator B)
51		(No Code)
52		(No Code)
53	COMB,	Complement (Accumulator B)
54	LSRB,	Logical Shift Right (Accumulator B)
55		(No Code)
56	RORB,	Rotate Right (Accumulator B)
57	ASRB,	Arithmetic Shift Right (Accumulator B)
58	ASLB,	Arithmetic Shift Left (Accumulator B)
59	ROLB,	Rotate Left (Accumulator B)
5A	DECB,	Decrement (Accumulator B)
5B		(No Code)
5C	INCB,	Increment (Accumulator B)
5D	TSTB,	Test (Accumulator B)
5E		(No Code)

(Numeric, continued)

5F	CLRB,	Clear (Accumulator B)
60	NEG,	Negate (Indexed, X)
61		(No Code)
62		(No Code)
63	COM,	Complement (Indexed, X)
64	LSR,	Logical Shift Right (Indexed, X)
65		(No Code)
66	ROR,	Rotate Right (Indexed, X)
67	ASR,	Arithmetic Shift Right (Indexed, X)
68	ASL,	Arithmetic Shift Left (Indexed, X)
69	ROL,	Rotate Left (Indexed, X)
6A	DEC,	Decrement (Indexed, X)
6B		(No Code)
6C	INC,	Increment (Indexed, X)
6D	TST,	Test (Indexed, X)
6E	JMP,	Jump (Indexed, X)
6F	CLR,	Clear (Indexed, X)
70	NEG,	Negate (Extended Mode, X)
71		(No Code)
72		(No Code)
73	COM,	Complement (Extended Mode)
74	LSR,	Logical Shift Right (Extended Mode)
76	ROR,	Rotate Right (Extended Mode)
77	ASR,	Arithmetic Shift Right (Extended Mode)
78	ASL,	Arithmetic Shift Left (Extended Mode)
79	ROL,	Rotate Left (Extended Mode)
7A	DEC,	Decrement (Extended Mode)
7B		(No Code)
7C	INC,	Increment (Extended Mode)
7D	TST,	Test (Extended Mode)
7E	JMP,	Jump (Extended Mode)
7F	CLR,	Clear (Extended Mode)
80	SUBA,	Subtract from Accumulator A
81	CMPA,	Compare to Accumulator A
82	SBCA,	Subtract from Accum A with Carry
83	SUBD,	Subtract from Accumulator D
84	ANDA,	Logical AND to Accumulator A
85	BITA,	Bit Test Accumulator using Accum A
86	LDAA,	Load Accumulator A
87		(No Code)
88	EORA,	Exclusive OR to Accumulator A
89	ADCA,	Add to Accum A with Carry
8A	ORAA,	Logical Inclusive OR to Accumulator A
8B	ADDA,	Add to Accum A without Carry
8C	CPX,	Compare Index Register X
8D	BSR,	Branch to Subroutine
8E	LDS,	Load Stack Pointer
8F	XGDX,	Exchange Double Accumulator and Index Reg X
90	SUBA,	Subtract from Accumulator A (Direct)
91	CMPA,	Compare to Accumulator A (Direct)

92	SBCA,	Subtract from Accum A with Carry (Direct)
93	SUBD,	Subtract from Accumulator D (Direct)
94	ANDA,	Logical AND to Accumulator A (Direct)
95	BITA,	Bit Test Accumulator using Accum A (Direct)
96	LDAA,	Load Accumulator A (Direct)
97	STAA,	Store Accumulator A (Direct)
98	EORA,	Exclusive OR to Accumulator A (Direct)
99	ADCA,	Add to Accum A with Carry (Direct)
9A	ORAA,	Logical Inclusive OR to Accum A (Direct)
9B	ADDA,	Add to Accum A without Carry (Direct)
9C	CPX,	Compare Index Register X (Direct)
9D	JSR,	Jump to Subroutine (Direct)
9E	LDS,	Load Stack Pointer (Direct)
9F	STS,	Store Stack Pointer (Direct)
A0	SUBA,	Subtract from Accumulator A (Indexed, X)
A1	CMPA,	Compare to Accumulator A (Indexed, X)
A2	SBCA,	Subtract from Accum A w/Carry (Indexed, X)
A3	SUBD,	Subtract from Accumulator D (Indexed, X)
A4	ANDA,	Logical AND to Accumulator A (Indexed, X)
A5	BITA,	Bit Test Accumulator A (Indexed, X)
A6	LDAA,	Load Accumulator A (Indexed, X)
A7	STAA,	Store Accumulator A (Indexed, X)
A8	EORA,	Exclusive OR to Accumulator A (Indexed, X)
A9	ADCA,	Add to Accum A with Carry (Indexed, X)
AA	ORAA,	Logical Inclusive OR to Accum A (Indexed, X)
AB	ADDA,	Add to Accum A without Carry (Indexed, X)
AC	CPX,	Compare Index Register X (Indexed, X)
AD	JSR,	Jump to Subroutine (Indexed, X)
AE	LDS,	Load Stack Pointer (Indexed, X)
AF	STS,	Store Stack Pointer (Indexed, X)
B0	SUBA,	Subtract from Accumulator A (Extended)
B1	CMPA,	Compare to Accumulator A (Extended)
B2	SBCA,	Subtract from Accum A with Carry (Extended)
B3	SUBD,	Subtract from Accumulator D (Extended)
B4	ANDA,	Logical AND to Accumulator A (Extended)
B5	BITA,	Bit Test Accum using Accum A (Extended)
B6	LDAA,	Load Accumulator A (Extended)
B7	STAA,	Store Accumulator A (Extended)
B8	EORA,	Exclusive OR to Accumulator A (Extended)
B9	ADCA,	Add to Accum A with Carry (Extended)
BA	ORAA,	Logical Inclusive OR to Accum A (Extended)
BB	ADDA,	Add to Accum A without Carry (Extended)
BC	CPX,	Compare Index Register X (Extended)
BD	JSR,	Jump to Subroutine (Extended)
BE	LDS,	Load Stack Pointer (Extended)
BF	STS,	Store Stack Pointer (Extended)
C0	SUBB,	Subtract from Accumulator B
C1	CMPB,	Compare to Accumulator B
C2	SBCB,	Subtract from Accum B with Carry
C3	ADDD,	Add to Accum D without Carry
C4	ANDB,	Logical AND to Accumulator B

(Numeric, continued)

C5	BITB,	Bit Test Accumulator using Accum B
C6	LDAB,	Load Accumulator B
C7		(No Code)
C8	EORB,	Exclusive OR to Accumulator B
C9	ADCB,	Add to Accum B with Carry
CA	ORAB,	Logical Inclusive OR to Accum B
CB	ADDB,	Add to Accum B without Carry
CC	LDD,	Load Accumulator D
CD		(PAGE 4)
CE	LDX,	Load Index Register X
CF	STOP,	Stop Processing
D0	SUBB,	Subtract from Accumulator B (Direct)
D1	CMPB,	Compare to Accumulator B (Direct)
D2	SBCB,	Subtract from Accum B with Carry (Direct)
D3	ADDD,	Add to Accum D without Carry (Direct)
D4	ANDB,	Logical AND to Accumulator B (Direct)
D5	BITB,	Bit Test Accumulator using Accum B (Direct)
D6	LDAB,	Load Accumulator B (Direct)
D7	STAB,	Store Accumulator B (Direct)
D8	EORB,	Exclusive OR to Accumulator B (Direct)
D9	ADCB,	Add to Accum B with Carry (Direct)
DA	ORAB,	Logical Inclusive OR to Accum B (Direct)
DB	ADDB,	Add to Accum B without Carry (Direct)
DC	LDD,	Load Accumulator D (Direct)
DD	STD,	Store Accumulator D (Direct)
DE	LDX,	Load Index Register X (Direct)
DF	STX,	Store Index Register X (Direct)
E0	SUBB,	Subtract from Accumulator B (Indexed, X)
E1	CMPB,	Compare to Accumulator B (Indexed, X)
E2	SBCB,	Subtract from Accum B w/Carry (Indexed, X)
E3	ADDD,	Add to Accum D without Carry (Indexed, X)
E4	ANDB,	Logical AND to Accumulator B (Indexed, X)
E5	BITB,	Bit Test Accumulator B (Indexed, X)
E6	LDAB,	Load Accumulator B (Indexed, X)
E7	STAB,	Store Accumulator B (Indexed, X)
E8	EORB,	Exclusive OR to Accumulator B (Indexed, X)
E9	ADCB,	Add to Accum B with Carry (Indexed, X)
EA	ORAB,	Logical Inclusive OR to Accum B (Indexed, X)
EB	ADDB,	Add to Accum B without Carry (Indexed, X)
EC	LDD,	Load Accumulator D (Indexed, X)
ED	STD,	Store Accumulator D (Indexed, X)
EE	LDX,	Load Index Register X (Indexed, X)
EF	STX,	Store Index Register X (Indexed, X)
F0	SUBB,	Subtract from Accumulator B (Extended)
F1	CMPB,	Compare to Accumulator B (Extended)
F2	SBCB,	Subtract from Accum B with Carry (Extended)
F3	ADDD,	Add to Accum D without Carry (Extended)
F4	ANDB,	Logical AND to Accumulator B (Extended)
F5	BITB,	Bit Test Accum using Accum B (Extended)
F6	LDAB,	Load Accumulator B (Extended)
F7	STAB,	Store Accumulator B (Extended)
F8	EORB,	Exclusive OR to Accumulator B (Extended)

(Numeric, continued)

F9	ADCB,	Add to Accum B with Carry (Extended)
FA	ORAB,	Logical Inclusive OR to Accum B (Extended)
FB	ADDB,	Add to Accum B without Carry (Extended)
FC	LDD,	Load Accumulator D (Extended)
FD	STD,	Store Accumulator D (Extended)
FE	LDX,	Load Index Register X (Extended)
FF	STX,	Store Index Register X (Extended)

Ascending Order by Generated Op Code
Two Byte Codes

1808	INY,	Increment Index Register Y
1809	DEY,	Decrement Index Register Y
181C	BSET,	Set Bit(s) in Memory (Indexed, Y)
181D	BCLR,	Clear Bit(s) in Memory (Indexed, Y)
181E	BRSET,	Branch if Bit(s) Set (Indexed, Y)
181F	BRCLR,	Branch if Bit(s) Clear (Indexed, Y)
1830	TSY,	Transfer from Stack Pointer to Index Reg Y
1835	TYS,	Transfer from Index Reg Y to Stack Pointer
1838	PULY,	Pull Index Register Y from Stack
183A	ABY,	Add Accumulator B to Index Register Y
183C	PSHY,	Push Index Register Y onto Stack
1860	NEG,	Negate (Indexed, Y)
1863	COM,	Complement (Indexed, Y)
1864	LSR,	Logical Shift Right (Indexed, Y)
1866	ROR,	Rotate Right (Indexed, Y)
1867	ASR,	Arithmetic Shift Right (Indexed, Y)
1868	ASL,	Arithmetic Shift Left (Indexed, Y)
1869	ROL,	Rotate Left (Indexed, Y)
186A	DEC,	Decrement (Indexed, Y)
186C	INC,	Increment (Indexed, Y)
186D	TST,	Test (Indexed, Y)
186E	JMP,	Jump (Indexed, Y)
186F	CLR,	Clear (Indexed, Y)
188C	CPY,	Compare Index Register Y
188F	XGDY,	Exchange Accumulator D and Index Reg Y
189C	CPY,	Compare Index Register Y (Direct)
18A0	SUBA,	Subtract from Accumulator A (Indexed, Y)
18A1	CMPA,	Compare to Accumulator A (Indexed, Y)
18A2	SBCA,	Subtract from Accum A w/ Carry (Indexed, Y)
18A3	SUBD,	Subtract from Accumulator D (Indexed, Y)
18A4	ANDA,	Logical AND to Accumulator A (Indexed, Y)
18A5	BITA,	Bit Test Accumulator A (Indexed, Y)
18A6	LDAA,	Load Accumulator A (Indexed, Y)
18A7	STAA,	Store Accumulator A (Indexed, Y)
18A8	EORA,	Exclusive OR to Accumulator A (Indexed, Y)
18A9	ADCA,	Add to Accum A with Carry (Indexed, Y)
18AA	ORAA,	Logical Inclusive OR to Accum A (Indexed, Y)
18AB	ADDA,	Add to Accum A without Carry (Indexed, Y)
18AC	CPY,	Compare Index Register Y (Indexed, Y)

(Numeric, continued)

18AD	JSR,	Jump to Subroutine (Indexed, Y)
18AE	LDS,	Load Stack Pointer (Indexed, Y)
18AF	STS,	Store Stack Pointer (Indexed, Y)
18BC	CPY,	Compare Index Register Y (Extended)
18CE	LDY,	Load Index Register Y
18DE	LDY,	Load Index Register Y (Direct)
18DF	STY,	Store Index Register Y (Direct)
18E0	SUBB,	Subtract from Accumulator B (Indexed, Y)
18E1	CMPB,	Compare to Accumulator B (Indexed, Y)
18E2	SBCB,	Subtract from Accum B w/Carry (Indexed, Y)
18E3	ADDD,	Add to Accum D without Carry (Indexed, Y)
18E4	ANDB,	Logical AND to Accumulator B (Indexed, Y)
18E5	BITB,	Bit Test Accumulator B (Indexed, Y)
18E6	LDAB,	Load Accumulator B (Indexed, Y)
18E7	STAB,	Store Accumulator B (Indexed, Y)
18E8	EORB,	Exclusive OR to Accumulator B (Indexed, Y)
18E9	ADCB,	Add to Accum B with Carry (Indexed, Y)
18EA	ORAB,	Logical Inclusive OR to Accum B (Indexed, Y)
18EB	ADDB,	Add to Accum B without Carry (Indexed, Y)
18EC	LDD,	Load Accumulator D (Indexed, Y)
18ED	STD,	Store Accumulator D (Indexed, Y)
18EE	LDY,	Load Index Register Y (Indexed, Y)
18EF	STY,	Store Index Register Y (Indexed, Y)
18FE	LDY,	Load Index Register Y (Extended)
18FF	STY,	Store Index Register Y (Extended)
1A83	CPD,	Compare Accumulator D
1A93	CPD,	Compare Accumulator D (Direct)
1AA3	CPD,	Compare Accumulator D (Indexed, X)
1AAC	CPY,	Compare Index Register Y (Indexed, X)
1AB3	CPD,	Compare Accumulator D (Extended)
1AEE	LDY,	Load Index Register Y (Indexed, X)
1AEF	STY,	Store Index Register Y (Indexed, X)
CDA3	CPD,	Compare Accumulator D (Indexed, Y)
CDAC	CPX,	Compare Index Register X (Indexed, Y)
CDEE	LDX,	Load Index Register X (Indexed, Y)
CDEF	STX,	Store Index Register X (Indexed, Y)

