UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

GENOS - PROTÓTIPO DE UM MONTADOR DE SISTEMAS OPERACIONAIS PARA SISTEMAS EMBARCADOS

FILIPE RENALDI

BLUMENAU
2006

2006/I-14

FILIPE RENALDI

GENOS - PROTÓTIPO DE UM MONTADOR DE SISTEMAS OPERACIONAIS PARA SISTEMAS EMBARCADOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação – Bacharelado.

Prof. Antônio Carlos Tavares – Orientador

BLUMENAU
2006

2006/I-14

GENOS - PROTÓTIPO DE UM MONTADOR DE SISTEMAS OPERACIONAIS PARA SISTEMAS EMBARCADOS

Por

FILIPE RENALDI

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

____

Presidente: Prof. Antônio Carlos Tavares – Orientador, FURB

____

Membro: Prof. Mauro Marcelo Mattos, Dr. FURB

____

Membro: Prof. Miguel Alexandre Wisitainer, Mestre, FURB

BLUMENAU
2006

Dedico este trabalho aos

meus pais, irmãos e namorada.

O pensamento lógico pode levar você de A a B,

mas a imaginação te leva a qualquer parte do Universo.

(Albert Einstein).

AGRADECIMENTOS

Aos meus pais e irmãos pelo apoio.

À Miriam, minha namorada pelo seu amor e companherismo.

Aos amigos, pela amizade.

Aos professores, em especial a meu orientador e amigo Tavares.

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.

Palavras Chave: Sistemas operacionais. Sistemas embarcados. Desenvolvimento baseado em componentes. Arquitetura ARM7.

ABSTRACT

This work presents the specification and implementation of an environment of development for embedded systems based in ARM7 processor. The system encloses an embedded operating system, a development method using component and the proper software of the development environment. The operating system is based on the kernel of the FreeRTOS that offers a base structure where the developer expands it as its necessities through reusables components. The result is an environment that can to provide some easinesses to build embedded system software like project management and compilation.

Key-Words: Operating systems. Embedded systems. Component-based development.

ARM7 architecture.

LISTA DE ILUSTRAÇÕES

Figura 2.1 – Registradores do ARM7TDMI-S 22

Figura 2.2 – Placa LPC-P2106 da Olimex 23

Figura 3.1 – Arquitetura da ferramenta 27

Figura 3.2 – Diagrama de seqüencia da inicialização do GenosOS 29

Figura 3.3 – Estrutura de diretórios do GenosOS 30

Quadro 3.1 – Boot do GenosOs 32

Figura 3.4 – Localização das pilhas 32

Quadro 3.2 – Chamada main() 33

Quadro 3.3 – Exemplo de criação de uma tarefa com xTaskCreate() 33

Figura 3.5 – Possíveis transições de estados das tarefas 35

Quadro 3.4 – Component.conf 36

Quadro 3.5 – Exemplo de uma função documentada com o Doxygen 38

Figura 3.6 – Resultado de uma função documentada com o Doxygen 38

Quadro 3.6 – Definição do componente comp-uart 39

Quadro 3.7 – Definição do componente mensageiro 39

Figura 3.7 – Exemplo da técnica de normalização 41

Quadro 3.8 – Definição da GPIO. 42

Quadro 3.9 – Especificação de um componente de SoC 42

Figura 3.8 – Configuração dos diretórios 44

Figura 3.9 – Criação do SoC 44

Figura 3.10 – Criação do componente 44

Figura 3.11 – Novo projeto 45

Figura 3.12 – Processo de normalização 45
Figura 3.13 – Seleção de componentes 46
Figura 3.14 – Criação dos arquivos de programas 46
Figura 3.15 – Diagrama de sequência do processo de construção do projeto 46
Figura 3.16 – Diagrama de classes do Genos 47
Figura 3.17 – Detalhes da classe Project 48
Figura 3.18 – Detalhes da classe Genos 48
Figura 3.19 – Genos sendo executado 49
Quadro 3.10 – Construtor da classe Genos 50
Quadro 3.11 – Início da tarefa Novo Componente 51
Quadro 3.12 – Criando arquivo de definição com *QSettings* 52
Quadro 3.13 – Método da classe ProjectNew que cria um novo projeto. 53
Figura 3.20 – Genos 54
Figura 3.21 – Edição do arquivo soc.c 55
Figura 3.22 – Tela de criação do componente de SoC 55
Figura 3.23 – Tela de criação do componente 56
Figura 3.24 – Paleta de componentes 56
Figura 3.25 – Novo projeto 57
Figura 3.26 – Seleção do SoC 57
Figura 3.27 – Edição do projeto 58
Figura 3.28 – Normalização da função *enviarMsg()* 59
Figura 3.29 – Configuração do componente comp-uart 59
Figura 3.30 – Configuração da placa do sistema embarcado 60
Figura 3.31 – Gerenciamento dos arquivos de programas 61
Quadro 3.14 – Arquivo do programa - prog.c 62

Quadro 3.15 – Arquivo do programa - program.c 62
Figura 3.32 – Gravação do sistema embarcado 63
Figura 3.33 – Mensagens sendo enviada pelo sistema embarcado 63

SUMÁRIO

1 INTRODUÇÃO **14**

1.1 ESCOPO E PROBLEMA 14

1.2 OBJETIVO 15

1.3 ESTRUTURA DO TEXTO 15

2 FUNDAMENTAÇÃO TEÓRICA **17**

2.1 SISTEMAS EMBARCADOS 17

2.2 SISTEMAS OPERACIONAIS 18

2.2.1 Sistemas operacionais embarcados e de tempo real 19

2.3 ARQUITETURA ARM 20

2.3.1 Histórico 20

2.3.2 Características do ARM7TDMI-S 20

2.3.3 Características do SoC LPC2106 22

2.4 DESENVOLVIMENTO BASEADO EM COMPONENTES 24

2.5 TRABALHOS CORRELATOS 25

3 DESENVOLVIMENTO **27**

3.1 SISTEMA OPERACIONAL BASE - GENOSOS 28

3.1.1 Requisitos principais 28

3.1.2 Visão geral da solução proposta 28

3.1.3 Especificação 29

3.1.4 Implementação 31

3.2 COMPONENTE . . . 35
3.2.1 Requisitos principais . . . 35
3.2.2 Visão geral da solução proposta . . . 36
3.2.3 Especificação . . . 36
3.2.4 Implementação . . . 37
3.2.5 O componente SoC . . . 41
3.3 AMBIENTE DE DESENVOLVIMENTO - GENOS . . . 42
3.3.1 Requisitos principais . . . 42
3.3.2 Visão geral da solução proposta . . . 43
3.3.3 Especificação . . . 43
3.3.4 Implementação . . . 49
3.4 UTILIZAÇÃO DO SISTEMA . . . 54
3.5 RESULTADOS E DISCUSSÃO . . . 63

4 CONCLUSÃO **65**
4.1 TRABALHOS FUTUROS . . . 66

REFERÊNCIAS BIBLIOGRÁFICAS **68**

1 INTRODUÇÃO

Um sistema embarcado (SE) é um computador de propósito específico encapsulado no dispositivo o qual ele controla. O SE é projeto para fins bem específicos e executa tarefas pré-definidas (WIKIPÉDIA, 2006a). Estando presentes nos mais diversos objetos do cotidiano, os sistemas embarcados são encontrados em telefones celulares, televisões, automóveis, brinquedos e tantos outros equipamentos de consumo, além de máquinas industriais, sistemas militares e médicos.

Por tratar-se de sistemas quase sempre de dimensões reduzidas, os sistemas embarcados tendem a ter recursos (processamento, memória, barramentos, periféricos) limitados, mas com o avanço tecnológico da eletrônica, permitiu-se integrar 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, que é produzido pela Philips, e conta com um processador ARM7TDMI-S e diversos dispositivos como memória RAM, memória *flash*, várias interfaces seriais, relógio de tempo real, temporizadores, interface *joint test action group* (JTAG), entre outros (PHILIPS, 2003).

Com esta **convergência digital**, o hardware vem se tornando mais complexo. Em mesmo grau, cresce a demanda de desenvolvimento de software para gerenciar tais recursos. É neste contexto que os sistemas operacionais são empregados, abstraindo os detalhes do hardware e criando uma interface mais simples para os programas de usuário (TANENBAUM, 2003).

1.1 ESCOPO E PROBLEMA

Cada projeto de sistema embarcado tende a ser específico de sua aplicação, ou seja, os periféricos e requisitos do software abrangem o ambiente e objetivo do produto do qual ele faz parte. Esta característica exige um alto nível de modularidade dos sistemas

operacionais para que sejam adaptáveis.

A utilização de diferentes sistemas operacionais embarcados para diferentes projetos pode trazer um atraso de desenvolvimento pela curva de aprendizagem do programador, ou seja, a necessidade de se desenvolver no menor intervalo de tempo, garantindo menores custos e um reduzido *time-to-market*[1] faz do software (sistema operacional e aplicação) uma parte sensível no projeto de sistemas embarcados.

Este trabalho busca, dentro deste contexto, implementar uma ferramenta 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.

1.2 OBJETIVO

O objetivo deste trabalho é criar uma plataforma de desenvolvimento de software para sistemas embarcados com base em um sistema operacional embarcado modular construído com a agregação de componentes.

Os objetivos específicos do trabalho são:

a) criar um sistema operacional modular, com metodologia de construção orientada a componentes para plataformas que utilizam o processador ARM7TDMI-S;
b) criar um modelo de componente a ser utilizado pelo sistema;
c) criar um ambiente de desenvolvimento para compor o sistema operacional e software embarcado utilizando componentes.

1.3 ESTRUTURA DO TEXTO

O trabalho está organizado em 4 capítulos. O capítulo 1 provê a introdução do trabalho apresentando ainda o escopo e objetivos deste trabalho.

No capítulo a seguir (capítulo 2), está a fundamentação teórica, onde são apresentados os conceitos e tópicos mais importantes para o elaboração e compreensão deste trabalho. São tratados os sistemas embarcados e sistemas operacionais com aplicações em

[1] Intervalo de tempo levado desde a concepção de uma idéia até a comercialização como produto final.

ambientes embarcados. São ainda descritas as principais características da plataforma e o desenvolvimento modular com componentes bem como suas contribuições. O capítulo é finalizado descrevendo-se os trabalhos correlatos.

No capítulo 3, é relatado o desenvolvimento do protótipo e está divido em 4 principais seções: Sistema operacional base, Componentes e Ambiente de desenvolvimento. O capítulo é finalizado apresentando-se a utilização do sistema.

O capítulo final conta com as conclusões e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos necessários para a compreensão deste trabalho, abordando as tecnologias e princípios utilizados ao longo de sua idealização e confecção.

2.1 SISTEMAS EMBARCADOS

Sistemas computacionais estão em todos os lugares. Não é surpresa que milhões são construídos todos os anos para o uso pessoal. O que é surpreendente é que são construídos todos os anos bilhões de sistemas embarcados (VAHID; GIVARGIS, 2001, p. 1).

Sistemas embarcados, acima de tudo, são computadores e possuem características comuns como processador, memória e periféricos, então, o que os diferenciam de outros sistemas computacionais ? Segundo Li e Yao (2003, p. 5), uma definição geral para sistemas embarcados considera os mesmos como sendo sistemas computacionais especialmente projetados para uma função dedicada. A palavra embarcado (alguns textos utilizam embutido ou até mesmo *embedded*) reflete o fato desses sistemas serem parte integrante de um sistema maior. Não necessariamente em tamanho, mas em utilidade, ou melhor, em aplicação.

Alguns autores caracterizam os sistemas embarcados como sendo sistemas de recursos limitados. Tal limitação beira as necessidades da aplicação, ou seja, nada impede que os mesmo tenham recursos superiores quando comparados à alguns computadores pessoais utilizados atualmente. Por se tratar de um custo no projeto de um produto, seus recursos limitam-se ao necessário.

Uma vez que esse trabalho trata o desenvolvimento de sistemas embarcados, é válido citar as métricas de projetos enumeradas por Vahid e Givargis (2001, p. 4-5). As métricas de projeto são características mensuráveis da implementação do sistema, conforme são vistas a seguir:

a) custo de protótipo (*nonrecurring engineering cost* (NRE)) - custo para desenvol-

vimento de uma unidade em tempo de projeto. Uma vez projetado, as demais unidades manufaturadas não terão custo adicional;

b) custo unitário - custo para a fabricação de cada unidade, sem contar o custo de prototipação;

c) tamanho - mensurado em bytes para o software e transístores para o hardware;

d) potência - potência de consumo de energia do sistema o qual determina a autonomia com baterias e os requisitos de refrigeração dos circuitos integrados, uma vez que o acréscimo de potência significa aumento do calor dissipado;

e) flexibilidade - possibilidade de mudanças na funcionalidade do sistema com o menor NRE possível. O software é considerado a parte mais flexível no sistema;

f) tempo de prototipação - tempo necessário para construir um protótipo funcional do sistema, o qual pode ser maior e mais caro que o produto final e que atenda os requisitos do projeto e valide-o;

g) *time-to-market* - intervalo de tempo levado desde a concepção da idéia do projeto até a sua comercialização como produto final;

h) manutenabilidade - possibilidade de modificação ou atualização do sistema após a versão inicial, principalmente para manutenção;

i) acurácia - implementação dos requisitos conforme o especificado. Exatidão nos resultados esperados;

j) segurança - probabilidade do sistema não causar danos aos seus operadores ou ao próprio sistema.

2.2 SISTEMAS OPERACIONAIS

A programação direta de aplicativos sobre o hardware exigiria por parte dos programadores um conhecimento aprofundado de cada arquitetura e periférico, além do domínio da aplicação a ser desenvolvida. Outro agravante é a interação entre softwares (ou partes de um mesmo) feitos por diferentes programadores num mesmo projeto, onde questões fora do escopo da aplicação deveriam ser formalizadas. A utilização de um sistema operacional (SO) busca resolver esses problemas, "estendendo"o sistema para que o programador

da aplicação tenha uma visão mais abstrata do hardware, sendo possível interagir com os periféricos do sistema sem se preocupar com detalhes mais técnicos. O SO também tem o perfil de "gerenciador de recursos", administrando os recursos computacionais como por exemplo o acesso a periféricos por diferentes processos e a divisão do tempo em que cada processo estará executando (TANENBAUM, 2003, p. 2-4).

2.2.1 Sistemas operacionais embarcados e de tempo real

Dada a relação direta com o hardware do sistema computacional, o SO para um sistema embarcado segue a mesma filosofia de restrição de recursos. Uma vez que haja tal limitação, o sistema operacional, como camada de software intermediária entre aplicativos e hardware, deve consumir o mínimo de recursos possível.

