Der Barcodeparser im Hackfisch-Bot

Antity, Gruppe Hackfisch
Johnny

2002-05-21

Zusammenfassung

Eine der Aufgaben innerhalb der Entwicklung eines sich selbst steuernden Robotors innerhalb des Projekts U23 ist es, daß die Maschine einen auf dem Boden eingezeichneten Strichcode erkennen und auf darin kodierte Steuerbefehle reagieren können muß. Das folgende Dokument soll einen kurzen Überblick über die dabei Verwendung findende Routinen und ihre Funktionsweise für den Roboter der Gruppe Hackfisch bieten.

1 Der Barcode-Parser

Zum Einlesen eines Barcodes, über den der Hackfisch fährt, steht ein eigener Codeteil parat: `recog`.

`recog` ist eine Routine, die regelmäßig oder unregelmäßig aufgerufen werden kann und die dann die aktuelle Position und die Werte der Lichtsensoren ausliest. Über die Timestamps, die die Routine dann ausliest, ist sie relativ unabhängig davon, regelmäßig aufgerufen werden zu müssen. Die Lichtsensorwerte werden einfach auf die Zeit seit dem letzten Aufruf skaliert.

Die Routine springt mit dem ersten Aufruf an, bei dem ein Schwarz-Wert (logisch `1`) am Sensor anliegt, und liest von dort an ein, bis eine gewisse Zeit- oder Entfernungsspanne lang kein weiterer Barcode-Balken gefunden wurde.

Die eingelesenen Daten werden in einem Strom abgelegt. Aus diesem ermittelt die Routine bereits während des Laufes die (wahrscheinliche) Länge eines einzelnen Barcode-Balkens im Muster. Dieser Wert wird während des Scannens kontinuierlich angepaßt.

Wird dann eine bestimmte Entfernung lang (definierbar, z.B. fünfmal „minimal mögliche Balkenlänge“ lang) kein weiterer Balken entdeckt, betrachtet die Routine den Barcode als beendet.

Das Ende des Codes begründet sich für die Routine auf einer Regel, nach der nur maximal `X` Balken derselben Farbe (schwarz/weiß; im Moment auf 2 fixiert) direkt aufeinander folgen dürfen. Über diese Konstante wird auch hergeleitet, daß nach 5 Streifen „weiß“ wirklich kein weiteres schwarz mehr auftauchen dürfte.

Auf diese Weise kann der Parser auch parallel zu vorhandenem Steuerungscode laufen und muß nur ab und an aufgerufen werden, um einen eventuell vorhandenen Code zu lesen.

Die Information, ob gerade ein Code unter dem Bot besteht, könnte noch nach außen geführt werden, so daß die restliche Kontrolllogik darauf reagieren kann und den Auslese-Code öfter aufruft.

Es sollten sowenig Zyklen wie möglich mit unnötigem Scannen verbraten werden. Daher könnte der erste Trigger mit dem ersten

schwarzen Balken auch von der Steuerungssoftware ausgehen, die Schwarz gesehen hat.

2 Interface

Das Interface von `recog` nach außen besteht aus Sicht des übrigen Codes momentan nur aus zwei Funktionen: `initscan()` und `upstat()`. Durch die erste wird der Code mit Startwerten initialisiert, während `upstat()` jederzeit wieder aufgerufen werden kann, sich einen Überblick über die aktuelle Situation und die Statusänderungen seit dem letzten Aufruf verschafft und entsprechende Daten anlegt.

Der Code beendet sich über `lexit()` später selbst, sobald das Ende des Barcodes erkannt wurde und kann die eingelesenen und verarbeiteten Daten weitergeben.

2.1 **void** `initscan`**(void)**

[public] Muß einmalig vor Benutzung des Codes aufgerufen werden. `initscan()` setzt einige interne Variablen zurück (die aus Gründen der Zugriffsgeschwindigkeit für den Embedded-Code als `static` ausgelegt sind) und liest ein erstes Mal die anfänglichen Werte der verschiedenen Sensoren ein.

2.2 **timeiden_t** `currenttime`**(void)**

[private] Liefert einen Zeitindex zurück. Das ist vorzugsweise ein wirklicher Index über eine Echtzeituhr, sodaß der Code eine ungefähre Vorstellung davon bekommt, wieviel Zeit nun wirklich zwischen den einzelnen Calls vergangen ist.

Alternativ kann auch, bei hinreichend genauen Bewegungen, der Impuls aus den Bewegungssensoren eingefüttert werden.

