**CAPL Básico**

1. **Introdução**

CAPL é uma sigla para (Communication Access Programming Language), e é uma linguagem orientada a eventos desenvolvida pela empresa Vector Informatik GmbH. Essa linguagem é utilizada nos softwares CANoe e CANalyzer para o desenvolvimento dos blocos de programação e os blocos simulados.

1. **O que seria uma linguagem orientada a eventos?**

Como dito anteriormente, CAPL é uma linguagem orientada a eventos, porém o que seria isso? Diferentemente de C, que é uma linguagem estruturada, ou seja, que segue uma sequência. A linguagem CAPL é baseada em “reações” a eventos preestabelecidos, ou seja “quando acontecer isso... Faça isso”.

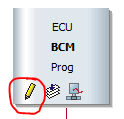
Por esse motivo, todo código o CAPL será desenvolvido em diferentes blocos, que seriam funções chamadas através de gatilhos (*trigger’s*), esses gatilhos por sua vez, podem ser através de Temporizadores, Eventos I/O e Eventos na rede CAN. Isso será abordado posteriormente.

1. **Semelhanças com C**

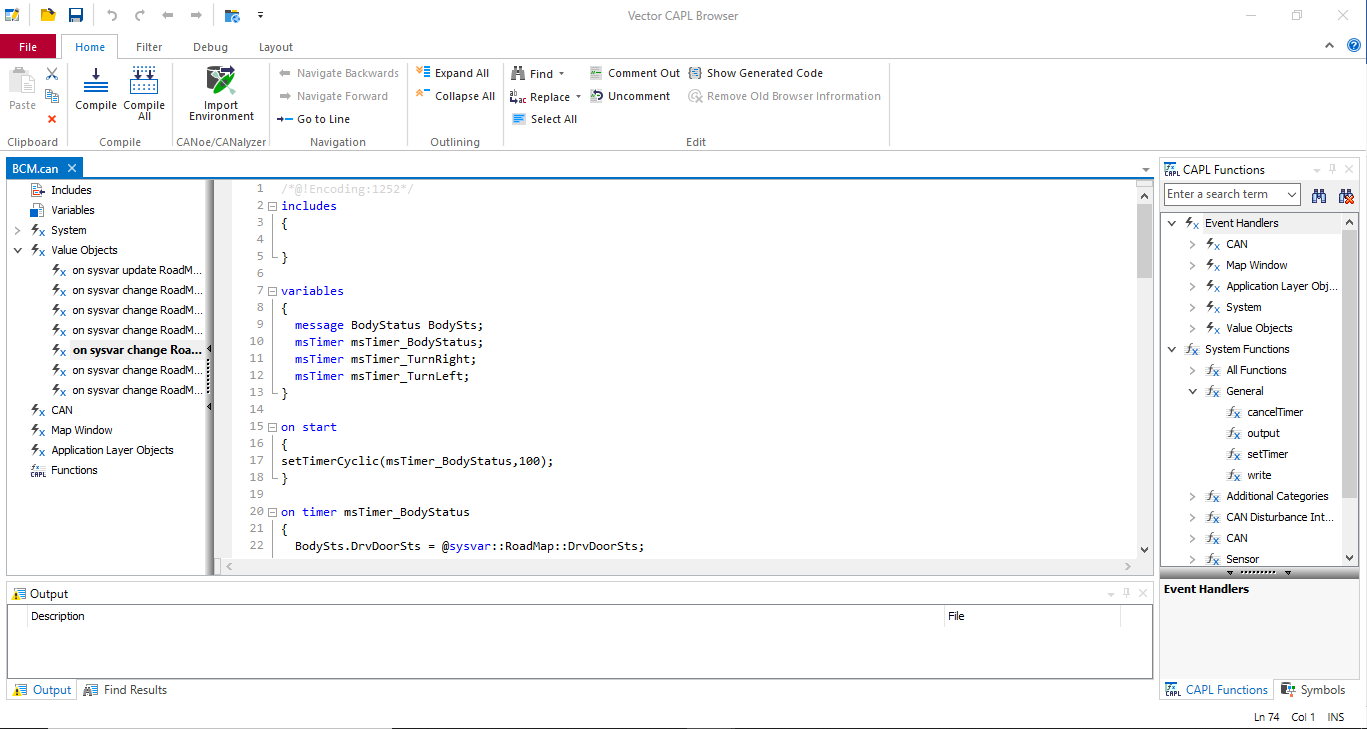
A semântica de CAPL é bastante parecida com C, portanto durante o desenvolvimento é possível aplicar funções **if**, **if else**, **for**, **switch**, etc.

