Universidade Federal de Pernambuco
Graduação em Engenharia da Computação

Centro de Informática
2009.1

KEDBG-JTAG: UM DEPURADOR ARMJTAG PARA O ERESI REVERSE ENGINEERING FRAMEWORK

Trabalho de Graduação

Autor: Jesus Sanchez-Palencia Fernandez Filho (jspff@cin.ufpe.br)
Orientador: Sérgio Cavalcante (svc@cin.ufpe.br)

Recife, 26 de Fevereiro de 2009

Universidade Federal de Pernambuco
Graduação em Engenharia da Computação

Centro de Informática
2009.1

KEDBG-JTAG: UM DEPURADOR ARMJTAG PARA O ERESI REVERSE ENGINEERING FRAMEWORK

Trabalho de Graduação

Trabalho de Graduação apresentado para o Centro de Informática da Universidade Federal de Pernambuco por Jesus Sanchez-Palencia, orientado pelo Prof. PhD. Sérgio Cavalcante, como um requerimento para se obter a graduação em Engenharia da Computação.

Orientador: Sérgio Cavalcante (svc@cin.ufpe.br)

Recife, PE
2009

FOLHA DE APROVAÇÃO

JESUS SANCHEZ-PALENCIA FERNANDEZ FILHO

KEDBG-JTAG: UM DEPURADOR ARMJTAG PARA O ERESI REVERSE ENGINEERING FRAMEWORK

Trabalho aprovado em 30 de Junho de 2009.

Banca Avaliadora

_____

Prof. Sérgio Cavalcante (svc@cin.ufpe.br), PhD – UFPE

(Orientador)

_____

Prof. Abel Guilhermino, PhD – UFPE

(Avaliador)

Para todos aqueles que me **apoiaram** e **acompanharam** nesses últimos cinco anos, em qualquer aspecto. Meu mais **sincero obrigado** a minha mãe, meu pai e meus irmãos, namorada, meus professores, colegas de classe, amigos e companheiros de trabalho. Foram os anos mais prazerosos e difíceis da minha vida. Faria, porém, **tudo de novo**.

"I have to Change to stay the Same."

Willem De Kooning

AGRADECIMENTOS

Seria injusto não começar essa seção agradecendo a minha família. A meu pai, Jesus Sanchez-Palencia Fernandez, por ser um dos homens mais íntegros e honestos que já conheci, além de um dos engenheiros mais fantásticos. A minha mãe, Iraci Freitas Sanchez-Palencia, por ser a mulher mais forte e sagaz de que tenho conhecimento. Aos meus irmãos, que mereceriam todos várias linhas de elogios mas que vou resumir chamandos a ambos de "seus diiiis!!!". Posso garantir que vocês em suas qualidades de pai, mãe, irmão e irmã fizeram seus papéis da melhor forma possível e me ensinaram que uma família é a coisa mais importante que podemos ter, principalmente quando ela é unida como nós sempre fomos. Também a Lorena, que não poderia ficar de fora. *La familia*...

Seria injusto também não citar todos aqueles que nos últimos 23 anos contribuíram de qualquer maneira para a formação do meu intelecto, para as minhas escolhas. Peço desculpas de coração se esqueci de mencionar algum de vocês nas próximas linhas. Os nomes que seguem serão apresentados na ordem que minhas lembranças permitiram, e **não** em ordem de importância.

Agradeço ao professor Sérgio Cavalcante, não só por ter sido orientador deste trabalho, mas também por ter sido quase que um conselheiro durante todos os 5 anos do curso de engenharia. Foram 3 disciplinas e um Trabalho de Graduação, além conversas e discussões em salas de aula no CIn ou em bares do Recife.

Agradeço a Julio Cesar Fort, Gustavo Pimentel e Gustavo Monteiro por terem me acolhido no *underground*. Foram inúmeras horas de discussão, viagens, congressos e projetinhos, que com certeza ajudaram muito no desenvolvimento da minha forma de pensar e analisar problemas. Além, é claro, de terem se demonstrado grandes amigos, mesmo com toda a minha "frieza".

Agradeço a todo o *ERESI Crew*, pela força e apoio no desenvolvimento deste trabalho. Em especial preciso agradecer a Thiago Figueredo por seu esforço no porte da *libasm* para a arquitetura *ARM* e a Eric Bisolfalti por seu incrível trabalho de desenvolvimento da *libgdbwrap* e do *KEDBG*. Preciso destacar meus agradecimentos a Julio Auto a Julien Vanegue, tanto pela dedicação no projeto *ERESI*, quanto pela ajuda imensurável no desenvolvimento deste projeto. Além disso, preciso agradecer-lhes pelas horas que gastamos como amigos e por vocês terem me mostrado o que significa ser um "acadêmico". Foi um prazer conhecer pessoalmente dois *hackers* tão talentosos e dedicados, sempre comprometidos em se obter o melhor resultado. Sempre.

Agradeço a Farid, Diego, Diogo e Jubby e nina (!), Lucas, Matheus, Pontes, Daniel, Dirceu, Felipe "o Ogro", pelas infinitas horas juntos. Vocês todos formam o grupo de amigos mais heterogêneo que já conheci. Cada um com sua mania/qualidade/chatice/loucura/jeito/qualidade foi capaz de contribuir de uma forma singular para a formação da minha personalidade. Tem sido ótimos anos...

Agradeço também ao Instituto Nokia de Tecnologia, *INdT*, pelo simples fato de ser o melhor lugar para se trabalhar, pesquisar e aprender do país. Em especial, gostaria de agradecer a Anselmo, Artur, Caio, Fleury, Etrunko e Chenca, por terem me ensinado **tanto** desde que tive a oportunidade de começar a trabalhar com eles. Vocês tem me ensinado a pensar em soluções de uma forma que eu nunca havia imaginado antes, além das técnicas, *hacks* e "magias negra" que

sempre vou absorvendo no dia-a-dia. Meu conceito de impossível ganhou uma nova definição após esses 15 meses de trabalho. Também não poderia deixar de agradecer a Ademar, Adriano, Sheldon, Danilo, Kenneth e Leo, pela época de aprendizagem no projeto em que trabalhamos (e nos divertimos!) juntos. Gostaria de agradecer também a todos os designers do *INdT* pelas tantas outras coisas que foram capazes de me ensinar, além de todas as tendências que puderam me mostrar.

Por último, mas não menos importante, agradeço a todos os meus amigos (ou colegas!) da turma 2004.2 . Thiago, Bruno, Kakimoto, Ademir, Guedes, Chico, David, Felipe "moxinho", Felipe "Gravatá", Rebeka, Rilter, Digão, que são os que correram para Ciência da Computação mas que ainda merecem meu respeito :P!!! A Rafael, André "age", Marcelo, Guilherme, Bruno, Hudson, Renata, Pyetro, por todas as horas de estudo para as provas de Engenharia e de conversas genéricas. E foram muitas horas...

A **todos** os integrantes do PET Computação com que tive contato nesses 3 anos e meio de participação no grupo e ao professor tutor Fernando Fonseca. Vocês me deixam orgulhosos de ter participado de um programa tão completo e importante no contexto da educação nacional.

E também, é claro, a Ana Cecília, ou aninha como ela prefere. Simplesmente por ser a namorada, amiga e companheira linda e carinhosa que você sempre é. Você e seu jeito único, as constantes conversas sobre a Vida, o Universo e Tudo Mais... e que ainda ensina a programar nas horas vagas! (ha ha!).

RESUMO

Há mais de duas décadas que sistemas computacionais estão presentes na vida das pessoas, auxiliando nas mais diversas tarefas do dia-a-dia. Quando nos referimos a esses sistemas não estamos apenas falando dos computadores pessoais, mas também dos sistemas embarcados – ou embutidos.

São encontradas boas ferramentas de programação de sistemas embarcados, além de ferramentas para testes e depuração. A maioria dessas ferramentas fazem uso do padrão *IEEE 1149.1*, ou padrão *JTAG*. Essa tecnologia foi criada para se existisse um método universal para execução de testes e *boundary-scan* em placas de circuito impresso.

Neste trabalho será apresentada a pesquisa que envolveu o estudo do padrão *JTAG*, da arquitetura *ARM* e de técnicas de depuração e engenharia reversa, para o desenvolvimento de um depurador remoto de sistemas embarcados como parte integrante do framework de engenharia reversa *ERESI*.

SUMÁRIO

1 INTRODUÇÃO

Há mais de duas décadas que sistemas computacionais estão presentes na vida das pessoas, auxiliando nas mais diversas tarefas do dia-a-dia. Quando nos referimos a esses sistemas não estamos apenas falando dos computadores pessoais, mas também dos sistemas embarcados – ou embutidos. Estes são sistemas computacionais de propósito especial, projetados para executar um conjunto específico de funções, normalmente inseridos em sistemas maiores.

São várias as arquiteturas, memórias, barramentos, processadores e componentes disponíveis, o que torna sistemas embarcados um campo de pesquisa e desenvolvimento bem diversificado. Além disso, são muitos os fabricantes de placas e circuitos impressos, cada qual com sua própria linha de modelos de dispositivos, cabos e kits de desenvolvimento – ou *SDKs*.

Essas ferramentas são projetadas para os desenvolvedores de sistemas embarcados e procuram facilitar ao máximo o trabalho destes, oferecendo uma gama de funcionalidades que vão desde editores de código até simuladores e emuladores das arquiteturas alvo, para facilitar os testes do projeto em desenvolvimento. Existem várias maneiras para se testar um projeto desse nicho, mas enquanto em ambiente de desenvolvimento, os projetistas de sistemas embarcados – e desenvolvedores de software no geral - recorrem a depuradores. Depuração é o processo de se seguir a execução de um sistema, instrução por instrução.

Ainda que os simuladores e emuladores providos pelos já citados *SDKs* sejam bastante completos e funcionais, testar o sistema em seu dispositivo alvo real pode apontar para falhas que não tenham sido percebidas durante os testes simulados (ou emulados). Mas como executar um depurador em um sistema embarcado juntamente com o código sendo desenvolvido uma vez que estes dispositivos dispõem de recursos de memória e processamento limitados ?

Em 1990, o padrão *IEEE* 1149.1 para testes de placas e sistemas embarcados foi lançado. Este provê a definição de uma porta de acesso para execução de *boundary-scan* – um método de acesso às partes internas de dispositivos - e realização de testes em dispositivos eletrônicos, que ficou conhecido como *JTAG*. Atualmente o protocolo e a porta *JTAG* em sistemas embarcados são utilizados também como um mecanismo de transporte de dados provindos do módulo de depuração integrado ao processador, quando disponível, para o ambiente externo de depuração. Em resumo: *JTAG* se tornou uma interface para testes e depuração de sistemas embarcados, largamente utilizada e presente em incontáveis dispositivos.

Este trabalho se propõe a desenvolver um depurador para dispositivos embarcados, utilizando o padrão *JTAG*. Esta ferramenta será desenvolvida para ser executada em um computador comum, arquitetura *x86*, podendo assim ser classificada como um depurador remoto uma vez que ela não estará em execução no sistema alvo. O resultado final deste trabalho deverá ser parte integrante do framework provido pelo projeto *ERESI* [6]. O *ERESI Reverse Engineering Software Interface* – ou *ERESI* – é um projeto em desenvolvimento há mais de 6 anos, que tem como objetivo oferecer um conjunto de bibliotecas e ferramentas para obtenção do maior número de informações possíveis sobre um programa compilado.

Além de todo o código que precisa ser desenvolvido como prova de conceito, este trabalho também têm como objetivo efetuar uma pesquisa sobre depuração de sistemas embarcados via *JTAG*, analisando ferramentas existentes e levantando possíveis arquiteturas para execução de tais projetos. Ferramentas comerciais de depuração não serão avaliadas uma vez que o autor não possui acesso a elas. Sendo assim, podemos afirmar que este projeto de pesquisa tem um último objetivo: desmistificar a idéia de que *JTAG* é uma tecnologia cara para ser utilizada por desenvolvedores independentes ou em pequenos projetos comerciais.

O capítulo 2 introduz os conceitos necessários para uma melhor compreensão do trabalho, divagando sobre depuração e engenharia reversa de sistemas embarcados, além de uma pequena introdução sobre o protocolo remoto *GDB*. No capítulo 3 temos uma apresentação e avaliação de ferramentas disponíveis para *JTAG interfacing* e uma explicação mais completa sobre o framework *ERESI*. No capítulo 4 apresentamos a arquitetura escolhida e os motivos que levaram a tal escolha, além da explanação sobre o desenvolvimento completo da prova de conceito. Por último, no capítulo 5, teremos uma demonstração do uso da ferramenta, através de uma sessão real e completa de depuração de sistema embarcado.

2 BACKGROUND

Neste capítulo teremos uma introdução sobre as áreas de conhecimento que o leitor precisa conhecer antes de seguir para a proposta e o desenvolvimento desse trabalho. Em um primeiro momento, teremos uma visão geral sobre depuradores e engenharia reversa, seguindo para uma visão sobre depuração de sistemas embarcados. Aqui apresentaremos uma tecnologia chave nessa pesquisa: o padrão *JTAG*. Por último, será introduzido o protocolo remoto *GDB*.

2.1 Depuração

Depuração – ou *Debugging*, em Inglês – é o processo metódico de se seguir, instrução por instrução, a execução de um sistema. Para a realização de tal tarefa utilizamos um depurador – ou *Debugger* -, que nos auxilia durante a tentativa de encontrarmos falhas no código ou processo em análise. Vamos definir falha como uma ocorrência não esperada em um programa, podendo ela interromper a execução do processo ou não.

O fluxo comum do processo de depuração é executar o software em questão até que uma falha seja encontrada. Quando isso acontece, o depurador apontará no código o ponto onde o problema ocorreu. Depuradores simbólicos mostrarão a falha diretamente no código da linguagem de alto nível em que o software foi escrito, quando temos acesso ao código-fonte, enquanto que depuradores de baixo nível apontarão para a instrução na linguagem de máquina. Existem técnicas de depuração nas quais o código a ser examinado é executado em um simulador – os chamados *Instruction Set Simulators* (ISS) – mas que costumam ser mais lentas que executar o código diretamente no processador alvo.

Algumas funcionalidades são bastante comuns de serem encontradas em depuradores. Executar um programa passo a passo, pausar a execução para análise, acompanhar valores de variáveis e inserir *breakpoints* – pontos que indicam um momento para o depurador pausar a execução -, formam um conjunto básico de funções poderosas que normalmente são oferecidas.

Alguns depuradores oferecem opções bem mais avançadas, tais como alterar o estado, fluxo ou variáveis de um programa em tempo de execução. Nos últimos anos, os desenvolvedores do *GNU Debugger*, que será apresentado ainda nesse capítulo, focaram em uma funcionalidade que eles chamaram de Depuração Reversa - a capacidade de "andar" para trás na execução de um programa se mostrou uma necessidade já que por muitas vezes passa-se despercebido por um ponto importante durante o processo de depuração [1].

Esse trabalho tem como foco o processo de depuração de sistemas embarcados, mas veremos que todos os conceitos aqui apresentados sobre depuração de software também se aplicam.

2.1.1 Engenharia Reversa

Engenharia Reversa é a arte de se entender os princípios de um sistema ou dispositivo, estudando sua estrutura e funcionamento. Basicamente, deve-se entender a fundo as suposições e decisões feitas pelos desenvolvedores de um sistema para, em seguida, subvertê-las. É um processo de análise comportamental no qual aspectos internos do projeto têm que ser levados em consideração [26].

