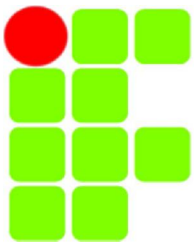

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL DE MINAS GERAIS
Câmpus Inconfidentes

CLAITON APARECIDO ARRUDA

ANÁLISE DE FERRAMENTAS QUE USAM COMUNICAÇÃO EM REDE NO AUXILIO DE DESENVOLVIMENTO E DEPURAÇÃO DE PROGRAMAS PARA SISTEMAS EMBARCADOS

INCONFIDENTES-MG

2013

CLAITON APARECIDO ARRUDA

ANÁLISE DE FERRAMENTAS QUE USAM COMUNICAÇÃO EM REDE NO AUXILIO DE DESENVOLVIMENTO E DEPURAÇÃO DE PROGRAMAS PARA SISTEMAS EMBARCADOS

Trabalho de conclusão de curso apresentado ao curso de Tecnologia em redes de computadores – Instituto Federal Sul de Minas – Campus Inconfidentes, como requisito parcial de aprovação.

Orientador: Prof. Luiz Carlos Branquinho Caixeta Ferreira

INCONFIDENTES

2013

CLAITON APARECIDO ARRUDA

ANÁLISE DE FERRAMENTAS QUE USAM COMUNICAÇÃO EM REDE NO AUXILIO DE DESENVOLVIMENTO E DEPURAÇÃO DE PROGRAMAS PARA SISTEMAS EMBARCADOS

Data de aprovação: ___ de _____20___

_____
Orientador: Prof. Luiz Carlos Branquinho Caixeta Ferreira
IFSULDEMINAS - Câmpus Inconfidentes

_____
Prof. André Luigi Amaral di Salvo
IFSULDEMINAS - Câmpus Inconfidentes

_____
Prof. Leonardo Martins Alves
IFSULDEMINAS - Câmpus Inconfidentes

RESUMO

Este trabalho tem como objetivo analisar o uso de ferramentas que ajudam no desenvolvimento, teste e depuração de programas para sistemas embarcados utilizando a comunicação em rede. Para isso foi feito um estudo sobre os sistemas embarcados a fim de conhecer suas características, componentes e ferramentas utilizadas no desenvolvimento e depuração para sistemas embarcados bem como técnicas para se testar um *software* e erros mais comuns no desenvolvimento de programas. Ao decorrer do estudo foi visto que uma ferramenta intuitiva para comunicação entre o sistema usado nos testes o *PC* não existia e por isso houve o trabalho em cima do desenvolvimento de uma aplicação em *python* que suprisse esta falta de tal ferramenta. Porém, os testes foram focados em cima das ferramentas existentes e disponíveis mostrando o funcionamento destas e comentando sobre as vantagens em comparação com a não utilização das mesmas.

ABSTRACT

This paper aims to analyze the use of tools that help in the development, testing and debugging programs for embedded systems using network communication. For this was a study done on embedded systems in order to know its features, components and tools used in the development and debugging for embedded systems as well as techniques to test a software and common errors in program development. In the course of the study it was seen that an intuitive system for communication between the PC used in the tests did not exist and so there was the job upon development of an application in python that met this lack of such a tool. However, tests have been focused upon the existing tools available and illustrating the operation and commenting on these advantages in comparison with no use of them.

SUMARIO

LISTA DE FIGURAS

LISTA DE SIGLAS

ARM.....dvanced RISC Machine
ASIC.....Application Specific Integrated Circuit
CDT.....C/C++ Development Tooling
CPU.....Central Processing Unit
DDD.....Data Display Debugger
DMA.....Direct Memory Access
EEPROM.....Electrically-Erasable Programmable Read-Only Memory
ELF.....Executable and Linking Format
EPROM.....Erasable Programmable Read-Only Memory
FPGA.....Field-Programmable Gate Array
GCC.....GNU Compiler Collection
GDB.....GNU Debugger
GNU.....General Public License
HD.....Hard Disk
ICSP.....In-Circuit Serial Programming
IDE.....Integrated Development Environment
IOP.....Input Output Processor
JTAG.....Joint Test Action Group
KHZ.....Kilohertz
MHZ.....Megahertz
MIPS.....Microprocessor Without Interlocked Pipeline Stages
PC.....Personal Computer
RAM.....Random Access Memory
RISC.....Reduced Instruction Set Computer
ROM.....Read-only memory
SIMD.....Single Instruction, Multiple Data
TPC/IP.....Transmission Control Protocol/ Internet Protocol
UDP.....User Datagram Protocol
USB.....Universal Serial Bus
VGA.....Video Graphics Array

1. INTRODUÇÃO

O trabalho de desenvolver, manter sempre atualizado e sem *bugs* um *software* não é fácil e quando se trata de um sistema embarcado o trabalho é dobrado já que não só o conhecimento da linguagem e bibliotecas são importantes, mas do sistema como um todo.

Assim, se faz necessário o uso de ferramentas para ajudar a resolver, ou pelo menos minimizar, os problemas encontrados durante as etapas de desenvolvimento, manutenção e atualização destes programas.

Porém, estes dispositivos dependem de um *hardware* especifico para manipular dados e programas que estão em sua memória utilizando para isto uma linha de comunicação serial.

Em contrapartida estes mesmos dispositivos já vem equipados com uma forma de comunicação em rede, seja ela sem fio ou cabeada, fazendo com que dispositivos tão diferentes possuam algo em comum.

Usar ferramentas que fazem o uso de comunicação em rede para o trabalho de depurar e desenvolver aplicativos para os sistemas embarcados pode ser uma opção a fim de ganhar agilidade e mobilidade para estas tarefas.

Com o intuito de analisar o uso destas ferramentas, neste documento foram avaliados alguns casos de teste numa plataforma real e comparado como seria sem o uso destas.

1.1. MOTIVAÇÃO

Facilitar o trabalho de desenvolvimento para os sistemas embarcados, em especial para a plataforma de teste escolhida, já que isso propiciará, num futuro, trabalhar de forma mais rápida e segura no desenvolvimento de um projeto para esse sistema que apesar de ser um sistema mais antigo possui um *hardware* muito bom o que possibilita muitos trabalhos na área científica e acadêmica assim como clusters e simuladores de forma de vida, como podemos ver em [19] [20].

1.2. OBJETIVOS GERAIS

Compreender o uso de ferramentas e técnicas que auxiliem no desenvolvimento, teste e depuração de programas para sistemas embarcados utilizando para isso a comunicação de rede presente nestes dispositivos.

1.3. OBJETIVOS ESPECÍFICOS

- Pesquisar ferramentas utilizadas para desenvolvimento e depuração para sistemas embarcados.
- Analisar o processo de inicialização dos programas em sistemas embarcados.
- Compreender os métodos de comunicação utilizados pelas ferramentas de desenvolvimento e depuração para comunicação em rede com os programas executados no sistema embarcado.
- Estudar e aplicar as técnicas utilizadas para testar *software* a fim de encontrar falhas.
- Demonstrar a utilização dessas ferramentas e técnicas em um programa real.
- Desenvolver uma aplicação que seja intuitiva a fim de ajudar na utilização do ps2client que funciona somente em linha de comando.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. SISTEMAS EMBARCADOS

Quando falamos em microprocessador o que nos vem à cabeça são *desktops* ou *laptops* rodando aplicativos, mas eles não são os únicos a ter um em seu interior. São estes objetos de uso cotidiano na vida das pessoas, objetos tais como micro-ondas, TVs, aparelhos de som, câmeras fotográficas digitais entre muitos outros equipamentos. Sem estes microprocessadores eles não seriam tão sofisticados o quanto são. A ideia de incorporar microprocessadores nesses equipamentos veio antes dos *PCs* e *laptops* surgirem. [11]

Mas o que estes têm a ver com sistemas embarcados? Existem inúmeras definições para estes dispositivos, mas de forma geral o sistema embarcado é aquele dispositivo baseado em um microprocessador que tem funções especificas tais como os citados acima. Nesses, usuários finais podem fazer configurações de como o sistema deve se comportar, mas não se pode fazer qualquer tipo de alteração de funcionalidade no sistema como se pode fazer nos *PCs*. [11]

2.1.1. Estrutura dos sistemas embarcados

Mesmo com funções bem distintas para cada sistema, existe algo que é um padrão para todos eles, sua estrutura. Esta consiste em:

- **Microprocessador**: É o coração de todo sistema embarcado, sua velocidade de operação pode variar de alguns poucos KHZ até centenas de MHZ dependendo das funções a qual ele é empregado. Por conta desses dispositivos, na maioria das vezes, ter de funcionar com uma bateria, eles devem ter um consumo muito baixo para que estes funcionem por um maior período de tempo, para este fim a arquitetura RISC (*Reduced Instruction Set Computer* - Computador com um Conjunto Reduzido de Instruções) são os empregados, além de terem implementados em si a capacidade de reduzirem seu *clock* e até entrar em modo de hibernação com isso reduzindo ainda mais o consumo de energia. Dentre os processadores mais utilizados em sistemas embarcados os MIPS (Microprocessor without Interlocked Pipeline Stages - microprocessador sem estágios interligados de pipeline) e os ARM (primeiramente *Acorn RISC Machine*, posteriormente *Advanced RISC Machine*) se destacam como podemos ver na figura 1. [1]

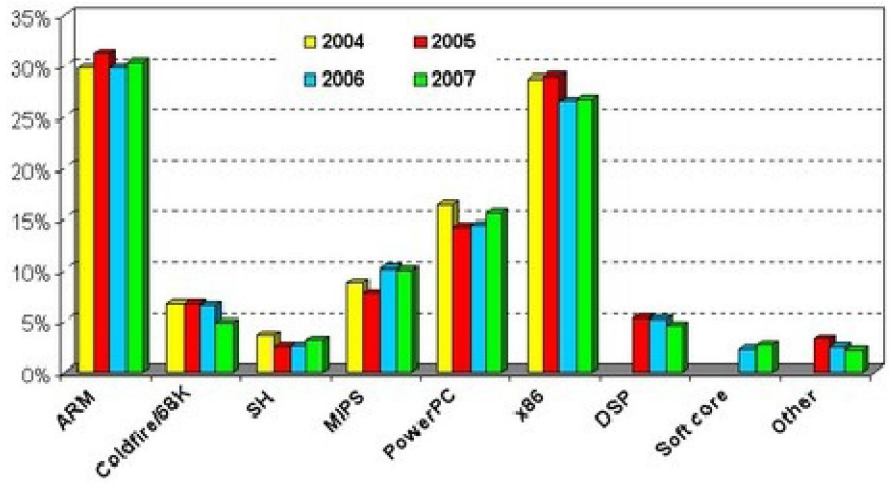

Figura 1. Tendência de uso dos microprocessadores.

Outros componentes largamente empregado na construção dos dispositivos embarcados são os microcontroladores e FPGAs.

> *"Microcontrolador é um circuito integrado programável que contém todos os componentes de um computador como CPU (unidade central de processamento), memória para armazenar programas, memória de trabalho, portas de entrada e saídas.*
>
> *para comunicar-se com o mundo exterior, sistema de controle de tempo interno e externo, conversores analógico digital, uart de comunicação e outros.*
>
> *Pode-se controlar qualquer coisa ou estar incluído em unidades de*

controle para:

- máquinas pneumáticas, hidráulicas comandadas

- máquinas dispensadoras de produtos

- motores, temporizadores

-sistemas autônomos de controle, incêndio, umidade temperatura - telefonia, automóveis, medicina,... etc." [2]

Este pode ser definido como um computador em um chip, em um único encapsulamento ele traz processador, memória de programa e de trabalho ficando a cargo de quem o utiliza só prover o aplicativo e os periféricos que são colocados em suas portas. [2]

A grande vantagem deste é ter tudo em um único chip e não precisar de muitos componentes para ter seu projeto funcionando.

Dentre os mais utilizados estão os PICs da Microchip os LCP da NXP e os Atmel. Na figura 2 é mostrado um diagrama de blocos de um microcontrolador onde se pode observar os componentes que estão encapsulados.

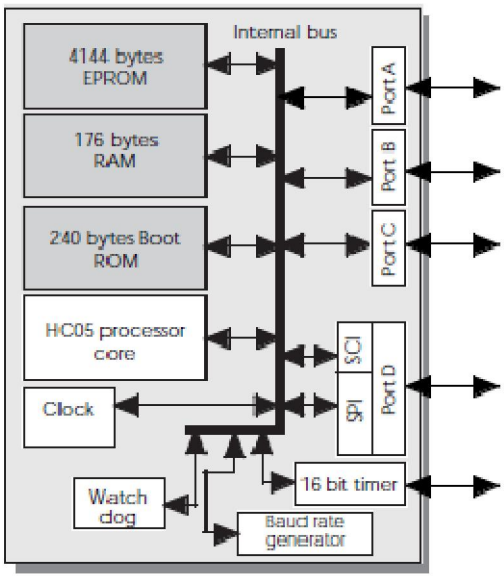

Figura 2. Exemplo de microcontrolador (Motorola MC68HC705C4A).

