Felipe Gonçalves Assis

Hardware Dedicado para Demodulação S-FSK Baseada em Verossimilhança

Campina Grande - PB
Abril de 2014

Felipe Gonçalves Assis

Hardware Dedicado para Demodulação S-FSK Baseada em Verossimilhança

Orientador: Marcos Ricardo Alcântara Morais

Universidade Federal de Campina Grande

Campina Grande - PB
Abril de 2014

Felipe Gonçalves Assis
Hardware Dedicado para Demodulação S-FSK Baseada em Verossimilhança/
Felipe Gonçalves Assis. – Campina Grande - PB, Abril de 2014-
151 p. : il. (algumas color.) ; 30 cm.

Orientador: Marcos Ricardo Alcântara Morais

Trabalho de Conclusão de Concurso – Universidade Federal de Campina Grande,
Abril de 2014.
1. Hardware. 2. PLC. I. Marcos Ricardo Alcântara Morais. II. Universidade
Federal de Campina Grande. III. Centro de Engenharia Elétrica e Informática.
IV. Hardware Dedicado para Demodulação S-FSK Baseada em Verossimilhança

CDU 621.38

Felipe Gonçalves Assis

Hardware Dedicado para Demodulação S-FSK Baseada em Verossimilhança

Trabalho aprovado. Campina Grande - PB, 25.04.2014:

Professor Marcos Ricardo Alcântara Morais
Universidade Federal de Campina Grande
Orientador

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Campina Grande - PB
Abril de 2014

Agradecimentos

Em primeiro lugar a meus pais, por fazerem de mim o que sou e me apoiarem durante toda minha trajetória.

Aos professores Marcos Morais e Elmar Melcher, por suas orientações e ensinamentos.

À toda a equipe do projeto PLCM, sem o trabalho da qual esta contribuição teria um alcance muito menor.

Resumo

Nas últimas décadas, o interesse em aplicações como automação residencial e infraestruturas avançadas de medição têm motivado extensas investigações sobre comunicação pela rede elétrica, ou PLC (Power Line Communication). Essas aplicações requerem modems viáveis que possam se comunicar em banda estreita usando a rede elétrica como canal.

Nessa área, o desafio primordial está em superar as características de ruído e atenuação da rede, não projetada para ser um canal de comunicação. Estratégias para superá-lo incluem o desenvolvimento de padrões robustos de codificação e modulação, técnicas eficientes de demodulação e protocolos de rede adequados às inevitáveis oscilações da capacidade de transmissão.

Aqui trabalhamos na demodulação. Partindo de ideias teóricas encontradas na literatura, procedemos à especificação completa de um *hardware* dedicado para demodulação S-FSK, segundo o padrão IEC 61334-5-1. Um modelo é escrito para validação.

Palavras-chave: Hardware. PLC. S-FSK. Verossimilhança.

Abstract

In the last decades, interest in applications such as home automation and advanced metering infrastructures has motivated extensive investigations on the subject of power line communication (PLC). This applications require viable modems which can communicate in narrow band using the power line as the communication channel.

In this area, the primary challenge is in overcoming the noise and attenuation characteristics of the power line, which was not designed to be a communication channel. Among the strategies for dealing with this are the development of robust codification and modulation standards, efficient demodulation techniques and network protocols adequate to the inevitable oscillations of the transmission capability.

Here we work on demodulation. Starting from theoretical ideas found in the literature, we proceed to complete specification of a dedicated hardware for S-FSK demodulation, following the IEC 61334-5-1 standard. A model is written for validation.

Keywords: Hardware. PLC. S-FSK. Likelihood.

Lista de ilustrações

Sumário

1 Introdução

1.1 Comunicação pela Rede Elétrica

Tecnologias para transmissão de dados pela rede de distribuição de energia, genericamente referidas pela sigla em inglês PLC (*Power Line Communication*) têm sido exploradas há algumas décadas. O padrão X10, um esquema simples de comunicação para redes residenciais, por exemplo, data dos anos 1970.

Essas tecnologias variam em alcance, taxa de transmissão e aplicabilidade. Nesse trabalho estamos particularmente interessados em comunicação em faixa estreita, com freqüências da ordem de 100 kHz e taxas da ordem de kbps. Aplicações visadas incluem, por exemplo, automação residencial e medição de consumo energético.

A *automação residencial*, ou domótica, se refere à aplicação de tecnologia da informação no controle de sistemas residenciais. Redes PLC são uma opção conveniente para a comunicação entre diferentes aparelhos.

Na área de medição de consumo energético, redes PLC podem permitir o recolhimento automático de informações de consumo.

Em geral, a vantagem do uso da rede elétrica para comunicação é a sua disponibilidade, não requerendo uma estrutura adicional para conexão dos nós. Isso é particularmente saliente em aplicações como as supracitadas, nas quais os nós da rede de comunicação já estão naturalmente interligados por uma rede de distribuição de energia.

Esse canal de comunicação, no entanto, apresenta características de ruído e atenuação particularmente desafiadoras. Superar essas características na implementação de um sistema eficiente passa pelo desenvolvimento de padrões robustos de codificação e modulação, técnicas eficientes de demodulação e decodificação, e protocolos de rede adequados às inevitáveis limitações e oscilações da capacidade de transmissão.

Nesse sentido, um padrão relevante é o IEC 61334, em particular, sua parte 5-1 ((1)), que especifica as camadas física e MAC de um sistema PLC, definindo a modulação conhecida como S-FSK (*Spread Frequency Shift Keying*). O padrão, contudo, não especifica a técnica de demodulação, apesar de indicar um princípio heurístico que pode ser usado, dando espaço para avanços tecnológicos.

Neste trabalho desenvolvemos o projeto completo de um hardware dedicado para demodulação S-FSK, partindo das ideias propostas em (2) e em (3).

1.2 Projeto PLCM

Este trabalho também faz parte de um projeto maior, denominado PLCM.

O projeto "Módulo IP-core para Comunicação pela Rede Elétrica" (PLCM) está sendo desenvolvido no Laboratório de Arquiteturas Dedicadas (LAD) da UFCG como parte do programa federal Brazil-IP.

O Brazil-IP tem o objetivo de formar recursos humanos no projeto de circuitos integrados, circuitos VLSI e *IP cores*.

Dentro dessa iniciativa, o PLCM visa a desenvolver um IP core que implemente as funcionalidades das camadas mais baixas (física e MAC) de um sistema de comunicação pela rede elétrica em faixa estreita. Uma arquitetura, esquematizada na Figura 1, já foi proposta. O módulo foi dividido nos seguintes submódulos:

- TX_MAC: Cálculo de CRC e composição das mensagens.
- ENC: Codificação para correção de erros.
- MOD: Modulação digital S-FSK.
- PLL: Multiplicação de freqüência para sincronização.
- DEM: Demodulação digital S-FSK.
- DEC: Decodificação para correção de erros.
- RX_MAC: Verificação de CRC e de endereçamento.
- STATUS/CTRL: Interface de controle e monitoramento.

Aqui propomos uma especificação para o DEM, e desenvolvemos o seu modelo de referência.

1.3 O Padrão IEC 61334-5-1

O padrão IEC 61334-5-1 alcançou o status de padrão internacional em 2001. Optamos por utilizá-lo porque o esquema de modulação por ele proposto é relativamente simples e sua aplicabilidade já foi validada em produtos comerciais.

Alguns outros padrões para comunicação pela rede elétrica tem sido propostos mais recentemente. Notavelmente os padrões PRIME, G3-PLC e o padrão IEEE 1901.2. Eles têm em comum o uso de modulação OFDM (*Orthogonal Frequency Division Multiplexing*), mais sofisticada. Contudo, pelo menos um estudo ((4)), no qual os autores comparam um sistema PRIME a um tradicional, fornece resultados desanimadores para a aplicação dessa tecnologia em uma rede real, brasileira.

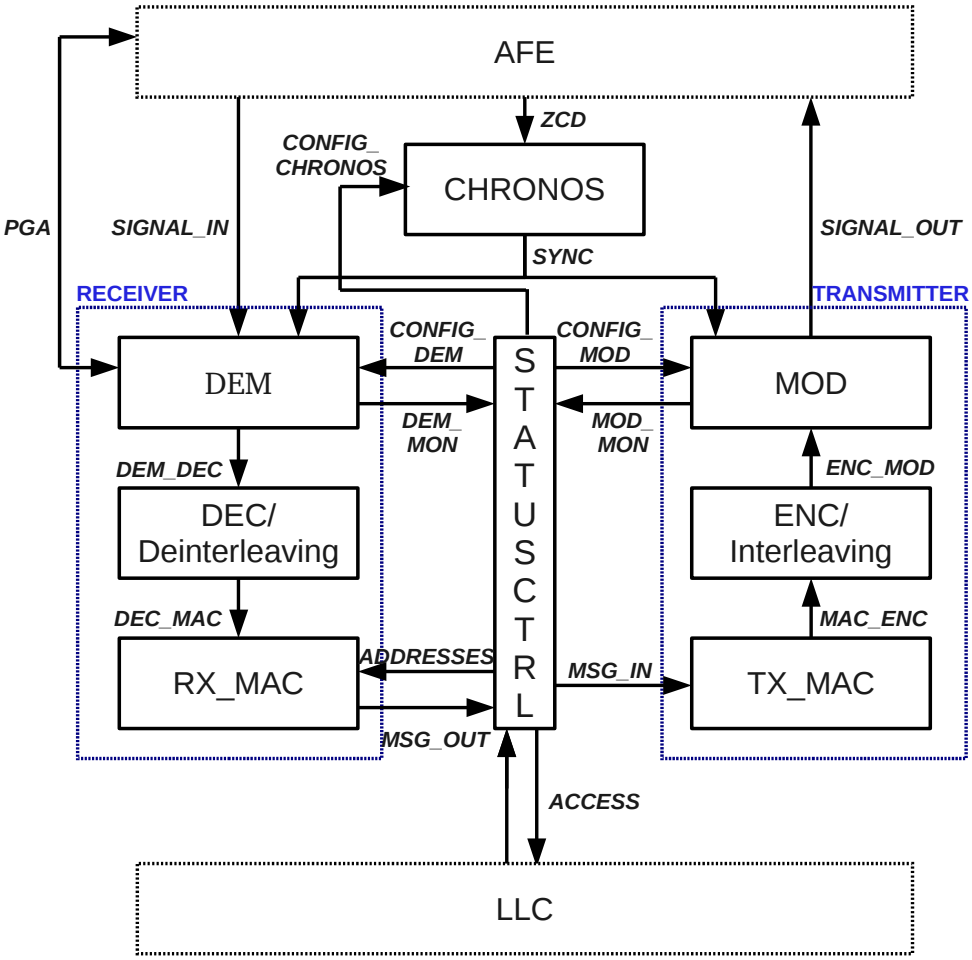

Figura 1 – Arquitetura do PLCM.

2 Análise Teórica

O projeto de um receptor envolve a resolução de vários problemas, como sincronização, detecção de sinal e por fim a demodulação e decodificação das mensagens. Neste capítulo concentramo-nos na análise teórica do problema da decodificação de uma mensagem, que consideramos essencial ao projeto.

Após descrever o modelo para o sistema, estabelecemos e justificamos as hipóteses que fundamentam a técnica de demodulação utilizada, em particular esclarecendo quais são as possibilidades de melhora sobre o esquema proposto.

No próximo capítulo, veremos como todos os aspectos da recepção se articulam neste trabalho, e no contexto mais geral do projeto do modem.

2.1 Modulação S-FSK

A modulação S-FSK, segundo definida pelo padrão IEC 61334-5-1, consiste essencialmente de uma modulação FSK onde as freqüências f_0 e f_1, correspondentes a bits 0 e 1 são escolhidas de modo a estarem afastadas uma da outra.[1]

O princípio teórico é de que, para freqüências suficientemente afastadas, as componentes do sinal em cada freqüência devem comportar-se como independentes uma da outra. Essa será a base de nossa análise.

Um sinal S-FSK pode também ser visto como dois sinais ASK redundantes: Quando há sinal numa freqüência, há silêncio na outra e vice-versa. A intuição básica por trás do esquema é que, na perda do sinal em uma das freqüências, seja por ruído elevado ou atenuação, ainda se tem na outra a possiblidade de recuperar a mensagem.

Na prática, o padrão IEC recomenda, com referência a estudos empíricos, que a distância entre as freqüências seja maior que 10 kHz.

As freqüências disponíveis para comunicação, e os níveis aceitáveis de sinal são limitados por normas. Neste trabalho, consideramos a norma europeia CENELEC EN 50065: “Signaling on low-voltage electrical installations in the frequency range 3 kHz to 148,5 KHz”.

Essa norma atribui faixas de freqüências a diferentes utilizações, conforme a Tabela 1, adaptada de (4). Nosso demodulador foi projetado para operar nas faixas A e B.

[1] No padrão IEC, essas freqüências são referidas respectivamente como f_S e f_M, freqüências de espaço (*space*) e marca (*mark*).

Tabela 1 – Faixas CENELEC.

Classificação	Faixa (kHz)	Proposta
A	3 – 95	Companhias de distribuição com suas licenças.
B	95 – 125	Disponível para consumidores sem restrição.
C	125 – 140	Disponível para uso com protocolo de acesso de mídia.
D	140 – 148.5	Disponível para consumidores sem restrição.