Ascending Order by Mnemonic

ABA,	1B	Add Accumulator B to Accumulator A
ABX,	3A	Add Accumulator B to Index Register X
ABY,	183A	Add Accumulator B to Index Register Y
ADCA,	89	Add to Accum A with Carry
	99	Add to Accum A with Carry (Direct)
	A9	Add to Accum A with Carry (Indexed, X)
	B9	Add to Accum A with Carry (Extended)
	18A9	Add to Accum A with Carry (Indexed, Y)
ADCB,	C9	Add to Accum B with Carry
	D9	Add to Accum B with Carry (Direct)
	E9	Add to Accum B with Carry (Indexed, X)
	F9	Add to Accum B with Carry (Extended)
	18E9	Add to Accum B with Carry (Indexed, Y)
ADDA,	8B	Add to Accum A without Carry
	9B	Add to Accum A without Carry (Direct)
	AB	Add to Accum A without Carry (Indexed, X)
	BB	Add to Accum A without Carry (Extended)
	18AB	Add to Accum A without Carry (Indexed, Y)
ADDB,	CB	Add to Accum B without Carry
	DB	Add to Accum B without Carry (Direct)
	EB	Add to Accum B without Carry (Indexed, X)
	FB	Add to Accum B without Carry (Extended)
	18EB	Add to Accum B without Carry (Indexed, Y)
ADDD,	C3	Add to Accum D without Carry
	D3	Add to Accum D without Carry (Direct)
	E3	Add to Accum D without Carry (Indexed, X)
	F3	Add to Accum D without Carry (Extended)
	18E3	Add to Accum D without Carry (Indexed, Y)
ANDA,	84	Logical AND to Accumulator A
	94	Logical AND to Accumulator A (Direct)
	A4	Logical AND to Accumulator A (Indexed, X)
	B4	Logical AND to Accumulator A (Extended)
	18A4	Logical AND to Accumulator A (Indexed, Y)
ANDB,	C4	Logical AND to Accumulator B
	D4	Logical AND to Accumulator B (Direct)
	E4	Logical AND to Accumulator B (Indexed, X)
	F4	Logical AND to Accumulator B (Extended)
	18E4	Logical AND to Accumulator B (Indexed, Y)
ASL,	68	Arithmetic Shift Left (Indexed, X)
	78	Arithmetic Shift Left (Extended Mode)
	1868	Arithmetic Shift Left (Indexed, Y)
ASLA,	48	Arithmetic Shift Left (Accumulator A)
ASLB,	58	Arithmetic Shift Left (Accumulator B)
ASLD,	05	Arithmetic Shift Left Double Accumulator
ASR,	67	Arithmetic Shift Right (Indexed, X)
	77	Arithmetic Shift Right (Extended Mode)
	1867	Arithmetic Shift Right (Indexed, Y)
ASRA,	47	Arithmetic Shift Right (Accumulator A)
ASRB,	57	Arithmetic Shift Right (Accumulator B)
BCC,	24	Branch if Carry Clear
BCLR, 15	Clear Bit(s) in Memory	

(Alphabetic, continued)

BCLR,	1D	Clear Bit(s) in Memory (Indexed, X)
	181D	Clear Bit(s) in Memory (Indexed, Y)
BCS,	25	Branch if Carry Set
BEQ,	27	Branch if Equal
BGE,	2C	Branch if Greater than or Equal to Zero
BGT,	2E	Branch if Greater than Zero
BHI,	22	Branch if Higher
BHS,	24	Branch if Higher or Same
BITA,	85	Bit Test Accumulator using Accum A
	95	Bit Test Accumulator using Accum A (Direct)
	A5	Bit Test Accumulator A (Indexed, X)
	B5	Bit Test Accum using Accum A (Extended)
	18A5	Bit Test Accumulator A (Indexed, Y)
BITB,	C5	Bit Test Accumulator using Accum B
	D5	Bit Test Accumulator using Accum B (Direct)
	E5	Bit Test Accumulator B (Indexed, X)
	F5	Bit Test Accum using Accum B (Extended)
	18E5	Bit Test Accumulator B (Indexed, Y)
BLE,	2F	Branch if Less than or Equal to Zero
BLO,	25	Branch if Lower
BLS,	23	Branch if Lower or Same
BLT,	2D	Branch if Less than Zero
BMI,	2B	Branch if Minus
BNE,	26	Branch if Not Equal to Zero
BPL,	2A	Branch if Plus
BRA,	20	Branch Always
BRCLR,	13	Branch if Bit(s) Clear
	1F	Branch if Bit(s) Clear (Indexed, X)
	181F	Branch if Bit(s) Clear (Indexed, Y)
BRN,	21	Branch Never
BRSET,	12	Branch if Bit(s) Set
	1E	Branch if Bit(s) Set (Indexed, X)
	181E	Branch if Bit(s) Set (Indexed, Y)
BSET,	14	Set Bit(s) in Memory
	1C	Set Bit(s) in Memory (Indexed, X)
	181C	Set Bit(s) in Memory (Indexed, Y)
BSR,	8D	Branch to Subroutine
BVC,	28	Branch if Overflow Clear
BVS,	29	Branch if Overflow Set
CBA,	11	Compare Accumulators
CLC,	0C	Clear Carry
CLI,	0E	Clear Interrupt Mask
CLR,	6F	Clear (Indexed, X)
	7F	Clear (Extended Mode)
	186F	Clear (Indexed, Y)
CLRA,	4F	Clear (Accumulator A)
CLRB,	5F	Clear (Accumulator B)
CLV,	0A	Clear Twos-Complement Overflow Bit
CMPA,81	Compare to Accumulator A	
	91	Compare to Accumulator A (Direct)
	A1	Compare to Accumulator A (Indexed, X)
	B1	Compare to Accumulator A (Extended)

(Alphabetic, continued)

CMPA,18A1		Compare to Accumulator A (Indexed, Y)
CMPB,	C1	Compare to Accumulator B
	D1	Compare to Accumulator B (Direct)
	E1	Compare to Accumulator B (Indexed, X)
	F1	Compare to Accumulator B (Extended)
	18E1	Compare to Accumulator B (Indexed, Y)
COM,	63	Complement (Indexed, X)
	73	Complement (Extended Mode)
	1863	Complement (Indexed, Y)
COMA,	43	Complement (Accumulator A)
COMB,	53	Complement (Accumulator B)
CPD,	1A83	Compare Accumulator D
	1A93	Compare Accumulator D (Direct)
	1AA3	Compare Accumulator D (Indexed, X)
	1AB3	Compare Accumulator D (Extended)
	CDA3	Compare Accumulator D (Indexed, Y)
CPX,	8C	Compare Index Register X
	9C	Compare Index Register X (Direct)
	AC	Compare Index Register X (Indexed, X)
	BC	Compare Index Register X (Extended)
	CDAC	Compare Index Register X (Indexed, Y)
CPY,	188C	Compare Index Register Y
	189C	Compare Index Register Y (Direct)
	18AC	Compare Index Register Y (Indexed, Y)
	18BC	Compare Index Register Y (Extended)
	1AAC	Compare Index Register Y (Indexed, X)
DAA,	19	Decimal Adjust Accumulator A
DEC,	6A	Decrement (Indexed, X)
	7A	Decrement (Extended Mode)
	186A	Decrement (Indexed, Y)
DECA,4A		Decrement (Accumulator A)
DECB,	5A	Decrement (Accumulator B)
DES,	34	Decrement Stack Pointer
DEX,	09	Decrement Index Register X
DEY,	1809	Decrement Index Register Y
EORA,88		Exclusive OR to Accumulator A
	98	Exclusive OR to Accumulator A (Direct)
	A8	Exclusive OR to Accumulator A (Indexed, X)
	B8	Exclusive OR to Accumulator A (Extended)
	18A8	Exclusive OR to Accumulator A (Indexed, Y)
EORB,	C8	Exclusive OR to Accumulator B
	D8	Exclusive OR to Accumulator B (Direct)
	E8	Exclusive OR to Accumulator B (Indexed, X)
	F8	Exclusive OR to Accumulator B (Extended)
	18E8	Exclusive OR to Accumulator B (Indexed, Y)
FDIV,	03	Fractional Divide
IDIV,	02	Integer Divide
INC,	6C	Increment (Indexed, X)
	7C	Increment (Extended Mode)
	186C	Increment (Indexed, Y)
INCA,	4C	Increment (Accumulator A)
INCB,	5C	Increment (Accumulator B)

(Alphabetic, continued)

INS,	31	Increment Stack Pointer
INX,	08	Increment Index Register X
INY,	1808	Increment Index Register Y
JMP,	6E	Jump (Indexed, X)
	7E	Jump (Extended Mode)
	186E	Jump (Indexed, Y)
JSR,	9D	Jump to Subroutine (Direct)
	AD	Jump to Subroutine (Indexed, X)
	BD	Jump to Subroutine (Extended)
	18AD	Jump to Subroutine (Indexed, Y)
LDAA,	86	Load Accumulator A
	96	Load Accumulator A (Direct)
	A6	Load Accumulator A (Indexed, X)
	B6	Load Accumulator A (Extended)
	18A6	Load Accumulator A (Indexed, Y)
LDAB,C6	Load Accumulator B	
	D6	Load Accumulator B (Direct)
	E6	Load Accumulator B (Indexed, X)
	F6	Load Accumulator B (Extended)
	18E6	Load Accumulator B (Indexed, Y)
LDD,	CC	Load Accumulator D
	DC	Load Accumulator D (Direct)
	EC	Load Accumulator D (Indexed, X)
	FC	Load Accumulator D (Extended)
	18EC	Load Accumulator D (Indexed, Y)
LDS,	8E	Load Stack Pointer
	9E	Load Stack Pointer (Direct)
	AE	Load Stack Pointer (Indexed, X)
	BE	Load Stack Pointer (Extended)
	18AE	Load Stack Pointer (Indexed, Y)
LDX,	CE	Load Index Register X
	DE	Load Index Register X (Direct)
	EE	Load Index Register X (Indexed, X)
	FE	Load Index Register X (Extended)
	CDEE	Load Index Register X (Indexed, Y)
LDY,	18CE	Load Index Register Y
	18DE	Load Index Register Y (Direct)
	18EE	Load Index Register Y (Indexed, Y)
	18FE	Load Index Register Y (Extended)
	1AEE	Load Index Register Y (Indexed, X)
LSLD,	05	Logical Shift Left Double
LSR,	64	Logical Shift Right (Indexed, X)
	74	Logical Shift Right (Extended Mode)
	1864	Logical Shift Right (Indexed, Y)
LSRA,	44	Logical Shift Right (Accumulator A)
LSRB,	54	Logical Shift Right (Accumulator B)
LSRD,	04	Logical Shift Right Double Accumulator
MUL,	3D	Multiply Unsigned
NEG,	60	Negate (Indexed, X)
	70	Negate (Extended Mode, X)
	1860	Negate (Indexed, Y)
NEGA,	40	Negate (Accumulator A)

(Alphabetic, continued)

NEGB,50	Negate (Accumulator B)	
NOP,	01	No Operation
ORAA,	8A	Logical Inclusive OR to Accumulator A
	9A	Logical Inclusive OR to Accum A (Direct)
	AA	Logical Inclusive OR to Accum A (Indexed, X)
	BA	Logical Inclusive OR to Accum A (Extended)
	18AA	Logical Inclusive OR to Accum A (Indexed, Y)
ORAB,	CA	Logical Inclusive OR to Accum B
	DA	Logical Inclusive OR to Accum B (Direct)
	EA	Logical Inclusive OR to Accum B (Indexed, X)
	FA	Logical Inclusive OR to Accum B (Extended)
	18EA	Logical Inclusive OR to Accum B (Indexed, Y)
PSHA,	36	Push Data onto Stack (from Accumulator A)
PSHB,	37	Push Data onto Stack (from Accumulator B)
PSHX,	3C	Push Index Register X onto Stack
PSHY,	183C	Push Index Register Y onto Stack
PULA,	32	Pull Data from Stack (into Accumulator A)
PULB,	33	Pull Data from Stack (into Accumulator B)
PULX,	38	Pull Index Register X from Stack
PULY,	1838	Pull Index Register Y from Stack
ROL,	69	Rotate Left (Indexed, X)
	79	Rotate Left (Extended Mode)
	1869	Rotate Left (Indexed, Y)
ROLA,49	Rotate Left (Accumulator A)	
ROLB,	59	Rotate Left (Accumulator B)
ROR,	66	Rotate Right (Indexed, X)
	76	Rotate Right (Extended Mode)
	1866	Rotate Right (Indexed, Y)
RORA,	46	Rotate Right (Accumulator A)
RORB,	56	Rotate Right (Accumulator B)
RTI,	3B	Return from Interrupt
RTS,	39	Return from Subroutine
SBA,	10	Subtract Accumulators
SBCA,	82	Subtract from Accum A with Carry
	92	Subtract from Accum A with Carry (Direct)
	A2	Subtract from Accum A w/Carry (Indexed, X)
	B2	Subtract from Accum A with Carry (Extended)
	18A2	Subtract from Accum A w/ Carry (Indexed, Y)
SBCB,	C2	Subtract from Accum B with Carry
	D2	Subtract from Accum B with Carry (Direct)
	E2	Subtract from Accum B w/Carry (Indexed, X)
	F2	Subtract from Accum B with Carry (Extended)
	18E2	Subtract from Accum B w/Carry (Indexed, Y)
SEC,	0D	Set Carry
SEI,	0F	Set Interrupt Mask
SEV,	0B	Set Twos-Complement Overflow Bit
STAA,	97	Store Accumulator A (Direct)
	A7	Store Accumulator A (Indexed, X)
	B7	Store Accumulator A (Extended)
	18A7	Store Accumulator A (Indexed, Y)
STAB,	D7	Store Accumulator B (Direct)
	E7	Store Accumulator B (Indexed, X)

(Alphabetic, continued)

STAB,	F7	Store Accumulator B (Extended)
	18E7	Store Accumulator B (Indexed, Y)
STD,	DD	Store Accumulator D (Direct)
	ED	Store Accumulator D (Indexed, X)
	FD	Store Accumulator D (Extended)
	18ED	Store Accumulator D (Indexed, Y)
STOP,	CF	Stop Processing
STS,	9F	Store Stack Pointer (Direct)
	AF	Store Stack Pointer (Indexed, X)
	BF	Store Stack Pointer (Extended)
	18AF	Store Stack Pointer (Indexed, Y)
STX,	DF	Store Index Register X (Direct)
	EF	Store Index Register X (Indexed, X)
	FF	Store Index Register X (Extended)
	CDEF	Store Index Register X (Indexed, Y)
STY,	18DF	Store Index Register Y (Direct)
	18EF	Store Index Register Y (Indexed, Y)
	18FF	Store Index Register Y (Extended)
	1AEF	Store Index Register Y (Indexed, X)
SUBA,	80	Subtract from Accumulator A
	90	Subtract from Accumulator A (Direct)
	A0	Subtract from Accumulator A (Indexed, X)
	B0	Subtract from Accumulator A (Extended)
	18A0	Subtract from Accumulator A (Indexed, Y)
SUBB,	C0	Subtract from Accumulator B
	D0	Subtract from Accumulator B (Direct)
	E0	Subtract from Accumulator B (Indexed, X)
	F0	Subtract from Accumulator B (Extended)
	18E0	Subtract from Accumulator B (Indexed, Y)
SUBD,	83	Subtract from Accumulator D
	93	Subtract from Accumulator D (Direct)
	A3	Subtract from Accumulator D (Indexed, X)
	B3	Subtract from Accumulator D (Extended)
	18A3	Subtract from Accumulator D (Indexed, Y)
SWI,	3F	Software Interrupt
TAB,	16	Transfer from Accumulator A to Accumulator B
TAP,	06	Transfer from Accum A to Condition Code Reg.
TBA,	17	Transfer from Accumulator B to Accumulator A
TEST,	00	Test Operation
TPA,	07	Transfer from Condition Code Reg. to Accum A
TST,	6D	Test (Indexed, X)
	7D	Test (Extended Mode)
	186D	Test (Indexed, Y)
TSTA,	4D	Test (Accumulator A)
TSTB,	5D	Test (Accumulator B)
TSX,	30	Transfer from Stack Pointer to Index Reg X
TSY,	1830	Transfer from Stack Pointer to Index Reg Y
TXS,	35	Transfer from Index Reg X to Stack Pointer
TYS,	1835	Transfer from Index Reg Y to Stack Pointer
WAI,	3E	Wait for Interrupt
XGDX,	8F	Exchange Double Accumulator and Index Reg X
XGDY,188F	Exchange Accumulator D and Index Reg Y	