"FPGAs representam uma classe de dispositivos que possui a capacidade de implementar diferentes funções lógicas em hardware, permitindo a obtenção de desempenho similar ou próximo ao de ASICs, ao possibilitar a implementação de sistemas com um alto grau de paralelismo. Também oferecem a flexibilidade de sistemas programáveis, podendo ter seu comportamento configurado de maneira fácil e rápida" [3]

Destes os mais utilizados são os da XILINX e ALTERA.

Ter uma intimidade com esses componentes e *Assembly* facilita a tarefa para criar ferramentas que tirem proveito da característica de cada um dos mesmos.

- **Memórias**: Em se tratando de sistemas embarcados existem dois tipos, distintos, de memória. Uma memória não volátil onde o firmware (SO) do dispositivo é gravado, podendo esta ser uma EPROM (*Erasable Programmable Read-Only Memory* - memória programável apagável somente de leitura), estas podendo ser apagadas através de luz ultravioleta; EEPROM (*Electrically-Erasable Programmable Read-Only Memory* - memória programável eletricamente apagável somente de leitura); ou FLASH que nada mais é do que um tipo de EEPROM que funciona como uma RAM o que possibilita acesso a endereços múltiplos. Uma memória volátil (RAM) que os aplicativos utilizam para guardar os dados temporários que vão utilizar.[11]

- **Periféricos**: São os dispositivos que nos faz ter uma interação com o dispositivo ou pelo qual ele exerce sua função, tais como teclados, luzes, leitores de cartões, interfaces *ethernet*. [11]

2.2. SISTEMA DE TESTES

Para os objetivos foi necessário uma plataforma de testes onde foi aplicada a teoria descrita neste documento. Existem vários kits no mercado pra desenvolvimento para os mais variados microprocessadores citados acima, onde alguns exemplos podem ser vistos nas figuras 3 e 4. Porém para este estudo, foi utilizado um *Playstation* 2, já que entre outras vantagens ele possui um *hardware* já preparado para os testes por já possuir integrado periféricos tais como conector para *ethernet* para comunicação em rede e entradas USB, uma grande quantidade de documentação disponível na internet e as ferramentas base para os testes existem necessitando apenas de melhorias e adaptações.

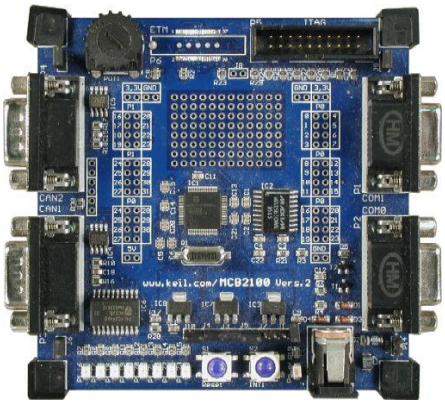

Figura 3. Kit Keil MCB2100 para microcontroladores ARM.

Figura 4. Kit para Microchip PIC18f.

2.2.1. Características do Playstation 2

- Possui como componente central um microprocessador com arquitetura RISC, é um MIPS R5900, desenvolvido pela Toshiba exclusivamente para o Playstation 2, muito parecido com o R5000 mas com a inclusão de algumas instruções multimídia SIMD (*Simple Instruction Multiple Data* – Instrução Simples Múltiplos Dados) de 128 bits que fazem uso total dos registradores que também são de 128 bits. Ele trabalha numa frequência de 300MHz. Na figura 5 é mostrado um diagrama de blocos do coração do Playstation 2;
- Tem 32MB de memória de trabalho; [13]
- Leitor de DVD de 4X o que significa uma taxa de leitura de 5400KB/s;
- Duas entradas USB 1.1; [13]
- Leitor de cartões de memória de formato proprietário de 8MB e 16MB oficiais além de alguns modelos paralelos de até 128MB; [13]
- E o principal, conexão *ethernet* 10/100, fundamental para os experimentos, que pode atingir até 12500KB/s o que em teoria seria mais que o dobro de velocidade que pode ser oferecido pelo leitor de DVD viabilizando o trafego de dados de informação de debug sem comprometer a transferência de dados do aplicativo em execução. [13]
- Para controlar os periféricos ele possui um coprocessador, um MIPS R3000 trabalhando a uma frequência de 37MHz que conta com uma memória própria para programas de 2MB, um circuito DMA exclusivo para comunicação com os periféricos. Quando o processador central precisa obter informações dos estados dos botões, leitor de cd, *memorycard* é este quem entrega ao mesmo.

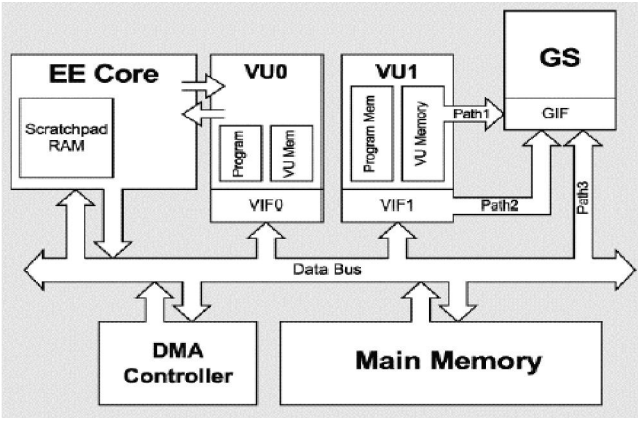

Figura 5. Diagrama de blocos do EE componente central do Playstation 2.

2.3. FERRAMENTAS PARA DESENVOLVIMENTO

Diversos métodos são utilizados para programar esses dispositivos desde programadores caros e específicos para determinado kit a soluções e esquemas gratuitos disponíveis na internet. Através de uma comunicação serial entre o PC e o microcontrolador o programa e copiado para a ROM aonde ele será executado assim que o sistema for iniciado. Na figura 6 é mostrado um diagrama de montagem de um gravador para PIC baseado no JDM para o PIC16F84, circuito muito utilizado pelos desenvolvedores hobistas. Através de uma montagem dos pinos na própria placa também é possível de fazer uma gravação ICSP (*In-Circuit Serial Programing*) que nada mais é do que a programação do circuito montado na placa sem a necessidade de removê-lo. [2]

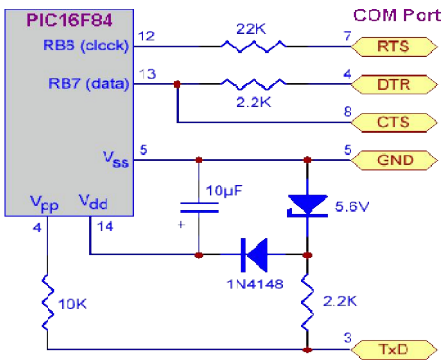

Figura 6. Diagrama de montagem do circuito gravador de PIC.

Para alguns modelos de microcontroladores a gravação é feita através do JTAG (Joint Test Access Group), esta se trata de uma interface de programação e testes de circuitos digitais que é padronizada pela IEEE 1,149,1. Desenvolvida para programadores lógico e

frequentemente utilizado em microcontroladores e FPGAs. A vantagem deste método é que além da programação no próprio circuito ele tem as funcionalidades de debug para o programa que está sendo gravado na memória. Na figura 7 pode se ver um exemplo de um gravador/depurador JTAG para microcontroladores ARM.

Figura 7. Gravador/depurador JTAG para microcontroladores ARM.

Além do gravador, se faz necessário algumas ferramentas para digitação do código fonte do programa e o programa que vai transformar o código fonte em um arquivo binário aceito pra ser transferido para a memória.

A duas principais linguagens de programação disponíveis para estes dispositivos são o *assembly* e o C/C++. A primeira gera código otimizado em tamanho, o que ajuda já que a quantidade de memória disponível para estes dispositivos acabam sendo bem limitadas. Cada um destes possui seu próprio conjunto de ferramentas que pode ser adquirido junto ao site do fornecedor do produto, seja ela a Microchip, Keil ou LPC.

Para o *Playstation* 2 existem 3 kits disponíveis para usar no desenvolvimento de aplicativos, podendo estes ser adquirido facilmente numa busca na internet.

- Sony PS2TOOL – Este é o kit de desenvolvimento distribuído para as empresas licenciadas pela Sony mediante a compra do mesmo por mais de

15 mil dólares. Hoje é facilmente encontrado na internet para *download*, porem a utilização do mesmo em projetos *homebrew* é inviável já que ele precisa do *hardware* também utilizado pelas empresas. Com ele se tem total acesso aos componentes do *Playstation* 2 possuindo todas as bibliotecas e aplicativos necessários para o desenvolvimento de aplicativos. Este pode ser visto na figura 8. [13]

- PS2Linux Kit– A Sony viu o interesse de muita gente em desenvolver aplicativos para seu *videogame* e lançou em 2002 esse kit que consistia em dois DVDs contendo uma versão do Linux e aplicativos, mouse, teclado, um adaptador VGA, adaptador de rede, manual de instrução e um HD de 40GB. Com este era possível criar e executar aplicativos que rodariam somente dentro do próprio Linux e oferecia apenas algumas bibliotecas de acesso aos recursos do *videogame*. Era bem limitado para quem gostaria de tirar proveito de toda a capacidade que possuía o sistema. Visto na figura 9. [13]
- PS2SDK - É um conjunto de ferramentas criadas por terceiros com base no GNU GCC distribuído de forma gratuita através do código fonte. Ao longo dos anos o projeto foi crescendo e ganhando com isso mais bibliotecas e ferramentas que hoje possibilitam acesso total ao *hardware* do playstation 2. O que faz desta a melhor escolha quando se trata de desenvolvimento *homebrew* para o console. [13]

Diferente dos outros, para se trabalhar com o PS2SDK, os únicos pré-requisitos são ter as ferramentas devidamente compiladas e um método de colocar o programa criado com estas ferramentas para ser executado no sistema, métodos estes citados no capitulo 2.4.

Figura 8. Sony PS2TOOL.

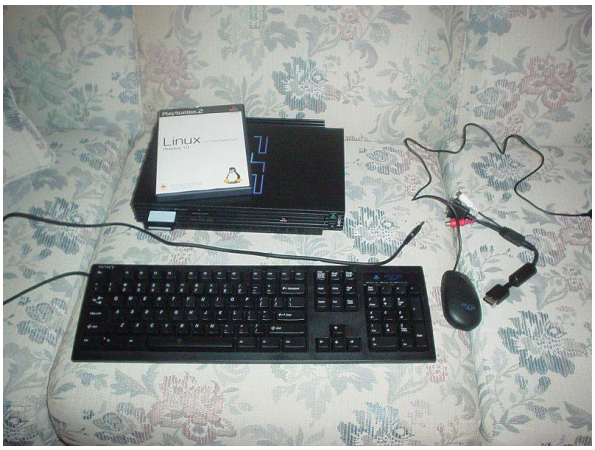

Figura 9. Sony PS2Linux Kit.

2.4. *BOOTLOADER*

Nada mais é do que um *software* que é executado toda vez que o sistema é iniciado, podendo este estar em uma EEPROM ou FLASH, separada ou junto onde também está armazenado o programa principal.

Geralmente este é construído em linguagem de mais baixo nível, *assembly*, já que este depende muito de arquitetura de processador utilizada e dá baixa quantidade de memória disponível para ele.

Assim que o sistema se inicia este é o primeiro programa a ser executado e ele carrega para a memória o programa principal passando nesse exato momento a executar a partir do inicio deste, é este também o responsável por fazer a transferência do código do programa principal que está no computador para a memória própria no sistema embarcado.

Em se tratando dos PICs, em especial da família 16F e 18F, e ARMs que dão suporte a auto programação pode ser usado um protocolo serial ou mesmo a comunicação ethernet disponível em alguns destes dispositivos para transferir o arquivo para a memória que não foi alocada para o *bootloader* e a partir deste rodar o programa carregado. [21][22][23]

No caso do *Playstation* 2 ele já possui um *bootloader*, um *firmware* que vai buscar o programa principal direto no DVD, porem existem alguns modos de burlar esse *bootloader* e usar o seu próprio. Um dos modos mais simples é ter o *videogame* desbloqueado e gravar no DVD o programa que se quer executar assim como se fosse um jogo normal, esse programa seria o nosso *bootloader* e através deste poderíamos fazer o resto da comunicação utilizando qualquer outro meio disponível nele seja *memorycard*, *pendrive*, o próprio leitor de DVD ou comunicação *ethernet*, foco dos testes.

Outros métodos de se executar um *bootloader* no PS2 é através de um *exploit* de *memorycard*, que se aproveita de algumas vulnerabilidades existente no sistema de salvamento de dados, disponível somente para algumas versões do aparelho, e através de algum chip de destravamento que possibilite o uso das portas USB. No documento foi utilizado um aparelho desbloqueado com o chip *ThunderPro II* que possibilita que um programa compilado para o *Playstation* 2 rode diretamente de uma pasta especifica dentro do *pendrive*, a pasta *boot*, esta contendo o arquivo renomeado como "boot.elf".

2.5. TESTE DE SOFTWARE

Como a parte do *hardware* do sistema embarcado e do que é necessário para desenvolver um software é descrita no capitulo anterior, neste se fala de como testar estes *softwares*, citando as técnicas aplicadas.