Sistemas operacionais embarcados são projetados para serem bastante compactos e eficientes, eliminando funcionalidades que não sejam necessárias para os aplicativos executados no sistema. A área de aplicação dos sistemas embarcados é extensa, as necessidades de um sistema hospitalar de monitoração de sinais vitais são bem diferentes de um sistema de exploração espacial, por exemplo. Agregar tantas características em um mesmo SO para criar um produto abrangente se torna inviável sem uma estratégia de modularidade (GERVINI et al., 2003, p. 2).

Dada a natureza das aplicações, é bastante comum um sistema embarcado apresentar restrições de tempo. Por este motivo, grande parte dos sistemas operacionais embarcados, também são sistemas operacionais de tempo real – *real time operatings system* (RTOS). Santo (2001, p. 38) define RTOS como sendo um sistema operacional determinístico quanto ao tempo. Ou seja, determinada computação precisa ser processada antes de um determinado tempo limite imposto pela aplicação. As aplicações de tempo real são classificados em (LI; YAO, 2003, p. 15):

a) *soft real-time* - é um sistema com tempos limites mas com certo grau de flexibilidade. O sistema tolera certo nível de variação e a perda de um tempo limite não resulta numa falha, embora possa causar atrasos, o que dependerá

da aplicação;

b) *hard real-time* - é um sistema inflexível com o tempo. Se um determinado tempo limite for perdido, irá ocorrer uma catástrofe. Os resultados de uma computação obtidos após o tempo limite é inútil ou com grande margem de erro.

2.3 ARQUITETURA ARM

Esta seção apresenta a arquitetura ARM e suas principais características, uma vez que este trabalho tem como objetivo o desenvolvimento para esta arquitetura.

2.3.1 Histórico

A história do processador ARM deu-se início em 1983 pela Acorn Computer Ltd. projetando o processador CMOS 6502. Em 1985 a mesma projetou o ARM1 que virou realmente produto no ano seguinte com o ARM2. Em 1990 uma parceria com a Apple Computers e VLSI Technology, a Acorn Computer Ltd. criou a Advanced RISC Machine Ltd.

A Advanced RISC Machine Ltd. não fabrica os processadores ARM, ela os projeta e licencia, assim como vários outros módulos utilizados para compor um SoC. Várias empresas de semicondutores produzem os processadores ARM (e seus agregados), adicionando suas próprias funcionalidades e periféricos. Exemplos de empresas que produzem o ARM são: Philips, Atmel, Texas Instruments, Cirrus Logic, Intel, IBM, Sharp e Samsung entre outras (WIKIPÉDIA, 2006b; ARM, 2005).

2.3.2 Características do ARM7TDMI-S

As características apresentadas aqui referem-se à família de processadores ARM7TDMI-S especificada pela Advanced RISC Machine Ltd. e extraídas de ARM (2001). Na subseção seguinte (página 22), são apresentadas as características do processador LPC2106 pertencente à família ARM7TDMI-S e produzido pela Philips e no qual os softwares deste trabalho estão sendo desenvolvidos.

O ARM7TDMI-S é um processador com arquitetura *reduced instruction set computer* (RISC) de 32 bits sendo que possui 2 conjuntos de instruções: 32 bits (*ARM mode*) e 16 bits (*Thumb mode*). O *Thumb mode* é um conjunto das instruções de 32 bits mais utilizadas utilizando-se um conjunto de instruções de 16 bits. Isso permite maior densidade de código, o que resulta em economia de memória, e melhor performance sem abrir mão das características de 32 bits do processador como endereçamento, registradores e cálculos de 32 bits mas representados em 16 bits. O processador pode alterar entre *ARM mode* e *Thumb mode* em tempo de execução.

O ARM7TDMI-S tem 7 modos de operação:

a) *user mode* - é o de menor privilégio e no qual as aplicações devem executar;

b) *fast interrupt* (FIQ) - é um modo de interrupção rápida, com uma latência menor que o modo *interrupt* pois possui 8 registradores privados, eliminando a necessidade de salvá-los durante as trocas de contextos;

c) *interrupt* (IRQ) - modo de interrupção normal. Tem prioridade menor que o FIQ;

d) *supervisor mode* - modo para implementar interrupções por software;

e) *abort mode* - o processador entra neste modo quando uma instrução gera uma situação de "abortar", como por exemplo um acesso a um endereço de memória não disponível;

f) *system mode* - modo privilegiado para o sistema operacional;

g) *undefined mode* - o processador entra neste modo quando uma instrução indefinida é executada.

Todos os modos são ditos privilegiados, exceto o *user mode*, que não pode acessar determinados recursos como, por exemplo, escrever no registrador *current program status register* (CPSR) que é descrito a seguir.

Quanto aos registradores, o ARM7TDMI-S pode acessar 16 registradores (r0-r15) de propósito geral de 32 bits cada um, sendo que em determinados modos de operação os registradores são privados (duplicados) o que resulta em 31 registradores ao todo. Há

também o registrador de status CPSR que mantém informações como o modo de operação atual, estado global das interrupções, as *flags* de condição e o modo de execução (*ARM mode* ou *Thumb mode*). O processador conta ainda com o registrador *saved program status register* (SPSR) que é privado para cada modo privilegiado menos para *system mode* e *user mode*. Neste registrador fica salvo o CPSR do modo anterior ao atual. Veja em detalhes os registradores na fig. 2.1.

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Registrador duplicado

Fonte: adaptado de ARM (2001).

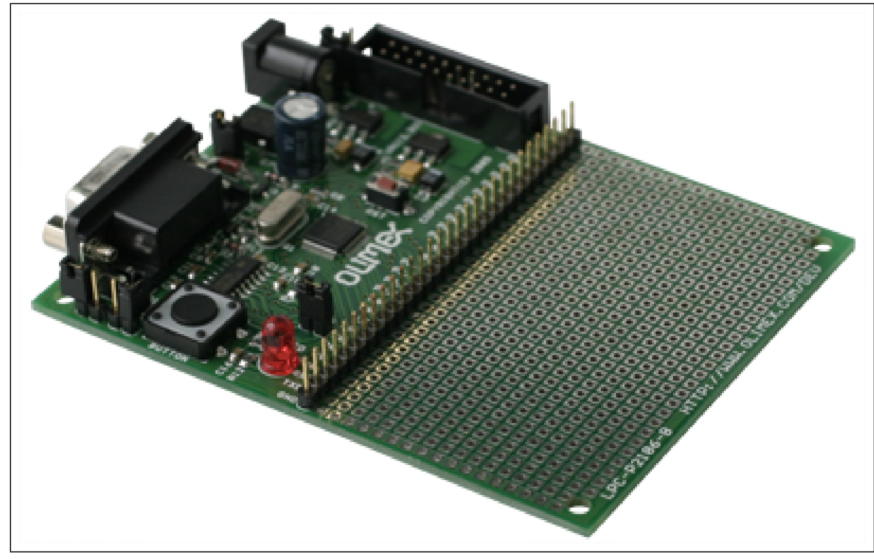
Figura 2.1 – Registradores do ARM7TDMI-S

2.3.3 Características do SoC LPC2106

Para completar a visão de um processador ARM, esta subseção apresenta o LPC2106, um processador ARM7TDMI-S produzido pela Philips como um SoC de propósito geral, pertencente à familia LPC2000 que conta com mais de 25 modelos. Os dados foram extraídos do manual do fabricante do SoC (PHILIPS, 2003).

O LPC2106 é o processador presente na placa LPC-P2106 (fig. 2.2) produzida pela Olimex Ltd. e que foi utilizado no desenvolvimento deste trabalho. É um processador extremamente pequeno, medindo apenas 7x7 mm. Além de implementar o núcleo do

ARM7TDMI-S, o processador conta com 128 *KBytes* (KB) de memória *flash* e 64 KB de memória ram internas e podendo trabalhar em uma freqüência de até 60 Mhz.



Fonte: Olimex (2006).

Figura 2.2 – Placa LPC-P2106 da Olimex

Sua lista de periféricos conta com 2 *universal asynchronous receiver/transmitter* (UART), 1 *serial peripheral interface* (SPI), *pulse width modulation* (PWM) com até 6 saídas, 1 *inter-integrated circuit* (I^2C), 2 *timers*, relógio de tempo real, *watchdog* e uma porta de entrada e saída de propósito geral de 32 pinos que podem ser setados individualmente.

Para que o LPC2106 possa utilizar a estrutura de interrupções do ARM7TDMI-S, ele implementa um hardware chamado *vectored interrupt controller* (VIC) que passa a ser uma interface de acesso. Com o VIC, o processador oferece 32 entradas de interrupções para serem utilizadas com os periféricos do SoC, interrupções por software ou interrupções externas.

Quanto à gravação na memória *flash*, o SoC provê dois métodos:

a) *in-system programming* (ISP) - neste método, um software de *boot loader* da Philips que reside nos últimos 8 KB da memória *flash* é iniciado quando o pino "P0.14"está em nível lógico zero. Utilizando a UART, ele recebe o programa e grava o mesmo na memória *flash*;

b) *in-application programming* (IAP) - neste método o programa de usuário

pode acessar funções específicas (residentes na memória *flash*) para gravar na memória *flash*.

Estas e outras informações detalhadas, como o protocolo do ISP e funções do IAP estão disponíveis no manual do produto (PHILIPS, 2003).

2.4 DESENVOLVIMENTO BASEADO EM COMPONENTES

> A noção de reutilização é antiga e teve inicio desde o tempo em que as pessoas começaram a encontrar soluções consistentes para problemas. Esta busca foi motivada pela idéia de que, uma vez encontrada a solução, esta poderia ser aplicada a novos problemas. A aplicação de uma solução, repetida por várias vezes, acaba por torná-la aceita, generalizada e padronizada. (GIMENES; HUZITA, 2005).

Um componente é definido como uma unidade de software que encapsula em si 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; HUZITA, 2005):

a) as interfaces fornecidas de um componente devem ser identificadas e definidas separadamente, ou seja, deve ser fornecida a especificação, de forma clara, de seus serviços. Cada interface consiste em serviços especificados, mediante uma ou mais operações, sendo cada uma delas separadamente identificada e especificada de acordo com seus parâmetros de entrada e saída e respectivos tipos estabelecidos. Essas definições constituem a assinatura da interface;
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;

d) 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.

2.5 TRABALHOS CORRELATOS

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

O VxWorks (VXWORKS, 2005), é um software comercial desenvolvido pela empresa WindRiver e é encontrado em várias áreas, como por exemplo, sistemas de controle de processos em indústrias, simuladores de vôo e sistemas de navegação em automóveis. Suporta memória compartilhada e escalonamento preemptivo. A WindRiver oferece ainda um conjunto de ferramentas de desenvolvimento como compilador e depurador.

O uClinux (DIONE; DURRANT, 2005) é um derivativo do sistema operacional Linux, com o diferencial de executar em processadores sem *memory management unit* (MMU). O projeto tem o código fonte aberto e suporta uma variedade de processadores.

Outra característica do uClinux é que ele herda do Linux toda a variedade de aplicativos disponíveis.

Singh et al. (2004) propõem um método de geração de sistemas operacionais embarcados baseado nas aplicações. A geração parte da identificação dos recursos solicitados pelo aplicativo. Os serviços de 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. De modo semelhante,

O FreeRTOS (BARRY, 2006) é um sistema operacional multitarefa de código aberto. Todas as características básicas de um sistemas 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. Um binário do FreeRTOS para o ARM7TDMI-S compilado para o SoC LPC2106 tem seu tamanho de aproximadamente apenas 20KB.

3 DESENVOLVIMENTO

Este capítulo 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 unidade 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 fig. 3.1 e são descritas nas seções seguintes.

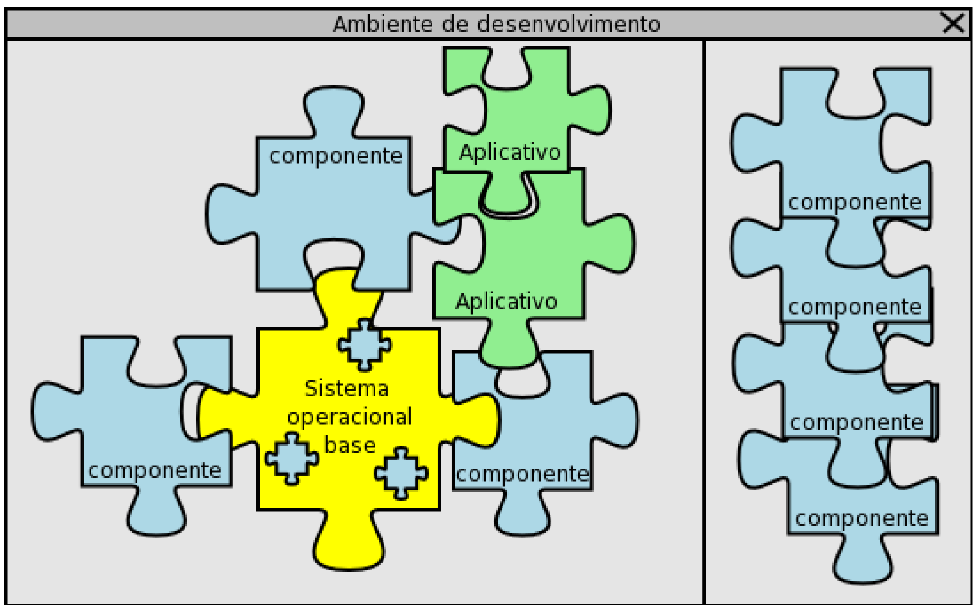


Figura 3.1 – Arquitetura da ferramenta

3.1 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.

3.1.1 Requisitos principais

O GenosOS deve ser um núcleo multitarefa, ou seja, com suporte a processos executando de forma concorrente, e para isso, deve prover funções para “iniciar tarefas[1]” e “finalizar tarefas”. Deve ter também um escalonador de processos que suporte prioridades.

O sistema deve prover mecanismos de controle para regiões críticas, como por exemplo semáforos, possibilitando a mútua exclusão em recursos compartilhados de forma atômica.

Quanto à memória, embora o ARM7TDMI-S não possua uma MMU com proteção de memória, o núcleo deve oferecer rotinas de alocação de memória dinâmica, como “alocar memória” e “liberar memória”.

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.

A linguagem utilizada é o C e as ferramentas são do conjunto de desenvolvimento GNU.

3.1.2 Visão geral da solução proposta

Após estudos de especificação de toda a arquitetura deste trabalho, concluiu-se ser mais viável partir para a implementação do GenosOS tendo como base um sistema operacional já implementado. Os fatores que levaram a isso foram o tempo e as características encontradas no sistema operacional de código aberto FreeRTOS (BARRY, 2006) que foi

[1]Durante todo o texto, tarefa será sinônimo de processo.

escolhido para tal propósito.