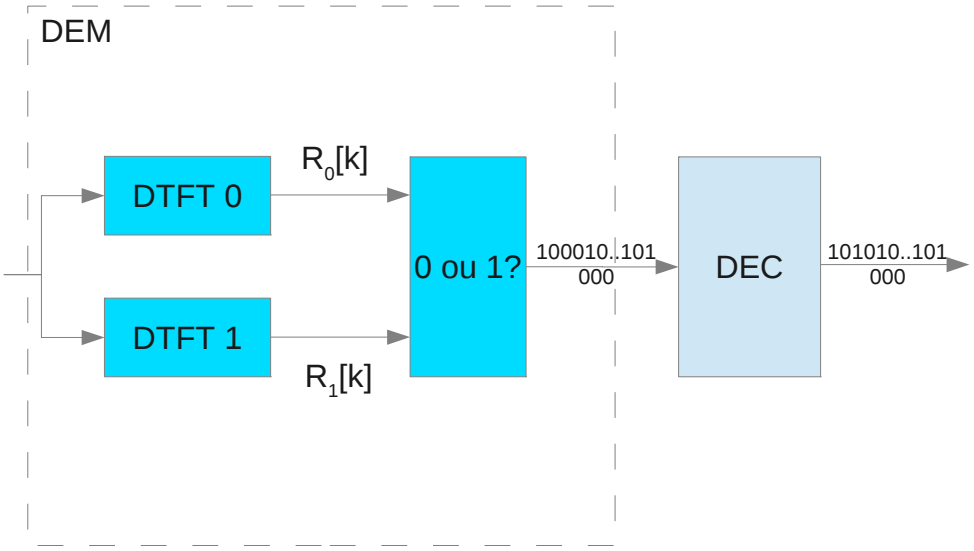

Figura 2 – Diagrama de blocos do receptor.

2.2 Receptor

Em (2), Schaub deriva um algoritmo de demodulação S-FSK ótimo partindo dos valores absolutos da Transformada de Fourier do sinal recebido em cada uma das freqüências correspondentes aos bits 0 e 1.

O sistema pode ser descrito pelo diagrama de blocos da Figura 2. Para cada bit a ser recebido, os valores R_0 e R_1 dos valores absolutos da transformada de Fourier do sinal nas freqüências de interesse são usados como base para decidir se o bit transmitido corresponde a 0 ou 1. Essa informação é então passada a um decodificador para correção de erros (supondo, claro, que a mensagem trasmitida foi codificada com algum código corretor de erros).

2.2.1 Modelo Estatístico

Para derivar o algoritmo ótimo de decisão, estabelecemos um modelo estatístico para as variáveis R_0 e R_1, ou seja, os valores absolutos da Transformada de Fourier do sinal nas freqüências de modulação.

O sinal é suposto ser uma senoide perfeita, com freqüência dependente do dado que transmite. Ao passar pelo canal PLC, este sinal sofrerá uma atenuação e a ele será adici-

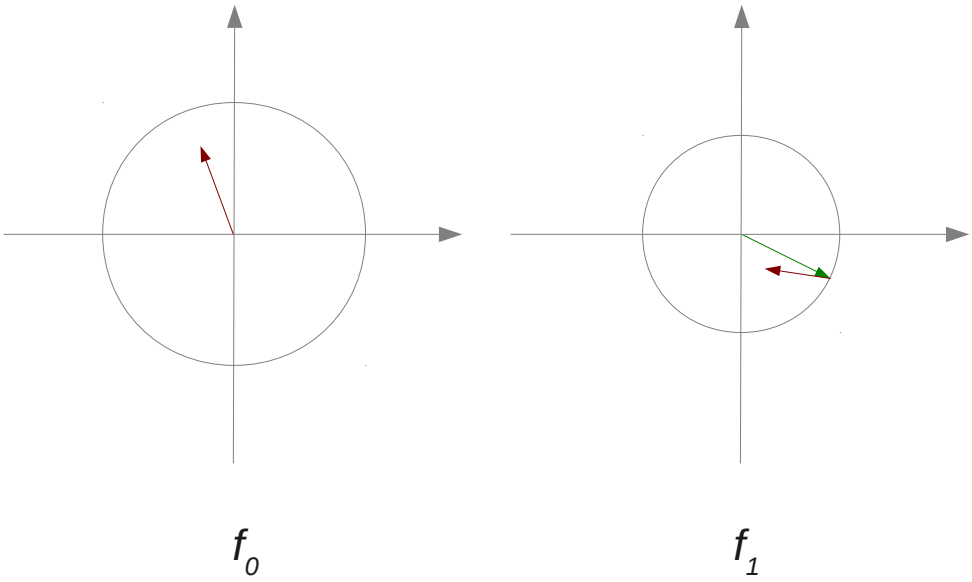

Figura 3 – Composição das componentes de freqüência na transmissão de um bit 1.

onado um ruído. A atenuação depende da freqüência do sinal, de modo que a amplitude do sinal de freqüência f_0 não é necessariamente a mesma do de freqüência f_1.

Ao tomar a Transformada de Fourier do sinal recebido, teremos em uma freqüência apenas ruído e na outra ruído mais sinal.

A componente do sinal é um valor complexo de magnitude fixa ν e fase arbitrária. A componente do ruído é modelada como uma variável aleatória gaussiana (bivariada) com matriz de covariância da forma $\sigma^2 I$. A Figura 3 ilustra uma possível amostra correspondente a transmissão de um bit 1.

Sob essas hipóteses, pode-se demonstrar que, para a freqüência onde só há ruído, o módulo da resultante terá uma distribuição de Rayleigh, enquanto para a freqüência onde há sinal, a dsitribuição será de Rice, caracterizada pela função densidade de probabilidade

$$f(x;\nu,\sigma) = \frac{x}{\sigma^2} e^{-\frac{x^2+\nu^2}{2\sigma^2}} I_0\left(\frac{\nu x}{\sigma^2}\right). \tag{2.1}$$

(I_0 representa função modificada de Bessel do primeiro tipo.)

A distribuição de Rayleigh corresponde ao caso particular $\nu = 0$.

Em síntese, temos quatro parâmetros: ν_0, ν_1, σ_0 e σ_1. Na transmissão de um bit 0, as densidades de probabilidade para os valores medidos R_0 e R_1 serão

$$f_{R_0|0}(r_0) = f(r_0;\nu_0,\sigma_0) \tag{2.2}$$
$$f_{R_1|0}(r_1) = f(r_1;0,\sigma_1), \tag{2.3}$$

enquanto que, na transmissão de um bit 1, elas serão

$$f_{R_0|1}(r_0) = f(r_0;0,\sigma_0) \tag{2.4}$$
$$f_{R_1|1}(r_1) = f(r_1;\nu_1,\sigma_1). \tag{2.5}$$

Em todo caso, partindo da hipótese fundamental de independência dos canais, a função densidade de probabilidade da distribuição conjunta é simplesmente o produto das distribuições marginais:

$$f_{(R_0,R_1)|i}(r_0, r_1) = f_{R_0|i}(r_0) f_{R_1|i}(r_1), \qquad i \in \{0, 1\}. \tag{2.6}$$

A grandeza $L_i(r_0, r_1) = f_{(R_0,R_1)|i}(r_0, r_1)$ é dita a *verossimilhança* do evento "bit i".

2.2.2 Receptor Ótimo

Considerando que o bit transmitido X é uma variável aleatória de Bernoulli com parâmetro $p = p_0 = p_1 = 0.5$, a probabilidade de que o valor do bit seja i dado que foi recebido o sinal (r_0, r_1) pode ser obtida por uma versão do Teorema de Bayes:

$$P(X = i|R_0 = r_0, R_1 = r_1) = \frac{p_i L_i(r_0, r_1)}{p_0 L_0(r_0, r_1) + p_1 L_1(r_0, r_1)}. \tag{2.7}$$

Para minimizar a probabilidade de erro, escolhemos o valor mais provável para i, o que equivale a maximizar o produto $p_i L_i(r_0, r_1)$ da probabilidade a priori pela verossimilhança. No caso, como as probabilidades a priori são idênticas, só precisamos maximizar a verossimilhança. O algoritmo de decisão ótimo é portanto:

Escolha 1 se $L_1 > L_0$, escolha 0 se $L_0 > L_1$.

(caso $L_0 = L_1$, a decisão pode ser qualquer).

Substituindo as expressões para L_0 e L_1 e simplificando, obtemos que a condição $L_1 > L_0$ é equivalente a

$$\log I_0\left(\frac{\nu_1 r_1}{\sigma_1^2}\right) - \frac{\nu_1^2}{2\sigma_1^2} > \log I_0\left(\frac{\nu_0 r_0}{\sigma_0^2}\right) - \frac{\nu_0^2}{2\sigma_0^2}. \tag{2.8}$$

2.2.3 Decisão Suave

O critério derivado acima garante a minimização da taxa de erro de bit. Contudo, observando o sistema mais globalmente, estamos na verdade interessados em minimizar a taxa de erros de pacote.

Um pacote estará tipicamente codificado segundo um código corretor de erros, de modo que nem todas as seqüências de bits são padrões válidos.

No esquema proposto até aqui, o demodulador entrega ao decodificador uma seqüência de bits. Este por sua vez procura entre as seqüências válidas a que mais aproxima a fornecida pelo demodulador. Esse procedimento é o que se chama de *decisão abrupta* (*hard decision*). Sob esse esquema, a decodificação terá sucesso se os erros forem suficientemente poucos, e o melhor que podemos fazer é minimizar a probabilidade de erro de bit, como descrito acima.

Contudo, para realmente minimizar a probabilidade de erro de pacote, poderíamos maximizar não a verossimilhança de cada bit individualmente, mas sim a verossimilhança de todas as seqüências de bits válidas.

Ora, a verossimilhança de uma seqüência é simplesmente o produto das verossimilhanças dos bits individuais (supondo independência entre as componentes de ruído em diferentes *bit slots*). Assim, basta que o demodulador entregue, em vez de uma seqüência de palpites sobre se os bits são 0 ou 1, as seqüências de verossimilhanças L_0 e L_1 para cada bit. Na verdade, como só interessa o valor relativo delas na comparação, basta informar L_0/L_1, ou, equivalentemente, $\log(L_0/L_1)$ (o uso do logaritmo tem a conveniência de simplificar as fórmulas, além de transformar produtos em soma). No nosso caso, temos

$$\log(L_0/L_1) = \left(\log I_0\left(\frac{\nu_0 r_0}{\sigma_0^2}\right) - \frac{\nu_0^2}{2\sigma_0^2}\right) - \left(\log I_0\left(\frac{\nu_1 r_1}{\sigma_1^2}\right) - \frac{\nu_1^2}{2\sigma_1^2}\right). \tag{2.9}$$

Esse procedimento é o que se chama de *decisão suave* (*soft decision*). Para suportá-lo, como vimos, o demodulador só precisa informar o valor de $\log(L_0/L_1)$ para cada bit a ser estimado. Um demodulador de decisão suave pode ser trivialmente convertido num de decisão abrupta observando-se que $L_1 > L_0 \Leftrightarrow \log(L_0/L_1) < 0$, i.e., basta restringir-se ao bit de sinal do logaritmo da razão de verossimilhança.

Já um exemplo notável de decodificação com decisão suave é o Algoritmo de Viterbi para códigos convolucionais, que, partindo dos logaritmos das razões de verossimilhança individuais de cada bit, maximiza eficientemente a verossimilhança de uma seqüência.

Com efeito, as razões de verossimilhança são a base mesmo de aplicações mais sofisticadas.

Um caso interessante é o da repetição de mensagem. Assim, por exemplo, se, de acordo com o protocolo adotado, temos certeza de que o pacote que estamos recebendo é a repetição de um outro previamente enviado que foi detectado mas incorretamente decodificado, podemos minimizar a probabilidade de erro nesta nova tentativa somando os logaritmos das razões de verossimilhança previamente calculados com os obtidos na nova transmissão.

O importante aqui é que a implementação disso não exige nenhuma mudança no demodulador. A razão de verossimilhança resume toda a informação que temos sobre o sinal transmitido, dado o modelo adotado.

No projeto PLCM, não consideramos utilizar, ao menos nesta iteração, um decodificador com decisão suave. Por isso, no projeto aqui apresentado, este demodulador é de decisão abrupta. Ainda assim, dado que nos baseamos no princípio de máxima verossimilhança, não é difícil adaptar o projeto para suportar decisão suave. Sobre isso, ver Seção 3.7.1.

2.3 Princípios Utilizados

Nesta seção analisamos as condições sobre o qual o esquema de demodulação proposto é de fato ótimo. Com isso, pretendemos esclarecer quais são as possibilidades de melhora sobre o esquema proposto.

Dividimos a análise em duas partes. Primeiro estudamos a possível relevância da informação descartada ao nos restringirmos aos valores absolutos da Transformada de Fourier do sinal em f_0 e f_1. Em seguida investigamos a adequação do modelo estatístico.

2.3.1 Pré-Processamento

Em (2), o uso dos valores absolutos do espectro do sinal nas duas freqüências é motivado pelo aproveitamento de infraestruturas já existentes. Como não estamos limitados por essas estruturas (o artigo é de 1994), fazemos algumas considerações sobre o que Schaub toma como base, antes de analisar o algoritmo em si.

