GENOS-OS: Um sistema operacional base para a construção de sistemas embarcados

Filipe Renaldi, Antonio Carlos Tavares, Mauro Marcelo Mattos

Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil
{filipe,tavares,mattos}@inf.furb.br

Resumo. *Este trabalho apresenta a especificação e implementação de um ambiente de desenvolvimento para sistemas embarcados baseados no processador ARM7. O sistema engloba um sistema operacional embarcado, um método de desenvolvimento utilizando componentes além do próprio software do ambiente de desenvolvimento. O sistema operacional é baseado no núcleo do FreeRTOS que oferece uma estrutura base onde o desenvolvedor o expande conforme suas necessidades através de componentes reutilizáveis. O resultado é um ambiente que provê várias facilidades de construção de software de sistemas embarcados como gerenciamento de projeto e compilação.*

1. Introdução

Um Sistema Embarcado (SE) é um computador de propósito específico encapsulado no dispositivo que ele controla, geralmente projetado para fins específicos, com dimensões reduzidas, com recursos (processamento, memória, barramentos, periféricos) limitados, executa tarefas pré-definidas e está presentes nos mais diversos objetos do cotidiano como telefones celulares, televisões, automóveis, brinquedos e tantos outros equipamentos de consumo.

Os avanços na área de microeletrônica permitiram a integração de um grande número de componentes em um único chip. Os System-on-a-chip (SoC), como são conhecidos, têm em seu encapsulamento um computador completo: processador, memória e periféricos. Um exemplo de SoC é o LPC2106 (produzido pela Philips) que conta com um processador ARM7TDMI-S e diversos dispositivos como memória RAM, memória flash, várias interfaces seriais, relógio de tempo real, temporizadores, interface JTAG, entre outros (PHILIPS, 2003).

A especificidade de cada projeto exige um alto nível de modularidade dos componentes de sistemas operacionais para que sejam adaptáveis as características acima apresentadas. Assim, a utilização de diferentes sistemas operacionais embarcados para diferentes projetos pode trazer um atraso de desenvolvimento pela curva de aprendizagem do programador. Além disso, as pressões de mercado demandando reduzido *time-to-market*, fazem do software (sistema operacional e aplicação) uma parte sensível no projeto de sistemas embarcados.

Este trabalho descreve uma ferramenta voltada para o desenvolvimento de software de sistemas embarcados utilizando componentes, de forma a prover eficiência na reutilização de código uma vez que o ambiente pode ser expandido com os componentes do próprio usuário. Assim sendo, o objetivo deste trabalho foi o de criar

uma plataforma de desenvolvimento de software para sistemas embarcados com base em um sistema operacional embarcado modular construído com a agregação de componentes.

O trabalho está organizado em cinco seções. A primeira seção provê a introdução do trabalho apresentando ainda o escopo e objetivos do mesmo. Na seção dois, está a fundamentação teórica, onde são apresentados conceitos relacionados ao desenvolvimento de software baseado em componentes. A seção três apresenta alguns trabalhos correlatos. A seção quatro descreve a arquitetura do sistema e a seção cinco descreve o processo de normalização de interfaces e apresenta um estudo de caso. Na última seção, são apresentadas as considerações finais.

2 Desenvolvimento baseado em componentes

Um componente é definido como uma unidade de software que encapsula em si o projeto e implementação, oferecendo-o através de interfaces. A motivação no uso de componentes relaciona reutilização com enfoque em tempo de desenvolvimento e consistência no código, uma vez que se trabalha sempre na melhor solução encontrada.

Nos componentes, as interfaces são pontos de interação com o sistema e outros componentes. Um componente pode ter “interfaces oferecidas" (ou fornecidas), onde os serviços de um componente são acessados, e “interfaces requeridas" (ou dependentes), onde são conectados outros componentes necessários. Quanto às características, os componentes devem incluir os seguintes requisitos (GIMENES e HUZITA; 2005): (a) as interfaces fornecidas de um componente devem ser identificadas e definidas separadamente; (b) as interfaces requeridas também devem ser definidas explicitamente. Essas interfaces definem os serviços necessários de outros componentes, para que um componente possa completar seus próprios serviços; (c) o componente deve interagir com outros componentes apenas através de suas interfaces. Um componente deve garantir o encapsulamento de seus dados e processos; (e) componentes devem fornecer informações sobre propriedades não funcionais, como por exemplo desempenhos.