O GenosOS é uma adaptação do projeto FreeRTOS. Como já mencionado anteriormente nos trabalhos correlatos, o FreeRTOS oferece 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 referente a outros processadores foram removidos e os arquivos re-arranjados.

3.1.3 Especificação

Para desenvolver o GenosOS, inicialmente foi feito um diagrama de seqüencia[2] (fig. 3.2) para definir 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.

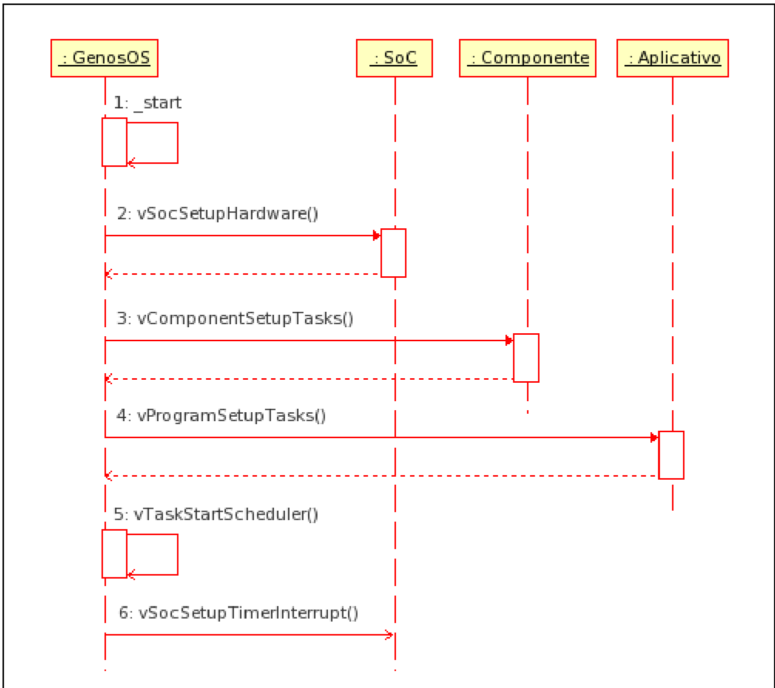


Figura 3.2 – Diagrama de seqüencia da inicialização do GenosOS

[2]O GenosOS não foi construído orientado à objetos. Os objetos do diagrama representam as partes que compõem um projeto final desenvolvido pelo usuário.

As seqüências são descritas a seguir:

a) seqüencia 1 - a função *_start* faz as inicializações do processador como definir as pilhas para cada modo de operação;

b) seqüencia 2 - em seguida é 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;

c) seqüencia 3 - no próximo passo é chamada *vComponentSetupTasks()* que 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) seqüencia 4 - o GenosOS invoca a função *vProgramSetupTasks()* que é chamada para iniciar as tarefas do aplicativo do usuário;

e) seqüencia 5 - é efetuada a chamada *vTaskStartScheduler()* que cria a tarefa *"Idle"*[3] e em seguida invoca a inicialização do escalonador;

f) seqüencia 6 - o início das atividades do escalonador é efetivado com a inicialização do relógio do sistema através da chamada *vSocSetupTimerInterrupt()*.

Conforme a fig. 3.3, foi formalizada uma estrutura de diretórios para poder separar os arquivos do SoC, componentes e programas.

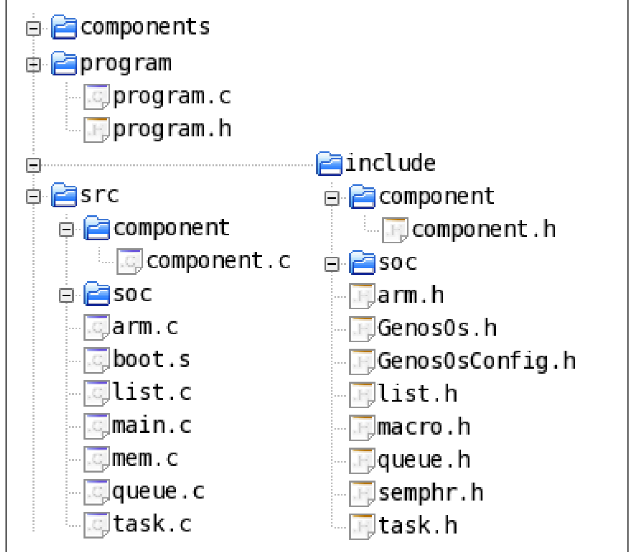


Figura 3.3 – Estrutura de diretórios do GenosOS

[3]A tarefa Idle é um processo do sistema e tem a menor prioridade.

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 2 arquivos importantes que serão gerados pelo Genos, são eles:

a) Makefile - arquivo com as definições de compilação;
b) script.ld - script do ligador com as definições de memória da placa do sistema embarcado.

3.1.4 Implementação

Esta subseção apresenta os aspectos de implementação do GenosOS. Os tópicos incluem a inicialização do sistema, criação/destruição de tarefas, estados das tarefas, alocação de memória e escalonamento.

3.1.4.1 Boot - inicialização do GenosOS

O GenosOS inicia definindo o tamanho da pilha e desabilitando as interrupções para cada modo conforme pode ser visto no quadro 3.1 (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. 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 excessão 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 fig. 3.4:

Resta ao GenosOS entrar em modo supervisor e efetuar a chamada *main()* dando

```
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
```

Quadro 3.1 – Boot do GenosOs

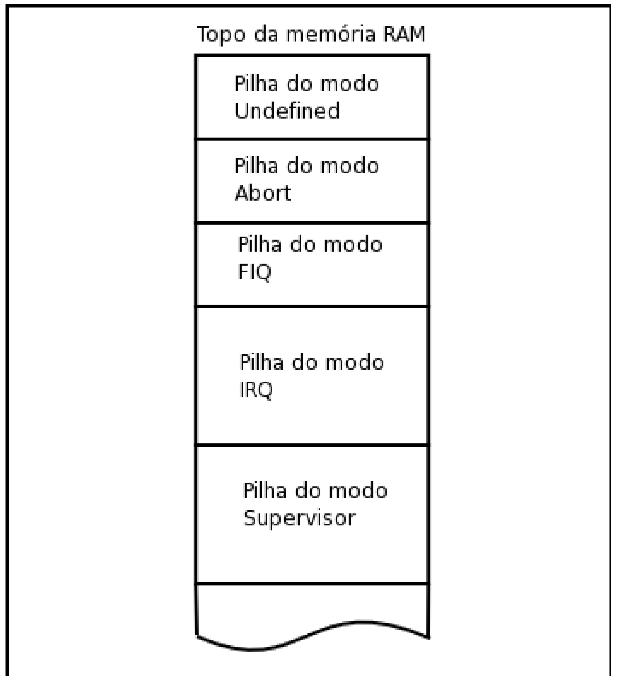


Figura 3.4 – Localização das pilhas

continuidade na inicialização (agora escrito em linguagem de programação C) e entrar em operação (quadro 3.2).

```
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
```

Quadro 3.2 – Chamada main()

3.1.4.2 Criação e remoção de tarefas

Passadas as inicializações, 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()*. Um exemplo pode ser visto no quadro 3.3.

```
1   void minhaTarefa( void * parametros )
2   {
3       // Inicializações da minha tarefa.
4
5       for(;;)
6       {
7           // Faz alguma coisa
8       }
9   }
10
11
12  xTaskCreate(   minhaTarefa,            // Código da tarefa
13                 "nomeDaMinhaTarefa",    // Nome da tarefa
14                 0x400,                  // Tamanho da pilha da tarefa
15                 NULL,                   // Parâmetros
16                 3,                      // Prioridade
17                 handle );               // usado caso queira destruir a tarefa
```

Quadro 3.3 – Exemplo de criação de uma tarefa com 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.

3.1.4.3 Estados das tarefas no núcleo do GenosOS

As tarefas podem estar em 4 estados possíveis:

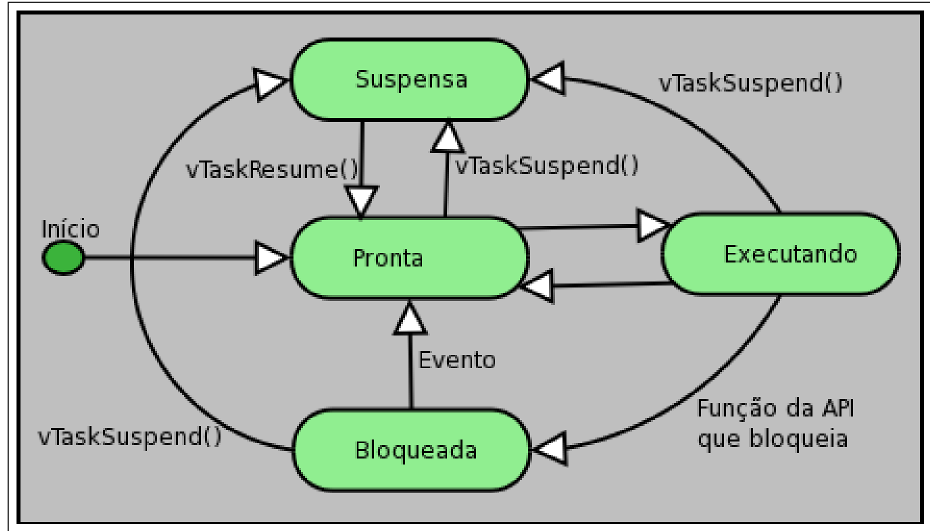
a) pronto (*ready*) - as tarefas iniciam neste estado e permanecem enquanto estiverem prontas para serem executadas;

b) executando (*running*) - este é o estado enquanto a tarefa estiver executando, ou seja, utilizando o processador;

c) bloqueada (*blocked*) - as tarefas entram neste estado quando estão esperando por algum recurso ou evento. As tarefas neste estado sempre têm um tempo limite e serão desbloqueadas quando este tempo for atingido;

d) suspensa (*suspended*) - as tarefas neste estado não são escalonadas. Uma tarefa sai do estado suspensa apenas de forma explicita com a função *vTaskResume()*.

Um diagrama com as possíveis transições entre os estados é mostrado na fig. 3.5.



Fonte: adaptado de Barry (2006).

Figura 3.5 – Possíveis transições de estados das tarefas

3.2 COMPONENTE

Um componente é uma unidade de software que exerce uma determinada tarefa. Os componentes são agregados ao GenosOS e disponibilizam serviços para as aplicações ou outros componentes. O objetivo do componente é modularizar e extender as funcionalidades do sistema. A extensão do sistema com o uso dos componentes, provê ainda uma estratégia de reutilização de software formal.

3.2.1 Requisitos principais

Os componentes deverão conter 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.

Um componente poderá conter configurações atribuidas 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 afim de prover instruções de utilização do componente para o programador que vier a fazer

uso.

Os componentes deverão ser escritos na linguagem de programação C.

3.2.2 Visão geral da solução proposta

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, o nome 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 requeridos.

3.2.3 Especificação

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 do quadro 3.4 e descrita a seguir.

```
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
```

Quadro 3.4 – 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*).

Quanto à documentação dos componentes, é utilizada a ferramenta Doxygen (HEESCH, 2006) que se baseia em marcações inseridas pelo desenvolvedor em meio ao código fonte. Após escrever a documentação direto na criação do código fonte, executa-se o Doxygen que procura nos arquivos fontes as marcações e gera uma documentação em HTML. Um exemplo das marcações do Doxygen pode ser visto no quadro 3.5 e o resultado após a geração na fig. 3.6.

3.2.4 Implementação

Nesta subseção será apresentada a implementação de 2 componentes. Com este estudo de caso, será apresentada a técnica de normalização, que permite a um compo-

```
1  /**
2        \brief Envia um caracter pela porta serial
3        \param byte - Caracter a ser enviado
4        \return Sem retorno
5  */
6  void sendChar(char byte);
```

Quadro 3.5 – Exemplo de uma função documentada com o Doxygen

Funções

void sendChar (char *byte*)

Envia um caracter pela porta serial.

Parâmetros:
byte - Caracter a ser enviado

Retorna:
Sem retorno

Figura 3.6 – Resultado de uma função documentada com o Doxygen

nente 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 no quadro 3.6. 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 comp_uartInit(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

```
 1 [General]
 2 Arch=Arm7
 3 Author=Filipe
 4 Description=driver serial para o lpc2106
 5 Group=serial
 6 Name=comp_uart
 7 ShortDescription=driver serial
 8 Soc=lpc2106
 9 Version=1.0
10
11 [Api]
12 api0=void sendString(char *str, int length)
13 api1=void sendChar(char byte)
14 api2=char getChar(void)
15 api3=void comp_uartInit(void)
16
17 [ArmFiles]
18 file0=comp_uart.c
19
20 [HeaderFiles]
21 file0=comp_uart.h
```

Quadro 3.6 – Definição do componente comp-uart

comp-uart. No quadro 3.7 pode ser visto sua descrição.

```
 1 [General]
 2 Arch=Arm7
 3 Author=Filipe
 4 Description=Este componente manda mensagens através de outro componente.
 5 Group=software
 6 Name=mensageiro
 7 ShortDescription=Componente que manda mensagens
 8 Soc=*
 9 Version=1.0
10
11 [Api]
12 api0=sendMsg(char *msg)
13
14 [Api-dependent]
15 api0=sendPorOutroComp(char *msg)
16
17 [ArmFiles]
18 file0=mensageiro.c
19
20 [HeaderFiles]
21 file0=mensageiro.h
```

Quadro 3.7 – Definição do componente mensageiro

Ao adicionar-se os componentes no projeto, ambos são copiados para os diretórios indicados na seção que apresenta o GenosOS (página 30).

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:

a) $N \geq 0$ e corresponde à mesma entrada na seção *Api-dependent*;

b) *parâmetro1* é o nome do componente que está sendo utilizado;

c) *parâmetro2* é a função que está sendo utilizada do componente do *parâmetro1*;

d) *parâmetro3* é o formato da chamada que será utilizada pelo componente requisitante (para este exemplo, o mensageiro).

Os nomes dos argumentos serão sempre arg$N$ onde $N \geq 0$.

O programador deve adequar os parâmetros na função a ser utilizada. 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*[4] 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.

A fig. 3.7 apresenta de forma gráfica o relacionamento de todas as partes no exemplo do componente mensageiro utilizando a interface oferecida do componente comp-uart.

[4]Custo adicional em processamento que tem como conseqüência a perda de performance.