Esse tipo de estudo começou décadas atrás, quando os desenvolvedores tinham acesso a poucos códigos-fonte que pudessem precisar – software, *drivers*, sistemas operacionais, etc. – e, por isso, precisavam encontrar maneiras de entender aquilo que estavam utilizando. Essa necessidade podia ser apenas por curiosidade ou por um motivo mais prático, como tentar resolver problemas que os fabricantes não conseguiam, mas o alvo era o mesmo: entender a estrutura de um programa e sua lógica.

Em contrapartida, havia também as práticas de espionagem industrial ou militar que faziam uso freqüente de engenharia reversa de hardware. O princípio era o mesmo, entender por completo o funcionamento de um dispositivo sem ter acesso aos planos de projeto, mas os desafios eram ainda maiores, dada a complexidade dos artefatos em questão.

O processo de Engenharia Reversa envolve algumas ferramentas e o maior número de informações possíveis acerca da arquitetura de um sistema. Depuradores, já apresentados, são ferramentas básicas nesse processo e o próprio conceito de depuração em si pode ser considerado uma etapa básica da Engenharia Reversa. Somam-se a eles os *disassemblers*, que "desmontam" o software em linguagem *assembly*, e os descompiladores. Estes são ferramentas que tentam converter código *assembly* ou código de máquina em linguagem de alto nível para que se possa ter uma idéia de como um sistema foi escrito.

Nos últimos anos, outras ferramentas se mostraram bastante úteis no auxílio desse processo. Análise em tempo-real, geração de grafos de chamadas e de fluxo de execução, injeção de dados aleatórios, *dynamic patching*, etc., são algumas das palavras-chaves que podem interessar ao leitor.

A indústria de software e hardware proprietário tem tentado dificultar ao máximo as ações de engenheiros reversos em seus produtos, utilizando técnicas de criptografia e de ofuscação de código. Estes, porém, são tópicos que fogem do escopo desse trabalho e que, portanto, não serão abordados.

2.2 Depuração de Sistemas Embarcados

Sistemas embarcados – ou sistemas embutidos – são sistemas computacionais de propósito específico que são projetados para realizar um pequeno conjunto dedicado de funções. Em muitos casos, sistemas embarcados são utilizados como componentes de um sistema maior, ou produto, estando presentes em todos dispositivos eletrônicos que utilizamos hoje em dia. Em 2005, por exemplo, dos nove bilhões de processadores produzidos no mundo menos de 2% foram utilizados em computadores pessoais (PC), tendo os 98% restantes sido utilizados na produção de sistemas embarcados [3].

Conforme visto na introdução deste trabalho, existem boas ferramentas não-comerciais existentes para desenvolvimento de sistemas embarcados e estas são largamente utilizadas, mas existe uma certa mistificação das ferramentas de depuração desses sistemas. Depuração de sistemas embarcados é um processo que difere em alguns pontos de depuração tradicional de software. Primeiro, porque sistemas embutidos dispõem, quase sempre, de menos recursos de memória, processamento e de dispositivos de entrada/saída quando comparados a computadores *Desktop*. Essas restrições tornam inconveniente, ou até mesmo impossível, a execução de um depurador em conjunto com o software embarcado no sistema alvo. Segundo, porque em alguns casos o sistema embarcado em questão pode nem possuir um software embarcado e ainda assim os desenvolvedores podem precisar testar um *bootloader* – o software que inicializa o sistema – ou precisar se certificar que algum componente isolado do sistema – CPU, memória, barramento, etc. – está funcionando corretamente.

Devido às restrições apresentadas é que sistemas embarcados usualmente são depurados remotamente, ou seja, o depurador é executado em uma máquina – o chamado *host computer* – e controla o sistema alvo remotamente utilizando algum hardware e/ou software auxiliar. A vantagem é clara: o desenvolvedor pode utilizar uma ferramenta robusta, rodando em seu ambiente de desenvolvimento, enquanto que o sistema embarcado a ser analisado não precisa nem executar algum software extra que possibilite a depuração remota [16].

Nessa seção apresentaremos algumas estratégias de hardware para depuração de sistemas embarcados. No próximo capítulo, algumas ferramentas (software) serão analisadas. Vale ressaltar que em ambos os momentos, apenas os conceitos mais importantes para o foco do trabalho serão apresentados, e que existem várias outras estratégias disponíveis.

2.2.1 *In-Circuit Emulator*

Um emulador *In-Circuit*, ou *ICE* apenas, é um dispositivo que é conectado a um *host computer* e ao sistema embarcado alvo e que acaba por substituir o processador central do sistema para fins de depuração. Em sistemas que não dispõem de sinais internos para depuração, um *ICE* atua como uma ponte entre o sistema alvo e o *host* executando o depurador [3]. Os sinais, dados e instruções que antes deveriam seguir para o processador agora passam pelo *ICE*, que possui uma versão emulada do processador alvo para possibilitar o processo, e que vai ser utilizada no lugar do processador real graças a essa nova rota que os dados devem seguir.

Essa estratégia oferece uma visão não-intrusiva do fluxo de execução do sistema alvo e permite tanto uma abordagem passiva de depuração quanto uma abordagem mais ativa – quando o fluxo de execução, o estado da memória ou dos registradores pode ser alterado [16].

A desvantagem desse método é a necessidade de um dispositivo extra, que muitas vezes precisa ser adquirido separadamente dos sistemas sendo desenvolvidos. Ainda assim, é a técnica de depuração remota mais antiga já desenvolvida e que influenciou todas as posteriores, inclusive a que utilizaremos nesse trabalho.

2.2.2 *On-Chip Debug Module*

Com o aumento do uso de *SoC's*, ou *System-on-Chips*, o uso de *ICE's* para depuração de sistemas embarcados se tornou problemático. Barramentos de alta freqüência e processadores centrais mais complexos contribuíram para o aumento da complexidade de desenvolvimento dos antigos emuladores e quase que inviabilizaram seu uso em muitos cenários.

Por outro lado, tornou-se possível que um módulo inteiro de depuração fosse projetado e construído em um único chip. São os chamados *On-Chip Debug Module*, ou *OCD Module*. Estes são módulos adicionados aos processadores que trazem as mesmas funcionalidades que os antigos *ICE's* mas sem a sobrecarga de um emulador. São, portanto, mais flexíveis, uma vez que não precisam substituir o processador alvo e atuam, na verdade, em paralelo com o mesmo. Várias famílias de processadores já saem de fábrica com esse módulo e, portanto, não há a necessidade de se adquirir um hardware externo para poder fazer uso de tal método de depuração.

Um canal de comunicação serial de alta freqüência é utilizado para se efetuar a conexão do *host computer* com o *debug module* e, assim, uma porta de comunicação entre os sistemas se faz necessário. Dito isso, temos um gancho ideal para a próxima tecnologia a ser apresentada: o padrão IEEE 1149.1, comumente chamado de padrão *JTAG*.

2.2.3 Padrão *JTAG*

O aumento da complexidade no design de chips de silício, ou circuitos integrados, diminuiu consideravelmente a eficácia dos métodos tradicionais de teste de hardware [21]. Em 1985, um consórcio entre os maiores fabricantes mundiais de circuitos integrados foi formado. Esse grupo se auto-intitulava "*The Joint Test Access Group*", ou *JTAG*, e era focado em desenvolver padrões de testes para *CI's* e *PCB's – printed circuit boards*. O trabalho desenvolvido por esse grupo foi de tamanho reconhecimento que acabou sendo aprovado pelo Instituto de Engenheiros Eletricistas e Eletrônicos, ou *IEEE*, como o padrão *IEEE Std. 1149-1*: "*Standard Test Access Port and Boundary-Scan Architecture*". Ou, simplesmente, o padrão *JTAG* [20].

O protocolo foi projetado de tal maneira que com apenas uma porta de acesso pode-se testar todos os chips presentes em uma placa. Isso só é possível pois os sinais *JTAG* de cada chip são interconectados seguindo uma configuração chamada *Daisy-chain* – quando um chip A está conectado a um chip B que está conectado a um chip C, e assim por diante. Em *JTAG*, são cinco os sinais:

- **TDI** (*Test Data In*): Dados inseridos no dispositivo;
- **TDO** (*Test Data Out*): Dados lidos do dispositivo;
- **TCK** (*Test Clock*): Sinal de clock;
- **TMS** (*Test Mode Select*): Controla a máquina de estados *JTAG*;
- **nTRST** (*Test Reset*): Sinal opcional para *reset* assíncrono do dispositivo.

O comportamento do protocolo é serial, já que temos apenas uma linha de dados (TDI e TDO). Os outros sinais – TMS, TCK e opcionalmente nTRST – formam o chamado controlador TAP, ou *Test Access Port controller*. A [figura 1] nos ajuda a resumir isso:

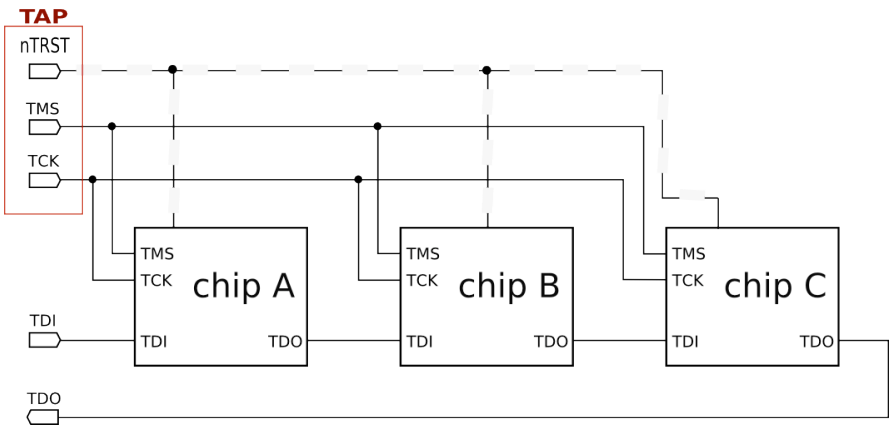
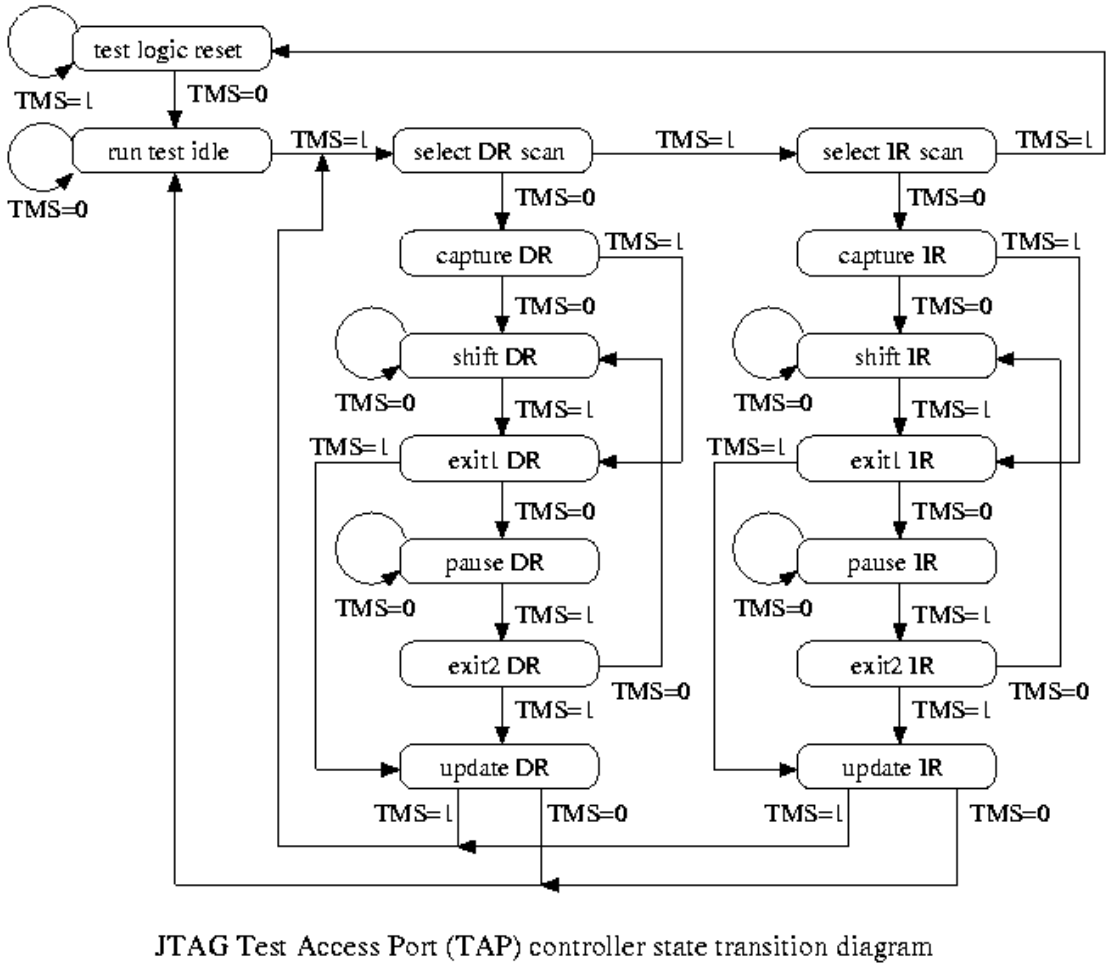


Figura 2.1 - *JTAG chain*

Esse controlador, *TAP,* é responsável pela seleção interna de registradores e por toda a máquina de estados que controla o fluxo de dados na placa, quando em modo *JTAG*. Nesse contexto, o sinal TCK é responsável pela sincronia interna da máquina de estados, TMS é o sinal que determina o próximo estado e nTRST, quando disponível, é o sinal que pode re-iniciar a máquina de estados *JTAG*. O estado inicial de operação é o estado *Test-Logic-Reset*, ou *TLR*, que é atingido após 5 ciclos de TCK com TMS em nível lógico alto. Quando nesse estado, se TMS estiver em nível lógico baixo em um dado pulso de TCK, a máquina de estados entrará em *Run-Test/Idle*. Os próximos estados serão, então, um conjunto formado por seleção, captura, deslocamento, pausa, fim e atualização de registradores de dados (DR) ou de registradores de instrução (IR). A [figura 2] pode nos ajudar a visualizar a máquina de estados *JTAG* [18]:



JTAG Test Access Port (TAP) controller state transition diagram

Figura 2.2 - Máquina de Estados *JTAG*

Em um dispositivo, do ponto de vista externo, *JTAG* se apresenta como uma porta de acesso à placa. Usualmente, essa porta se apresenta em 3 configurações de pinos: 8 pinos, 14 pinos e 20 pinos. A decisão de qual configuração utilizar cabe ao fabricante do dispositivo, mas, usualmente, a configuração de 20 pinos é utilizada na arquitetura *ARM* e a configuração de 8 pinos é utilizada em *FPGA's*. Aqui não será apresentada a relação sinal-pino para cada configuração; isto pode ser encontrado em [11].

Com o passar dos anos, esse padrão ganhou confiança na indústria eletrônica, mas sua verdadeira adoção veio após 1994. Nesse ano, um suplemento foi adicionado ao padrão *IEEE 1149.1*, lançando a *BSDL – Boundary Scan Description Language.* Essa linguagem é uma especialização de *VHDL* e é utilizada para modelagem de dispositivos que sejam compatíveis com *Boundary Scan*, ou seja, com *JTAG*. Ela foi projetada para ser capaz de descrever como *JTAG* é implementado em um dado dispositivo, de maneira simples, clara e direta. Em [22] pode-se encontrar uma descrição mais detalhada da linguagem, mas como ela só será utilizada como um critério de avaliação no próximo capítulo, quaisquer outras informações que aqui fossem apresentadas fugiriam do escopo desse trabalho.

