Lista Teórica de Sistemas Embarcados 1

Aluna: Heloisa Marimoto

Matricula: 11821ETE009

1. Explique brevemente o que é compilação cruzada (*cross-compiling*) e para que ela serve.

A compilação cruzada refere-se ao procedimento no desenvolvimento de software em que o código é compilado em um tipo de sistema operacional ou arquitetura de hardware (o sistema hospedeiro) para ser operado em um sistema ou arquitetura distintos (o sistema alvo). Essencialmente, é o ato de utilizar um computador para gerar um aplicativo que será executado em outro computador com uma configuração arquitetônica diferente daquela em que foi originalmente compilado.

Este método é particularmente valioso e essencial no desenvolvimento de sistemas embarcados, onde os dispositivos frequentemente funcionam em arquiteturas de hardware especializadas que não são as mesmas do computador usado para o desenvolvimento e a compilação do código. Por exemplo, a criação de software para um microcontrolador ARM geralmente exige a compilação do código em um PC que opera sob uma arquitetura x86 ou x86_64.

O processo de compilação cruzada habilita os desenvolvedores a aproveitar as capacidades e ferramentas de um sistema hospedeiro mais robusto (por exemplo, um PC) para desenvolver software destinado a sistemas alvo, os quais podem possuir capacidades reduzidas ou ser incapazes de realizar a compilação de software independentemente. Isso torna mais ágil e eficiente o desenvolvimento, teste e depuração de aplicativos embarcados, facilitando a transição do software para o dispositivo final.

2. O que é um código de inicialização ou *startup* e qual sua finalidade?

O código de inicialização, ou startup code, representa um segmento crucial de código que é executado por microcontroladores ou sistemas embarcados durante a fase de inicialização ou após um reset. Este código tem a função de configurar o ambiente de execução necessário para que o software principal funcione de maneira adequada e eficaz. Executado antes de qualquer outro software de aplicação, ele realiza várias tarefas essenciais para a operação adequada do sistema, que incluem:

Configuração do Hardware: Realiza a configuração dos registros de hardware, inicializa os clocks do sistema, ajusta dispositivos periféricos e deixa o sistema pronto para rodar o

software. Isso engloba tarefas como a configuração de memória, ativação dos sistemas de interrupção, entre outras ações específicas ao hardware utilizado.

Estabelecimento da Stack: Define a área de stack (pilha) do programa, que é vital para o suporte a chamadas de funções, gestão de interrupções e execução de tarefas, funcionando como um local de armazenamento para variáveis locais, endereços de retorno e o gerenciamento de interrupções e threads.

Preparação de Variáveis Estáticas e da Seção de Dados: Transfere valores iniciais pré-determinados para variáveis estáticas e zera variáveis não inicializadas (seção bss), assegurando que o programa inicie com valores corretos nas variáveis e evitando comportamentos inesperados por conta de dados residuais na memória.

Execução da Função Principal: Com as configurações iniciais completas, o código de inicialização invoca a função principal do software (tipicamente main() em linguagens como C ou C++), momento no qual o software embarcado começa a operar normalmente.

Dessa forma, o propósito do código de inicialização é preparar adequadamente o sistema embarcado para a execução do software de aplicação, assegurando que todos os pré-requisitos, tanto de hardware quanto de software, estejam devidamente estabelecidos antes do programa principal começar a rodar.

3. Sobre o utilitário make e o arquivo Makefile responda:

(a) Explique com suas palavras o que é e para que serve o Makefile.

Um Makefile serve como uma ferramenta fundamental para o utilitário make, facilitando a automatização do processo de compilação e montagem de programas. Dentro deste arquivo, encontram-se diversas instruções detalhando o processo para transformar os arquivos de código fonte em executáveis e outros tipos de artefatos. O Makefile estrutura-se em torno de "targets" (metas), enumerando as dependências necessárias para cada um desses targets, além de especificar as instruções (comandos) para construir cada target baseando-se em suas dependências. Esta configuração viabiliza a execução automática e sistemática de tarefas comuns no desenvolvimento de software, como a compilação, e linkagem de bibliotecas e a remoção de arquivos objeto, promovendo uma gestão mais eficaz e uniforme do processo de construção do software.