Restringir-se ao espectro do sinal apenas nas freqüências da modulação pode parecer óbvio, já que não é transmitido sinal em outras freqüências, mas na verdade estamos descartando informação possivelmente relevante sobre o ruído presente. Em princípio, poderíamos utilizar essa informação para estimar parâmetros do ruído e subtraí-lo do sinal a ser processado.

Evidentemente, isso só é possível se tivermos à mão um modelo confiável do ruído que forneça alguma dependência relevante entre diferentes faixas do seu espectro.

Um exemplo concreto que pode ser imaginado é o uso de um modelo de ruído impulsivo periódico para detectar e subtrair esse tipo de ruído do sinal medido. Tal processamento presumivelmente seria feito sobre o sinal não-filtrado.

O receptor proposto por Schaub portanto só pode ser considerado ótimo sob a condição de que não há informação útil nas freqüências descartadas. Essa condição é verificada sob qualquer uma das seguintes hipóteses:

- O espectro do ruído em diferentes freqüências é independente.
- Não há modelo disponível para explorar a dependência entre freqüências.
- Qualquer dependência entre diferentes freqüências foi eliminada por um pré-processamento.

A outra condição básica para que o receptor possa ser considerado ótimo é que a fase do sinal seja irrelevante. Schaub explicitamente se restringe a demoduladores não-coerentes. O padrão IEC 61334-5-1 de fato não faz referência a demodulação coerente nem impõe nenhuma restrição sobre a fase das senoides transmitidas em cada *bit slot*.

Poderíamos em princípio introduzir esse tipo de restrição sobre o modulador, e projetar um demodulador coerente, se dispostos a comprometer a interoperabilidade com outros

sistemas. Claro, para aproveitar a informação de fase, os requisitos sobre a precisão do sistema de sincronização são bem mais estritos do que para uma demodulação não-coerente. Mais precisamente, os *bit slots* precisariam ser delimitados com erro de ordem de grandeza menor que o período das portadoras. Não adotamos essa abordagem.

2.3.2 Modelo Estatístico

As hipóteses básicas do nosso modelo estatístico são

- As componentes do ruído em cada freqüência são variáveis aleatórias independentes.
- Cada uma tem distribuição normal bivariada com variância da forma $\sigma^2 I$.

A primeira hipótese é a essência da modulação S-FSK. Um modelo mais sofisticado poderia incluir um parâmetro de correlação, mas, com um parâmetro adicional a ser estimado, é possível que a performance sofresse uma degradação em vez de melhora.

A segunda suposição, como formulada, talvez pareça um tanto arbitrária. Argumentamos, contudo, que ela é uma hipótese fraca e que, portanto, o modelo é robusto.

Em primeiro lugar, não há razões para supor qualquer anisotropia do ruído nas freqüências de modulação se elas não guardam relação com a freqüência da rede.

Em segundo lugar, se olhamos para a expressão da parte real, por exemplo, da Transformada de Fourier do ruído numa certa freqüência, ela corresponde a uma média (ponderada por uma função senoidal) desse processo estocástico. Se o processo é suposto gaussiano, pela propriedade de fechamento, uma soma ou integral resultará ainda numa variável gaussiana. Mesmo que o processo não seja gaussiano, é de se esperar, por alguma versão do Teorema Central do Limite, que a distribuição dessa média, tomada ao longo de um intervalo suficientemente longo, seja aproximadamente normal.

Estas duas propriedades da transformada do ruído – distribuição independente da fase e projeção com distribuição normal – caracterizam completamente uma variável com distribuição normal bivariada e variância da forma $\sigma^2 I$.

2.3.3 Estimação de Parâmetros

Apesar dessa robustez do modelo, ele não pode ser mais confiável do que seus parâmetros, que precisam ser estimados de alguma forma.

A ideia adotada é estimá-los durante a recepção do cabeçalho da mensagem. Segundo o padrão IEC 61334-5-1, todo pacote é iniciado por um padrão alternado de 16 bits (0xAAAA) seguido de um outro padrão pré-definido de delimitação de início de subtrama (0x54C7) (bits mais significativos são transmitidos primeiro).

Schaub (2) propõe um estimador simples e eficaz para os parâmetros da distribuição. Denominando por χ_i o conjunto dos índices do cabeçalho correspondentes a bit i e por

$r_i[k]$ o valor absoluto da Transformada de Fourier na freqüência f_i calculada para o bit de índice k, estimamos

$$\sigma_i^2 = \frac{1}{|\overline{\chi_i}|} \sum_{k \in \overline{\chi_i}} r_i^2[k] \tag{2.10}$$

$$\nu_i^2 = \left[-\sigma_i^2 + \frac{1}{|\chi_i|} \sum_{k \in \chi_i} r_i^2[k] \right]^+ . \tag{2.11}$$

Aqui, $\overline{\chi_i}$ denota o complemento de χ_i, e $[\cdot]^+$ denota a função parte-positiva.

No projeto aqui proposto, implementamos essa ideia. Todavia, em retrospectiva, enxergamos ao menos uma limitação.

A hipótese fundamental por trás desses estimadores é que as diferentes medições são amostras independentes das mesmas distribuições de probabilidade, com os mesmos parâmetros.

É de se esperar que os níveis ν_0 e ν_1 de sinal permaneçam constantes ao longo da transmissão, apesar de ser em princípio possível que um evento abrupto durante a transmissão mude as características do canal.

Contudo, temos sérias razões para esperar que os níveis σ_0 e σ_1 de ruído variem de forma previsível.

Isso porque, pelo padrão IEC 61334-5-1, os *bit slots* são sincronizados com os cruzamentos por zero da rede elétrica, e também é síncrono com o sinal da rede o ruído produzido por diversas fontes.

Em (5), Katayama et al. fornecem evidência experimental da dependência do nível do ruído com a fase da rede. Com efeito, essa dependência é uma das características fundamentais do modelo de ruído em canais PLC de banda estreita proposto nesse artigo.

Para aproveitar esse fato e obter um desempenho superior, poderíamos estimar variâncias de ruído separadamente para cada fase de um semiperíodo da rede.

Assim por exemplo, se estamos transmitindo à mínima taxa possível pelo IEC 61334-5-1, de 3 bits por semiperíodo (360 bps numa rede de 60 Hz), poderíamos estimar até três pares de parâmetros $\sigma_{i,j}$. Se transmitirmos à 6 bits por semiperíodo, seriam até seis pares, e assim por diante. Claro, também poderíamos limitar a granularidade da estimação, escolhendo um meio-termo entre ter apenas um par de parâmetros para todas as fases, e um par para cada fase de *bit slot*. Seria importante levar em conta os picos estreitos de ruído que Katayama et al. observam, e incluem em seu modelo.

Com a introdução de mais parâmetros, os 32 bits de cabeçalho não seriam suficientes para obter estimadores confiáveis. Felizmente, ruído é algo que está disponível desde muito antes do início da transmissão de um pacote.

A Equação 2.10 poderia ser assim adaptada: Em vez de calcular uma média sobre o conjunto $\overline{\chi_i}$, calcularíamos uma média sobre um conjunto de amostras de ruído coletadas na mesma fase de diferentes semi-períodos da rede.

Na Equação 2.11, o termo σ_i^2 teria de ser substituído por uma média das variâncias estimadas sobre as fases ocupadas por χ_i.

(Outros detalhes precisariam ser considerados. No projeto atual, estimamos novos parâmetros a cada novo pacote. Se adotarmos a ideia acima de usar um período de silêncio antes das mensagens para estimar a dependência do ruído com a fase da rede, precisaríamos reaproveitar essas estimativas ao receber pacotes adjacentes. Isso porque o padrão IEC 61334-5-1 prescreve um espaçamento de 24 bits de silêncio entre dois pacotes, o que pode não ser suficiente para refazer as estimações, especialmente a taxas de trasnmissão mais altas. Por outro lado, em transmissões rápidas, é menos provável que as características do canal mudem.)

3 Projeto

Aqui discutimos em detalhes todos as decisões de projeto.

Já vimos que o objetivo do demodulador aqui proposto é integrar um IP-core de um modem PLC em desenvolvimento, cuja arquitetura já está definida. Assim, começamos analisando o ambiente em que este bloco se encaixa, e as definições de suas entradas e saídas.

A seguir, partindo da ideia fundamental estabelecida no capítulo anterior, propomos uma arquitetura para o nosso bloco, e vemos como um hardware dedicado viável pode aproximar o receptor ótimo lá definido.

Este capítulo contém uma razoável quantidade de análise teórica na justificativa de algumas decisões de projeto. Elas foram incluídas aqui, e não no capítulo anterior, porque só fazem sentido no contexto específico das estratégias de implementação que escolhemos.

3.1 Vizinhança

A Figura 4 é uma repetição da Figura 1 e ilustra a arquitetura do PLCM.

O bloco DEM corresponde ao demodulador, que é objetivo deste trabalho.

O bloco AFE não faz parte do IP-core em si, e representa o *front end* analógico entre ele e a rede elétrica. Para testes e implementações, ele está sendo projetado como uma combinação do circuito integrado AFE031, com o conversor A/D MAX1426, e uma lógica de cola a ser implementada por um microcontrolador ou em FPGA.

Suas responsabilidades básicas na recepção são a filtragem para corte da componente de 60 Hz e *anti-aliasing*, a conversão analógico-digital e a detecção dos cruzamentos por zero do sinal da rede, base da sincronização. Ele também deve fornecer amplificação com ganho programável do sinal a ser digitalizado.

O bloco CHRONOS representa o sincronizador. Ele consiste basicamente de um PLL digital, produzindo pulsos em sincronia com o sinal de cruzamento por zero, marcando os limites dos *bit slots*. Em conformidade com o IEC 61334-5-1, cada semiperíodo da rede é dividido em $3n$ *bit slots*. n é um valor inteiro positivo configurável.

O bloco DEC representa o decodificador e é puramente funcional. Para cada mensagem recebida, o demodulador envia seu palpite sobre o conteúdo para correção de erros. A interface DEM_DEC é portanto de decisão abrupta. Isso porque, ao menos na iteração atual, não se planeja implementar um algoritmo de decisão suave.

Finalmente, o bloco STATUS/CTRL fornece a interface com o usuário, representado pelo LLC (*Logical Link Control*), e centraliza a configuração e o monitoramento dos demais

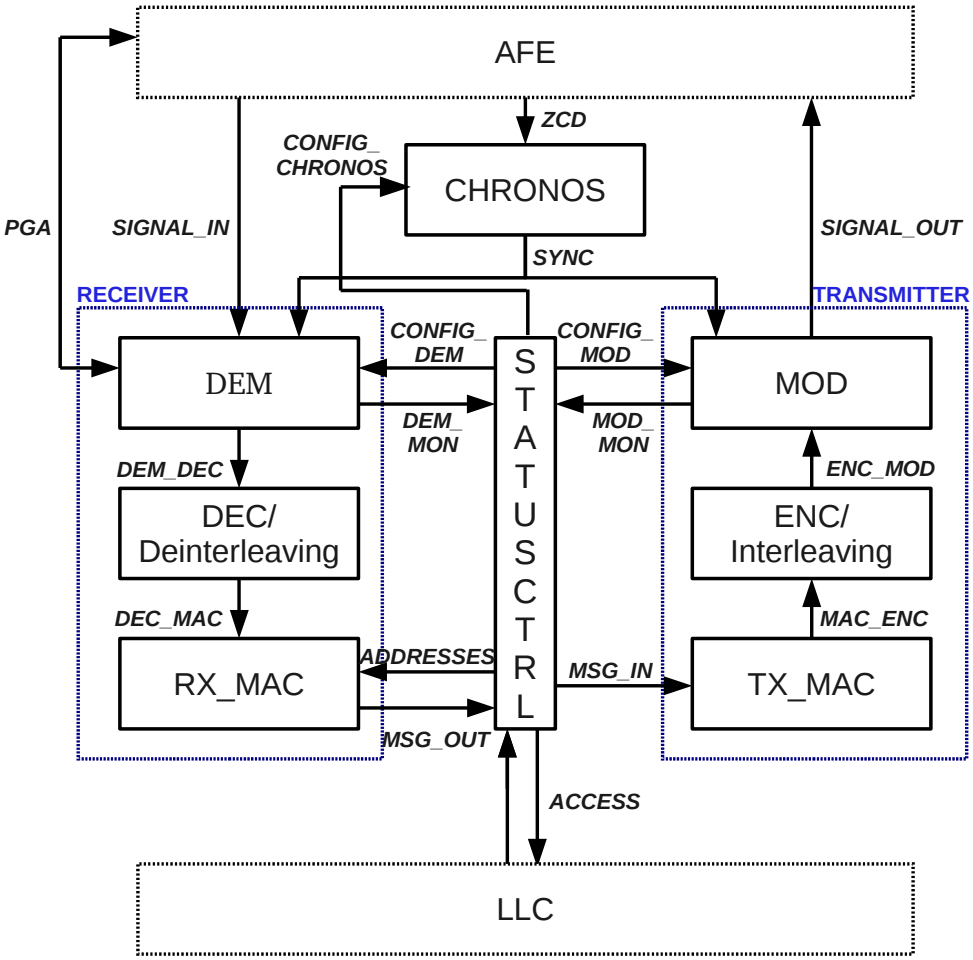

Figura 4 – Arquitetura do PLCM.

blocos.