Conforme visto, o padrão *JTAG* foi criado para possibilitar uma nova metodologia de teste de hardware alternativa à tradicional *In-Circuit Tests*, ou *ICT*. Porém, com a ampla adoção do padrão, verificou-se que o mesmo poderia ser utilizado para outros fins. Atualmente, *JTAG* é largamente utilizado para programação de *FPGA's*, por exemplo, e para programação de memórias *Flash* do tipo *NOR* e *NAND* [19]. Entretanto, é uma última característica de *JTAG* que tornou este o padrão ideal a ser utilizado nesse trabalho: seu mecanismo de depuração de sistemas embarcados.

De fato, o próprio projeto desse padrão garante que ele terá acesso a sub-blocos dos circuitos integrados. Quando estamos falando de um processador que foi fabricado com um *On-Chip Debug Module*, e que o dispositivo segue o padrão 1149.1, temos que a porta *JTAG* se transforma em um mecanismo de transporte desse módulo. Em outras palavras, temos uma "*backdoor*" real no sistema, tanto para fins de depuração quanto para fins de engenharia reversa. No capítulo 3, será apresentada de que maneira o padrão *JTAG* foi realmente utilizado nesse trabalho.

2.3 Protocolo Remoto GDB

O famoso projeto *GNU Debugger*, além de um excelente depurador, também provê um protocolo para depuração remota – o chamado *GDB Remote Serial Protocol* [23]. Esse protocolo serve como base para aqueles que, por motivos diversos, necessitem implementar um servidor *GDB*. Clientes também podem ser implementados, mas a situação mais típica é aquela na qual um desenvolvedor implementa um servidor e utiliza o próprio *GDB* como um cliente depurador.

A estrutura do protocolo em si é bastante simples, sendo este um típico protocolo cliente-servidor: estabelece-se uma conexão, uma requisição é feita, obtêm-se uma resposta, envia-se um *ACK* – de *acknowledgement*, reconhecimento – e assim até o fim da conexão. As mensagens também são simples, sempre seguindo o formato:

$packet-data#checksum

Isso nos mostra que um pacote sempre se inicia com um caractere '$', que é seguido pela instrução do protocolo até que um caractere '*#*' seja encontrado. Em seguida, temos os dois dígitos de *checksum,* que indicam se o pacote chegou sem erros. Esse valor é calculado como o módulo 256 da soma de todos os caracteres da instrução, ou seja, de todos os caracteres presentes entre '$' e '*#*'. Grande parte das instruções, ou comandos, do protocolo *GDB* são escritas puramente em caracteres *ASCII*. As demais instruções – como leitura/escrita de memória/registradores, por exemplo – fazem uso dos formatos binário e hexadecimal. Vale destacar também que no servidor *GDB* oficial e em alguns outros *stubs*, os pacotes podem estar codificados em formato chamado *Run-Length Encode,* que é uma técnica utilizada para redução de tamanho de *strings* com repetição de caracteres. Os pacotes que por ventura iniciem com '+$' representam um *ACK* para o último pacote trocado [5].

Devemos ressaltar que não existe exatamente um mapeamento "um para um" das instruções desse protocolo com suas respostas; é de certo modo comum que diferentes comandos levem aos mesmos resultados. É uma especificação bastante flexível que se por um lado a torna poderosa, por outro cria situações conflitantes quando se tem a intenção de se desenvolver um cliente que suporte diferentes implementações de servidor *GDB* [5]. Os comandos do protocolo serão apresentados no decorrer desse trabalho e, por hora, as informações fornecidas devem bastar – mas a especificação completa pode ser encontrada em [23].

Uma vez que a solução a ser desenvolvida foi escolhida, o protocolo remoto serial *GDB* teve um papel fundamental na continuação desse projeto. Isso será explicado no capítulo 4.

3 ESTADO DA ARTE

Uma vez que os conceitos básicos necessários ao entendimento desse trabalho já foram revisados no capítulo anterior, chega-se o momento de apresentar tecnologias que foram estudadas para o desenvolvimento da solução proposta. Cabe, portanto, apresentar e avaliar as ferramentas existentes, de hardware e software, que pertencem ao contexto proposto – depuração de sistemas embarcados via JTAG.

Para tal, na seção que se segue apresenta-se alguns modelos de cabos *JTAG* existentes e softwares que utilizam este padrão como interface entre computadores e sistemas embarcados. Após a apresentação destas ferramentas, faz-se uma avaliação das mesmas. Na última seção do presente capítulo, o framework ERESI, alvo deste trabalho, é enfim apresentado.

3.1 *JTAG Interfacing*

Comunicação com uma placa de sistema embarcado via *JTAG* só é possível com a utilização de alguns componentes. Para qualquer que seja o fim – programação de memória flash, depuração, engenharia reversa, etc. – precisa-se de um cabo *JTAG* e de uma ferramenta que implemente o protocolo de comunicação com este padrão sendo executada no computador *host*. É a este conjunto que denomina-se *JTAG interfacing*. Para auxiliar na visualização deste termo, façamos uso da seguinte figura:

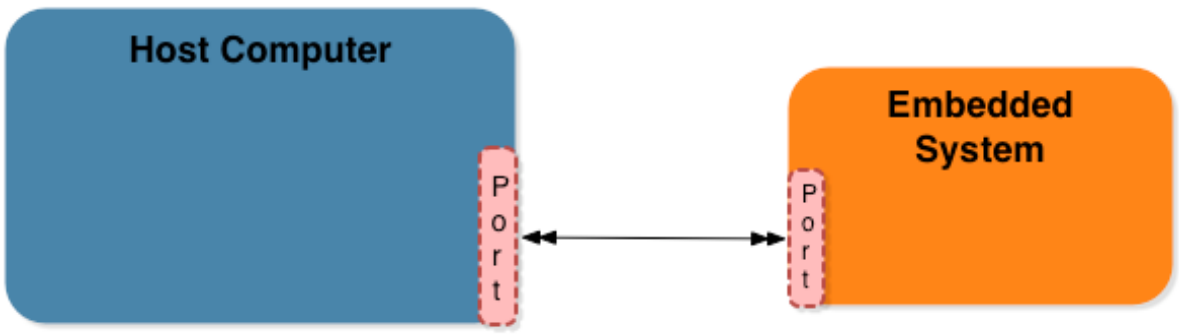


Figura 3.1 - *JTAG interfacing*

A caixa da direita é o sistema embarcado alvo, a da esquerda é o *host computer* e as portas de comunicação servem para conectar o cabo *JTAG*. Essa figura e seus elementos serão estendidos no decorrer do texto, representando os novos conceitos que forem introduzidos. Na verdade, apenas a caixa da esquerda sofrerá alterações, visto que este

projeto foca no depurador remoto a ser executado no *host*. Pensemos no sistema embarcado como o alvo de estudos deste trabalho e não de desenvolvimento.

3.1.1 Hardware

No capítulo anterior, foi dito que as placas que implementam o padrão *JTAG* possuem uma porta para tal tipo comunicação e que estas se apresentam em 3 configurações possíveis. Em [11] temos a configuração sinal-pino para cada uma das distribuições de pinos.

Um cabo *JTAG* possui em uma de suas pontas um conector de 8, 14 ou 20 pinos, para ser ligado ao sistema embarcado na porta *JTAG*, e na outra ponta um conector compatível com alguma porta existente nos computadores atuais: *USB*, paralela ou serial. É comum que cada fabricante de placas produza um cabo próprio a ser utilizado com seus produtos, mas, em teoria, deveria ser possível se utilizar qualquer cabo *JTAG* desde que a configuração de pinos – ou "pinagem" – seja respeitada [20].

Os cabos *USB* tem se tornado bastante comuns na indústria, visto que as outras portas, serial e paralela, quase caíram em desuso. Entretanto, existe um cabo paralelo em específico que merece certa atenção: o modelo conhecido como *Wiggler* compatível. Este foi originalmente fabricado pela *Macgraigor's Systems*, mas foi rapidamente copiado por outros fabricantes. A simplicidade de seu circuito – vide [28] – o torna um cabo que pode, inclusive, ser construído em casa. Suas características o mantém como um dos cabos mais utilizados até hoje [11]. Entretanto, devido ao desuso das portas paralelas, pode ser necessário um conversor USB-Paralelo de forma a podermos conectar o cabo *Wiggler* a computadores mais modernos.

Nossa plataforma de testes, ainda a ser apresentada no capítulo 5, utilizará um modelo de cabo *Wiggler* compatível, fabricado pela Olimex [10], como visto abaixo:

Figura 3.2 - ARM-JTAG da Olimex

Hora de continuarmos com o próximo elemento que compõe nossa interface *JTAG*: um software para comunicação.

3.1.2 Software

Antecedendo o desenvolvimento deste projeto, era necessário pesquisar outras ferramentas disponíveis que se parecessem de alguma forma com a nossa proposta. Buscou-se entender soluções que já haviam sido propostas e implementadas, para comunicação com sistemas embarcados via *JTAG*.

Uma vez que essas soluções foram estudadas, decidiu-se que algumas delas deveriam ser de fato testadas. Isso nos trouxe um melhor entendimento de como a comunicação *host*-placa ocorria e quais eram as funcionalidades disponíveis em cada solução. Antes que uma avaliação sobre estes softwares possa ser apresentada, façamos uma introdução sobre os mesmos:

- **GDBICE** [29]: é uma ferramenta que possibilita uma interface de comunicação entre o *GDB* e sistemas embarcados com *CPU ARM7TDMI*, apenas se utilizando cabos *Wiggler* compatíveis. Sua última versão data do ano 2000 e possui alguns elementos até obsoletos em sua implementação – como o modo de acesso à porta paralela, por exemplo. O projeto inteiro é constituído de apenas um arquivo e suas funcionalidades se resumem a estabelecer uma conexão e ler endereços de memória. Além disso, não existe documentação do projeto;

- **JTAGER** [30]: é um depurador que estabelece conexão com sistemas embarcados das arquiteturas *ARM7TDMI, ARM9TDMI e ARM920T*. Sua última versão, 0.3.0, data de 2004, sendo mais completa que o *GDBICE*. Possibilita leitura e escrita de registradores e leitura e escrita de memórias *RAM* e *Flash*, além do estabelecimento de uma conexão *host*-placa via *JTAG*. Possui documentação tanto para desenvolvedores quanto para usuários e também só suporta cabos *Wiggler* compatíveis;

- **JTAGPACK** [31]: é a única que se apresenta tanto como um conjunto de ferramentas *JTAG* como uma *API* – uma biblioteca – para desenvolvimento de outras aplicações que necessitem fazer uso deste padrão. Suporta apenas cabos *Wiggler* e as arquiteturas *ARM7TDMI* e *MIPS*. Possui um servidor *GDB* e uma *engine* básica para controle da máquina de estados *JTAG* – controlador *TAP*. Sua última atualização data de 2005. Mesmo sendo uma das ferramentas com mais funcionalidades encontradas – vide tabela na próxima seção – entender por completo sua estrutura e funcionamento tornou-se virtualmente impossível, uma vez que a documentação disponível – dentro do código-fonte, inclusive – está toda escrita em japonês;

- **UrJTAG** [14]: pode ser definido como uma ferramenta para comunicação via *JTAG*. Foi construído sobre um projeto mais antigo, chamado *Openwince*. Pretende se tornar um conjunto universal de ferramentas, bibliotecas e servidores para o

padrão *JTAG*, mas por enquanto foca em placas *FPGA's*. Suporta cabos e placas de vários fabricantes, como pode ser conferido em [32]. Possui uma comunidade de desenvolvimento e sua última atualização data de abril de 2009, até o presente momento. Atualmente, uma *API* externa está sendo desenvolvida, então dentro de pouco tempo uma biblioteca de interface *JTAG* deverá estar disponível;

- **OpenOCD** [15]: é uma solução completa para depuração de arquiteturas *ARM7*, *ARM9*, *Cortex-M3* e *XSCALE*, que teve início com uma tese de mestrado [16]. Suporta cabos dos modelos *Wiggler* compatíveis e *FT2232*, para portas *USB*. Provê um servidor *GDB* – para leitura/escrita de memória e registradores, além de *breakpoints*, parar/resumir execução do sistema alvo além de execução por passos – e ferramentas para leitura e programação de memórias *Flash*. Suporte a arquitetura *OMAP* da *Texas Instruments* está em andamento, assim como uma melhoria na integração com ambientes de desenvolvimento – *IDE's*, como o *Eclipse*. Possui uma comunidade de desenvolvimento bastante ativa tendo atualizações semanais em seu código-fonte. Além disso, possui uma ampla documentação disponível para usuários e desenvolvedores, que pode ser conferida em [15], [16] e [17].

Vale ressaltar que as soluções analisadas tratam-se todas de projetos *OpenSource*. Esse foi um caminho natural a ser seguido uma vez que acesso ao código-fonte era um pré-requisito para se entender as possíveis abordagens de interface para comunicação *JTAG*. Além disso, conforme apresentamos no capítulo 1, essa pesquisa também se propõe a desmistificar esse senso comum que trata depuração de sistemas embarcados como uma atividade "cara", que só pode ser feita com ferramentas comerciais. Apenas se utilizando do cabo *JTAG* correto e de uma das ferramentas supra-citadas, o leitor pode executar atividades próprias de depuração em um sistema embarcado qualquer, ainda que as funcionalidades apresentadas sejam simples. Não cabe em nosso contexto, entretanto, avaliarmos se as ferramentas comerciais são melhores ou não que as livres; o objetivo principal desta seção se mantém: entendermos e avaliarmos métodos de *interfaceamento JTAG*.

De fato, analisar e testar essas ferramentas foi o que possibilitou que as possíveis arquiteturas para este projeto fossem elaboradas. Isso ficará claro quando da apresentação das mesmas no próximo capítulo. Por hora, faz-se necessário uma avaliação mais objetiva dos softwares testados, para que as diferenças sejam diretamente apresentadas. Isso facilitará, também, uma futura comparação com o depurador desenvolvido neste trabalho.

3.1.3 Avaliação das ferramentas

Algumas métricas foram estabelecidas visando a objetividade da tabela de avaliação. Optou-se por escolher 4 definições gerais e avaliar as ferramentas pelo que elas apresentam em cada uma delas. Foram elas:

- **Abrangência de Suporte:** arquiteturas e cabos suportados;
- **Interfaces disponibilizadas:** o que está disponível para que outras aplicações façam uso da ferramenta;
- **Funcionalidades:** o que está disponível para a ferramenta seja utilizada como depurador e/ou como interface *JTAG*;
- **Integridade e Facilidade de compreensão:** apesar de parecer um tanto quanto subjetivo, a integridade de um projeto *opensource* pode ser avaliado apenas como um reflexo do tamanho e da dedicação de sua comunidade de desenvolvimento. O fator compreensão também se mostra subjetivo, mas a estrutura de um projeto e a documentação disponibilizada diz muito sobre isso.