(b) Descreva brevemente o processo realizado pelo utilitário make para compilar um programa.

O comando make opera lendo o Makefile no diretório atual e segue os passos abaixo para compilar um programa:

Selecionar o Target Principal: Inicialmente, make foca no primeiro target mencionado no Makefile, exceto se um target distinto for especificado na linha de comando.

Analisar Dependências: Para cada target mencionado, make examina as dependências indicadas para determinar se necessitam de atualização. Uma dependência pode ser tanto um arquivo necessário para a compilação quanto outro target.

Executar Atualizações: Caso uma dependência tenha sido alterada desde a última compilação do target (ou caso o target não exista previamente), make ativa as regras (ou comandos) descritas no Makefile para renovar o target.

Processo Iterativo: Este procedimento é aplicado iterativamente para as dependências de cada target, mantendo a ordem correta para assegurar a atualização de todas as dependências antes de prosseguir para a atualização do próprio target.

(c) Qual é a sintaxe utilizada para criar um novo target?

A sintaxe básica para definir um target no Makefile é:

```makefile
target: dependencies
    commands
```

- `target`: Nome do target a ser construído.

- `dependencies`: Lista de arquivos ou outros targets dos quais o target depende.

- `commands`: Sequência de comandos a serem executados para construir o target. Estes comandos devem ser precedidos por um caractere de tabulação.

(d) Como são definidas as dependências de um target, para que elas são utilizadas?

As dependências de um alvo (target) no Makefile são estabelecidas logo após o nome do alvo, sendo separadas entre si por espaços na linha subsequente. Essas dependências delineiam os arquivos ou alvos que devem ser previamente existentes ou atualizados para que o alvo em questão possa ser compilado. A função das dependências é dupla: elas ajudam a definir a sequência de execução dos comandos e contribuem para a eficiência do processo de compilação ao prevenir a reconstrução desnecessária de partes do programa que permaneceram inalteradas.

(e) O que são as regras do Makefile, qual a diferença entre regras implícitas e explícitas?

Dentro de um Makefile, as regras são responsáveis por orientar a construção de um alvo (target) a partir de suas dependências. Elas se classificam em dois grupos principais:

Regras Explícitas: Correspondem às instruções definidas de forma direta no Makefile para construir um alvo específico, com os passos de construção claramente especificados.

Regras Implícitas: Representam um conjunto de padrões preestabelecidos que o comando `make` é capaz de aplicar automaticamente a vários alvos, eliminando a necessidade de detalhar os comandos para cada um deles de forma individual. O `make` vem com uma série de regras implícitas pré-configuradas, mas também permite a personalização e criação de novas regras implícitas pelos usuários. Essas regras são especialmente valiosas para minimizar repetições e facilitar a organização do Makefile, permitindo a construção de múltiplos alvos por meio de comandos genéricos que levam em conta seus nomes e dependências.

4. Sobre a arquitetura ARM Cortex-M responda:

(a) Explique o conjunto de instruções *Thumb* e suas principais vantagens na arquitetura ARM. Como o conjunto de instruções *Thumb* opera em conjunto com o conjunto de instruções ARM?

A funcionalidade Thumb nos processadores ARM introduz a capacidade de executar instruções de 16 bits, oferecendo um contraponto às tradicionais instruções ARM de 32 bits. Essa característica traz várias vantagens significativas:

Redução no Consumo de Memória: Ao serem mais compactas, as instruções Thumb diminuem o tamanho geral do código, beneficiando dispositivos com memória limitada.

Aumento de Desempenho: A compactação do código permite uma utilização mais eficiente do cache, potencializando o desempenho do programa.

Flexibilidade e Compatibilidade: Os processadores que adotam o conjunto de instruções Thumb são capazes de executar tanto instruções de 32 bits (ARM) quanto de 16 bits (Thumb), facilitando a transição entre diferentes abordagens de codificação e aumentando a versatilidade no desenvolvimento de software.

