**Entendendo e Compilando o meu Primeiro Exemplo**

Neste documento vamos compilar e rodar nosso primeiro programa exemplo, como se trata em aplicações de sistemas embarcados, a terminologia mais correta é firmware. Neste primeiro exemplo será discernido pontos importantes na organização tanto da estrutura do compilador como do firmware, para que o programador tenha condições de modificar a estrutura para atender as suas necessidades de projeto.

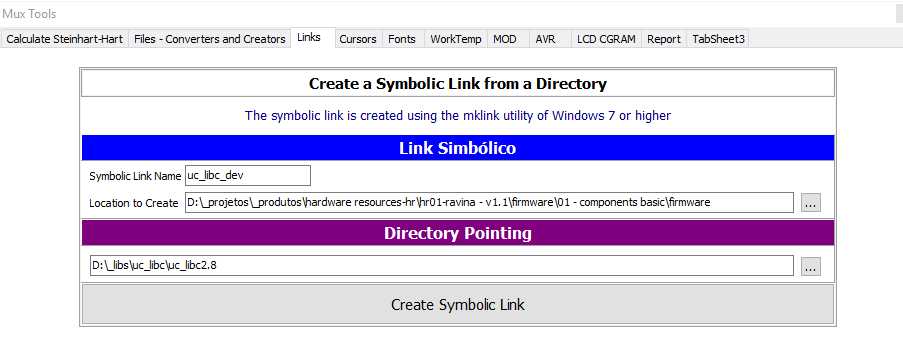
**Biblioteca uc\_libc**

A primeira coisa a ser feita é disponibilizar a biblioteca uc\_libc para os firmwares, sejam os exemplos aqui propostos ou os novos projetos que possam vir. A primeira intuição é copiar toda uc\_libc para dentro do firmware do projeto, funciona, mas não é uma boa ideia quando está em fase de desenvolvimento, e nesta fase é melhor separar uc\_libc do projeto. A separação da uc\_libc dos programas, se deve ao fato de que a uc\_libc está em constante atualização, com correções, adições de novos códigos e melhoria na performance. Dessa forma, sempre que vamos iniciar um novo projeto, vamos usar um uc\_libc mais confiável.

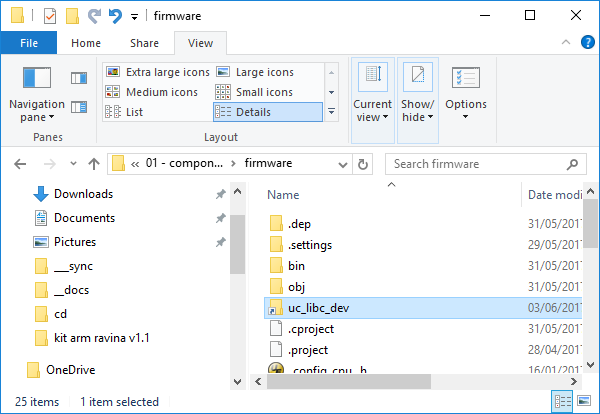
Uma boa técnica sem mexer na estrutura de compilação é criar um link simbólico para a biblioteca uc\_libc. Um link simbólico de uma pasta ou arquivo é um apontamento de seus respectivos conteúdos em outro local. Para auxiliar na criação de o link simbólico da uc\_libc utilize a ferramenta muxtools, vá na aba Links e entre com os seguintes campos:

* **Symbolic Link Name**: Entre com o nome da pasta simbólica que vai ser criada na pasta do firmware;
* **Location to Create**: Local onde será criado a pasta simbólica, neste caso é na pasta firmware do projeto;
* **Directory Point**: Local onde se encontra a uc\_libc.

Entrando com os devidos campos clique em criar o link simbólico.



A pasta simbólica uc\_libc\_dev foi criada dentro da pasta firmware do projeto. Para as ferramentas de desenvolvimento como make, Eclipse e outras, o conteúdo dentro de dentro da pasta simbólica é como se fosse o conteúdo de uma pasta local:

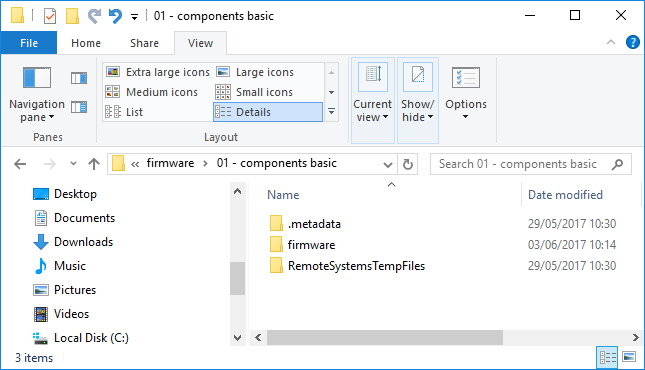


Faça isto para todos os programas que estão em desenvolvimento, ou se preferir, ignore esse procedimento e copie a uc\_libc para dentro do uc\_libc\_dev, que não está errado, fazemos este último procedimento quando finalizamos o produto. Pois isto garante que firmwares muitos antigos possam serem recompilados com uma uc\_libc sem as alterações drásticas que complique com a compilação.

**Projeto no Eclipse**

Há muitas maneiras de organizar um projeto no Eclipse, e vamos adotar onde criamos um ambiente Eclipse para cada projeto de firmware, este modelo tem com finalidade de obtermos uma melhor organização e clareza em relação aos projetos. Isto porque no modo padrão do Eclipse todos os projetos são concentrados em uma única área de trabalho.

O Eclipse trabalha em um conceito de workspace, contendo as configurações de trabalhado tanto do próprio Eclipse como o do projeto em si. Como citado em nosso modelo de organização no início, para cada projeto contém seu próprio workspace, logo para cada instância do Eclipse aberto é para um determinado projeto, ou seja, cada projeto terá sua própria pasta de trabalho e seu workspace. Abaixo mostra a pasta de um projeto onde estão os arquivos workspace do Eclipse e a pasta firmware com os arquivos fontes de programa:

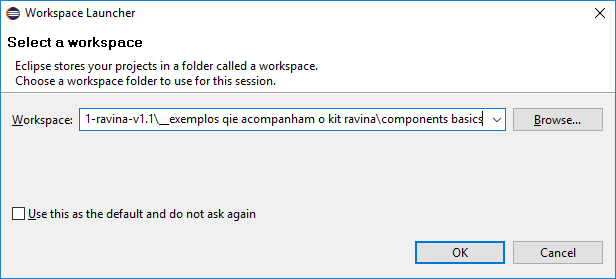


Atenção: Sempre deixe as fontes do firmware de uma pasta separada do workspace, senão o Eclipse se perde em sua organização.

Inicie o Eclipse:



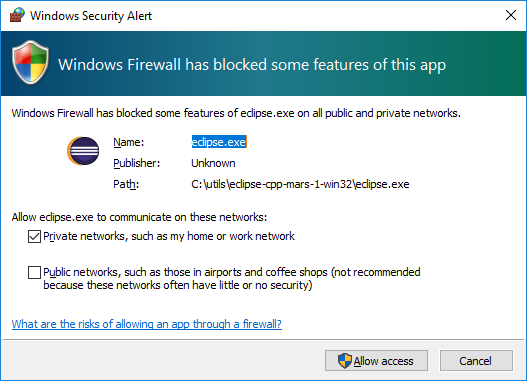
Aponte o workspace do projeto, começaremos com o projeto exemplo “componentes basics” porque contém os elementos principais do kit ARM Ravina para estudo desse documento:



Espere abrir o projeto:



Na primeira vez que o Eclipse é aberto o mesmo vai pedir a liberação do Firewall do Windows. Libere o acesso, isto porque o Eclipse utiliza a conexão de rede para instalar plug-ins e fazer depurações do firmware:



***As teclas de atalhos e outras ações do Eclipse foram mudadas baseado no documento*** “eclipse 04 - configurações pessoais.docx”.

**Compilador GCC**

O compilador GNU C foi desenvolvido incialmente pelo Richard Stallman em 1987 e está sendo mantido e atualizado por muitas comunidades hoje em dia. O GNU C abrange muitas arquiteturas, incluído ARM e AVR. O pacote GUN C (GCC) é completo com linker, assembler, depuradores e gravadores.

O compilador C e o assembler são usados para compilar os arquivos fontes e suas saídas são arquivos objetos. Os arquivos objetos ainda não são códigos executáveis no processador, ainda faltam os endereços fixos de execução que são determinados e preenchidos pelo linker.



O GNU Linker é usado para coletar os arquivos objetos, bibliotecas, consultar o linker script para saber a organização das memórias da plataforma computacional, e por fim criar um código de saída.

O arquivo de saída é usado para ser executado em RAM ou para ser usado como assistente de depuração em IDEs para identificar símbolos, endereços de memórias e outras ações. Caso preciso código ter o código executável seja para gravar em FLASH dos micros controladores, ou rodar nos processadores, é preciso ter o código binário puro, para isto é usado o utilitário do GCC chamado GNU Objcopy.

**Linker do Compilador**

O linker fornece um controle sobre o processo de compilação e linkagem do programa, permitindo uma especificação completa do arquivo de entrada e sua saída. O linker controla:

* Arquivo de entrada;
* Formato do arquivo;
* Formato de saída;
* Secções de endereços de memória de programa e de dados;
* Localização de blocos comuns.

O linker e dotado de uma linguagem de comandos e estes podem ser colocados em um arquivo chamado de linker script que veremos a seguir. O linker script é anexado ao programa durante a linkagem através da diretiva “–T” passado ao linker do compilador.