Im Debug-Modus wird eine Zeiteinheit von `1` pro Aufruf (pro Wegeinheit im `stdin`) angenommen.

2.3 **sensorda_t** `sensorin`**(void)**

[private] Liefert `0` oder $! = 0$ zurück, je nachdem, ob der Sensor sich gerade über einem dunklen oder hellen Untergrund befand.

Im Debuggingmodus erzeugt diese Routine Sensorinformationen aus dem Characterstream von `stdin`.

2.4 **spee_t** `getspeed`**(void)**

[private] [unimplemented] Kann die aktuelle Geschwindigkeit des Robotors aufnehmen und mit Hilfe des Zeitindexes von `currenttime()` dem Rest des Codes dann dabei helfen, eine Vorstellung der aktuellen Position zu bekommen, an der die Sensordaten eingelesen wurden.

Im Test- und Debuggingmodus wird ein konstanter Speed von `1` angenommen.

2.5 **void** `upstat`**(void)**

[public] Die Hauptroutine, die von außerhalb, wann immer möglich, aufgerufen wird. Sie macht sich dann einen Überblick über die Umweltdaten mit Hilfe der drei zuvor aufgeführten Funktionen, liest sie ein und schreibt danach die Sensordaten in einen internen Puffer.

Schon während des Lesevorgangs wird versucht, den Code auf die Länge von Balken zu analysieren, sodaß dem abschließenden Check in `lexit()` weniger Arbeit entsteht.

Die Routine zählt auch nicht jeden einzelnen Aufruf oder Impuls, sondern reagiert erst auf eine Statusänderung am Sensor. Dann vergleicht sie es mit dem Zeit- und Wegindex des letzten Scans und baut danach die vorläufige Liste von Scanereignissen auf. Oder halt eben auch nicht.

Erkennt diese Routine das Ende eines möglichen Barcodes, ruft sie `lexit()` auf, um externen Code zu informieren.

2.6 void `lexit(void)`

[private] Eine eigentlich interne Routine des Codes. Sie wird aufgerufen, wenn `upstat()` meint, das Ende eines Barcode-Streifens entdeckt zu haben.

`lexit()` benutzt die bisher gesammelten Informationen, um die Kette von Ein-Aus-Informationen nocheinmal neu zu skalieren und den inzwischen gefundenen Impulsen (eventuell kürzere Streifen) anzupassen. Daraufhin wird der Code zur weiteren Verwendung ausgegeben.

3 Konstanten

3.1 `MAXSAMESTATI`

Maximale Anzahl von weißen oder schwarzen Balken, die nach Barcode-Definition nacheinander auftauchen dürfen. Der Rest des Codes richtet sich nach dieser Spezifikation.

Default: 3.

3.2 `ENDISFACTOR`

Anzahl von weißen Balken, nach der das Ende des Codes angenommen werden soll.

Default: `MAXSAMESTATI + 3`

3.3 `MAXPATTERNLEN`

Maximale Anzahl von Bars im Barcode. Im Moment für die Testphase:

Default: `2000`

4 Debugging

Debugging ist standardmäßig an. Verschiedene Ausgaben von `<assert.h>` sowie des Parser-Codes lassen sich zur Kompilationszeit mit `-DNDEBUG` ausschalten.

Gerade die Beispielimplementation `recog.c` gibt bei erweitertem Debugging eine Unmenge an weiteren Informationen aus, die den fortschreitenden Vorgang der Erkennung verfolgen lassen.

`recog.c` enthält – neben der eigentlichen Parser-Implementation – noch genügend Code, um die Ein- und Ausgaben des wirklichen Bots zu simulieren.

So läßt sich die Routine mit Zeichenfolgen über die Standardeingabe füttern (bestehend aus weiß (`'-'`) und schwarz (`'#'`), die wie Events von den Lichtsensoren behandelt werden.

5 Tools

5.1 genbars

`genbars` wird benutzt, um für das kompilierte Demo-`recog` einfach Eingaben zu generieren.

`genbars`, gestartet, nimmt Eingaben in der Syntax (Multiplikator)(Zeichen) entgegen, in beliebiger Länge, und spuckt entsprechende Dateien für `recog` aus.

So ergibt sich beispielsweise aus der Eingabe: `8#3-4#6-` die Ausgabe: `########---####------`, die dann in `recog` gefüttert werden kann.

6 Nachtrag

Gesetzt mit LaTeX2e und Hyperref auf dem Laptop.