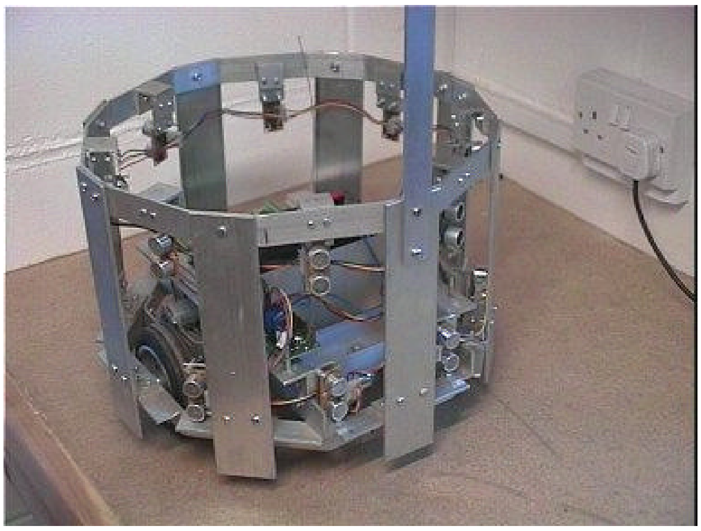
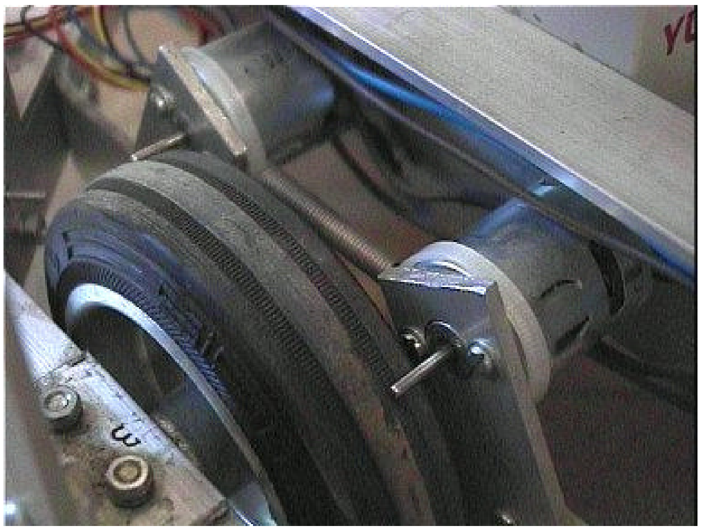
Chucky - putting it all together

Who's Chucky then?
Chucky is our test robot. He grown and evolved over the years as we've hacked him around to test new modules. Chucky is ever changing, and this is a snapshot as he is today (March 2004). The version being described here uses behavior based principles for navigation. This article will describe Chucky, his sensors and the software used for obstacle avoidance.