Ascending Order by System Resource

ACCUMULATOR A

ABA,	1B	Add Accumulator B to Accumulator A
ADCA,	89	Add to Accum A with Carry
ADDA,	8B	Add to Accum A without Carry
ANDA,	84	Logical AND to Accumulator A
ASLA,	48	Arithmetic Shift Left (Accumulator A)
ASRA,	47	Arithmetic Shift Right (Accumulator A)
BITA,	85	Bit Test Accumulator using Accum A
CBA,	11	Compare Accumulators
CLRA,	4F	Clear (Accumulator A)
CMPA,81	Compare to Accumulator A	
COMA,	43	Complement (Accumulator A)
DAA,	19	Decimal Adjust Accumulator A
DECA,4A	Decrement (Accumulator A)	
EORA,88	Exclusive OR to Accumulator A	
INCA,	4C	Increment (Accumulator A)
LDAA,	86	Load Accumulator A
LSRA,	44	Logical Shift Right (Accumulator A)
MUL,	3D	Multiply Unsigned
NEGA,	40	Negate (Accumulator A)
ORAA,	8A	Logical Inclusive OR to Accumulator A
PSHA,	36	Push Data onto Stack (from Accumulator A)
PULA,	32	Pull Data from Stack (into Accumulator A)
ROLA,49	Rotate Left (Accumulator A)	
RORA,	46	Rotate Right (Accumulator A)
SBA,	10	Subtract Accumulators
SBCA,	82	Subtract from Accum A with Carry
STAA,	97	Store Accumulator A (Direct)
SUBA,	80	Subtract from Accumulator A
TAB,	16	Transfer from Accumulator A to Accumulator B
TAP,	06	Transfer from Accum A to Condition Code Reg.
TBA,	17	Transfer from Accumulator B to Accumulator A
TPA,	07	Transfer from Condition Code Reg. to Accum A
TSTA,	4D	Test (Accumulator A)

ACCUMULATOR B

ABA,	1B	Add Accumulator B to Accumulator A
ABX,	3A	Add Accumulator B to Index Register X
ABY,	183A	Add Accumulator B to Index Register Y
ADCB,	C9	Add to Accum B with Carry
ADDB,	CB	Add to Accum B without Carry
ANDB,	C4	Logical AND to Accumulator B
ASLB,	58	Arithmetic Shift Left (Accumulator B)
ASRB,	57	Arithmetic Shift Right (Accumulator B)
BITB,	C5	Bit Test Accumulator using Accum B
CBA,	11	Compare Accumulators
CLRB,	5F	Clear (Accumulator B)
CMPB,	C1	Compare to Accumulator B
COMB,	53	Complement (Accumulator B)

(Resource-ACCB, continued)

DECB,	5A	Decrement (Accumulator B)
EORB,	C8	Exclusive OR to Accumulator B
INCB,	5C	Increment (Accumulator B)
LDAB,C6		Load Accumulator B
LSRB,	54	Logical Shift Right (Accumulator B)
MUL,	3D	Multiply Unsigned
NEGB,50		Negate (Accumulator B)
ORAB,	CA	Logical Inclusive OR to Accum B
PSHB,	37	Push Data onto Stack (from Accumulator B)
PULB,	33	Pull Data from Stack (into Accumulator B)
ROLB,	59	Rotate Left (Accumulator B)
RORB,	56	Rotate Right (Accumulator B)
SBA,	10	Subtract Accumulators
SBCB,	C2	Subtract from Accum B with Carry
STAB,	D7	Store Accumulator B (Direct)
SUBB,	C0	Subtract from Accumulator B
TAB,	16	Transfer from Accumulator A to Accumulator B
TBA,	17	Transfer from Accumulator B to Accumulator A
TSTB,	5D	Test (Accumulator B)

DOUBLE ACCUMULATOR (D)

ADDD,	C3	Add to Accum D without Carry
ASLD,	05	Arithmetic Shift Left Double Accumulator
CPD,	1A83	Compare Accumulator D
FDIV,	03	Fractional Divide
IDIV,	02	Integer Divide
LDD,	CC	Load Accumulator D
LSLD,	05	Logical Shift Left Double
LSRD,	04	Logical Shift Right Double Accumulator
MUL,	3D	Multiply Unsigned
STD,	DD	Store Accumulator D (Direct)
SUBD,	83	Subtract from Accumulator D
XGDX,	8F	Exchange Double Accumulator and Index Reg X
XGDY,188F		Exchange Accumulator D and Index Reg Y

INDEX REGISTER X

ABX,	3A	Add Accumulator B to Index Register X
CPX,	8C	Compare Index Register X
DEX,	09	Decrement Index Register X
FDIV,	03	Fractional Divide
INX,	08	Increment Index Register X
LDX,	CE	Load Index Register X
PSHX,	3C	Push Index Register X onto Stack
PULX,	38	Pull Index Register X from Stack
STX,	DF	Store Index Register X (Direct)
TSX,	30	Transfer from Stack Pointer to Index Reg X
TXS,	35	Transfer from Index Reg X to Stack Pointer
XGDX,	8F	Exchange Double Accumulator and Index Reg X

INDEX REGISTER Y

ABY,	183A	Add Accumulator B to Index Register Y
CPY,	188C	Compare Index Register Y
DEY,	1809	Decrement Index Register Y
INY,	1808	Increment Index Register Y
LDY,	18CE	Load Index Register Y
PSHY,	183C	Push Index Register Y onto Stack
PULY,	1838	Pull Index Register Y from Stack
STY,	18DF	Store Index Register Y (Direct)
TSY,	1830	Transfer from Stack Pointer to Index Reg Y
TYS,	1835	Transfer from Index Reg Y to Stack Pointer
XGDY,188F	Exchange Accumulator D and Index Reg Y	

CONDITION CODE REGISTER

CLC,	0C	Clear Carry
CLI,	0E	Clear Interrupt Mask
CLV,	0A	Clear Twos-Complement Overflow Bit
SEC,	0D	Set Carry
SEI,	0F	Set Interrupt Mask
SEV,	0B	Set Twos-Complement Overflow Bit
TAP,	06	Transfer from Accum A to Condition Code Reg.
TPA,	07	Transfer from Condition Code Reg. to Accum A

STACK POINTER

BSR,	8D	Branch to Subroutine
DES,	34	Decrement Stack Pointer
INS,	31	Increment Stack Pointer
JSR,	9D	Jump to Subroutine (Direct)
LDS,	8E	Load Stack Pointer
PSHA,	36	Push Data onto Stack (from Accumulator A)
PSHB,	37	Push Data onto Stack (from Accumulator B)
PSHX,	3C	Push Index Register X onto Stack
PSHY,	183C	Push Index Register Y onto Stack
PULA,	32	Pull Data from Stack (into Accumulator A)
PULB,	33	Pull Data from Stack (into Accumulator B)
PULX,	38	Pull Index Register X from Stack
PULY,	1838	Pull Index Register Y from Stack
RTI,	3B	Return from Interrupt
RTS,	39	Return from Subroutine
STS,	9F	Store Stack Pointer (Direct)
SWI,	3F	Software Interrupt
TSX,	30	Transfer from Stack Pointer to Index Reg X
TSY,	1830	Transfer from Stack Pointer to Index Reg Y
TXS,	35	Transfer from Index Reg X to Stack Pointer
TYS,	1835	Transfer from Index Reg Y to Stack Pointer
WAI,	3E	Wait for Interrupt

MACHINE STORAGE

ADCA,	89	Add to Accum A with Carry
ADCB,	C9	Add to Accum B with Carry
ADDA,	8B	Add to Accum A without Carry
ADDB,	CB	Add to Accum B without Carry
ADDD,	C3	Add to Accum D without Carry
ANDA,	84	Logical AND to Accumulator A
ANDB,	C4	Logical AND to Accumulator B
ASLB,	58	Arithmetic Shift Left (Accumulator B)
ASL,	68	Arithmetic Shift Left (Indexed, X)
ASR,	67	Arithmetic Shift Right (Indexed, X)
BITA,	85	Bit Test Accumulator using Accum A
BITB,	C5	Bit Test Accumulator using Accum B
CLR,	6F	Clear (Indexed, X)
CMPA,81		Compare to Accumulator A
CMPB,	C1	Compare to Accumulator B
COM,	63	Complement (Indexed, X)
CPD,	1A83	Compare Accumulator D
CPX,	8C	Compare Index Register X
CPY,	188C	Compare Index Register Y
EORA,88		Exclusive OR to Accumulator A
EORB,	C8	Exclusive OR to Accumulator B
DEC,	6A	Decrement (Indexed, X)
INC,	6C	Increment (Indexed, X)
LDAA,	86	Load Accumulator A
LDAB,C6		Load Accumulator B
LDD,	CC	Load Accumulator D
LDS,	8E	Load Stack Pointer
LDX,	CE	Load Index Register X
LSR,	64	Logical Shift Right (Indexed, X)
NEG,	60	Negate (Indexed, X)
NOP,	01	No Operation
ORAA,	8A	Logical Inclusive OR to Accumulator A
ORAB,	CA	Logical Inclusive OR to Accum B
ROL,	69	Rotate Left (Indexed, X)
ROR,	66	Rotate Right (Indexed, X)
SBCA,	82	Subtract from Accum A with Carry
SBCB,	C2	Subtract from Accum B with Carry
STAA,	97	Store Accumulator A (Direct)
STAB,	D7	Store Accumulator B (Direct)
STD,	DD	Store Accumulator D (Direct)
STS,	9F	Store Stack Pointer (Direct)
STX,	DF	Store Index Register X (Direct)
STY,	18DF	Store Index Register Y (Direct)
SUBA,	80	Subtract from Accumulator A
SUBB,	C0	Subtract from Accumulator B
SUBD,	83	Subtract from Accumulator D
TST,	6D	Test (Indexed, X)

EXECUTION SEQUENCE

BCC,	24	Branch if Carry Clear
BCLR,	15	Clear Bit(s) in Memory
BCS,	25	Branch if Carry Set
BEQ,	27	Branch if Equal
BGE,	2C	Branch if Greater than or Equal to Zero
BGT,	2E	Branch if Greater than Zero
BHI,	22	Branch if Higher
BHS,	24	Branch if Higher or Same
BLE,	2F	Branch if Less than or Equal to Zero
BLO,	25	Branch if Lower
BLS,	23	Branch if Lower or Same
BLT,	2D	Branch if Less than Zero
BMI,	2B	Branch if Minus
BNE,	26	Branch if Not Equal to Zero
BPL,	2A	Branch if Plus
BRA,	20	Branch Always
BRCLR,	13	Branch if Bit(s) Clear
BRN,	21	Branch Never
BRSET,	12	Branch if Bit(s) Set
BSET,	14	Set Bit(s) in Memory
BSR,	8D	Branch to Subroutine
BVC,	28	Branch if Overflow Clear
BVS,	29	Branch if Overflow Set
JMP,	6E	Jump (Indexed, X)
JSR,	9D	Jump to Subroutine (Direct)
NOP,	01	No Operation
RTS,	39	Return from Subroutine
STOP,	CF	Stop Processing
TEST,	00	Test Operation

INTERRUPT HANDLING

RTI,	3B	Return from Interrupt
SWI,	3F	Software Interrupt
WAI,	3E	Wait for Interrupt

APPENDIX D

This Appendix contains the programming narrative for the ASM6811 compiler, explaining the internal logic, processing patterns, and design trade-offs.

```
( ASM6811 VERSION 1.11 05/14/93 )
```

The assembler will almost always be loaded first, prior to any portion of the user's application, to create the environment for development. This is not a hard and fast rule, but it is doubtful the programmer will want to inject a 4K chunk of somebody else's code into his own source code before writing any assembly language. Besides which, the Assembler creates finished machine code definitions; it is only needed during program creation. The memory space need not be included in the final user application. It can only be "thrown away" if it is kept separated from the user's mainline code. The first few lines of ASM6811 describe an environment for loading ASM6811 on a clean system. These lines have been "commented out" in Version 1.8 and two additional environment files have been provided to the user. These files, LOAD0100.PRE and LOADC000.PRE, can be downloaded prior to the ASM6811 file to create either a low memory or high memory installation respectively.

```
( ASM6811 )

( COLD


( FORGET TASK
( HEX

( C400 DP !
( C100 1C !
( 50 1E !


( VERSION 1.10 02/02/93 )
```

An oversight in the original definition of CODE-SUB, a new kind of machine code defining words, was that it did not invoke the assembler vocabulary.

```
: CODE-SUB [COMPILE] CODE-SUB [COMPILE] ASSEMBLER ;
```

CODE-SUB is therefore redefined to do all its original functions, with the addition of invocation of the assembler vocabulary. The word [COMPILE] is necessary in this definition because both the words CODE-SUB and ASSEMBLER are immediate and would have run rather than compile otherwise. This also explains why,

```
IMMEDIATE
```

follows the ending of the new CODE-SUB definition.

Repairs made, it is now time to invoke the assembler vocabulary and identify that any new definitions will be assigned to that vocabulary.

```
ASSEMBLER DEFINITIONS
```

Next a new variable is required to hold the addressing mode either by default or selection in the input line. The variable MODE is therefore created.

```
VARIABLE MODE
```

Five definitions that allow the programmer to explicitly select the addressing mode to be used by a mnemonic are defined: # DIR ,X ,Y and EXT. Each definition puts a coded number into MODE to be used later by the particular defining word.

```
: #   00 MODE ! ;
: DIR 10 MODE ! ;
: ,X  20 MODE ! ;
: ,Y 120 MODE ! ;
: EXT 30 MODE ! ;
```

Similarly, five words follow that check the state of the MODE variable and return a Boolean, which indicates whether the mode variable is set to its namesake setting.

```
: ?# MODE @ 0= ;
: ?DIR MODE @ 10 = ;
: ?,X MODE @  20 = ;
: ?,Y MODE @ 120 = ;
: ?EXT MODE @ 30 = ;
```

The constant assigned to MODE in each addressing mode was not an arbitrary choice. The least significant byte of that number is used by the later definitions to construct the actual op-code that is inserted into the in-line machine code. As might be expected then, there is a definition which strips out that LSB.

```
: MODE-LSB MODE @ FF AND ;
```

The ERROR function exists in the kernel but is orphaned, meaning it has no name available to call it from the outer interpreter. (Again, a tradeoff of the "heads" of some words used by FORTH but not mentioned in the '83 Standard, thus saving room for more words that were actually defined in the standard.) In order to not have to recreate ERROR and all its subcomponent parts, the definition that follows creates an external method of calling the headerless internal ERROR words. It uses a known offset from a named word to find the entry point to the ERROR routine. Before control is returned to the FORTH outer interpreter, the mode variable is reinitialized for business by EXT.

```
: ERROR EXT ' ID. CFA 4A + EXECUTE ;
```

The next definition is a segment that will be used in later relative branching instructions to check if the range of a branch is within a one byte offset. If the upper byte of the branch offset is zero, it is taken to be the actual byte offset to be compiled. (E.G., 3 BRA, causes a branch ahead three bytes, FD BRA, is a branch back three bytes.) If the upper byte is nonzero, it is taken to be the absolute address to be branched to. The offset value for that address is computed and its range again checked. If it is still not a one byte range, an Error Message 3 is issued. If there was not a size problem, the offset is instead compiled without error.

```
: RANGE-C, DUP FF00 AND IF HERE 1+ SWAP - DUP ABS FF00 AND
                                    IF 3 ERROR THEN THEN C, ;
```

The majority of the assembler's named op-codes are implemented using defining words based on <BUILDS DOES> constructs. This scheme greatly reduces the total memory size of the resulting code. The first of these defining words, also the simplest, is CPU, which creates single byte op-codes. This word takes the number from the stack when the new op-code is defined and compiles it into the dictionary with a C, (per the <BUILDS portion). Upon use of the op-code word in application the value compiled is fetched from the dictionary and compiled in-line in the machine code program under construction as a single byte instruction (per the DOES> portion).