**Linker Script**

Antes de um programa ser compilado, o compilador deve ser instruído em que regiões se encontram as memorias do processador, independentemente do tipo e modelo do mesmo, mesmo que seja o modelo mais simples de todos. Essas informações são passadas através de um arquivo chamado linker script.

O linker script define um mapa de memórias do processador, caso esse arquivo não for especificado, um linker script padrão do compilador será assumido.

No ARM contém internamente memória FLASH e memória SRAM. Uma aplicação típica é executada na FLASH, embora possa ser executada em qualquer outra área de memória, enquanto as variáveis e STACKs points sempre são armazenadas da RAM. A ferramenta GNU GCC organiza as memórias em secções, tais como:

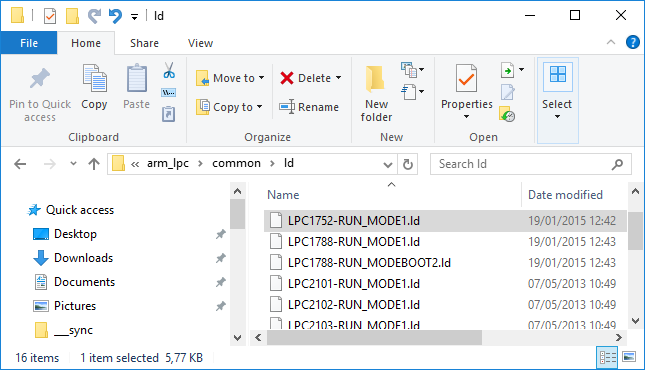
* Secção .text: Nesta secção é anexado o código executável e deve estar em uma parte da memória não volátil do sistema, como a FLASH;
* Secção .rodata: Nesta secção é anexada as variáveis de somente leitura (constantes) e deve estar em uma parte da memória não volátil do sistema, como a FLASH;
* Secção .data: Nesta secção é anexada as variáveis inicializadas e deve estar em uma parte de memória que pode ser lida e escrita, como a SRAM;
* Secção .bss: Nesta secção é anexada as variáveis não inicializadas e deve estar em uma parte de memória que pode ser lida e escrita, como a SRAM.

Na aplicação a secção “*.data*” contém todas as variáveis globais e estáticas inicializadas. O linker GNU criará uma cópia exata dessas variáveis no fim da secção “*.text*” com seus valores iniciais para leitura. Isto porque é preciso inicializar as variáveis após que ligar o processador. É de responsabilidade do programador, ou bootloader, copiar essas variáveis da secção “*.text*” para a secção “*.data*”.

Na aplicação a secção “*.bss*” contém as variáveis globais e estáticas não inicializadas e devem conter os valores iniciais zero. Também é de responsabilidade do programador, ou bootloader, limpar essa região de memória.

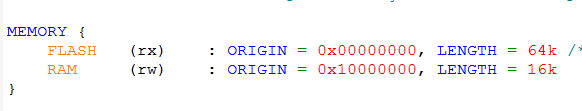
**Linker Script do Kit ARM Ravina**

Há muitos modos de definir como serão usadas as memórias para o mesmo sistema, em que local vai ficar o programa, se vai ficar na FLASH interna do processador ou em uma memória externa. Em que local serão alocadas as variáveis do sistema, se vai ficar na RAM interna ou na SDRAM externa. Diferente dos micros controladores como AVR e PIC por serem muitos simples essas configurações já estão explicitas nos seus compiladores, mesmo assim podemos modificar via comandos. Já no ARM as possibilidades são inúmeras, então é preciso escrever um linker script para mapear as memórias da plataforma e orientar o linker do compilador em quais as porções de memórias serão usadas para cada elemento do sistema. Convencionalmente o linker script tem a extensão “ld” e estão na pasta “uc\_libc\\_arch\arm\_lpc\common\ld”:

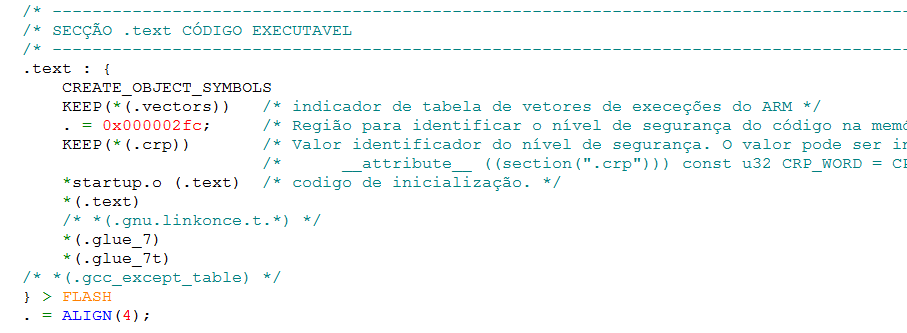


No kit ARM Ravina usamos uma linker script chamado “LPC1752-RUN\_MODE1.ld” e vamos descrever alguns pontos interessantes.

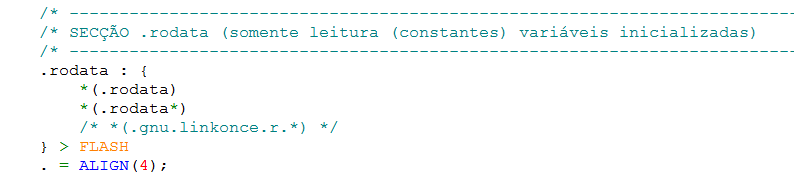
MEMORY: Especifica tipos, locais e tamanhos das memórias do processador. Neste linker script descreve que o micro controlador contém 64KBytes de memória de programa (FLASH), começando no endereço 0, e contém 16KBytes de memória de dados (SRAM) começando no endereço 0x10000000:



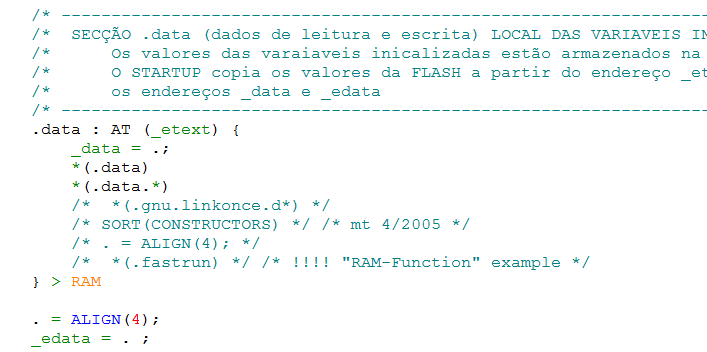
SECTIONS .text: Memória de programa (FLASH). Aqui orientamos ao compilador a colocar os vetores de exceções nos primeiros bytes da memória de programa, seguido do código startup do processador, e depois colocar o resto do programa.



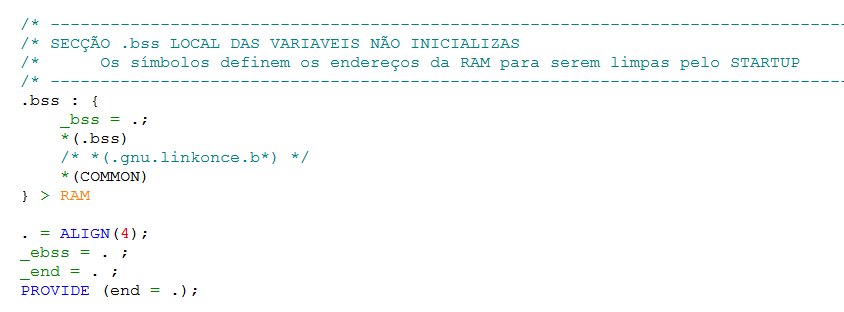
SECTIONS .rodata: Memória de programa. Aqui orientamos ao compilador a colocar as constantes e os valores das variáveis inicializadas no final da FLASH.



SECTIONS .data: Memória de dados. Aqui orientamos ao compilador que as variáveis inicializadas vão ser alocadas no início da SRAM:



SECTIONS .bss: Memória de dados. Aqui orientamos ao compilador que as variáveis não inicializadas vão ser alocadas na SRAM, logo após .data :

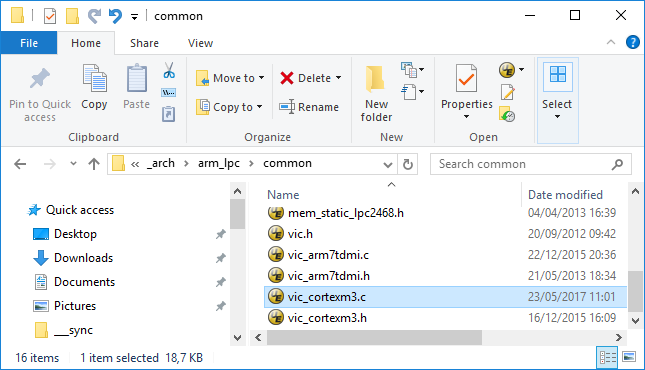


**Vetores de Interrupções**

Os vetores de interrupções do ARM lpc17xx ocupam no máximo 1024 bytes e por padrão se localizam entre os endereços 0 a 0x400 da FLASH. Digo o máximo, porque isto depende da quantidade de vetores que seu sistema vai manipular. Os 16 primeiros vetores pertencem às interrupções de sistemas e são obrigatórios e devem ser implementados, os demais vetores pertencem aos periféricos, e somente serão implementados o que forem utilizados.

Os vetores podem ser remanejados em qualquer posição de memória seja FLASH ou SRAM utilizando o registrador VTOR. Este é um recurso muito importante quando se trabalha com bootloaders, pois podemos usar os vetores de interrupções do bootloader em uma região de memória diferente da aplicação que será carregada.

