Instalar e caracterizar um RTOS em Arduino

Ana Luísa Cardoso Macedo
MIEEC - FEUP
up201504274
dedicação - 33,3%

Eduardo Monteiro Costa
MIEEC - FEUP
up201505780
dedicação - 33,3%

Irene Garcia do Amaral
MIEEC - FEUP
up20150664
dedicação - 33,3%

***Resumo* – Projeto desenvolvido no âmbito da unidade curricular de Sistemas Embarcados, com o objetivo de analisar sistemas operativos em tempo real.**

Palavras-chave – RTOS, Arduino, FreeRTOS.

I. Introdução

O projeto desenvolvido tem como objetivo instalar e caracterizar um sistema operativo em tempo real em Arduino.

De forma a entender o funcionamento do RTOS, efetuamos um conjunto de cinco testes: uso da memória e o seu overhead, comportamento em condições de overload, tempo de switch entre tarefas, jitter causado por overhead do sistema operativo e, por último, testes de hartstone.

II. Arquitectura

A nível de hardware utilizamos o Arduino Uno, que tem as seguintes características:

- Memória Flash: 32 KB
- SRAM: 2 KB
- Velocidade de clock: 16MHz

O kernel utilizado foi o FreeRTOS que, por ser desenhado para executar em microcontroladores, permite a sua instalação no Arduino Uno. Oferece também funcionalidades de comunicação entre tarefas e mecanismos de timing e sincronização.

III. Especificações e Abordagem do Problema

A. *Context Switching*

A mudança de contexto envolve todos os procedimentos que o CPU tem de executar para poder saltar de uma tarefa para outra, garantindo que não há conflitos entre as duas.

Uma mudança rápida e eficiente permite que um só CPU opere facilmente sobre várias tarefas, sendo assim um requisito para qualquer sistema de tempo real com multitasking.

Para obtermos os tempos de context switching, foi desenvolvido um programa que, através de timestamps, guardava e computava a diferença entre os tempos à saída de uma task e à entrada da seguinte. Com isto, obtivemos uma extensa lista com estes tempos. A média dos mesmos é de 49,81µs, o valor mínimo de 48 µs, e o máximo de 60 µs.

Assim, concluímos que quando instalado num Arduino Uno, o FreeRTOS apresenta tempos de context switching suficientemente pequenos e estáveis para uma confortável execução de programas com multitasking, desde que não sejam demasiado exigentes do ponto de vista temporal.

B. *Jitter Caused by OS Overhead*

Quando se executa um determinado task set, é complicado calcular exatamente quando é que este vai terminar de o executar. E o mesmo task set, ao correr várias vezes, pode, e provavelmente vai apresentar tempos de execução diferentes. A esta incerteza temporal podemos chamar jitter.

Quando se projeta um programa, é bom ter noção dos valores que o jitter pode tomar, de modo a não correr o risco de depender desse tempo para a sua correta execução.

Para obter uma representação destes tempos, corremos o mesmo task set 50 vezes e comparámos o tempo que demorava a finalizar o mesmo. Obtendo os seguintes valores:

TABELA I. Considerações Estatísticas Sobre O Jitter

Medidas	**Valores**
Média tempos de execução	9012649 µs
Amplitude tempos de execução	1076 µs
Média Jitter	102,2857 µs
Mínimo Jitter	0 µs
Máximo Jitter	280 µs

Com isto podemos constatar que a não ser que estejamos muito perto do limite de utilização do CPU, o jitter não se deverá revelar um grande obstáculo ao correto funcionamento do programa. Um valor médio de pouco mais de 102 µs e uma amplitude de 1076 µs, serão em grande parte dos casos, irrisórios.

C. *Memory use and Overhead*

O espaço ocupado por variáveis globais e estáticas o compilador sabe à partida porque esse espaço é definido ao escrever o código. Já o espaço que é usado pela stack e pela heap só é requisitado dinamicamente em *runtime*, e por essa mesma razão o compilador não pode prever o espaço que ocuparão, pelo que terá de ser estimado pelo programador.

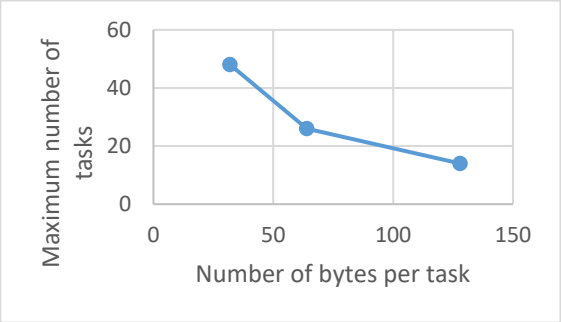
Dado o grande indeterminismo aportado pelo uso da heap, geralmente opta-se por não se usar em sistemas embarcados. Contudo não podemos abdicar do uso da stack pelo que teremos de estimar o valor do espaço ocupado pela mesma. Quando no decorrer do nosso código chamamos uma função, é criado espaço na stack para os parâmetros, variáveis dinâmicas e para o endereço e valor de retorno da função. Este valor é fácil de estimar quando são usadas tasks, dado que é o próprio user que define o espaço em bytes que estas ocuparão quando as cria. Assim sendo testamos criar várias tasks, com distintos valores para o espaço em bytes de forma a verificar o espaço total que o nosso código ocuparia e o máximo de tarefas que o Arduino Uno suportaria.