Essa definições representam uma quantidade de informação suficiente para que as soluções estudadas sejam avaliadas e deve servir como base para decisões em projetos futuros. A [tabela 1] resume o que foi discutido nesta seção:

Tabela 3.1 - Análise de soluções similares

	Abrangência de Suporte	**Interfaces disponibilizadas**	**Funcionalidades**	**Integridade e Facilidade de compreensão**
GDBICE	ARM7TDMI e cabo paralelo Wiggler compatível	Servidor GDB.	Conexão via *JTAG* e leitura da memória.	Abandonado no ano 2000 e sem documentação.
JTAGER	ARM7TDMI, ARM9TDMI e ARM920T. Cabos Wiggler compatíveis	Nenhuma.	Conexão via *JTAG*, Leitura/Escrita de registradores e memórias RAM e Flash.	Abandonado em 2004. Possui documentação para usuários e desenvolvedores.
JTAGPACK	ARM7TDMI e MIPS. Cabos Wiggler.	Servidor GDB e API para uso externo.	Várias ferramentas *JTAG [31]* e uma *engine* de controle	Abandonado em 2005. Sua documentação em

			da máquina de estado TAP.	japonês inviabilizou o estudo da ferramenta.
UrJTAG	Amplo suporte de FPGA's e cabos. Vide [32].	Nenhuma.	Várias ferramentas para comunicação *JTAG*. Suporta descrição de placas em BSDL.	Projeto ativo e em desenvolvimento, sendo bem estruturado. Possui documentação disponível, além de listas de email para os desenvolvedores.
OpenOCD	ARM7, ARM9, Córtex-M3, XSCALE e Omap em desenvolvimento. Cabos Wiggler e FT2232 (USB).	Servidor GDB.	Conexão via *JTAG*, leitura/escrita de memória e registradoes, inserção/remoçao de breakpoints, parar/resumir execução do sistema alvo e execução por passos. Possui ferramentas para programação de memória Flash.	Projeto bastante ativo e com uma ampla comunidade de desenvolvimento. Sua estrutura visa facilitar a adição de novas placas, cabos e arquiteturas. Documentação e canais de discussão estão disponíveis.

Como pode ser visto, as soluções analisadas possuem poucas funcionalidades específicas para depuração ou engenharia reversa de sistemas embarcados. Embora o problema de *interfacing JTAG* seja resolvido pela maioria delas, alguns conceitos básicos de depuradores não são encontrados - nenhuma das ferramentas disponibiliza um *disassembler*, por exemplo. De todos estes projetos, o *OpenOCD* foi o que mais apresentou funcionalidades para fins de depuração – ainda que simples – e impressionou pela facilidade de uso e potencial de expansão. Para este trabalho, esse projeto merece uma atenção especial nos próximos capítulos, assim como o framework que será apresentado na próxima seção.

3.2 ERESI Framework

O *ERESI Reverse Engineering Software Interface* é um framework para engenharia reversa que começou a ser desenvolvido em 2007, sendo baseado em um projeto mais antigo chamado *ELF Shell – elfsh*. Em uma tradução direta de [6], temos que:

> "*ERESI* é um framework unificado para análise binária em múltiplas arquiteturas, com foco em sistemas operacionais baseados no *Executable & Linking Format* (*ELF*), tais como Linux, BSD, Solaris, HP-UX, IRIX e BeOS, desde que executados nas arquiteturas INTEL, SPARC, MIPS, ALPHA ou ARM. Embora muitas das funcionalidades estejam disponíveis para programas em *user-land*, o framework também oferece opções para análise de kernel Linux em execução. *ERESI* é um framework de propósito geral híbrido: ele é capaz de executar análises estáticas e dinâmicas. Essas funcionalidades são acessadas através de nossa própria linguagem de *scripts*. Ela nos traz um ambiente com escolhas para análise de programas através de instrumentação, transformação, depuração e rastreamento de binários. *ERESI* pode ser utilizado também para auditorias de segurança, *hooking*, checagem de integridade ou *logging*. (...)"

3.2.1 Visão Geral

Graças a sua arquitetura modular é que o *ERESI* tem se tornado um projeto tão versátil e abrangente. Em sua estrutura atual, são 6 entidades de alto nível:

- **Elfsh:** um programa interativo a ser utilizado como uma ferramenta para instrumentação estática de binários *ELF*;
- **E2dbg:** um depurador interativo de alta performance para *userland* – espaço computacional das aplicações - que funciona independentemente da *API* de depuração de sistemas operacionais – uma *API* conhecida como *ptrace*. Funciona como um depurador embarcado que é executada em conjunto com o programa alvo, em um mesmo processo. Isto só é possível porque o *backend* do *ERESI* provê um gerenciador que evita corrupção de memória em ambos os programas;
- **Etrace:** o *tracer* – rastreador – embarcado *ELF*. É capaz de rastrear um processo em sua velocidade normal de execução, capturando chamadas internas e externas;
- **Kernsh:** é uma ferramenta – uma *Shell* – para instrumentação de kernels em tempo de execução, capaz de executar injeção de código dinamicamente, modificação e redirecionamento de fluxo de execução. É executada em *userland* e se utiliza da linguagem *ERESI* para representar estruturas próprias de um *kernel*;

- **Evarista:** um analisador estático de programas totalmente implementado na linguagem *ERESI*, que se utiliza de transformação de programas para encontrar de forma automática falhas em código binário. Encontra-se em desenvolvimento;
- **Kedbg:** o depurador remoto de *kernels.* Será explicado na próxima seção.

Todas essas ferramentas, ou entidades de alto nível, são baseadas em um conjunto de 11 bibliotecas, ou entidades de baixo nível:

- **Libelfsh:** a biblioteca de manipulação de binários na qual se baseiam o *ELFsh*, *E2dbg* e *Etrace*;
- **Libe2dbg:** a biblioteca de depuração embarcada utilizada pelo *E2dbg*. É "*linkada*" ao programa alvo em *load-time*;
- **Libmjollnir:** a biblioteca de análise de fluxo e *fingerprinting* – identificador único de dados;
- **Librevm:** contém o *Reverse Engineering Vector Machine*, responsável pelo interpretador da meta-linguagem *ERESI*;
- **Libstderesi:** a biblioteca padrão do *ERESI*, com mais de 100 comandos de análise;
- **Libaspect:** é utilizada para que estruturas de dados e outros aspectos de códigos possam ser refletidos na linguagem *ERESI*;
- **Libedfmt:** responsável pelo formato de depuração *ERESI*;
- **Libetrace:** biblioteca do *tracer ERESI*, na qual a ferramenta *Etrace* se baseia;
- **Libkernsh:** responsável pelo acesso à *kernels*. *Kernsh* utiliza essa biblioteca;
- **Libasm:** a *engine* de *disassembling* com suporte para *x86, SPARC, MIPS e ARM*. Atribui as características sintáticas e semânticas às instruções e seus operandos;
- **Libgdbwrap:** implementa o protocolo remoto serial *GDB*. Utilizada no *Kedbg*.

Todos esses componentes foram projetados de maneira que eles fossem o mais independentes entre si possível. Por isso, o *ERESI* framework é considerado flexível: o usuário pode utilizar apenas alguns componentes ou construir outros novos de acordo com suas necessidades. De fato, algumas das bibliotecas de mais baixo nível – como a libasm ou a libgdbwrap, por exemplo – podem ser utilizadas em outras aplicações externas ao *ERESI*, sem que nenhum outro componente do framework seja compilado ou "*linkado*" [7].

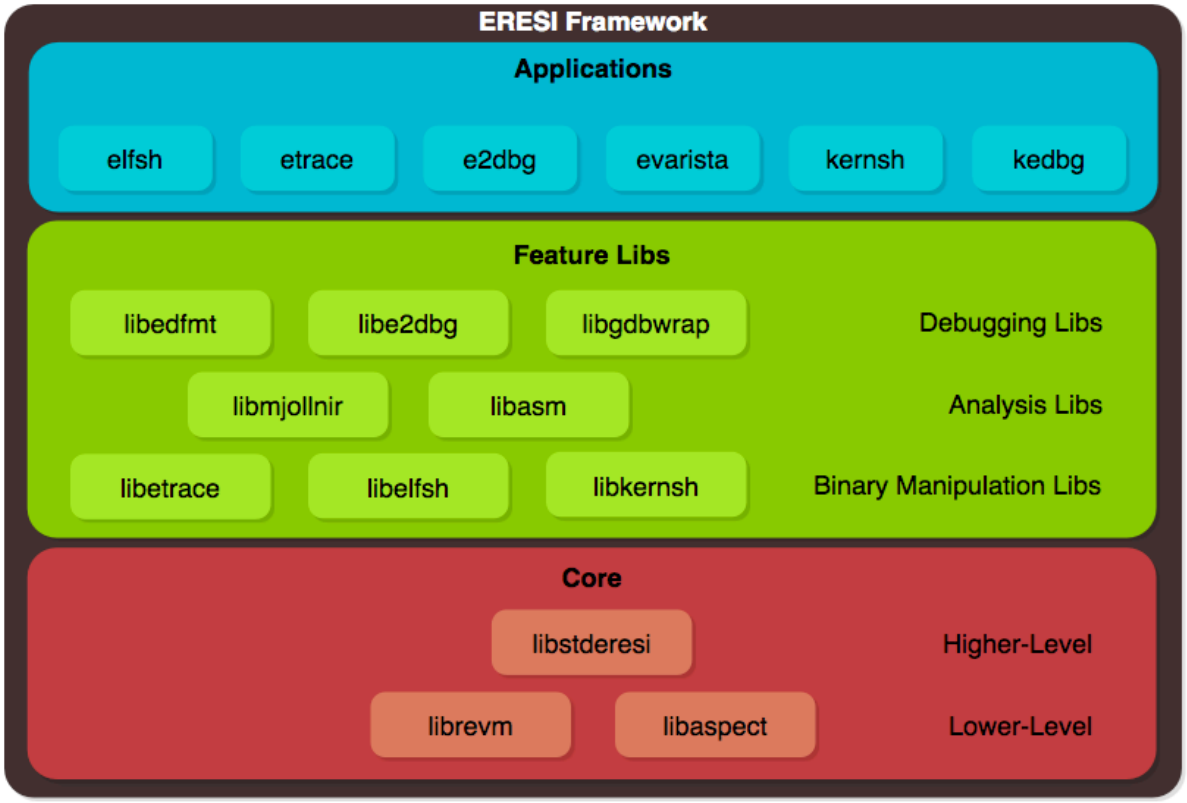


Figura 3.3 - Representação em camadas do framework ERESI

3.2.2 Depuração Remota: KEDBG

Duas aplicações importantes do *ERESI*, *e2dbg* e *etrace*, funcionam em modo embarcado. Elas se inserem dinamicamente no binário alvo, e quando este é executado a aplicação embarcada é carregada em conjunto, tomando o controle do processo. Além de melhorar a performance do processo como um todo, uma vez que a depuração acontece dentro do próprio espaço de endereçamento do processo, esse modo de depuração embarcada permite que este tipo de atividade possa ser efetuada em ambientes que não disponham de API's de depuração. Ou seja, em sistemas onde depuradores que utilizem a biblioteca *ptrace* não possam ser executados. São técnicas complexas de injeção e realocação – como *proxy allocation* –, presentes na libelfsh, que permitem esse modo de operação. O leitor interessado pode encontrar mais explicações sobre o assunto em [4].

Depuração embarcada era o único modo suportado no *ERESI* até o surgimento do *KEDBG*. Esta era a peça que faltava para que o framework fosse capaz de executar análises remotas. Um mundo de possibilidades foi aberto agora que *ERESI* é capaz de depurar software e máquinas virtuais remotamente, por exemplo.

Este depurador foi construído sobre a biblioteca *libgdbwrap*, que é uma implementação do protocolo remoto serial *GDB*, totalmente independente do restante do framework, capaz de efetuar conexões e se comunicar com servidores *GDB* de outras aplicações. Todas as funções comuns a um depurador estão disponíveis: execução passo-

a-passo, *breakpoints*, resumir/pausar execução, escrita/leitura de memória e registradores, etc. Até o início deste trabalho, ela suportava os *stubs GDB*, *VMWare* e *QEmu*, sendo estes dois últimos aplicações para execução de máquinas virtuais.

O *Kedbg* é, portanto, um *front-end* de depuração remota para o *ERESI*. Um depurador que é executado em *ring 0* – ou seja, no nível zero de prioridade, o maior possível em computação. Além da *libgdbwrap*, a *libe2dbg* também é utilizada. Dela é utilizada o conceito de vetores e registro de *hooks* de funções [5]. São oferecidas algumas funções mais avançadas de depuração, tais como *backtracing*, conhecimento do mode de execução da *CPU* alvo (*protected mode x real mode*), *disassembly*, rastreamento de interrupções e instruções, dentre outras.

Em resumo, esta é uma aplicação que nos serve como um exemplo concreto da escalabilidade do framework em questão. Projetos complexos e inovadores na área de engenharia reversa podem ser desenvolvidos de forma simples. O que impressiona, na verdade, é que uma aplicação como o *Kedbg* já nasce herdando uma série de funcionalidades avançadas do *ERESI*. Basta se construir uma nova interface – neste caso a *libgdbwrap* – que o framework se expande semi-automaticamente.

3.2.3 Diferencial do ERESI

Além de tudo que já foi apresentado, o *ERESI* ainda oferece algumas outras funcionalidades que o distingue ainda mais de outros frameworks de seu nicho de aplicação. Para que fique mais claro: adicione a tudo que já foi citado nesta seção o fato de que este framework foi projetado para funcionar em *hardened* ou *raw systems* [6], ou seja, em sistemas que não apresentam símbolos, segmentos executáveis de dados e API nativa de depuração.

Some a isso a habilidade de se gerar grafos, sob demanda, de fluxo de chamadas para funções específicas ou de um processo como um todo, baseando-se em primitivas de análise automática de fluxo. Como se não fosse suficiente, ainda temos a disposição depuração de sistemas *multi-thread*, rastreamento de instruções em tempo real, e várias outras funcionalidades avançadas que se beneficiam de uma linguagem própria de engenharia reversa, com conceitos de programação orientada a aspectos: *ERESI programming language*.

Com tantas portas abertas, não podemos nos queixar de faltas de justificativas para que esse tenha sido o framework escolhido como alvo deste trabalho. Este projeto vai tratar de expandi-lo para um novo patamar em engenharia reversa: sistemas embarcados. Devemos, enfim, seguir para a nossa contribuição. No próximo capítulo, argüiremos sobre as possíveis abordagens para alcançar nosso objetivo, justificaremos a decisão tomada e explicaremos o desenvolvimento da mesma. Note, porém, que neste ponto do texto, apenas considerando a idéia de termos depuração de sistemas

embarcados no *ERESI*, nós já estamos diante de um projeto conceitualmente mais completo que todos os similares apresentados. É mais um estímulo para transformá-lo em realidade.

4 UM DEPURADOR JTAG PARA O ERESI KERNEL DEBUGGER

Revisemos a [figura 3.1] e o objetivo deste trabalho: um depurador remoto *JTAG* para o framework de engenharia reversa *ERESI*. Lembre-se que o depurador e o framework serão executados no *host computer* e se comunicarão com o sistema embarcado através de sua porta *JTAG*. Entretanto, para que isto seja possível é necessária uma interface *JTAG* entre o depurador e o sistema alvo. Além deste protocolo, ela precisa ser capaz de acessar a porta a ser utilizada pelo cabo *JTAG* no computador *host*. Uma expansão da figura citada pode auxiliar nesta contextualização:

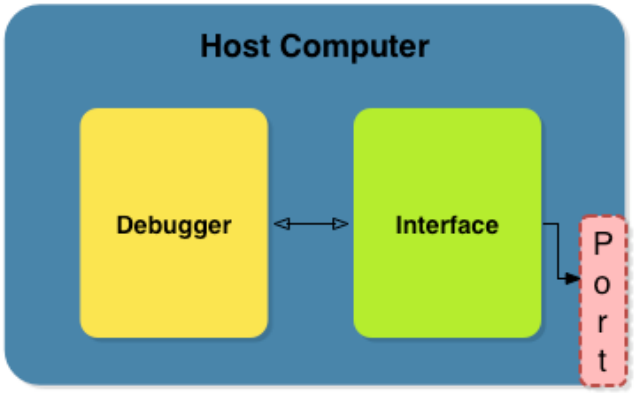


Figura 4.1 - Arquitetura em Alto Nível

A partir deste ponto será dado foco apenas para o diagrama que representa o *host*.