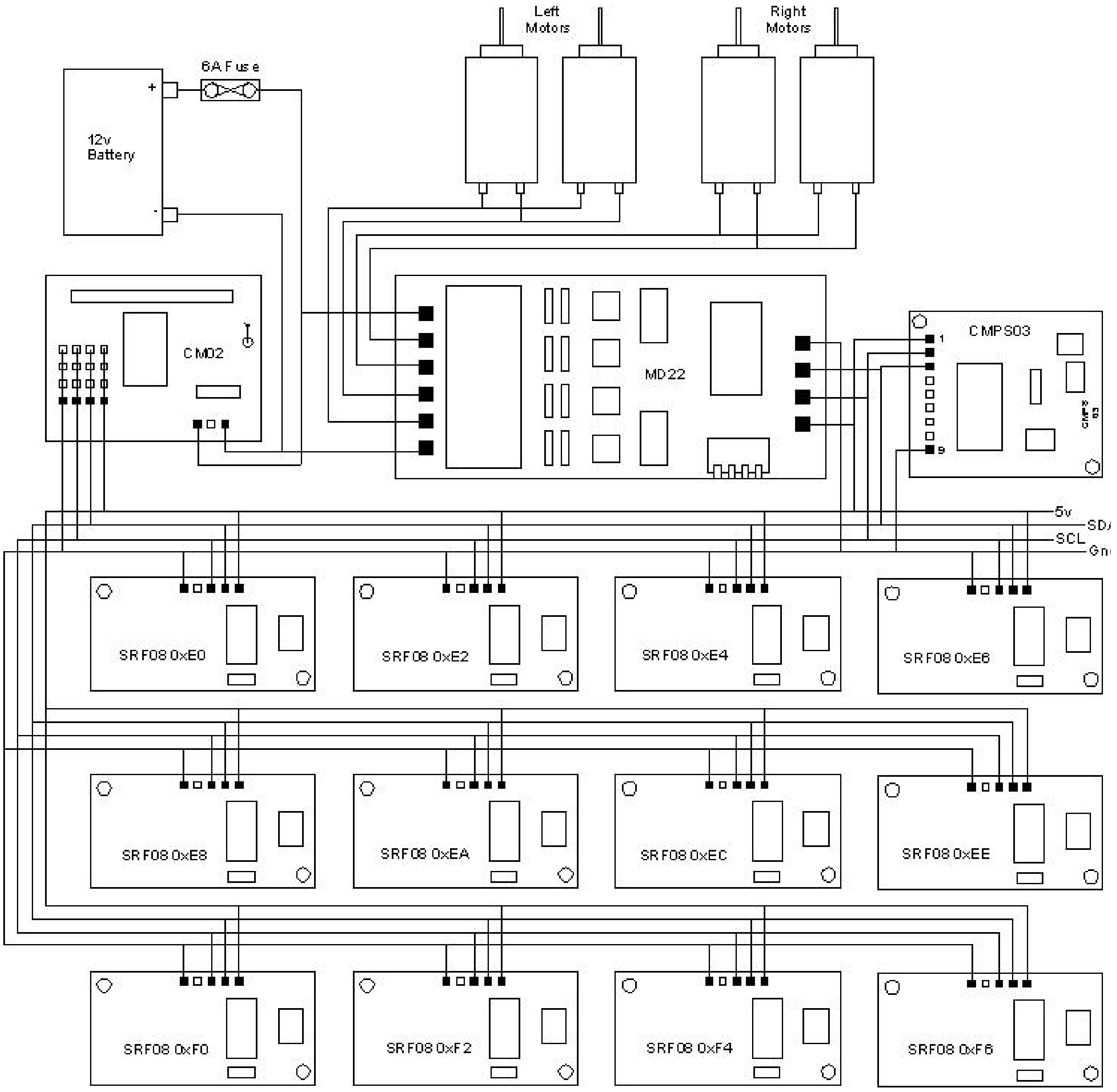


Chucky is made predominantly from aluminum stock. Its strong, light, cheap and easy to cut and drill, which made Chucky quick to build. The wheels are 100mm rubber tyred steel castor types and cost about £3 each. They are driven by a pair of small 12v motors on each wheel. The motors are mounted on loose brackets and pulled together by a spring which forces the motor shafts directly onto the rubber tyre. Its cheap-n-easy with enough grip to propel Chucky around our workshop, and to provide a clutching action if he gets stuck. Its also amazingly quite with no gears to make a noise. There is also a smaller caster wheel front and back for stability.

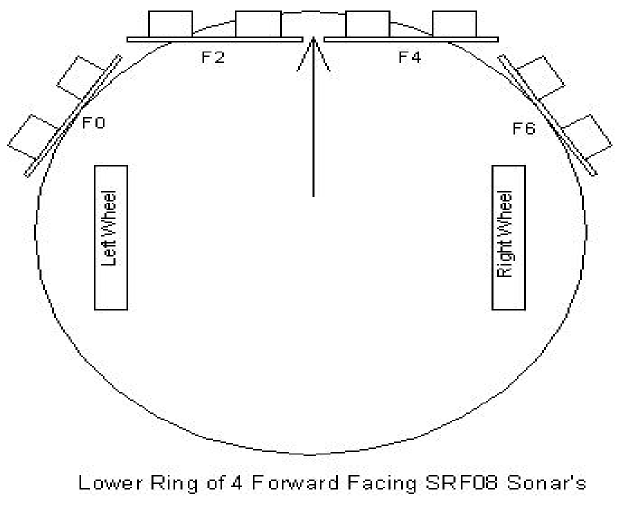


The Electronics
Chucky uses a selection of control modules and sensors. The motors are powered by an MD22 Dual 5amp controller. SRF08's are used as the sensors. There is an upper ring of eight of them for surround detection, and four in a lower forward facing arc. The lower ones are used to detect chair castors and other short objects that it would otherwise collide with. There is a CMPS03 compass module mounted high up and out of site on the above photo. A CM02 radio module provides communication with an RF04 module connected to the PC's USB port. The CM02 also provides the CMPS03/MD22 and SRF08's with their 5v supply from its on-board 5v regulator. Chucky is powered from a 7A/Hr 12v sealed lead acid battery which goes to the CM02 module and also to the MD22 for motor power. All of the modules are connected together with a four wire I2C loop. 5v, 0v, SCL and SDA. The PC can now control Chucky's motors and upload all sensor data from Chucky. The PC itself now becomes the robots brain.

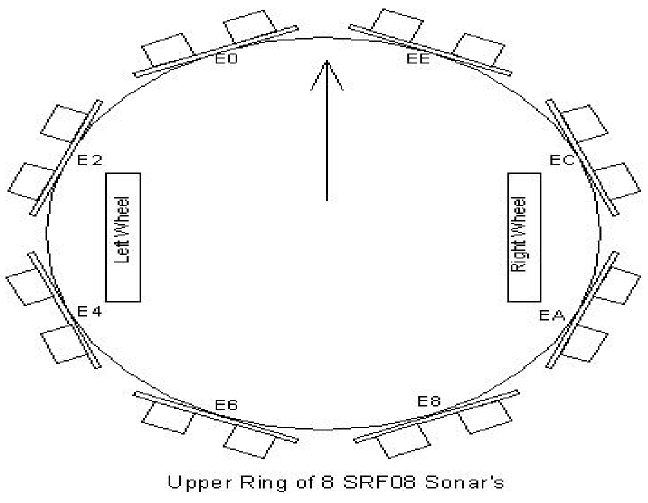
Schematic of Chucky's Electronics



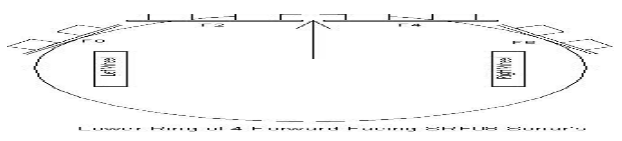
Layout of the sonar's on Chucky's lower and upper rings

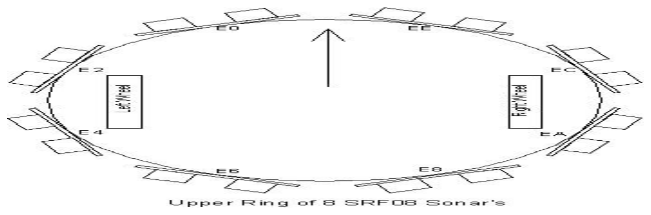


Lower Ring of 4 Forward Facing SRF08 Sonar's

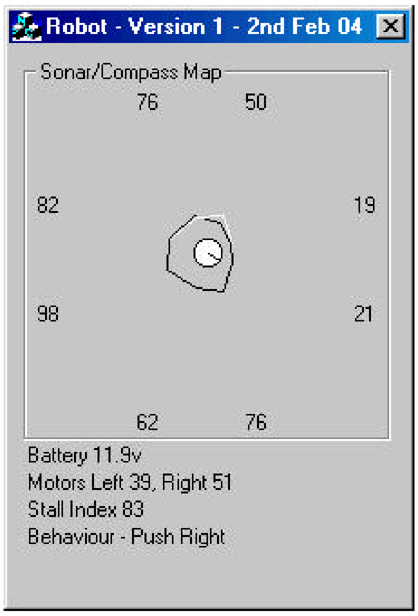


Upper Ring of 8 SRF08 Sonar's





The Software
We have a small windows based PC application which controls Chucky. The robot can successfully navigate around our workshop, following walls and avoiding obstacles. It code is based on a behavior based control system with a simple priority arbitration scheme. The Robot program displays a small white circle to indicate the robot with a line from the center to indicate the compass bearing. Around this is an irregular black polygon which is the range the robot can see to be clear. It comes from the upper ring of eight SRF08's. The range is also displayed numerically. The white arc is from the four lower forward facing SRF08's.



```
Battery 11.9v
Motors Left 39, Right 51
Stall Index 83
Behaviour - Push Right
```

Below the graphic area is a text display showing the battery voltage and the values being sent to the Left and Right motors. Stall index is used to detect that the robot has stopped moving, and is derived from differences in the front and rear sonar's. The behavior is the current winning behavior.

Behaviors
Chucky uses 12 behaviors to navigate, a simple priority order being used for arbitration. A global structure called BD (for Brain Data) is used, amongst other things, to store the motor speeds. These are BD.Left and BD.Right for left and right motors. The lowest level Cruise behavior always sets the motors for driving forwards. Other higher priority behaviors change the motor settings as, and if, they are triggered by sensor readings. When all behaviors have run in order from lowest (Cruise) to highest (Escape), then the resultant motor speeds are sent to the MD22 module.

Cruise. The lowest level behavior is "Cruise". This does not look at any of the sensors, it simply sets the motors going forwards. Although not essential, my Cruise routine does do a couple of other small jobs. Its sets the actual speed of the motors in proportion to the clear space ahead of it (and in that, it does look at the sensors). It also adjusts the Left/Right motor speeds to keep the robot driving in a straight line using the compass bearing (and again, in that it looks at the sensors). The essential job of Cruise though, is to propel the robot forwards, and sensors are not required for that. Cruise is always active.