No Kit ARM Ravina os tratamentos das interrupções são feitos pela biblioteca “vic\_cortexm3.c”:



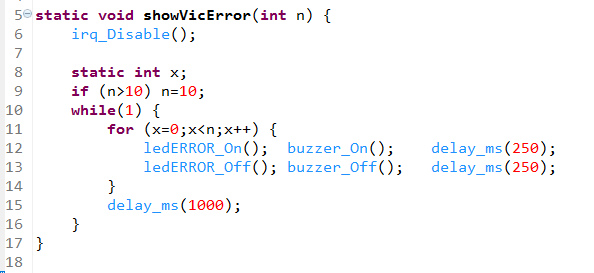
Em “vic\_cortexm3.c” tratamos todas as interrupções do sistema. Colocamos um recurso de segurança caso alguma interrupção inesperada ocorrer e uma função padrão pode ser chamada notificando o erro. Esta interrupção inesperada pode ocorrer quando:

* Acesso a um hardware não ligado ou com o clock desligado;
* Acesso a hardware inexistente;
* Acesso a memória inexistente;
* Ocorreu uma interrupção de algum hardware cujo não foi implementado a sua rotina de tratamento.

Usando essa função padrão fica mais fácil identificar e solucionar o problema. A função padrão é escrita em qualquer parte do teu programa e anexar via função “nvic\_SetIntHandler”. No exemplo abaixo mostra logo no início do programa anexando uma função padrão:

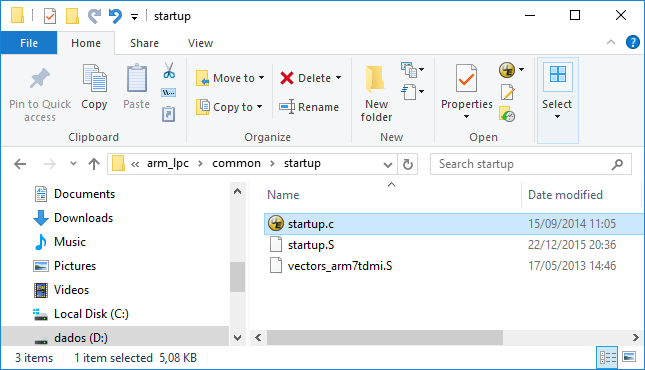


A função padrão usada nos exemplos tem como função de emitir um aviso sonoro e piscadas de um LED. De acordo com a quantidade de piscadas determina o tipo de erro de interrupção:



**Startup**

Todos os processadores precisam ser inicializados antes de executar o programa principal. Dependendo da arquitetura computacional há bootloaders para fazer essa função, em nosso projeto usamos um arquivo chamado “startup.c”.



O mínimo que um startup deve oferecer:

* Inicializar o vetor de exceções, atribuindo os endereços das funções de tratamento para cada exceção. Importante mencionar que sempre os primeiros bytes da memória de programa são reservados para esse vetor;
* Configurar os STACKS para cada modo de operação do ARM. São sete modos de operação do ARM: modo supervisão; modo usuário, modos de interrupções e outros;
* Carregar os valores das variáveis inicializadas da secção “*.rodata*” para suas respectivas posições na secção “*.data*”. Esse procedimento lê os valores das variáveis na memória de programa e atribui nas suas respectivas varáveis na memória RAM;
* Limpar a área das memórias não inicializadas na secção “*.bss*”. Esse processo coloca valor 0 para todas;
* Configurar os clocks da CPU e periféricos;
* Inicializar memórias como SDRAM;
* Buscar o programa principal (main) para execução.

**Makefile**

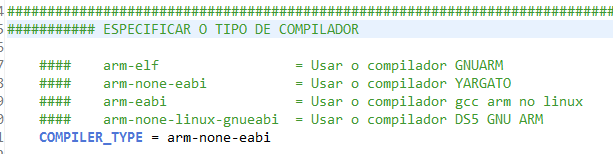
O arquivo makefile é que orquestra toda a compilação, ele é interpretado pelo utilitário do compilador chamado make. É no makefile que contém todas as instruções de como o compilador deve compilar todos os arquivos fontes de programa e em que circunstâncias. O make controla todas as dependências dos arquivos fontes através de um conjunto de regras armazenado em script makefile.

O uso do make com makefile é muito útil em trabalhar com grande número de arquivos, no qual cada arquivo tem seu próprio modo de configuração e compilação. O make compila somente os arquivos que foram modificados depois da última compilação, evitando a compilação de arquivos fontes de forma desnecessária, sem assim não fosse, tornando dispendioso durante a compilação de programas que utiliza grandes número de arquivos.

Em nosso makefile contém regras explícitas, regras implícitas, macros e blocos de comandos, vamos descingir os principais pontos para que o programador possa alterar quando necessário, os demais conteúdos estão devidamente comentados.

**Apontar o Compilador**

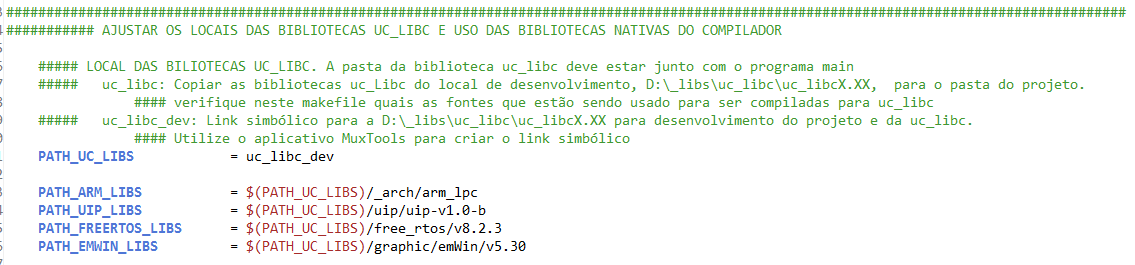
COMPILER\_TYPE: Aponta qual compilador que será usado para compilação dos arquivos fontes do projeto. Todos os compiladores adotam uma sintaxe de usar o prefixo sempre como mesmo nome em qualquer ferramenta do compilador, em nosso caso estamos usando o compilador YAGARTO, e este inicia com arm-none-eabi, logo teremos: arm-none-eabi-gcc.exe, arm-none-eabi-gdb.exe, arm-none-eabi-c++.exe e assim por diante.



Dessa forma podemos usar e testar outros compiladores no mesmo projeto.

**Apontadores da Biblioteca uc\_libc**

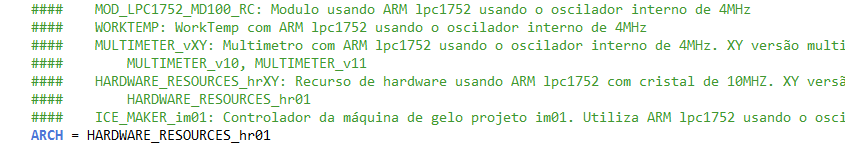
PATH\_UC\_LIBS: Aponta em que diretório a uc\_libc se encontra, por padrão usamos o diretórui uc\_libc\_dev. O uc\_libc\_dev pode ser um link simbólico para o verdadeiro local da uc\_libc, ou tem realmente os conteúdos da biblioteca, isto é indiferente para as ferramentas de desenvolvimento. Junto com PATH\_UC\_LIBS há derivações de apontamentos de outros sub conjuntos de bibliotecas que completam a uc\_libc como freeRTOS, pilha TCP/IP e gerenciadores de janelas gráficas:



Essa sub divisões podemos usar diferentes versões destes sub sistemas que possam estar contidas no uc\_libc.

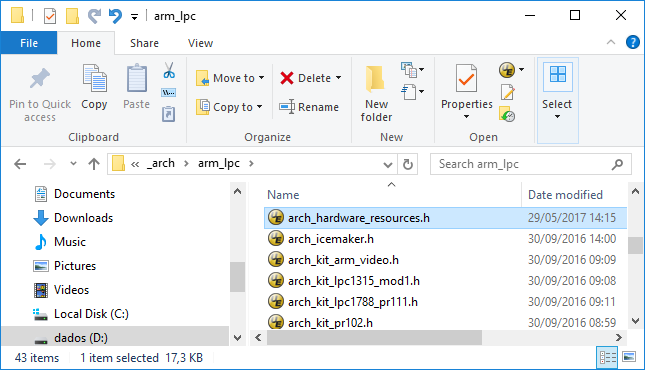
**Arquitetura Computacional**

ARCH: Instrui ao compilador qual arquitetura computacional a ser usado. Instrui qual processador e que tipo de núcleo que vai ser usado. No kit ARM Ravina estamos usando a arquitetura HARDWARE\_RESOURCES modelo hr01 que utiliza o micro controlador lpc1752 com núcleo Cortex-M3.