No início da fase de desenvolvimento deste projeto, o cerne do mesmo era justamente essa interface. Após a avaliação e estudo das ferramentas apresentadas no capítulo anterior, três abordagens foram pensadas:

- **Uma biblioteca *JTAG* própria:** a implementação de uma *API* para comunicação *JTAG* que seria utilizada pelo depurador. Vamos chamá-la de *libejtag*;
- **Expansão de um projeto existente:** criar uma *API* externa para uma das ferramentas apresentadas, tornando-a capaz de ser utilizada pelo depurador. Os candidatos aqui seriam os projetos *OpenOCD* ou *UrJTAG*, por suas funcionalidades e abrangência. Vamos chamá-la de *liburjtag*;
- **Arquitetura cliente-servidor com Protocolo *GDB*:** tornar o *ERESI* capaz de se comunicar com uma das ferramentas apresentadas através do protocolo *GDB*, sendo o *OpenOCD* o projeto ideal para este caso.

Todas essas soluções possuem prós e contras. A *libejtag* era o modelo preferido no início do projeto pois seria uma implementação própria do protocolo para o *ERESI*, o que

deveria garantir alta performance na comunicação e uma interface independente de outros projetos. Por outro lado, a abrangência de arquiteturas seria reduzida. Conforme foi explicado nos capítulos anteriores, existem diferentes implementações do protocolo *JTAG* variando entre placas, arquiteturas e fabricantes. No tempo disponível para conclusão deste projeto, estimou-se que só seria possível a conclusão da biblioteca suportando talvez apenas uma arquitetura e uma porta disponível – paralela, serial ou *USB*. O depurador em si ficaria ameaçado de não ser implementado e o escopo deste trabalho mudaria drasticamente. Em resumo, suporte mono-arquitetural e viabilidade do cronograma foi o que nos levou a descartar essa abordagem.

Expandir um projeto existente traria uma base de código estável e testada, possivelmente com suporte a múltiplas arquiteturas. Além disso, optar por um projeto como o UrJTAG, por exemplo, que possui muitas funcionalidades mas nenhuma *API* externa (vide tabela 3.1), traria uma contribuição real para tal projeto. Um levantamento efetivo dos requisitos necessários para essa abordagem tornou-se difícil uma vez que isto exigiria uma completa imersão no projeto escolhido. A curva de aprendizado poderia comprometer a viabilidade do cronograma e, sendo assim, esse modelo também foi descartado.

Utilizar o servidor *GDB* de uma das ferramentas apresentadas também nos traria estabilidade e maior suporte a arquiteturas. Além disso, essa seria a abordagem mais tangível para que o objetivo deste trabalho fosse atingido, uma vez que parte da interface estaria pronta de antemão. O projeto *OpenOCD* atende a todos os requisitos citados, possuindo uma estrutura facilmente expansível e compreensível (vide tabela 3.1) e um servidor *GDB* para comunicação com aplicações externas. O framework *ERESI* possui um depurador remoto, o *KEDBG*, que se utiliza da *libgdbwrap* para se comunicar via protocolo *GDB*. Unindo-se esses pontos temos uma solução que contempla nosso objetivo fazendo reuso de ferramentas existentes, que sofreriam as alterações e atualizações necessárias. E, enfim, nossa contribuição.

4.1 A solução: *KEDBG-JTAG*

Uma expansão da [figura 4.1] nos serve de apoio para a visualização da arquitetura proposta nesta solução:

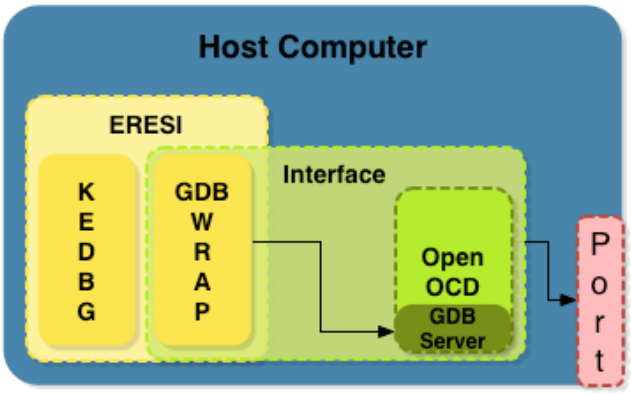


Figura 4.2 - Arquitetura Final KEDBG-JTAG

Conforme explicado na seção 3.2, a *libgdbwrap* (e por conseguinte o *KEDBG*) suporta o servidor *GDB* padrão além dos servidores do *QEmu* e do *VMWare*. Agora ela precisa suportar o servidor da ferramenta *OpenOCD*, ou seja, ser capaz de efetuar conexão e suportar o modo de operação e o formato das mensagens do protocolo *GDB* que ainda não tenham sido implementadas. Já o *KEDBG* por sua vez possui uma estrutura própria para depuração remota de *kernels*, com estados e funções específicas para lidar com esse nicho [5]. Como sistemas embarcados estavam fora do escopo de seu desenvolvimento inicial, será necessário uma adaptação dessa estrutura, adicionando as definições, funções e estados necessários. Nasceria assim o projeto *KEDBG-JTAG*: o depurador remoto de *kernels* e *JTAG* do framework *ERESI*.

Explicando em uma única frase: o *KEDBG-JTAG* se utiliza da *libgdbwrap* para se conectar ao servidor *GDB* do *OpenOCD* - daqui em diante, vamos chamar esse servidor de *OpenOCD-GDB* - que por sua vez se comunica via protocolo *JTAG* com o sistema embarcado alvo. Para que estes objetivos fossem atingidos, o desenvolvimento deste projeto foi sub-dividido em 3 grandes etapas:

1. Modificar a *libgdbwrap* e o *KEDBG* possibilitando o estabelecimento de conexão com o servidor *OpenOCD-GDB*;

2. Atualizar as estruturas internas da biblioteca e do depurador para que os novos comandos sejam suportados;

3. Atualizar as estruturas internas da biblioteca e do depurador para que a arquitetura *ARM* seja suportada.

Essas etapas são constituídas de várias tarefas menores que, em conjunto, formam a história de desenvolvimento desse projeto. Nas próximas seções e subseções deste capítulo elas serão explicadas, mas para uma melhor compreensão do desenvolvimento como um todo o leitor pode se dirigir ao código fonte do projeto, presente em [33].

4.2 Implementação

Sendo nossa arquitetura um modelo cliente-servidor, a primeira etapa de desenvolvimento deste trabalho precisava garantir que as partes envolvidas pudessem se conectar e se comunicar. Era preciso entender o funcionamento do servidor *OpenOCD-GDB* para que então se fosse possível adicionar suporte ao mesmo na *libgdbwrap*.

Para isso fez-se uso de um *sniffer* de tráfego chamado *Wireshark*. Com essa ferramenta, e utilizando-se o *GDB* como um cliente para o *OpenOCD-GDB*, vários testes foram executados e *traces* da comunicação foram capturados. Esses trechos capturados foram analisados e assim o modo de operação do servidor alvo pode ser compreendido. De um modo resumido, pode-se dizer que o fluxo de comunicação apresentado foi (rever o formato de mensagens do protocolo remoto *GDB* na seção 2.3 pode ajudar):

1. *Handshake* inicial entre cliente e servidor;
2. O cliente envia a mensagem ***$qSupported#37***;
3. O servidor *OpenOCD-GDB* responde com ***$PacketSize=3fff;qXfer:memory-map:read+;qXfer:features:read-;QStartNoAckMode+#08***;
4. O cliente requisita então um *memory map*, com a mensagem ***$qXfer:memory-map:read::0,fff#1c*** , onde zero é o *offset* e *0xfff* é o tamanho a ser lido;
5. O servidor responde com o mapa da memória do sistema alvo, que apresentaremos mais a frente;
6. A conexão está estabelecida e a depuração já pode ser iniciada.

No passo 2, a mensagem do tipo *$qSupported* funciona como uma pergunta para o servidor sobre o que ele suporta, quais comandos ou modos de operação. Todas as mensagens e respostas do protocolo remoto *GDB* podem ser conferidas em [23], mas neste trabalho focaremos naquelas que influenciam diretamente no desenvolvimento do *KEDBG-JTAG*. No passo 3 verificamos que o servidor *OpenOCD-GDB* responde afirmando que suporta:

- ***PacketSize=3fff*:** indica o tamanho máximo de um pacote para o servidor. Nesse caso temos *0x3fff*, ou 16383 bytes;
- ***qXfer:memory-map:read+*:** indica que o servidor entende a requisição de *memory-map*, sendo capaz de enviar um pacote resposta com tais informações;
- ***qXfer:features:read-*:** indica que o servidor não é capaz de responder uma mensagem que requisite a descrição da plataforma alvo;

- ***QStartNoACKMode+*:** indica que o servidor pode trabalhar em *No ACK Mode*, ou seja, sem que mensagens *ACK – Acknowledgement* – sejam trocadas entre cliente-servidor após cada mensagem.

A biblioteca *libgdbwrap* possui uma função de *parsing* desse tipo de mensagem e se utiliza dela para ler esses valores e construir uma estrutura na memória – *struct gdbwrap_t* - com informações próprias da conexão em questão [5]. Ela armazena, por exemplo, o *PacketSize* para determinar qual o tamanho a ser utilizado nos pacotes construídos durante uma conexão. O modo de operação No*ACK* não era suportado e foi necessário adaptações para tal, adicionando uma variável – *flag* – na estrutura que indicasse se a conexão operava com *ACK's* ou não. Essa informação passou a ser sempre checada antes que uma mensagem desse tipo fosse enviada ao servidor ou que o gerenciador da conexão ficasse travado em *loop* a espera de um *ACK* do servidor.

A informação mais relevante que encontramos nessa resposta ao *$qSupported*, entretanto, foi a indicação de que o *OpenOCD-GDB* é capaz de informar um *memory-map*, um mapa que representa a estrutura da memória do sistema embarcado alvo. No tráfego capturado, passo 4, notamos que o cliente fazia uso dessa informação ao requisitar o mapa de memória. Como nossos testes foram executados com uma plataforma real, que será apresentada no próximo capítulo, a resposta que obtivemos do servidor no passo 5 é real e representa a estrutura da memória do sistema alvo em formato *XML*:

$l<memory-map><memory type="flash" start="0x0" length="0x40000"><property name="blocksize">0x2000</property></memory><memory type="ram" start="0x40000" length="0xfffc0000"/></memory-map>#0e

Essa resposta nos diz que o sistema alvo possui memória *Flash* com início em 0x0, tamanho 262144 bytes (0x40000) e blocos de 8192 bytes (0x2000), além de memória *RAM* com início em *0x40000* e tamanho 0xfffc0000. Em [23] podemos conferir que essa mensagem poderia conter mais uma parte *<memory />* para representar a *ROM* do sistema. A ausência dessa informação é um indicativo de que a plataforma alvo não possui memória *ROM*.

A *libgdbwrap* não era capaz de lidar com mensagens do tipo *qXfer* e, portanto, não podia requisitar, armazenar ou lidar com *memory-maps*. As definições e funções necessárias foram adicionadas a biblioteca, assim como as seguintes estruturas:

```
typedef struct meminfo_t
{
  int       type;
  u_int     start;
  u_int     length;
  u_int     blocksize;
} meminfo_t;
```

```
typedef struct gdbmemap_t
{
  meminfo_t     ram;
  meminfo_t     rom;
  meminfo_t     flash;
} gdbmemap_t;
```

A *struct meminfo_t* representa as informações de uma memória e a *struct gdbmemap_t* guarda a informação de todas as memórias lidas do *memory-map*.

A verdadeira importância desses mapas de memória é que o depurador *KEDBG-JTAG* foi projetado para ser executado recebendo um arquivo binário *ELF* como parâmetro, o binário alvo a ser analisado. Quando não se possui o arquivo *ELF* a disposição, o *KEDBG* precisa construir uma estrutura *ELF* simples, para servir como guia no processo de depuração e análise. No modo de depuração de *kernels*, quando não se possuía o *ELF*, a função *kedbg_biosmap_load()* era chamada e um arquivo binário era construído a partir das informações do mapa de *bios*. Agora no *KEDBG-JTAG*, adicionou-se a função *kedbg_jtagmap_load()*. Ela cria um objeto *ELF* fazendo uso de funções da *libelfsh* – *elfsh_create_obj()* – e precisa inserir múltiplos *Program Header*s (*PHDR)* no objeto *ELF*, sendo um para cada descrição de memória presente na estrutura *gdbmemap_t*. Com isso, têm-se um objeto *ELF* que representa o alvo da depuração.

Além disso, outras alterações na estrutura original do *KEDBG* foram necessárias. Com esse novo modo de depuração remota sendo desenvolvido, adicionou-se uma variável para indicar em que estado o depurador estava sendo executado, além de definições para 3 modos de operação: *KEDBG_USERLAND, KEDBG_VM e KEDBG_EMBEDDED* – depurando em *userland*, ou *kernel* em uma máquina virtual ou um sistema embarcado via *JTAG*, respectivamente. Na função de execução principal, *kedbg_main*, checa-se se um *memory-map* foi encontrado ou se o arquivo *ELF* fornecido possui a seção *.vectors_ram*. Em ambos os casos entra-se no estado *KEDBG_EMBEDDED*. Com essas modificações, bastou um pequeno *refactor* no código para distribuir a estrutura original nos estados de *userland* e *vm*, e assim o código do *KEDBG-JTAG* só precisou ser adicionado na máquina de estados no estado *embedded*.

Basicamente, quando os passos acima descritos foram conquistados, *KEDBG-JTAG* (e conseqüentemente a *libgdbwrap*) se tornou capaz de efetuar e manter conexão com o servidor *OpenOCD-GDB*, armazenando um *memory-map* corretamente e tendo um fluxo de execução pronto para lidar com sistemas embarcados. Vale ressaltar que várias outras funções, variáveis, estruturas, definições, enfim, blocos de código, foram adicionados tanto ao depurador quanto à biblioteca. No final deste capítulo o leitor pode conferir nas tabelas 4.1 e 4.2 um resumo do que foi acrescentado à *libgdbwrap* e ao *KEDBG*, respectivamente.

Com esse primeiro *milestone* alcançado, o projeto pode-se mover para uma nova etapa. Era chegada a hora de se adicionar os comandos que faltavam para que funções intrínsecas a um depurador pudessem ser executadas.

4.2.1 Suportando novos comandos

Uma vez que foi possível estabelecer-se uma conexão entre o *KEDBG-JTAG* e o *OpenOCD-GDB*, pode-se iniciar a segunda fase de desenvolvimento deste projeto: suporte a comandos de depuração. Alguns funções já existentes no *KEDBG* apenas precisaram que novos *handlers* fossem registrados e passaram a funcionar com sistemas embarcados. Em [2] encontra-se toda documentação necessária para a compreensão do mecanismo de *handlers* e *hooks* de funções do framework *ERESI*, mas vejamos um exemplo:

```
elfsh_register_readmem(ELFSH_OS_ARM,  ELFSH_IOTYPE_GDBPROT, kedbg_readmem);
```

Na linha acima uma função da *libelfsh* está sendo chamada para que seja registrado um novo *handler* para a função de leitura de memória *readmem*. De uma forma literal, essa linha nos diz que para a arquitetura *ARM* (*ELFSH_OS_ARM*), se o tipo de entrada/saída a ser utilizado for o protocolo *GDB* (*ELFSH_IOTYPE_GDBPROT*), a função *readmem* deve chamar a função *kedbg_readmem* sempre que for requisitada. *Handlers* semelhantes foram registrados para outras funções – como a de escrita de memória, por exemplo. Assim podemos garantir que a *libelfsh* e outras bibliotecas do *ERESI* sempre saibam quais funções devem ser chamadas dependendo do contexto de operação atual do framework. É graças a esse mecanismo que novas arquiteturas podem ser suportadas pelo *ERESI*. O leitor interessado pode fazer uso de [2] caso queria compreender melhor o assunto.

Apenas registrando-se os *handlers* necessários corretamente, nós fomos capazes, por exemplo, de executar comandos para se obter informações sobre o sistema alvo (comandos *e, s e p*), para leitura da tabela de símbolos (comando *symtab*), fazer *dump* de regiões específicas da memória (comando *X*) e de executar o "disassemble" do conteúdo da memória (comando *D*). Além disso, uma vez que a tabela de símbolos já podia ser compreendida o nome de funções ou de seções podia ser utilizado no lugar de endereços de memória em formato hexadecimal. Note que esses comandos já existiam no *KEDBG* e necessitaram de pouquíssimo código extra para que passassem a ser suportados - adição de algumas definições e registro de novos *handlers*. Outras funções apresentadas em [5] e que não foram listadas acima também estavam disponíveis já nesse ponto do desenvolvimento. O resultado dos testes com esses comandos será apresentado no próximo capítulo, e deverá servir como auxílio na compreensão da aplicação de cada um.

Faltavam, portanto, comandos mais específicos do processo de depuração: *step*, *resume, reset* e *breakpoint*. Quando foi feita a captura de tráfego com o *sniffer Wireshark* – explicado na seção anterior – aproveitou-se para entender como o *OpenOCD-GDB* implementava esses comandos. Verificamos que todas essas operações eram feitas através da instrução *monitor*, que utiliza a mensagem "*qRcmd*" do protocolo *GDB* [23]. Isso significa que o servidor *OpenOCD-GDB* possui implementações próprias para esses

comandos e que eles devem ser chamados sempre com o comando *monitor* como prefixo. A resposta para essas mensagens de *remote commands* do protocolo *GDB* – *$qRcmd* – sempre são mensagens de *output* de texto - *$O#* - e, portanto, precisam ser "parseadas" se quisermos imprimir o retorno desses comandos.

Com essas informações, resolvemos então adicionar suporte ao comando *monitor* ao console do *KEDBG*, assim como aos sub-comandos citados:

- *monitor step:* "anda" em um passo na execução. Ou seja, incrementa em 4 bytes o registrador *PC – program counter*;
- *monitor halt*: coloca o sistema alvo em modo *halt*. Para a execução do sistema;
- *monitor reset_halt*: reinicia o sistema alvo o coloca em modo *halt*;
- *monitor resume*: resume a execução do sistema alvo a partir do ponto atual;
- *monitor breakpoint ADDRESS SIZE*: coloca um *breakpoint* no endereço *ADDRESS* e de *SIZE* bytes. Lembre-se que a arquitetura *ARM* só suporta *breakpoints* de 2 ou 4 bytes de tamanho, dependendo do modo de operação [34].

Funções para converter o retorno dessas funções para *ASCII* também foram adicionadas à *libgdbwrap*.

Feito isso, já podíamos considerar que tínhamos um depurador para sistemas embarcados via *JTAG* funcionando no framework *ERESI*. Graças a todo trabalho executado anteriormente nas bibliotecas *libelfsh*, *libe2dbg* e *libgdbwrap*, o processo de mudança para esse novo modo de operação do *KEDBG* ocorreu de forma simples. Além disso, a equipe de desenvolvimento do *ERESI* estava executando um bom trabalho na adaptação da *libasm* para a arquitetura *ARM*, o que possibilitou que este trabalho fosse capaz de suportar as operações de *disassembling* de instruções. A última etapa para que o depurador ficasse um pouco mais completo seria suportar a estrutura de registradores *ARM* e ter funções para leitura e escrita destes, o que nos traria também o suporte a grafos de chamada de funções. Tínhamos então uma última etapa bem definida.

4.2.2 Suportando arquitetura ARM

O suporte ao repertório de instruções *ARM* no *ERESI* é fornecido pela *libasm*. O que faltava na estrutura do *framework* era uma estrutura apropriada para salvar o contexto de registradores *ARM* e as funções para manipulação dessas informações.

Representamos os registradores na forma da *struct gdbwrap_gdbARMreg32*:

```
typedef struct  gdbwrap_gdbARMreg32{
ureg32  r0;
```

```
ureg32  r1;
ureg32  r2;
ureg32  r3;
ureg32  r4;
ureg32  r5;
ureg32  r6;
ureg32  r7;
ureg32  r8;
ureg32  r9;
ureg32  r10; //SL reg
ureg32  r11; //FP reg
ureg32  r12;
ureg32  r13_usr; //SP reg (XXX: add pointers with these names also?!)
ureg32  r14_usr; //LR reg
ureg32  r15; //PC reg
ureg32  r8_fiq;
ureg32  r9_fiq;
ureg32  r10_fiq;
ureg32  r11_fiq;
ureg32  r12_fiq;
ureg32  r13_fiq;
ureg32  r14_fiq;
ureg32  r13_irq;
ureg32  r14_irq;
ureg32  r13_svc;
ureg32  r14_svc;
ureg32  r13_abt;
ureg32  r14_abt;
ureg32  r13_und;
ureg32  r14_und;
ureg32  cpsr;
ureg32  spsr_fiq;
ureg32  spsr_irq;
ureg32  spsr_svc;
ureg32  spsr_abt;
ureg32  spsr_und;
} gdbwrap_gdbARMreg32;
```

Desde a versão 3 da arquitetura *ARM* os processadores dessa família possuem 37 registradores de 32 bits, sendo 30 de propósito geral. Destes, 15 estão sempre disponíveis – r0 a r14. Por convenção da linguagem *assembly ARM*, r13 é utilizado como *stack pointer*, r15 é o *program counter* (*PC*) e r11 armazena o *frame pointer* (*FP*) [35].

Processadores *ARM* podem operar em 7 modos distintos:

- ***User Mode (USR)*:** o modo usual de operação, utilizado na execução da maioria das aplicações;
- ***Fast Interrupt (FIQ)*:** modo com suporte a transferência de dados e processamento de canais;
- ***Interrupt (IRQ)*:** para manipulação genérica de interrupções;

- ***Supervisor Mode (SVC):*** um modo protegido, utilizado pelo sistema operacional;
- ***Abort Mode (ABT):*** é acionado sempre que ocorre um cancelamento de *prefetch* de dados ou instrução;
- ***System Mode:*** equivalente ao *User Mode* mas com privilégios;
- ***Undefined Mode (UND):*** quando da execução de uma instrução indefinida.

Cada modo de operação tem seu próprio contexto de registradores. É por isso que representamos registradores repetidamente na estrutura *gdbwrap_gdbARMreg32* – por exemplo, temos os registradores *r13_usr*, *r13_fiq*, *r13_svc*, etc. Alguns registradores assumem papéis diferentes dependendo do modo de operação. O registrador r14 em modo *usr*, por exemplo, armazena o endereço de retorno quando uma sub-rotina é chamada, mas armazena o endereço de retorno de uma exceção quando o modo de operação suporta *exeception handling* [36].

É o registrador *CPSR – Current Program Status Register* – que armazena a flag que indica o modo de operação atual do processador. Temos também os 5 registradores *SPSR – Saved Program Status Register* – que salvam o valor de *CPSR* sempre que uma exceção ocorre.

A arquitetura *ARM* também dispõem de dois estados distintos de operação, que indicam qual repertório de instruções está sendo utilizado. Quando no *ARM state* as instruções de 32 bits são utilizadas, enquanto que no *Thumb State* as instruções possuem 16 bits. Nem todos os processadores *ARM* suportam *Thumb State*, e por isso o padrão é que eles sempre iniciem em *ARM State* e depois mudem para o outro estado explicitamente através de uma instrução especial chamada *BX – branch and Exchange instruction set* [37].

Essa dualidade de estados é um problema que cabe à biblioteca *libasm*. Em nosso escopo, *libgdbwrap* e *kedbg*, a preocupação era suportar a estrutura de registradores *ARM*, além da capacidade de se identificar o modo de operação para que os registradores corretos fossem utilizados. O leitor interessado em mais informações sobre a arquitetura *ARM* pode seguir para [34], [35], [36] e [37].

Além da estrutura já apresentada, foi necessário que algumas funções para manipulação desses registradores fossem implementadas. Estas fizeram uso de funcionalidades da *libe2dbg*, de maneira semelhante a que as funções de manipulação de memória fizeram uso da *libelfsh*. Por exemplo, em:

```
e2dbg_register_gregshook(ELFSH_ARCH_ARM, ELFSH_HOST_GDB, ELFSH_OS_ARM,
kedbg_get_regvars_ARM);
```

Temos um *hook* para a função de leitura de registradores sendo registrado. Literalmente, sempre que arquitetura *ARM* (*ELFSH_ARCH_ARM* e *ELFSH_OS_ARM*) estiver sendo manipulada e o *host* for um servidor *GDB* (*ELFSH_HOST_GDB*), a *libe2dbg* saberá que sua função de *get_registers* deverá chamar a função *kedbg_get_regvars_ARM*. Seguindo esse mesmo modelo foram registrados *hooks* para funções de escrita em registradores (*kedbg_set_regvars_ARM*), imprimir o valor de todos os registradores (*kedbg_print_ARMreg*), além de funções para o retorno do valor de registradores específicos (*kedbg_getpc_ARM, kedbg_getfp_ARM, etc.*) e para o endereço de retorno (*kedbg_getret_ARM*). Essa funções foram devidamente implementadas e registradas, e com isso passaram a fazer uso deste mecanismo de *handlers* e *hooks* do *ERESI*. Mais uma vez, fica a referência para a documentação contida em [2].

Conforme foi explicado, foram várias as funções e definições adicionadas à *libgdbwrap* e ao *KEDBG* para o desenvolvimento do *KEDBG-JTAG*. As duas tabelas a seguir concentram e resumem o significado de cada uma destas funções e nos servem, portanto, como um fechamento deste capítulo:

Tabela 4.1 - Resumo da implementação adicional na *Libgdbwrap*

meminfo_t	É a estrutura que define um bloco de memória do *memory-map*. Possui variáveis que armazenam o tipo, endereço de início, tamanho e *blocksize* da memória.
gdbmemap_t	Possui 3 estruturas *meminfo_t*, uma para cada tipo de memória: *RAM*, *ROM* ou *Flash*.
gdbwrap_memorymap_get	Função que requisita o *memory-map* para o sistema alvo e retorna uma estrutura *gdbmemap_t* armazenando o resultado.
gdbwrap_init_memap	Função que aloca memória necessária para a estrutura gdbmemap_t e inicializa ela com zeros.
gdbwrap_gdbARMreg32	Estrutura que representa o contexto de registradores *ARM*.
gdbwrap_readgenARMreg	Função que requisita o valor dos registradores do sistema alvo e retorna a estrutura *gdbwrap_gdbARMreg32* preenchida com esses valores.
gdbwrap.is_no_ack_mode	Uma variável do tipo *boolean* que indica se a *libgdbwrap* está operando em *NoACK mode* ou não.

Tabela 4.2 - Resumo da implementação adicional no *Kedbg*

kedbg_file_is_embedded	Função que indica se um dado arquivo *ELF* é ou não de um sistema embarcado.

kedbg_is_embedded	Função que indica se o alvo é um sistema embarcado ou não, consultando o arquivo *ELF* fornecido ou verificando se o alvo fornece ou não um *memory-map*.
kedbg_jtagmap_load	Função que constrói um objeto *ELF* a partir do *memory-map* fornecido.
definições para comandos *monitor*	Vários *#defines* para os comandos *monitor*. São *MONITORSTEP*, *MONITORHALT*, *MONITORRESUME*, *MONITORRESETHALT*, para os comandos *step*, *halt*, *resume* e *reset_halt*, respectivamente.
cmd_kedbgmonitor	Função que adiciona os comandos *monitor* ao console (*prompt*) do *KEDBG*, executando a função *gdbwrap_remotecmd* com o *define* correto do comando sendo utilizado.
kedbg_print_reg_ARM	Função que imprime o valor atual dos registradores. É utilizada pelo comando *dumpregs*, do *KEDBG*.
kedbg_getpc_ARM	Função que retorna o valor do registrador *PC*, *Program Counter*. Na arquitetura *ARM* temos que o registrador *PC* é o registrador r15.
kedbg_getfp_ARM	Função que retorna o valor do registrador *FP*, *Frame Pointer*. Na arquitetura *ARM* temos que o registrador *FP* é o registrador r11.
kedbg_get_latest_ret_ARM	Função que retorna o último *return address* armazenado no registrador r14
kedbg_set_regvars_ARM	Armazena valores da estrutura interna da *libe2dbg* nos registradores.
kedbg_get_regvars_ARM	Função que executa uma leitura dos valores atuais dos registradores e armazena na estrutura interna da *libe2dbg*.

5 AMBIENTE DE TESTES E DEMONSTRAÇÃO

Tendo todo o processo de desenvolvimento do depurador *KEDBG-JTAG* sido explicado, podemos seguir para uma pequena demonstração de utilização do mesmo. O intuito aqui é apresentar o ambientes de testes utilizado e como feito esse *setup*, além de apresentar as informações que puderam ser obtidas durante uma sessão de depuração do sistema embarcado.

5.1 Ambiente utilizado

O ambiente de testes utilizado como sistema alvo durante o desenvolvimento deste trabalho foi uma placa de prototipação *LPC-E2124*, da fabricante *Olimex* [8]:

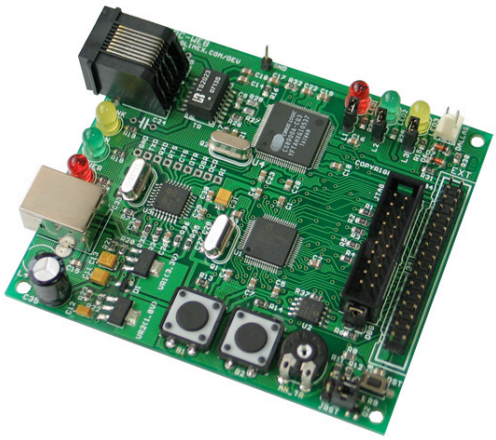


Figura 5.1 - Placa LPC-E2124 da Olimex

O cabo utilizado foi um cabo paralelo *Wiggler compatível* chamado *ARM-JTAG*, também da *Olimex* [10], apresentado na [figura 3.2].

Uma vez que a placa estava pronta para ser utilizada, seguimos o tutorial presente em [38] para que ela fosse programada com o sistema operacional *FreeRTOS*, executando um servidor *Web*. A ferramenta *CrossWorks for ARM* [38] foi utilizada para programação da *flash* com esse exemplo pronto. Bastou que um arquivo fosse alterado para que a placa obtivesse as configurações de rede desejadas – endereço ip, *netmask*, etc. – e assim ela estava pronta para ser acessada por uma máquina na mesma sub-rede. O resultado era o acesso à uma página *HTML* simples, que guardava um *log* de todos os acessos efetuados:

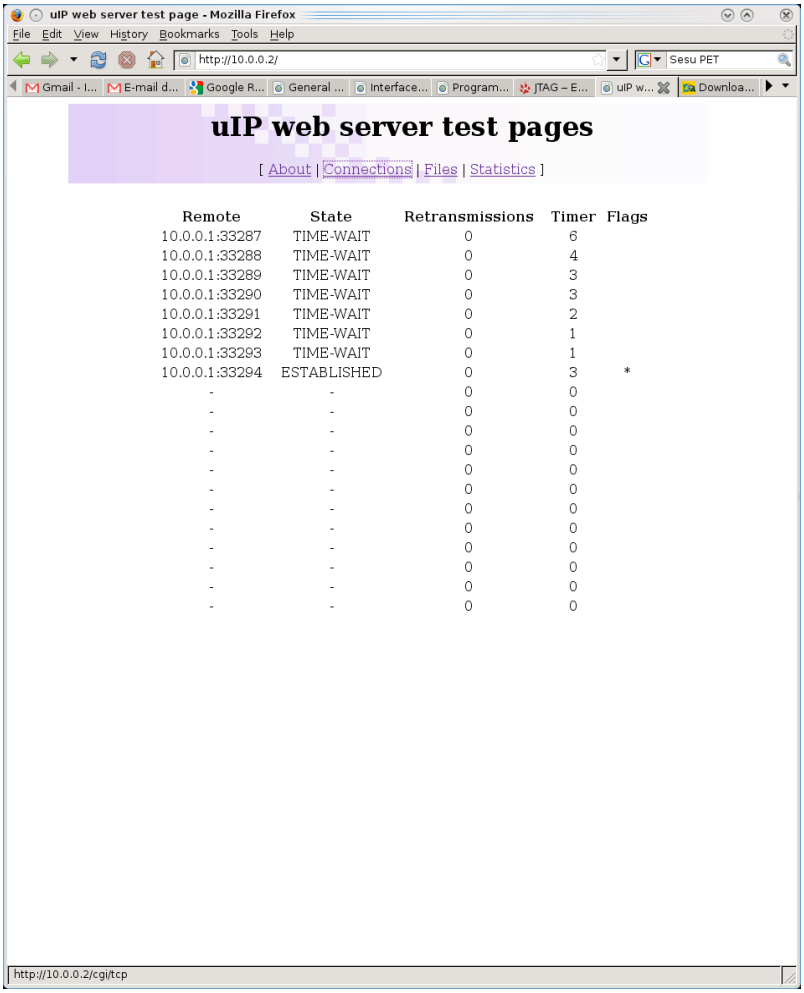


Figura 5.2 - Página de testes sendo executada no sistema embarcado

Isso configura um ambiente ideal para testes, visto que é simples de se saber se o sistema ainda está em execução ou não. Bastava que se clicasse em um link da página de testes e tínhamos um retorno imediato do *status* de execução do sistema. Caso a página não respondesse e tivéssemos um *timeout* da requisição sabíamos que o sistema não estava mais executando. Isto foi importante para testarmos os comandos de *halt* e *resume*, por exemplo.

Com o *setup* da placa finalizado, era necessário que ela pudesse se conectar com o *OpenOCD* (lembre-se da arquitetura apresentada na seção 4.1 deste trabalho). Seguindo a documentação desta ferramenta e contando com auxílio de alguns de seus desenvolvedores, pudemos escrever um arquivo de configuração próprio para ser utilizado com a placa em questão:

```
#daemon
telnet_port 4444
gdb_port 5555
#interface
interface parport
parport_port 0
```

```
parport_cable wiggler

#LPC-2124 CPU based on LPC-2129

if { [info exists CHIPNAME] } {
  set  _CHIPNAME $CHIPNAME
} else {
  set  _CHIPNAME lpc2124
}
if { [info exists ENDIAN] } {
  set  _ENDIAN $ENDIAN
} else {
  set  _ENDIAN little
}
if { [info exists CPUTAPID ] } {
  set _CPUTAPID $CPUTAPID
} else {
 # force an error till we get a good number
   set _CPUTAPID 0x4f1f0f0f
}
#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst
jtag_nsrst_delay 10
jtag_khz 1000


#jtag scan chain
jtag newtap $_CHIPNAME cpu -irlen 4 -ircapture 0x1 -irmask 0xf -expected-id $_CPUTAPID
set _TARGETNAME [format "%s.cpu" $_CHIPNAME]
target create $_TARGETNAME arm7tdmi -endian $_ENDIAN -chain-position $_TARGETNAME -variant
arm7tdmi-s_r4
$_TARGETNAME configure -work-area-virt 0 -work-area-phys 0x40000000 -work-area-size 0x2000 -work-
area-backup 0
#flash bank <driver> <base> <size> <chip_width> <bus_width>
flash bank lpc2000 0x0 0x40000 0 0 0 lpc2000_v1 14745 calc_checksum
```

Nesse arquivo, utilizando-se a linguagem de *script* própria da ferramenta *OpenOCD*, define-se desde a porta em que o servidor *OpenOCD-GDB* será executado até toda a estrutura de memória e o processador utilizado na placa alvo. Define-se também qual interface *JTAG* será utilizada. Mais informações sobre os arquivos de configuração de placas e arquiteturas em [15].

5.2 Uma sessão de depuração

A melhor maneira de demonstrar o depurador *KEDBG-JTAG* em execução é através de uma sessão de depuração. Ou seja, ver quais informações puderam ser obtidas e como a ferramenta nos apresenta esses dados. Para tal, vamos dividir a sessão em 4 etapas: conexão e informações iniciais, *dump* de memória e *disassembling* de instruções, funções de depuração, leitura de registradores e análise em grafos. No decorrer desta seção, a

saída de texto real da ferramenta será apresentada e explicada, mas em vários pontos ela foi truncada para economia de espaço.

5.2.1 Conexão e primeiras informações

Começamos executando a ferramenta, informando qual servidor deverá ser utilizado e qual arquivo *ELF* servirá como "mapa":

```
jeez@andaluzia ~/usr/usr/local/bin $ ./kedbg localhost:5555 /home/jeez/rtosdemo.elf
---MemoryMap---
--Flash--
-start:0x0
-length:262144
-blocksize:8192
--RAM--
-start:0x40000
-length:4294705152
--ROM--
-start:0x0
-length:0
 [*] No configuration in ~/.eresirc
 [*] Wed May 13 20:11:18 2009 - New object loaded : /home/jeez/rtosdemo.elf
(kedbg-0.82-b2-dev@local)
```

Quando a execução ocorre com sucesso, temos o *prompt* de comando do *KEDBG* esperando por uma entrado do usuário. Note que as informações de *memory*-map obtidas foram impressas antes do *prompt*. O comando *help* lista as opções disponíveis. Podemos então seguir para comandos que nos forneçam mais informações sobre o sistema alvo, analisando o arquivo *ELF*:

```
(kedbg-0.82-b2-dev@local) e
 [ELF HEADER]
 [Object /home/jeez/rtosdemo.elf, MAGIC 0x464C457F]
 Architecture      :          ARM   ELF Version      :          1
 Object type       : Executable object   SHT strtab index   :          20
 Data encoding     :    Little endian   SHT foffset      :    0000087080
 PHT foffset       :      0000000052   SHT entries number  :          23
 PHT entries number  :            3   SHT entry size     :          40
 PHT entry size    :           32   ELF header size    :          52
 Runtime PHT offset  :      1179403657   Fingerprinted OS   :      Unknown
 Entry point       :      0x00000040   [?]
 {OLD PAX FLAGS = 0x606}
 PAX_PAGEEXEC      :        Disabled   PAX_EMULTRAMP      :  Not emulated
 PAX_MPROTECT      :      Restricted   PAX_RANDMAP       :   Randomized
 PAX_RANDEXEC      :   Not randomized   PAX_SEGMEXEC      :      Enabled

(kedbg-0.82-b2-dev@local) p
 [Program Header Table .::. PHT]
 [Object /home/jeez/rtosdemo.elf]
 [00] 0x00000000 -> 0x000083D0 rwx memsz(00033744) foffset(00000160) filesz(00033744)
align(00000016) => Loadable segment
```

```
[01] 0x4000003C -> 0x4000007C rw- memsz(00000064) foffset(00033904) filesz(00000064) align(00000004) => Loadable segment
[02] 0x4000007C -> 0x40003650 rw- memsz(00013780) foffset(00033968) filesz(00000000) align(00000004) => Loadable segment

[SHT correlation]
[Object /home/jeez/rtosdemo.elf]
[*] SHT is not stripped
[00] PT_LOAD .vectors .init .text .rodata
[01] PT_LOAD .data
[02] PT_LOAD

(kedbg-0.82-b2-dev@local) s
[SECTION HEADER TABLE .::. SHT is not stripped]
[Object /home/jeez/rtosdemo.elf]
[000] 0x00000000 ------- foffset:00000000 size:00000000 link:00 info:0000 entsize:0000 align:0000 => NULL section
[001] 0x00000000 a-x---- .vectors foffset:00000160 size:00000060 link:00 info:0000 entsize:0000 align:0004 => Program data
[002] 0x0000003C a-x---- .init foffset:00000220 size:00000496 link:00 info:0000 entsize:0000 align:0016 => Program data
[003] 0x0000022C awx---- .text foffset:00000716 size:00026668 link:00 info:0000 entsize:0000 align:0016 => Program data
[004] 0x00006A58 a------ .rodata foffset:00027384 size:00006520 link:00 info:0000 entsize:0000 align:0004 => Program data
[005] 0x4000003C aw----- .data foffset:00033904 size:00000064 link:00 info:0000 entsize:0000 align:0004 => Program data
[006] 0x40000000 -w----- .vectors_ram foffset:00033968 size:00000060 link:00 info:0000 entsize:0000 align:0001 => Program data
[007] 0x4000007C aw----- .bss foffset:00033968 size:00013780 link:00 info:0000 entsize:0000 align:0004 => BSS
[008] 0x4000003C -w----- .data_run foffset:00034028 size:00000064 link:00 info:0000 entsize:0000 align:0001 => Program data
[009] 0x40003650 -w----- .stack_irq foffset:00034092 size:00000512 link:00 info:0000 entsize:0000 align:0001 => Program data
[010] 0x40003850 -w----- .stack_svc foffset:00034604 size:00000512 link:00 info:0000 entsize:0000 align:0001 => Program data
[011] 0x00000000 ------- .debug_abbrev foffset:00035116 size:00005096 link:00 info:0000 entsize:0000 align:0001 => Program data
[012] 0x00000000 ------- .debug_info foffset:00040212 size:00026210 link:00 info:0000 entsize:0000 align:0001 => Program data
[013] 0x00000000 ------- .debug_line foffset:00066422 size:00006492 link:00 info:0000 entsize:0000 align:0001 => Program data
[014] 0x00000000 ------- .debug_frame foffset:00072916 size:00004232 link:00 info:0000 entsize:0000 align:0004 => Program data
[015] 0x00000000 ------- .debug_loc foffset:00077148 size:00004472 link:00 info:0000 entsize:0000 align:0001 => Program data
[016] 0x00000000 ------- .debug_pubnames foffset:00081620 size:00003013 link:00 info:0000 entsize:0000 align:0001 => Program data
[017] 0x00000000 ------- .debug_aranges foffset:00084640 size:00000776 link:00 info:0000 entsize:0000 align:0008 => Program data
[018] 0x00000000 ------- .debug_str foffset:00085416 size:00001023 link:00 info:0000 entsize:0000 align:0001 => Program data
[019] 0x00000000 ------- .comment foffset:00086439 size:00000414 link:00 info:0000 entsize:0000 align:0001 => Program data
[020] 0x00000000 ------- .shstrtab foffset:00086853 size:00000225 link:00 info:0000
```

```
entsize:0000 align:0001 => String table
 [021] 0x00000000 ------- .symtab                foffset:00088000 size:00010992 link:22 info:0433 entsize:0016
align:0004 => Symbol table
 [022] 0x00000000 ------- .strtab                foffset:00098992 size:00007014 link:21 info:0000 entsize:0000
align:0001 => String table

(kedbg-0.82-b2-dev@local) sym
 [SYMBOL TABLE]
 [Object /home/jeez/rtosdemo.elf]
 [000] 0x00000000 NOTYPE   (NULL)                          size:0000000000 foffset:000000 scope:Local
sctndx:00 => .vectors
 [001] 0x00000000 SECTION  (NULL)                          size:0000000000 foffset:000000 scope:Local
sctndx:01 => .vectors
(...)
[040] 0x00000160 NOTYPE   memory_copy                      size:0000000036 foffset:000512 scope:Local
sctndx:02 => .init + 292
 [041] 0x00000184 NOTYPE   memory_set                      size:0000000000 foffset:000548 scope:Local
sctndx:02 => .init + 328
 [042] 0x00000128 NOTYPE   ctor_loop                       size:0000000032 foffset:000456 scope:Local
sctndx:02 => .init + 236
 [043] 0x00000148 NOTYPE   ctor_end                        size:0000000000 foffset:000488 scope:Local
sctndx:02 => .init + 268
 [044] 0x00000148 FUNCTION start                           size:0000000020 foffset:000488 scope:Local
sctndx:02 => .init + 268
 [045] 0x0000015C NOTYPE   exit_loop                       size:0000000056 foffset:000508 scope:Local
sctndx:02 => .init + 288
(...)
```

Os comandos *e*, *p*, *s* analisam o arquivo *ELF* do alvo e nos fornecem respectivamente os *headers* deste arquivo, a tabela de *Program Headers* e a tabela de *headers* das seções do arquivo, respectivamente. Além destes, temos o comando *sym*, que nos fornece uma tabela completa dos símbolos (nome de seções, funções, etc.) encontrados no arquivo binário *ELF*. Essas informações podem servir como um guia inicial para o processo de depuração/engenharia reversa do sistema alvo.

5.2.2 Lendo a memória e decifrando instruções

Agora podemos seguir para a análise da memória do sistema. Escolhemos uma das seções disponíveis, mas poderíamos ter utilizado um endereço de memória em formato hexadecimal (0x0000ffff, por exemplo):

```
(kedbg-0.82-b2-dev@local) X .text
[*] Analysing section .text [*]
 0000022D [foff: 00000717] print_stats + 0          00 C0 9F E5 1C FF 2F E1 61 41 00 00 00 C0 9F E5
......../.aA.....
 0000023D [foff: 00000733] print_stats + 16         1C FF 2F E1 6D 40 00 00 78 47 C0 46 00 95 00 1A
../.m@..xG.F....
 0000024D [foff: 00000749] print_stats + 32         60 0F 4B 1B 68 1A 1C 0C 4B 1B 68 13 60 0C 4B 1B
```

```
`.K.h...K.h.`.K.
(...)
000003A1 [foff: 00001089] tcp_stats + 0                B5 82 B0 03 1C 01 AA 13 70 01 AB 1B 78 00 2B 02
........p...x.+.
 000003B1 [foff: 00001105] tcp_stats + 16               D0 01 23 00 93 1C E0 11 4B 1A 68 11 4B 1A 60 0F
..#.....K.h.K.`.
(...)
```

O comando *X* executa um *dump* na memória, imprimindo seu conteúdo. Graças a tabela de símbolos encontrada (veja a saída do comando *sym*, na seção anterior) o depurador é capaz de associar o nome das funções aos endereços de memória lidos. Na saída acima, temos uma parte das funções *print_stats* e *tcp_stats*.

Mas essas informações obtidas estão em hexadecimal. Conforme foi explicado, o *ERESI* possui a biblioteca *libasm* própria para efetuar *disassembling* de arquiteturas suportadas. Com o comando *D* temos então:

```
(kedbg-0.82-b2-dev@local) D .text
 [*] Analysing section .text [*]
 0x0000022D [foff: 717] print_stats + 0      cmpmi r4, r0, lsl #20          00 4A 54 41
 0x00000231 [foff: 721] print_stats + 4      cmnvs r4, r7, asr #32          47 20 74 61
 0x00000235 [foff: 725] print_stats + 8      stc                            70 3A 20 6C
 0x00000239 [foff: 729] print_stats + 12     teqcc r2, r0, ror r3           70 63 32 31
 0x0000023D [foff: 733] print_stats + 16     msr                            32 34 2E 63
 0x00000241 [foff: 737] print_stats + 20     strvct r7, [r0], #-1392!       70 75 20 74
(...)
```

Perceba que é a mesma função *print_stats* sendo analisada, mas agora com suas instruções passando pelo *disassembler*. É possível até sabermos com quais registradores as instruções estão operando.

5.2.3 Depurando

Aqui temos, provavelmente, o conjunto de funções mais úteis, mas que ao mesmo tempo são as mais difíceis de serem demonstradas. Os comandos do tipo *monitor – monitor reset_halt*, *monitor halt*, *monitor resume*, *monitor step* e *monitor breakpoint* – não nos fornecem nenhum tipo de saída, mas seus resultados podiam ser conferidos na alteração de valores dos registradores ou no *status* do sistema alvo, no caso dos comandos de *halt* e *resume*. Para fins ilustrativos, vamos colocar o uso dos comandos durante a sessão de depuração:

```
(kedbg-0.82-b2-dev@local) monitor reset_halt

(kedbg-0.82-b2-dev@local) monitor resume
```

```
(kedbg-0.82-b2-dev@local) monitor halt

(kedbg-0.82-b2-dev@local) monitor step

(kedbg-0.82-b2-dev@local) monitor bp 0x0000ffff 0x4
```

5.2.4 Lendo registradores e obtendo grafos

A forma mais simples de ilustrar o suporte a registradores da arquitetura *ARM* é através do comando *dumpregs*:

```
(kedbg-0.82-b2-dev@local) dumpregs

.:: Registers ::.

    [r0] 00000000 (0000000000) <unknown>

    [r1] 40003650 (1073755728) <.stack_irq@rtosdemo.elf>

    [r2] 00000000 (0000000000) <unknown>

    [r3] 00000000 (0000000000) <unknown>

    [r4] 00000000 (0000000000) <unknown>

    [r5] 00000000 (0000000000) <unknown>

    [r6] 06060606 (0101058054) <____udivsi3_from_thumb@rtosdemo.elf + 101030837>

    [r7] 07070707 (0117901063) <____udivsi3_from_thumb@rtosdemo.elf + 117873846>

    [r8] 08080808 (0134744072) <____udivsi3_from_thumb@rtosdemo.elf + 134716855>

    [r9] 09090909 (0151587081) <____udivsi3_from_thumb@rtosdemo.elf + 151559864>

    [r10] 10101010 (0269488144) <____udivsi3_from_thumb@rtosdemo.elf + 269460927>

    [r12] 00000000 (0000000000) <unknown>

    [r8_fiq] 00000000 (0000000000) <unknown>

    [r9_fiq] 00000000 (0000000000) <unknown>

    [r10_fiq] 00000000 (0000000000) <unknown>

(...)

    [SP (r13_usr)] 00000000 (0000000000) <unknown>

    [FP (r11)] 11111111 (0286331153) <____udivsi3_from_thumb@rtosdemo.elf + 286303936>

    [PC (r15)] 00000184 (0000000388) <start@rtosdemo.elf + 00000060>
```

As outras funções para leitura de registradores são usadas internamente pelo depurador. O comando *dumpregs* imprime o valor atual de todos os registradores, mesmo que estes não tenham valor algum ainda definido.

Por último, mas não menos importante, temos a análise e geração de grafos de fluxo de chamadas. Esta seja talvez uma das funcionalidades mais importantes desse depurador, podendo ser considerada um grande diferencial fornecido pelo framework *ERESI*. São várias opções de geração de grafos, podendo ser passado como parâmetro qualquer função do sistema alvo. Como exemplo real do nosso sistema analisado temos:

```
(kedbg-0.82-b2-dev@local) graph
```

Que nos fornece o seguinte grafo simples:

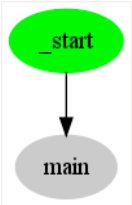


Figura 5.3 - Grafo simples

Passando uma função como parâmetro:

```
(kedbg-0.82-b2-dev@local) graph bloc main
```

Temos um grafo mais completo do fluxo do sistema:

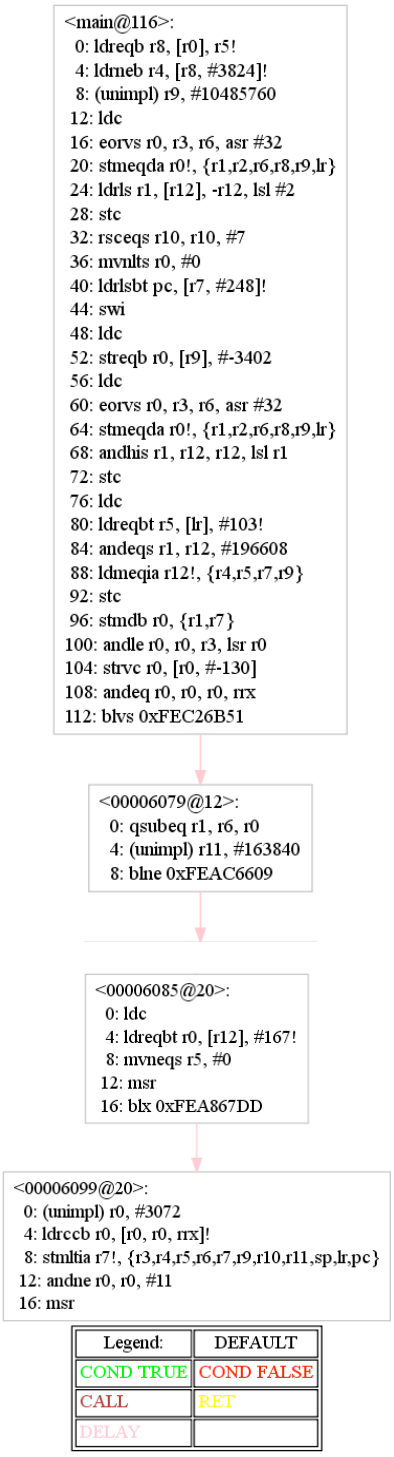


Figura 5.4 - Grafo da função Main

Também podemos utilizar um registrador como parâmetro para geração de um grafo:

```
(kedbg-0.82-b2-dev@local) graph bloc $pc
```

E assim temos:

6 CONCLUSÃO

O objetivo central deste trabalho era o desenvolvimento de um depurador de sistemas embarcados para o framework *ERESI*. Entretanto, além desta ferramenta obtivemos como resultado um amplo trabalho de pesquisa sobre programação, depuração e engenharia reversa de sistemas embarcados, além de um estudo sobre o padrão *IEEE 1149.1* (*JTAG*). Fez-se também um estudo acerca da arquitetura *ARM*, que apontou detalhes de seus modos de operação, registradores e estados de execução. Ferramentas existentes para programação e análise de placas via *JTAG* foram estudadas, testadas e avaliadas, o que mostrou problemas e soluções que outros projetos e desenvolvedores já haviam enfrentado.

Essa massa de informações obtida durante os 4 meses de projeto resultou em um senso crítico, centrado nesta área de pesquisa. Com isso, fomos capazes de propor 3 possíveis arquiteturas para o projeto. Todas elas resultariam em um depurador remoto, mas cada uma possuía suas vantagens e desvantagens frente as outras, conforme apresentamos. Por fim, optamos pela opção que nos ofereceu maior abrangência de placas, arquiteturas, cabos, fabricantes, etc., e que, principalmente, foi plausível de ser implementada no tempo disponível para a conclusão do projeto e deste relatório.

Possivelmente, a solução que envolvia escrever uma biblioteca interface *JTAG* nova, desde o princípio, seria a escolhida pelo projeto *ERESI* pois não dependeria de uma ferramenta externa (*OpenOCD*) e nem de outro protocolo remoto (protocolo remoto *GDB*) além do protocolo *JTAG*. Note, entretanto, que as duas soluções descartadas tornariam o foco deste trabalho o desenvolvimento de uma interface de software para o protocolo *JTAG*, o que fugiria do escopo originalmente proposto.

Ao utilizarmos a solução "*OpenOCD-GDB + libgdbwrap*" como interface *JTAG*, herdamos uma base de código confiável e abrangente, além de todo um conjunto de funcionalidades que a ferramenta original já nos oferecia. Isso possibilitou a construção de um depurador funcional para o framework *ERESI*. O *KEDBG-JTAG* em seu estado atual nos permite parar, resumir e pausar execução do sistema alvo, inserir e remover breakpoints, ler blocos de memória e registradores e ainda nos fornece grafos de execução do sistema, tanto grafos globais como aqueles com início em uma dada função, ou endereço de memória ou registrador. Esta última funcionalidade pode ser considerada um diferencial, pois não está presente em nenhuma das ferramentas avaliadas.

O processo de desenvolvimento deste trabalho contou com a orientação remota dos desenvolvedores do *ERESI*, além da co-orientação fundamental de Julien Vanegue. Semanalmente reuniões *online* ocorriam e todo processo de revisão do código era feito

de maneira distribuída, uma vez que eram 3 fusos horários diferentes envolvidos no trabalho – Brasil, Itália e Canadá. O projeto foi inteiramente desenvolvido em ambiente *Linux*, na linguagem C, resultando em cerca de 28 *commits* no repositório oficial do framework, ou aproximadamente 1600 linhas de código.

6.1 Trabalhos Futuros

Apesar do projeto ter sido concluído com sucesso, algumas funcionalidades ainda merecem mais alguns testes e revisões. Por exemplo, a base necessária para atribuição de valores à registradores foi desenvolvida, mas ainda necessita que algumas funções sejam finalizadas antes que esta funcionalidade possa ser oferecida. Outra pendência é falta de mensagens de retorno para o usuário quando os comandos de depuração *monitor* são utilizados.

Atualmente a função disponível que fornece o último endereço de retorno é a *kedbg_get_latest_ret_ARM*, que utiliza o valor do registrador *cpsr* conforme foi explicado. Ainda é necessário que uma função chamada *kedbg_getret_ARM* seja implementada. Essa função retornará o verdadeiro endereço de retorno atual de uma função em execução, e não apenas o último endereço salvo no registrador *cpsr*. A implementação dessa função se mostrou um tanto quanto complexa, devido a algumas particularidades da arquitetura *ARM* e por isso não foi implementada a tempo.

No mais, seria interessante testar o depurador com outras placas e arquiteturas suportadas pela *libasm*. Isso poderia apontar novos caminhos e funcionalidades necessárias, ou problemas que não foram encontrados na versão atual da ferramenta. Executar testes com produtos reais, disponíveis comercialmente para o consumidor final, serviria como estudo de caso de uma sessão real de engenharia reversa e colocaria à prova a verdadeira capacidade de análise do *KEDBG-JTAG*.

REFERÊNCIAS

[1] "GDB and Reversible Debugging" http://sourceware.org/gdb/news/reversible.html

[2] Vanegue J., "How to Port ERESI". http://www.eresi-project.org/browser/trunk/doc/HOWTO-port-ERESI-0.8.txt

[3] Barr, Michael. "Embedded Systems Glossary." http://www.netrino.com/Embedded-Systems/Glossary

[4] Vanegue J., Garnier T., Auto J., Roy S. & Lesniak R., "Next-Generation Debuggers for Reverse Engineering", 2007

[5] Bisolfati, E. "Kedbg: the ERESI kernel debugger", 2008

[6] The ERESI project. http://www.eresi-project.org

[7] Auto J., "Developing an Intermediate Representation for the Analysis of Binary Code", 2007

[8] The LPC2124 Board Specification. http://www.olimex.com/dev/lpc-e2124.html

[9] The LPC2124 Datasheet. http://www.olimex.com/dev/images/lpc-e2124-sch.gif

[10] The ARM-JTAG Macgraigor Wiggler Compatible cable. http://www.olimex.com/dev/arm-jtag.html

[11] The JTAG F.A.Q. http://hri.sourceforge.net/tools/jtag_faq_org.html

[12] ERESI's manpage. http://www.eresi-project.org/wiki/EresiManPage

[13] ERESI's coding style. http://www.eresi-project.org/browser/trunk/doc/ERESI-coding_style-README.txt

[14] UrJTAG's documentation. http://urjtag.org/book/index.html

[15] OpenOCD's documentation. http://openocd.berlios.de/web/?page_id=54

[16] OpenOCD Diploma Thesis. http://developer.berlios.de/docman/display_doc.php?docid=1367&group_id=4148

[17] OpenOCD JTAG Debugger. http://www.amontec.com/openocd.shtml

[18] JTAG Introduction. http://www.inaccessnetworks.com/projects/ianjtag/jtag-intro/jtag-intro.html

[19] Boundary-Scan Information site. http://www.boundary-scan.co.uk/page3.html

[20] JTAG Technologies. http://www.jtag.com/

[21] JTAG – A Technical Guide. http://www.xjtag.com/support-jtag/jtag-technical-guide.php

[22] Boundary Scan Description Language Tutorial. http://www.corelis.com/products/BSDL_Tutorial.htm

[23] GDB Remote Serial Protocol. http://sources.redhat.com/gdb/onlinedocs/gdb_33.html

[24] ARM7TDMI Specification. http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf

[25] Linux Device Drivers, chapters 4, 9 and 10.

[26] How to Break Code, chapter 3.

[27] Revision 1461 on OpenOCD projects. http://svn.berlios.de/wsvn/openocd/?rev=1461&sc=1

[28] Simple *JTAG* interface circuit. http://jtag-arm9.sourceforge.net/circuit.txt

[29] GDBICE. http://sourceforge.net/projects/gdbice/

[30] JTAGER. http://jtager.sourceforge.net/

[31] JTAGPACK. http://jtagpack.sourceforge.net/

[32] URJTAG v0.10 Documentation. http://urjtag.svn.sourceforge.net/viewvc/urjtag/tags/URJTAG_0_10/web/htdocs/book/_system_requirements.html - _supported_host_operating_systems

[33] Código fonte do KEDBG-JTAG. http://www.eresi-project.org/browser/branches/kedbg-jtag-dev

[34] ARM Architecture Reference Manual

[35] ARM Registers. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/ch05s02s05.html

[36] ARM Operating Modes. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0084f/I2029.html

[37] ARM States. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0040d/CACCIDAH.html

[38] Embedded Web Server Demo. http://www.freertos.org/index.html?http://www.freertos.org/portrowleylpc2124.html