1. **CAPL Browser**

O CANoe possui sua própia IDE, o ‘*CAPL Browser’*, esta é uma ferramenta muito útil, pois ajuda muito durante o desenvolvimento do código, como por exemplo, enquanto o código está sendo escrito a própia IDE da sugestões baseadas no DBC (Banco de Dados), que está sendo utilizado na simulação.  
Por esse motivo, é sempre importante abrir o *CAPL Browser* através do CANoe, para que o mesmo associe o DBC presente na simulação. Para isso, basta entrar na parte de *Simulation* do CANoe e clicar no “lápis” presente no bloco da simulação.



Após clicar no lápis, o *CAPL Browser* abrirá com o DBC já associado, conforme imagem abaixo:



O *CAPL Browser* também facilita o desenvolvimento do código através da barra presente no canto esquerdo, com ela é possível selecionar os ‘**Eventos**’, ‘**Sinais** e ‘**Variáveis de Sistema**’, e com o mouse, apenas arrastar para a área de código.

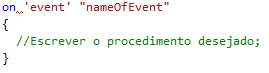
|  |
| --- |
|  |

1. **Criando um evento**

Como dito anteriormente, o CAPL é uma linguagem orientada a eventos. Nessa seção abordaremos os principais eventos utilizados na construção de uma simulação, citados abaixo:

* **on message**
* **on timer**
* **on sysvar**
* **on key**
* **on preStart**
* **on start**
* **on stopMeasurement**
* **on errorFrame**

A estrutura básica de um evento é a seguinte:



É importante não se esquecer de fechar as chaves, e também de finalizar cada linha de código com ponto e vírgula, para evitar erros de compilação.

Abordaremos agora cada um dos eventos citados, quando e como eles são utilizados no código. Inicialmente iremos focar apenas nos eventos, e posteriormente abordaremos algumas funções nativas de CAPL.

|  |
| --- |
| **Evento – On message** |
| **Comando:** on message ‘*MessageName*’:  **Descrição**: Evento utilizado quando é necessário executar uma ação ao receber uma mensagem específica na rede.  **Exemplo:** Ao receber a mensagem *EngineSts*, verificar se a ignição está ou não ligada.  **Código:** |

|  |
| --- |
| **Evento – On timer** |
| **Comando:** on timer*‘TimerName’*:  **Descrição**: Evento utilizado quando é necessário executar uma ação após um tempo determinado, seja ou não-cíclico.  **Exemplo:** Enviar uma mensagem cíclica na rede a cada 100 ms.  **Código:** |

|  |
| --- |
| **Evento – On sysvar** |
| **Comando:** on sysvar*‘SysvarName’*:  **Descrição**: Evento utilizado quando é necessário executar uma ação após a mudança de valor em uma variável de sistema.  **Exemplo:** Ao ativar um botão no painel, acionar o faról baixo.  **Código:** |

|  |
| --- |
| **Evento – On key** |
| **Comando:** on key*‘Key’*:  **Descrição**: Evento utilizado quando é necessário executar uma ação após acionar algum botão do teclado.  **Exemplo:** Ligar a ignição após apertar a tecla ‘A’.  **Código:**    **Importante sempre lembrar que CAPL é uma linguagem *‘Case Sensitive’*, ou seja, letras minúsculas e maiúsculas são diferentes.** |