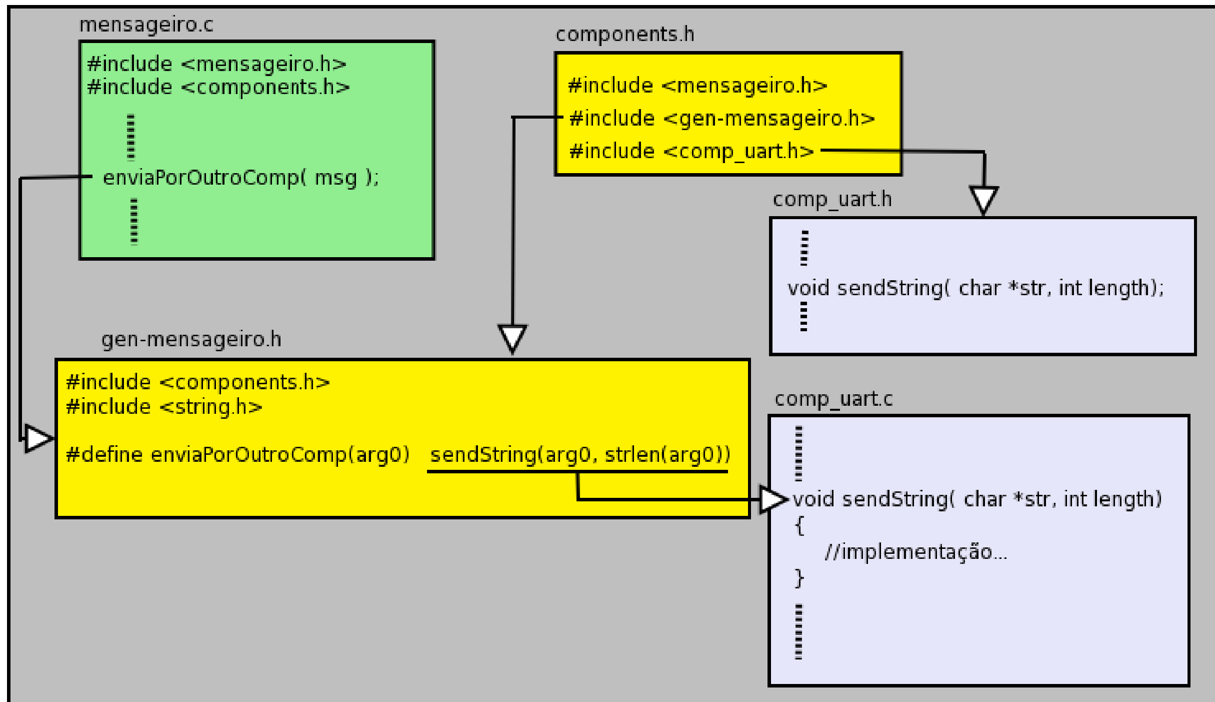


Figura 3.7 – Exemplo da técnica de normalização

3.2.5 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:

a) *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;

b) *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. Um exemplo é mostrado no quadro 3.8 com um trecho da definição dos endereços dos registradores da *GPIO*.

Assim como existe o arquivo *component.conf* que especifica um componente, para um componente de SoC existe o *soc.conf*. Um exemplo de especificação de um SoC é

mostrado no quadro 3.9.

```
1   #define GPIO0_IOPIN     (*(REG32 (0xE0028000)))
2   #define GPIO0_IOSET     (*(REG32 (0xE0028004)))
3   #define GPIO0_IODIR     (*(REG32 (0xE0028008)))
4   #define GPIO0_IOCLR     (*(REG32 (0xE002800C)))
```

Quadro 3.8 – Definição da GPIO.

```
1 [General]
2 Arch=Arm7
3 Author=Filipe Renaldi
4 FilesC=soc.c
5 FilesH=soc.h
6 Model=lpc2106
7 Name=soc-lpc2106
8 Version=1.0
```

Quadro 3.9 – Especificação de um componente de SoC

A criação e edição de um componente de SoC no Genos são feitas utilizando-se um módulo específico.

3.3 AMBIENTE DE DESENVOLVIMENTO - GENOS

O ambiente de desenvolvimento, ou simplesmente Genos, é onde todas as tecnologias apresentadas até agora são agregadas e oferecidas ao desenvolvedor. O Genos é em essência, a área de trabalho do programador.

3.3.1 Requisitos principais

O sistema deverá prover um ambiente de desenvolvimento para que o usuário possa interagir com o GenosOS e seus componentes. Para tanto, haverá um módulo de criação e edição de SoC e um módulo de criação e edição de componentes. O Genos deverá gerenciar a criação e edição de um projeto, e para isso, o usuário poderá criar um novo projeto, adicionar componente a partir de uma paleta e construir o projeto. A construção do projeto envolve a geração dos arquivos necessários para a normalização dos componentes com interface requerida, a geração do arquivo Makefile para compilação, a geração do arquivo script.ld para a ligação, a compilação propriamente dita, a geração da documentação dos componentes e GenosOS e a programação do binário resultante no dispositivo eletrônico.

Quanto ao projeto, o usuário deverá informar a memória disponível na placa do sis-

tema embarcado para o qual está sendo desenvolvido, as ferramentas de desenvolvimento (compilação, depuração e gravação) e os parâmetros das ferramentas. O usuário poderá ainda criar arquivos de código fonte para desenvolver as aplicações do sistema utilizando um editor de arquivos fontes integrado ao próprio Genos. Os componentes do projeto deverão estar visíveis bem como os arquivos fontes dos componentes que o usuário poderá abrir para ler ou editar. O sistema deverá prover ainda uma tela de normalização de interfaces requeridas.

Todo o software foi desenvolvido utilizando a linguagem C++ com a biblioteca de interface gráfica QT (TROLLTECH, 2006).

3.3.2 Visão geral da solução proposta

O software foi desenvolvido buscando a integração de todas as funcionalidades em uma só janela principal. Para manter os módulos organizados, foi utilizada a navegação por abas. Para a paleta de componentes, foi utilizada uma janela móvel dentro da janela principal que pode ser posicionada onde o usuário escolher. Da mesma forma estão as áreas de componentes do projeto e arquivos dos componentes. Foi adicionada ainda uma quarta janela anexa com informações de eventos que estão acontecendo durante o uso do software.

3.3.3 Especificação

A seguir são apresentados os diagramas de caso de uso das principais implemementações.

Na fig. 3.8 está a configuração dos diretórios de projetos onde o sistema apresenta uma tela de configuração para o usuário informar o local padrão para criar projetos, o diretório com os SoCs disponíveis ao usuários e o diretório dos componentes do sistema.

A fig. 3.9 apresenta a criação do SoC. O sistema exibe uma tela para o usuário informar: nome, arquitetura, modelo, autor versão, arquivos ".c", arquivos ".h"e local para salvar o novo componente.

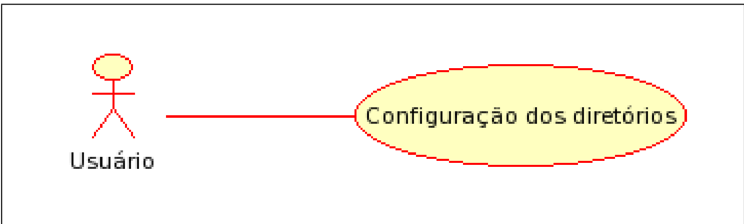


Figura 3.8 – Configuração dos diretórios

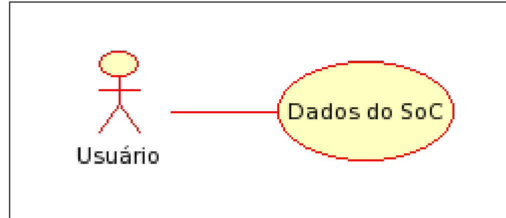


Figura 3.9 – Criação do SoC

Na fig. 3.10 está a criação de um componente sendo que o sistema apresenta uma tela com 4 abas para que o usuário informe:

a) informações (aba 1): nome, arquitetura, SoC, autor, versão, grupo, descrição e descrição completa;

b) arquivos do componente (aba 2): arquivos de cabeçalho ".h", arquivos fontes ".c"Thumb e Arm mode separadamente;

c) api (aba 3): funções oferecidas e funções requeridas;

d) configurações (aba 4): nome, tipo, valor padrão e descrição.

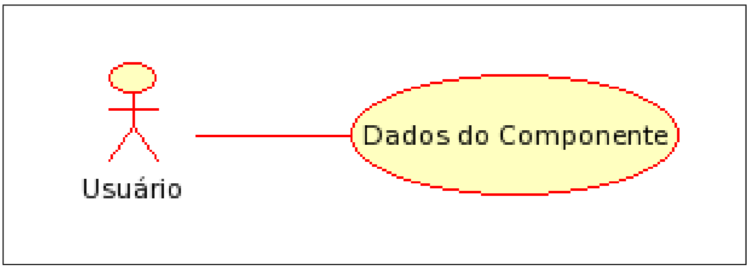


Figura 3.10 – Criação do componente

A criação de um projeto é mostrado na fig. 3.11. O sistema apresenta uma tela para o usuário informar: nome do projeto e diretório. O sistema então faz uma cópia do GenoOS para o diretório informado pelo usuário. Em seguida o sistema apresenta uma tela para o usuário informar o SoC a ser utilizado no projeto. O sistema agora copia os arquivos do SoC para o diretório do projeto. Por último, o sistema apresenta a tela de edição do projeto.

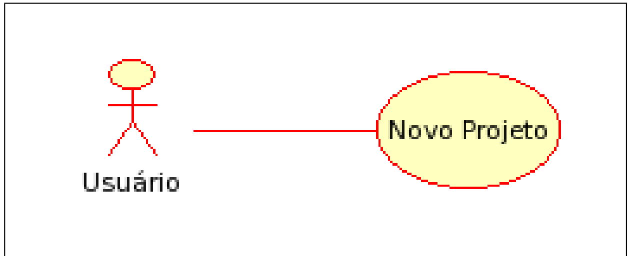


Figura 3.11 – Novo projeto

O caso de uso da normalização está na fig. 3.12, e o cenário é descrito a seguir:

a) sistema apresenta uma tela com as funções requeridas do componente selecionado;
b) usuário seleciona com duplo-clique a função a ser normalizada;
c) sistema apresenta uma tela com todos os componentes;
d) usuário seleciona o componente do qual será utilizado uma interface oferecida;
e) sistema apresenta todas as funções oferecidas do componente selecionado;
f) usuário seleciona com duplo-clique a função desejada;
g) sistema disponibiliza uma área de edição para o usuário adequar os parâmetros;
h) usuário faz as edições desejadas e confirma clicando no botão "Ok";
i) sistema edita o arquivo de definição do componente com as opções do usuário.

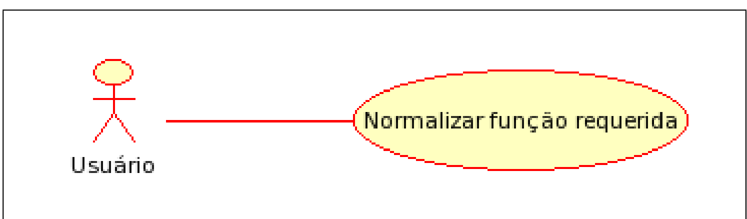


Figura 3.12 – Processo de normalização

Para a seleção de componentes (fig. 3.13) o sistema apresenta a paleta de componentes. Usuário então seleiona o componente desejado e clica no botão "adicionar". Sistema copia os arquivos do componente para o diretório do projeto e adiciona uma entrada na área de componentes do projeto.

O último caso de uso define a criação de arquivos de programas e pode ser visto na fig. 3.14. O sistema apresenta uma tela para que o usuário informe o nome do arquivo e em seguida salva o nome do arquivo no arquivo de definições do projeto. Ele então adiciona

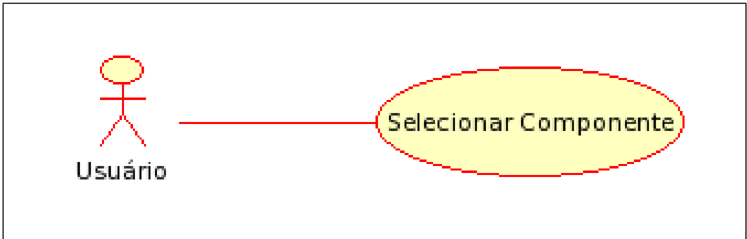


Figura 3.13 – Seleção de componentes

o nome em uma lista visível ao usuário. O usuário seleciona o arquivo com duplo-clique. O sistema então abre um editor com o arquivo para que o usuário edite-o.

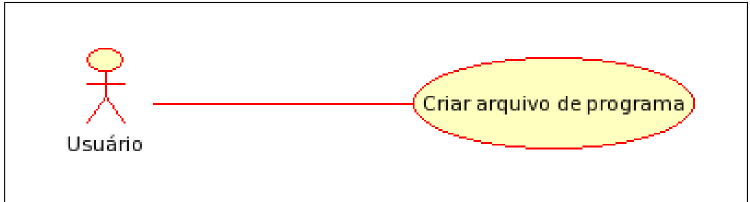


Figura 3.14 – Criação dos arquivos de programas

A fig. 3.15 apresenta o diagrama de seqüência bem como a descrição do processo de construção de um projeto.