Uma vez que o componente serve de “matéria-prima”, sua documentação é imprescindível. Um componente, pelo menos, deve incluir uma especificação, um relatório de validação que o qualifica no ambiente para o qual foi projetado e propriedades não funcionais. Ainda quanto aos benefícios com a reutilização, Gimenes e Huzita (2005): destacam: (a) redução de custo e tempo de desenvolvimento; (b) gerenciamento de complexidade, uma vez que o software é subdividido em partes (componentes); (c) desenvolvimento paralelo, dado ao fato de cada componente ser independente dentro do seu domínio; (d) aumento da qualidade, sabendo-se que foram previamente utilizados e testados; (e) facilidade de manutenção, novamente argumentando-se sua independência.

3 Trabalhos correlatos

Existem no mercado uma significativa quantidade de sistemas operacionais embarcados, cada qual com suas características.

Sing et al (2004) propõem um método de geração de sistemas operacional embarcado baseado nas aplicações. A geração parte da identificação dos recursos solicitados pelo aplicativo. Os serviços de Sistema Operacional (SO) são então agregados de forma a construir um núcleo, de acordo com a necessidade do aplicativo

que nele executará, gerando-se ao final um SO específico para a aplicação.

O FreeRTOS (BARRY, 2006) é um sistema operacional multitarefa de código aberto. Todas as características básicas de um sistema operacional como escalonador de processos colaborativo e preemptivo com níveis de prioridades, alocação de memória e semáforos estão disponíveis nele. Seu diferencial é oferecer uma estrutura simples e eficiente.

Em Marcondes et al (2006) é descrito o desenvolvimento do EPOS, um sistema operacional orientado a aplicação e baseado em componentes, com ênfase na portabilidade. O projeto foi concebido utilizando meta-programação e programação orientada a aspectos. Como resultado foi obtido um sistema operacional que pode ser executado em uma grande variedade de arquiteturas.

4 Desenvolvimento

Esta seção apresenta o desenvolvimento, utilização e resultados deste trabalho sendo que o desenvolvimento do protótipo está dividido da seguinte forma: (a) GenosOs - é o sistema operacional base com código independente de SoC e sem nenhum suporte a dispositivos que deverão ser implementados em forma de componentes; (b) Componentes - são as unidades de software que provêm funcionalidades ao sistema em forma de *drivers* ou rotinas especializadas; (c) Genos - é a parte que abrange a ferramenta e com ele o usuário interage. Ao criar um novo projeto, é feita uma cópia do GenosOS para o diretório de trabalho do usuário. Na seqüência o usuário escolhe para qual SoC ele está desenvolvendo. O passo seguinte é a escolha dos componentes que irão compor o seu projeto e a implementação do aplicativo. A relação entre as partes citadas pode ser vista na figura 1 e são descritas nas seções seguintes.

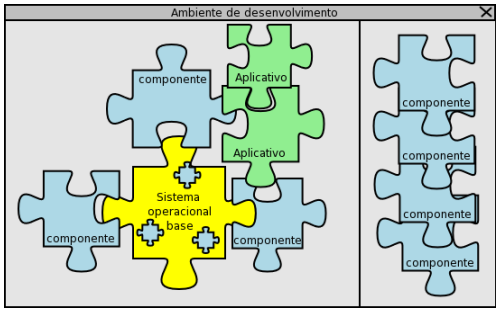


Figura 1: Arquitetura da ferramenta

O sistema foi construído utilizando a linguagem C++ e a biblioteca QT que provê classes para construir a interface gráfica e várias classes de uso geral como manipuladores de arquivos e listas entre outros.