|  |
| --- |
| **Evento – On preStart** |
| **Comando:** on preStart:  **Descrição**: Evento utilizado quando é necessário executar uma ação durante a inicialização da simulação.  **Exemplo:** Inicializar uma variável e escrever uma frase na tela “*Write*”.  **Código:** |

|  |
| --- |
| **Evento – On start** |
| **Comando:** on start:  **Descrição**: Evento utilizado quando é necessário executar uma ação assim que a simulação é iniciada.  **Exemplo:** Inicializar timers, mudar valores de sinais e enviar mensagens na rede.  **Código:** |

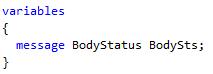
|  |
| --- |
| **Evento – On stopMeasurement** |
| **Comando:** on stopMeasurement:  **Descrição**: Evento utilizado quando é necessário executar uma ação quando a simulação é pausada.  **Exemplo:** Escrever na tela “Write” ou escrever em um arquivo log.  **Código:** |

|  |
| --- |
| **Evento – On errorFrame** |
| **Comando:** on errorFrame:  **Descrição**: Evento utilizado quando é necessário executar uma ação ocorre um ‘Error Frame’ no barramento.  **Exemplo:** Escrever na tela “*Write*” o barramento que ocorreu um erro.  **Código:** |

1. **Declarando Mensagens**

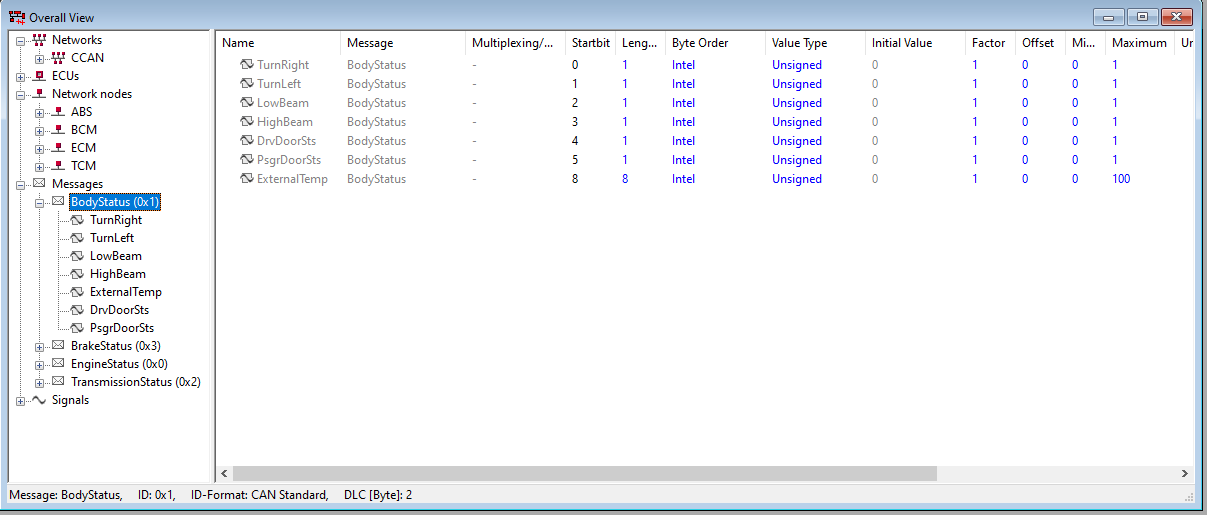
As vantagens de se trabalhar com um DBC no CAPL Browser, é que podemos usar suas respectivas mensagens e sinais facilmente. Porém, para isso é necessário executar a declaração das mensagens a serem utilizadas no programa.

Para declarar uma mensagem, basta colocar ‘**message** messageName’ seguido do nome da variável que assumirá a estrutura dessa mensagem e que será usada no código.