O teste de *software* é o processo pelo qual deve determinar se as expectativas para ele foram atingidas cumprindo com as especificações e funcionando corretamente no ambiente para o qual foi criado. O objetivo do teste de *software* é revelar falhas no mesmo para que as causas possam ser tratadas. [4]

De uma forma simples, testar um *software*, significa verificar através de uma execução controlada se o comportamento ocorre de acordo com o especificado resolvendo o número máximo de falhas com o mínimo esforço. [4]

Alguns conceitos são necessários antes de começar a falar dos testes de *software* propriamente ditos. Primeiro precisamos conhecer as diferenças entre Defeitos, Erros e Falhas. [4]

- Defeito é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto. [4]
- Erro é uma manifestação concreta de um defeito num artefato de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro. [4]
- Falha é o comportamento operacional do *software* diferente do esperado por um usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha. [4] [5]

A figura 10 expressa a diferença entre esses conceitos. Onde defeitos fazem parte da própria aplicação e são causadas pelas pessoas que a criam. Defeitos podem vir a fazer aparecer erros e este gera falhas que são comportamentos inesperados em um *software* e afetam diretamente o usuário.

Figura 10. Defeito x erro x falha (http://www.projectcartoon.com/cartoon/611).

2.5.1. Conceitos básicos sobre teste de *software*

Testes de *software* somente revelam falhas em um produto. Após estes testes se faz necessário um processo de depuração para identificar e corrigir os defeitos que deram origem a essas falhas. O *debugging* é utilizado para encontrar e resolver os defeitos e este é discutido no capítulo 2.6.

Testar um *software* envolve alguns elementos que formalizam essa atividade. São eles:

- Caso de teste - Condição a ser testada é composta por valores de entrada, restrições para sua execução e um resultado esperado. [4]
- Procedimento de teste - Descrição dos passos necessários para executar um caso de teste. [4]
- Critério de teste - Usado para selecionar e avaliar os casos de teste de forma que as falhas apareçam. Estes podem ser utilizados como:
    - Critério de cobertura dos testes - permite a identificação de partes do programa que devem ser executadas para garantir a qualidade do *software* e indicar quando o mesmo foi suficientemente testado. Ou seja, determinar o percentual de elementos necessários por um critério de teste que foram executados pelo conjunto de casos de teste. [4]
    - Critério de Adequação de Casos de Teste - Quando, a partir de um conjunto de casos de teste T qualquer, ele é utilizado para verificar se T satisfaz os requisitos de teste estabelecidos pelo critério. Ou seja, este critério avalia se os

casos de teste definidos são suficientes ou não para avaliação de um produto ou uma função. [4]

- Critério de Geração de Casos de Teste - quando o critério é utilizado para gerar um conjunto de casos de teste T adequado para um produto ou função, ou seja, este critério define as regras e diretrizes para geração dos casos de teste de um produto que esteja de acordo com o critério de adequação definido anteriormente. [4]

2.5.2. Níveis de teste de *Software*

Com os conceitos apresentados é hora de apresentar os diferentes níveis de teste de software:

- Teste de Unidade - também conhecido como testes unitários. Tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código. [4]
- Teste de Integração - visa procurar falhas associadas às interfaces entre os módulos quando esses são interligados para construir a estrutura do *software* que foi estabelecida na fase de projeto. [4]
- Teste de Sistema - avalia o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação de *software*. Verifica se o *software* satisfaz seus requisitos. [4]

2.5.3. Técnicas de teste de software

Existem muitas maneiras de se testar um *software* porem, ainda hoje, são utilizados métodos empregados a muito tempo em linguagens estruturadas nas linguagens orientadas a objetos, já que o foco destes testes acaba sendo o mesmo, encontrar falhas em *softwares*. Sendo assim, continuam sendo de grande valia para os testes. [4]

Estas técnicas são classificadas de acordo com a origem das informações utilizadas e contemplam diferentes perspectivas do *software* tendo que estabelecer uma estratégia a fim de

contemplar as vantagens e aspectos complementares de cada técnica. As técnicas existentes são a funcional e estrutural.

- Técnica Estrutural: O teste de caixa branca ou teste de caixa de vidro, como também é conhecido, é aplicado em cima do código fonte do programa e analisa a estrutura do mesmo tal como saltos, loops e estruturas condicionais. Este garante uma verificação mais precisa do seu comportamento já que permite a concentração da atenção com precisão nos pontos considerados mais importantes do código fonte da aplicação. Então são elaborado casos de teste que cubram todas as possibilidades do componente, e desse modo, todas as variações originadas por estruturas de condições são testadas. [4][14][15]
- Técnica Funcional: O teste de caixa preta ou comportamental diferente do teste de caixa branca é realizado ao final dos testes dando atenção aos requisitos. O que é testado é abordado como uma caixa preta e sua parte interna, códigos e estruturas, são irrelevantes, este avalia somente os requisitos totalmente ou parcialmente satisfeitos pelo *software*.

  Enfatiza as entradas, saídas e princípios funcionais dos módulos para elaborar e realizar testes, desta forma verificando as funcionalidades do *software* a partir da apresentação de suas entradas aos componentes ou *software* e analise da saída gerada.

Os testes são feitos com os dados de entrada fornecidos e então a saída é comparada com o resultado esperado. Com este pode ser testado um método, uma função interna, um ou um conjunto de componentes sistemas. [15]

Este tipo de teste não é uma alternativa ao teste de caixa branca, mas sim um teste complementar ao mesmo que tem a capacidade de detectar uma classe de erros não encontrada pelo teste de caixa branca. [15]

Os erros que o teste procura detectar se encontram nas seguintes categorias:

- Funções incorretas ou ausentes;
- Erros de Interface;
- Erros de comportamento;
- Erros nas estruturas de dados ou acesso à base de dados externa;

- Erros de desempenho;
- Erros de inicialização e término.

Os testes deverão ser guiados pelos Casos de Teste e/ou Especificações dos Casos de Uso, porque é neles que estão as descrições do comportamento da aplicação, através deste dá para simular erros que usuários poderiam cometer ou que fogem da especificação.

Mesmo assim é impossível testar todas as possibilidades e afirmar o perfeito funcionamento do programa, mas com testes bem realizados terá bons indícios da qualidade do software. [15]

2.6. DEPURAÇÕES DE CÓDIGO

Depois de aplicadas as devidas técnicas de teste de *software*, vistas no capítulo anterior, e encontrados as falhas chega a hora da depuração do código a fim de resolver os defeitos que originaram as falhas encontradas. [4][5]

Esta consiste em, através de um depurador, monitorar a execução do programa e utilizando de suas funções tais como parando a execução, reiniciá-la, ativar pontos de parada, rodar linha a linha de código, alterar áreas de memória, analisar mudança das variáveis e do comportamento do código e com isso identificando e tratando o defeito. [4] [5]

Uma outra funcionalidade dos depuradores, que ajuda muito a identificar e corrigir os defeitos e a possibilidade de uma análise semântica do código. Com essa análise se pode verificar e apontar expressões no código fonte que quebram as regras determinadas. Utilizando se da verificação te tipo verifica se um determinado operando recebe outro de mesmo, o que retorna um erro caso estes sejam diferentes. [4] [5]

Alguns erros típicos são:

- Uma variável não declarada
- Multiplicação entre tipos diferentes
- Atribuição de um literal a outro tipo

2.6.1. Técnicas de depuração

- Algoritmo *Wolf Fence* (Cercar o Lobo): Descrito por E. J. Gauss [9] em um artigo publicado em 1982 é um simples e útil algoritmo descrito da seguinte forma: "Existe um lobo no Alaska, como você o encontra? Primeiro construa uma cerca no meio do estado, espere o lobo uivar, determine qual o lado da cerca ele está. Repita o processo deste lado somente, até que você chegue no ponto onde você pode ver o lobo." Em outras palavras; coloque algumas instruções dentro do código que imprimam na tela informações até que você encontre o lugar onde está a falha (trabalhando por "trás dos panos" para encontrar de onde o lobo/bug vem). [9]
- *Print*(ou *tracing*) *debugging*: é o ato de assistir (ao vivo ou gravadas) traçando declarações, ou funções de impressão, que indica o fluxo de execução de um processo. Este é algumas vezes chamado de *printf debugging*, devido ao uso da função *printf* em C, que serve para imprimir na tela. Este tipo de *debugging* foi habilitado pelo comando *TRON* na versão original da linguagem de programação *BASIC*. *TRON* é a representação de "*TRace ON*". Este fazia com que o numero das linhas executadas fossem exibidas durante a execução do programa.
- *Debugging* Remoto: é o processo de depurar um programa que está rodando em um sistema diferente de onde o *debugger* está. Para iniciar a depuração remota, o *debugger* conecta ao sistema remoto através da rede. Uma vez conectado, o *debugger* pode controlar a execução do programa no sistema remoto e recuperar informações de seu estado.
- Post-mortem *debugging*: é a depuração do programa depois que ele parou de funcionar. Esta técnica inclui várias técnicas de "*tracing*", por exemplo a analise do *memory dump* do processo que parou de funcionar. O *dump* pode ser obtido automaticamente pelo sistema ou por uma instrução inserida pelo programador, ou manual pelo uso interativo.
- *Delta Debugging*: técnica que automatiza o teste de caso de simplificação.
- *Saff Squeeze*: técnica que isola a falha com o uso progressivo das partes *inline* do teste de falha.

2.6.2. Depuração para sistemas embarcados

Em contraste com o design de software para computadores de propósito geral, a principal característica do sistema embarcado é simplesmente o numero de plataformas diferentes existentes para o desenvolvedor (arquitetura de CPU, vendedores, sistemas operacionais e seus variantes).

Sistemas embarcados são, por definição, não designados para propósitos gerais: eles são tipicamente desenvolvidos para uma simples tarefa (ou uma pequena quantidade de tarefas), e a plataforma é escolhida especialmente para otimizar a aplicação. Não é somente este fato que faz difícil a vida de um desenvolvedor para sistemas embarcados, o *debugg* e o teste desses sistemas fica tão difícil quanto, desde que diferentes ferramentas de *debugging* são necessárias para plataformas diferentes. Abaixo estão algumas tarefas para as quais um *debugger* é indispensável:

- Para identificar e corrigir *bugs* no sistema (lógicos, sincronização de código ou erro de design no *hardware*)
- Para coletar informações sobre o estado operacional do sistema que podem então ser usado para analisar o sistema: para encontrar modos de aumentar a performance ou para otimizar outras importantes características (com sumo de energia, confiabilidade e resposta em tempo real).

2.6.3. GDB

Escrito por Richard Stallman em 1996 como parte do sistema GNU (General Public License), após seu GNU Emacs atingir estabilidade.

> *"Depois do GNU Emacs ficar razoavelmente estável, que demorou ao todo cerca de um ano e meio, eu comecei a rever as outras partes do sistema. Eu desenvolvi um depurador que eu chamei GDB que depura os símbolos do código C, que recentemente entrou em distribuição. Agora este depurador é em grande parte no espírito do Dbx, que é o depurador que acompanha o Berkeley Unix."*
>
> *—Richard Stallman*

Este é um *software* livre licenciado pela GNU, mantida de 1990 a 1993 por John Gilmore e agora pelo comitê GDB. [7]

Suporta uma grande variedade de linguagem de programação como C, C++, Fortran, Objective-C, Free Pascal, Java e outras parcialmente. Suporta os seguintes processadores ARM, MIPS, x86, x86-64, PowerPC e SuperH entre muitos outros. Estes acima citados são comumente encontrados em sistemas embarcados. [7]

> *"O propósito de um debugger como o GDB é permitir ver o que está acontecendo "dentro" de outro programa enquanto este é executado ou o que o programa estava fazendo no momento que parou de funcionar.*
>
> *GDB pode fazer 4 tipos de coisa para ajudar a pegar os bugs no ato:*
>
> - *Iniciar seu programa, especificando qualquer coisa que pode afetar seu comportamento.*
> - *Fazer seu programa parar por uma determinada condição.*
> - *Examinar o que aconteceu, quando o seu programa parou.*
> - *Mudar coisas em seu programa, então você pode experimentar a correção de um efeito de um bug e ir aprender a respeito de outro.*
>
> *Se você está tentando depurar um programa que está rodando em uma máquina que não roda o GDB de modos normais, muitas vezes é útil usar a depuração remota. Por exemplo, você pode usar a depuração remota num kernel de um sistema aberto, ou num sistema pequeno que não tem um sistema operacional de propósito geral poderoso o bastante para rodar um debugger.*
>
> *Algumas configurações do GDB têm uma interface especial serial ou TCP/IP para fazer o trabalho com o alvo de debugging. Em adição, GDB vem com um protocolo genérico serial que pode ser usado para escrever os stubs remotos-- código que roda no sistema remoto para comunicar com o GDB."* [7]

2.6.3.1. Interfaces gráficas de usuário

O GDB é uma ferramenta que funciona em linha de comando não possuindo sua própria interface gráfica, entretanto existem algumas ferramentas gráficas que servem de interface entre o utilizador e a ferramenta em si.