Esses processadores permitem a operação em um modo híbrido, alterando dinamicamente entre instruções ARM e Thumb conforme necessário. Esta capacidade é controlada pelo bit T no Registro de Status do Programa Atual (CPSR), possibilitando uma integração fluida e eficiente dos dois tipos de instruções.

(b) Explique as diferenças entre as arquiteturas *ARM Load/Store* e *Register/Register*.

As arquiteturas de processadores, ARM Load/Store e Register/Memory, diferem fundamentalmente na forma como interagem com a memória para realizar operações com dados. A arquitetura ARM Load/Store é caracterizada pelo seu método restrito de executar

operações aritméticas e lógicas exclusivamente com dados armazenados em registradores. Isso significa que os dados devem ser carregados da memória para um registrador antes que possam ser manipulados, e os resultados das operações devem ser armazenados de volta na memória. Esta abordagem minimiza os acessos à memória, o que pode melhorar o desempenho e a eficiência energética, especialmente em dispositivos móveis e sistemas embarcados, mas coloca mais demanda sobre o compilador para gerenciar eficientemente os registradores disponíveis.

Por outro lado, as arquiteturas Register/Memory permitem que as operações sejam realizadas tanto em dados localizados diretamente na memória quanto em registradores, oferecendo uma maior flexibilidade. Isso pode facilitar a geração de código pelo compilador, pois elimina a necessidade de carregar e armazenar dados explicitamente para operações simples. No entanto, essa conveniência pode aumentar o número de acessos à memória, potencialmente resultando em um desempenho mais lento se o sistema tiver um acesso à memória relativamente lento em comparação com a velocidade do processador.

Em essência, a diferença chave entre essas arquiteturas reside na sua abordagem de manipulação de dados: a ARM Load/Store foca na eficiência através da limitação das operações a dados em registradores, exigindo movimentos explícitos de dados entre a memória e os registradores, enquanto a Register/Memory oferece uma abordagem mais flexível e direta ao permitir operações em dados tanto na memória quanto em registradores, às custas de possíveis acessos à memória mais frequentes.

(c) Os processadores ARM Cortex-M oferecem diversos recursos que podem ser explorados por sistemas baseados em RTOS (*Real Time Operating Systems*). Por exemplo, a separação da execução do código em níveis de acesso e diferentes modos de operação. Explique detalhadamente como funciona os níveis de acesso de execução de código e os modos de operação nos processadores ARM Cortex-M.

Os processadores ARM Cortex-M adotam uma estrutura de privilégios simplificada que suporta operações em dois distintos níveis de acesso:

Modo Privilegiado: Concede permissão irrestrita sobre o hardware, habilitando a configuração de elementos de segurança e gestão do sistema.
Modo Não Privilegiado: Limita o acesso a certas funcionalidades do sistema, promovendo uma maior segurança e isolamento entre os diferentes segmentos do software.

Esta organização de níveis de acesso é fundamental para o desenvolvimento eficaz de sistemas operacionais que distinguem claramente entre o código do kernel, que opera em modo privilegiado, e as aplicações do usuário, que funcionam em modo não privilegiado. Tal diferenciação é crucial para a construção de ambientes computacionais seguros.

(d) Explique como os processadores ARM tratam as exceções e as interrupções. Quais são os diferentes tipos de exceção e como elas são priorizadas? Descreva a estratégia de group priority e sub-priority presente nesse processo.

Nos processadores ARM, existe uma distinção clara entre exceções (que abrangem erros e eventos do sistema) e interrupções (que são solicitações de atenção originadas de fontes externas ou internas). As exceções são organizadas e tratadas com base em sua relevância e imediatismo. Especificamente, a linha ARM Cortex-M implementa um sistema de prioridades que suporta tanto prioridades de grupo quanto sub prioridades, oferecendo um controle detalhado sobre como as exceções são gerenciadas e resolvidas em ordem de importância.