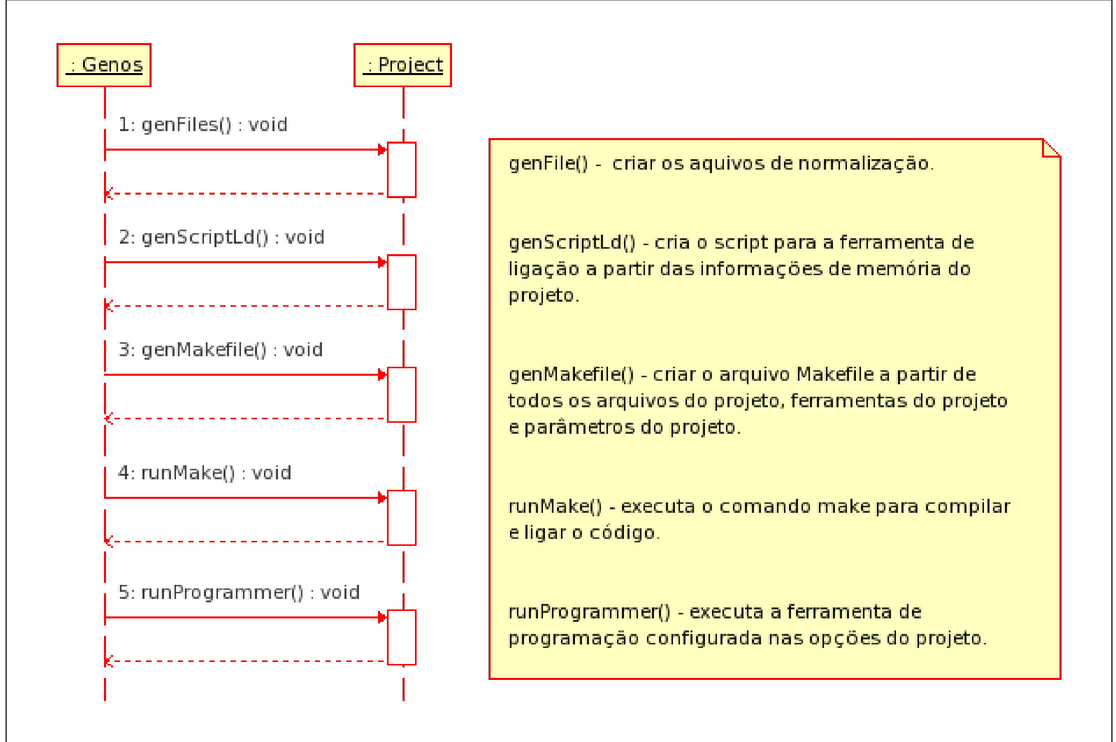


Figura 3.15 – Diagrama de sequência do processo de construção do projeto

Na fig. 3.16 é apresentado o diagrama de classes do Genos. As interfaces laranjas referem-se às classes de interface com o usuário e são construídas pelo editor de interfaces

da biblioteca QT chamado Designer, sendo que são salvas em *eXtensible Markup Language* (XML) e transformadas em código C++ em tempo de compilação de todo o sistema pela ferramenta Uic que faz parte da biblioteca QT. As classes que compõem a interface com o usuário como botões, menus, etc foram omitidos.

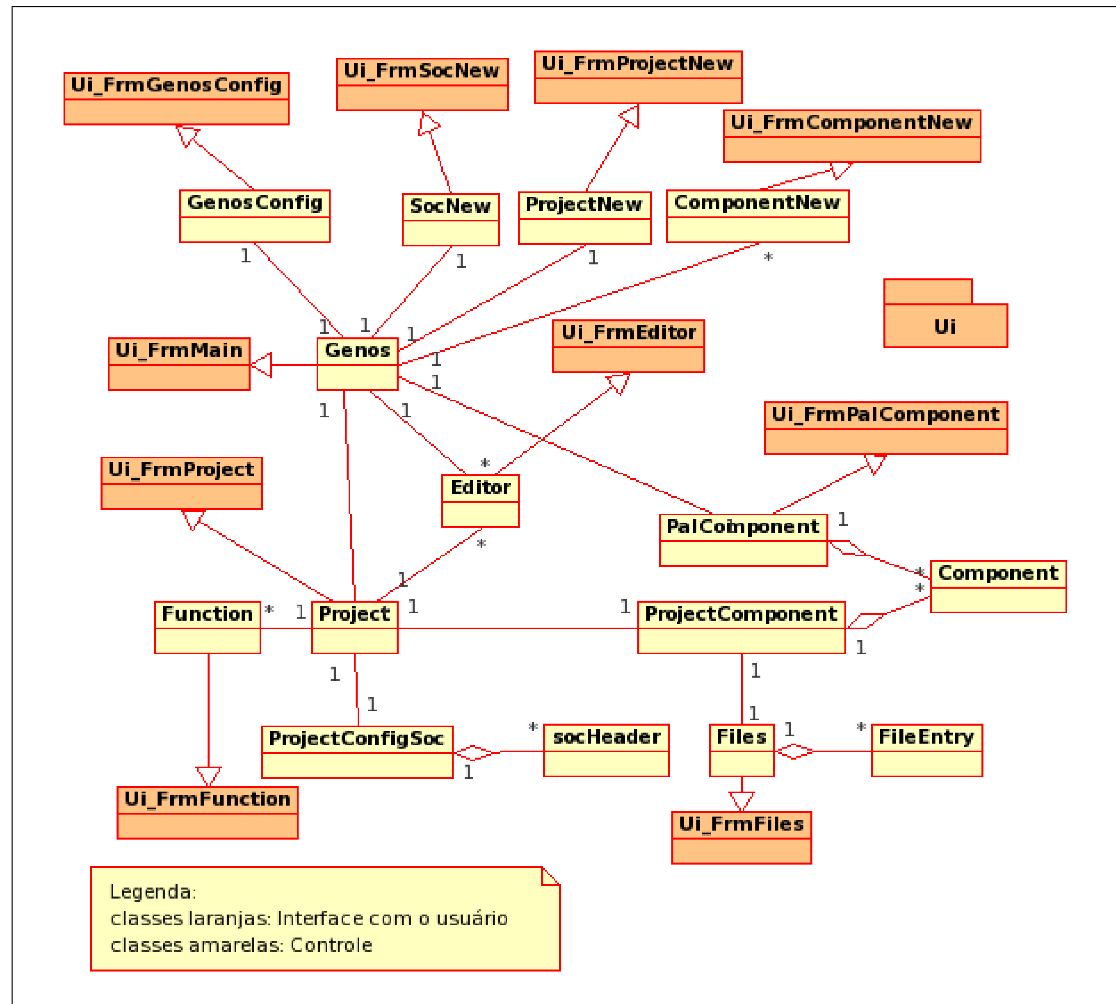


Figura 3.16 – Diagrama de classes do Genos

As classes *Project* e *Genos* são ainda mostradas em detalhes, respectivamente, na fig. 3.17 e fig. 3.18.

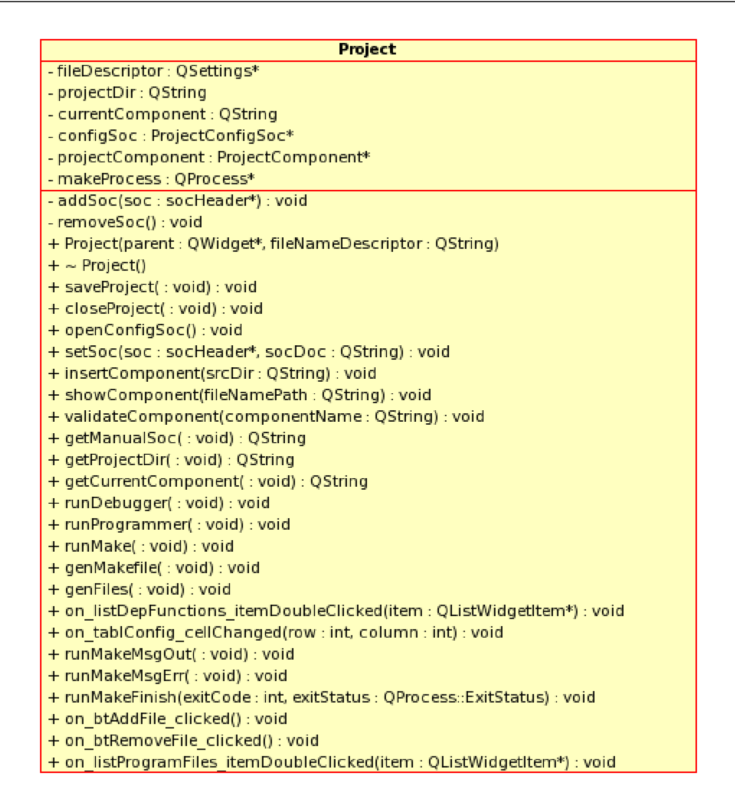


Figura 3.17 – Detalhes da classe Project

Genos
- config : GenosConfig*
- edLog : QTextEdit*
- project : Project*
- projectNew : ProjectNew*
- palComponent : PalComponent*
- projectComponent : ProjectComponent*
- socNew : SocNew*
- setupDocks(: void) : void
+ Genos()
+ ~ Genos()
+ addTab(widget : QWidget*, icon : QIcon, caption : QString) : void
+ removeTab(widget : QWidget*) : void
+ addLog(message : QString, type : LogType) : void
+ getSocDir(: void) : QString
+ getProjectDir(: void) : QString
+ getComponentDir(: void) : QString
+ currentProject(: void) : Project*
+ getPallete(: void) : PalComponent*
+ getProjectComponent(: void) : ProjectComponent*
+ getFormFiles(: void) : Files*
+ openProject(fileDescriptor : QString) : void
+ scanComponent(: void) : void
+ openEditor(filename : QString) : void
+ copyDir(src : QString, dest : QString) : bool
# on_actFileNew_triggered(: void) : void
# on_actOpenFile_triggered(: void) : void
# on_actProjectNew_triggered(: void) : void
# on_actProjectOpen_triggered(: void) : void
# on_actProjectClose_triggered(: void) : void
# on_actProjectSave_triggered(: void) : void
# on_actGenFiles_triggered(: void) : void
# on_actGenMakefile_triggered(: void) : void
# on_actMake_triggered(: void) : void
# on_actProgram_triggered(: void) : void
# on_actDebug_triggered(: void) : void
# on_actGenosConfig_triggered(: void) : void
# on_actSocNew_triggered(: void) : void
# on_actComponentNew_triggered(: void) : void
# on_actManualSoc_triggered(: void) : void
# on_actAbout_triggered(: void) : void
# on_actQtAbout_triggered(: void) : void
+ on_actConfigSoc_triggered(: void) : void

Figura 3.18 – Detalhes da classe Genos

3.3.4 Implementação

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.

Uma imagem do aplicativo sendo executado é apresentado na fig. 3.19 para uma melhor compreensão do texto a seguir.

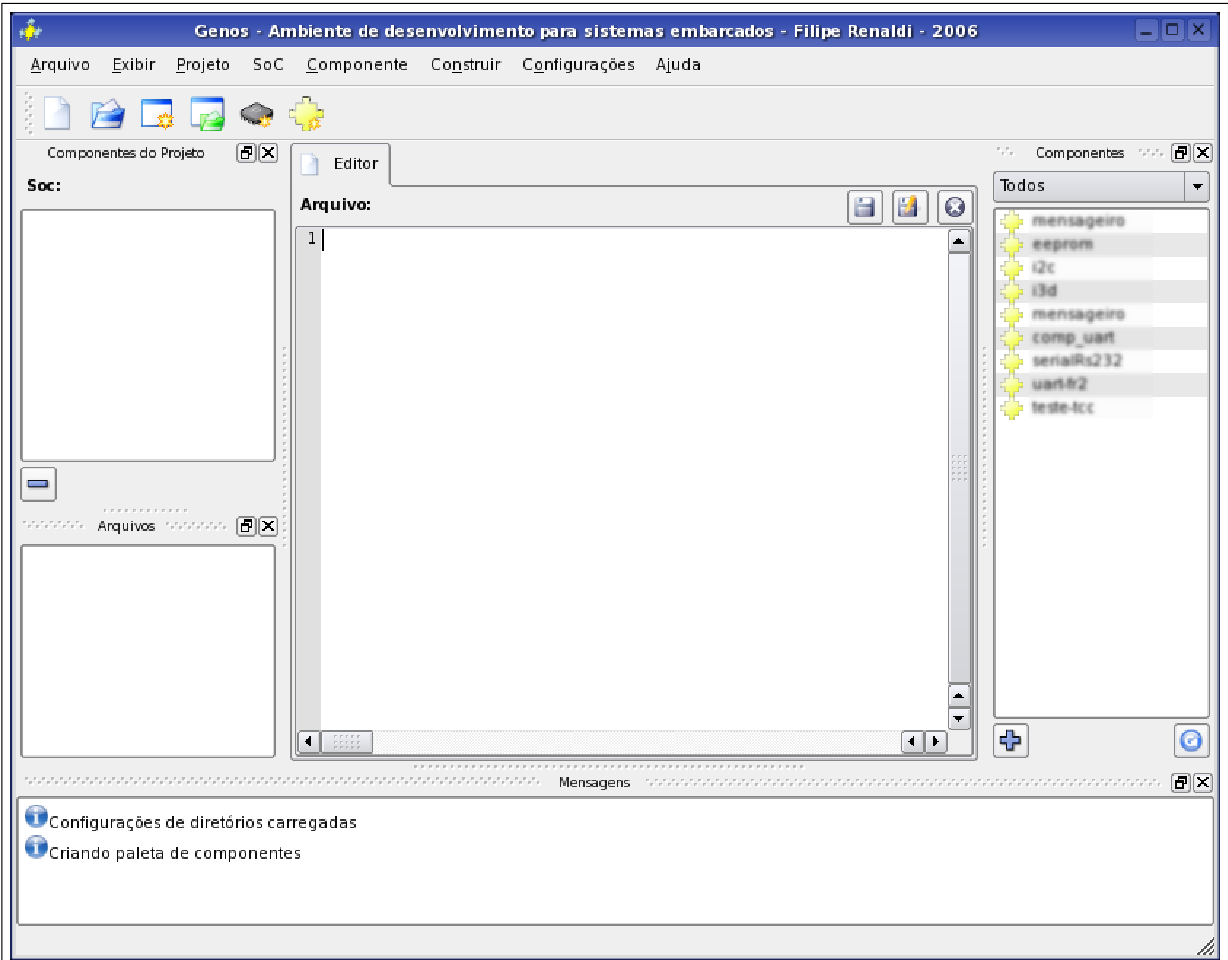


Figura 3.19 – Genos sendo executado

O sistema inicia com a instância da classe *Genos*, que pode ser vista em detalhes na fig. 3.18, e é a classe principal do sistema. O quadro 3.10 apresenta o método construtor. O método inicia construindo a interface com a chamada *setupUi()* que constrói a janela principal do Genos que logo após é maximizada. A chamada da linha 8 remove a aba padrão que o Designer insere ao utilizar o componente *QTabWidget* que é utilizado como componente central da janela. Após algumas inicializações, a classe *Genos* invoca

o método *setupDocks()* que constrói as janelas anexas: Mensagens, Componentes, Componentes do Projeto e Arquivos (fig. 3.19). A chamada da linha 17 torna a interface com o usuário visível. Na linha 19 é instanciado um objeto da classe *GenosConfig* que carrega as configurações que são: últimas posições das janelas anexas e diretórios de projeto, SoC e componentes. O construtor finaliza com a criação da paleta de componentes onde primeiro é feita a busca dos componentes e em seguida exibida na paleta.

```
1   Genos::Genos()
2       : QMainWindow(0)
3   {
4       setupUi(this);
5       this->showMaximized();
6
7       // Remove the Designer's default tab
8       tbCenter->removeTab(0);
9
10      project = 0;
11      socNew = 0;
12      projectNew = 0;
13
14      // Setup docks
15      setupDocks();
16
17      this->show();
18
19      config = new GenosConfig(this);
20
21      //Scanning components...
22      scanComponent();
23      palComponent->getList()->show();
24  }
```

Quadro 3.10 – Construtor da classe Genos

As tarefas (criar componente, criar SoC, criar projeto, criar novo arquivo de fonte e configurar Genos) executadas no ambiente de desenvolvimento são manipuladas de forma semelhante. Uma tarefa inicia com uma ação[5] do usuário. A seguir é mostrado um exemplo com a tarefa "criar componente".