TABELA II. Resultados dos cálculos de memória

Number of bytes per task	Maximum number of tasks	Memory used*
32	48	2035
64	26	2009
128	14	2047

* in bytes, using the maximum number of tasks in the same code

GRÁFICO 1. Relação entre o número de bytes por task e o número máximo de tasks no mesmo programa suportado pelo Arduino Uno



D. *Behaviour under Overload Conditions*

Considera-se que um sistema está em overload quando uma task usa mais CPU do que aquele que é planeado para essa task. Criamos um programa composto por três tasks com a mesma função, mas diferentes prioridades. Começamos por verificar se era respeitada a prioridade usando o mesmo período de 100ms para as três tasks. De seguida realizamos quatro testes distintos. No primeiro reduzimos o período da task de maior prioridade para 1ms, neste caso só corre essa task até, eventualmente, falhar a deadline. No segundo teste reduzimos o período da task com prioridade média, mantendo o período inicial das outras, aqui a task de prioridade média corre até falhar a deadline. No terceiro teste, reduzimos o período da task de menor prioridade. Começa por correr a task de menor prioridade até falhar. No último teste reduzimos o período das três tasks para 1ms, obtendo o mesmo resultado obtido no primeiro teste.

Analisando os resultados obtidos podemos verificar que em overload, quando todas as tasks têm o mesmo período, sai privilegiada a task de maior prioridade, já quando os períodos são distintos dá-se prioridade à task de menor período.

E. *Hartstone*

Das 5 séries de testes que este benchmark possui, optamos pela PH (Periodic Tasks, Harmonic Frequencies) e pela PN (Periodic Tasks, Nonharmonic Frequencies), fez-se uma versão simplificada do script de testes, de modo a aplicar à arquitetura em análise. Cada série tem 4 tipos de testes, no entanto, apenas foram realizados os primeiros 3. O Arduino Uno, estando limitado a 2 kB de SRAM, não permite a criação de muitas tasks complexas, o que derrota o propósito deste teste, uma vez que é nisso mesmo que se baseia.

Para a realização dos testes da série PH, começamos com o seguinte Task Set.

TABELA III. TASK SET INICIAL PH

Task	P1	P2	P3	P4	P5
Frequency (Hz)	0,5	1	2	4	8
Workload (kiloWhetstone)	16	8	4	2	1

O workload é descrito em kilo whetstone, que por sua vez representa mil operações do tipo floating pointer, de notar que neste task set, o workload rate se mantém constante para todas as tasks.

O primeiro teste consiste em aumentar a frequência da tarefa com frequência mais curta no valor da segunda mais curta, até que a execução falhe.

O segundo teste consiste em adicionar sucessivamente, um décimo da frequência original a todas as tasks, até que a execução falhe.

O terceiro teste consiste em multiplicar a workload por 2, 3, 4 e por daí em diante, até que a execução falhe.

TABELA IV. FALHAS TESTES PH

Task	P1	P2	P3	P4	P5
Freq test 1 (Hz)	0,5	1	2	4	16
Freq test 2 (Hz)	0,8	1,6	3,2	6,4	12,8
WL test 3 (kWst)	48	24	12	6	3

Para a série PN workload foi calculado de forma a manter uma workload aproximadamente constante para cada task. O método utilizado para a realização dos três testes foi o mesmo que para a série PH.

TABELA V. TASK SET INICIAL PN

Task	P1	P2	P3	P4	P5
Frequency (Hz)	0,5	0,833(3)	1,166(6)	2,833(3)	5,166(6)
Workload (kWst)	16	9,6	6,9	2,8	1,5

TABELA VI. FALHAS TESTES PN

Task	P1	P2	P3	P4	P5
Freq test 1 (Hz)	0,5	0,833(3)	1,166(6)	2,833(3)	8
Freq test 2 (Hz)	0,55	0,91(6)	1,28(3)	3,11(6)	5,68(3)
WL test 3 (kWst)	32	20	14	6	3

Tendo realizado estas duas séries podemos retirar daqui algumas conclusões práticas mais significativas. Através dos testes PH, verificamos que o sistema tem um comportamento bastante robusto ao incremento gradual de workload rate e de scheduling stress (teste 2), mas não tanto quanto se começa a exigir mais paralel Computing (teste1). Comparando as duas séries verificamos que existe perda de performance (apesar de não ser significativa) quando a frequência das tarefas não é harmónica.

REFERENCES

[1] Free RTOS™ Real- Time Operating System for Microcontrollers https://www.freertos.org/

[2] Nelson H. Weidermaan and Nick I. Kamenoff, Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time System, The Journal of Real-Time Systems, 4,1992