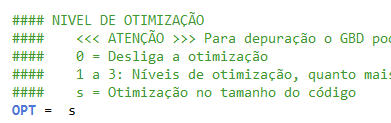


Para criar novas arquiteturas é preciso defini-las em “uc\_libc\\_arch\arm\_lpc”. Nas arquiteturas estão definidos todos os quesitos de compilação para a plataforma alvo de compilação:



**Nível de Otimização**

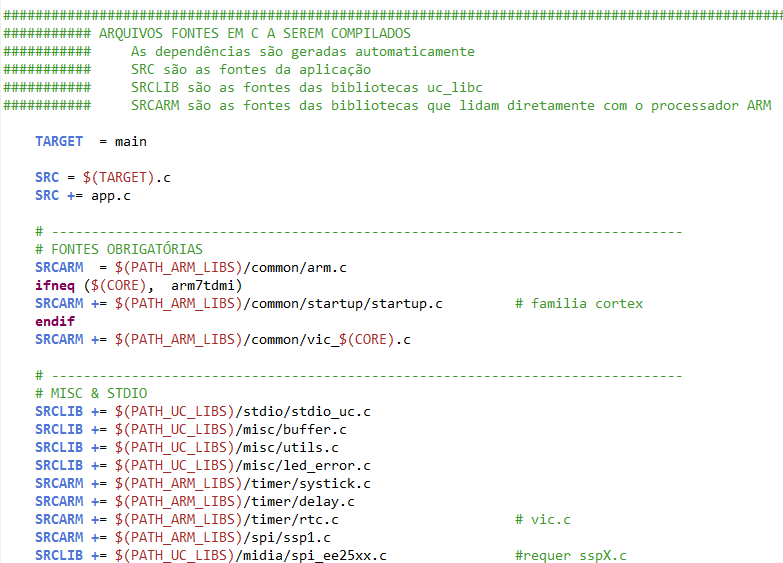
OPT: Com essa opção instruímos o compilador fazer ou não a otimização do código. Quando optamos para fazer a otimização, é sinalizado qual será o seu nível. Como estamos trabalhando com micro controladores, e estes têm recursos limitados de memórias, sempre usamos a opção de otimização de redução do código, que é a opção s.



**Arquivos Fontes para Compilação**

Há variáveis que determinam quais arquivos fontes do projeto, cada macro corresponde a um grupo de arquivos fontes, são eles:

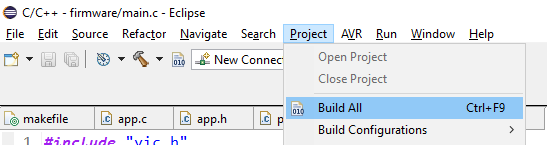
* TARGET: Nome do arquivo fonte principal da aplicação, e por consequência o binário terá o mesmo nome;
* SRC: São fontes de alto nível, ou seja, as fontes a aplicação de fato;
* SRCLIB: São fontes que corresponde ao uc\_libc que não estão vinculadas a nenhum tipo de processador;
* SRCARM: São fontes que corresponde ao uc\_libc que são ligadas diretamente ao processador ARM.



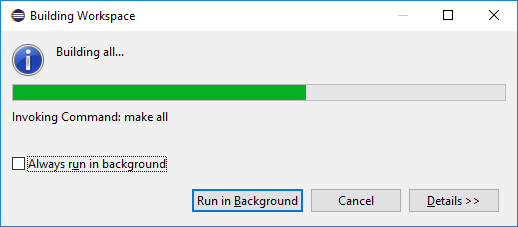
São nestas variáveis que o programador vai inserir ou retirar códigos para o projeto, e se usar sempre essa mesma arquitetura computacional é somente nessa secção do makefile que será mexido no projeto.

**Rodando o Primeiro Exemplo**

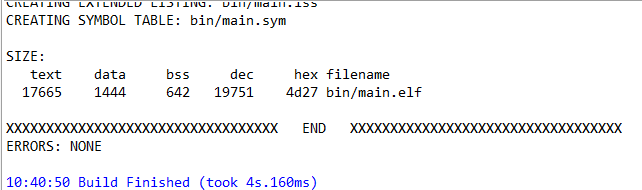
Com todas as ferramentas devidamente instaladas e configuradas, e com os arquivos fontes nos seus devidos lugares, basta instruir o Eclipse a compilar o projeto para criar o programa executável. Isto é muito simples de se fazer no Eclipse, simplesmente clique no menu Project >> Build ALL ou tecle CTRL+F9:

****

Espere pela compilação:



No console mostra o status de compilação, o importante notar que não houver erro algum durante a compilação, sinalizando que está tudo bem para gravar e executar no ARM:



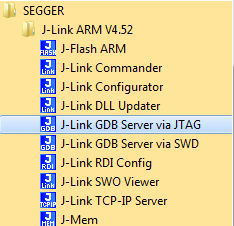
Informações importante de compilação são:

* text: Tamanho em bytes do código final de programa. Aqui são os bytes que serão gravados na FLASH, são bytes de programas, constantes e valores iniciais das variáveis. Caso esse valor passar de 64KB, limite desse modelo de ARM, um aviso será emitido do compilador, alertando que o tamanho do código estourou a capacidade da FLASH. Neste caso retire códigos não utilizados, ou diminuía as strings de mensagens, ou diminua a quantidade de variáveis inicializadas;
* data: Tamanho em bytes da memória RAM usadas pelas variáveis inicializadas;
* bss: Tamanho em bytes da memória RAM usadas pelas variáveis não inicializadas;
* dec: Total em bytes incluindo text, data e bss.

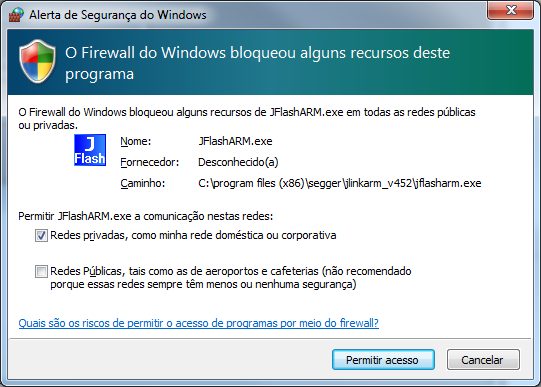
**Rodando no ARM**

Com o firmware compilado e livre de erros, o mesmo está pronto para ser gravado e depurado no ARM. Neste documento descreveremos como usar o emulador J-link para gravar e depurar o firmware no ARM, contudo, se deseja somente gravar o firmware consulte o documento “gravando via porta serial UART.docx”.

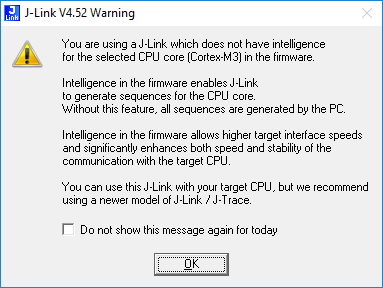
Certifique se o J-link está conectado entre o PC e a porta JTAG do ARM, e se o Kit esteja ligado. Rode o servidor GDB (daemon) via JTAG da Segger para que o GDB do compilador possa se comunicar com emulador:



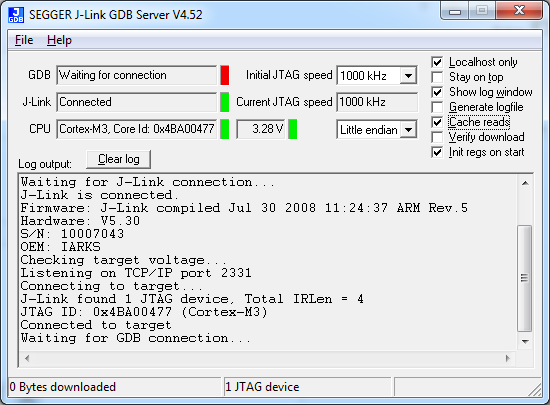
Quando o GDB rodar pela primeira vez, é preciso liberar a porta TCP que ele usa no Firewall do Windows:



Depende da versão do emualdor J-Link uma janela de aviso pode ser emitida, o aviso diz que falta alguns recursos para usar o emulador, neste caso não se preocupe, pois não estamos usando, clique OK para ignorar o aviso e continue com o processo:



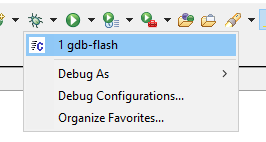
Na janela do GDB halite cache read, Show log Windows e init regs on start. Stay ontop é opcional. A opção Show log Windows ajuda se algo errado acontecer com o Eclipse. A opção cache read habilita a leitura de memória antecipada acelerando a depuração, caso venha a ter problemas com ela desabilite-a:



Status do Servidor:

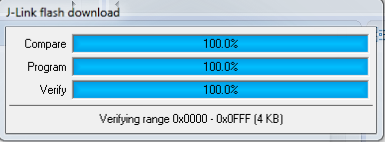
* GBD wait for connection: Esperando pela conexão via TCP/IP do depurador do compilador. Em nosso caso o Eclipse, uma vez que o Eclipse abra um socket com depurador esse indicador passa a ser verde;
* J-LINK Connect. Sinaliza que a interface JTAG está conectado ao ARM.
* CPU: Reconhecimento do lpc1752 Cortex M3.

Agora o daemon está esperando pelos comandos do Eclipse. Como o GDB server rodando em segundo plano, podemos iniciar a gravação e depuração. Clique no ícone de depuração e selecione o gdb-flash ou tecle F11:

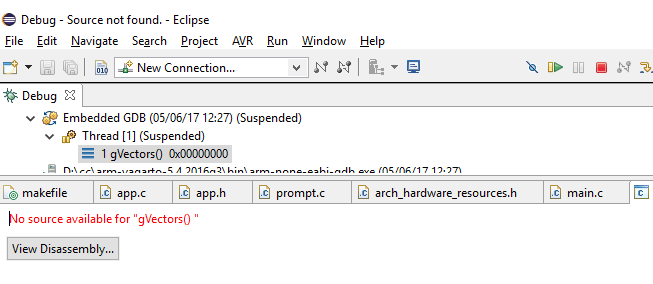


No documento “eclipse 07 - configurando o emulador j-link.docx” explica como foi feita essas configuração e depurações.