Ao selecionar o botão (ou entrada no menu) "Novo Componente", a classe *Genos* trata o evento com o método *on_actComponentNew_triggered()* que pode ser visto no quadro 3.11. Um objeto da classe *ComponentNew* é instanciado e adicionado à uma aba do Genos. A classe *ComponentNew* assim como as classes *SocNew, ProjectNew, Editor* e

[5]Uma ação pode ser um botão na barra de ferramentas ou uma entrada no menu. Uma mesma ação pode ainda estar nos dois locais.

Project, inicia executando em seu método construtor a criação da interface com o método *setupUi()*.

```
1    void Genos::on_actComponentNew_triggered(void)
2    {
3        addTab( new ComponentNew(this),
4                QIcon(":/icons/img/component-new.png"),
5                QString("Componente") );
6    }
```

Quadro 3.11 – Início da tarefa Novo Componente

Após a entrada de dados por parte do usuário, as tarefas acabam por manipular arquivos. Dando continuidade ao exemplo do "Novo Componente", assim que o usuário clicar no botão "Criar" presente na tela, as informações que compõem o novo componente são gravadas em um arquivo de definição (apresentado na seção componentes, página 36). A manipulação deste arquivo, assim como todos os arquivos com a finalidade de prover configuração ou definição, são efetuados utilizando-se a classe *QSettings* provida pela biblioteca QT. O quadro 3.12 apresenta alguns trechos utilizando a classe *QSettings*. Inicialmente é criado um objeto de *QSettings* (linha 5). Os dados são salvos em tuplas com um identificador e um valor. Pode-se ainda criar seções dentro do arquivo. Conforme a linha 14, a seção "Api" agrega as funções que compõem a *application programming interface* (API) do componente que está sendo criado. Pode ser visto ainda, na linha 25 o método *addLog()* da classe *Genos* que adiciona mensagens na janela anexa "Mensagens" da janela principal. Ao final de uma tarefa, a aba é removida com o método *removeTab()* (linha 26) da classe *Genos*.

A classe *ProjectNew* é responsável por criar um novo projeto e segue um fluxo semelhante ao descrito para a classe *ComponentNew*. A principal diferença encontra-se no método que atende ao evento do clique no botão "Criar". Conforme o quadro 3.13, o método inicia verificando se o nome do projeto foi informado. Na linha 12, é feita a cópia do GenosOS para o diretório do novo projeto que a função Genos::copyDir() trata de criar. Um arquivo de configuração é criado com o nome do projeto seguido da extensão ".genos" (linhas 15 e 16). Em seguida são inicializados com valores padrões as ferramentas (compilador, ligador, etc) que o projeto utilizará (linha 21 em diante) e depois

```
1   void ComponentNew::on_btOk_clicked()
2   {
3       (...)
4
5       QSettings compConf( edDir->text()+"/component.conf", QSettings::IniFormat, this );
6
7       // [General]
8       compConf.setValue("Name", edName->text());
9       compConf.setValue("Arch", edArch->text());
10      (...)
11      compConf.setValue("Description", edDescription->toPlainText());
12
13      // [Api]
14      compConf.beginGroup("Api");
15      for (i=0; i<listFunc->count(); i++)
16      {
17          key = QString("api%1").arg(i);
18          value = listFunc->item(i)->text();
19          compConf.setValue(key, value);
20      }
21      compConf.endGroup();
22
23      (...)
24
25      genos->addLog("Componente criado", Genos::Info );
26      genos->removeTab( this );
27      close();
28  }
```

Quadro 3.12 – Criando arquivo de definição com *QSettings*

os parâmetros padrões (linha 27 em diante). O método finaliza abrindo o novo projeto criado, enviando uma mensagem de confirmação que o projeto foi criado e removendo a aba da tela “Novo Projeto”.

```
1   void ProjectNew::on_btCreate_clicked()
2   {
3       if ( edName->text().isEmpty() )
4       {
5           genos->addLog( "Coloque um nome no projeto", Genos::Error );
6           return;
7       }
8
9       QString projectDir = edDir->text() +"/" +edName->text() + "/";
10
11      // copy the base GenosOS
12      Genos::copyDir( GenosOSDir, projectDir  );
13
14      // Create the project file descriptor
15      QString projectFile( projectDir + edName->text() + ".genos" );
16      QSettings conf( projectFile, QSettings::IniFormat, this );
17      conf.clear();
18      conf.setValue( "Name", edName->text() );
19
20      // Saving default tools settings
21      conf.beginGroup("Tools");
22      conf.setValue( "Compiler", "arm-unknown-linux-gnu-gcc" );
23      (...)
24      conf.endGroup();
25
26      // Saving default Parameters settings
27      conf.beginGroup("Parameters");
28      conf.setValue( "Cpu", "arm7tdmi" );
29      (...)
30
31      conf.sync();
32
33      //Open the new project
34      genos->openProject( projectFile );
35
36      genos->addLog( "Projeto criado com sucesso", Genos::Info);
37      genos->removeTab( this );
38      close();
39  }
```

Quadro 3.13 – Método da classe ProjectNew que cria um novo projeto.

3.4 UTILIZAÇÃO DO SISTEMA

Esta seção aborda a utilização do sistema apresentando a criação de um projeto que envolve:

a) criação de um SoC;

b) criação de dois componentes, sendo que um deles tem interfaces requeridas para ilustrar o processo de normalização através do Genos;

c) criação do projeto;

d) criação do aplicativo;

e) construção de todo o conjunto;

f) gravação do binário no hardware do sistema embarcado.

O equipamento a ser utilizado é a placa LPC-P2106 da Olimex (fig. 2.2 da página 23).

Ao iniciar o sistema, a tela da fig. 3.20 será exibida.

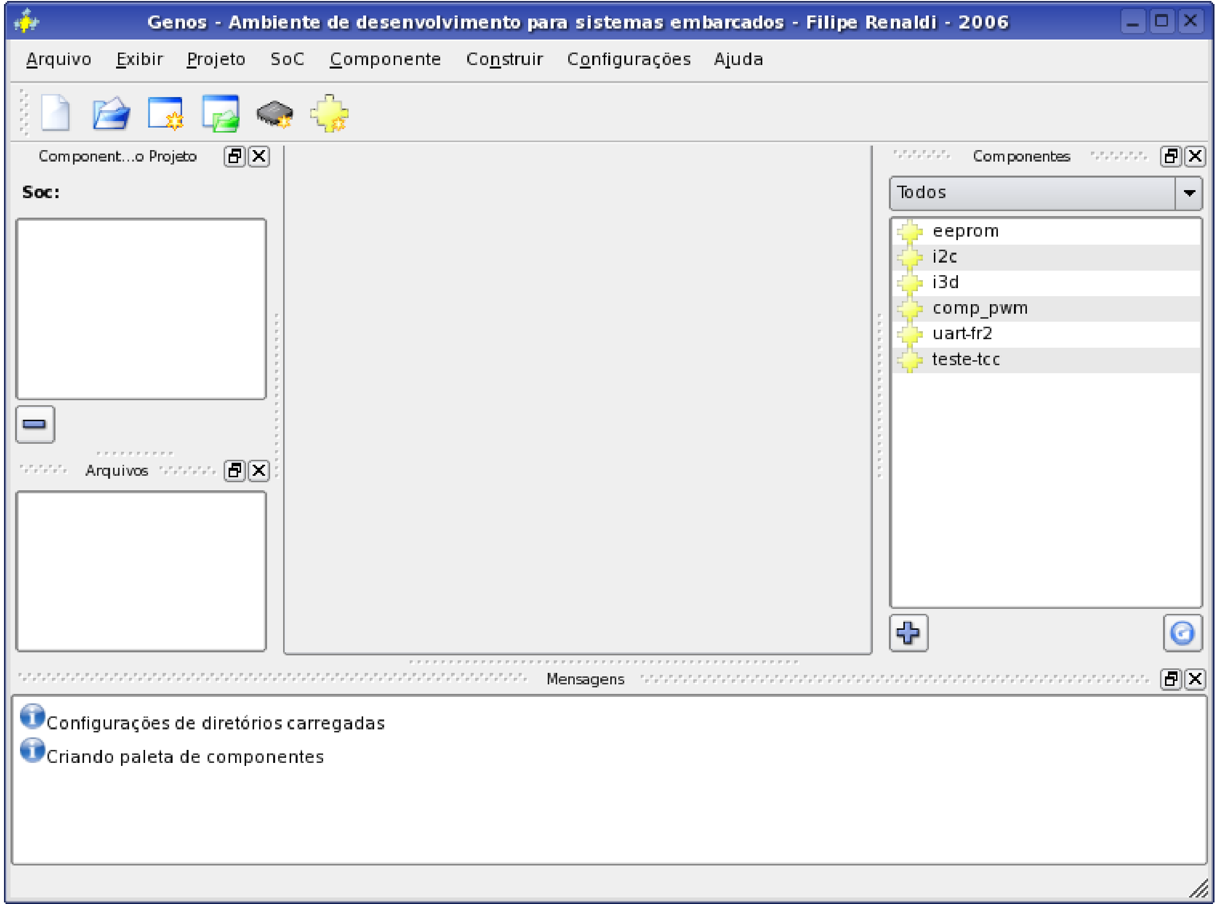


Figura 3.20 – Genos

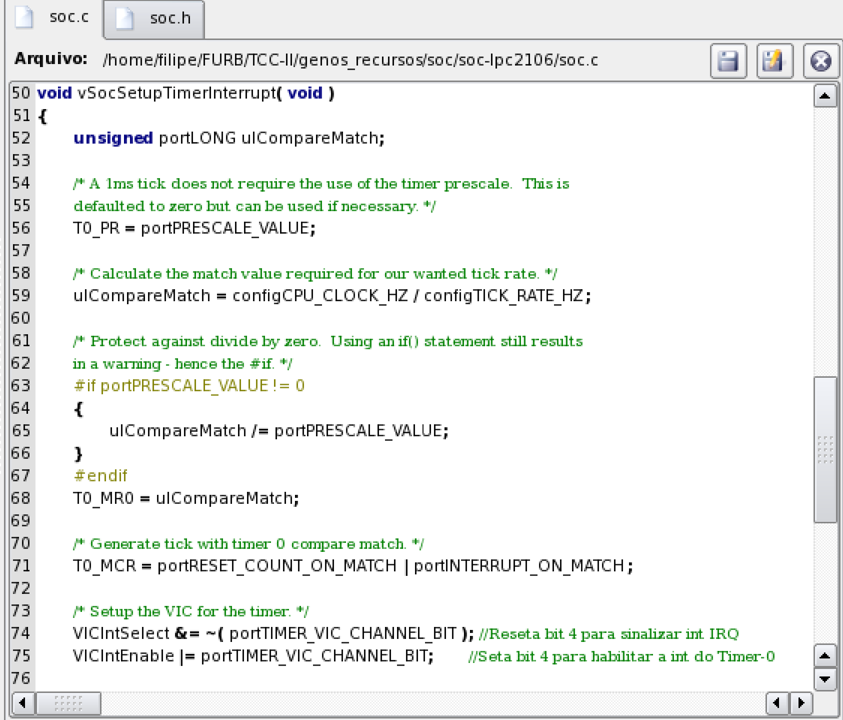


Figura 3.21 – Edição do arquivo soc.c

Para criar um componente de SoC, o usuário deve inicialmente criar os arquivos soc.h e soc.c contendo as funções e definições descritas na seção Componentes (página 41). Para utilizar o editor do Genos, basta selecionar a opção no menu "Arquivo / Novo Arquivo". A fig. 3.21 mostra a duas abas de edição (soc.c e soc.h). Em seguida, o usuário seleciona a opção no menu "SoC / Novo Soc". Após informar todos os dados, o usuário pressiona o botão "Criar" e o componente de SoC estará concluído (fig. 3.22).

soc.c | soc.h | Novo SoC

Novo SoC

Nome: soc-lpc2106
Arquitetura: Arm7
Modelo: lpc2106
Autor: Filipe Renaldi
Versão: 1.0
Arquivos (.h): soc.h
Arquivos (.c): soc.c
Salvar em: /home/filipe/FURB/TCC-II/genos_recursos/soc/soc-lpc2106/

Criar | Cancelar

Figura 3.22 – Tela de criação do componente de SoC

O próximo passo é criar os componentes. Primeiro é criado o componente "comp-uart" que provê funções para enviar e receber dados pela porta serial do SoC LPC2106. Após editar os arquivos fontes, o usuário deve selecionar a opção no menu "Componente / Novo Componente". Após informar todos os dados, o usuário pressiona o botão "Criar" e o componente estará concluído (fig. 3.23).

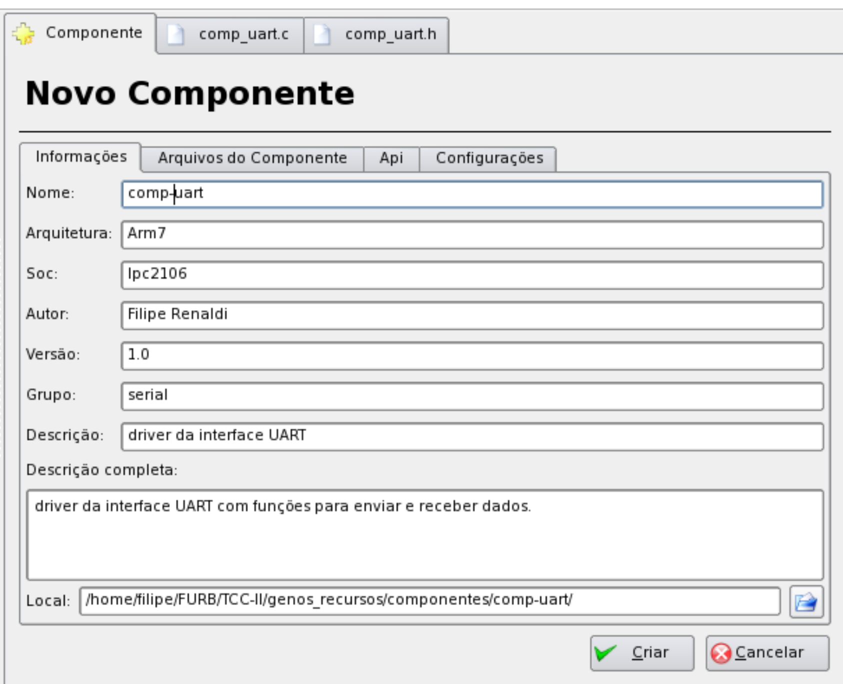


Figura 3.23 – Tela de criação do componente

O mesmo processo é seguido para criar o componente mensageiro. Após a criação dos componentes, basta pressionar o botão "Atualizar" da paleta de componentes para que ambos estejam disponíveis na paleta conforme pode ser visto na fig. 3.24.