4.1 Sistema operacional base - GenosOS

O sistema operacional base, ou simplesmente GenosOS, provê a estrutura inicial do sistema a ser desenvolvido e constitui a parte mais "interna"' do trabalho, caracterizando-se como o alicerce de todo o projeto a ser desenvolvido pelo usuário final. As subseções a seguir descrevem as especificações e implementação do mesmo.

Foram definidos os seguintes requisitos para o sistema: (a) o GenosOS deve ser um núcleo multitarefa, ou seja, com suporte a execução de processos concorrentes, e

para isso, deve prover funções para “iniciar tarefas” e “finalizar tarefas”. Deve ter também um escalonador de processos que suporte prioridades; (b) o sistema deve disponibilizar algum mecanismo para implementar exclusão mútua e sincronização de tarefas, como por exemplo, semáforos, possibilitando o gerenciamento de recursos compartilhados; (c) quanto à memória, embora o ARM7TDMI-S não possua uma MMU com proteção de memória, o núcleo deve oferecer rotinas de alocação dinâmica de memória, como “alocar memória”' e “liberar memória”; (d) este núcleo deve ainda fazer toda a inicialização do sistema como: inicializar a pilha de cada modo de operação, e executar as chamadas de inicialização do SoC, componentes e tarefas das aplicações e, (f) a linguagem utilizada é C e as ferramentas são do conjunto de desenvolvimento GNU.

Como ponto de partida para implementação do GenosOS foi utilizado o sistema operacional FreeRTOS. Esta alternativa mostrou-se viável pois o FreeRTOS apresenta código aberto (BERRY,2006), com uma série de características que atende aos requisitos deste trabalho. Portanto, o GenosOS é uma adaptação do projeto FreeRTOS que apresenta uma gama de funcionalidades e ainda assim com uma reduzida utilização dos recursos de hardware. Pelo fato dele suportar outros processadores, o projeto é bem dividido nas rotinas internas. Todo o código dependente de plataforma é separado.

Para criar o GenosOS, foi elaborada uma versão do FreeRTOS exclusiva para a arquitetura ARM7. Todos os códigos referentes a outros processadores foram removidos e os arquivos re-arranjados. Além disso, foi definido o fluxo do sistema que representa a interação com o restante do projeto. SoC, Componentes e Aplicativo não fazem parte do núcleo do GenosOS, eles são agregados pelo usuário durante o desenvolvimento de um projeto.

O fluxo do sistema compreende: (a) a função *start* faz as inicializações do processador como definir as pilhas para cada modo de operação; (b) a chamada *vSocSetupHardware()* que é uma função implementada nos arquivos do SoC e que estará disponível no projeto após o usuário selecionar o modelo desejado na paleta de SoCs. Esta função tem o objetivo de definir os parâmetros do SoC; (c) a chamada *vComponentSetupTasks()* faz as inicializações dos componentes. Essa chamada é conveniente para uma melhor organização e previne possíveis esquecimentos por parte do desenvolvedor durante a implementação do aplicativo; (d) o GenosOS invoca a função *vProgramSetupTasks()* que é chamada para iniciar as tarefas do aplicativo do usuário; (e) a chamada *vTaskStartScheduler()* cria a tarefa “*Idle*” e em seguida invoca a inicialização do escalonador; (f) o início das atividades do escalonador é efetivado com a inicialização do relógio do sistema através da chamada *vSocSetupTimerInterrupt()*.

Conforme a figura 2 foi formalizada uma estrutura de diretórios para poder separar os arquivos do SoC, componentes e programas. O diretório *components*, que parte do diretório raiz, acomodará os arquivos de configurações dos componentes em uso. Em *program* estarão todos os arquivos pertinentes ao aplicativo, sendo que *program.c* contém a chamada principal *vProgramSetupTasks()*. Os diretórios *component* e *soc* dentro de *src* e *include* conterão os arquivos relacionados aos componentes e ao SoC, respectivamente. O diretório raiz ainda conterá dois arquivos importantes que serão gerados pelo Genos, são eles: (a) Makefile - arquivo com as definições de compilação, e (b) script.ld - script do ligador com as definições de memória da placa do sistema embarcado.