(e) Qual a diferença entre os registradores CPSR (*Current Program Status Register*) e SPSR (*Saved Program Status Register*)?

CPSR (Registro de Status do Programa Atual): Esse registro mantém as flags de status e detalhes de controle referentes ao programa que está sendo executado no momento, abrangendo também o bit que indica o modo Thumb.
SPSR (Registro de Status do Programa Salvo): Este registro é utilizado para preservar o estado do CPSR antes da ocorrência de uma exceção ou interrupção, viabilizando a recuperação deste estado anterior quando o controle é retomado após a exceção.

(f) Qual a finalidade do LR (*Link Register*)?

O Registro de Ligação (LR, do inglês Link Register) serve para guardar o endereço de retorno durante a execução de uma chamada de função, assegurando que, após a finalização da função, o fluxo de execução possa voltar ao ponto apropriado no código.

(g) Qual o propósito do Program Status Register (PSR) nos processadores ARM?

No contexto dos processadores ARM, o PSR (Program Status Register) integra informações sobre o status do programa e dados de controle, abrangendo tanto as flags de condição quanto o bit que indica o modo Thumb. Essa combinação contribui para uma gestão eficiente dos estados do programa e um controle aprimorado sobre o fluxo de execução.

(h) O que é a tabela de vetores de interrupção?

A tabela de vetores de interrupção constitui uma estrutura de dados crucial que associa cada exceção ou interrupção ao seu manipulador específico (uma função designada para responder ao evento), possibilitando uma reação ágil a diferentes ocorrências.

(i) Qual a finalidade do NVIC (Nested Vectored Interrupt Controller) nos microcontroladores ARM e como ele pode ser utilizado em aplicações de tempo real?

O Controlador de Interrupção Vetorizada de Núcleo (NVIC, na sigla em inglês) desempenha um papel fundamental nos microcontroladores ARM, gerenciando interrupções de maneira eficaz através da priorização, aninhamento e tratamento adequado desses eventos. Esse componente é indispensável em aplicações de tempo real, nas quais é crucial uma resposta imediata e organizada a eventos externos.

(j) Em modo de execução normal, o Cortex-M pode fazer uma chamada de função usando a instrução BL, que muda o PC para o endereço de destino e salva o ponto de execução atual no registador LR. Ao final da função, é possível recuperar esse contexto usando uma instrução BX LR, por exemplo, que atualiza o PC para o ponto anterior. No entanto, quando acontece uma interrupção, o LR é preenchido com um valor completamente diferente, chamado de EXC_RETURN. Explique o funcionamento desse mecanismo e especifique como o Cortex-M consegue fazer o retorno da interrupção.

No evento de uma interrupção, o conteúdo do Registro de Ligação (LR) é atualizado para um valor distinto, conhecido como EXC_RETURN. Esse valor especial carrega detalhes cruciais sobre o procedimento de retorno da interrupção, especificando, por exemplo, se a execução deve prosseguir no modo e na pilha de privilégios ou não privilégios. Essa funcionalidade assegura que o processador possa restabelecer adequadamente o estado anterior e retomar a execução de forma correta após a interrupção ser atendida.

(k) Qual a diferença no salvamento de contexto, durante a chegada de uma interrupção, entre os processadores Cortex-M3 e Cortex M4F (com ponto flutuante)? Descreva em termos de tempo e também de uso da pilha. Explique também o que é *lazy stack* e como ele é configurado.

Os processadores Cortex-M4F, equipados com suporte a cálculos de ponto flutuante, apresentam uma diferença significativa em relação aos Cortex-M3 no que tange à preservação do contexto durante uma interrupção. No caso dos Cortex-M4F, o salvamento do contexto de ponto flutuante é realizado de maneira "lazy" ou preguiçosa. Isso significa que o estado de ponto flutuante é armazenado apenas se e quando uma instrução de ponto flutuante é executada durante a interrupção. Tal abordagem visa otimizar tanto o uso da pilha quanto o tempo de processamento necessário para as operações de entrada e saída das interrupções.