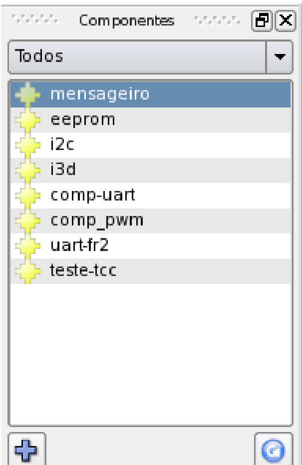


Figura 3.24 – Paleta de componentes

A seguir o projeto é criado. Para isso, o usuário seleciona a opção no menu "Projeto / Novo Projeto". A tela da fig. 3.25 será exibida para que o usuário informe o nome e local do projeto.

Figura 3.25 – Novo projeto

Ao pressionar o botão "Criar", o Genos abre o novo projeto para edição e o módulo de seleção do SoC onde o usuário seleciona o SoC e informa o manual do fabricante (fig. 3.26). Imediatamente aparecerá na parte superior da janela anexa "Componentes do Projeto" o nome do SoC utilizado.

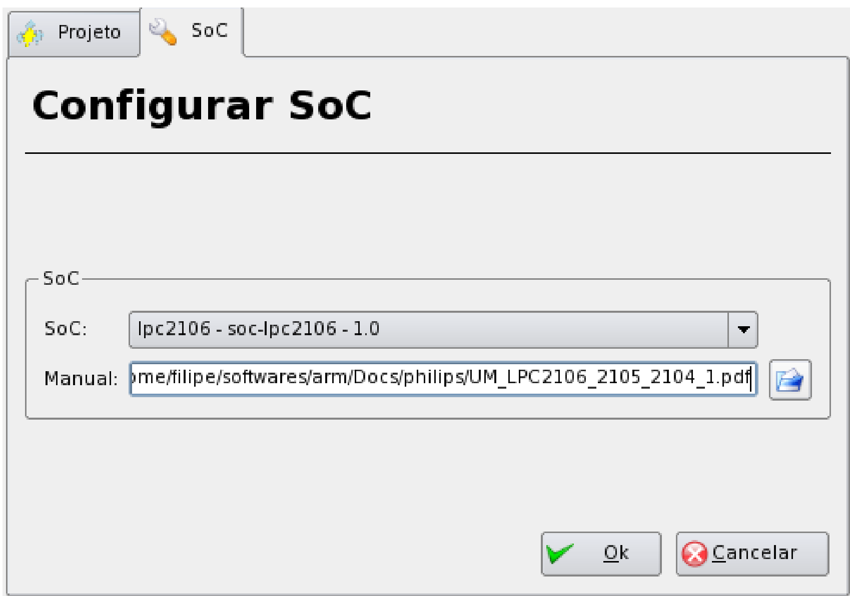


Figura 3.26 – Seleção do SoC

O novo projeto agora está pronto para ser editado. Inicialmente é adicionado os componentes comp-uart e mensageiro ao projeto. Para adicionar um componente da

paleta, basta selecioná-lo e em seguida pressionar o botão com o símbolo "+" (mais) na parte inferior esquerda da paleta.

Ao selecionar um componente do projeto, os arquivos do mesmo são mostrados na janela anexa "Arquivos" e suas demais informações na área de edição do projeto, conforme pode ser visto na fig. 3.27. O arquivo pode ser aberto com um duplo clique sobre o nome dele.

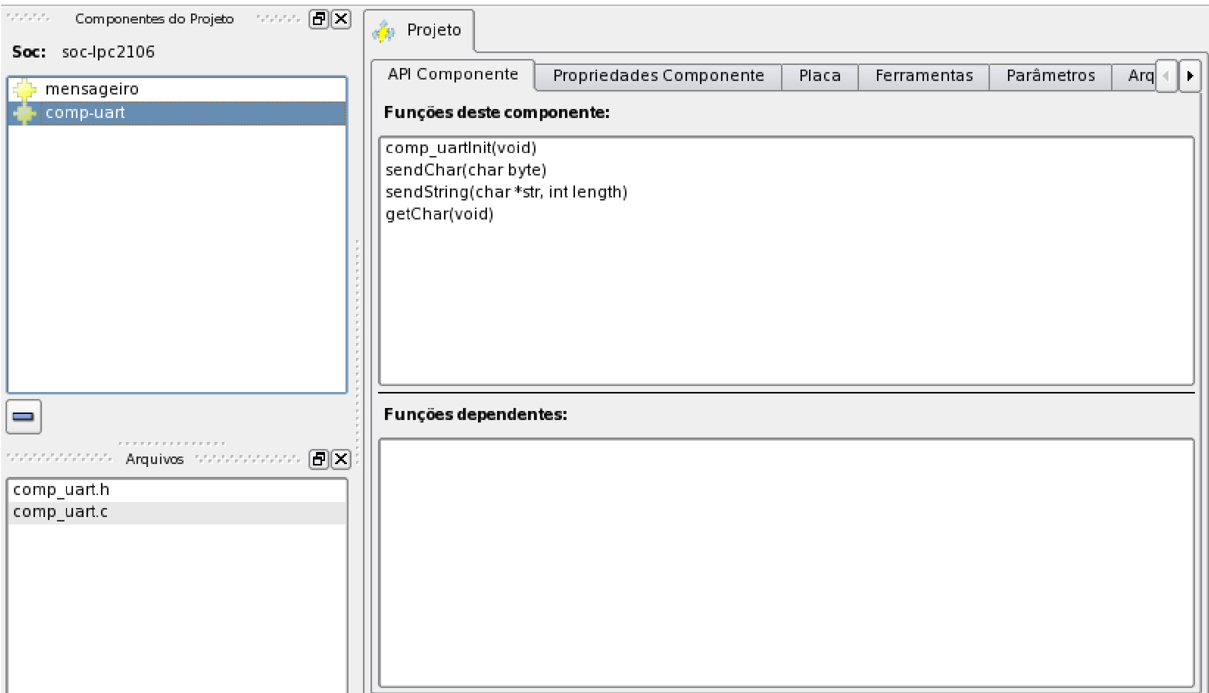


Figura 3.27 – Edição do projeto

O usuário agora deve fazer a normalização das funções requeridas. Para tanto, basta selecionar com duplo clique sobre a função requerida na área chamada "Funções dependentes" e o sistema irá exibir uma tela de normalização da função selecionada. Inicialmente o usuário seleciona o componente desejado sendo que será exibida a lista de funções que o componente oferece. Em seguida o usuário seleciona uma das funções. O sistema copia a mesma para uma área de edição onde o usuário possa adequar os parâmetros. A fig. 3.28 apresenta a tela de normalização onde a função *sendMsg()* do componente mensageiro está sendo normalizada.

Para os componentes com configurações, o usuário deve informar os dados desejados. Para isso, na aba "Propriedades Componente", o usuário edita os campos da tabela

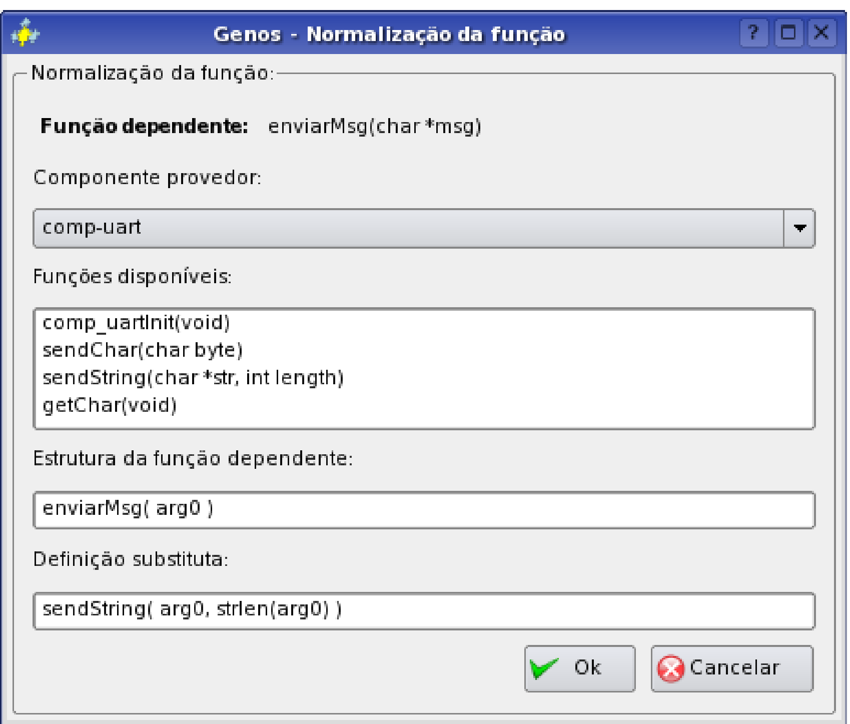


Figura 3.28 – Normalização da função *enviarMsg()*

que contém as configurações. Na fig. 3.29 pode ser vista a configuração dos pinos da interface serial do componente comp-uart.

Após efetuar uma normalização ou configuração, é importante que o usuário selecione a opção no menu "Construir / Gerar Arquivos". Opcionalmente, poderá ser feito apenas uma vez ao final de toda a edição.

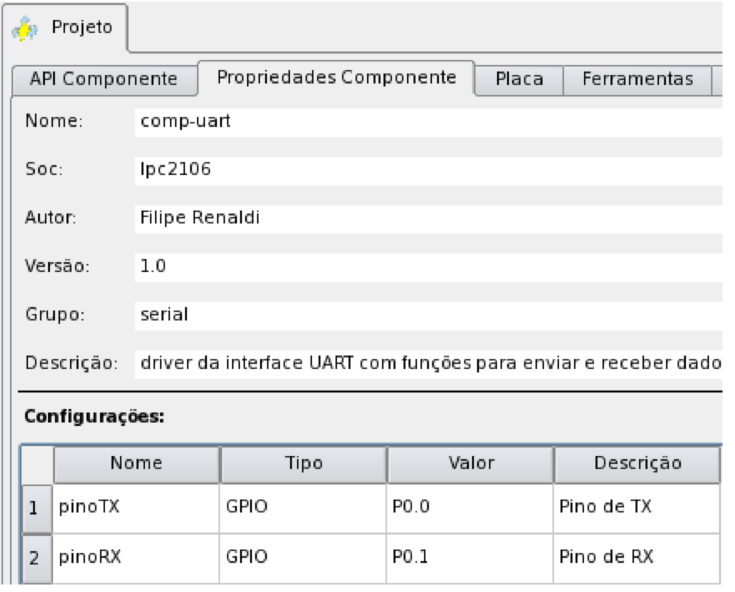


Figura 3.29 – Configuração do componente comp-uart

Dando seqüência às configurações do projeto, o usuário deve configurar na aba "Placa" o valor do cristal disponível no hardware do sistema embarcado e o *clock* desejado. Ainda nesta aba, o usuário informa os tipos e quantidades de memória disponível no sistema embarcado. Esta tela pode ser vista na fig. 3.30.

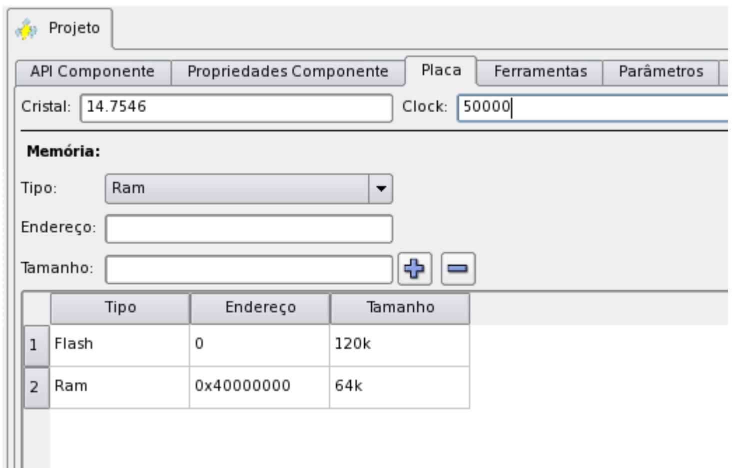


Figura 3.30 – Configuração da placa do sistema embarcado

O usuário pode ainda configurar as ferramentas de desenvolvimento na aba "Ferramentas" e os parâmetros na aba "Parâmetros".

Na aba "Arquivos de programas" o usuário cria os arquivos para desenvolver o aplicativo. Sempre que um projeto é criado, por padrão ele vem com um arquivo chamado *program.c* que contém a implementação da função *vProgramSetupTasks()* onde o usuário irá colocar a criação de suas tarefas (processos). Na fig. 3.31 pode ser vista a tela de criação de arquivos de programas. O usuário pode criar tantos arquivos quanto forem necessários sendo eles de cabeçalho ".h" ou de fonte ".c". Para editar os arquivos, basta selecionar com um duplo clique que ele será aberto em uma aba de edição.

Para desenvolver o aplicativo, o usuário poderá a qualquer momento selecionar a opção do menu "Construir / Gerar Documentação" para que tenha disponível a documentação das funções dos componentes do projeto. Para visualizar a documentação ele deve selecionar a opção do menu "Ajuda / Documentação do GenosOS" ou ainda, "Ajuda / Manual do SoC". A documentação do GenosOS será aberta em um navegador web e o manual do SoC, em um leitor de arquivos PDF.

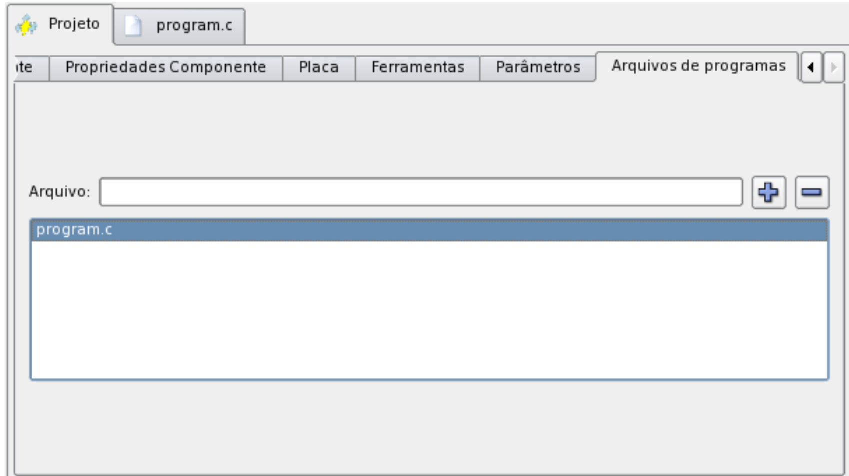


Figura 3.31 – Gerencimento dos arquivos de programas

Para este projeto de exemplo, foram criados 2 arquivos de programas. São eles prog.h e prog.c. O quadro 3.14 apresenta o arquivo prog.c que contém a tarefa do aplicativo, sendo que prog.h há apenas o protótipo da função.

O objetivo é criar um simples aplicativo que utiliza o componente "mensageiro" para mandar um mensagem ("Minha mensagem") que é enviada a cada 1 segundo pela porta serial do sistema embarcado. Foi ainda editado o arquivo program.c para inserir a chamada de início da aplicação que pode ser visto no quadro 3.15.

Finalizado o aplicativo, basta apenas compilar todo o sistema (GenosOS e aplicativo). Para isso, o usuário seleciona a opção no menu "Construir / Construir Tudo". O sistema efetua as seguintes ações:

a) gerar documentação do GenosOS;

b) gerar arquivos provenientes da normalização e configuração;

c) gerar arquivo de script de ligação (script.ld);

d) gerar arquivo de compilação (Makefile);

e) executar o comando make para compilar.

Todo o processo de compilação é exibido na janela anexa "Mensagens" e o resultado são 2 tipos de arquivos binários: genosOS.hex (formato utilizado pela maioria dos gravadores) e genosOS.elf.

Projeto | prog.c | prog.h | program.c

Arquivo: /home/filipe/FURB/TCC-II/genos_recursos/projetos/prj-olimex-1/program/prog.c

```
1 #include <stdlib.h>
2 #include <string.h>
3 #include <program.h>
4 #include "FreeRTOS.h"
5 #include "task.h"
6 #include "components.h"
7
8 #include "prog.h"
9
10 portTASK_FUNCTION( app, pvParameters )
11 {
12     portTickType xMsgRate, xLastMsgTime;
13     char msg[]="Minha Mensagem";
14
15     vSerialInit();
16
17     xMsgRate = 1000;
18     xMsgRate /= portTICK_RATE_MS;
19
20     xLastMsgTime = xTaskGetTickCount();
21
22     for(;;)
23     {
24         vTaskDelayUntil( &xLastMsgTime, xMsgRate );
25         enviarMsg( msg );
26     }
27 }
```

Quadro 3.14 – Arquivo do programa - prog.c

Projeto | prog.c | prog.h | program.c

Arquivo: /home/filipe/FURB/TCC-II/genos_recursos/projetos/prj-olimex-1/program/program.c

```
1 #include <stdlib.h>
2 #include <program.h>
3 #include "GenosOS.h"
4 #include "task.h"
5
6 #include <components.h>
7
8
9 void vProgramSetupTasks( void )
10 {
11     /* Criar as tarefas aqui */
12     xTaskCreate( app,(const char * const)"App", tStack, NULL, 2, ( xTaskHandle * ) NULL );
13 }
```

Quadro 3.15 – Arquivo do programa - program.c

O usuário poderá em seguida gravar o GenosOS no sistema embarcado utilizando a opção no menu "Construir / Programar" que irá utilizar o gravador configurado em "Ferramentas". Na fig. 3.32 pode ser visto o Genos gravando o dispositivo utilizando a porta serial do computador ligada no sistema embarcado.

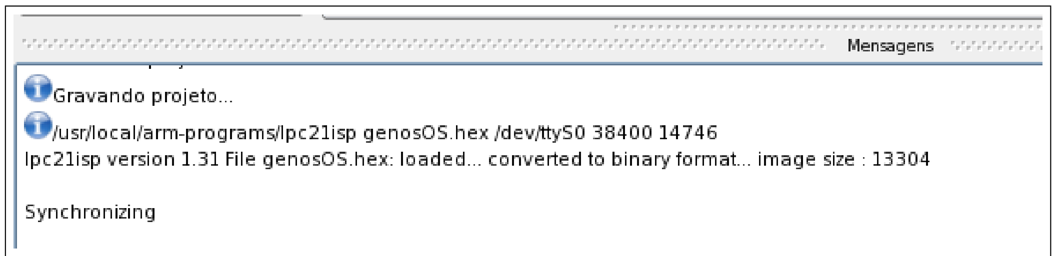


Figura 3.32 – Gravação do sistema embarcado

Ao reiniciar o sistema embarcado, podem ser visualizadas as mensagens lendo-se do dispositivo */dev/ttyS0* no Linux diretamente com o comando *cat* (fig. 3.33).

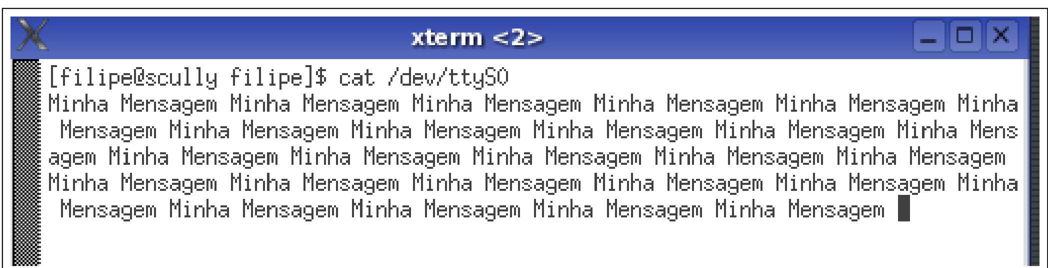


Figura 3.33 – Mensagens sendo enviada pelo sistema embarcado

Opcionalmente o usuário poderá depurar o GenosOS utilizando a opção do menu "Construir / Depurar" que irá utilizar o depurador configurado em "Ferramentas".

3.5 RESULTADOS E DISCUSSÃO

Os exemplos apresentados nas seções anteriores, embora simples, serviram para apresentar as principais características do protótipo evidenciando a abrangência das possibilidades que a ferramenta pode prover. 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.

Em comparação aos sistemas com geração baseado no aplicativo como o OaSis (SINGH et al., 2004) o GenosOS provê uma maior flexibilidade quanto a escolha dos componentes mais apropriados pois no GenosOS o usuário não apenas define se um determinado serviço faz parte do núcleo como também define qual a variante mais adequada ao aplicativo. Em contra partida os trabalhos citados buscam uma maior autonomia no desenvolvimento. Outro fator positivo do Genos é a possibilidade do aplicativo ser contruído sob componentes, e não limitando-se apenas ao núcleo ou gerenciamento dos periféricos.

Quanto ao próprio FreeRTOS, o GenosOS ficou mais organizado, facilitando a manutenção do código além da maior facilidade na adição de características específicas do processador Arm7.

O resultado do GenosOS foi um sistema operacional simples e ao mesmo tempo completo e flexível podendo suportar vários modelos de SoC. O exemplo da seção "Utilização do sistema" gerou um arquivo binário com o tamanho de aproximadamente apenas 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.

4 CONCLUSÃO

Os objetivos foram alcançados e a ferramenta mostrou-se eficiente em suas 3 partes: Genos, GenosOS e componentes.

A utilização baseada em componentes permitiu tornar o sistema operacional uma base simples ao mesmo tempo em que o usuário pode ter uma infinidade de opções na forma de componentes. Assim, a utilização dos recursos do sistema embarcado limitam-se ao necessário o que, para um projeto, significa custos minimizados. A reutilização de código é outro ponto positivo deste tipo de abordagem, sabe-se que para uma dada unidade de software, estão relacionadas ainda documentação e testes além do desenvolvimento de código, o que torna a reutilização uma forma de redução de custos e tempo de desenvolvimento.

Alguns resultados foram mais satisfatórios do que o previsto, como a técnica de normalização que não constituiu um dos objetivos iniciais deste trabalho. A necessidade da técnica de normalização foi identificada durante o desenvolvimento do protótipo visando a possibilidade de interação entre diferentes componentes e criando a possibilidade de adquirir-se componentes de "terceiros". Um cenário possível seria para fabricantes de periféricos. Por exemplo, memórias seriais utilizando I^2C, disponibilizarem os componentes de software de seus produtos. Assim, mesmo que um desenvolvedor utilizasse esse componente do fabricante, ele não estaria limitado à construir outros componentes compatíveis, bastaria normalizar as interfaces para qualquer outro componente de barramento I^2C.

O empacotamento do componente em forma de código torna possível criar variações de um determinado componente rapidamente. Para uma empresa de circuitos integrados disponibilizar um componente de forma a alavancar seus produtos é aceitável, mas talvez para uma empresa onde os produtos são os componentes, pode vir a ser uma limitação caso a mesma queira proteger seus fontes. O Genos não trabalha com componentes em

formato binário, apenas com os arquivos fontes.

A utilização do FreeRTOS como base para a construção do GenosOS mostrou-se bastante adequada.

Uma limitação quanto ao GenosOS é que suas partes internas (escalonador, gerenciador de memória, etc) não estão em forma de componentes embora a arquitetura do GenosOS possibilite a adição dessa característica sem muitas alterações.

O Genos incorpora os elementos básicos de um ambiente de desenvolvimento provendo facilidades na construção de software para sistemas embarcados. Na medida do possível buscou-se prover uma edição o mais visual possível. Essa característica é visível na área de componentes do projeto, onde o usuário visualiza todos os componentes e com poucas ações (cliques) é possível ter todas as informações do componente, inclusive o código fonte aberto diretamente em um editor integrado.

4.1 TRABALHOS FUTUROS

Este trabalho possui diversas possibilidades de expansão, uma vez que abordou o desenvolvimento de não apenas um núcleo mas toda uma infra-estrutura de desenvolvimento de sistemas embarcados. Os seguintes trabalhos são sugeridos:

a) transformar o GenosOS em um núcleo multiplataforma provendo suporte para outros tipos de processadores;

b) prover suporte a componentes em formato binário. Quanto ao atual modelo que utiliza código fonte, poderia ser ainda utilizada uma técnica de transformação do componente em uma biblioteca estática para que apenas as funções utilizadas sejam ligadas ao código executável;

c) dar suporte às métricas de performance dos componentes e núcleo criando sistema operacional determinístico para atender sistemas de tempo real forte (*hard real-time*);

d) quebrar a estrutura do GenosOS de forma a ser possível desenvolver componentes como escalonadores e gerenciadores de memória;

e) criar um módulo de geração do GenosOS baseado na aplicação (SINGH et al.,

2004);

f) prover funcionalidades ao ambiente de desenvolvimento.

REFERÊNCIAS BIBLIOGRÁFICAS

ARM. Arm7dtm-s technical reference. [s.l], 2001. Disponível em: <http://www.arm-.com/pdfs/DDI0234A_7TDMIS_R4.pdf>. Acesso em: 27 abr. 2006.

ARM. Arm corporate backgrounder. [s.l], 2005. Disponível em: <http://www.arm.com-/miscPDFs/3822.pdf>. Acesso em: 27 abr. 2006.

BARRY, Richard. Freertos - a free rtos for small embedded real time systems. [s.l], 2006. Disponível em: <http://www.freertos.org/>. Acesso em: 8 abr. 2006.

DIONE, D. Jeff; DURRANT, Michael. uclinux – embedded linux microcontroller project – home page. [s.l], 2005. Disponível em: <http://www.uclinux.org/>. Acesso em: 26 abr. 2006.

GERVINI, Alexandre I. et al. Avaliação de desempenho, área e potência de mecanismos de em sistemas embarcados. In: XXIII CONGRESSO DA SOCIEDADE BRASILERIA DE COMPUTAÇÂO, 22., 2003, Florianópolis. **Anais...** Florianópolis: SBC, 2003. p. 50–73.

GIMENES, Itana Maria de Souza; HUZITA, Elisa Hatsue Moriya. **Desenvolvimento baseado em componentes**: conceitos e técnicas. Rio de Janeiro: Ciência Moderna, 2005.

HEESCH, Dimitri Van. Doxygen. [s.l], 2006. Disponível em: <http://www.stack.nl-/˜dimitri/doxygen/>. Acesso em: 03 maio 2006.

LI, Qing; YAO, Caroline. **Real-time concepts for embedded systems**. San Francisco: CMP Books, 2003.

OLIMEX. Development and prototype boards and tools for pic avr and msp430. Plovdiv, 2006. Disponível em: <http://www.olimex.com/dev/lpc-p1.html>. Acesso em: 28 abr. 2006.

PHILIPS. Lpc2104/2105/2106 user manual: single-chip 32-bit microcontrollers; 128 kb isp/iap flash with 64 kb/32 kb/16 kb ram. [s.l], 2003. Disponível em: <http://www-.semiconductors.philips.com/acrobat/usermanuals/UM_LPC2106_2105_2104_1.pdf>. Acesso em: 26 ago. 2005.

SANTO, Brian. Embedded battle royle. **IEEE Spectrum**, IEEE, v. 38, n. 12, p. 36–41, 2001.

SINGH, Gurashish et al. Oasis: an application specific operating system for an embedded environment. In: 17TH INTERNATIONAL CONFERENCE ON VLSI DESIGN, 17., 2004, [s.l]. **Proceedings...** [S.l.]: IEEE, 2004. p. 776–779.

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. 2. ed. São Paulo: Prentice Hall, 2003.

TROLLTECH. Homepage - trolltech. Oslo, 2006. Disponível em: <http://www.trolltech-.com/>. Acesso em: 03 maio 2006.

VAHID, Frank; GIVARGIS, Tony D. **Embedded system design**: a unified hardware/software introduction. Danvers: Wiley, 2001.

VXWORKS. Wind river company. Alameda, 2005. Disponível em: <http://www-.windriver.com/>. Acesso em: 10 nov. 2005.

WIKIPÉDIA. **Embedded system - wikipedia, the free encyclopedia**. [s.l], 2006a. Disponível em: <http://en.wikipedia.org/wiki/Embedded system>. Acesso em: 26 abr. 2006.

WIKIPÉDIA. **ARM architecture - wikipedia, the free encyclopedia**. [s.l], 2006b. Disponível em: <http://en.wikipedia.org/wiki/ARM_architecture>. Acesso em: 27 abr. 2006.