Also note, as will be the case in all words of this type, after compiling functions are finished, the extended addressing mode is selected. This is to ensure at the beginning of each new program line the default mode of extended addressing is selected. The programmer therefore does not have to specify addressing modes explicitly on every line of the program. It is only necessary when a special mode is desired, such as when using immediate, the # words is necessary.

```
: CPU <BUILDS C, DOES> C@ C, EXT ; ( SINGLE BYTE OP-CODE )
```

Similarly, PG-2-CPU performs an identical function with second page, two-byte instructions. These are in the form of a fixed "18", indicating an instruction page switch to processor, followed by the single byte instructions (as in the CPU example before).

```
: PG-2-CPU <BUILDS C, DOES> 18 C, C@ C, EXT ;
( 18 AND SINGLE BYTE OP-CODE )
```

Prior to the definition of some higher complexity defining words, a few utilities are added. HHLL/LL compiles either the high/low byte combination if the mode is extended, or the low byte only if it is not. 2HHLL/LL is similar, however, it checks for the immediate mode and will compile the longer high/low combination in that case. This word takes care of occurrences of such functions as 1234 # LDD as distinguished from 00 ,X LDD. In the former, only two bytes are compiled with the LDD op-code; in the later, only one. MODE-ADJ will automatically change extended addressing to direct addressing, if possible, to chose the smaller and faster of two possible instructions to accomplish the same thing. 18,Y checks to see if the addressing action is a second page derivative of an "X" instruction that needs only a precursor 18 to change it into a "Y" instruction and compiles the 18 appropriately. SOK? checks to see if the attempt to use direct addressing is being used in a store instruction, which is not allowed. An Error Message 3 will be issued if the illegal action is detected.

```
: HHLL/LL, ?EXT IF , ELSE C, THEN EXT ;
: 2HHLL/LL, ?EXT ?# OR IF , ELSE C, THEN EXT ;
: MODE-ADJ, ( a n --- a )
  ?EXT IF OVER FF00 AND 0= IF DIR THEN THEN MODE-LSB OR C, ;
```

```
: 18,Y ?,Y IF 18 C, THEN ;
: SOK? ?# IF 3 ERROR THEN ;
```

Utilities established, the remaining defining words follow. The names selected for these defining words indicate their purpose. Examination of the Op-code Map for the 68HC11 will help in understanding this relationship. Over half of the Op-code Map Page 1 is arranged with either ACCA or ACCB headings over four columns representing the addressing modes: Immediate, Direct, Indexed and Extended. Each of the following defining words refers to the number of bytes (following the op-code) required for a particular column under those four rows. Consequently, the 1112-CPU defining word installs an op-code, modified by the MODE variable, followed by one byte if Immediate Addressing, one byte if Direct Addressing, one byte if Indexed, or two bytes if Extended (hence the 1112 portion of the name). The x112-CPU function is very similar; however, it does not allow immediate addressing. It is used for the STAA and STAB instructions that exclude that possibility. The xx12 definition is used on the words that are only valid in the Indexed mode for one byte and Extended mode for two. The other two modes are disallowed (accounting for the xx12 name). The 2112-CPU word is also similar; however, it compiles two bytes for the immediate mode, such as is required by the LDD instructions, etc.

```
: xx12-CPU <BUILDS C, DOES> 18,Y ?# ?DIR OR IF 3 ERROR THEN
  C@ MODE-LSB OR C, HHLL/LL, ;
: 1112-CPU <BUILDS C, DOES> 18,Y C@ MODE-ADJ, HHLL/LL, ;
: x112-CPU <BUILDS C, DOES> 18,Y SOK? C@ MODE-ADJ, HHLL/LL,;
: 2112-CPU <BUILDS C, DOES> 18,Y C@ MODE-ADJ, 2HHLL/LL, ;
```

A partial definition (OP-DD-MM) follows. It is a factored segment of the last two defining words shown below, separated only for the purpose of code-size savings. The OP-RR defining word is used to create the branching instructions that occur in the form of op-code followed by relative branching offset. The OP-DD-MM defining word handles the relatively unique BSET and BCLR instructions. These are in the format of op-code followed by addressing byte, either direct or indexed offset, then the bit mask to be used for the operation. OP-DD-MM-RR is a similar construct used for the BRSET and BRCLR instructions. They have all the features of the OP-DD-MM instructions plus a relative branch offset byte at the end.

```
: (OP-DD-MM) 18,Y SOK? MODE-LSB 20 = IF C OR THEN C, C, C, ;
: OP-RR <BUILDS C, DOES> C@ C, RANGE-C, EXT ;
: OP-DD-MM <BUILDS C, DOES> C@ (OP-DD-MM) EXT ;
: OP-DD-MM-RR <BUILDS C, DOES> C@ (OP-DD-MM) RANGE-C, EXT ;
```

With these preliminaries complete, the individual op-code words can be defined.

A few words were so complex that making special defining words to handle their cases would not have resulted in any reduction of code space. These special case op-codes were defined explicitly, without the use of defining words. Notice each refers to Op-code Map Pages 3 or 4 (i.e., may use 1A or CD prebytes).

```
: CPX, ?,Y IF CD C, THEN 8C MODE-ADJ, 2HHLL/LL, ;
: LDX, ?,Y IF CD C, THEN CE MODE-ADJ, 2HHLL/LL, ;
: STX, ?,Y IF CD C, THEN SOK? CF MODE-ADJ,  HHLL/LL, ;
: CPY, ?,X IF 1A ELSE 18 THEN C, CC MODE-ADJ, 2HHLL/LL, ;
: LDY, ?,X IF 1A ELSE 18 THEN C, CE MODE-ADJ, 2HHLL/LL, ;
: STY, ?,X IF 1A ELSE 18 THEN C, SOK? CF MODE-ADJ, HHLL/LL,;
: CPD, ?,Y IF CD C, ELSE 1A C, THEN 83 MODE-ADJ, 2HHLL/LL, ;
```

The following definitions are easily handled by the carefully laid out defining words. They follow closely the Op-code Map Page One in order of definition, starting with 00 and progressing toward FF. Each possible binary number is accounted for in sequence. Notations are made where there are "holes" in the Op-code Map, meaning the unused bytes could be left blank or used in future revisions as op-codes with new meanings or functions, or as page switches to indicate the use of an alternate Op-code Map Page yet undefined. Notice the 05 op-code has two names for the same function. At the very end a few Op-code Map Page Two instructions that have no equivalent Page One structures are filled in, and a few stragglers that don't match the main pattern of Op-code Map Page One are included.

```
00 CPU TEST,
01 CPU NOP,
02 CPU IDIV,
03 CPU FDIV,
04 CPU LSRD,
05 CPU ASLD,
05 CPU LSLD,
06 CPU TAP,
07 CPU TPA,
08 CPU INX,
09 CPU DEX,
0A CPU CLV,
0B CPU SEV,
0C CPU CLC,
0D CPU SEC,
0E CPU CLI,
0F CPU SEI,
10 CPU SBA,
11 CPU CBA,
12 OP-DD-MM-RR BRSET,
13 OP-DD-MM-RR BRCLR,
14 OP-DD-MM BSET,
15 OP-DD-MM BCLR,
16 CPU TAB,
17 CPU TBA,
( 18 PAGE 2 )
19 CPU DAA,
( 1A PAGE 3 )
1B CPU ABA,
( 1C BSET,
( 1D BCLR,
( 1E BRSET,
( 1F BRCLR,
20 OP-RR BRA,
21 OP-RR BRN,
22 OP-RR BHI,
23 OP-RR BLS,
24 OP-RR BCC,
24 OP-RR BHS,
25 OP-RR BCS,
25 OP-RR BLO,
26 OP-RR BNE,
27 OP-RR BEQ,
28 OP-RR BVC,
29 OP-RR BVS,
2A OP-RR BPL,
```

```
2B OP-RR BMI,
2C OP-RR BGE,
2D OP-RR BLT,
2E OP-RR BGT,
2F OP-RR BLE,
30 CPU TSX,
31 CPU INS,
32 CPU PULA,
33 CPU PULB,
34 CPU DES,
35 CPU TXS,
36 CPU PSHA,
37 CPU PSHB,
38 CPU PULX,
39 CPU RTS,
3A CPU ABX,
3B CPU RTI,
3C CPU PSHX,
3D CPU MUL,
3E CPU WAI,
3F CPU SWI,
40 CPU NEGA,
( 41 NC )
( 42 NC )
43 CPU COMA,
44 CPU LSRA,
( 45 NC )
46 CPU RORA,
47 CPU ASRA,
48 CPU ASLA,
49 CPU ROLA,
4A CPU DECA,
( 4B NC )
4C CPU INCA,
4D CPU TSTA,
( 4E NC )
4F CPU CLRA,
50 CPU NEGB,
( 51 NC )
( 52 NC )
53 CPU COMB,
54 CPU LSRB,
( 55 NC )
56 CPU RORB,
57 CPU ASRB,
58 CPU ASLB,
59 CPU ROLB,
5A CPU DECB,
( 5B NC )
5C CPU INCB,
5D CPU TSTB,
( 5E NC )
5F CPU CLRB,
( 60-7F )
40 xx12-CPU NEG,
43 xx12-CPU COM,
44 xx12-CPU LSR,
46 xx12-CPU ROR,
47 xx12-CPU ASR,
```

```
48 xx12-CPU ASL,
49 xx12-CPU ROL,
4A xx12-CPU DEC,
4C xx12-CPU INC,
4D xx12-CPU TST,
4E xx12-CPU JMP,
4F xx12-CPU CLR,
( 80-BF )
80 1112-CPU SUBA,
81 1112-CPU CMPA,
82 1112-CPU SBCA,
83 2112-CPU SUBD,
84 1112-CPU ANDA,
85 1112-CPU BITA,
86 1112-CPU LDAA,
87 x112-CPU STAA,
88 1112-CPU EORA,
89 1112-CPU ADCA,
8A 1112-CPU ORAA,
8B 1112-CPU ADDA,
( 8C CPX, )
8D x112-CPU JSR,
8E 2112-CPU LDS,
8F x112-CPU STS,
8F CPU XGDX,
( C0-FF )
C0 1112-CPU SUBB,
C1 1112-CPU CMPB,
C2 1112-CPU SBCB,
C3 2112-CPU ADDD,
C4 1112-CPU ANDB,
C5 1112-CPU BITB,
C6 1112-CPU LDAB,
C7 x112-CPU STAB,
C8 1112-CPU EORB,
C9 1112-CPU ADCB,
CA 1112-CPU ORAB,
CB 1112-CPU ADDB,
CC 2112-CPU LDD,
CD x112-CPU STD,
( CE LDX, )
( CF STX, )
CF CPU STOP,

08 PG-2-CPU INY,
09 PG-2-CPU DEY,
30 PG-2-CPU TSY,
35 PG-2-CPU TYS,
38 PG-2-CPU PULY,

3A PG-2-CPU ABY,
3C PG-2-CPU PSHY,

8F PG-2-CPU XGDY,

8D OP-RR BSR,
```

The bulk of a useful assembler is now in place. A few niceties follow. The macro definitions TOP and SEC are installed to make reference to the FORTH data stack more convenient.

```
: TOP ,Y    0  ;     (  ADDRESS THE BOTTOM OF THE STACK  *)
: SEC ,Y    2  ;        ( ADDRESS SECOND ITEM ON STACK  *)
```

Two security utilities are constructed to check the correctness of using the soon to follow structured programming branching words.

```
: ?EXEC STATE @ IF 12 ERROR THEN ;
: ?PAIRS - IF 13 ERROR THEN ;
```

Basic FORTH style branching words are added to allow structured programming techniques to be extended to assembly language routines. Each word works with security, leaving on the data stack a number code that will be compared by ?PAIRS to ensure matching structured types are used together and in the correct sequence.

Notice that each structured word's name follows the convention of the other op-codes and end in a ",". This helps distinguish them from their high level FORTH counterparts.

```
: BEGIN,       HERE  1  ;
: UNTIL,       ?EXEC >R 1 ?PAIRS R> C, HERE  1+ - C, ;
: AGAIN,       20 UNTIL, ;
: IF,          C,  HERE  0  C,  2  ;
: THEN,        ?EXEC  2  ?PAIRS  HERE OVER 1+ - SWAP C! ;
: ELSE,        2 ?PAIRS HERE 1+  0 BRA, SWAP HERE OVER 1+ - SWAP  C!   2  ;
```

To use these structures, it is necessary to precede each decision point with an indication of the type of assembly language test to be selected. In ASM6811 this is accomplished using one of sixteen Boolean relationship selection words in the form of .XX., or one of the words in conjunction with the negating word .NOT. .

```
: .NOT. 1 XOR ;                ( REVERSE ASSEMBLY TEST )
20 CONSTANT .FL.
21 CONSTANT .TR.
22 CONSTANT .LS.
23 CONSTANT .HI.
24 CONSTANT .CS.
24 CONSTANT .LO.
25 CONSTANT .CC.
25 CONSTANT .HS.
26 CONSTANT .EQ.
27 CONSTANT .NE.
28 CONSTANT .VS.
29 CONSTANT .VC.
2A CONSTANT .--.
2B CONSTANT .++.
2C CONSTANT .LT.
2D CONSTANT .GE.
```

```
2E CONSTANT .LE.
2F CONSTANT .GT.
```

Each leaves the op-code of a particular branch instruction that will be used for the constructed decision point that follows. It should be noted that the interpreted op-code of the branch instruction compiled is the opposite instance of the condition named in the source. For example, the statement

```
.GT. IF, <stmt-1> ELSE, <stmt-2> THEN,
```

is interpreted as

```
BLE <stmt-1> <stmt-2>
```

where <stmt-2> is selected when the condition is satisfied, otherwise <stmt-1> is selected. Therefore, the interpreted op-code of a branch instruction is that of the opposite conditional branch instruction.