DDD - O *Data Display Debugger* (Depurador com Apresentação de Dados) é um *front-end*, uma interface para interação com o GDB que funciona em linha de comando. Este permite a integração do programa que está sendo depurado com seu fonte e mostra o código de máquina gerado em *assembly*. Tem, entre outras características, a capacidade de exibir a memória utilizada pelo programa, ou seja, as variáveis utilizadas no programa bem como seu conteúdo e ponteiros com suas referências e estrutura de dados. [12] [10]

Na figura 11 podemos ver a janela principal do DDD rodando uma aplicação para depuração.

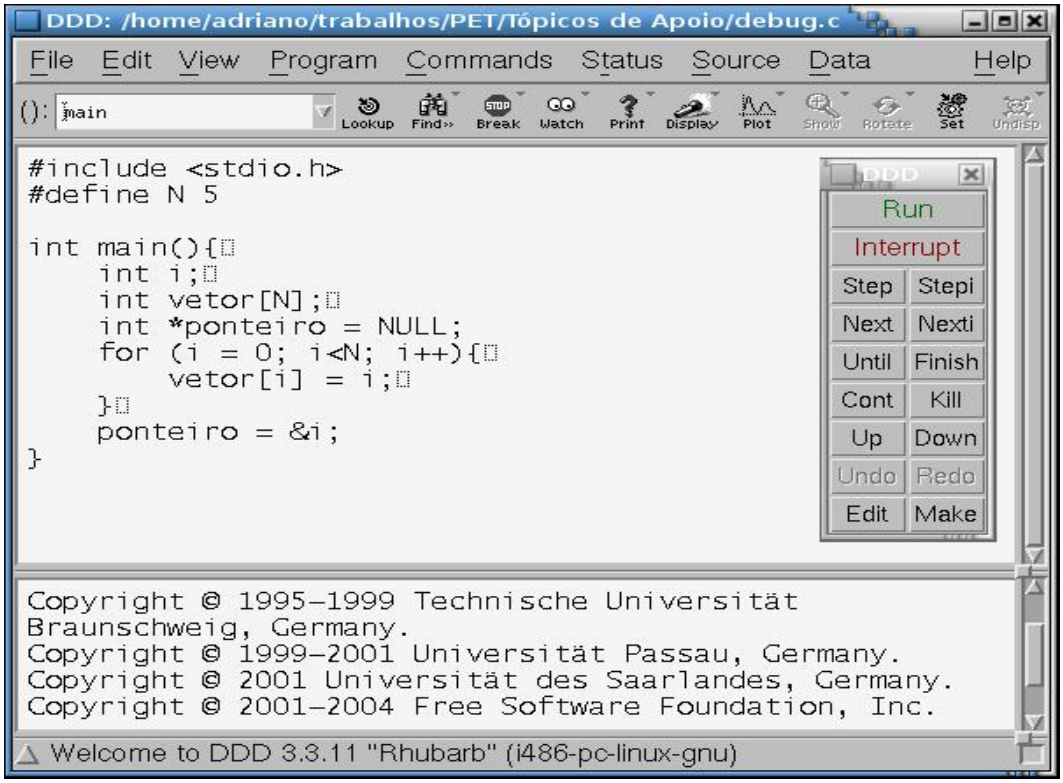

Figura 11. Janela principal do DDD.

GDBtk/Insight - É uma interface gráfica para o GDB, escrita em Tcl/Tk. Em sua primeira versão chamado de GDBtk e em sua segunda mudou para Isight, um sistema maduro que assim como o DDD é mais uma interface para o GDB. Na figura 12 temos o Insight sendo usado para depuração de um código. [16]

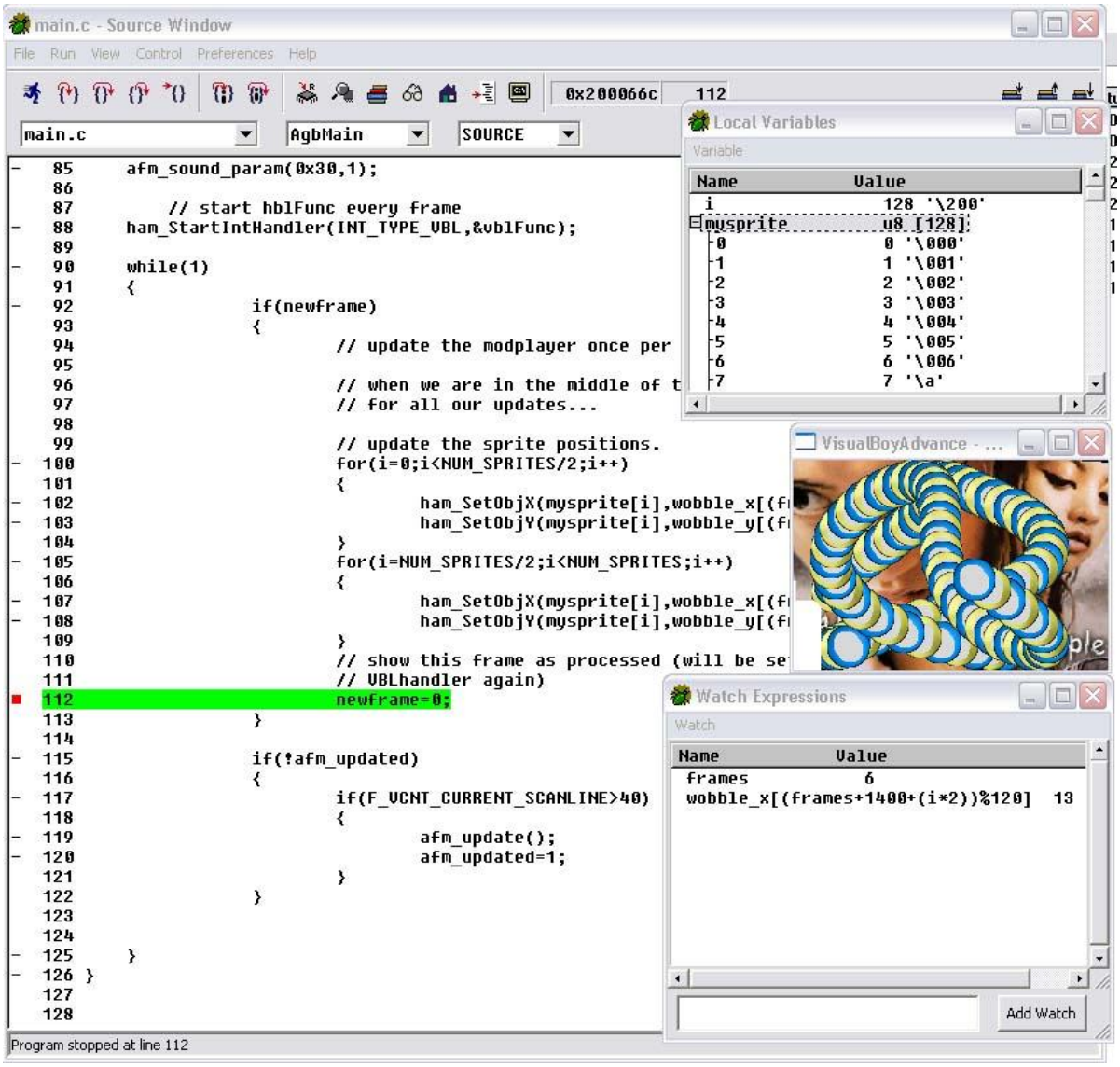

Figura 12. Insight sendo usado para depuração de código.

Eclipse com o *plugin* CDT - O *Eclipse C/C++ Development Tooling* é um dos mais bem conhecidos ambientes de desenvolvimento open source para C/C++ com função de *debugging* através de *plugins* CDI.[17]

Estes oferecem facilidades como as encontradas nas IDEs(*Integrated Development Environment* - Ambiente Integrado para Desenvolvimento) agilizando o trabalho de depuração oferecido pelo GDB.

2.6.3.2. Debug remoto

O principal aspecto que dificulta o trabalho de depurar um software para sistemas embarcados está no fato do mesmo não estar rodando na maquina onde se encontra o programa usado para depuração. Então se faz necessário algum tipo de comunicação entre o sistema para depuração (*target*) e o sistema onde se encontra a ferramenta de depuração (*host*) e é aí que entra o *debugging* remoto e o uso da ferramenta GDB.

Uma das grandes características do GDB é o de conseguir depurar programas que não estão rodando na máquina onde este está sendo executado, possuindo para isso uma interface de comunicação serial ou TCP/IP que possibilita a comunicação com o programa que se esta depurando na maquina alvo. [7] [18]

Para que o GDB se comunique com a aplicação alvo e essa tarefa de depuração seja executada é necessário que no sistema embarcado esteja rodando um pedaço de *software*, '*stub*', que implementa o protocolo de comunicação com o GDB.[7][18]

O *stub* é dependente da arquitetura do processador e deve ser criada para o sistema alvo em questão. Este se trata de uma biblioteca que deve ser compilada junto ao software que será depurado. [7] [18]

Na figura 13 é mostrado um diagrama de como é feita a comunicação entre o sistema hospedeiro e o alvo.

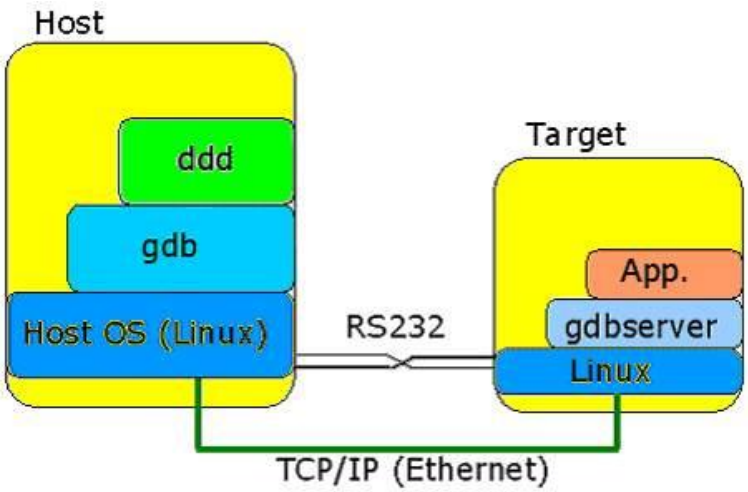

Figura 13. Iustração da comunicação entre as máquinas.

3. METODOLOGIA

Foi realizada uma pesquisa em busca de ferramentas que auxiliassem no trabalho de desenvolver e depurar aplicativos para sistemas embarcados que utilizassem comunicação em rede alem de técnicas de programação para auxiliar nos testes de *softwares* para utilizar com estas ferramentas.

Dentre as ferramentas encontradas para desenvolvimento para o *Playstation* 2, citadas no capitulo 2.3, foi utilizado o PS2SDK por ser o único *software* distribuído livremente. Com este é fornecido duas ferramentas, o Ps2Link e o Ps2Client usados para comunicação entre o sistema embarcado e o computador. Além disso, foi feito o *download* do GDB no site oficial e o *stub* para ser rodado no sistema embarcado em um site de um terceiro para usar na depuração dos programas criados.

O Ps2sdk, como foi dito no capítulo 2.3, se trata de um conjunto de ferramentas baseada no GNU binutils e GCC compilados para trabalhar com a arquitetura MIPS do *Playstation* 2. Além dos compiladores para C e C++ usados para criar os executáveis que são enviados para a plataforma, com este vem um conjunto de bibliotecas que dão acesso aos dispositivos que fazem parte desta é encontrado no site https://github.com/ps2dev/ps2toolchain.

Ps2link é o programa servidor e funciona como *bootloader* para no sistema *Playstation* 2. Este se divide em 2 partes: O aplicativo principal, que roda no processador central, e monitora o programa que é executado e um modulo, executado no IOP, que é um coprocessador utilizado para controlar os periféricos, e monitora a comunicação que é feita

através da rede, este responsável por aguardar um comando ser enviado para ele através do Ps2client e retorna para o Ps2client algum tipo de mensagem enviada pelo programa que está rodando no processador principal.

A configuração deste é feita através de um arquivo simples onde é colocado somente o seu ip na rede, a mascara e mais o *gateway* padrão, podendo também conter módulos adicionais que serão carregados para o IOP ao iniciar o programa.

Na figura 14 é apresentada a tela inicial do Ps2link esperando um comando ser enviado para ele.

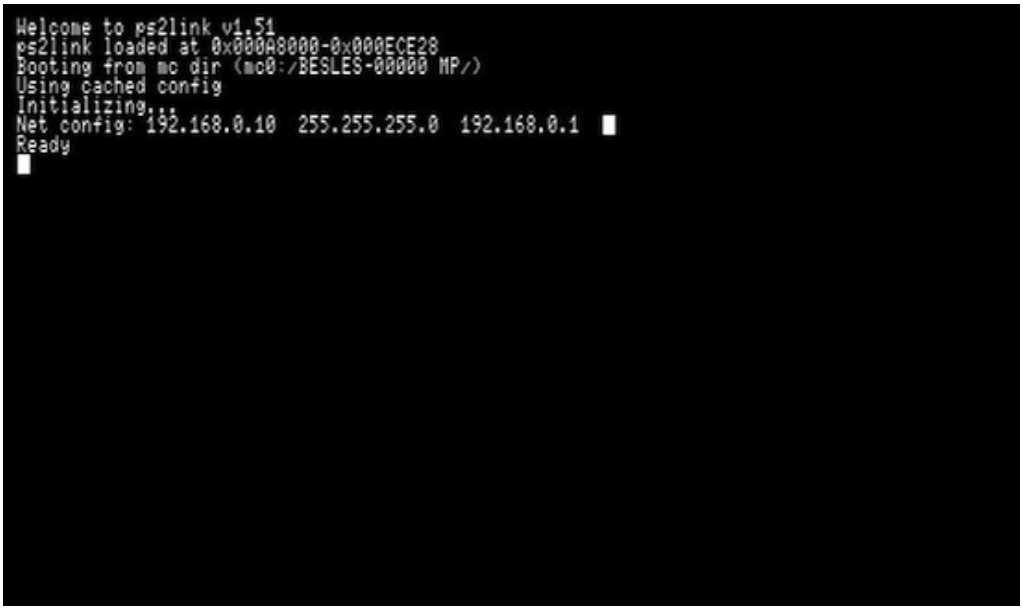

Figura 14. Tela inicial do ps2link.

Ps2client é encarregado de enviar os comandos para o servidor e monitorar todo o retorno da aplicação. A aplicação, de um modo geral, funciona de uma forma bem simples, porém, por ser de código fonte aberto é possível incluir nela novas funções que não existem no projeto original.

O ps2gdb *stub* consiste em uma biblioteca que teve ser incluída nos projetos a fim de dar ao GDB acesso aos símbolos de debug que são inseridos no programa que esta rodando no Playstation 2.

Os testes foram realizados criando-se pequenos programas com alguns dos mais comuns erros cometidos durante a programação e por cima destes aplicados algumas das técnicas de teste de código e com a ajuda das ferramentas citadas em capítulos anteriores.

Entre os erros citados estão:

- Confundir o operador identidade (==) com o operador de atribuição (=) em comandos condicionais ou loops.
- Passagem de parâmetros fora do escopo permitido pela aplicação.
- Usando um ponteiro não inicializado.
- Passagem do endereço de uma variável ao invés de seu valor

Na grande maioria são erros de lógica cometidos pelo programador em um minuto de desatenção e que mais para frente pode vir a se tornar um problema muito grande para o projeto em desenvolvimento, erros que dificilmente serão detectados pelo compilador.

4. DESENVOLVIMENTO

Com tudo o que era necessário chega a hora de partir para a prática do projeto que descreve passo a passo todas as etapas usadas para configurar e testar as ferramentas e teorias descritas em capítulos anteriores neste documento.

4.1. FERRAMENTAS

Como as ferramentas são fornecidas em seu código fonte foram compiladas para uso no sistema operacional escolhido, Windows 7, com a ajuda do Cygwin um programa que emula o ambiente Linux no windows.

O PS2SDK então foi baixado do repositório e o script `toolchain-sudo.sh` executado, este está na pasta scripts, e tem por sua vez a função de baixar os programas necessários para criar o ambiente de desenvolvimento para a plataforma, tal como binutils, gcc, newlib, ps2sdk e o ps2client e os compila nesta mesma ordem.

Foi preciso colocar o caminho da aplicação na variável PATH do sistema acrescentando as seguintes linhas no arquivo .bashrc que está no diretório home do usuário:

```
export PS2DEV=/usr/local/ps2dev
export PATH=%PATH:$PS2DEV/Bin
export PATH=%PATH:$PS2DEV/ee/bin
export PATH=%PATH:$PS2DEV/iop/bin
export PATH=%PATH:$PS2DEV/dvp/Bin
```

```
export PS2SDK=$PS2DEV/ps2sdk
export PATH=$PATH:$PS2DEV/Bin
export PATH=$PATH:$PS2DEV/ps2eth
export PATH=$PATH:$PS2DEV/gsKit
```

Isso foi necessário para poder acessar os programas diretamente pelo shell sem ter que digitar o caminho todo das ferramentas. Todas estas alterações foram fitas nos arquivos contidos na pasta de usuário criada com a instalação do emulador de terminal Sigwin.

Com o ps2sdk configurado pode-se então compilar o fonte do projeto ps2gdbstub apenas rodando o comando make dentro da pasta do programa, então o ps2gdbStub.a já está pronto para ser utilizado e incluído nos projetos a serem depurados.

O próximo passo foi compilar o GDB, foi preciso passar a opção --target=mips-idt-elf na hora de rodar o comando *configure* e um patch criado pelo mesmo autor do ps2gdbStub. Isso fez com que o gdb consiga entender os símbolos de debug e o arquivo binário que é gerado com o PS2SDK para o *Playstation* 2, onde roda um processador MIPS com formato binário ELF. A versão usada foi a 5.3.

O Ps2Client-gui é um programa que foi desenvolvido durante a criação trabalho, criado na linguagem *python* utilizando da biblioteca wxpython para criar a interface. Este serve como uma interface gráfica para o programa ps2client que roda em linha de comando tendo, por tanto implementadas todas as funções deste, mas de uma forma mais amigável, na forma de janelas e menus. Uma ferramenta deste tipo ainda não se encontrava disponível para o sistema mesmo este já estando a um bom tempo no mercado.

Com este é possível carregar o programa para a memória principal do *Playstation* 2 bem como código para os seu coprocessadores de entrada e saída, matemática matricial e coprocessador gráfico além de poder obter dados dos registradores e memórias tendo a vantagem de não precisar abrir múltiplos terminais, um para cada função, assim como tem que ser feito com ps2client.

Na figura 15 é mostrada a tela principal da aplicação onde se pode escolher o programa que será executado e onde é exibido o *log* com todas as mensagens enviadas pelo programa que esta sendo executado e o ps2client-gui.

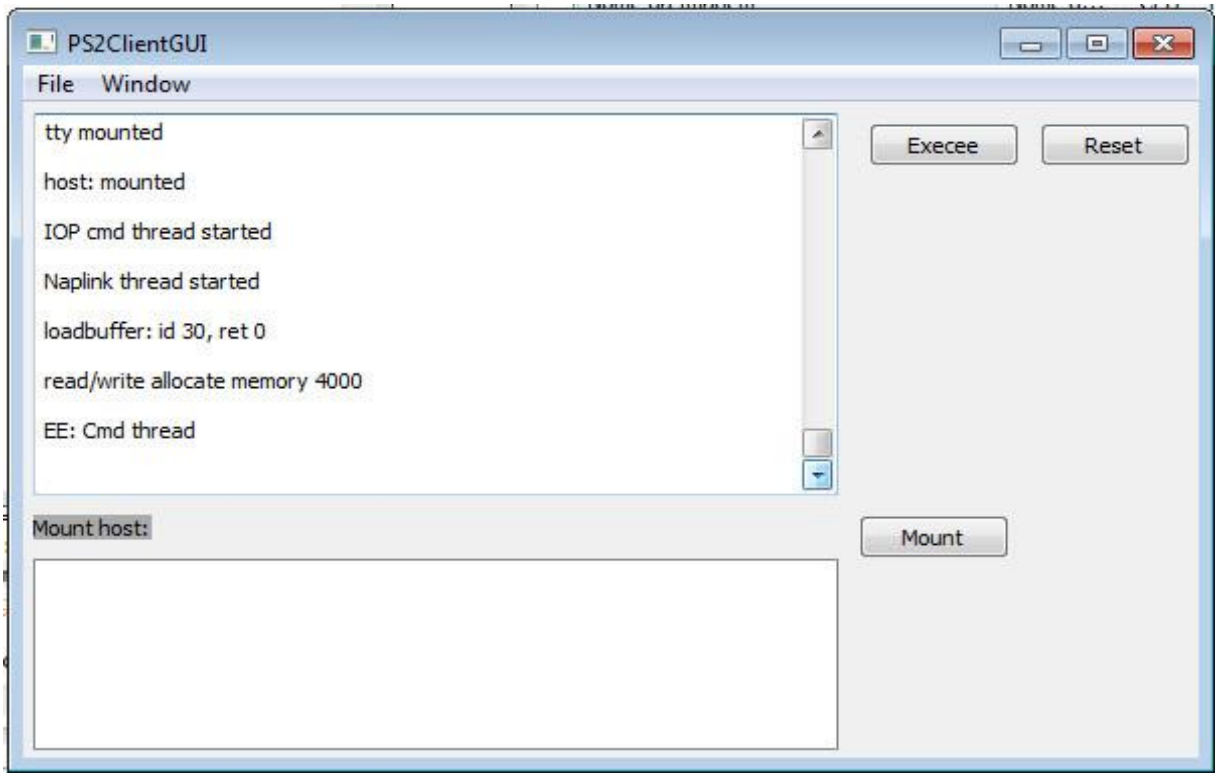

Figura 15. Tela principal do PS2ClientGUI.

Como se trata de uma interface para o ps2client este é executado de fundo pela aplicação ps2client-gui como um sub processo. O subprocesso criado é gerenciado por rotinas da biblioteca *subprocess* do *python* que dá a possibilidade de passar argumentos na hora da execução e também de monitorar a saída do subprocesso enquanto este está em execução.

Afim de não travar as outras funções enquanto o monitoramento do sub processo é feito, uma *thread* é usada. Esta *thread* então passa a saída do subprocesso para os respectivos campos no programa.

O uso de tais atributos disponíveis na linguagem *python* propiciou o funcionamento do programa, porem, ao se gerenciar um subprocesso através de threads fica difícil saber quando este foi liberado para executar uma outra função, o que pode causar um mal funcionamento do programa e até o seu travamento devido a várias instancias do subprocesso rodando na memória ao mesmo tempo.

Por sua vez, é o ps2client quem faz a troca de mensagens entre o PC e a aplicação rodando no sistema de testes se utilizando de *sockets* e dos protocolos TCP para receber requisições e enviar respostas durante a execução do programa e do protocolo UDP usando para envido dos comandos e receber os *logs* enviados pelo programa.

4.2. ESTUDOS DE CASO

De posse de todas as ferramentas necessárias para desenvolvimento e depuração de programas para a plataforma, seguiu-se com a criação de pequenos programas em linguagem C com a finalidade de testar, no ambiente real, o comportamento do programa contendo uma determinada falha. O objetivo foi de como chegar até ela para corrigi-la usando para isto as aplicações propostas.

Os códigos fonte precisaram ser compilados para gerar os símbolos de *debug*, usando a opção –g do compilador GCC, e foi incluída a biblioteca "ps2gdbStub.a" alem de ser necessário fazer uma chamada a função "gdb_stub_main(int argc, char *argv[])" antes da função principal do programa, "main(int argc, char *argv[])". Abaixo está a imagem 16 que mostra o script criado para automatizar a tarefa de compilar o código fonte e a imagem 17 que mostra como foi feita a declaração da função que inicia o ps2gdbStub e sua chamada na função *main()* através da chamada a função na linha 13.

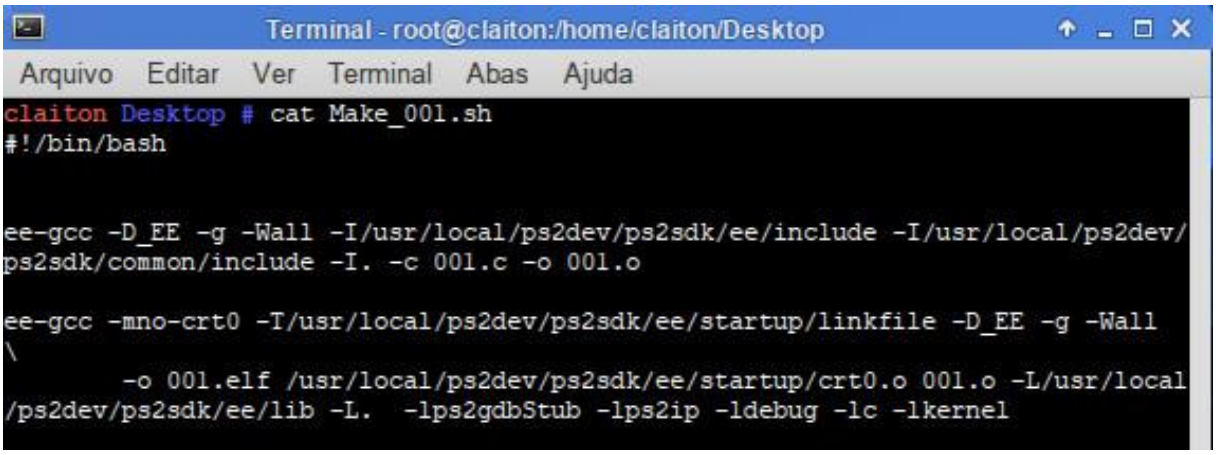

Figura 16. Script de automatização de compilação.

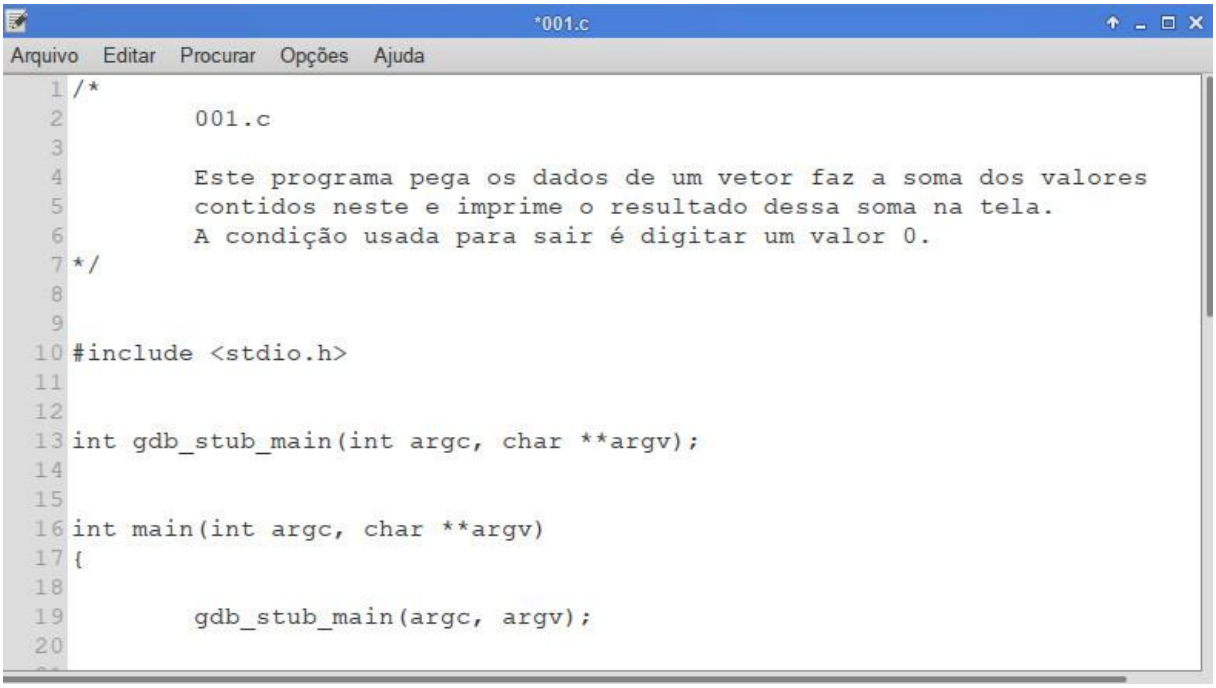

Figura 17. Declaração e invocação da função que inicia o debug.

4.2.1. Estudo de caso 1: Uso de operadores de forma errada

O primeiro programa a ser testado, 001.c, utiliza os valores de um vetor e soma estes até encontrar um valor “0” fazendo-o parar e mostrar o valor da soma, mas tinha como defeito a troca do operador de comparação "==" pelo operador de atribuição "=" no comando que testava o valor da variável o que fazia com que o programa entrasse em um loop infinito e nada fosse mostrado e causasse um erro devido ao acesso a áreas de memória não alocadas para o vetor. A seguir são mostrados os passos utilizados para compilar, executar e testar o programa:

O programa, em primeiro lugar, foi compilado utilizando as configurações mostradas na imagem 16. Com *playstation* 2 iniciado e o ps2link carregado através da função de *boot* por *pendrive*, existente no chip de desbloqueio Thunder Pro, uma tela igual a da figura 14 é exibida, em seguida o programa foi colocado para execução através do comando *ps2cliet -h 192.168.1.10 execee host:001.elf* , este foi executado no computador através do prompt de comandos do Windows 7. Na tela do terminal onde foi iniciado o ps2client vejo que não é exibido o resultado da soma. A técnica utilizada no teste foi a *Wolf Fence*, descrita no capitulo 2.6.1, e então foi colocada uma linha de código no final do laço para mostras os valores das variáveis, figura 18, um simples *printf*, função que esta contida na biblioteca *stdio.h* e que é

usada para imprimir texto na tela, a fim de ver seu comportamento durante a execução do programa, a saída é mostrada na figura 19.

```
32      while(1)
33      {
34          total = total + vetor[i];
35          if(vetor[i] = 0) break;
36
37          printf("pos: %d - val: %d \n", i, vetor[i]);
38
39          i=i+1;
40      }
```

Figura 18. Trecho do código usado para mostrar o valor dos dados no vetor.

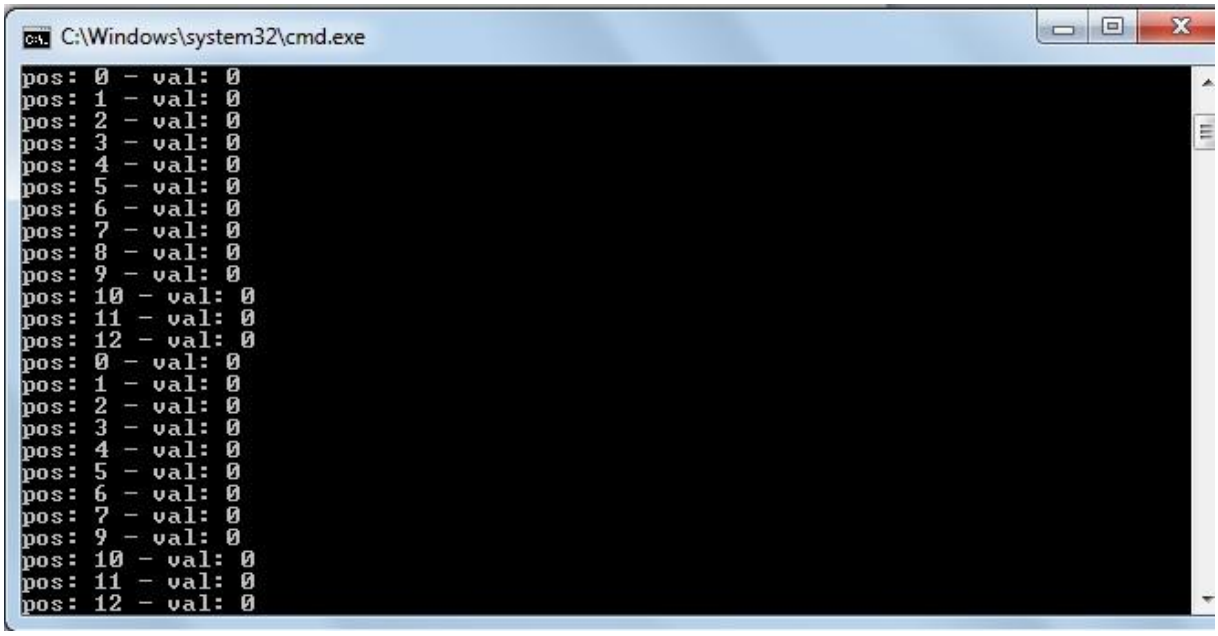

Figura 19. Saída do programa 001.

Como o problema encontrado fez o programa não só zerar o conteúdo do vetor, mas também fazer com que a condição de saída do loop não fosse satisfeita chega-se facilmente a conclusão de que é nesta parte do programa que se deve dar atenção para fazer a correção deste.

A simples inclusão de uma linha de comando fez com que o problema ficasse mais evidente facilitando assim encontrar o lugar do erro.

Essa técnica só é eficaz se tiver como ver as mensagens emitidas, e em se tratando de um sistema embarcado para uma finalidade especifica quase nunca dispões de um meio de exibir essas mensagens são nestes casos que essas ferramentas auxiliam.

4.2.2. Estudo de caso 2: Forma errada de incrementar um ponteiro

O segundo caso de teste, 002.c, consiste em um programa que faz uso de ponteiros. Para um bom gerenciamento da memória em sistemas, e em especial sistemas embarcados onde a quantidade de memória é reduzida.

É criado e inicializado um ponteiro que vai servir de contador num laço, a hora que a variável chegar a certo valor o laço deve parar de ser finalizado. Contudo, o incremento dá variável ponteiro é feita da forma errada e ao invés de seu valor ser incrementado é o endereço da mesma na memória que é incrementado uma posição a frente.

De forma diferente do teste anterior que mostrou o uso de uma técnica de depurarão de código, este foi feito utilizando a ferramenta de depuração GDB.

```
14  #include <stdio.h>
15  #include <stdlib.h>
16
17
18  int main(int argc, char **argv)
19  {
20
21      // Declaração da variável
22      int *i;
23      // Alocando memória para variável
24      i = (int*)malloc(1);
25      *i = 9;
26
27      if(*i  < 10)
28      {
29          *i++;
30      }
31      printf("valor de i eh: %d", *i);
32
33      return 0;
34  }
```

Figura 20. Fonte do programa 002.

Assim como no caso de teste anterior, o primeiro passo foi rodar o programa e verificar a saída no ps2client, figura 21. O valor de retorno foi 0 ao invés de 10.

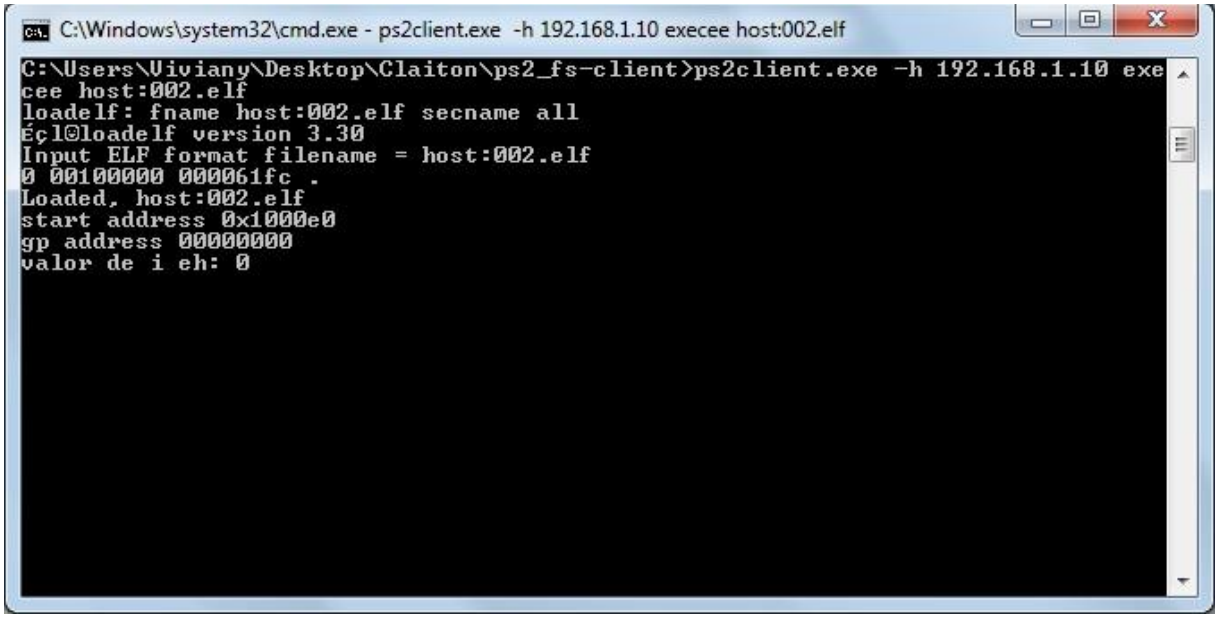

Figura 21. Saída do programa 002.elf.

O passo seguinte foi executar o programa no gdb, passo a passo, exibindo o valor da variável "i" e seu endereço. A saída do GDB é exibida na figura 22.

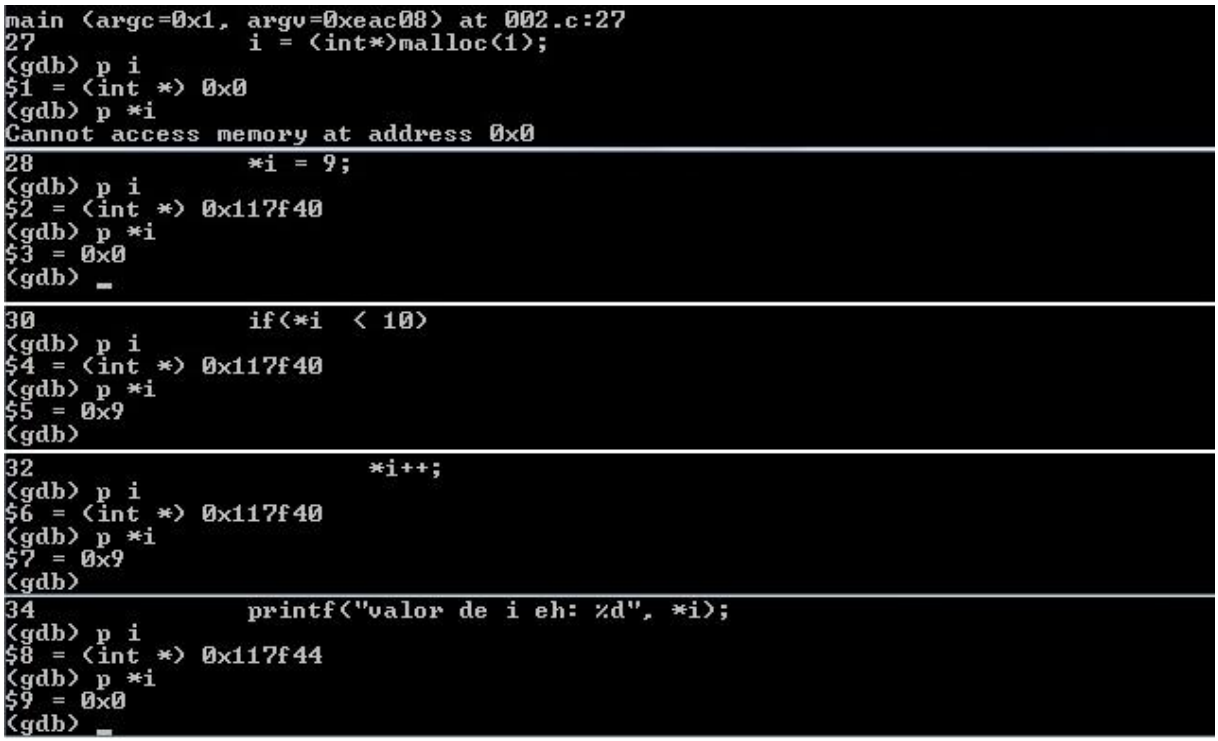

Figura 22. Saida do GDB.