4.2 A estrutura do sistema

Esta subseção apresenta os principais aspectos de arquitetura do GenosOS. São considerados: a inicialização do sistema, o processo de criação/destruição de tarefas, os estados das tarefas, o mecanismo de alocação de memória e escalonamento.

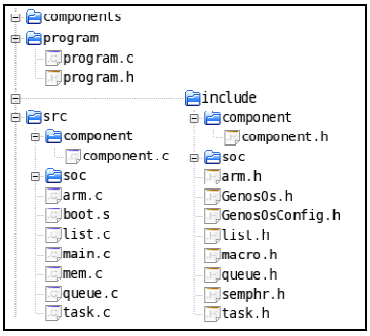


Figura 2: Estrutura de diretórios do GenosOS

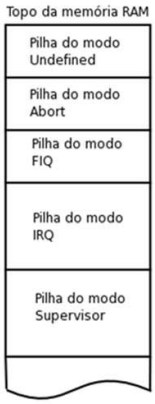
O GenosOS inicia definindo o tamanho da pilha e desabilitando as interrupções para cada modo conforme pode ser visto na figura 3 (em linguagem *assembly*). A partir da linha 2 até a linha 6 são criadas as variáveis com o tamanho da pilha de cada modo de operação. Em seguida (linhas 9 a 15) são definidos, como variáveis, os valores do CPSR de cada modo e mais duas variáveis que representam o bit de estado das interrupções no CPSR.

```
1     /*** Tamanho da pilha em cada modo ***/
2     .set  UND_STACK_SIZE, 0x00000004
3     .set  ABT_STACK_SIZE, 0x00000004
4     .set  FIQ_STACK_SIZE, 0x00000004
5     .set  IRQ_STACK_SIZE, 0X00000400
6     .set  SVC_STACK_SIZE, 0x00000400
7
8     /*** Definição padrão de cada modo ***/
9     .set  MODE_USR, 0x10      /* User Mode */
10    .set  MODE_FIQ, 0x11      /* FIQ Mode */
11    .set  MODE_IRQ, 0x12      /* IRQ Mode */
12    .set  MODE_SVC, 0x13      /* Supervisor Mode */
13    .set  MODE_ABT, 0x17      /* Abort Mode */
14    .set  MODE_UND, 0x1B      /* Undefined Mode */
15    .set  MODE_SYS, 0x1F      /* System Mode */
16    /*** Bits de interrupção ***/
17    .equ  I_BIT, 0x80         /* Quando I é 1, IRQ está desabilitada */
18    .equ  F_BIT, 0x40         /* Quando I é 1, IRQ está desabilitada */
19
20 _start:
21
22    ldr   r0, .LC6
23    msr   CPSR_c, #MODE_UND|I_BIT|F_BIT /* Undefined Instruction Mode */
24    mov   sp, r0
25    sub   r0, r0, #UND_STACK_SIZE
26    msr   CPSR_c, #MODE_ABT|I_BIT|F_BIT /* Abort Mode */
27    mov   sp, r0
28    sub   r0, r0, #ABT_STACK_SIZE
29    msr   CPSR_c, #MODE_FIQ|I_BIT|F_BIT /* FIQ Mode */
30    mov   sp, r0
31    sub   r0, r0, #FIQ_STACK_SIZE
32    msr   CPSR_c, #MODE_IRQ|I_BIT|F_BIT /* IRQ Mode */
33    mov   sp, r0
34    sub   r0, r0, #IRQ_STACK_SIZE
35    msr   CPSR_c, #MODE_SVC|I_BIT|F_BIT /* Supervisor Mode */
36    mov   sp, r0
37    sub   r0, r0, #SVC_STACK_SIZE
38    msr   CPSR_c, #MODE_SYS|I_BIT|F_BIT /* System Mode */
39    mov   sp, r0
40
```

Figura 3: Boot do Genos-Os.