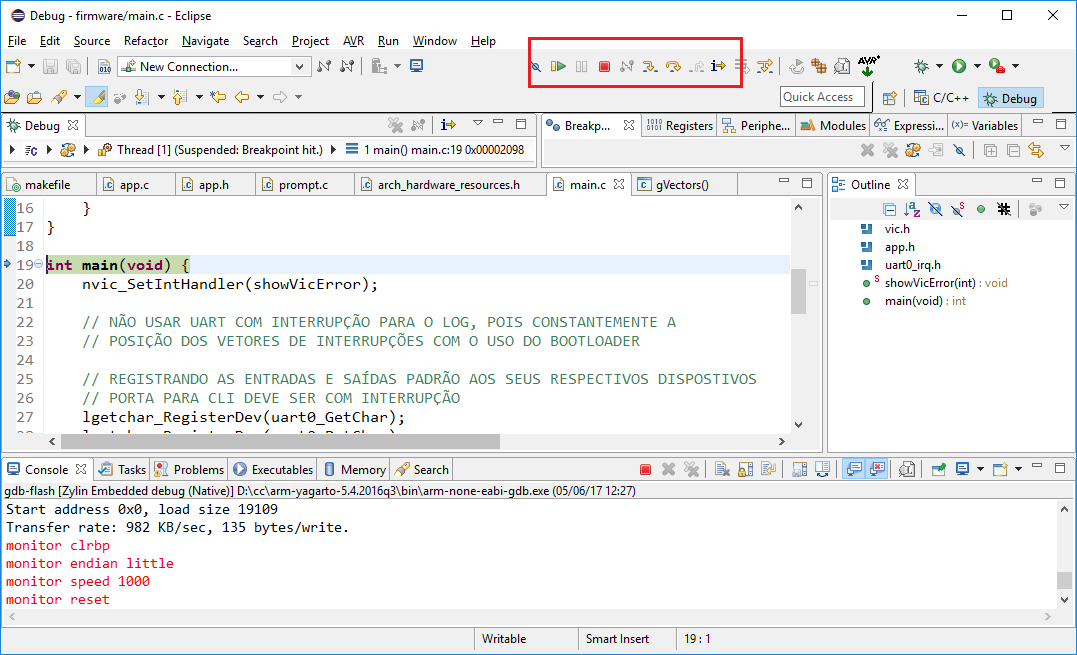
Espere pela gravação e comparação serem feitas:



Após a gravação e início da depuração, o Eclipse vai apontar para os vetores de interrupções, porém, isto não foi orientado o Eclipse buscar esse código, fica inviável fazer esse procedimento, então ele emite um aviso que não foi encontrado tais fontes:



Vamos ignorar este aviso e continue com execução clicando no ícone de RUN, ou teclando F9. A depuração vai parara no primeiro break point que é no main, foi configurado para isto. Na imagem abaixo mostra o programa parado no main e os comandos de depuração como RUN, rodar passo a passo e entre outros:

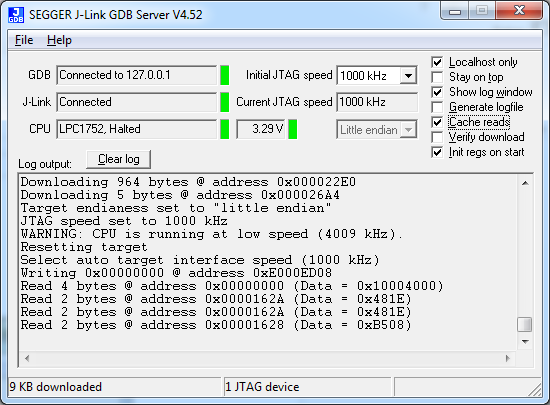


Clique no ícone de RUN novamente, ou tecle F9, para rodar de fato todo o programa sem novas paradas. Neste primeiro exemplo o led tick do kit está piscando a cada 1 segundo sinalizando que o processador foi inicializado com sucesso e rodando as demais funções.

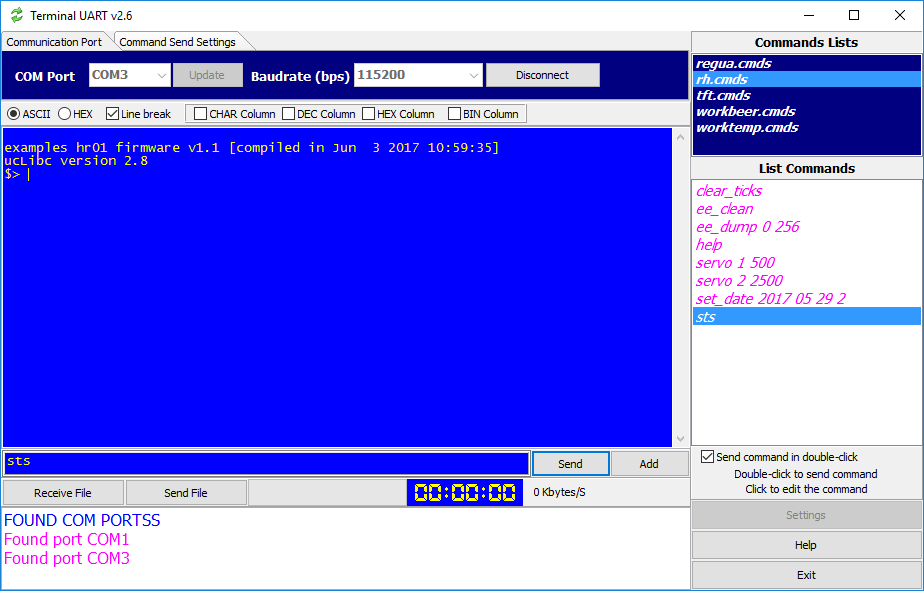
Sempre que se inicia uma depuração a perspectiva do Eclipse muda de edição para depuração, isto para exibir recursos de depuração que não são interessantes durante a programação. Para evitar de usar ícones de troca de perspectiva, utilize a tecla CRTL+F8.

**Cuidados:** Antes de finalizar a depuração, sempre clique no ícone de stop. Caso haver algum tipo de problema para iniciar uma nova depuração, desconecte o J-Link GDB Server, deligue o kit e feche o Eclipse, depois volte a ligar tudo. Em alguns casos é preciso apagar a FLASH através do programa FlahsMagic.

Na imagem abaixo mostra o JLINK se comunicando com o ARM e Eclipse via GDB do compilador:

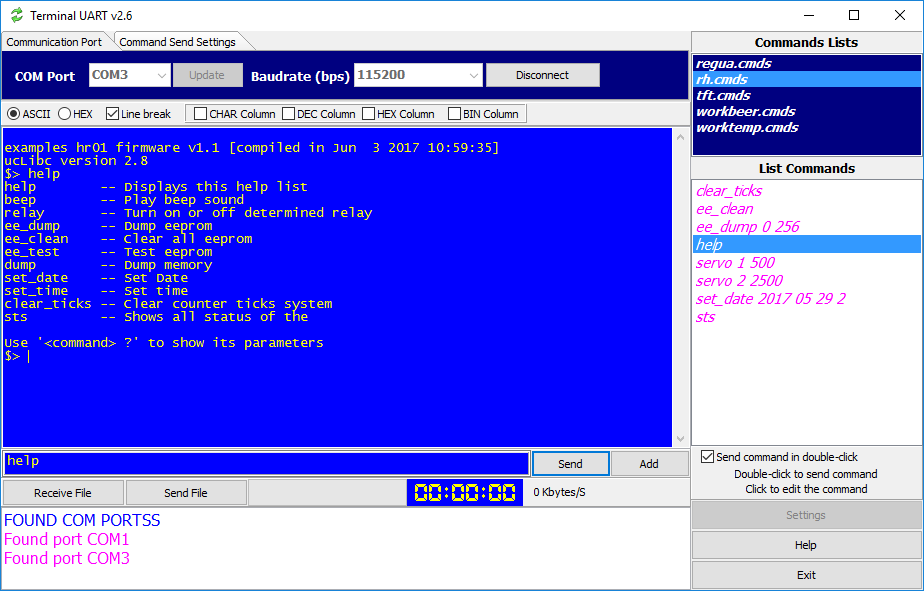


Conecte o cabo serial UART ao Kit ARM e abra o terminal UART. Selecione a porta COM na qual está conectado ao kit, coloque o baudrate a 115200bps e clique em conectar. Um prompt emitido pelo ARM está esperando por comandos:

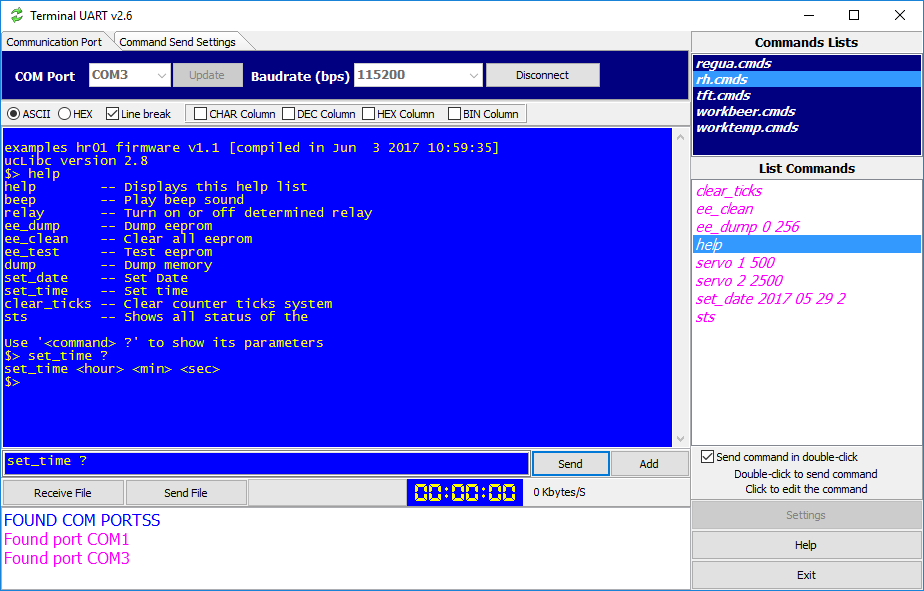


**CLI (Command Line Interpreter)**

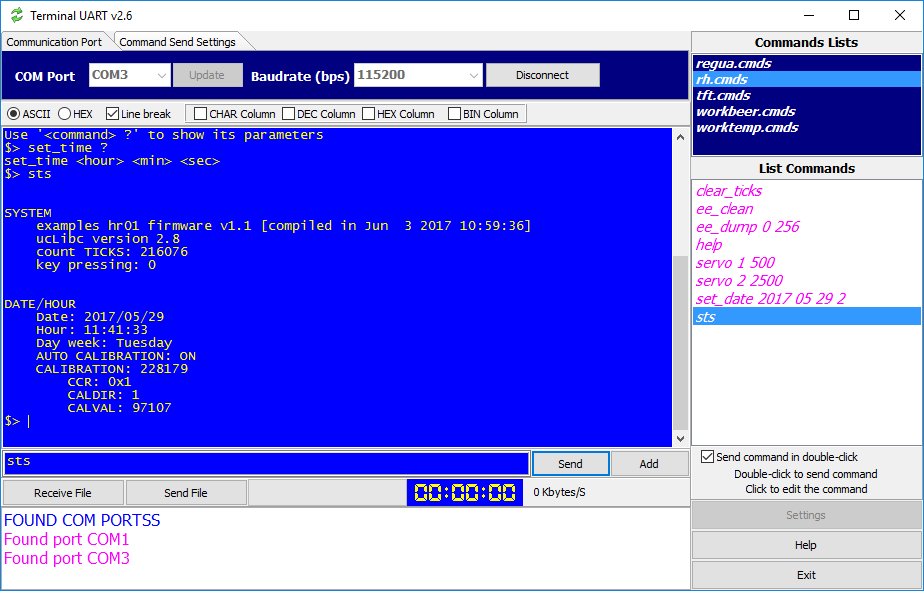
Junto com uc\_libc acompanha um interpretador de comandos via prompt, este se chama CLI (Command Line Interpreter). Em todos os exemplos e aplicações é utilizado essa CLI para auxiliar o desenvolvimento ou uma interface básica para os usuários. Os comandos estão armazenados arquivo fonte “prompt.c”. Para ver os comandos disponíveis pela aplicação digite help no prompt:



Para saber a sintaxe de cada comando digite o nome do comando seguido de ?. Exemplo: set\_time ?:

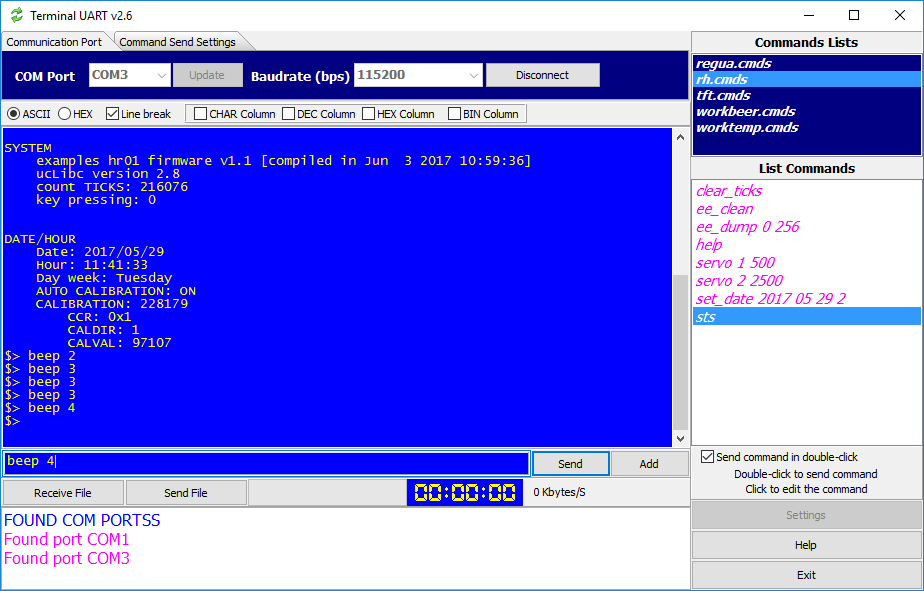


Em todos os exemplos e aplicações existe o comando “sts” de status, este diz o status atual do sistema:



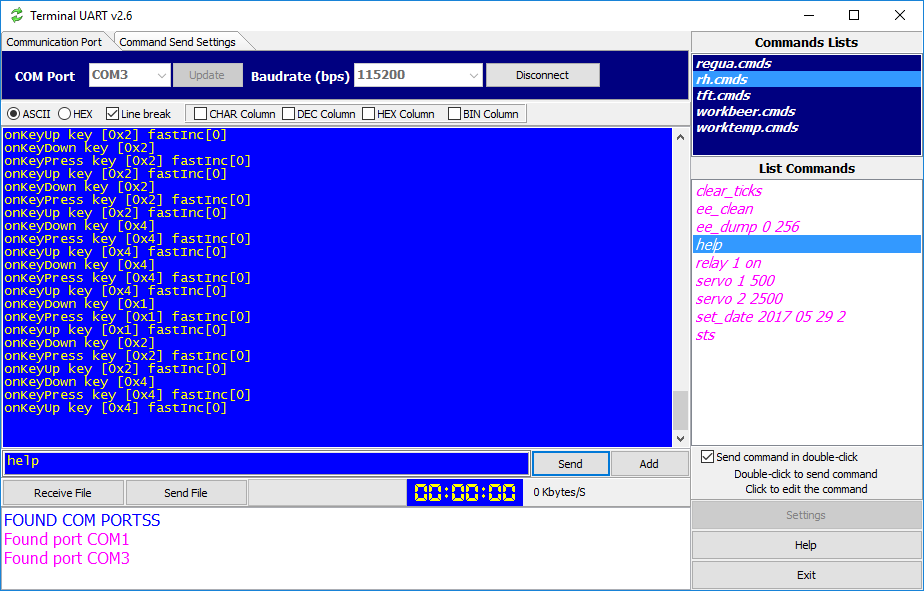
**Teste do beep**

No prompt de comando digite comando beep com a opção do tipo de som a emitir:

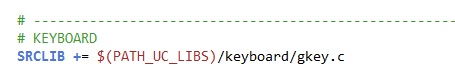


**Testando Teclado**

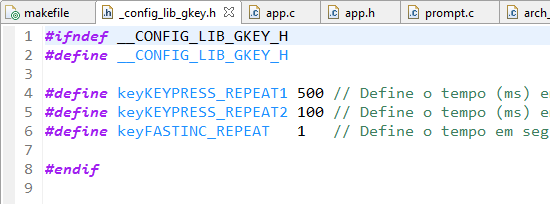
Pressione cada chave do kit ARM e uma mensagem será exibida no terminal junto com um som do buzzer distinto:



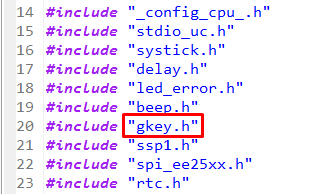
O uso deste recurso de teclado fornecido pela uc\_libc é a biblioteca gkey.c. Esta biblioteca foi adicionada na secção de fontes no makefile:



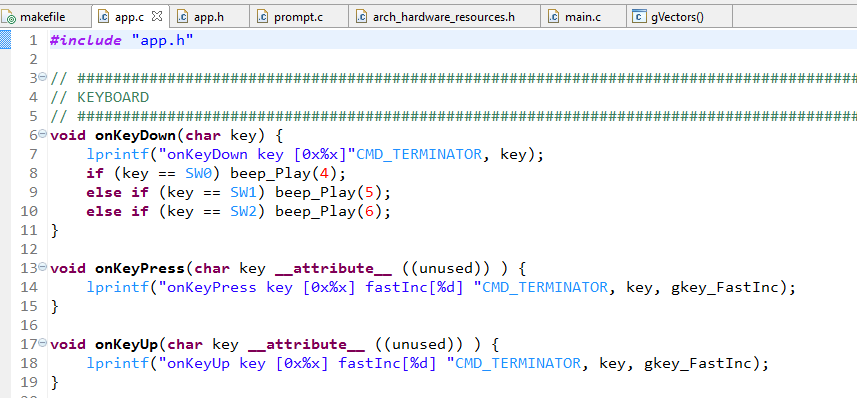
Em \_config\_lib\_gkey.h configuramos o comportamento do gkey:



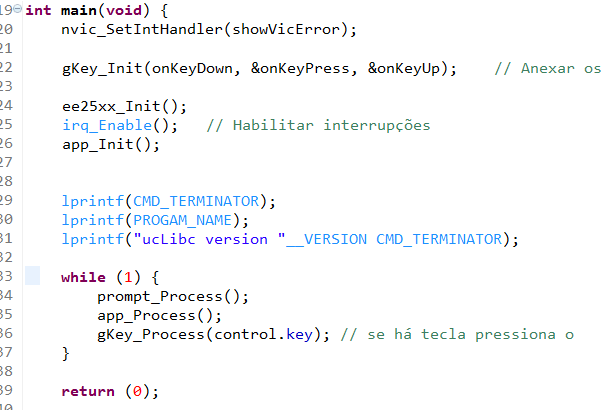
Incluímos o recurso do gkey no app.h:



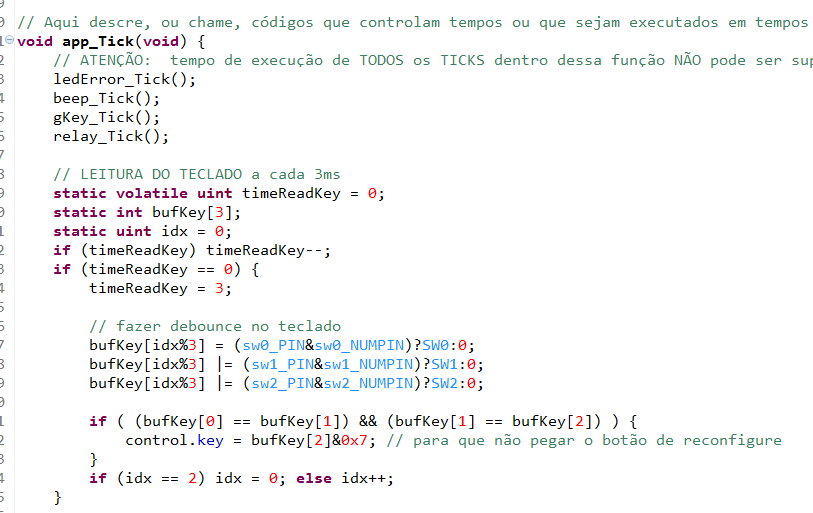
Dentro do main iniciamos o gkey através do gKey\_Init e passando por parâmetros as funções que vão responder aos eventos de teclado, como evento de teclar; evento de mantendo pressionado a tecla; e o evento quando a tecla é solta. As respectivas funções estão no app.c com os nomes: onKeyDown, onKeyPress e onKeyUp:



Ainda no main iniciamos todos os recursos de aplicação via app\_Init, como iniciar relés, leds e outras funções. Após as inicializações o programa main entra em um laço infinito chamando todos os recursos da aplicação e das bibliotecas, dentro desse main sempre denominados de processos, e nesses processos que o sistema analisa se uma tecla fora pressionada e solta através do gKey\_Process:



A leitura da tecla é feita a cada exatos 3ms usando o método debonce. Para isto usamos um timer que gera uma interrupção a cada 1ms, isto é feito através da função systime\_SetIntHandler no início do programa, ali passamos a função que será chamada a cada 1ms. No app\_tick não só controlamos a leitura do teclado, mas sim de todos os códigos que requerem ser chamados nos tempos exatos, assim esses códigos sempre chamamos de ticks, e para controle dos eventos do teclado chamamos a função gKey\_Tick:



**Debouce**: O método debouce são leituras consecutivas do dispositivo, e somente assumimos o valor se todas as leituras forem iguais. No caso do teclado é feito três leituras consecutivas, se as três forem iguais assumimos o valor, senão descartaremos. Isto é útil porque se há ruídos elétricos, seja por captação pelos fios, trilhas ou no momento de teclarmos devido ao contato da chave, a velocidade de ruído é muito maior que os 9ms, e caso o ruído venha alterar o nível lógico dentro desta janela de tempo, a leitura é ignorada.

**Conceitos de Processos e Ticks de Sistemas**

Aqui desreveremos os conceitos que diferenciam entre ticks de sistemas e processos em nossa forma de trabalho.

Ticks de sistemas: São códigos que precisam serem executados em tempos reguladores e precisos. Estes tipos de códigos denominamos de funções ticks, e todos as funções dessa categoria colocamos tick() no final do seu nome. Geralmente controlamos os ticks através de timer que gera uma interrupção a cada 1ms que chama todos os ticks do sistema. O tempo de chamada dos ticks não necessariamente deve ser a 1ms, este foi o valor ideal encontrado para esse tipo de ARM rodando a 100Mhz, o tempo de ticks depende da velocidade da CPU e quantidade de recursos que vão ser processados.

Processos: São códigos que não precisam ser executados em tempo regulares e não fazem tratamento de interrupções. São códigos que precisam de respostas de ticks de sistemas, dados recebidos ou enviados por uma interrupção. Estes códigos chamamos de funções processos e colocamos process no final do seu nome e são executados, ou ficam, no loop principal do programa.

Com esse tipo de estrutura podemos fazer muita coisa em paralelo sem o uso de um sistema operacional. Mas para que isso funcione é preciso que cada função processo não fique parado na espera de algum evento, é preciso trabalhar em máquinas de estados. E as funções ticks devem ser a mais curtas possíveis sendo executadas muito rápidas, deixando um bom intervalo de tempo entre as chamadas dos ticks de sistema para executar as interrupções e funções processos do sistema.

**stdio**

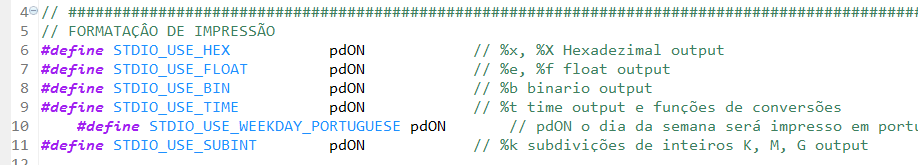
A biblioteca stdio permite que o usuário interage com firmware enviando comandos via CLI, ou recebendo informação de depuração ou erros durante a execução do sistema. As funções padrões de entrada e saída que o stdio possa oferecer são:

* lgetchar: Função padrão de entrada de dado, caractere, pelo usuário;
* lputchar: Função padrão de saída de dado, caractere, para o usuário;
* lprintf: Saída de mensagem formatada para o usuário, funciona semelhante ao padrão printf de uso em C/C++, porém modificada para sistemas embarcados ocupando menos espaço de programa possível;
* rprintf: Funciona semelhante ao lprintf, porém podemos descriminar qual o veículo de saída de dados;
* plog: Saída de mensagem formatada de depuração para o usuário. Logo no início da mensagem há uma impressão de identificador de LOG. Isto é útil para descriminar o que é LOG ou que é informações da aplicação quando usamos o mesmo veículo de saída de dados;
* plognp: Funciona semelhante ao plog, porém não imprime o identificador de LOG;
* perr: Saída de mensagem formatada para o usuário, utiliza a saída exclusivamente para saída de erros. Em nossas aplicações não estamos usados essa função de saída.

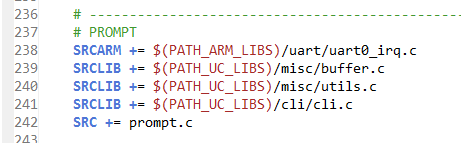
Para usar o stdio no kit ARM é preciso adicionar o código fonte stdio\_uc.c no makefile:



No arquivo de configuração \_config\_lib\_stdio.h determinamos quais os recursos stdio que serão adicionados na compilação do programa, quanto menos recursos usados, menor será o tamanho do código final do firmware:

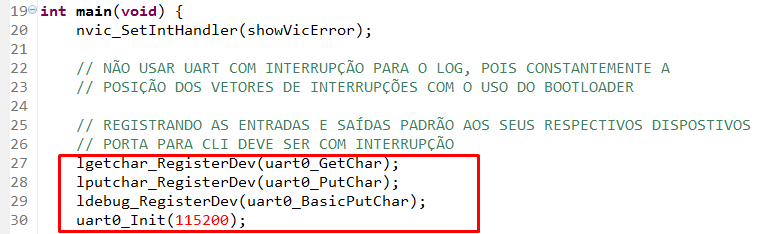


A biblioteca stdio somente fornece meios de formatação e gerenciamentos para os veículos de entrada e saída de dados, então é preciso anexar esses veículos ao stdio. Neste exemplo usamos a porta UART0 como nosso veículo, a porta UART é o meio de comunicação mais comum em sistemas embarcados, inclusive Linux Embedded, então habitua-se a usar ela. No makefile vamos adicionar os demais códigos de programa para ter uma stdio completamente funcional:



* uart0\_irq.c: Porta UART0 usada como veículo de entrada e saída de dados para stdio;
* buffer.c: Buffers de dados usados pela UART e para outros recursos de firmware;
* utils.c: Funções uteis para as bibliotecas ou código de aplicação;
* cli.c: Gerenciador do prompt de comandos;
* prompt.c: Comandos do prompt da aplicação.

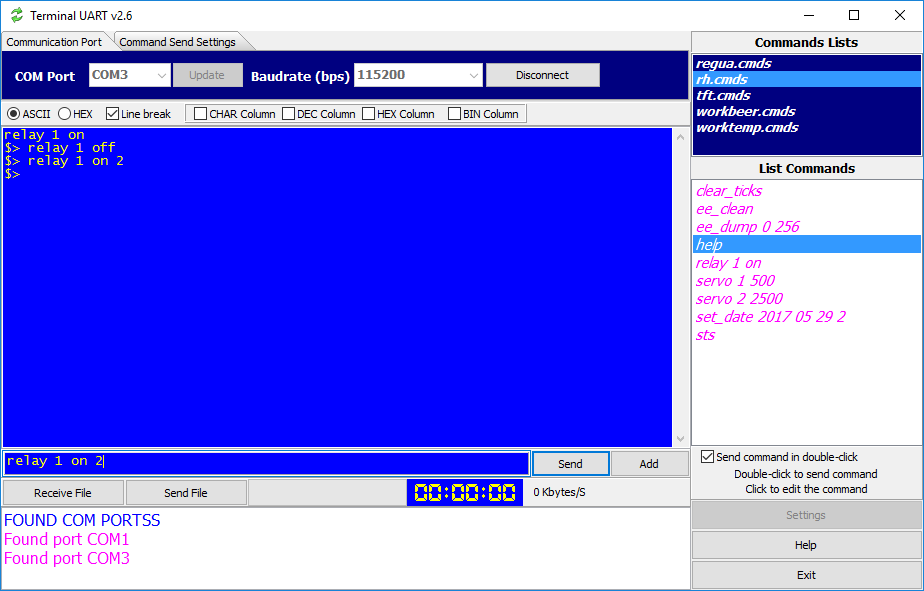
Com o veículo para stdio determinado, agora é preciso iniciar o veículo e anexar as funções de entrada de saída ao stdio. Isto é feito logo no início do firmware dentro da função principal main:



* lgetchar\_RegisterDev: Anexa a função de entrada de dados da UART0 para a stdio. Isto orienta stdio qual o veículo de entrada de dados para a função lgetchar;
* lputchar\_RegisterDev: Anexa a função da saída de dados da UART0 para a stdio. Isto orienta stdio qual o veículo de saída de dados para as funções lputchar e lprintf;
* ldebug\_RegisterDev: Anexa a função da saída de dados da UART0 para a stdio. Isto orienta stdio qual o veículo de saída de dados para as funções plog e plognp. Observe que usamos uma função básica de saída de dados da UART, esta função básica não utiliza interrupção para envio de dados, sendo ideal fazer LOGs logo no início do firmware, onde as interrupções ainda não foram inicializadas, ou usar em outras circunstâncias que é preciso desligar as interrupções;
* uart0\_Init(115200): Inicializa a porta UART0 com velocidade e comunicação de 115200bps.