3.2 Interfaces

As entradas de dados do demodulador são basicamente o sinal digitalizado da rede, e o sinal de sincronização. As entradas de configuração são as freqüências de modulação e um limiar usado na detecção de cabeçalho.

Como saída, o demodulador deve produzir basicamente: (i) as mensagens recebidas, (ii) um sinal de controle para o amplificador de ganho programável e (iii) uma indicação do estado da recepção.

A interface SIGNAL_IN consiste do sinal da rede digitalizado mais um bit de validade.

A interface SYNC contém o sinal de sincronização mais um bit de validade. O sinal de sincronização consiste de duas seqüências de pulsos: Uma correspondendo aos limites dos *bit slots* e outra apenas àqueles que correspondem a cruzamento por zero.

A interface PGA contém um sinal de requisição de ganho para o amplificador de entrada e um sinal de resposta indicando o valor atualmente efetivo. A proporção entre

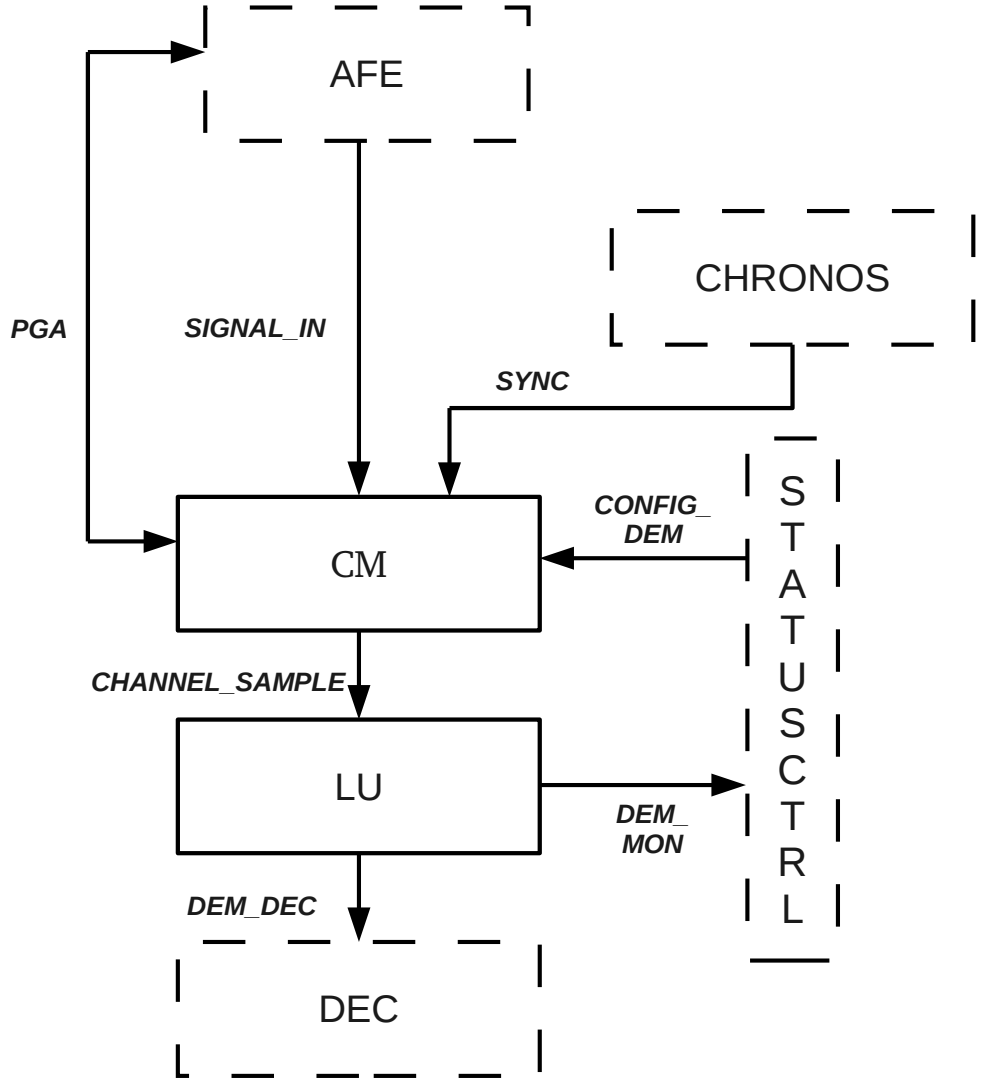

Figura 5 – Arquitetura do demodulador.

os ganhos é sempre uma potência de dois.

A interface CONFIG_DEM contém os valores da configuração de freqüência de modulação e de limiar de detecção de cabeçalho. A codificação e o significado desses valores é discutida mais abaixo.

A interface DEM_MON contém os parâmetros do sinal mais recentemente estimados e o estado da recepção.

Finalmente, a interface DEM_DEC é por onde as mensagens recebidas são repassadas ao decodificador. Dois bits de controle, `valid` e `ready` são usados para implementar um protocolo simples.

3.3 Arquitetura Geral

A Figura 5 mostra a arquitetura do demodulador. Dividimo-lo em duas partes: o *Monitor de Canal* (CM) e a *Unidade de Verossimilhança* (LU).

O Monitor de Canal é responsável pelo cálculo da Transformada de Fourier em Tempo Discreto (DTFT) nas freqüências configuradas para cada *bit slot* indicado. Os resultados são repassados à Unidade de Verossimilhança, que é responsável por detectar quando uma mensagem está sendo enviada e proceder à decisão sobre os dados transmitidos.

Como veremos adiante, escolhemos para o cálculo da DTFT o Algoritmo de Goertzel, com aritmética de ponto fixo. A Unidade de Verossimilhança usa aritmética de ponto flutuante, e uma aproximação linear por partes é usada no cálculo de uma função transcendente, seguindo uma ideia de Veronesi et al.(3).

Aqui não propomos nem testamos um algoritmo de controle de ganho específico. Em vez disso, projetamos o Monitor de Canal para que não dependa do algoritmo de controle de ganho.

Também não incluímos no nosso módulo a capacidade de fazer um ajuste fino da sincronização. Em vez disso, confiamos no sinal fornecido pelo sincronizador.

3.4 Cálculo da DTFT

Para calcular a Transformada de Fourier de Tempo Discreto foi considerado inicialmente apenas o Algoritmo de Goertzel, escolhido para o projeto. Há porém ao menos uma alternativa que merece consideração.

3.4.1 Algoritmo de Goertzel

O algoritmo de Goertzel permite o cálculo da DTFT em uma freqüência específica por meio de um filtro recursivo de segunda ordem com apenas uma constante não unitária.

Da fórmula da DTFT temos

$$y[n] = \sum_{k=0}^{n} e^{-j\omega k} u[k]. \tag{3.1}$$

O algoritmo básico para cálculo de somatórios é calcular iterativamente $y[n]$ segundo a fórmula

$$y[n] - y[n-1] = e^{-j\omega n} u[n], \tag{3.2}$$

que corresponde a um filtro recursivo de primeira ordem cuja entrada é o somando. Para nos livrarmos do cálculo de $e^{-j\omega n}u[n]$, escrevemos $y[n] = e^{-j\omega n}v[n]$. Substituindo na equação e multiplicando por $e^{j\omega n}$, vem

$$v[n] - e^{j\omega} v[n-1] = u[n], \tag{3.3}$$

que só exige a multiplicação pela constante complexa $e^{j\omega}$. Como só nos interessa o valor absoluto da transformada, podemos muito bem usar $v[n]$ em lugar de $y[n]$.

Contudo, podemos ir além. Para evitar operações com complexos, tomamos a função de transferência do filtro $\frac{V(z)}{U(z)} = \frac{1}{1-e^{j\omega}z^{-1}}$, e a multiplicamos por seu conjugado, resultando no filtro de segunda ordem

$$\frac{X(z)}{U(z)} = \frac{1}{1-e^{j\omega}z^{-1}}\frac{1}{1-e^{-j\omega}z^{-1}} = \frac{1}{1-2\cos\omega\cdot z^{-1}+z^{-2}}, \tag{3.4}$$

de equação

$$x[n] - 2\cos\omega\cdot x[n-1] + x[n-2] = u[n], \tag{3.5}$$

que só envolve variáveis reais e uma multiplicação por constante.

Podemos recuperar $v[n]$ a partir de

$$x[n] - e^{-j\omega}x[n-1] = v[n]. \tag{3.6}$$

O quadrado do valor absoluto da transformada é portanto

$$|y[n]|^2 = |v[n]|^2 = x[n]^2 - 2\cos\omega\cdot x[n]x[n-1] + x[n-1]^2. \tag{3.7}$$

Note que esse último cálculo só precisa ser realizado uma única vez por transformada. Tudo o que precisa ser calculado à freqüência do sinal de entrada é a atualização

$$x[n] = 2\cos\omega\cdot x[n-1] - x[n-2] + u[n]. \tag{3.8}$$

Na nossa implementação, registradores $x1$ e $x2$ representam os últimos valores de x, β guarda a constante $\cos\omega$, entrada u corresponde à saída do conversor A/D, e s representa o shift a ser aplicado para compensar o ganho do PGA, de modo que ficamos com

$$x1 \leftarrow (\beta\cdot x1 \ll 1) - x2 + (u \ll s) \tag{3.9}$$

$$x2 \leftarrow x1 \tag{3.10}$$

onde "$\ll$" representa shift a direita. A inicialização sendo

$$x1 \leftarrow (u \ll s) \tag{3.11}$$

$$x2 \leftarrow 0 \tag{3.12}$$

Observamos ainda que, no produto $\beta\cdot x1$, β representa um número entre -1 e 1 em ponto fixo.

3.4.2 Método Direto com CORDIC

Outra opção seria partir da Equação (3.1) e de fato calcular o produto $e^{-j\omega n}u[n]$.

A maneira mais direta seria usar uma tabela de senos e um multiplicador, mas uma alternativa bem mais conveniente à implementação em hardware é o algoritmo CORDIC, que descrevemos a seguir.

A ideia é decompor rotações no plano complexo em rotações elementares, fáceis de calcular.

Representando um número complexo como um vetor em $\mathbb{R}^2$, a multiplicação por $e^{j\theta}$ equivale a aplicar a matriz de rotação

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \cos\theta \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix} \tag{3.13}$$

($\cos\theta \neq 0$). Para $\theta = \alpha_i := \arctan(2^{-i})$, $i \in \mathbb{Z}^+$, a matriz fica

$$R_\theta = \cos\theta \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix}, \tag{3.14}$$

de modo que, esquecendo por hora o fator $\cos\theta$, sua aplicação só envolve, além de adições, produto por potências de 2, o que significa simplesmente operações de deslocamento nas representações binárias das coordenadas. A chave para o algoritmo está em escrever um ângulo $\theta \in [-\frac{\pi}{4}; \frac{\pi}{4}]$ como

$$\theta = \sum_{i=1}^{N} \sigma_i \alpha_i + \varepsilon_N, \tag{3.15}$$

com $\sigma_i = \pm 1$ e erro $|\varepsilon_N| < \alpha_N$.

Para ver que isso é sempre possível, basta proceder por indução, observando que $\alpha_{i+1} < \alpha_i < 2\alpha_{i+1}$.

O nosso algoritmo fica:

3.5 Taxa de Amostragem

A taxa de amostragem é um aspecto importante do processamento de sinais ruidosos. Na ausência de ruído, é suficiente amostrar à taxa de Nyquist. No entanto, na presença de ruído, pode ser importante buscar taxas de amostragem maiores a fim de obter medidas mais precisas.

Os ganhos com aumento da taxa não são ilimitados, no entanto. Apesar de o sinal ser contínuo, à medida que aumenta a taxa de amostragem, aumenta também a correlação entre o ruído de amostras vizinhas, até que o aumento das amostras se torna irrelevante.

Vejamos um exemplo numérico.

3.5.1 Freqüência Zero

Considere um processo gaussiano estacionário de variância unitária e densidade espectral

$$\alpha(f) = \frac{a}{2} \exp(-a|f|), \tag{3.16}$$

como em (5). A função de autocorrelação pode ser obtida pela transformada inversa de Fourier:

$$\gamma(\tau) = \frac{1}{1 + (2\pi\tau/a)^2}. \tag{3.17}$$

Suponha que tentemos estimar a média do sinal num intervalo $[0; T]$ a partir de amostras nos instantes $\frac{k}{N}T$, $k \in \{0, \ldots, N-1\}$, $N \in \mathbb{N}$. A variância da média será

$$V = \frac{1}{N^2} \sum_{0 \le i,j < N} \gamma\left(\frac{T}{N}(i-j)\right) \tag{3.18}$$

$$= \frac{1}{N^2} \sum_{d=-N}^{N} (N - |d|) \cdot \gamma\left(\frac{T}{N}d\right) \tag{3.19}$$

$$= \frac{1}{N} \cdot \gamma(0) + 2\sum_{r=1}^{N} \frac{1 - r/N}{N} \cdot \gamma\left(\frac{T}{N}r\right). \tag{3.20}$$

Escrevendo $\Delta x = \frac{1}{N}$, obtemos

$$V = \frac{1}{N} + 2\sum_{r=1}^{N} (1 - r\Delta x) \cdot \gamma\left(Tr\Delta x\right) \Delta x. \tag{3.21}$$

A partir daqui consideramos duas coisas. Primeiro, o somatório pode ser considerado uma soma de Riemann, de modo que, quando N vai para infinito, V converge para a integral

$$I(T/a) = \int_0^1 (1-x)\gamma(Tx)\,\mathrm{d}x \tag{3.22}$$

$$= \int_0^1 \frac{1-x}{1 - (2\pi Tx/a)^2}\,\mathrm{d}x \tag{3.23}$$

$$= \frac{1}{2\pi T/a}\arctan(2\pi T/a) - \frac{1}{(2\pi T/a)^2}\log(1 + (2\pi T/a)^2) \tag{3.24}$$

$$\approx \frac{1}{4} \cdot \frac{1}{T/a} \qquad (T/a \text{ grande}). \tag{3.25}$$

Segundo, como o integrando é decrescente, a integral é na verdade uma cota superior para o somatório, de modo que podemos estimar

$$V < 2I(T/a) + \frac{1}{N}. \tag{3.26}$$

Desta estimativa, concluímos que, fixado T, o melhor que podemos fazer é garantir $\frac{1}{N} \ll 2I(T/a)$ ($\approx \frac{1}{2}\frac{a}{T}$). A partir daí, não há grandes ganhos em aumentar N.

Em outras variáveis: Fixado o *baud rate* $b = \frac{1}{T}$, o melhor que podemos fazer é garantir que a taxa de amostragem $f_s = \frac{N}{T}$ seja razoavelmente maior que $\frac{2}{a}$. No caso do valor encontrado em (5), isso corresponde a cerca de 170 kHz. Uma taxa maior que essa só melhorará a relação sinal-ruído em menos de 3 dB.

3.5.2 Freqüência Positiva

Claro, não estamos interessados na média do sinal, portanto estudamos a média do ruído multiplicado por um sinal da forma $\cos(2\pi ft + \varphi)$. Ainda considerando a variância cons-

tante e unitária ao longo do intervalo de interesse, a variância da média será

$$V = \frac{1}{N^2} \sum_{0 \le i,j < N} \gamma\left(\frac{T}{N}(i-j)\right) \cos\left(2\pi f \frac{T}{N} i + \varphi\right) \cos\left(2\pi f \frac{T}{N} j + \varphi\right).$$

Denotando por $\omega = 2\pi f \frac{T}{N}$ a freqüência discreta, a expressão acima pode ser reescrita como

$$V = \frac{1}{N^2} \sum_{0 \le i,j < N} \gamma\left(\frac{T}{N}(i-j)\right) \cos\left(\omega i + \varphi\right) \cos\left(\omega j + \varphi\right). \tag{3.27}$$

Usando a identidade $\cos(a)\cos(b) = \frac{1}{2}(\cos(a+b) + \cos(a-b))$, e a soma discreta do cosseno, podemos reduzir a soma dupla acima a

$$V = V_1 + V_2 \cos(2(\varphi - \varphi_0)), \tag{3.28}$$

onde $\varphi_0 = \frac{1}{2}\omega(1-N)$, $V_1 = V_{a1} + V_{b1}$ e $V_2 = V_{a2} + V_{b2}$, com

$$V_{a1} = \frac{1}{2N} \tag{3.29}$$

$$V_{b1} = \frac{1}{N^2} \sum_{r=1}^{N-1} (N-r) \cdot \gamma\left(\frac{T}{N} r\right) \cos(\omega r) \tag{3.30}$$

$$V_{a2} = \frac{1}{2N^2} \csc(\omega) \sin(N\omega) \tag{3.31}$$

$$V_{b2} = \frac{1}{N^2} \csc(\omega) \sum_{r=1}^{N-1} \gamma\left(\frac{T}{N} r\right) \sin(\omega(N-r)) \tag{3.32}$$

Como ω depende de N, é interessante reescrever as equações acima em termos dos parâmetros originais:

$$V_{a1} = \frac{1}{2N} \tag{3.33}$$

$$V_{b1} = \frac{1}{N^2} \sum_{r=1}^{N-1} (N-r) \cdot \gamma\left(\frac{T}{N} r\right) \cos\left(2\pi f \frac{T}{N} r\right) \tag{3.34}$$

$$V_{a2} = \frac{1}{2N^2} \csc\left(2\pi f \frac{T}{N}\right) \sin(2\pi f T) \tag{3.35}$$

$$V_{b2} = \frac{1}{N^2} \csc\left(2\pi f \frac{T}{N}\right) \sum_{r=1}^{N-1} \gamma\left(\frac{T}{N} r\right) \sin\left(2\pi f T \left(1 - \frac{r}{N}\right)\right) \tag{3.36}$$

Quando N vai para infinito,

$$V_{a1} \to 0 \tag{3.37}$$

$$V_{b1} \to \int_0^1 (1-x)\gamma(Tx)\cos(2\pi f T x)\,\mathrm{d}x \tag{3.38}$$

$$V_{a2} \to 0 \tag{3.39}$$

$$V_{b2} \to \frac{1}{2\pi f T} \int_0^1 \gamma(Tx)\sin(2\pi f T(1-x))\,\mathrm{d}x \tag{3.40}$$

Em termos da variável tempo $t = Tx$, o limite de V_{b1} pode ser escrito como

$$\widetilde{V_1} = \frac{1}{T} \int_0^T \left(1 - \frac{t}{T}\right) \gamma(t) \cos(2\pi f t)\,\mathrm{d}t \tag{3.41}$$

$$\approx \frac{1}{T} \int_0^\infty \gamma(t) \cos(2\pi f t)\,\mathrm{d}t \tag{3.42}$$

$$= \frac{1}{4} \frac{1}{T/a} \exp(-af) \qquad (T/a \text{ e } fT \text{ grandes}) \tag{3.43}$$

Tabela 2 – Taxa de amostragem de acordo com ruído.

b	f	f_s (3 dB)	$\sqrt{V_2/V_1}$	$\sqrt{\widetilde{V}_2/\widetilde{V}_1}$
360 bps	90 kHz	198 kHz	0.082	0.032
360 bps	122 kHz	253 kHz	0.115	0.030
1800 bps	90 kHz	198 kHz	0.184	0.070
1800 bps	122 kHz	254 kHz	0.241	0.067

(Os termos dependentes de fase são desprezíveis. Variar a fase equivale a mudar no máximo um semiperíodo de uma amostra que contém fT períodos.)

Para decidir que valor de N é suficiente para garantir uma variância próxima à variância média limite $\widetilde{V}_1$ já não podemos recorrer a estimativas tão diretas como no caso de freqüência zero. É mais simples recorrer a um experimento numérico. É o que fizemos.

Para isso, usamos as expressões (3.33) a (3.36). Para a componente dependente da fase, para cada freqüência f_c de interesse, tomamos o cuidado de variar f num intervalo $[f_c; f_c + 2\pi/T]$ (observe a influência de $\sin(2\pi fT)$ no valor de V_2). Comparando os valores obtidos com os valores limites, procuramos a taxa de amostragem f_s que garantiria um máximo de 3 dB de ruído acima do limite.

Usamos como ferramenta o GNU Octave. O *script* utilizado, `sample_rate.m`, está no Anexo B.

A Tabela 2 resume os resultados para freqüências de 90 kHz e 122 kHz e taxas de 360 bps (taxa mínima do padrão IEC 61334-5-1) e 1800 bps. Para o parâmetro a da autocorrelação do ruído, não dispomos de estudos extensivos. Usamos o valor relatado em (5), de 1.2×10^{-5} s.

Observamos que, em todo caso, a taxa de amostragem é pouco maior que a de Nyquist, do que concluímos que o limitante aqui não é o ruído, mas sim o *aliasing*.

O *front-end* analógico (AFE) utilizado no projeto, por exemplo, possui filtros com freqüências de corte 90 kHz e 145 kHz. Se quisermos trabalhar na faixa B da CENELEC, precisamos utilizar o segundo, e usar uma freqüência de corte de pelo menos 290 kHz, já acima do previsto na Tabela 2.

Mesmo com um filtro mais estreito, outra consideração acaba entrando em jogo. Conforme veremos na Seção 3.8, trabalhar em freqüências discretas muito próximas a π nos obriga a usar registradores maiores (e conseqüentemente somadores e multiplicadores maiores), de forma que isso é o bastante para justificar uma freqüência de amostragem maior que as da Tabela 2.

3.6 Detecção de Cabeçalho

A detecção de cabeçalho é baseada numa estatística simples.

Particionamos as últimas 32 amostras do monitor de canal em dois conjuntos χ_1 e χ_0, correspondentes às posições dos bits 1 e 0 no caso de essas amostras corresponderem exatamente ao cabeçalho esperado. Definimos

$$A_{ij} = \sum_{k \in \chi_j} r_i^2[k]. \tag{3.44}$$

A decisão sobre a presença de sinal é baseada na estatística

$$\frac{A_{11}}{A_{10}} + \frac{A_{00}}{A_{01}}, \tag{3.45}$$

a qual comparamos com um limiar configurável T.

Após superar esse limiar, olhamos ainda 31 bits à frente a fim de maximizar a estatística, no que chamamos de detecção de delimitador, para evitar problemas com a correlação entre o cabeçalho esperado e o cabeçalho deslocado.

Sob o nosso modelo, na presença de ruído, essa estatística é a soma de duas variáveis com distribuição F de Fisher-Snedecor, com parâmetros $d_1 = d_2 = 32$ (cada amostra $r_i^2[k]$ corresponde a dois graus de liberdade). Seu valor deve se concentrar em torno de $1 + 1 = 2$. Mais precisamente, em média e desvio-padrão, o resultado é 2.13 ± 0.56.

Já no caso de um casamento perfeito com o cabeçalho, o valor deve se concentrar em torno de $2 + 2\frac{\nu^2}{2\sigma^2}$.

Para escolher um limiar adequado, deve-se considerar o nível de relação sinal-ruído que se deseja suportar e em seguida ponderar os custos da perda de uma mensagem, e da eventual detecção de um falso cabeçalho.

A detecção de uma falsa mensagem não é um evento tão ruim como se pode pensar a princípio. Tipicamente e, em particular, no projeto PLCM, as mensagens são protegidas por um código de detecção de erro (por exemplo, um CRC, *cyclic redundancy check*), de modo que é altamente improvável que uma mensagem aleatória seja interpretada como válida. Talvez o maior custo de uma falsa detecção de cabeçalho seja a possibilidade de perder um cabeçalho real durante a reepção da falsa carga útil.

Observação: Na verdade, para evitar a necessidade de divisões, calculamos

$$A_{01}A_{11} + A_{10}A_{00} > T \cdot A_{01}A_{10}. \tag{3.46}$$

3.7 Cálculo de Verossimilhança

No capítulo anterior, seguindo (2), mostramos como derivar a expressão de verossimilhança a ser empregada pelo receptor ótimo ideal. Como se vê na Equação (2.8), sua implementação envolve o cálculo de uma função transcendente.

No mesmo artigo ((2)), o autor propõe como abordagens de implementação o uso de tabelas (em função das três variáveis envolvidas) e a alternância entre o algoritmo FSK

clássico (comparação dos sinais) e a decisão ASK (comparação de um dos sinais com um limiar, na freqüência de melhor relação sinal-ruído).

Em (3), Veronesi et al. propõem o uso de uma aproximação linear por partes para o cálculo de $\log I_0(\cdot)$. Aproveitamos a ideia desse artigo, com a pequena modificação de aproximar a função $\log I_0(\sqrt{\cdot})$, já que a unidade de verossimilhança recebe os quadrados dos valores absolutos da DTFT. Usamos 17 pares de coeficientes.

Usando a mesma técnica da seção anterior, de escrever uma comparação do tipo $\frac{a}{b} > \frac{c}{d}$ como $ad > bc$ (todas as grandezas positivas), conseguimos evitar a necessidade de divisão na nossa Unidade de Verossimilhança.

3.7.1 Decisão Suave

Para suportar decisão suave, temos apenas que trocar a Equação (2.8) por (2.9). O maior impacto sobre o *hardware* resultante é que não podemos mais evitar o uso de divisões. Ao menos uma por bit é necessária.

3.8 Dimensionamento dos Registradores

Sendo este um projeto de hardware dedicado que efetuará cálculos numéricos, o tamanho dos registradores a serem utilizados são parâmetros que merecem atenção.

Em geral, registradores muito grandes requerem maior complexidade lógica nas operações, implicando maior área, custo e latência. Por outro lado, registradores muito pequenos incorrem em maiores erros de quantização ou em perigo de estouro, afastando a implementação de seu modelo ideal.

Usamos aqui a seguinte notação: Para um registrador α, denotaremos por ε_α sua resolução, a menor magnitude por ele representada, por w_α seu tamanho e por M_α o valor definido por

$$M_\alpha = 2^{w_\alpha}\varepsilon_\alpha. \tag{3.47}$$

Para valores sem sinal, a faixa vai de 0 a $M_\alpha - \varepsilon_\alpha$. Para valores com sinal, vão de $-\frac{1}{2}M_\alpha$ a $\frac{1}{2}M_\alpha - \varepsilon_\alpha$.

3.8.1 Algoritmo de Goertzel

3.8.1.1 Ruído de Quantização

O efeito da resolução ε_x dos registradores $x1$ e $x2$ manifesta-se como um erro de quantização a cada atualização. Modelamos isso como um ruído aleatório uniforme entre $-\varepsilon_x$ e 0 (considerando truncamento do resultado).

Ora, mas o próprio sinal de entrada já inclui um ruído de quantização da ordem de $2^s\varepsilon_u$, de modo que não ganhamos muito escolhendo ε_x muito menor que ε_u.

Por outro lado, escolher ε_x maior que ε_u equivale a descartar os bits menos significativos de u para os valores mais altos de ganho, o que torna esses valores mais altos inúteis, sendo melhor nem exigi-los do PGA.

Considerando isso, escolhemos $\varepsilon_x = \varepsilon_u$.

3.8.1.2 Estouro

No nosso projeto, escolhemos, por simplicidade, não tratar qualquer problema de estouro dos registradores. Em vez disso, queremos dimensioná-los a fim de evitar essa possibilidade.

Para tal, analisamos o valor máximo que precisará ser guardado pelos registradores $x1$ e $x2$, em função do valor máximo do sinal de entrada e do número N de iterações, o que fica fácil se escrevermos $x[N]$ como convolução do sinal de entrada com a resposta ao impulso do filtro:

$$x[N] = \sum_{k=0}^{N-1} h[k]u[N-k] \quad \Rightarrow \tag{3.48}$$

$$|x[N]| \leq \frac{M_u}{2} \sum_{k=0}^{N-1} |h[k]| \tag{3.49}$$

$$= \frac{M_u}{2} \csc\omega \sum_{k=0}^{N-1} |\sin(\omega(k+1))| \tag{3.50}$$

Se quisermos uma estimativa independente de ω, o melhor que podemos fazer é usar $\sin(\omega k) \leq k \sin\omega$ $(k \geq 1)$ para obter

$$|x[N]| \leq \frac{M_u}{2} \sum_{k=1}^{N} k = \binom{N+1}{2} \frac{M_u}{2}, \tag{3.51}$$

que é exatamente a cota para $\omega = 0$.

Para $f_s = 500$ kHz, e baud rate de 360 bps (o menor permitido pelo IEC 61334-5-1), isso exigiria $w_x = w_u + 20$. Esse valor pode parecer exagerado. Podemos melhorá-lo se limitarmos ω. É o que faremos a seguir. Outras opções são consideradas posteriormente.

Estamos interessados em valores de ω que produzam alguns períodos de $\cos(\omega t)$ no intervalo do somatório em (3.50). Pensando assim, decompomo-lo em intervalos onde $\sin(\omega(k+1))$ não muda de sinal. Em cada um deles, o resultado pode ser escrito com

$$\sum_{k=0}^{n-1} \sin(\omega k + \varphi) = \sum_{k=0}^{n-1} \frac{\cos(\omega(k-\frac{1}{2}) + \varphi) - \cos(\omega(k+\frac{1}{2}) + \varphi)}{2\sin\frac{\omega}{2}} \tag{3.52}$$

$$= \frac{\cos(-\frac{\omega}{2} + \varphi) - \cos(\omega(n+\frac{1}{2}) + \varphi)}{2\sin\frac{\omega}{2}} \tag{3.53}$$

$$\leq \csc\frac{\omega}{2}. \tag{3.54}$$

Contando o total de semiperíodos da função real $\sin(\omega t)$ no intervalo de 0 a $N-1$, concluímos que há no máximo $\left\lceil \frac{\omega N}{\pi} \right\rceil = \lceil 2fT \rceil$ de parcelas como acima, com o que podemos

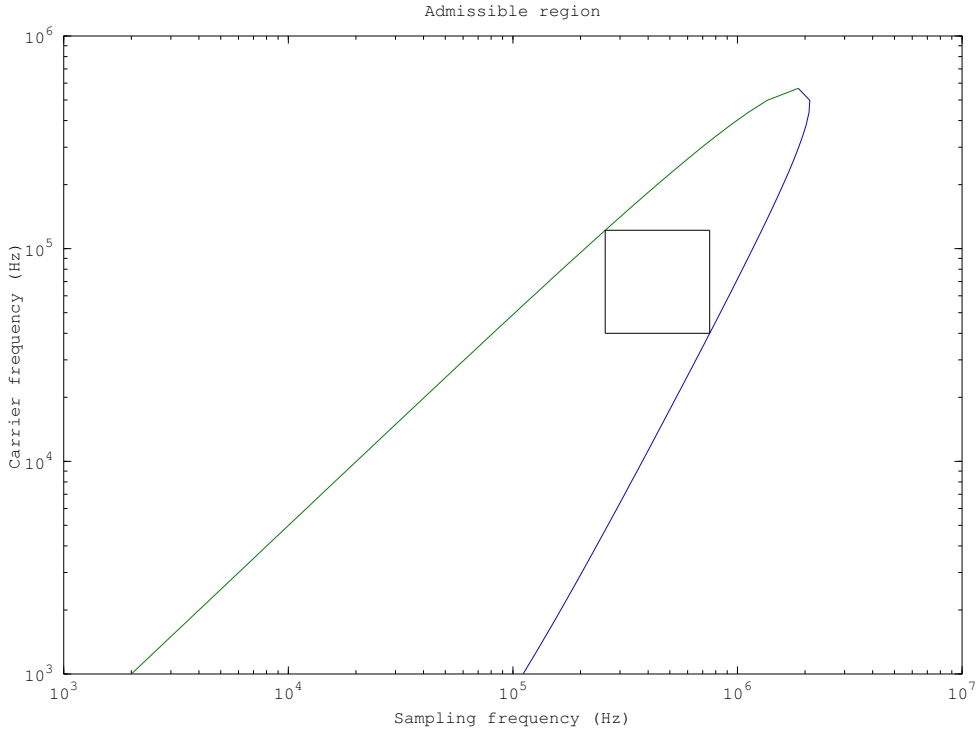

Figura 6 – Região admissível para 12 bits extras, a 360 bps.

finalmente estimar

$$|x[N]| \leq \lceil 2fT \rceil \, csc\frac{\omega}{2} \csc\omega \cdot \frac{M_u}{2}. \tag{3.55}$$

Observamos que esse valor diverge tanto para $\omega \to 0$ como para $\omega \to \pi$. Queremos portanto limitar a faixa de valores de f para uma dada freqüência f_s de amostragem. Consideramos aqui o suporte à faixa de 91 kHz a 122 kHz.Além disso, queremos permitir alguma liberdade na taxa de amostragem, para eventuais experimentações. Decidimos acrescentar uma oitava de folga.

Sob essas restrições, conseguimos fazer $|x[N]| \leq 2^{11.1}\frac{M_u}{2}$, de modo que é suficiente ter

$$w_x = w_u + 12, \tag{3.56}$$

uma economia de 8 bits em relação à estimativa anterior[1].

A Figura 6 mostra a resultante região admissível no plano f versus f_s. Em destaque o retângulo correspondente a

$$\begin{aligned} 258 \text{ kHz} \leq f_s \leq 752 \text{ kHz} \\ 91 \text{ kHz} \leq f \leq 122 \text{ kHz} \end{aligned} \tag{3.57}$$

O *script* GNU Octave `goertzel_overflow_bits_design.m` foi utilizado para resolver a restrição. Ele utiliza a função em `goertzel_overflow_bits.m`, correspondente à estimativa (3.55). Finalmente, a função em `goertzel_beta_limits.m` foi escrita para indicar

[1]Notamos que, apenas para suportar a freqüência máxima da CENELEC B, já precisaríamos de $w_x = w_u + 10$.

os valores de $\beta = \cos\omega$ aceitáveis em função de $\log_2 \frac{M_x}{M_u}$, T e f_s, como demonstrado em `goertzel_beta_limits_demo.m`.

Note que, como o efeito de um estouro é extremo, todo o cuidado é preciso para garantir que os parâmetros são limites estritos, não apenas aproximações. Assim, por exemplo, T deve representar o maior valor esperado para o bit slot, incluindo o *jitter* da detecção de zero.

3.8.1.3 Alternativa

Outra estratégia possível seria decompor o cálculo da DTFT em passos menores. A cada passo, recuperaríamos as partes real e imaginária de v, e resetaríamos o filtro. As partes real e imaginária das parciais de v seriam acumuladas. Como os passos seriam menores, teríamos menores valores de n. As parciais de v poderiam ser somadas mais lentamente.

Não escolhemos essa estratégia a princípio, mas ela poderia ser interessante se estivéssemos interessados em taxas de amostragem mais altas. Sobre essa taxa, ver Seção 3.5.

3.8.1.4 Freqüência

A freqüência do filtro é, como vimos na Seção 3.4.1, determinada pelo parâmetro $\beta = \cos\omega$. No nosso projeto, esse parâmetro é uma entrada do módulo (um β para cada canal). Pela natureza do parâmetro, temos $M_\beta = 1-(-1) = 2$. Fica a questão de escolher ε_β.

Um tamanho menor para o registrador β simplificaria o multiplicador necessário ao custo de reduzir a resolução da configuração de freqüência do módulo. Analisaremos então qual valor de ε_β é necessário para garantir uma resolução em freqüência satisfatória.

A resolução em freqüência do demodulador pode ser avaliada assim: Dado um $\beta = \cos\omega$ escolhido, ao variar β para $\beta' = \beta + \varepsilon_\beta$, ω varia para $\omega' \approx \omega - \csc\omega \cdot \varepsilon_\beta$.

Para determinar que valor é satisfatório, baseamo-nos na faixa de passagem do nosso filtro.

Para encontrar essa faixa, estudamos a "resposta em freqüência" da DTFT, calculada ao longo de um intervalo finito. Para isso, consideramos o valor absoluto da DTFT de uma exponencial complexa de freqüência ω calculada na freqüência ω_0 com N amostras:

$$\left| \sum_{k=0}^{N-1} e^{j\omega k} e^{-j\omega_0 k} \right| = \frac{\sin\left(N\Delta\omega/2\right)}{\sin\left(\Delta\omega/2\right)}, \tag{3.58}$$

$\Delta\omega = \omega - \omega_0$.

Para $\Delta\omega = 0$, o somatório acima é evidentemente N. Queremos pois achar $\Delta\omega$ para o qual a expressão acima valha $\frac{N}{\sqrt{2}}$.

Para isso, observamos que, para $\Delta\omega$ pequeno, podemos aproximá-la por $\frac{\sin(N\Delta\omega/2)}{\Delta\omega/2}$, e portanto basta resolver

$$\frac{\sin\left(N\Delta\omega/2\right)}{N\Delta\omega/2} = \frac{1}{\sqrt{2}}, \tag{3.59}$$

o que dá

$$\Delta\omega = \frac{2\Delta\omega_c}{N}, \tag{3.60}$$

$\Delta\omega_c \approx 0.443\pi \approx 1.39$.

Com isso, se quisermos ter uma resolução de pelo menos metade da faixa de passagem, precisamos garantir que

$$\csc\omega \cdot \varepsilon_\beta \leq \Delta\omega \qquad \Leftrightarrow \tag{3.61}$$

$$\varepsilon_\beta \leq \frac{2\Delta\omega_c}{N \csc\omega}. \tag{3.62}$$

No retângulo da Figura 6, o valor máximo de $N \csc\omega$, com *baud rate* de 360 bps, é cerca de 6368, correspondente ao canto superior esquerdo. Se nos restringirmos a esse retângulo, portanto, basta que $\varepsilon_\beta \leq \frac{1}{2290}$. É então suficiente fazer $w_\beta = 13$.

3.8.1.5 Sinal de entrada

No nosso projeto, o registrador u deve ser capaz de guardar a saída do conversor A/D deslocada de um número s de bits de acordo com o ganho do PGA. Seu tamanho deve ser portanto a soma do número de bits lidos da saída do ADC com o valor máximo de s, $s_{\max} = \log_2 \frac{G_{\max}}{G_{\min}}$. Como vimos na Seção 3.8.1.2, seu tamanho afeta o tamanho de $x1$ e $x2$, e portanto a complexidade do multiplicador.

O ruído de quantização não precisa ser muito menor do que o ruído do sinal que esperamos tolerar. Como as relações sinal-ruído esperadas para um modem PLC são razoavelmente baixas, isso sugere o uso de um A/D com baixa resolução.

Por outro lado, para aproveitar os benefícios da independência de canais da modulação S-FSK, precisamos estar preparados para receber um sinal muito ruidoso que tenha, contudo, uma boa relação sinal-ruído em pelo menos um dos canais. A informação pode estar nos bits menos significativos do sinal não-filtrado.

Escolhemos aqui projetar pelo excesso: Como o ADC utilizado é de 10 bits, e o AFE oferece dois PGAs com ganhos de $\frac{1}{4}$ a 2, e de 1 a 64, respectivamente, resolvemos utilizar todos os bits do ADC e permitir s variar de 0 a $2^3 - 1 = 7$, segundo uma entrada de três bits. Com isso $w_u = 10 + 7 = 17$.

Deixamos para estudos, simulações e experimentos posteriores decidirem sobre o efeito de uma redução da resolução do A/D.

3.8.2 Valor Absoluto da DTFT

O quadrado do valor absoluto da DTFT do sinal de entrada é a saída dos módulos de Goertzel e entrada da Unidade de Verossimilhança. Ele é, como vimos, dado pela expressão (3.7). Se quiséssemos representar o resultado exatamente para todos os valores

possíveis de $x1$, $x2$ e β, precisaríamos de

$$\log_2 \frac{M_x^2}{2\varepsilon_\beta \varepsilon_x^2} = 2w_x + w_\beta - 2 \tag{3.63}$$

bits. De acordo com os valores projetados acima isso corresponde a 69 bits!

Evidentemente, não estamos interessados em todos esses bits. Sabemos que os bits menos significativos são pouco relevantes. O interessante é que os mais significativos também são irrelevantes, ao contrário do que o nome sugere!

3.8.2.1 Estouro

O cálculo acima se baseia na estimativa de que o valor máximo para o resultado é M_x^2, no caso em que $x1 = x2 = -\frac{M_x}{2}$ e $\beta = -1$. Contudo, os valores de $x1$ e $x2$ não são independentes, e podemos obter estimativas melhores.

Para isso, olhamos diretamente para a expressão da DTFT:

$$y[N] = \sum_{k=0}^{N-1} e^{-j\omega k} u[k]. \tag{3.64}$$

Sem restringir ω, segue trivialmente que

$$y[N] \leq N \cdot \frac{M_u}{2}, \tag{3.65}$$

o que é bem menor que a cota em (3.51), e já é menor que mesmo a cota em (3.55). Restringindo-nos ao intervalo dado por (3.57), com baud rate mínimo de 360 bps, temos $N < 2090$, daí segue que

$$R < 2^{20.06} M_u^2, \tag{3.66}$$

enquanto $M_x^2 = 2^{24} M_u^2$. Isso já prova que os supostos três bits mais significativos são sempre zero, e que podemos ignorá-los.

Podemos ignorar mais alguns bits se restringirmos ω. Nesse caso, podemos provar que, para $\sin \omega \neq 0$, a DTFT do sinal está limitada por

$$y[N] \leq \lceil 2fT \rceil \csc \frac{\omega}{2} \cdot \frac{M_u}{2}. \tag{3.67}$$

No retângulo de interesse, essa expressão é maximizada no canto superior direito. Substituindo $f = 122$ kHz, $f_s = 752$ kHz, e $T = \frac{1}{360}$ s obtemos $y[N] < 1390 \frac{M_u}{2}$, seguindo que

$$R < 2^{18.89} M_u^2, \tag{3.68}$$

o que nos permite ignorar mais um par de bits. Ao fazer isso, estamos restringindo ainda mais a região admissível do plano $f \times f_s$. A Figura 7 mostra a nova região, em comparação à anterior. O importante é que o retângulo permanece, por projeto, intacto.

Não precisamos, no entanto, nos preocupar com todo o retângulo. Lembre-se que na Seção 3.8.1.2, estabelecemos como meta um retângulo de altura mínima de uma oitava.

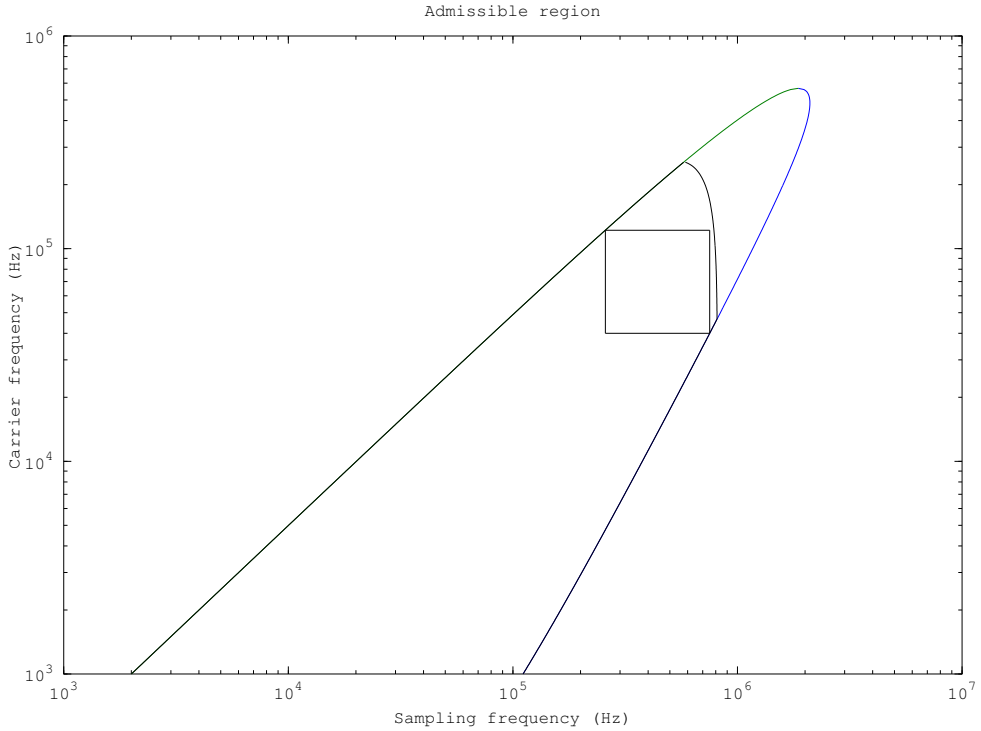

Figura 7 – Região admissível a 360 bps para $M_R = 2^{19} M_u^2$.

Isso nos permite reduzir a freqüência máxima de amostragem a até 516 kHz, o que resulta no enxugamento de mais um bit.

A Figura 8 mostra a região final para *baud rate* de 360 bps, com $M_R = 2^{18} M_u^2$. O novo retângulo corresponde a

$$\begin{aligned} 258 \text{ kHz} \leq f_s \leq 528 \text{ kHz} \\ 91 \text{ kHz} \leq f \leq 122 \text{ kHz} \end{aligned} \tag{3.69}$$

3.8.2.2 Resolução

Para decidir sobre ε_R, examinamos quais de seus bits são estatisticamente significativos. Para isso, lembramos que o sinal de entrada possui um erro de quantização mínimo da ordem de ε_u.

Modelando o ruído de quantização como variáveis aleatórias independentes e identicamente distribuídas com variância $\sigma_u^2 \sim \varepsilon_u^2$, a distribuição da DTFT resultante será aproximadamente uma normal bivariada (no plano complexo). A distribuição marginal ao longo de uma reta de fase φ terá variância

$$\sigma^2 = \sum_{k=0}^{N-1} \cos^2(\omega k + \varphi) \sigma_u^2, \tag{3.70}$$

o que dá em média $\sigma^2 = \frac{1}{2} N \sigma_u^2$.

O módulo da DTFT terá então uma distribuição de Rayleigh. A variância do quadrado do módulo será, nesse caso,

$$\sigma_R^2 \geq 4\sigma^4 = N^2 \sigma_u^4. \tag{3.71}$$

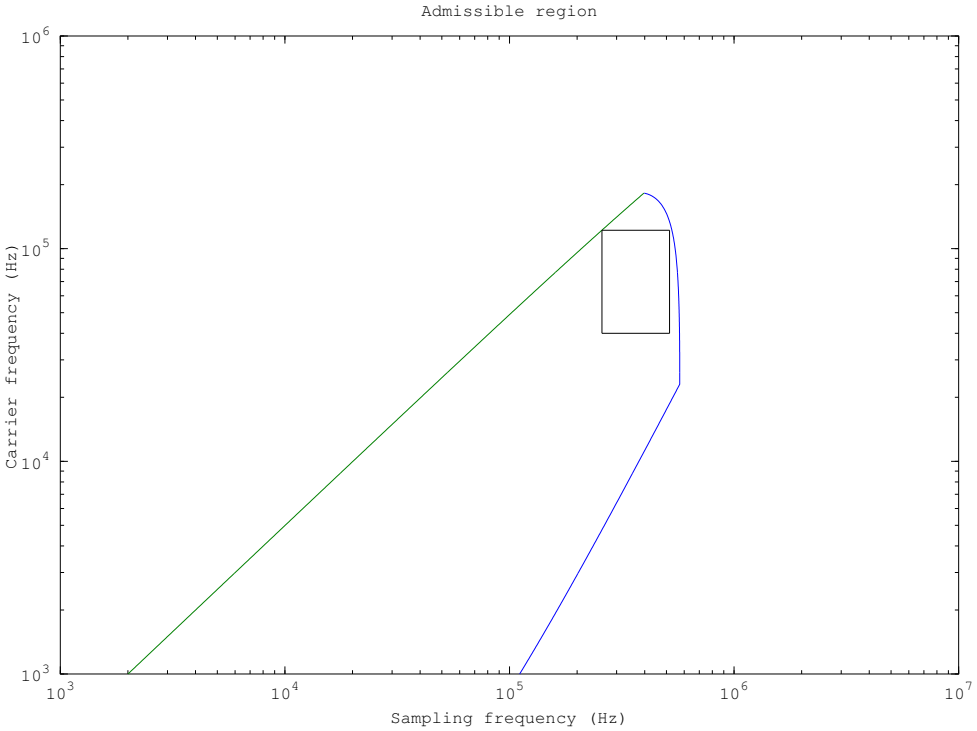

Figura 8 – Região admissível final, a 360 bps.

Assim, não faz sentido fazer ε_R muito menor que $N\varepsilon_u^2$.

Considerando $N \geq 512$ (na nossa região de interesse para *baud rate* de 360 bps, $N \geq 716$), isso sugere $\varepsilon_R = 2^9\varepsilon_u^2$. Fazendo essa escolha, e lembrando que $M_R = 2^{18}M_u^2$, ficamos com $w_R = 2w_u + 18 - 9 = 2w_u + 9 = 43$, um valor ainda exagerado.

Examinemos de outro modo: Qual a resolução de que precisamos para, quando possível, tomar uma decisão correta sobre qual bit está sendo enviado durante uma transmissão?

Essencialmente, precisamos ser capazes de medir o menor nível de sinal que pretendemos suportar, e, além disso, medir o maior nível de ruído correspondente a esse sinal que nos dê alguma chance de decidir qual bit está sendo transmitido.

Como menor nível de sinal a ser suportado, consideramos um sinal cuja amplitude ocupa toda a faixa dinâmica do A/D quando o PGA está programado com ganho mínimo, ou seja, um sinal de amplitude $\frac{M_{\mathrm{A/D}}}{2}$. Este sinal corresponde a $R = \frac{N^2}{4}\frac{M_{\mathrm{A/D}}^2}{4}$.

No pior caso, estaremos dependendo de ASK sobre um canal, e precisamos ter uma resolução compatível com o ruído máximo tolerável. Examinando a Figura 4 em (2), que mostra a taxa de erro de bits para ASK e S-FSK com diferentes relações sinal-ruído, observamos que com uma relação sinal-ruído de 15 dB, uma taxa de erro de bit de 10^{-3} é alcançada com ASK. Consideramos essa taxa segura, mesmo para os esquemas mais simples de controle de erro[2].

[2]Na ausência de qualquer código corretor de erros, a taxa de erro para pacotes de 304 bits é de 26%. Usando um código de Hamming(7,4) como atualmente planejado no projeto, a taxa de erro de pacote é inferior a 0.1%.

Com isso, estabelecemos que é mais que suficiente ter

$$\varepsilon_R \leq 10^{-1.5} \cdot \frac{N^2}{4} \frac{M_{\mathrm{A/D}}^2}{4}. \tag{3.72}$$

Considerando N mínimo de 716 e lembrando que $M_{\mathrm{A/D}} = 2^{10}\varepsilon_u$, obtemos ser suficiente ter $\varepsilon_R \leq 2^{29.99}\varepsilon_u^2$. Fazendo $\varepsilon_R = 2^{30}\varepsilon_u^2$ não perdemos muita coisa.

Com estas considerações, obtemos finalmente

$$w_R = 2w_u + 18 - 30 = 22. \tag{3.73}$$

3.8.3 Unidade de Verossimilhança

Para a unidade de verossimilhança, concluímos que o uso de ponto-fixo é inviável, se realmente queremos lidar com um amplo espectro de relações sinal-ruído com precisão adequada.

Especificamos então um ponto-flutuante de complexidade mínima. A representação não envolve offset do expoente, bit escondido ou valores especiais. Zero é representado com mantissa nula e expoente mínimo, para simplificação da lógica de adição.

Como a aproximação linear adotada nos dá uma precisão da ordem de 10^{-3}, a mantissa adotada é de 10 bits.

O arquivo `lu_float.svh` contém a modelagem do ponto flutuante especificado.

4 Simulação e Resultados

Para validação das técnicas propostas, um modelo do *hardware* foi escrito e simulações foram executadas.

O mesmo modelo usado para validação será o modelo de referência usado na verificação do código RTL (*Register Transfer Level*) do *hardware*, a ser escrito em SystemVerilog. Os estímulos utilizados, e a infraestrutura do ambiente montado também devem ser reaproveitados.

Tendo isso em vista, o ambiente de simulação foi escrito em SystemVerilog e é baseado em UVM (6) (*Universal Verification Methodology*). A interface do modelo utiliza TLM-1.0 (*Transaction Level Modeling*) e a geração de estímulos é baseada em seqüências UVM (`uvm_sequence`).