Na linha 20 está o rótulo *_start* que marca o início do código a ser executado quando o sistema for ligado e será atribuído ao vetor de exceção na posição do *reset*. Na linha 22 o registrador R0 recebe *.L6* que é uma variável que contém o endereço do topo da memória RAM definido em tempo de ligação do código. Em seguida o processador entra em cada modo de operação e inicializa a pilha, atribuindo um endereço da RAM

no registrador *SP* que é privado para cada modo, ou seja, ele não está sendo sobrescrito. Ao final, as pilhas estarão de acordo com a figura 4(a). Resta ao GenosOS entrar em modo supervisor e efetuar a chamada *main()* dando continuidade na inicialização (agora escrito em linguagem de programação C) e entrar em operação (figura 4b).




```
1     msr   CPSR_c, #MODE_SVC|I_BIT|F_BIT
2
3     mov         r0, #0          /* argc = 0    */
4     mov         r1, #0          /* argv = null */
5
6     bl          main
```

Figura 4: (a) Localização das pilhas (b) chamada *main()*

Encerrado o processo de inicialização, o núcleo está pronto para criar tarefas. Uma tarefa é declarada como uma simples função no formato: void vFunction(void *pvParameters) e sua criação dá-se pela chamada xTaskCreate().

A função *xTaskCreate()* executa as seguintes ações: (a) alocação de memória para o descritor de tarefa (*tskTCB*); (b) alocação de memória para a pilha da tarefa; (c) preenchimento do descritor de tarefa com as informações passadas por parâmetro; (d) inicialização da pilha com o contexto da tarefa de forma a simular que a tarefa já estava sendo executada. Isto é necessário para que o escalonador possa extrair o contexto da tarefa ao iniciar a mesma; (e) inserção da tarefa na lista "tarefas prontas para executar"; (f) se a tarefa que está sendo criada tem a prioridade maior que a tarefa que está sendo executada, a nova tarefa inicia imediatamente.

Para a remoção de uma tarefa, é utilizado a função *xTaskDelete()*, que executa as seguintes ações: (a) a tarefa é removida da lista "tarefas prontas para executar"; (b) a tarefa é inserida na lista "tarefas para serem terminadas"; (c) é verificado se a tarefa esperava por algum evento e em caso afirmativo o pedido é removido da lista de eventos; (d) caso a tarefa a ser removida é a tarefa que está sendo executada, o escalonador é chamado para selecionar outra tarefa para ser executada. A lista de "tarefas para serem terminadas" é supervisionada pela tarefa do sistema *Idle* que termina de remover os recursos alocados pelas tarefas a serem removidas.

4.3 A estrutura de componentes no GenosOS

Os componentes contêm informações como nome, versão, autor, SoC compatível e descrição para que o usuário possa reconhecê-lo. Um componente deve conter de forma explícita as suas interfaces oferecidas e quando necessário, as interfaces requeridas. Uma interface constitui uma função que poderá ser usada por outro componente ou pelos programas do usuário. Além disso, um componente poderá conter configurações atribuídas em tempo de compilação, garantindo flexibilidade ao componente sem exigir recursos extras em tempo de execução. Uma configuração deverá ter nome, tipo esperado, valor padrão e descrição. Cada interface (oferecida ou requerida) e estrutura

deverá ser documentada tendo em vista prover instruções de utilização do componente para o programador que vier a fazer uso. Os componentes deverão ser escritos na linguagem de programação C.

Um componente é descrito através de um arquivo de definição. Neste arquivo estarão todas as informações pertinentes ao componente como nome, versão, autor, SoC compatível e descrição. Deve constar ainda no arquivo de definição, os nomes de todos os arquivos fontes que fazem parte do componente, as configurações que o componente oferece e as interfaces oferecidas e requeridas. Para as interfaces requeridas foi criada uma técnica chamada normalização. Essa técnica tenta prover uma camada de ligação entre a interface oferecida de um componente com a interface requerida de outro componente. A técnica é necessária, pois nem sempre um componente irá encontrar uma interface com o mesmo nome e parâmetros necessários.

Um componente é definido como um diretório contendo um arquivo texto chamado *component.conf*. Este arquivo contém todas as informações do componente. A especificação do arquivo é mostrada no exemplo da figura 5.

```
1  [General]
2  Name=mensageiro
3  Description="Envia mensagens através de outro componente (serial, LCD, etc)."
4  ShortDescription="Componente que manda mensagens"
5  Arch=Arm7
6  Soc=*
7  Group=software
8  Author=Filipe
9  Version=1.0
10
11 [Api]
12 api0=enviaMsg(char *msg)
13
14 [Api-dependent]
15 api0=enviaString(char *str)
16
17 [Api-dependent-user]
18 api0=
19
20 [ArmFiles]
21 file0=mensageiro.c
22
23 [HeaderFiles]
24 file0=mensageiro.h
```

Figura 5: Component.conf

A partir da linha 1, onde é definido o nome da seção de configuração geral à linha 9, são feitas as descrições básicas do componente na respectiva ordem: nome do componente, descrição, descrição breve para ser utilizada na "dica" da paleta de componentes do ambiente de desenvolvimento, a arquitetura do componente tendo-se em vista futuras versões que suportem outras arquiteturas, modelo do SoC que pode ser um asterisco representando independência de SoC, grupo no qual o componente se caracteriza, autor do componente e versão. A seção de nome *Api*, na linha 11, é onde são definidas as interfaces oferecidas pelo componente. Cada interface vem a seguir representada pela palavra *apiN* onde $N \geq 0$. Um exemplo pode ser visto na linha 12.A seção de nome *Api-dependent*, na linha 14, é onde são definidas as interfaces requeridas pelo componente. Cada interface vem a seguir representada pela palavra *apiN* onde $N \geq 0$. Um exemplo pode ser visto na linha 15.

A seção *Api-dependent-user* (linha 17) nunca é preenchida em tempo de criação do componente. Nesta seção estarão as configurações das interfaces requeridas após passarem pela normalização que será descrita mais adiante com um estudo de caso. Na seção *HeaderFiles* são definidos os arquivos de cabeçalho e em seguida os arquivos fontes nas seção *ArmFiles* (arquivos que não suportam *Thumb Mode*) e *ThumbFiles* (arquivos que podem ser compilados em *Arm* ou *Thumb mode*).

5 Estudo de caso

Esta seção apresenta um estudo de caso demonstrando o funcionamento da técnica de normalização que permite a um componente utilizar os serviços de outro componente mesmo que a interface requerida de um componente não tenha sido definida com o mesmo nome e parâmetros disponíveis em uma interface oferecida de outro componente. O objetivo é permitir uma maior compatibilidade entre componentes.

O primeiro componente é o comp-uart, um controlador de interface serial nativo para o SoC LPC2106. O arquivo de definição do componente comp-uart pode ser visto na figura 11. Este componente oferece 3 interfaces: *void sendChar(char byte)*, *void sendString(char *str, int length)* e *char getChar(void)* além de uma chamada de inicialização: *void compuartInit(void)*.

O segundo componente é o mensageiro, um componente de software que oferece apenas uma interface (*sendMsg(char *msg)*) e contém uma interface requerida chamada *enviaPorOutroComp(char *msg)*. Este componente envia uma mensagem utilizando outro componente que pode ser um controlador de *display* de cristal líquido (LCD), memória, interface serial, etc. Para este exemplo, o componente mensageiro utilizará o componente comp-uart. Na figura 6a pode ser vista sua descrição. Ao adicionarem-se os componentes no projeto, ambos são copiados para uma estrutura de diretórios pré-definida.

```
[General]
Arch=Arm7
Author=Filipe
Description=driver serial para o lpc2106
Group=serial
Name=comp_uart
ShortDescription=driver serial
Soc=lpc2106
Version=1.0

[Api]
api0=void sendString(char *str, int length)
api1=void sendChar(char byte)
api2=char getChar(void)
api3=void comp_uartInit(void)

[ArmFiles]
file0=comp_uart.c

[HeaderFiles]
file0=comp_uart.h
```

```
[General]
Arch=Arm7
Author=Filipe
Description=Este componente manda mensagens através de outro componente.
Group=software
Name=mensageiro
ShortDescription=Componente que manda mensagens
Soc=*
Version=1.0

[Api]
api0=sendMsg(char *msg)

[Api-dependent]
api0=sendPorOutroComp(char *msg)

[ArmFiles]
file0=mensageiro.c

[HeaderFiles]
file0=mensageiro.h
```