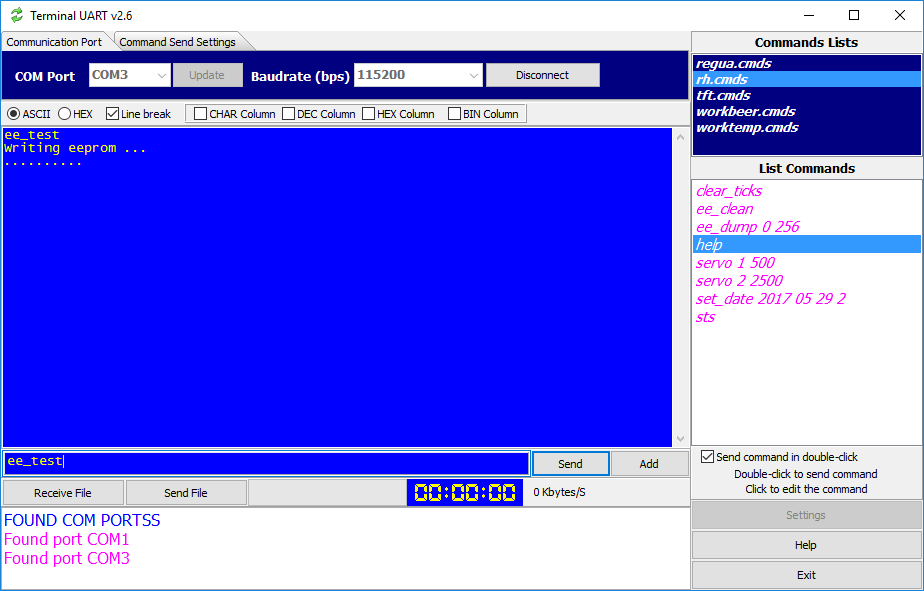
**Acionar Relés**

Através do comando CLI relay podemos ligar ou desligar um relé. O último parâmetro do comando é opcional, e quando inserido determina o tempo em segundos que o relé ficará no novo estado antes de voltar no estado anterior:

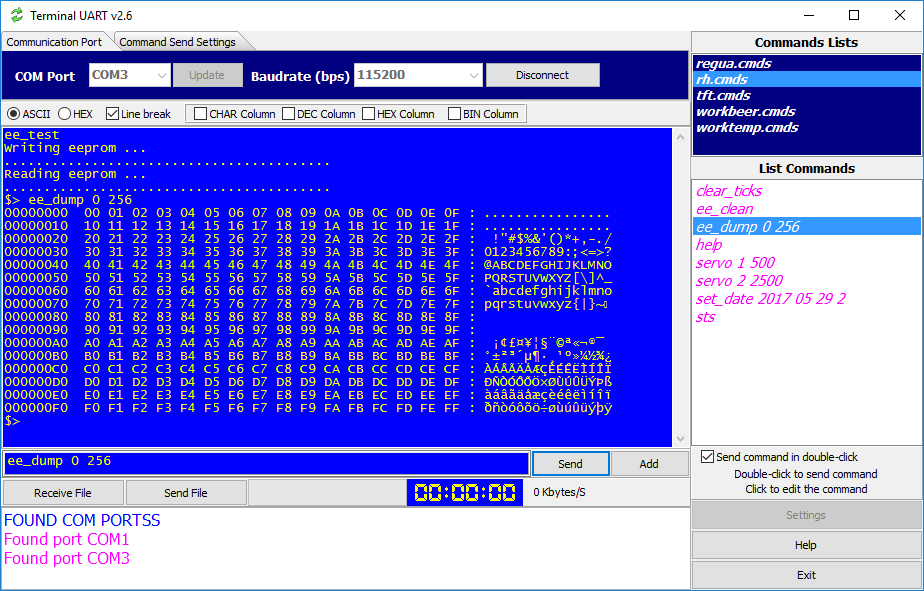


**Teste da EEprom**

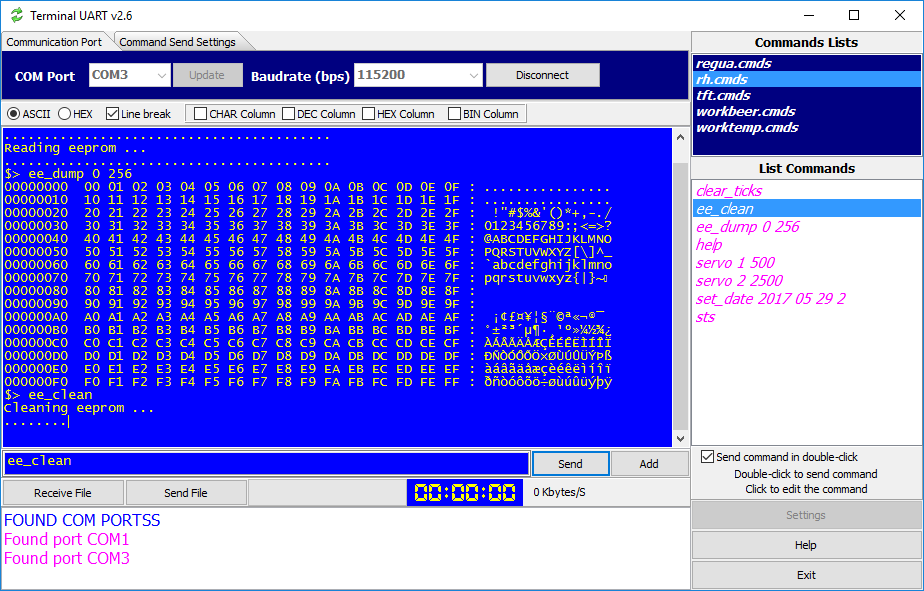
O Kit ARM contém uma EEprom SPI de 8KBytes, podemos fazer um teste de acesso e sua integridade usando o comando ee\_test:



Podemos ver o conteúdo dentro dela usando o comando ee\_dump:



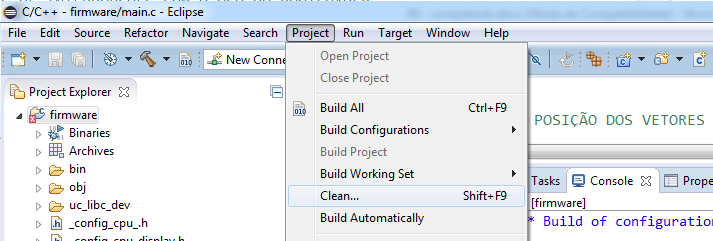
Ou apagar a memória completamente com o comando ee\_clean:



**Limpando Compilação**

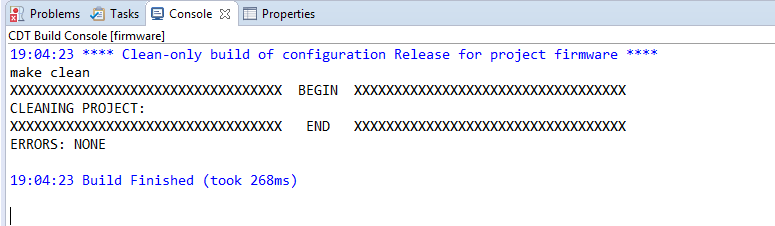
Entre muitos motivos para necessidade de uma limpeza de compilação, o motivo mais forte é garantir que todas as fontes de programa sejam compiladas para um determinado tipo de processador, por alguma razão foi trocado o tipo de processador no makefile, e o make não vai recompilar as fontes por não haver mudanças nelas, isto é umas das particularidades do make, então as fontes não alteradas ainda contêm os arquivos compilados para o antigo processador, e devemos excluir esses arquivos compilados para que as fontes sejam recompiladas para o novo tipo de processador.

Caso queira limpar uma compilação anterior clique no menu “Project >> Clean” ou “SHIFT+F9”:



Escolha a opção de limpar todos os projetos, e opcionalmente habilite a opção de recompilar novamente o projeto após limpeza:





O projeto está limpo para futuras compilações.

**Programa Base**

Quaisquer um dos exemplos fornecidos podem se tornar o projeto de ponto de partida para novos projetos, pois todos compartilham da mesma arquitetura organizacional, bastando retirar e adicionar os módulos desejados. Para isto faça uma cópia do projeto Eclipse que se aproxima da nova necessidade, e renomeie o projeto. Assim foi feito com todos os projetos exemplos tendo como base o “components basics”:

