Une alimentation numérique CC – 2ème partie : le logiciel

par Guido Socher (homepage)

L´auteur:

Guido adore Linux parce que c'est vraiment un bon système pour développer son propre matériel.

Traduit en Français par:
Jean–Etienne Poirrier (homepage)

Résumé:

Ceci est la seconde partie de la série parlant d'une alimentation numérique. Vous pourriez vouloir lire la première partie d'abord.

Il y aura une troisième partie où j'ajouterai la communication i2c pour contrôler l'alimentation via une commande provenant du PC et peut–être une quatrième partie où des choses plus amusantes seront ajoutées. Je pense non seulement à produire du courant continu (DC) mais aussi du DV + pulsations et crêtes. De cette manière, vous pouvez tester des circuits pour être sûr qu'ils sont résistants au bruits et variations de courant.

Un kit avec la carte mère et les pièces de cet article sont disponibles sur shop.tuxgraphics.org.

Introduction

En utilisant un design intelligent basé sur un micro–contrôleur, nous pouvons construire une alimentation électrique qui possède plus de possibilités et qui revient beaucoup moins cher que les alimentations électriques traditionnelles. Cela est possible parce que les fonctions qui sont traditionnellement implémentées dans le matériel, sont déplacées dans le logiciel.

Dans cet article, nous ferons deux choses :

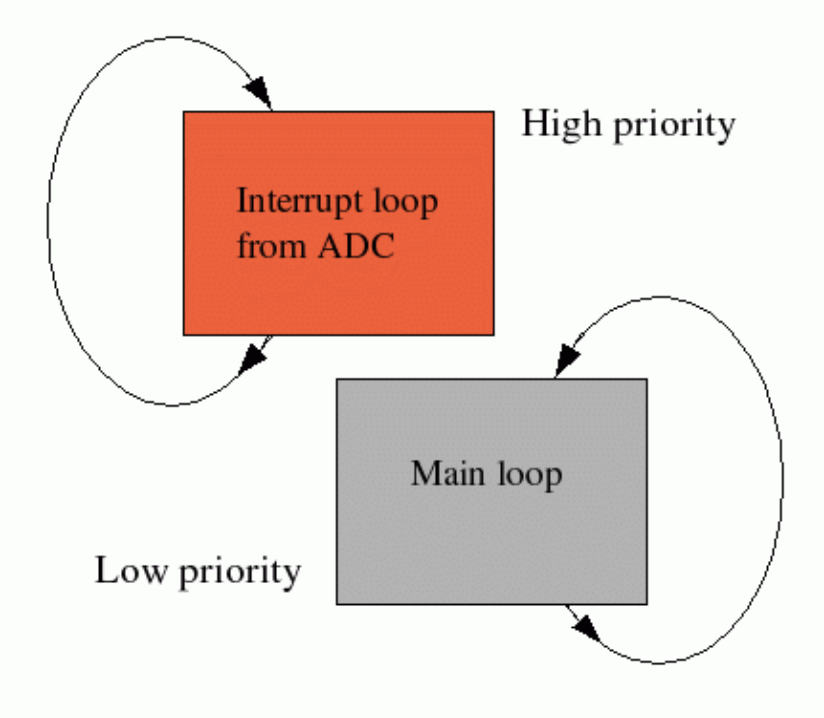
- Je vais expliquer comment fonctionnent les différentes parties du logiciel
- Ajouter du code pour stocker les paramètres de manière permanente.

Un mot d'avertissement

Cet article va vous donner des éléments sur la manière dont le logiciel fonctionne et vous pouvez utiliser ces connaissances pour effectuer des modifications. Cependant, faites attention au fait que la protection contre les court−circuits est également logicielle. Si, d'une manière ou d'une autre, vous faites une erreur, cette protection peut alors ne plus fonctionner. Si vous provoquez alors un court−circuit sur la sortie de votre matériel, ce dernier peut s'éteindre dans un nuage de fumée. Pour éviter cela, vous pourriez utiliser une grande résistance (par exemple une ampoule de phare avant de voiture) qui va prendre assez de courant pour déclencher la protection (par exemple 6A) mais pas assez pour détruire le matériel. De cette manière, vous pouvez tester un court−circuit sans danger de perdre le matériel.

La structure du logiciel

Lorsque vous regarder le programme principal (fichier ddcp.c, à télécharger à la fin de cet article), vous verrez qu'il y a seulement quelques lignes de code d'initialisation exécuté lors de la mise sous tension et qu'après, le logiciel entre dans une boucle infinie.
Il y a en fait 2 boucles infinies dans ce logiciel. La première est la boucle principale ("while(1){...}" dans le fichier ddcp.c) et l'autre est l'interruption périodique du convertisseur analogique−digital (fonction "SIGNAL(SIG_ADC){...}" dans le fichier analog.c). Durant l'initialisation, l'interruption est configurée pour s'exécuter toutes les 100μs. Toutes les fonctions et le code qui sont exécutés tournent dans le contexte d'une de ces tâches (tâche : nom d'un processus ou d'un thread d'exécution dans un système d'exploitation temps−réel, c'est pourquoi j'utilise ce mot ici, même s'il n'y a pas d'OS).