Figura 6: (a) Definição do componente comp-uart; (b) definição do componente mensageiro.

Para o componente mensageiro, é necessário resolver a pendência da interface requerida e isto é feito no arquivo de descrição localizado agora dentro do diretório do projeto. Para tanto, a seção *Api-dependent-user* será utilizada. Uma entrada nesta seção tem o formato api*N*= "*parâmetro1*;*parâmetro2*;*parâmetro3*" onde: (i) $N \geq 0$ e corresponde à mesma entrada na seção *Api-dependent*; (ii) o *parâmetro1* é o nome do componente que está sendo utilizado; (iii) o *parâmetro2* é a função que está sendo utilizada do componente do *parâmetro1*; (iv) o *parâmetro3* é o formato da chamada que será utilizada pelo componente requisitante (para este exemplo, o mensageiro).

Observe-se que os nomes dos argumentos serão sempre arg*N* onde $N \geq 0$ e o programador deve adequar os parâmetros na função a ser utilizada (figura 6b).

Para o componente mensageiro, uma solução poderia ser: "*comp-uart;sendString(arg0, strlen(arg0));enviaPorOutroComp(arg0)*". A técnica de normalização age em tempo de compilação utilizando o comando *#define* da linguagem C, ou seja, quando *mensageiro.c* for compilado e o compilador encontrar a chamada *enviarPorOutroComp(msg)*, esta será substituída por *sendString(msg, strlen(str))* que é uma chamada real. Apenas haverá *overhead* quando existir a necessidade de adequação dos parâmetros.

Para que o componente encontre as declarações e a chamada correta, ao se desenvolver um componente que tenha interfaces requeridas, o programador deve colocar um *#include* de um arquivo chamado *components.h*. O Genos cria o arquivo chamado *components.h* e adiciona todos os cabeçalhos de todos os componentes do projeto nele. O arquivo *gen-mensageiro.h* contém o *#define* para substituir a chamada da interface requerida pela chamada real extraída do arquivo de definição do componente. Dessa forma, tanto a declaração quanto a chamada da função utilizada pela interface requerida é encontrada em tempo de compilação. Todo este processo é feito através do Genos de forma visual e automatizada (figura 7a).