PullRight. This behavior checks the front right sonar. If it is less than a threshold distance it checks to see if it is closer to an object/wall than the front left sonar. If these conditions are met, then the right motor speed is reduced by 25% from the Left speed. This causes the robot to pull to the right.

PushRight. The opposite of the PullRight behavior above, The threshold, ie. the distance from the wall, is lower and this time its the Left motor that's reduced by 25% to cause the robot to bear to the left (or push from the right, if you prefer). If the robot is too close to a wall on the right, then Cruise will ignore it and try to make the robot go straight on. The robot will not really go straight on though because the higher priority PullRight behavior will see that it is near to a right hand wall and try to move the robot even closer to it. Next, the even higher priority PushRight behavior sees that the robot is actually too close to the wall and overrides what PullRight wants to do and pushes the robot away from the wall. Together PullRight and PushRight produce an emergent wall following behavior.

PullLeft.
PushLeft.
These two behaviors are the complement of the PullRight and PushRight described above.

AvoidLeft.
This behavior is used to make the robot avoid an object close up on the front left. It does this by reversing the right motor direction, which causes the robot to rotate right on the spot.

BearRight.
This is a longer range version of the AvoidLeft behavior described above. Whilst AvoidLeft works at close range, BearRight works at a longer range and provides a gentler movement. Instead of reversing the Right motor as AvoidLeft does, it reduces its forward speed by 25%.

AvoidRight.
BearLeft.
These two behaviors are the complement of the AvoidLeft and BearRight described above.

AntiCanyon.
All of the behaviors described so far are reactive or servo behaviors. This means that they react to the situation as it is from moment to moment. They have no knowledge of what’s gone on before, and that can lead to trouble. Consider what happens when a robot moves towards a corner of the room. At some point it sees the wall on the left and the AvoidLeft behavior kicks in to move the robot to the right. It does this until the wall on the left is out of its threshold range. Far from being free however, the robot has now moved to see the right wall. The AvoidRight routine now moves the robot to the left, and guess what? It sees the left wall again. This will continue forever with the robot shaking from side to side. What is needed is a higher level behavior which detects the cannoning effect and moves to get the robot out of trouble. The difference from the behaviors so far described is that it must continue even after the trigger has ceased, to continue to move the robot out of the canyon. This is a ballistic behavior because once triggered, it continues to completion. Ballistic behaviors are dangerous as you are overriding all of the obstacle avoidance behaviors so far provided. The AntiCanyon behavior only detects the robot is rocking from side to side. It does not itself try to do anything about it other than to trigger the Escape behavior. Escape is the one ballistic behavior in the system and used to get the robot out of trouble.

AntiStall.
This behavior detects a stalled condition. No matter how many sensors a robot has, they always seem to have an uncanny knack of getting themselves stuck somehow. Is that an emergent behavior? Like the Anticanyon behavior above, AntiStall just detects than the robot has stopped moving and triggers the Escape behavior. It doesn’t try to do it itself. Chucky does not have any wheel sensors to detect that he has stalled. Instead the forward and reverse sonar’s are checked for a continuing change the distances. If no change is detected the Escape behavior is triggered. Using sonar data for stall detection mostly works, but it isn’t foolproof. AntiStall generates an index rating based on the changes and triggers when this falls below a preset threshold. The stall index is displayed on screen as Chucky moves around.

Escape.
This is the only ballistic behavior currently installed. It does not activate by itself. Instead, it is triggered by either the AntiStall or AntiCanyon behaviors. Escape works by first backing off slightly, and then trying to move the robot in the direction of the greatest open space. It calculates the direction of the best open space and rotates on the spot to face that direction. If it is already facing the correct direction then the robot moves forwards. This continues until a cycle counter reaches zero, after which Escape returns control to the other behaviors.

Arbitration.
The order that the behaviors - which are really just subroutines in the program - are called defines the priority. I have arranged for the various range thresholds to be passed as parameters to make it easy to adjust the behaviors. Here is the call list:

```
Cruise();
PullRight(60);
PullLeft(60);
PushRight(35);
PushLeft(35);
BearRight(40);
BearLeft(40);
AvoidLeft(25);
AvoidRight(25);
AntiStall();
AntiCanyon();
Escape();
```

These behaviors allow Chucky to wander around our workshop without getting into too much trouble. Its amazing the places he manages to squeeze into when left on his own.
The software is written in Visual C version 6, and the source can be downloaded here.

As always with robotics - Have Fun!
Gerry.