La tâche d'interruption peut arrêter l'exécution de la boucle principale à tout moment. Elle s'exécutera alors sans être interrompue et, ensuite, l'exécution continuera de nouveau dans la boucle principale, à l'endroit où elle a été interrompue. Cela a deux conséquences :

1. Le code dans l'interruption ne doit pas être trop long car il doit être fini avant que l'interruption suivante n'arrive. Ce qui compte, ici, est la quantité d'instructions en code machine. Une formule

mathématique, qui peut être écrite en juste quelques lignes de code C, peut résulter en des centaines de lignes de code machine.

2. Les variables que vous partagez entre le code d'interruption et le code de la tâche principale peut soudainement changer au milieu de l'exécution. Ceci est aussi valide lorsque vous donnez plus d'un octet de données de l'interruption à la tâche principale. La copie de deux octets va requérir plus d'une instruction. A ce moment, il se peut que le premier octet soit copié avant l'interruption alors que le second octet soit copié après cette interruption. Que faire ? Dans la plupart des cas, ce n'est pas un problème car les résultats mesurés de l'ADC ne diffèreront pas trop entre les deux interruptions. Dans les cas où nous ne pouvons pas nous permettre ce type d'erreur occasionnelle (elle peut se produire seulement une fois toute les heures), vous devez utiliser un drapeau (flag) que vous pouvez consulter pour voir si votre code a été interrompu durant la copie.

Tout ceci signifie que les choses complexes comme mettre à jour l'affichage, vérifier les boutons poussoirs, la conversion des valeurs de courant et de tension en unités internes, etc. doivent être réalisées dans la tâche principale. Dans l'interruption, nous exécutons seulement les choses qui sont critiques dans le temps : contrôle du courant et de la tension, protection contre la surcharge et paramètres du DAC. Pour éviter des mathématiques complexes, tous les calculs dans l'interruption sont réalisés dans des unités d'ADC. Ce sont les mêmes unités que les procédures ADC (valeurs entières de 0 à 1023).

Voici le flux logique exact des opérations que nous faisons dans la tâche principale :

```
1) Copier les derniers résultats ADC à partir de la tâche
   d'interruption
2) Les convertir en valeurs d'affichage (ampères et volts)
3) Convertir les ampères et volts voulus (ce que l'utilisateur a
   défini) en des valeurs ADC internes équivalentes
4) Copier les valeurs ADC équivalentes dans des variables de
   manière à ce que la tâche d'interruption puisse
   l'utiliser.
5) Vider l'écran LCD.
6) Convertir les nombres que nous voulons afficher sur le LCD en chaînes.
7) Ecrire les valeurs de tension sur l'afficheur.
8) Vérifier si la tâche d'interruption est en train de
   réguler la tension ou le courant (limitation active du courant).
9) Si la tension est le facteur limitant, alors écrire une flèche
   derrière la tension sur l'afficheur.
10) Ecrire les valeurs d'ampère sur l'afficheur.
11) Vérifier si la tâche d'interruption est en train de
    réguler la tension ou le courant (limitation active du courant).
12) Si le courant est le facteur limitant, alors écrire une flèche
    derrière le courant sur l'afficheur.
13) Vérifier si un bouton a été pressé. Si ce n'est
    pas le cas, attendre 100ms et vérifier de nouveau. Si un bouton a
    été pressé, alors attendre 200ms. Cela permet d'avoir
    une bonne réponse des boutons et un défilement pas trop rapide
    s'ils sont appuyés en permanence.
14) Retour à l'étape 1).
```

La tâche d'interruption est beaucoup plus simple :

```
1) Copier les résultats de l'ADC dans des variables
2) Basculer le canal de mesure de l'ADC entre le courant et la tension
3) Vérifier si un excès de courant est mesuré. Si c'est le
   cas, définir immédiatement une valeur basse au DAC (pas besoin
   que ce soit zéro puisque le circuit d'amplification de la tension
   travaille seulement à partir de 0.6V (0.6V en entrée produit
   toujours 0V en sortie)).
4) Vérifier si la tension ou le courant a besoin d'être
   régulé
```

```
5) Vérifier si le DAC (convertisseur analogique-digital) a besoin de se
   mettre à jour en fonction de la décision du point 4).
```

C'est l'idée de base du logiciel. Je vais aussi expliquer ce que vous trouverez, dans quel fichier et, ensuite, vous devriez être capable de comprendre le code (étant supposé que vous êtes familier avec le C).

Quel fichier contient quoi

```
ddcp.c -- ce fichier contient le programme principal. Toute l'initialisation est
          réalisée ici. La boucle principale est également
          implémentée ici.

analog.c -- le convertisseur analogique vers digital et tout ce qui tourne dans
          le contexte de la tâche d'interruption qui se trouve ici.

dac.c -- le convertisseur digital vers analogique. Initialisé par ddcp.c
          mais seulement utilisé à partir de analog.c

kbd.c -- le code pour le clavier

lcd.c -- le pilote LCD. C'est une version spéciale qui ne
          nécessitera pas le pin rw de l'afficheur. Il utiliser, à
          la place, une horloge interne qui devrait être suffisamment
          longue pour que l'afficheur finisse sa tâche.
```

Nouvelle fonctionnalité : sauver les paramètres

La nouvelle fonctionnalité que nous ajoutons dans cet article n'est pas grand chose puisque j'ai déjà passé une partie de cet article à expliquer comment le logiciel fonctionne et je ne veux pas rendre cet article trop long.

Cependant, la fonction que nous ajoutons maintenant est essentielle : sauver les paramètres de manière à ce que la tension et le courant ne doivent pas être réglés de nouveau au prochain allumage. Nous sauvons ces valeurs dans l'eeprom du micro–contrôlleur. Toutes les eeproms (même les clés usb) ont des limites comme le nombre de fois qu'une cellule d'eeprom peut être écrite. Pour le Atmega 8, la limite est de 100000 fois. Après cela, l'eeprom n'est plus garantie et pourrait ne plus sauver de valeur. Une astuce pour obtenir une plus longue durée de vie est d'écrire sur plusieurs cellules mais calculons d'abord ce que cela signifie pour nous. 100000 cycles d'écriture correspondent à 10 fois la sauvegarde de nouveaux paramètres par jour, pendant 25 ans. C'est plus qu'assez. Nous pouvons dès lors juste utiliser la solution la plus simple et sauver dans une adresse eeprom.

Donc, comment faire pour sauver/lire quelque chose sur/dans l'eeprom ? Il y a deux instructions pour écrire ou lire des entiers de 16 bits dans l'eeprom : eeprom_read_word et eeprom_write_word. Les adresse d'eeprom partent de zéro et comptent en octets.

Une complication est que l'eeprom est effacée lorsque nous chargeons un nouveau programme. Ainsi, nous devons être capable de savoir si nous avons lu des aberrations de l'eeprom (parce que le logiciel a été flashé) ou si nous avons des valeurs valides de courant et de tension. Nous faisons cela en écrivant un nombre magique dans l'eeprom. En d'autres mots, nous sauvons chaque fois trois choses : la limite de courant, la limite de tension et le nombre magique. Si nous lisons après la mise en marche, nous vérifions d'abord le

nombre magique. Si c'est notre nombre magique, alors les valeur de courant et de tension sont correctes. Le nombre magique peut être n'importe quel nombre qui n'a que peu de chance d'être là par défaut (par exemple, 19).

Pour voir le code exact, regardez la fonction "store_permanent()" dans le fichier ddcp.c (à télécharger à la fin de cet article).

Le logiciel pour cet article est digitaldcpower−0.3.X où X est la révision que j'envisage d'effectuer s'il y a des mises à jour nécessaires (le logiciel pour l'article précédent était digitaldcpower−0.2.X).

Amusez−vous ! Le prochain article ajoutera une communication I2C à l'alimentation électrique, à partir du PC. Ainsi, vous pourrez non seulement appuyer un bouton sur l'alimentation électrique pour changer quelque chose mais aussi le faire via une commande.

Je suis à la recherche de personnes qui peuvent porter les programmes hôtes I2C vers différents systèmes d'exploitation. Faites moi savoir si vous pouvez aider pour cela. Vous devez avoir quelques connaissances de l'interface RS232 et un compilateur. Le changement actuel affecte probablement seulement une ligne de code (la fonction ioctl).

Le circuit entier avec toutes les pièces et une carte imprimée est disponible sur shop.tuxgraphics.org (voir ci−dessous).

Références / Téléchargements

- Page de téléchargement pour cet article (les mises à jour et les corrections seront aussi disponibles là).
- La première partie de cette série : Une alimentation numérique CC
- Comment programmer un atmega8 avec gcc : article de novembre 2004 (352)
- Section électronique de Tuxgraphics, une collection de tous les articles de cette série.
- shop.tuxgraphics.org , section microcontrôleur, vous pouvez y commander toutes les pièces (transistors, composants passifs, afficheur LCD, PCB, microcontrôleur, ...).

2005−09−27, generated by lfparser_pdf version 2.51