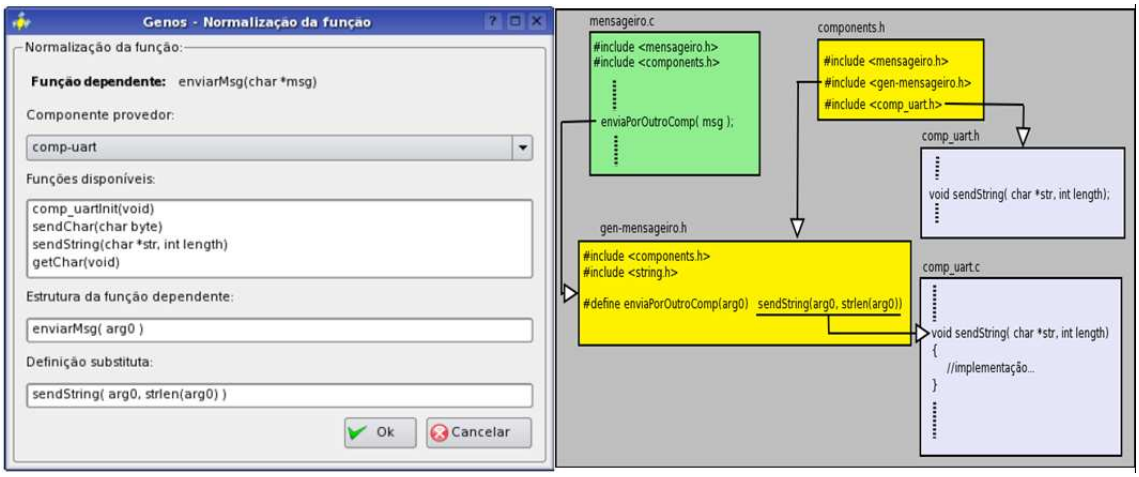


Figura 7 (a): Normalização da função *enviarMsg();* (b): Relacionamento entre módulos

A figura 7b apresenta de forma gráfica o relacionamento de todas as partes no exemplo do componente mensageiro utilizando a interface oferecida do componente comp-uart.

5.1 O componente SoC

O suporte a um determinado modelo de SoC é dado através de componentes de SoC. O seu funcionamento é muito semelhante ao componente de software. A diferença está em seu objetivo que é prover duas funções ao sistema que são: (i) *void vSocSetupTimerInterrupt(void)* - inicia um temporizador para prover uma chamada regular no determinado intervalo de tempo. A função do GenosOS vPreemptiveTick() deverá ser atribuída à esta interrupção; (ii) *volatile void vSocSetupHardware(void)* - faz as inicializações necessárias do SoC como *clock* do sistema, *clock* do barramento de periféricos e quaisquer outras características específicas do SoC.

O componente de SoC deve prover ainda um arquivo de cabeçalho contendo as definições de endereços dos registradores do sistema utilizando a nomenclatura usada pelos manuais do ARM ou do fabricante do SoC. Assim como existe o arquivo *component.conf* que especifica um componente, para um componente de SoC existe o *soc.conf*. A criação e edição de um componente de SoC no Genos são feitas utilizando-se um módulo específico.

6. Considerações finais

O presente trabalho descreveu a estrutura de uma ferramenta que utiliza o Free-RTOS como base para o desenvolvimento de software de sistemas embarcados utilizando componentes. O objetivo principal foi criar uma plataforma de desenvolvimento de software para sistemas embarcados com base em um sistema operacional embarcado modular construído com a agregação de componentes.

O uso do ambiente com uma metodologia de desenvolvimento (componentes) aliada a uma plataforma base (sistema operacional) mostrou-se eficiente, pois todos os elementos necessários encontram-se ao alcance do usuário em todos os momentos do desenvolvimento.

A técnica de normalização teve um resultado positivo no trabalho adicionando uma característica relevante de interoperabilidade com o mínimo de sobrecarga no sistema. Para muitas situações a sobrecarga é zero, uma vez que a adequação dos parâmetros é feita em tempo de compilação.

Como resultado o GenosOS demonstrou ser um sistema operacional simples e ao mesmo tempo completo e flexível podendo suportar vários modelos de SoC. O exemplo utilizado no estudo de caso gerou um arquivo binário com o tamanho de aproximadamente 13 KBytes. Uma comparação com o tamanho do binário dos outros sistemas operacionais não pode ser feita pois as condições não seriam as mesmas ou não houve acesso ao software.

Referências

Barry, R. FreeRTOS - a free RTOS for small embedded real time systems. 2006.

Gimenes, I.M.de S. Huzita, E.H.M. Desenvolvimento baseado em componentes: conceitos e técnicas. Ciência Moderna, Rio de Janeiro. 304 p. 2005.

Philips. Lpc2104/2105/2106 user manual. 2003.

Singh,G.;Biswas,B.S.;Kundu,S.;Mukhopadhaya,A.;Worah,P.;Basu,A. OaSis: an application specific operating system for an embedded environment. Proceedings..., pp 776-779, 2004. 17th International Conference on VLSI Design, IEEE.

Tanenbaum,A.S. Sistemas operacionais modernos. Prentice Hall, São Paulo, 2 ed, 2003.