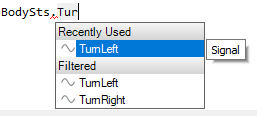
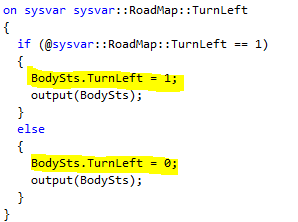
Como podemos ver, declarar uma mensagem em CAPL é muito simples. Porém, é importante entender o conceito. Ao declarar uma mensagem, estamos declarando todos os atributos pertencentes dessa mensagem a nossa variável.

Para entendermos melhor, abaixo temos nosso DBC com a mensagem ‘*BodyStatus’* destacada.



Podemos ver que a mensagem ‘*BodyStatus*’ possui 7 sinais, todos com seus respectivos tamanhos, e bit’s de início. Isso seria como a ‘estrutura da mensagem’. No código acima, ao declarar a variável ‘*BodySts*’, estamos criando uma instância da mensagem ‘BodyStatus’ para essa variável, igualmente quando trabalhamos com ‘**struct’** em C. Ou seja, **‘*BodyStatus*’** seria o tipo da variável ‘***BodySts’.***

Após declarada, a variável assume a estrutura da ‘mensagem’, e com isso conseguimos fazer referência aos sinais, utilizando o ponto ‘.’

É importante lembrar que usaremos o nome da mensagem (*BodyStatus)* apenas na declaração, após isso apenas usaremos nome da variável (*BodySts*), tanto para setar o valor dos sinais, quanto para enviá-la na rede, como podemos ver nas linhas grifadas e no comando “*output(BodySts)*”.

1. **Declarando e inicializando Temporizadores**

O temporizador, mais conhecido como *‘timer’*, é uma das ferramentas mais úteis quando se trata de desenvolvimento de simulações, com ele é possível o envio de mensagens cíclicas no barramento CAN.

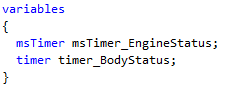
A utilização de um temporizador segue basicamente os 3 passos a seguir:

* Declaração do temporizador
* Inicialização do temporizador
* Evento a ser executado após timer expirado

**Declaração:**

A declaração de um temporizador, é feita da mesma maneira que a declaração de uma variável comum (int,float), coloca-se o tipo do temporizador seguido do nome do mesmo.  
 Existem dois tipos de temporizadores em CAPL, o ‘*msTimer*’ e o ‘*Timer*’, a única diferença entre eles é que um trabalha com o tempo em milisegundos e outro em segundos.

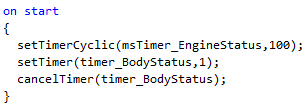
Segue o código abaixo com a declaração timers:



**Inicialização:**

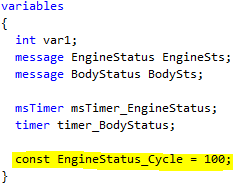
Após declarado o temporizador, ele pode ser inicializado para funcionar apenas uma vez ou ciclicamente, também pode ser reiniciado ou cancelado a qualquer momento. Para isso usamos as seguintes funções:

* setTimer(*temporizador,tempo*)
* setTimerCyclic(*temporizador, tempo*)
* cancelTimer(*temporizador*)

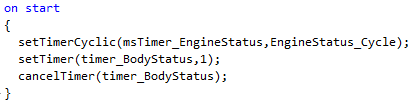


Sempre que uma função, *setTimer* ou *setTimerCyclic* é chamada, o contador indicado na função é zerado e começa sua contagem, ou seja, se um contador ‘X’ ja está em contagem, mesmo que quase expirando, e ocorre uma chamada da função setTimer ou setTimerCyclic para esse mesmo contador ‘X’, ele reinicia sua contagem do 0 sem a execução do evento *‘on timer X’*.

É importante ressaltar, que quando estamos lidando com mensagens cíclicas, é de boa prática declararmos o tempo cíclico da mensagem como constantes na seção superior do código.

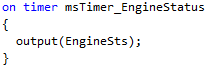


Reformulando o caso anterior, ficariamos com o seguinte código:



**Evento:**

Quando um temporizador é expirado, ocorre um ‘evento’, cada temporizador possui a seu respectivo ‘evento’. Utilizando a função **“on timer”**, a simulação executará os comandos dentro do evento do respectivo timer.



Caso um timer seja declarado, porém nenhum evento ‘*on timer’* seja atribuído ao mesmo, um ‘*Warning*’ aparecerá na hora que o código for compilado.

Agora que todas as partes para a utilização de um temporizador foram mostradas e explicadas separadamente, segue um exemplo de código de uma implementação completa de um temporizador responsável pelo envio cíclico da mensagem “*EngineSts*”:

|  |
| --- |
|  |

1. **Funções CAPL**

Como dito anteriormente, existem diversas funções internas em CAPL, abordarei algumas das mais utilizadas:

* write
* output
* sysGetValue
* sysSetValue

|  |
| --- |
| **Função – Write** |
| **Comando:** write (“Texto”):  **Descrição**: Função que escreve na janela ‘*Write*’, semelhante a função **printf** da linguagem C. É muito utilizada para “debugging” do código, pois auxília o desenvolvedor se localizar no código, e saber se um evento ou função estão sendo ativados corretamente. Também é muito útil quando se deseja passar informações ao usuário.  **Exemplo:** Quando houver uma falha no motor, escrever “Falha no motor” na janela *Write.*  **Código:**    **Janela Write:** |

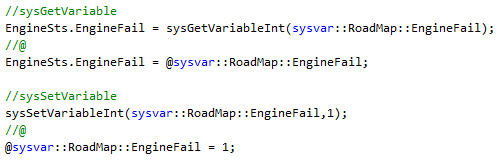
|  |
| --- |
| **Função – Output** |
| **Comando:** output (*message*):  **Descrição**: Função que envia uma mensagem para o barramento CAN, bastante utilizada em conjunto com temporizadores cíclicos ou quando é necessária a mudança do valor de um sinal, é possível acompanhar o envio das mensagens na janela ‘*Trace*’  **Exemplo:** Enviar a mensagem *EngineSts* no barramento quando começar a simulação.  **Código:**    **Janela Trace:** |

|  |
| --- |
| **Função – SysGetVariable** |
| **Comando:** sysGetVariableInt (*sysvar*)  sysGetVariableFloat (*sysvar*)  sysGetVariableString (*sysvar*)  sysGetVariableChar (*sysvar*)  **Descrição**: Função que lê o valor de uma variável de sistema.  **Exemplo:** Atribuir ao sinal *OilLevelFail* valor lido na variável de sistema *OilLevelFail*.  **Código:**    É necessário atribuir o tipo correto da variável para que não ocorra erros de compilação: |

|  |
| --- |
| **Função – SysSetVariable** |
| **Comando:** sysSetVariableInt (*sysvar,value*)  sysSetVariableFloat (*sysvar,value*)  sysSetVariableString (*sysvar,value*)  sysSetVariableChar (*sysvar,value*)  **Descrição**: Função que atribui um valor a uma variável de sistema.  **Exemplo:** Atribuir o valor ‘0’ a variável de sistema *EngineFail.*  **Código:**    É necessário atribuir o tipo correto da variável para que não ocorra erros de compilação: |

**Utilização do “@”:**

Quando estamos lidando com valores numéricos, podemos substituir as funções **sysGetVariable** e **sysSetVariable** por apenas o ‘@’, conforme apresentado no código abaixo:



A vantagem de utilizar o **‘@’** é que nao é necessário especificar se a variável é *int* ou *float*.

1. **Referências**

Vector CANtech, Inc. (14 de dezembro de 2004) - **Programming with CAPL**. USA.

Shyam Sundhar (12 de outubro de 2020) - **CANoe Test Feature Set Tutorial**.