Pode se ver claramente que na linha 27, antes de ser alocado espaço na memória para variável ela tem o valor 0 e não aponta para endereço algum na memória. Na linha 28, depois

de alocar espaço na memória para variável, ela possui um endereço e seu valor continua em 0. E então na linha 30 ela já está com seu valor alterado devido a atribuição feita na linha 28. Enfim na linha 34 podemos ver que a atribuição feita dentro do bloco condicional, seu endereço é alterado fazendo assim seu valor também ser alterado para algo diferente do correto.

4.2.3. Estudo de caso 3: Obtendo informações do programa

No terceiro caso de teste foi usado um código distribuído como exemplo junto ao PS2SDK. Este faz uso da capacidade gráfica do sistema através da inclusão da biblioteca libgs, que dá acesso aos comandos gráficos do *Playstation 2,* e gera na tela três retângulos coloridos que se movem de um lado para o outro.

O teste teve como objetivo mostrar o uso da ferramenta dos testes anteriores para exibir informações sobre a execução do programa.

Como o programa esta dividido em várias etapas este foi modificado para que envie uma mensagem para o ps2client a cada inicio e termino de uma destas e por fim mostra informações dos quadrados exibidos na tela, sua posição na mesma.

Funções *printf()* foram usadas no inicio e final das etapas de inicialização do vídeo, figura 23 linhas 93 e 118, da criação dos retângulos, figura 24 nas linhas 127 e 152.

```
90  int InitGraphics(void)
91  {
92
93      printf("Iniciando graficos...");
94
95      unsigned int FrameBufferVRAMAddress;
96
97      GsInit(GS_INTERLACED, GS_MODE_NTSC, GS_FFMD_INTERLACE);
98      FrameBufferVRAMAddress=GsVramAllocFrameBuffer(SCREEN_WIDTH, SCREEN_HEIGHT, GS_PIXMODE_32);
99      GsSetDefaultDrawEnv(&draw_env, SCREEN_WIDTH, SCREEN_HEIGHT);
100     //Retrieve screen offset parameters.
101     ScreenOffsetX=draw_env.offset_x;
102     ScreenOffsetY=draw_env.offset_y;
103     GsSetDefaultDrawEnvAddress(&draw_env, FrameBufferVRAMAddress, SCREEN_WIDTH/64, GS_PIXMODE_32);
104     GsSetDefaultDisplayEnv(&disp_env, SCREEN_WIDTH, SCREEN_HEIGHT, 0, 0);
105     GsSetDefaultDisplayEnvAddress(&disp_env, FrameBufferVRAMAddress, SCREEN_WIDTH/64, GS_PIXMODE_32);
106     //execute draw/display environment(s)  (context 1)
107     GsPutDrawEnv1(&draw_env);
108     GsPutDisplayEnv1(&disp_env);
109     //set some common stuff
110     GsOverridePrimAttributes(GS_DISABLE, 0, 0, 0, 0, 0, 0, 0, 0);
111     // contex 1
112     GsEnableAlphaTransparency1(GS_ENABLE, GS_ALPHA_GEQUAL, 0x01, 0x00);
113     // contex 2
114     GsEnableAlphaTransparency2(GS_ENABLE, GS_ALPHA_GEQUAL, 0x01, 0x00);
115     GsEnableAlphaBlending1(GS_ENABLE, 0);
116     GsEnableAlphaBlending2(GS_ENABLE, 0);
117
118     printf("OK\n");
119
120     return 0;
121 }
```

Figura 23. Trecho do código usado para inicializar o vídeo.

```
125  static int InitSprites(void)
126  {
127      printf("Iniciando sprites...");
128
129      int i,j;
130
131      for(i=0,j=0;i<MAX_SPRITES;i++,j++)
132      {
133          if(j>9)j=0;
134
135          sprites[i].x_pos    =i;
136          sprites[i].y_pos    =i/2;
137
138          sprites[i].x_dir    =0;
139          sprites[i].y_dir    =0;
140
141          sprites[i].x_speed  =i/x_randspeeds[j];
142          sprites[i].y_speed  =x_randspeeds[j]+1;
143
144          if(sprites[i].x_speed<=0)sprites[i].x_speed=2;
145          if(sprites[i].y_speed<=0)sprites[i].y_speed=1;
146
147
148          //set color
149          sprites[i].color = randcolor[j];
150      }
151
152      printf("OK.\n");
153
154      return 0;
155  }
```

Figura 24. Trecho do código usado para inicializar os retângulos.

Assim que as instruções nestas linhas são executadas é enviada uma mensagem ao ps2client que mostra na tela do computador o conteúdo delas.

Também colocada na função de movimentação, figura 25 linha 210, assim mostrando na tela do ps2client a posição destes.

```
157  static int MoveSprites(void)
158  {
159      int i;
160
161      for(i=0;i<MAX_SPRITES;i++)
162      {
163          // mod a little
164      //  sprites[i].x_speed++;
165      //  sprites[i].y_speed++;
166
167          if(sprites[i].x_speed>10)sprites[i].x_speed     =sprites[i].x_speed-10;
168          if(sprites[i].y_speed>10)sprites[i].y_speed     =sprites[i].y_speed-10;
169
170          //x
171          if(sprites[i].x_dir==0)
172          {
180          else if(sprites[i].x_dir==1)
181          {
189
190          //y
191          if(sprites[i].y_dir==0)
192          {
200          else if(sprites[i].y_dir==1)
201          {
209
210          printf("Sprite %d-> pos X=%d, Y=%d\n", i, sprites[i].x_pos, sprites[i].y_pos );
211
212      }
213
214      return 0;
215  }
```

Figura 25. Trecho do código usado para movimentar os retângulos na tela.

Na figura 26 é mostrada a tela do programa em execução e a saída gerada no ps2client executado no PC.

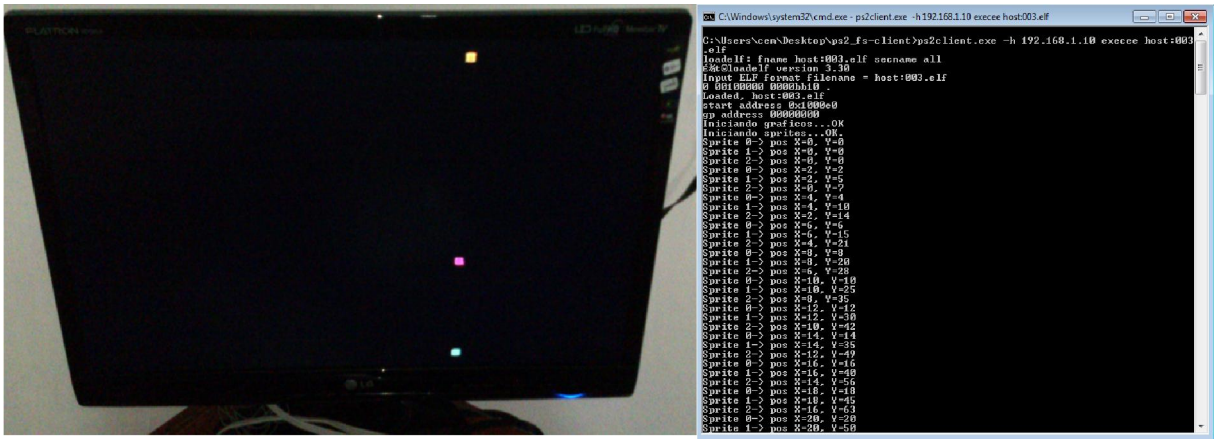

Figura 26. Programa 003 rodando no sistema e ao lado a saída no ps2client.

Estas informações também podem ser úteis para ajudar a descobrir possíveis problemas no programa já que se o programa parar se sabe exatamente onde foi alem de poder

ver qual parte do programa esta demorando mais para terminar, podendo assim otimizar estas partes.

4.3. VANTAGENS

Todo trabalho de enviar e testar os programas foram feitos utilizando as ferramentas e a comunicação em rede, contudo na maioria dos dispositivos embarcados o modo convencional de gravação e depuração é através de um hardware especifico, o gravador, que no caso dos PICs e ARMs podem ser usados o PICKIT e o JLINK respectivamente, ligados de forma serial aos mesmos.

Estes possuem a capacidade de gravar o programa na memória e função para depuração do código utilizando dos aplicativos disponibilizados junto ao kit.

Em se tratando do sistema testado, ele se utiliza de DVDs como método padrão para execução de aplicativos e por isso cada programa compilado teria que ser gravado em mídia para posteriormente ser executado no sistema.

O uso de um *bootloader*, da comunicação em rede oferecido por estes dispositivos junto a capacidades deles carregarem um programa para sua memória principal proporciona uma maior agilidade no processo de reprogramação não sendo preciso estar em contato direto, podendo tudo ser feito a distancia.

Como falado anteriormente neste documento um sistema embarcado possui um hardware especifico para uma única função ou uma gama pequena de funções não dispondo de periféricos específicos para exibir possíveis saídas de erros.

Por isso, sem o uso de tais recursos e ferramentas o processo de teste e depuração de erros seria mais trabalhosa visto que não possui um modo de ver a saída do programa enquanto este é executado do mesmo jeito que pode ser visto com estas ferramentas.

E o uso de uma forma de comunicação comum a todos os vários dispositivos permite que uma única ferramenta seja usada diminuindo assim a curva de aprendizado que se tem quando migra de um sistema para outro.

5 – CONCLUSÃO

Ferramentas para auxiliar nas tarefas de desenvolvimento e depuração existem muitas e este trabalho mostrou um estudo e um exemplo sobre a utilização de ferramentas que utilizam a comunicação em rede para estas tarefas.

Com estas foram feitos o envio do código para o sistema embarcado alvo bem como a depuração do código e exibir informações de status das variáveis do programa que esta sendo executado no sistema embarcado na tela do PC através da comunicação em rede.

As ferramentas são de uso bem simples porém, não possuem uma interface muito amigável o que pode assustar e dificultar o trabalho num primeiro momento.

Uma dificuldade encontrada é o fato de que boa parte da documentação e fóruns especializados se encontra na língua inglesa.

Para tentar resolver a questão da interface amigável, que não existe para o ps2client, o programa ps2client-gui foi criado para possuir todas as funções do ps2client e ficar independente do mesmo.

Há também um plano de um projeto futuro para a plataforma usando as ferramentas aqui descritas e os conhecimentos adquiridos durante o desenvolvimento deste documento.

6 – BIBLIOGRAFIA

[1] ALMEIDA, Marcelo Barros de. **Sistemas embarcados com Linux - primeiros passos**, 2008. Disponível em <http://linuxabordo.com.br/>, Acesso em: 12/03/2013.

[2] SILVA, R. A. **Programando microcontroladores PIC : Linguagem "C"**. São Paulo. Ensino profissional, 2006. 78 p.

[3] LEMOS, S. D., GUEDES L. A. **Contribuições ao Desenvolvimento de Sistemas Digitais com Reconfiguração Parcial Dinâmica.** Trabalho de conclusão de curso.

[4] NETO, Arilo Cláudio Dias. **Introdução a Teste de Software**. Engenharia de Software Magazine.

[5] MALDONADO, José Carlos; BARBOSA, Ellen Francine; VINCENZI, Auri Marcelo Rizzo. **Introdução ao teste de software,** Trabalho de conclusão de curso. Universidade de São Paulo. ICMC/USP, São Carlos . SP.

[7] STALLMAN R., PESCH R., SHEBS S. **Debugging with gdb**, 10 ed. Free Software Foundation, 2013.

[8] **Utilizando um debugger – OllyDbg.** Disponivel em:

<http://www.hardware.com.br/comunidade/utilizando-debugger/785195/>. Acesso em :05-06-2013.

[9] GOLDBERG, Corey. **The "Wolf Fence" Algorithm for Debugging**. Disponivel em: <http://coreygoldberg.blogspot.com.br/2008/12/wolf-fence-debugging.html>. Acesso em 23/05/20013.

[10] PEREIRA, Adriano. **Tutorial DDD**, Programa de Educação, Tutorial – PET Ciência da Computação, Santa Maria, agosto de 2008. Disponível em: <http://www-usr.inf.ufsm.br/~apereira/ed/ddd_tutorial.pdf>. Acesso em: 25/06/2013.

[11] Steve Heath. **Embedded Systems Design**, Burlington MA, Elsevier Ltd, 2002.

[12] **What is DDD?.** Disponível em: <http://www.gnu.org/software/ddd>. Acesso em: 04/07/2013.

[13] Nosotros. **Arquitectura PlayStation2.** Disponivel em: <http://www.yumpu.com/es/document/view/7092095/arquitectura-playstation2-departamento-de-arquitectura-y-> . Acesso em 2011.

[14] AMANN, Paul; OFFUTT, Jef. **Introduction to Software Testing.**

[15] WITCZAK, Ariel Bolzan; MAYER, Danielle; CHIARELLO, Marcos. **Guia Técnicas de Teste Metodologia Celepar**, 28/08/2009**.**Celepar Informática do Paraná.

[16] **The GDB GUI**, Disponivel em: <http://sourceware.org/insight/>. Acesso em 22/07/2013.

[17] SCARPINO, Matthew. **Interfacing with the CDT debugger, Part 1: Understand the C/C++ debugger interface**. Disponível em: <http://www.ibm.com/developerworks/library/os-eclipse-cdt-debug1/>. Acesso em: 22/07/2012.

[18] THOMAS, Boby. **Remote Debugging using GDB,** 28/07/2006**.** Disponivel em: <http://www.codeproject.com/Articles/14983/Remote-Debugging-using-GDB> . Acesso em: 23/07/2013.

[19] SÁNCHEZ, Alberto Fernández; GARCÍA, Diego Arnáiz; ESQUIFINO, Javier Martíns. **Cluster de Playstation 2**, trabalho de conclusão de curso. Univerdisad Complutense de Madrid, 2004.

[20] GREEN, Robin. **Procesural Rendering on Playstation 2**. Sony Computer Entertainment America, 2002. Disponivel em: <http://lukasz.dk/files/rgreen_procedural_renderinggdc2001.pdf>. Acesso em 05/10/2013.

[21] **PIC32 Bootloader**. Microchip. Disponivel em: <http://ww1.microchip.com/downloads/en/AppNotes/01388B.pdf>. Acesso em: 05/10/2013

[22] SMALLRIDGE, Andrew. **Ethernet Bootloader**. Brush Electronics. Disponivel em: <http://www.smallridge.com.au/download/BE%20Ethernet%20Bootloader.pdf>. Acesso em: 05/10/2013

[23] SVENDSLI ,Odd Jostein, **Atmel's Self-Programming Flash Microcontrollers**. Disponivel em: <http://www.atmel.com/Images/doc2464.pdf>. Acesso em: 05/10/2013

APÊNDICES

APÊNDICE A – Código fonte do primeiro estudo de caso.

C:\Users\cem\Desktop\CAA\ps2_fs-client\001.c quarta-feira, 13 de novembro de 2013 20:09

```
/*
    001.c

    Este programa pega os dados de um vetor faz a soma dos valores
    contidos neste e imprime o resultado dessa soma na tela.
    A condição usada para sair é digitar um valor 0.
*/


#include <stdio.h>


int main(int argc, char **argv)
{

    int vetor[10] = {5, 9, 12, 25, 1, 3, 16, 4, 9, 0};
    int total = 0;

    int i = 0;
    //for(int i=0;;i+=1)
    while(1)
    {
        total = total + vetor[i];
        if(vetor[i] = 0) break;

            printf("pos: %d - val: %d \n", i, vetor[i]);

        i=i+1;
    }

    printf("%d", total);


    return 0;
}


/*
    Neste caso, como é usado um array ao invez da entrada do usuário, ele
    gera um erro ao acessar áreas de memória não alocadas para o programa.
    No caso de aver uma interação com o usuário o programa iria entrar em
    um loop onde não pararia de pedir para o usuário digitar um valor.
*/
```

-1-

APÊNDICE B – Código fonte do segundo estudo de caso.


```c
/*
    002.c

    Uso de ponteiros na linguagem C é imprescindível, alocação dinâmica
    de memória se faz necessária a todo momento para gerenciar melhor
    os recursos disponiveis do sistema. Porem o mal uso ou falta de
    atenção pode levar a erros graves no programa.
*/

#include <stdio.h>
#include <stdlib.h>

int gdb_stub_main(int argc, char **argv);

int main(int argc, char **argv)
{
    gdb_stub_main(argc, argv);


    // Declaração da variável
    int *i;
    // Alocando memória para variável
    i = (int*)malloc(1);
    *i = 9;

    if(*i  < 10)
    {
        *i++;
    }
    printf("valor de i eh: %d", *i);

    return 0;
}
```

APÊNDICE C – Código fonte do terceiro estudo de caso.


```
/*
# _____     ___ ____     ___ ____
#  ____|   |    ____|   |        | |____|
# |     ___|   |____ ___|    ____| |    \    PS2DEV Open Source Project.
#-----------------------------------------------------------------------
# (c) 2009 Lion
# Licenced under Academic Free License version 2.0
# Review ps2sdk README & LICENSE files for further details.
#
*/

#include <kernel.h>
#include <libgs.h>
#include <stdio.h>

typedef struct
{
    int x_pos;      // x position of sprite
    int y_pos;      // y position of sprite

    char x_dir;     // x direction 0=left,1=right
    char y_dir;     // y direction 0=up,1=down;

    char x_speed;   // speed moving in x direction
    char y_speed;   // speed moving in y direction

    GS_RGBAQ color;
}MAVING_SPRITE;

const static char x_randspeeds[10] = {1,4,6,5,2,3,8,6,6,7};

const static GS_RGBAQ randcolor[10] ={  {28 ,200,200,0x80,0.0f},
                    {255,  0,128,0x80,0.0f},
                    {255,128,0,0x80,0.0f},
                    {255,255,255,0x80,0.0f},
                    {0  ,128,255,0x80,0.0f},
                    {255,255,128,0x80,0.0f},
                    {128,128,255,0x80,0.0f},
                    {226,137,245,0x80,0.0f},
                    {172,177,210,0x80,0.0f},
                    {221,180,162,0x80,0.0f}
                };


#define MAX_SPRITES     3 //tava 1000

#define SCREEN_WIDTH        640
#define SCREEN_HEIGHT       448

static short int ScreenOffsetX, ScreenOffsetY;

#define GIF_PACKET_MAX      10

static GS_DRAWENV       draw_env;
static GS_DISPENV       disp_env;

static GS_GIF_PACKET        packets[GIF_PACKET_MAX];
```

```
static GS_PACKET_TABLE      giftable;

static int InitGraphics(void);
static int InitSprites(void);
static int MoveSprites(void);
static int DrawSprites(GS_PACKET_TABLE *table);

int main(int argc, char *argv[])
{
    InitGraphics();

    giftable.packet_count   = GIF_PACKET_MAX;
    giftable.packets    = packets;

    InitSprites();

    while(1)
    {
        GsGifPacketsClear(&giftable);       // clear the area that we are going to put the
        sprites/triangles/....

        MoveSprites();
        DrawSprites(&giftable);         //add stuff to the packet area

        GsDrawSync(0);
        GsVSync(0);
        GsClearDrawEnv1(&draw_env);     // clear the draw environment before we draw stuff
        on it
        GsGifPacketsExecute(&giftable, 1);  // set to '1' becuse we want to wait for drawing
        to finish. if we dont wait we will write on packets that is currently writing to the
        gif
    }

    return 0;
}

int InitGraphics(void)
{
    ////
    printf("Iniciando graficos...");
    ////

    unsigned int FrameBufferVRAMAddress;

    GsInit(GS_INTERLACED, GS_MODE_NTSC, GS_FFMD_INTERLACE);

    FrameBufferVRAMAddress=GsVramAllocFrameBuffer(SCREEN_WIDTH, SCREEN_HEIGHT, GS_PIXMODE_32);
    GsSetDefaultDrawEnv(&draw_env, SCREEN_WIDTH, SCREEN_HEIGHT);
    //Retrieve screen offset parameters.
    ScreenOffsetX=draw_env.offset_x;
    ScreenOffsetY=draw_env.offset_y;
    GsSetDefaultDrawEnvAddress(&draw_env, FrameBufferVRAMAddress, SCREEN_WIDTH/64,
    GS_PIXMODE_32);

    GsSetDefaultDisplayEnv(&disp_env, SCREEN_WIDTH, SCREEN_HEIGHT, 0, 0);
    GsSetDefaultDisplayEnvAddress(&disp_env, FrameBufferVRAMAddress, SCREEN_WIDTH/64,
    GS_PIXMODE_32);
```

```
    //execute draw/display environment(s)  (context 1)
    GsPutDrawEnv1(&draw_env);
    GsPutDisplayEnv1(&disp_env);

    //set some common stuff
    GsOverridePrimAttributes(GS_DISABLE, 0, 0, 0, 0, 0, 0, 0, 0);

    // contex 1
    GsEnableAlphaTransparency1(GS_ENABLE, GS_ALPHA_GEQUAL, 0x01, 0x00);
    // contex 2
    GsEnableAlphaTransparency2(GS_ENABLE, GS_ALPHA_GEQUAL, 0x01, 0x00);

    GsEnableAlphaBlending1(GS_ENABLE, 0);
    GsEnableAlphaBlending2(GS_ENABLE, 0);

    ////
    printf("OK\n");
    ////

    return 0;
}

static MAVING_SPRITE sprites[MAX_SPRITES];

static int InitSprites(void)
{

    ////
    printf("Iniciando sprites...");
    ////

    int i,j;

    for(i=0,j=0;i<MAX_SPRITES;i++,j++)
    {
        if(j>9)j=0;

        sprites[i].x_pos    =i;
        sprites[i].y_pos    =i/2;

        sprites[i].x_dir    =0;
        sprites[i].y_dir    =0;

        sprites[i].x_speed  =i/x_randspeeds[j];
        sprites[i].y_speed  =x_randspeeds[j]+1;

        if(sprites[i].x_speed<=0)sprites[i].x_speed=2;
        if(sprites[i].y_speed<=0)sprites[i].y_speed=1;


        //set color
        sprites[i].color = randcolor[j];
    }

    ////
    printf("OK.\n");
```

```
    ////

    return 0;
}

static int MoveSprites(void)
{
    int i;

    for(i=0;i<MAX_SPRITES;i++)
    {
        // mod a little
    //  sprites[i].x_speed++;
    //  sprites[i].y_speed++;

        if(sprites[i].x_speed>10)sprites[i].x_speed     =sprites[i].x_speed-10;
        if(sprites[i].y_speed>10)sprites[i].y_speed     =sprites[i].y_speed-10;

        //x
        if(sprites[i].x_dir==0)
        {
            sprites[i].x_pos    -= sprites[i].x_speed;
            if(sprites[i].x_pos<0)
            {
                sprites[i].x_pos=0;
                sprites[i].x_dir=1;
            }
        }
        else if(sprites[i].x_dir==1)
        {
            sprites[i].x_pos    += sprites[i].x_speed;
            if(sprites[i].x_pos>SCREEN_WIDTH)
            {
                sprites[i].x_pos = SCREEN_WIDTH;
                sprites[i].x_dir=0;
            }
        }

        //y
        if(sprites[i].y_dir==0)
        {
            sprites[i].y_pos    -= sprites[i].y_speed;
            if(sprites[i].y_pos<0)
            {
                sprites[i].y_pos=0;
                sprites[i].y_dir=1;
            }
        }
        else if(sprites[i].y_dir==1)
        {
            sprites[i].y_pos    += sprites[i].y_speed;
            if(sprites[i].y_pos>SCREEN_HEIGHT)
            {
                sprites[i].y_pos=SCREEN_HEIGHT;
                sprites[i].y_dir=0;
            }
        }
```

```
        ////
        printf("Sprite %d-> pos X=%d, Y=%d\n", i, sprites[i].x_pos, sprites[i].y_pos );
        ////
    }

    return 0;
}

static int DrawSprites(GS_PACKET_TABLE *table)
{
    int i;
    QWORD *p;

    for(i=0;i<MAX_SPRITES;i++)
    {
        //Use the uncached segment, to avoid needing to flush the data cache.
        p = (QWORD*)UNCACHED_SEG(GsGifPacketsAlloc(table, 5)); //Allocate 5 qword for 1
        untextured strite

        gs_setGIF_TAG(((GS_GIF_TAG  *)&p[0]), 4,1,0,0,0,0,1,0x0e);
        gs_setR_PRIM(((GS_R_PRIM    *)&p[1]), GS_PRIM_SPRITE,0, 0, 0, 1, 0, 0, 0, 0);
        gs_setR_RGBAQ(((GS_R_RGBAQ  *)&p[2]), sprites[i].color.r, sprites[i].color.g, sprites
        [i].color.b, sprites[i].color.a, sprites[i].color.q);
        gs_setR_XYZ2(((GS_R_XYZ     *)&p[3]), (ScreenOffsetX+sprites[i].x_pos)<<4,  (
        ScreenOffsetY+sprites[i].y_pos)<<4, 0x00000000);
        gs_setR_XYZ2(((GS_R_XYZ     *)&p[4]), (ScreenOffsetX+sprites[i].x_pos+10)<<4,   (
        ScreenOffsetY+sprites[i].y_pos+10)<<4, 0x00000000);
    }

    return 0;

}

/*EOF*/
```