A few words are added at this point. They are the suggestions of Ken Butterfield of Los Alamos National Labs. They allow use of the bit set and bit clear testing words which are combined with IF,. The defining word BIT-BR compiles the op-codes for the .CLR.IF, and .SET.IF, operations.

```
: BOK? ?# ?EXT OR IF 3 ERROR THEN ;
: BIT-BR <BUILDS C, DOES> C@ 18,Y BOK? MODE-LSB 10 -
  IF C OR THEN C, C, C, 2 C, 20 C, HERE 0 C, 2 ; IMMEDIATE

13 BIT-BR .CLR.IF,
12 BIT-BR .SET.IF,
```

Finally, a number of constants are provided that represent fixed addresses in the kernel. Much like the method used to establish the location of ERROR as described above, these constants are established by reference to fixed offsets from named words that are not likely to change in any potential future revisions of the kernel.

```
' @  CFA FE43 FE22 - +       CONSTANT PUSHD
' @  CFA FE47 FE22 - +       CONSTANT NEXTSD
' @  CFA FE47 FE22 - +       CONSTANT PUT
' @  CFA FE4A FE22 - +       CONSTANT NEXT
' @  CFA FE4C FE22 - +       CONSTANT NEXT3
' @  CFA FE50 FE22 - +       CONSTANT NEXT1
' @  CFA FE52 FE22 - +       CONSTANT NEXT2
' 1+ CFA FC97 FC7C - +       CONSTANT POP
' 1+ CFA FC93 FC7C - +       CONSTANT POPTWO

0 CONSTANT W
2 CONSTANT IP
4 CONSTANT UP

FORTH DEFINITIONS
```

The listing is closed by a return to the FORTH vocabulary and the indication that future definitions will be added to the FORTH vocabulary, rather than the assembler's.

Appendix C

A Masters thesis based on ARC Technology

by George Sergio Vega

1993

Chapter 1

Introduction and the Problem

Introduction

In today's world of technology, robotics is a fairly new and upcoming facet of industry. Robotics has emerged dramatically in the past 10 years and has shown more and more potential for future multi-industry use. As defined by the Robot Institute of America (as cited in Minsky 1985), a robot is a programmable multifunctional manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions or the performance of a variety of tasks. Robots are machine tools that may be found in the most unkempt of industrial worksites, to the highest level of clean and sterile environments. Most intelligent robots utilized in industry, in one way or another, incorporate drive/power systems, a base, and some type of programmable system. Arms, grippers, and manipulators are also utilized for performance of a variety of tasks. Both single-function and multifunction robots are sometimes used in conjunction to utilize all robotic features to their utmost potential. In today's industry, robots are utilized in numerous ways. These include arc and spot welding, paint spraying, glass handling, and precision assembly, to name a few. In the military, robots are being utilized to handle dangerous jobs such as guarding and defending nuclear storage facilities. They are also used to defuse bombs and patrol battle fields in the face of toxic chemical weapon attack. Robots work tirelessly in any environment, and do not require great technological sophistication of the user.

The Problem

In order to facilitate this investigation, a statement of the problem, including the purpose of the study and definition of terms used in the research effort, has been established.

Statement of the Problem

The purpose of this study was to design, construct, and test an autonomous mobile robot utilizing pre-designed components. The robot includes a basic vision and tactile system capable of optical and tactile collision avoidance.

Importance of the Study

Present-day society must be informed, educated, and exposed to the robotics industry and its potential. Knowledge of this industry will increase the possibilities for new opportunities and technological advancement. Educators, students, and technical training personnel will especially benefit from this study due to the fact that knowledge of robotics will soon become crucial in education and the technical training curriculum. Educators can use this study to introduce and expose students to the basics of the upcoming technology of robotics. The robotics project can be used as a hands-on manipulative exercise. Technical training personnel can use this study to expose and train robot users in industry on basic evaluation and implementation of robotics concepts. They, too, can use the robot as a manipulative device to demonstrate robotic functions. The study includes a brief overview of the history of robotic technology.

Definitions of Terms Used

In order to simplify interpretation and data presented in this study, the following terms and phrases have been defined.

Adaptable A robot's adaptability is determined by its vision system, force, and tactile sensors. These features enable the robot to make self-directed corrections with little human intervention (Cardoza and Vlk 1985).

Adaptive control. A method by which input from sensors changes in an attempt to achieve better performance. Control parameters are automatically adjusted (Cardoza and Vlk 1985).

Anthropomorphic robots Conventional robots designed with motion capabilities similar to those of the human body (Minsky 1985).

Arm The joints, links, slides, and mechanical design of a robot which support a moving tool, attachment, or hand (Cardoza and Vlk 1985).

Base The fixed platform to which a robot's shoulder is attached (Cardoza and Vlk 1985) Drive power system. The electronic, mechanical, or electromechanical components that allow the robot to propel itself from place to place and to do useful things with its end-effectors (e.g., arms) (Kelly 1985).

Gripper A mechanical device for holding, releasing, and perhaps also manipulating an object (Kelly 1985).
Intelligent robot. A programmable machine tool that can make performance choices based on sensory inputs (Cardoza and Vlk 1985).

Multifunction robot A robot built to do many things. It must be programmable and must be capable of movement and maneuverability. It needs manipulation tools (arm, wrist, hand, etc.), sensors, and intelligence (Kelly 1985).

Programmable robot A robot that can be programmed and reprogrammed to perform a variety of tasks (Kelly 1985).

Single function robot A robot built to do one single function. These robots are often bolted to the floor and do one thing well, such as painting, welding, and other activities that are dangerous, hard, repetitious, or that require constant precision (Kelly 1985).

Chapter 2

Review of Literature

Robotic technology has many benefits for today's industries. In the manufacturing industry, the technology of robotics has been used in the building of a sophisticated pocket pager. By controlling, reducing, and automating numerous fabrication and assembly procedures, the pagers can be completed in only 2 hours. In the firefighting industry, an automated fire extinguisher robot can fight oil well fires from close range without endangering human personnel (Hampton 1988).

The medical industry has benefited much in the past few years by the use of robotics. Robots have been used to position limbs during knee surgery and to guide instruments in brain biopsies, and in various surgeries involving animals. Robodoc, an android variation of an IBM assembly line robot, is specifically programmed to drill and prepare the human femur for the installation of critical parts of hip replacement surgery (Weiss 1991).

Robodoc is the first robot to take an active role on the operating table in hip replacement surgery. Prior to the surgery, the surgeon loads CT scans of the patient's femur into the robot's computer and continues all the planning and programming on a computer screen. Technical and drilling information is supplied by the implant's manufacturer (Weiss 1991).

During hip replacement surgery, Robodoc drills the cavity in the femur where the implant is to be inserted. This cavity is fitted with the implant, to which the hip socket is mated. Robodoc has pressure sensors which monitor the drilling pressure. This safety feature stops the robot from drilling if it starts to cut soft tissue. Robodoc's higher precision and accuracy in drilling the cavity for the implant provides 96% contact between the bone and implant, which is 10 times more accurate than a surgeon using a hand-drill and who drills only a 20% total contact area. Even though Robodoc is capable of doing other parts of the operation, the surgeon is still very much needed to start and complete this surgical procedure (Weiss 1991).

As the bone heals, it fuses into the porous implant. With this type of surgical implant, cement is not used to fill cracks or crevices. Robodoc's high degree of accuracy provides a better fit for the human patient, resulting in less pain, fewer failed operations, and usually a shorter hospital stay (Weiss 1991).

Robodoc is foreseen to be very popular and helpful to orthopedic surgeons whose bone-hollowing techniques include using a mallet and huge spike. Robodoc is predicted to be an asset for the future in general surgery. With program modifications, it may also be used for procedures required in surgeries such as repairing tiny bones in the middle ear, precision cutting in optical surgery, and excision of brain tumors (Weiss 1991).

As Robodoc's potential is becoming more evident, lists of human volunteers have formed to partake in clinical trials (Weiss 1991).

Much work and study has been undertaken to have robots do tasks that are considered to be too dangerous, unsanitary, or repetitious for human beings.

Background

History emerges when one explicitly recognizes and names a phenomenon or event; then, it follows, traces, and monitors the event. Prior to the 1920s, when robotic devices emerged as an idea, there was no conscious history of robotics. Ironically, we can track the technology of robotics back to at least 1500 B.C. Around 1500 B.C., Egyptian water clocks supposedly used human figurines to strike the hour bells. In 1557 Giovanni Torrianni made for an emperor a wooden robot that could fetch the emperor's daily bread from the store. In 1738 Vaucanson created a mechanical duck that could eat, excrete passable iso-olfactoric excrement, walk, quack, and do various other duck-like things except to fly. In 1890 Edison developed a version of a talking doll. Robotics continues to make history in this day and age with new and innovative ways to incorporate this technology in present-day industries. The term robotics was coined in 1921; from that point, we can trace the development of the concept in the general culture (Kelly 1985; Minsky 1985).

Various Robots

Robots come in many different shapes, sizes, and prices. Robots can be used for many applications, such as welding, machine tending, assembly work, space exploration, spray painting, loading and unloading, and in law enforcement. Underwater robotics is another facet which seems to be revolutionizing our world under the sea. In this section, robotic applications are covered in concurrence with the specific existing robots in these industries. The manufacturer, a brief summary, primary uses, significant selling points, and an illustration of each robot are included (Minsky 1985).

Welding Applications

The Cyro robot was the result of research and design in technology developed over a period of more than 10 years. The technological base concept evolved from the welding and testing techniques developed in the Apollo space program. The Cyro robot is constructed as part of a work platform, with a vertical base and one attached horizontal arm. The Cyro design is for a production environment, specifically for arc welding and the control of arc welding variables. A system called adaptive control is utilized to correct the weld path and to process parameters in real time (Cardoza and Vlk 1985).

The Cyro 750 robot is a five-axis rectilinear robot, which is all electric. It possesses a 3/4 square meter work envelope, and is capable of performing weld processes with exceptional repeatability while maintaining program accuracy. This robot possesses user friendly software, and can be taught utilizing a teaching pendant, through off-line programming, or numerical control through a terminal. This robot and control apparatus weighs approximately 6,540 pounds and requires approximately 35.5 square feet of floor space. Noted features of the Cyro robot include 64K random access memory, along with permanent program storage on tape cassette. This design is specifically geared for high speed welding, with a high degree of torch accuracy (Cardoza and Vlk 1985).

An illustration of the Cyro 750 robot may be seen in Figure 1.

Machine Applications

The ASEA robot has been manufactured since 1973. These robots have an average of approximately 98% up-time working through three 8-hour shifts in a 24-hour period. ASEA robots have been manufactured in the United States and in Sweden and other European nations. The ASEA Robotics Company is seeking to broaden the sensing abilities of its robots through tactile recognition and simulated vision systems. They have exhibited vision-equipped robots which can reduce programming time by at least 25% (Cardoza and Vlk 1985).

The robot model IRb 6/2 is capable of operating in difficult environments. Its quiet electric-drive design incorporates a control cabinet with a portable programming unit, a measuring system, and a servo system. The servo powered arm can lift up to a 13-pound load. It is lightweight and can accelerate and decelerate rapidly with a repetition accuracy better than +0.008 inch. This robot is capable of radial and vertical arm movement, rotary and bending wrist movements, rotary movements, and horizontal travel. These movements are key features of its six degrees of freedom. A series of touch-sensitive buttons, a proportional-speed joystick control lever, and a plain English alphanumeric display, are used in teaching the IRb 6/2 robot. Complicated programs used with multi-axis robots can be compiled with diverse items. Along with the integration of the whole operating sequence, editing, deleting, and adding steps can be accommodated easily. This teaching system also accommodates additional variations and modifications of sequences. Curve and point adaptability, within the robot's adaptive control, makes it possible for the robot to automatically make program adjustments. This reduces programming time and allows the use of the same program for items having similar but not identical shapes. The Rib 6/2 robot, with its vertical and horizontal arms on a pedestal base with its exclusive grip functions, is used primarily for machine tending, injection molding, cleaning of castings, parts assembly grinding, burring, polishing, trimming, piercing, and hot embossing (Cards and Elk 1985).

Figure 1. The Cyro 750 Robot (manufactured by Advanced Robotics Corporation, Newark Ohio Industrial Park, Building 8, Route 79, Hebron, OH 43025)

An illustration of the ASEA Rib 6/2 robot may be seen in Figure 2.

Assembly Applications

The Intelledex 605 is the pioneer in a class of robots called light assembly robots. The robot's design is not for lifting heavy loads, but for tasks that require dexterity and high precision. This robot is one of the most popular robots in industry and constitutes a high percentage of the total robotics market. The Intelledex 605, classified as a light assembly robot, made its debut in April 1984. This robot emphasizes the flexibility requirements that are called for in the robots used in the electronics manufacturing industry. It was designed with an integrated, maximally adaptable hardware-software system. This design feature supports the many applications which require adapters, special tools, and computer interfaces (Cardoza and Vlk 1985).

Figure 2. The ASEA IRb 6/2 Robot (manufactured by ASEA Robotics, Inc., 16250 West Glendale Drive, New Berlin, WI 53151)

The Intelledex 605 embodies a vertical base assembly to which is attached a series of pivoted/counterweighted or axially rotating arms. This robot also incorporates a pneumatic pincer mechanism and stepper motor, as well as an optical vision system. Equipped with an optional vision system, the robot is capable of recognizing as many as 100 different electronic parts. Another feature of the Intelledex 605 is its ability to emulate human arm and wrist movements. Incorporating a 3-feet square work table, the robot arm can describe close to a 4-foot circle. The robot's sophisticated controller is capable of accepting input from its optical sensors. These sensors can detect part misfeed from force sensors that detect the presence of an object positioned in the end-effector and sense excessive pressure.

The controller also accepts input from a bar code reader that reads identification information on most assembly components. The controller also supports an important safety feature by monitoring input from pressure-sensitive floor mats, and a light curtain consisting of photocells surrounding the work area. This safety feature ensures a clear area before work can begin. Robot BASIC is considered a specialized, high-level language which can also be run easily on most personal computers. This robot is also supplied with a teaching pendant, including a joystick and 18 switches. This feature allows manual control of the robot for path point-entry, positioning, tool operation, and functions of speed (Cardoza and Vlk 1985).

The Model 605 robot is primarily used in electronics assembly work and in assembly of computer components. This model is used to unload circuit boards from racks, align them on specific tooling jigs and fixtures, and then begin the process of work to be performed.

An illustration of the Intelledex 605 robot may be seen in Figure 3.

Space Exploration Applications

The Jet Propulsion Laboratory (JPL) Rover robot is designed with the combination of visual and manipulative systems. Its uses are geared for research and study, and has been utilized on planetary exploration vehicles such as the space shuttle. The JPL Rover's capabilities include a diversely impressive display of extraterrestrial robotic applications.
The JPL Rover weighs approximately 700 pounds. It is 59 inches long and 51 inches wide. Its appearance is somewhat automotive in form, and it is roughly the size of an office desk. This robot has an instrumental payload of 220 pounds. The JPL Rover's current design incorporates twin camera pylons, an antenna mast, and a single manipulator arm. Future models of the JPL Rover are projected to include supplementary manipulators and drills necessary for the retrieval of soil and rock samples. These future models will also embody a 200-watt integrated radioactive thermal generator (RTG) to provide the power supply to on-board systems (Cardoza and Vlk 1985).

The JPL Rover's mobility is credited to loopwheel assembly systems. The loopwheel assembly systems consist of the wheel and track assembly systems, similar to those found in military tank treads. These are positioned on the lower end of each of the four jointed legs. The suspension system is comprised of interconnected thigh- and knee-like joints, making it possible for each foot to adjust independently to variegated terrain. The Rover moves at a speed of approximately 3 feet per minute. It can overcome depressions and obstacles within a zero to 24-inch range. The Rover can also descend and ascend on slopes to 30 degrees (Cardoza and Vlk 1985).

Figure 3. The Intelledex Model 605 Robot (manufactured by Intelledex Corporation, 33840 Eastgate Circle, Corvallis, OR 97333)

The JPL Rover's vision/guidance system is comprised of two television cameras set upon pylons attached to the chassis. These guidance cameras are wired to an on-board computer system. A laser range finder is also utilized to measure the distance between the robot and any possible obstacle. The video and range finder data are processed by a computer that then creates the best possible course to a specific destination. The robot is then guided along the charted path. The robot's vision system is also utilized to ascertain the distance between the manipulator arm and any other matter which it is subject to handle (Cardoza and Vlk 1985).

The JPL Rover's primary use is in scientific investigations in planetary and outer space research. It has been utilized to gather and relay scientific data from extraterrestrial regions back to earth for research, study, and interpretation. The

Rover is issued broad commands that initiate autonomous proceedings throughout the day. This feature allows the robot to perform a variety of tasks and scientific investigations with minimal instruction (Cardoza and Vlk 1985).
An illustration of the JPL Rover robot may be seen in Figure 4.

Spray Painting Applications

The Armstar Tokico Painting Robot is used worldwide in the automotive, appliance, plastics manufacturing, and electronics industries. It has been available since 1978 for a cost of approximately $150,000. The Armstar robot is used to accomplish industrial tasks such as assembly line undercoating, top coating, interior/ exterior priming, and adhesive application. It is primarily used for the toughest, most difficult of spraying and finishing tasks. Key features such as microprocessor-controlled pathminder finishing and high speed optical scanning capabilities allow this robot to be one of the fastest continuous-path robots in the industry (Cardoza and Vlk 1985).

Figure 4. The JPL Rover Robot (manufactured by Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91103)

The Armstar robot's design includes the use of a hydraulic power supply, a modifier console, a microprocessor console, and a manipulator. Two carriers and a rotation control panel are incorporated for automatic operation. This robot is capable of repeating painting operations automatically and continuously in accordance with previously memorized programs. It also has its own self-diagnostic system and innovates simple program editing. The robot is taught programs by one of two methods: continuous path or point-to-point. Continuous path is the method in which the painting operation is programmed by movement of the robot arm in accordance with how a human handles the spray gun. In the point-to-point method, the worker separates the movements of the robotic arm into linear motions; only the starting and stopping points are programmed. The robot's vertical and horizontal movements of its wrist allow it to perform complicated work tasks (Cardoza and Vlk 1985).

A pattern rotation of 210 degrees is possible with the Armstar six-axis robot. The arm movement is capable of moving 100 degrees horizontally, 75 degrees back and forth, and 70 degrees vertically. The robot's flexible wrist, similar to that of a human wrist, is capable of moving 210 degrees, with 260 degrees of wrist rotation. The Armstar robot's six elements are: speed of the robot, selection of spray patterns, continuous sweeps or back and forth sweeps, increasing/decreasing teaching points, spray on or off, and position and angle at each type of movement (Cardoza and Vlk 1985).

The Tokico painting robot is used primarily in the application of various coatings to objects of all shapes and sizes. Armstar's greatest technical advance has been with the primer/surfacer function. This robot can also be used in the application of adhesives to automobile windshields, including windshields with compound curvature, prior to direct glazing.

An illustration of the Armstar Tokico Painting Robot may be seen in Figure 5.
Loading/Unloading Applications

The Mobot Robot, manufactured by Mobot Corporation, is one of the world's largest parts handling robots. Mobots are utilized in various industries. Unlike conventional robots, also known as anthropomorphic robots, which are designed to duplicate human body motions, the Mobot is purposely designed with limited motion capabilities. This factor allows the Mobot to be priced from $15,000 to $40,000. Conventional industrial robots are priced from $45,000 up to approximately $900,000.

Mobots are considered to be second-generation robots, which generate straight-line motions directly by way of vector motion modules. First-generation conventional robots require servo-synchronized and complicated computer-generated motions to achieve straight-line movement. Mobots require fewer axes to achieve the same tasks with the bonuses of additional travelling distances, minimal electronic hardware, simplified programming, and easier maintenance. In addition, the Mobot can be acquired at a moderate system cost. Mobots also have the reputation of being highly reliable. It is possible for a Mobot to pay for itself, working a single shift, in a one-year period (Cardoza and Vlk 1985).

The Mobot Robot is a point-to-point robot used for material/parts handling. The robot's design is geared for the loading and unloading of molding machines, printed circuit manufacturing machines, conveyors, skids, pallets, and a diversity of machine tools. The Mobot is designed for use in limited floor space conditions. The Mobot's design consists of a series of modules with multi-power and position control options. This modular feature makes it possible to construct a Mobot for a specific work task with minimal effort. This robot can be constructed and

Figure 5. The Armstar Tokico Painting Robot (manufactured by Tokico, 15001 Commerce Drive North, Dearborn, MI 48120)

modified for an array of motions and tasks directly at the worksite. Its design also incorporates a carousel for the transfer of objects from rotary to linear conveyors (Cardoza and Vlk 1985).

The Mobot is based on a heavy-duty square box mast. It can be obtained with a single-column configuration which is capable of lifting up to 250 pounds; it is also available in a double-column carousel configuration which is capable of lifting up to 1,000 pounds. A roller-bearing carriage running on precise steel rails is utilized for vertical movement. Dual safety chains, rated at 8,000 pounds each, are utilized in tandem. A sensor system is incorporated to search for slack or broken chain conditions. Self-locking gearboxes are also utilized with normally-on brakes. This feature helps to prevent a moving load from falling down due to a loss of power. Mobots are designed with all mechanisms located overhead to minimize floor clutter and space requirements.

Mobot robots have been designed to operate efficiently in industrial and scientific industries. These robots operate well in heat, dust, and abrasive conditions, and solvent vapor and corrosive areas, as well as in ultra-clean and sterile conditions. The Mobot's highly modular structure and interchangeable parts make it adaptable for numerous industrial applications (Cardoza and Vlk 1985).

An illustration of the Mobot Robot may be seen in Figure 6.

Figure 6. The Mobot Robot (manufactured by Mobot Corporation, 980 Buenos Avenue, San Diego, CA 92110)

Law Enforcement/Dangerous Task Applications

The Pedsco Robot is used primarily for bomb defusing and removal, security surveillance, nuclear waste disposal, and riot control. This robot, manufactured in Scarborough, Ontario, is moderately priced at approximately $20,000 to $50,000 per unit. The Pedsco robot is utilized by numerous SWAT teams and bomb squads in both United States and Canadian police departments. This Canadian robot is also highly utilized at numerous nuclear reactor sites (Cardoza and Vlk 1985).

The robot's design somewhat resembles a tank-type configuration. Its horizontal cylindrical body is mounted on a six-wheel base which allows multi-directional mobility. Two arms with mechanical hands, which can be controlled remotely, are incorporated in the robot's design. One of the two arms is capable of lifting up to 70 pounds. Both end-effectors can be programmed for powerful grip and soft-touch modes. The grip mode selection is usually dependent on the weight of the object to be manipulated or moved (Cardoza and Vlk 1985).

The Pedsco robot incorporates a vision system comprised of a remote-controlled television camera capable of 360-degree rotational coverage. An X-ray vision system is an available option. The television camera is also removable, to allow the incorporation of a riot shotgun, one of the robot's primary options.

The Pedsco robot can perform many duties, including removal of toxic substances and waste, security and sentry duty of designated areas under hostile conditions, and deactivating and disposing of explosive devices. However, the robot's main function is to perform hazardous tasks in areas dangerous to human personnel (Minsky 1986).

The potential operator of the Pedsco robot can be easily trained and need not be an expert programmer. The operator controls the robot at a safe remote distance via a television monitor and an array of electronic control devices. The robot's highly sensitive control system requires the operator to possess manual dexterity skills necessary for the task to be accomplished.

The CURV III Underwater Recovery Vehicle and the Free Swimmer Submersible Robot

The U.S. Geological Survey Group at the Naval Ocean Systems Center (NOSC) funded and supported the development of the cable-controlled underwater recovery vehicle (CURV III). The robot's design includes an extendable manipulator arm, underwater light fixtures, and numerous tools necessary for submerged aquatic tasks. The CURV III robot is cable controlled and can operate at depths of up to 10,000 feet (Cardoza and Vlk 1985).

The CURV III robot was designed and developed for a variety of aquatic tasks. Some of these tasks include ocean engineering, search, salvage and recovery, and underwater inspection. Additional applications of the CURV III robot include pollution prevention and safety maintenance supporting offshore pipeline and drilling operations (Cardoza and Vlk 1985).

An illustration of the CURV III robot can be seen in Figure 7.

The Free Swimmer robot, similar to the CURV III robot, was also developed by NOSC. However, this model is untethered and used for similar applications in shallower depths. The Free Swimmer is limited to an operational depth of 2,000 feet and can operate for approximately one hour with energy supplied from its own independent power source. The

Free Swimmer's design includes motion picture or television cameras, including the necessary lighting required for video and filming. A fiber-optic communications link is utilized to allow the transmission of real time video signals to the operator's base. A magnetic pipe sensor is also included in the

Figure 7. The CURV III Underwater Recovery Vehicle Robot (funded and supported by Naval Ocean Systems Center, United States Navy)

design to allow the Free Swimmer robot to locate and follow metallic pipes autonomously, for investigative and inspection purposes (Cardoza and Vlk 1985).

The Haz-Trak Excavator Handler

Far off, at a distance in the middle of a radioactive waste site, a Haz-Trak Smart Shovel safely lifts and moves barrels filled with radioactive sludge. The smart shovel prototype appears to be an ordinary hydraulic excavator, except that this robot does not have a place for an operator. Instead, there are three cameras mounted above the engine area. Hundreds of yards away from the worksite, inside a building, an operator controls the excavator by the use of a hand grip and joystick while viewing the shovel's actions on several video screens (Langrath 1992).

The Haz-Trak Smart Shovel is unique. It is considered a pioneer in intelligent machinery in the low-tech world of heavy construction equipment. The Haz-Trak Excavator and Material Handler is designed by KRAFT TeleRobotics. The Haz-Trak is designed to function as an extension of the operator's hand. It is supposed to feel obstacles that the shovel may encounter (Langrath 1992).

The remotely controlled Haz-Trak robotic shovel is capable of conveying resistance from the worksite back to the operator's hand miles away. This feature makes the dangerous task of hazardous waste removal much safer and simpler.

Haz-Trak is known to be easier to master than most video games. An operator is capable of learning how to use Haz-Trak in a matter of minutes, versus the hours usually required to learn the use of standard construction equipment (Langrath 1992).

The Haz-Trak robot is ideal in performing delicate applications such as working and digging around pipes. Unlike conventional excavators and handlers which require multiple levers for control, the Haz-Trak uses a joystick and handgrip. This allows the robot to prove its usefulness in applications such as cleaning up hazardous chemical and radioactive wastes at weapons laboratories, removal of unexploded munitions buried at military sites, and searching and probing for buried wastes. It is also capable of doing these feats without the risk of puncturing buried containers and causing a toxic waste spill (Langrath 1992).

The Haz-Trak is a somewhat larger scale version of an existing robot called a grips manipulator. The grips manipulator consists of a 4-1/2 foot-long remote control robotic arm on a fixed mobile base.

There are two crucial and beneficial features of the Haz-Trak robot: the use and combination of force feedback and master-slave control. Master-slave control allows the robotic arm to perform exactly the same motions and movements as the operator performs on a joystick/handle controller. Force feedback is the feature that allows the robot to convey the resistance which the machine encounters back to the operator's hand with controlled sensitivity. For example, if the arm hits an encountered object, the hydraulic activators picks up this resistance and changes it to a radio signal which is then directed back to the controller. The controller receives this information by electronic activators and converts the resistance at a lower, more manageable level. Neither master-slave control or force feedback work wells without the other. Master-slave control helps to keep the controller's hand movements synchronized with the robotic arm and force feedback enables the precise control of gripper pressure from approximately one pound up to 250 pounds of force (Langrath 1992).

Although these two technologies have been available and used in nuclear power plants and other industries, Haz-Trak is the first excavator vehicle to combine the use of these technologies.

The Haz-Trak robot is designed to incorporate a store-bought excavator, thus keeping costs to a minimum. The first production model was designed to fit a variety of industrial tool attachments, including a shovel and a barrel handler. The Haz-Trak robot's capability to memorize movements and repeat them on command is beneficial in performing tasks that require precision and repetition. This robot is destined to make the entry of heavy industrial equipment into today's advanced technologies.

Similar Studies

In May 1983, Robert Hardy Demmon conducted a project study on a vertical milling machine. His statement of the problem was to design, construct, and test a milling machine capable of being produced and used in high school metalworking laboratories. In the conclusion to his study, it was determined that a vertical wheel stationary pan mulling machine was the most popular and readily available machine. With this in mind, an experimental prototype model of a vertical mulling machine was designed and fabricated. The overall design incorporated these desired features: (a) portability of machine, (b) power source of 110-volt standard current, (c) user friendly and fully operational in a high school shop environment, and (d) utilization of standard hardware and materials usually available from a local hardware store.

The overall project was successful; however, during the operational testing, some mechanical and operational problems were discovered. Demmon's recommendations for improvement were as follows: (a) addition of small vertical blades, (b) redirecting the angle of the discharge chute from the machine, (c) addition of a protective shield skirt, and (d) addition of a self-closing catch mechanism for the discharge door.

Considering these modifications, the future design, construction, and use of this type of sand mulling machine shows much promise for the educational shop environment.

In August 1988, Michael A. DeMiranda conducted a project study to design, construct, and test a performance vehicle. This vehicle would be used in a middle school Explorations in Technology instructional module. Personal interviews and data obtained and compiled from industrial technology educators established the design criteria to be used. Once the design was established, construction of the vehicle was under way. Design changes and modifications were made during the course of fabrication. Features such as suspension, steering, brakes, and alignment were designed to directly relate to land transportation vehicles and to fulfill educational standards set by the California State Department of Education for Industrial Technology Explorations.

Upon conclusion of design and construction of the performance vehicle, each system was tested for operation and safety. It was found that the vehicle met the established criteria. As for future recommendations for improvement, the following were determined: (a) the small engines module be coordinated with the performance vehicle modules for continuum purposes, (b) a hydraulic line lock be installed for operational safety, (c) explore positive lock fasteners to prevent fastener loss and component failure, (d) check systematic systems prior to use, and (e) develop performance vehicle module curriculum guide. It was also recommended that further study be conducted to improve the educational benefits of incorporating the performance vehicle into the middle school curriculum.

Robotic Mechanisms of the Future

United States industry is still in early stages of robotic technology. There is no accurate prediction of what robots may be like and capable of in the next few years. It would be safe to assume that future breakthroughs in robotics are dependent on the entrepreneurial instincts of robotic engineers. They are also dependent on how the general public responds to the availability of industrial and personal robots (Kelly 1985).

Japan's productivity gains and advanced technological accomplishments are clearly evident in their use of robots. This industrial fact has affected and motivated industry in the United States. It has also caused the market for industrial and personal robots to increase immensely in the past 5 years. Some American manufacturing companies have been using tooling facilities which are approximately 25 to 30 years old and seriously out of date. Influenced by the capabilities of advanced technologies utilized by foreign industry-wise nations, an apparent major drive seems to be under way to re-tool these U.S. manufacturing establishments with robotic tooling technology and systems. This nationwide re-tooling phase is evident in small as well as large corporate environments (Kelly 1985).

Currently, in today's market for industrial robots, robotic engineers are developing special purpose and limited task robots. As the market grows and becomes stronger, it is foreseen that the future of robotic technology will stimulate and create the demand for general purpose and versatile industrial robots which are economical, adaptable, and easily modified to operate in diverse environments (Kelly 1985).

The development of agricultural and personal robots is well under way. Prison guard robots, Australian sheep-shearing robots, and fruit picking robots are just a few examples of the versatility and potential of robotic technology. The future of robotics looks promising. Features such as accuracy, dependability, and high productivity strengthen the potential and positive capabilities of robotics (Kelly 1985).

Chapter 3

Designing an Autonomous Mobile Robot

The purpose of this study was to design, construct, and test an autonomous mobile robot.

A robot's design is usually based on the requirements necessary for the function and application for which the finished product is intended. The robot designed for this project and study was intended to demonstrate selected robotic functions of an autonomous mobile robot. The key factors and design parameters of this particular robot included total mobility, optical and tactile collision avoidance, and an independent untethered power source and programming system. Additional supportive features included sound effects, dual motor drive control, and interactive and instinctive level programming.

The design and fabrication techniques selected and utilized were based on the main criteria of economy, safety, and utilization of existing pre-designed sub-assemblies and parts. In addition to the design criteria, prototype parts and sub-assemblies were designed, fabricated, and utilized in the final assembly process.

Optical Collision Avoidance

The design of this robot incorporates four optical collision detector sensors. These detectors emit a high intensity light source, produced by the use of LEDs, which is monitored for any reflective activity. This reflection is monitored to determine if there is an existing obstacle. Monitoring is achieved by incorporating a light-sensitive photodetector called a phototransistor photodarlington. This system also utilizes a trigger level adjustment factor which ensures that any reflectance is valid and not environmental ambient light noise. The trigger level also controls the sensor sensitivity and achieves a reading differential between ambient light and valid obstacles. This is possible due to the fact that the light detectors also act as variable resistors. Resistance variances are caused by increasing or decreasing light sources. When there is no light present, the resistance is at its maximum. Exposure to light causes the resistance to be reduced.

Tactile Collision Avoidance

This robot's design also incorporates two tactile collision sensors. These sensors are constructed of spring steel piano wire. The wires are mounted on the front left and right corners of the robot's base. The sensors are securely connected to terminal posts on the CPU board. These wire detectors can be activated by either a sideways pull or push motion. When an obstacle is encountered, the tactile feelers are either deflected or depressed, which activates one of two position contacts on the CPU terminal posts. Depending upon which contact was activated, the responsive action of the robot is determined by the instinctive level of programming loaded in the CPU memory for a predetermined action of obstacle avoidance, such as pivot left, pivot right, reverse, or forward.

Dual Motor Control

This robot's design utilizes two 12-volt DC gear reduction motors and two 6-inch diameter wheels for full mobility. Each wheel is directly mounted on the motor shaft via a specially machined adapter. These wheels are mounted slightly above the lower baseline within the robot base.

The steering system is achieved by the use of a systematic technique termed differential direct drive. This concept allows the robot to execute multiple steering positions by simply reversing the direction of one motor in respect to the other motor's direction. With this concept in mind, adjusting the motor speeds to rates different from each other during operation regulates and changes steering capabilities.

Motor Speed Control

Pulse width modulation (PWM) is utilized for the robot's motor speed control system. This technique allows the control of the drive integrated circuit located on the central processing unit (CPU) to operate on a 100% full-on or 100% full-off mode. Therefore, a pulse train of electrical power consisting of on and off periods of equal time intervals (50% duty cycle) is utilized to drive the motors. Minimal power is dissipated by the semiconductor devices when operated in this manner. Each motor is capable of being powered and controlled in this fashion.

Additional motor speed is achieved by increasing the duty cycle, which, in essence, increases the on-period and reduces the off-period accordingly. This process increases the current power supply to the motors. Hypothetically, if a 75% motor speed requirement was prompted, the duty cycle would be increased to a 75% on and 25% off pulsation mode. This process of duty cycle adjustment is governed and controlled by the software architecture via the instinct level and by current programmed requirements. The electric power source pulsation is supplied to the motors in such an accelerated manner that the mechanical inertia of the robot smooths them out entirely. This action allows the robot to execute an average speed proportional to the pertinent duty cycle.

Pulse width modulation was selected based on the main criterion of a system capable of efficiently performing speed control tasks while also economizing electrical power consumption. These were crucial factors because of the limited amount of electrical power allowably stored in the robot's 12-volt 4-amp-hours (4-AH) rechargeable battery. Regulating the robot's electrical power consumption in an economically feasible fashion, while maximizing the efficiency of electricity used, allows the robot to function for prolonged durations of optimal operation.

Sound and Aural Effects

This robot is supplied with an electret condenser element which is a highly rated sensitive microphone for audio detection. Current aural effects include the monitoring of sounds produced within a local vicinity and memorizing the location of origin of this specific sound wave. Seeking the loudest sound in the vicinity, memorizing its point of origin, and proceeding to the designated location and executing a prompted command is just one of the main commands within the robot's hearing capabilities.

The design of this robot also includes a 3-inch transducer speaker connected to the digital output (SS1) from the CPU through an amplifier device (Q2). Selecting the correct parameters causes the speaker frequency output to be changed. This process is initiated by executing the cycles command within the software program. The software architecture includes four octaves of musical notes along with other various synthesized sound effects. This sound system allows the robot the capability of producing audio feedback as well as audio special effects. Audio feedback and sound effects are used in identifying certain system warnings, program debugging information, and satisfying any other aural or acoustic program requirements.

Independent Power Source

This robot is completely autonomous and untethered. The robot does not require any type of external cables or any remote source of electrical power.

A rechargeable 12-volt 4 AH hermetically sealed gel cell battery is utilized for the robot's electrical power source system. The battery is strategically mounted within the robot's base, with weight distribution factors considered. A float charge method regulated at 13.6 volts is utilized in recharging the battery. This method allows the charger to be activated and connected on a continuous recharge mode. The CPU is also supplied with an integral battery charging circuit, which prevents the battery from being overcharged. The instinct level software monitors the battery voltage set by the user and is capable of executing an alert signal when the battery voltage falls below the pre-set level.

Software Architecture

High-level programming on the CPU board is possible by utilizing FORTH computer language. The programming process for this robot was made possible by the use of a personal computer, using an interactive terminal program. The actual programming process may be manually switched to operate in an interactive or programmable generation mode. Basically, the interactive mode instantaneously executes a command as it is typed in and requested. This mode was utilized during testing operations. The programmable generation mode operates in the manner in which a series of commands is typed in and saved in program form. This unique program may be edited, deleted, or downloaded into the robot's CPU, which is capable of storing the data in a memory chip. This newly stored program is available to be automatically retrieved and executed at any requested time when the robot is activated.

Instinctive Level Software Architecture

The instinctive level of programming was considered the critical level of programming. These background tasks initiate instantaneous predetermined responses to outside stimuli of the current active sensors. This level does not elaborately process or determine the cause and description of a responsive action. A predetermined reactive command is simply executed. Hypothetically, if an obstacle is detected, the initial reactive command is to stop, avoid obstacle, and redirect course to seek a clear path. There is no processing of available alternate courses or determination of the type and size of the obstacle sensed. The initial command is based on reactive information, as distinguished from processed information.

Behavior Level Software Architecture

The behavior level was considered as the intelligent level where actions are based on system status. Processed tasks and decisions on obstacle avoidance are easily processed and attained. This process of programming monitors multiple sensors and hardware to distinctively identify and process a prompted behavior or sensed obstacle. An intelligent course of action is then determined. This programming level is also capable of making perceptive decisions and conclusions. A time element is involved in which a series of commands is processed before the most practical and profound command is selected and executed. Therefore, if the robot were to sense an obstacle, the behavior level would then automatically direct the robot to stop, reverse direction, and perform a series of maneuvers to avoid the obstacle. The maneuvers are based and derived from the information and facts processed by the instinctive and behavior levels of intelligence.

Chapter 4

Construction of an Autonomous Mobile Robot

The robot's body, base, and head were designed and fabricated utilizing principles and methods required to achieve a unibody design. A unibody design is simply the strategic fabrication, formation, and fastening of a multi-part configuration. This particular type of design does not require the use of any type of frame or foundation. The unibody design must serve as the nucleus of the configuration desired and provide the crucial factors required for strength, stability, and overall unit support. These critical factors are attained when all components of the project are assembled and secured.

The Base

The initial fabrication process for the base began by constructing the motor mounting channels. Two mounts were required for this particular design in order to accommodate two gear reduction motors and two wheels. An estimated sheared piece of aluminum sheet metal was measured, dimensioned, laid out, cut to size, and all necessary holes drilled or punched out. PEM nut threaded inserts were then pressed into holes requiring a threaded interior. These fabricated motor mounts were then formed and bent to final detail specifications.

The base casing was fabricated in the same manner. The motors were attached to the already fabricated motor mounts along with the installation of two required wheels. The base case required additional fabrication for an elongated rectangular opening for the serial personal computer (PC) cable jack. This rectangular opening was cut out on the back side of the base casing box. Three additional holes were drilled to accommodate the on/off toggle switch, a reset button, and the battery recharge receptacle jack. A small angle-like piece of aluminum was measured, cut to size, and mounted above and around the toggle and reset switch to serve as a guard from possible collision damage. PEM nuts were installed into all drilled holes with threaded requirements necessary for the assembly process. The motor mount assemblies were then dimensionally located, attached to the robot base, and secured with custom-fabricated L-shaped support brackets.

The Optical Vision System

Three pieces of aluminum material were laid out, cut, and drilled to the required specifications for the optical vision system. The three aluminum shapes were then bent to a channel-like shape to accommodate the receivers and LEDs required. This particular configuration was necessary to fulfill component protection requirements.
A small section of blank perforated circuit board was cut and hand-shaped to accommodate the series of LEDs and receivers (vision array) mounting requirements. The vision array and circuit system were assembled and mounted to the channel guards. The longest of the three completed vision array components was centered and mounted to the front surface of the base. The two remaining vision array components were dimensionally located and mounted onto the front left and right side corners of the robot base. The three-part vision system was mounted, using extender fasteners to allow for wiring and assembly distance requirements.

The Tactile System

Two pre-measured square holes were cut out on the upper front surface of the robot base to accommodate the installation of the tactile system. These holes were strategically placed directly in front of the pre-determined central processing unit (CPU) board location inside the robot base. Two pre-measured lengths of piano wire were then cut accordingly and the ends were formed to meet assembly and safety requirements. One end of the tactile feeler, formed in a U shape, was attached to the robot through the square-cut holes on the robot base. This end was directly attached to the CPU board with a fitted bolt and nut. The tactile feeler ends which protruded from the robot were formed with a closed square shape. This square shape was meant to prevent damage to any surfaces or obstacles encountered by the feelers.

CPU and Battery Installation

The CPU board was placed on the inside upper surface of the robot base to transfer hole locations for the assembly process. The holes which were laid out for this component were then drilled to diameter. Appropriate-size nuts and bolts were then used to attach the CPU board to the inside upper surface of the robot base.
The battery was strategically placed with weight and balance considerations as the criteria. The final location of the battery was determined to be at the rear of the inside upper ceiling of the robot base. A pre-measured piece of aluminum material was laid out, cut, drilled, and bent into a U-shaped type bracket. This shape served as the battery mount bracket, which was then placed on the predetermined battery location. Required holes were dimensionally transferred and drilled. PEM nuts were installed in selected holes required for final battery mounting and installation.

Support Casters

Initially, a crucial dimension was determined for the construction of the caster assembly. Distance measurements were taken from the lower edge of the robot base to the floor surface. This measurement was part of the criterion considered for the selection of the two casters, due to variations in caster heights and shapes. The type of mount was also based on this criterion. A channel bracket mount configuration was selected and fabricated. A pre-measured piece of aluminum stock was cut, dimensioned, drilled, and bent to the desired form. The two casters which were selected were supplied with an attached mounting plate. This plate was placed on the caster mount bracket and the pre-drilled holes were dimensionally transferred and drilled onto the bracket. The casters were attached to the completed bracket, which was mounted to the robot base by the use of machine screws and PEM nuts. The desired floor-to-base height dimension was then confirmed. The bracket with casters was then tightly secured to the robot base.

The Body and Neck Assembly

Five selected pieces of aluminum material were laid out, dimensioned, and cut to satisfy the body size requirements. All necessary holes were drilled and burred. The holes requiring threaded interiors were equipped with threaded PEM nut inserts. The panel that required assembly bracket-type angles was formed on the Box and Pan Brake. These five fabricated parts served as the front, left, right, top, and rear body panels.

The side panels were cut at slight angles to provide a wider and deeper lower base area necessary to provide a stronger base support. This was a required design factor which contributed to the overall strength and support of the total assembly.
The top side of the robot base was then dimensioned for a round cut-out which was drilled and cut. This allowed for the electrical and communication wiring from the base to extend through the body, neck, and head.

The front panel required two additional holes. The first hole was dimensioned, located, and drilled to size. This hole accommodated the power monitor flashing LED which was installed and wired to the appropriate wire leads that originated from the base. The second hole was dimensioned, located, and drilled to size. This hole accommodated the required condenser microphone which was installed and connected to the appropriate wire lead, which also originated from the robot base. The rear panel required no additional procedures and was then installed and assembled with the top and side panels. This body sub-assembly was attached and fastened to the robot base.

The neck was fabricated using a length of clear Plexiglass tubing. This tubing was cut to the desired length dimension. Small aluminum L-shaped brackets were fabricated. These brackets were required to install and attach the neck onto the centered location of the top panel of the robot body. The completed brackets and neck were drilled to size and fastened with machine screws and PEM nut threaded inserts.

The Head Assembly

The fabrication of the head assembly was initiated by cutting two estimated sheets of aluminum material. These two pieces of material were laid out, dimensioned, cut, and drilled to size. All drilled holes requiring a threaded interior were supplied with threaded PEM nut inserts. The two elongated ends of each fabricated part were bent to bracket-like forms on the Box and Pan Brake. These formed ends would later attach to each other when the desired box-like configuration was formed and assembled.

The bottom face of the head was drilled to allow access to the wire leads of the robot's wiring system. The front face of the robot's head required three holes. These holes were located, dimensioned, and drilled to size. The upper two holes were required to accommodate two red LEDs which served as power monitors as well as aesthetic personality features. The third hole was required to accommodate a yellow LED which served in the same manner as the two red LEDs. The three LEDs were installed and connected to the appropriate wire leads which were available through the robot's neck.

The top of the robot's head required additional fabrication procedures. One hole was required to accommodate a green power monitor light, which also enhanced the robot's appearance. The green light was installed and connected to the appropriate wire leads, made available through the robot's neck. A 3-inch circle was dimensioned and centered on the top of the robot's head. A series of small holes were drilled within this circle's circumference for a grill-like effect. This grill-like effect was necessary for sound emission from a 3-inch speaker. This speaker was installed and wired directly behind the grill, inside the robot's head. The two completed robot head sub-assemblies were joined and fastened together to form the head's box-like configuration. This box-like head was attached and fastened to the neck through the use of two pre-fabricated brackets, screws, and PEM nut threaded inserts.

An illustration of the CPU board silk screen drawing may be seen in Figure 8. A three-dimensional robot detail list view is shown in Figure 9, accompanied by the robot detail list in Table 1. A front and side blueprint view is shown in

Figure 8. Silk Screen Blueprint View of Kosmo the Robot (horizontal)

Figure 9. Three-dimensional Blueprint View of Kosmo the Robot (horizontal)

Table 1. Robot Detail List for Figure 9

Detail Letter	Quantity	Description
A	3	Power Monitoring LED Array
B	1	Neck
C	1	Electret Condenser Microphone
D	2	Tactile Collision Avoidance Assembly
E	2	Spherical Metal Support Casters
F	2	Motor, Gearbox, and Wheel Assembly
G	1/1	3-inch Speaker/Power Monitor Light
H	1	Robot Head
J	1	Robot Body
K	1	Flashing Battery Monitor LED
L	1	Robot Base
M	3	Optical Collision Avoidance Assembly

Figure 10, and a schematic drawing of the robot's wiring system is shown in Figure 11, accompanied by the wiring system schematic list in Table 2.

Various views of the completed robot are shown in photographs in Figures 12 through 16. The Testing and Demonstration Program; the Hardware, Materials, and Parts List; and the list of Tools and Equipment Utilized may be found in Appendices A, B, and C, respectively.

Figure 10. Front and Side Blueprint View of Kosmo the Robot (horizontal)

Figure 11. Schematic Drawing of the Robot's Wiring System

Table 2. Wiring System Schematic List for Figure 11

Detail Letter	Description
A	Center Optical Vision Array
B	Left Optical Vision Array
C	Right Optical Vision Array
D	Left Motor
E	Right Motor
F	Computer Serial Connector
G	Electret Condenser Microphone
H	System Reset Button
I	Battery Charger Receptacle
J	Power Monitor LED Array/Flashing Battery LED
K	On/Off Toggle Switch
L	DC Battery
M	Left Tactile Feeler
N	Right Tactile Feeler
P	Central Processing Unit
R	Magneto Speaker

Figure 12. A Front View of Kosmo the Robot

Figure 13. A Rear View of Kosmo the Robot

Figure 14. A Three-dimensional View of Kosmo the Robot

Figure 15. A Bottom View of Kosmo the Robot

Figure 16. A Top View of Kosmo the Robot

Chapter 5

Summary, Conclusions, and Recommendations

The purpose of this study was to design, construct, and test an autonomous mobile robot fabricated and assembled with the use of pre-designed components. This robot included a basic vision system capable of optical collision avoidance and a tactile collision avoidance system. This prototype robot successfully met these goals.

Summary

Various types of robots are used in industry. Each type is unique in its capabilities and intended uses. By reviewing different sources of literature, attending robot club meetings, and interviewing persons associated with robot construction, the gathered information enabled the design and construction of an autonomous mobile robot.

The robot which was constructed was based on a unibody design using aluminum sheet metal, pre-designed components and sub-assemblies, and prototype-fabricated parts and assemblies. An independent power source and on-board CPU were used for programming and multi-electrical functions. A personal computer, utilizing a terminal mode software, was used to program the robot to perform interactive as well as various programmed functions.

The prototype robot was capable of executing optical and tactile collision avoidance. Additional features and functions included sound and aural effects, full untethered mobility, interactive and instinctive level programming, and dual motor drive control.

Operation of the robot required entering a specific program into the robot's CPU memory, then activating the on/off toggle switch. Initially, the robot would execute the calibrating commands, then follow through with the entered programmed prompted commands. A sample of programmed commands would include forward, left pivot, right pivot, reverse, and play the song "Clementine." The robot would complete its programmed commands and automatically go into a "wander" programming mode. The "wander" programming mode allowed the robot to operate with mobility, avoiding all obstacles, while executing various background commands.

Conclusions

The robot performed well for a prototype version of a fully mobile autonomous mobile robot. However, the following minor mechanical and electrical operational problems were observed. Therefore, from a mechanical and electrical standpoint, specific improvements were made to the autonomous mobile robot.

1. The robot's stability was imbalanced at certain points, which caused it to tip over. Therefore, larger spherical ball bearing casters were substituted for the original plastic wheel-like casters, allowing for stability and minimal caster swing.

2. When the robot was programmed to move, it lacked sufficient torque and power to fully maneuver itself. Therefore, larger and heavier duty motors, gear boxes, and wheels were substituted to provide higher torque and mechanical driving power.

3. The robot's electrical power supply prevented the motors from running to full capacity. Therefore, a 12-volt 4-amp battery replaced the original 12-volt 2.2-amp battery, allowing an increase in the electrical power supply.

4. The robot's battery did not hold a full charge, causing programs to malfunction. Therefore, the battery charging unit on the robot's CPU was modified to accommodate twice the charge current, which required that a heat-sync be installed to prevent overheating within the CPU.

Recommendations

From the information gained in this study, the following recommendations are warranted:

1. Further study must be done in all areas of robotics in order to continue to allow for technological and industrial advancements.

2. The incorporation of an independent robotic arm would be beneficial and can be easily adapted to this particular design to allow for point-to-point manipulation capabilities.

3. Software recommendations include the further study and implementation of goal completion behaviors, animal behavior simulations, goal selection monitoring, and goal/environment mapping for broader programming capabilities.

Thesis Appendices

Appendix A

Testing and Demonstration Program

This software program was programmed and implemented based on the exclusive testing procedures and requirements.

KOSMO1.FTH

```
DECIMAL
DISABLE
FORGET INIT-KOSMO1

: INIT-KOSMO1
STOP
100 LEFT SPEED
100 RIGHT SPEED
20 TRIGGER-FACTOR
53 SUM-FACTOR
CALIBRATE
26 rLM-TRIGGER C!
26 rRM-TRIGGER C!
PVL rLW-MASK C!
PVR rRW-MASK C!

;
:  D DISABLE ;

:  KOSMO1
INIT-KOSMO1

STOP
DISABLE
BLOW_THE_MAN_DOWN!
ENABLE

rSENSE OFF
LEFT PIVOT
2 90 RAMP-UP
5 SECS
2 RAMP-DOWN
rSENSE ON

rSENSE OFF
RIGHT PIVOT
8 90 RAMP-UP
5 SECS
5 RAMP-DOWN
rSENSE ON

100 RIGHT SPEED
100 LEFT SPEED
60 CIRCLE
20 SECS

rSENSE OFF
30 STEP
FIND-SOUND
PIVOT-SOUND
FORWARD
rSENSE ON
5 SECS
```

```
DISABLE
CLEMENTINE
ENABLE

100 RIGHT SPEED
100 LEFT SPEED
70 CIRCLE
20 SECS

100 RIGHT SPEED
100 LEFT SPEED
FORWARD

SAN-FRANCISCO

INIT-KOSMO1
EXPLORE ;

REMEMBER
AUTO-START:  KOSMO1
```

Appendix B

Hardware, Materials, and Parts List

Hardware, Materials, and Parts List
Motors

- 12-volt DC Gear Reduction Motors_Multiproducts Specialties #EX935

Central Processing Unit Board

- Motorola 68HC11 chip
- Constant float charging battery circuit (13.8 volts)
- Light sensors
- Motor current
- Motor driver
- Prom
- RAM/Battery back-up
- RS-232 remote computer interface
- Sound acquisition
- Sound synthesis

Materials

- 4-inch diameter acrylic tubing for neck
- 6/32-inch screws_jet fasteners
- #6 PEM nuts_jet fasteners
- 18-gauge wire for power distribution
- 22-gauge wire for miscellaneous wiring
- Aluminum sheet metal .063 #5052 for unibody construction
- Miscellaneous in-line signal and power connectors
- Perforated circuit board
- Piano wire feelers
- SN 53 solder

Components

- 2000 MCD Superbright LEDs_Digikey
- 3-inch audio speaker
- 6-inch diameter wheels, 1 1/4-inches wide
- 12-volt 4 AH sealed lead acid battery
- 12-volt 500MA DC wall converter for charger
- 25-foot RS-232 cable (for PC connection required for programming robot)
- DB-25 RS 232 connector
- Electret condenser microphone
- Photodarlington phototransistors_Digikey
- On/Off toggle switch
- Reset switch
- Spherical casters
- Various multicolored LEDs

Appendix C

Tools and Equipment Utilized

36-inch Squaring Shear machine
30-inch Box and Pan Brake machine
Vertical drill press
Hand-held drill motor
Combination square
24-inch and 6-inch scales
Whitney hand-held punch with assorted size punch and dies
Various hand tools (screw drivers, hammer, pliers,
hack saw, and socket and open-end wrenches)
Soldering iron
Digital volt meter
Power driver
12-foot tape measure

Bibliography

Brodie, L. 1984. Thinking FORTH. Englewood Cliffs, NJ: Prentice-Hall.
Brodie, L. 1987. Starting FORTH (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.
Cardoza, A., and Vlk, S. J. 1985. Robotics. Blue Ridge Summit, PA: Tab Books, Inc.
DeMiranda, M. A. 1988. "Design and construction of a performance vehicle." Unpublished master's thesis. Long Beach, CA: California State University, Long Beach.
Demmon, R. H. 1983. "Construction of a vertical wheel mulling machine." Unpublished master's thesis. Long Beach, CA: California State University, Long Beach.
Golding, D. 1992. Whiskers, the intelligent mobile robot. Angelus Oaks, CA: Angelus Research.
Hampton, W. J. December 15, 1988. What is Motorola making at this factory? Business Week, p. 1000.
Heath, L. 1985. Fundamentals of robotics: Theory and applications. Reston, VA: Reston Publishing Company.
Iyengar, S., and Elfes, A. 1991a. Autonomous mobile robots. Volume 2: Perception, mapping, navigation. Los Alamitos, CA: IEEE Computer Society Press.
Iyengar, S., and Elfes, A. 1991b. Control, planning and architecture. Los Alamitos, CA: IEEE Computer Society Press.
Kandel, A., and Langholz, G. 1992. Hybrid architectures for intelligent systems. Boca Raton, FL: CRC Press.
Kelly, D. 1985. A layman's introduction to robotics. Princeton, NJ: Petrocelli Books, Inc.
Langrath, R. June, 1992. Smart shovel. Popular Science, pp. 82-109.
McComb, G. 1987. Robot builders bonanza: 99 inexpensive robotics projects. Blue Ridge Summit, PA: Tab Books.
Minsky, M. 1985. Robotics. Garden City, NY: Omni Publications International, Ltd.
Nof, S. Y. 1985. Handbook of industrial robotics. Toronto, Canada: Wiley and Sons.
Robillard, M. J. 1983. Microprocessor based robotics: Intelligent machine series (Vol. 1). Indianapolis, IN: Howard W. Sams and Company.
Robillard, M. J. 1989. Advanced robot systems. Indianapolis, IN: Howard W. Sams and Company.
Stonecipher, K. 1989. Industrial robotics: Machine vision and artificial intelligence. Carmel, IN: Howard W. Sams & Company.
Weiss, J., Jr. July, 1991. Robodoc. Discover, p. 16.

THE DESIGN, CONSTRUCTION, AND TESTING OF AN AUTONOMOUS ROBOT

A THESIS
Presented to California State University, Long Beach

In Partial Fulfillment
of the Requirements for a Masters Degree

By
George Sergio Vega
Bachelors, 1984, Hawaii Pacific University
August 1993

ACKNOWLEDGMENTS

I dedicate this study in memory of my loving father, George H. Vega, for whom without his direct and indirect teachings, my drive and ambition might not be as great, nor as determined, in moving forward and striving for greater knowledge and excellence.

I express my gratitude and appreciation to Don Golding, President of Angelus Research, for his technical support, advice, and numerous 18-hour+ vigils necessary in the development and creation of my right-hand man: Kosmo, the Robot.

I express my sincere appreciation and gratitude to Dr. Robert Kunst for his guidance and knowledge during the entire process in orchestrating this project.

I express my appreciation to the aforementioned individuals for their support, help, and contribution in the completion of this study and project: Sherry DeAna Skipper, for her grammar and typing abilities; Steve Roach and Mike Palomo, for their technical and artistic talents; and my brother, Rupert Vega, for his expertise in industrial photography. I also extend my gratitude to Mr. Skip, President of Sharecomm Enterprises, for his software support.

I thank my mother, Helen Vega, and two sisters, Josie and Elizabeth, and Takako-san, for their continued support and prayers.

TABLES

FIGURES

ABSTRACT

THE DESIGN, CONSTRUCTION, AND TESTING OF AN AUTONOMOUS ROBOT

By
George Sergio Vega
August 1993

The purpose of this study was to design, construct, and test an autonomous robot assembled from pre-designed components.

The design parameters and key factors of this robot included optical and tactile collision avoidance, total mobility, an independent untethered power source, and programmable system. The design included additional features: sound effects, interactive and instinctive level programming, and a dual motor drive control system. A basic series infra-red collision avoidance system, in conjunction with a tactile collision avoidance system, was developed and constructed. These systems were mounted to an aluminum unibody construction base, body, and head structure. The prototype base and body were

designed and constructed to house and support a dual gear reduction motor drive system attached to two 6-inch diameter wheels. This robot was mobile and maneuverable.

The robot performed well. A few minor mechanical and electrical problems were observed. Appropriate changes and modifications were made, which resolved these problems.

SHORT TITLE:

DESIGN AND CONSTRUCTION OF AN AUTONOMOUS ROBOT

Appendix C

Bibliography

Bibliography
Brodie, L. 1984. Thinking FORTH. Englewood Cliffs, NJ: Prentice-Hall.
Brodie, L. 1987. Starting FORTH (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.
Cardoza, A., and Vlk, S. J. 1985. Robotics. Blue Ridge Summit, PA: Tab Books, Inc.
DeMiranda, M. A. 1988. "Design and construction of a performance vehicle." Unpublished master's thesis. Long Beach, CA: California State University, Long Beach.
Demmon, R. H. 1983. "Construction of a vertical wheel mulling machine." Unpublished master's thesis. Long Beach, CA: California State University, Long Beach.
Golding, D. 1992. Whiskers, the intelligent mobile robot. Angelus Oaks, CA: Angelus Research.
Hampton, W. J. December 15, 1988. What is Motorola making at this factory? Business Week, p. 1000.
Heath, L. 1985. Fundamentals of robotics: Theory and applications. Reston, VA: Reston Publishing Company.
Iyengar, S., and Elfes, A. 1991a. Autonomous mobile robots. Volume 2: Perception, mapping, navigation. Los Alamitos, CA: IEEE Computer Society Press.
Iyengar, S., and Elfes, A. 1991b. Control, planning and architecture. Los Alamitos, CA: IEEE Computer Society Press.
Kandel, A., and Langholz, G. 1992. Hybrid architectures for intelligent systems. Boca Raton, FL: CRC Press.
Kelly, D. 1985. A layman's introduction to robotics. Princeton, NJ: Petrocelli Books, Inc.
Langrath, R. June, 1992. Smart shovel. Popular Science, pp. 82-109.
McComb, G. 1987. Robot builders bonanza: 99 inexpensive robotics projects. Blue Ridge Summit, PA: Tab Books.
Minsky, M. 1985. Robotics. Garden City, NY: Omni Publications International, Ltd.
Nof, S. Y. 1985. Handbook of industrial robotics. Toronto, Canada: Wiley and Sons.
Robillard, M. J. 1983. Microprocessor based robotics: Intelligent machine series (Vol. 1). Indianapolis, IN: Howard W. Sams and Company.
Robillard, M. J. 1989. Advanced robot systems. Indianapolis, IN: Howard W. Sams and Company.
Stonecipher, K. 1989. Industrial robotics: Machine vision and artificial intelligence. Carmel, IN: Howard W. Sams & Company.
Weiss, J., Jr. July, 1991. Robodoc. Discover, p. 16.