4.1 O Modelo

O modelo é composto de exatamente dois blocos: o Monitor de Canal (CM) e a Unidade de Verossimilhança (LU), conforme a arquitetura geral descrita na Seção 3.3. A aritmética é modelada exatamente como especificado. Num `testbench` completo, os valores produzidos pelo *hardware* devem corresponder exatamente aos do modelo. Apenas o *timing* não é exato.

4.2 Simulações

Foram montados então três ambientes de simulação: um para o Monitor de Canal, um para a Unidade de Verossimilhança e um para o demodulador completo.

Essa separação é conveniente não só para isolar os problemas a serem encontrados durante a verificação funcional, mas também porque, uma vez que cada amostra de DTFT usada pela Unidade de Verossimilhança corresponde a centenas de amostras de sinal processadas pelo Monitor de Canal, podemos gerar muito mais casos num mesmo intervalo de tempo ao simularmos só a Unidade de Verossimilhança em vez de o demodulador completo.

Para a geração de estímulos, partimos das transações (definidas em equipe no projeto PLCM) e elaboramos *seqüências*. Essas seqüências são padrões (algoritmos) de geração de transações. Seguindo as metodologias modernas de verificação, essas seqüências são, por padrão, (pseudo)aleatórias. Acrescentamos então restrições de modo a levar o sistema a estados interessantes.

No caso das entradas de configuração, simplesmente geramos valores razoáveis no início da simulação, e não os alteramos.

Para a entrada de sinal do Monitor de Canal, simplesmente geramos ruído branco.

Já para a entrada da Unidade de Verossimilhança, desenvolvemos uma extensa biblioteca de seqüências para levá-la a seus diferentes estados de operação. Essas seqüências são *estratificadas*, isto é, algumas seqüências são definidas em termo de outras, com seqüências de mais alto nível chamando seqüências de menor nível, até chegar à geração das transações individuais.

No teste básico de validação, a seqüência executada sorteia sucessivamente se produzirá a seguir um trecho de ruído ou um pacote completo. Em cada caso, são sorteados então os parâmetros de nível de ruído e de sinal (se houver) em cada freqüência de modulação. No caso de transmissão de pacote, são gerados o cabeçalho pré-definido, seguido de uma carga útil de conteúdo aleatório e inclusive o trecho de 24 bits de pausa que a deve suceder. Para cada bit da transmissão, sorteamos valores da DTFT de acordo com o dado a que corresponde e os níveis de sinal e ruído efetivos no momento.

Por simplicidade, as distribuições de probabilidade são uniformes, em vez de de Rice. Sob essas condições, o nosso modelo é na verdade sub-ótimo. Os resultados dão uma ideia de como desvios do comportamento ideal podem afetar nosso modelo.

Todos os testes são encerrados quando algum tipo de critério de cobertura é atingido. Para os nossos propósitos, o critério é apenas que um número determinado de transações *de saída* seja produzido.

4.3 Procedimentos de Validação

Os testes aqui relatados foram executados com a ferramenta VCS, e os resultados visualizados com o *software* DVE, ambos da Synopsys, versão G-2012.09.

4.3.1 Monitor de Canal

Executando em modo de depuração, apenas conferimos passo a passo a correta implementação do Algoritmo de Goertzel.

4.3.2 Unidade de Verossimilhança

Para a Unidade de Verossimilhança, conferimos primeiro o funcionamento da detecção de cabeçalho, observando que as transações de saída ocorreram nas posições esperadas.

Para facilitar a validação do algoritmo de decisão, extendemos o teste básico de modo a adicionar uma restrição na geração das mensagens, de modo que a carga útil seguisse um padrão pré-definido. Isso é feito facilmente na infra-estrutura UVM graças ao uso sistemá-

tico do padrão de programação orientada a objetos conhecido como *Fábrica*. Observamos então que a mensagem recebida correspondia ao padrão previsto.

4.3.3 Integração

Checamos a integração entre as duas partes do modelo apenas observando a correta comunicação entre eles. Como não escrevemos seqüências para geração de sinais modulados, não pudemos observar transações de saída na simulação do modelo completo num intervalo de tempo razoável. Conforme discutimos na Seção 5.1.1, o plano é usar um modelo para o modulador, desenvolvido por outros membros da equipe do PLCM para a validação do modelo completo.

5 Considerações Finais

Partindo dos princípios teóricos estabelecidos em (2), e de ideias em (3), procedemos ao projeto completo de um *hardware* dedicado viável que realiza uma aproximação do receptor ideal proposto. Apenas na elaboração de um projeto concreto é que pudemos entender melhor as possibilidades e limitações das abordagens adotadas.

Nas seções restantes, discutimos direções futuras. Primeiro, nos passos seguintes da iteração atual do projeto. Em seguida, consideramos possibilidades de avanço para iterações futuras, e abordagens alternativas.

5.1 Próximos Passos

O projeto aqui proposto e o modelo concomitantemente desenvolvido ainda devem passar por testes em simulação mais extensivos. Os passos seguintes são a implementação em RTL e verificação funcional, a serem executados pela equipe do projeto PLCM.

5.1.1 Testes Futuros

Os testes realizados foram mínimos e têm o objetivo apenas de demonstrar a funcionalidade básica do modelo. Mas a infraestrutura montada será aproveitada para uma investigação mais aprofundada do desempenho do modelo proposto. Em particular, estamos interessados em como ele se compara ao desempenho do receptor ótimo ideal que ele aproxima.

A proposta é montar um ambiente com os modelos do modulador e do demodulador, mediados por um modelo de canal, e investigar características não-funcionais como sensibilidade e especificidade da detecção de cabeçalho e taxa de erro de bits, para diferentes parâmetros de canal.

Com vistas a isso, já implementamos, em GNU Octave, um simulador de ruído baseado no modelo elaborado em (5).

Os arquivos `make_freqs.m`, `make_sigma.m` e `noise_sim.m` contêm as funções básicas para geração de uma amostra de ruído.

O script `demo_noise.m` demonstra seu uso.

5.2 Possibilidades de Avanço

Organizamos esta seção em três partes, de acordo com o nível das mudanças consideradas. Primeiro, discutimos as mudanças fundamentais na estratégia de demodulação, que afetam o modelo ideal do receptor que tentamos aproximar. Em seguida, discutimos as mudanças funcionais, nas táticas de implementação do modelo ideal, que afetam o seu modelo funcional. Finalmente, comentamos mudanças nos parâmetros de projeto.

5.2.1 Mudanças Fundamentais

Na Seção 2.3, tomamos o cuidado de esclarecer sob que condições o receptor proposto é de fato ótimo. Isso nos permite enxergar exatamente onde estão os espaços para avanços fundamentais.

Em resumo, os princípios estabelecidos foram:

1. Podemos ignorar o espectro fora das freqüências de modulação.
2. Podemos ignorar a fase do sinal.
3. As componentes do ruído em cada freqüência de modulação são variáveis aleatórias independentes.
4. Essas variáveis tem distribuição normal e anisotrópica.
5. O nível do sinal em cada freqüência é constante na duração da recepção de um pacote
6. Os parâmetros da distribuição são constantes na duração da recepção de um pacote.

Consideramos que a oportunidade de avanço mais imediata está no item 6, com a estimação da dependência do ruído com a fase da rede, como explicado na Seção 2.3.

Além disso, na disponibilidade de modelos mais sofisticados, mas confiáveis, do ruído, o item 1 também pode ser explorado, como consideramos na mesma seção.

5.2.2 Mudanças Funcionais

Na área de mudanças funcionais, a oportunidade mais clara de alternativa a ser explorada é talvez o uso do algoritmo CORDIC no cálculo da DTFT, como apresentado na seção 3.4.2.

Nesse caso, o *hardware* poderia ser inclusive compartilhado com o módulo de transmissão.

Outra vantagem seria no casamento das freqüências de transmissão e recepção. No projeto atual, o demodulador precisa ser configurado com o cosseno da freqüência digital,

enquanto o modulador provavelmente será configurado diretamente com a freqüência digital (como fração de 2π), de modo que, com precisão finita, não se pode em geral fazer a representação digital de um valor corresponder exatamente à do outro.

5.2.3 Mudanças Paramétricas

Provavelmente a principal contribuição deste trabalho está não no projeto em si, mas no esclarecimento das interdependências entre os diferentes parâmetros de projeto, como taxa de amostragem, taxa de transmissão e tamanhos dos registradores. A teoria documentada, e os *scripts* anexos podem ser usados para explorar diferentes possibilidades sob requisitos mais concretos. Os códigos do modelo e dos ambientes de teste podem ser facilmente reaproveitados.

Referências

1 IEC. International Standard, *Distribution automation using distribution line carrier systems – Part 5-1: Lower layer profiles – The spread frequency shift keying (S-FSK) profile*. [S.l.]: IEC, Geneva, Switzerland, 2001.

2 SCHAUB, T. Spread frequency shift keying. *Communications, IEEE Transactions on*, 1994, v. 42, n. 234, p. 1056–1064, Fevereiro 1994. ISSN 0090-6778.

3 VERONESI, D.; GUERRIERI, L.; BISAGLIA, P. Improved spread frequency shift keying receiver. In: *Power Line Communications and Its Applications (ISPLC), 2010 IEEE International Symposium on*. [S.l.: s.n.], 2010. p. 166–171.

4 NONEMACHER, H. B. et al. Smart grids: Aplicação da tecnologia PRIME-PLC em um sistema real para aplicações em smart metering. In: *XIX Congresso Brasileiro de Automática (CBA), 2012*. [S.l.: s.n.], 2012. p. 1452–1459.

5 KATAYAMA, M.; YAMAZATO, T.; OKADA, H. A mathematical model of noise in narrowband power line communication systems. *Selected Areas in Communications, IEEE Journal on*, 2006, v. 24, n. 7, p. 1267–1276, Julho 2006. ISSN 0733-8716.

6 ACCELERA ORGANIZATION. *Universal Verification Methodology (UVM) 1.1 Class Reference*. [S.l.], Junho 2011.

ANEXO A – Especificação

UFCG / LAD / BRAZIL-IP

Demodulator Specification – Power Line Communication Modem

Revision History

Date	Version	Description	Author
08/09/12	1.0	Initial Release.	Elton Brasil da Costa, André Gomes Medeiros de Souza
09/21/12	2.0	Full revision. Parameters and equations still unspecified.	Felipe Gonçalves Assis
09/28/12	2.1	Moved some internal variables (channel statistics) to output signals.	Felipe Gonçalves Assis
12/07/12	2.2	New signals and variables. List of parameters included, FSM rewritten, equations and procedures partially specified. Likelihood computation still unspecified. Better formatting.	Felipe Gonçalves Assis
04/10/13	2.3	State Diagram. Corrected HEADER_WIDTH and INITIALIZATION_DELAY values. Likelihood computation specification. Coefficients still unspecified.	Felipe Gonçalves Assis
05/20/13	2.4	Bug fixes in likelihood computation. Fix in PGA signals listing.	Felipe Gonçalves Assis
06/11/13	2.5	Bug fix in Goertzel algorithm: Variables were never reset.	Felipe Gonçalves Assis
07/20/13	2.6	Bug fix in parameter estimation (signal levels). Bug fix in header detection.	Felipe Gonçalves Assis

03/02/14	2.7	Goertzel unit parameters finally defined. Unnecessary restriction on baud rate removed from text.	Felipe Gonçalves Assis
04/25/14	2.8	Likelihood Unit with floating point fully specified. DTFT computation fully specified.	Felipe Gonçalves Assis

Authors: André Gomes Medeiros de Souza

Elton Brasil da Costa

Felipe Gonçalves Assis

1. Introduction

The **Demodulator** is responsible for detecting IEC[1] style S-FSK modulated subframes, and outputting likelihood information about each bit of their payload, which can be used by a soft-decision decoder[10].

In order to do that, the Demodulator has to perform signal detection, preamble identification, channel parameters estimation and, finally, the likelihood computation for each bit of each received subframe.

In addition to this, the Demodulator also signals the detection of a carrier in the working channel, supporting the implementation of a collision avoidance protocol[2].

Mark and space frequency[1] parameters are configurable (see Section 3.1). Estimated channel parameters are made available.

The Demodulator then implements the projects' Functional Requirements FR 03 (Monitor a Digital Representation of the Mains Signal) and FR 04 (Sense Carrier), and FR 12 (Produce Channel Statistics), is essential for the implementation of FR 05 (Demodulate and Decode Message) and supports FR 06 (Configure Central Frequency) and FR 14 (Configure Bandwidth)[6].

2. Parameters

2.1. Basic Parameters

Name	Description	Type	Value
INPUT_WIDTH	Width of input data.	Integer	10
SHIFT_WIDTH	Width of gain register.	Integer	3
BETA_WIDTH	Width of beta input	Integer	13
GOERTZEL_WIDTH	Width of Goertzel state variables	Integer	29
GOERTZEL_FRACTIONAL_BITS	Number of fractional bits in Goertzel state variables	Integer	0
DTFT_MSB	Most significant bit of square of absolute value of DTFT passed on to Likelihood Unit.	Integer	51
DTFT_LSB	Least significant bit of square of absolute value of DTFT passed on to Likelihood Unit	Integer	30
OUTPUT_WIDTH	Width of log-likelihood output.	Integer	1
MANTISSA_WIDTH	Width of floating point registers mantissa.	Integer	10
EXPONENT_WIDTH	Width of floating point registers exponent.	Integer	8
LU_TABLE_SIZE	Number of pairs of parameters used in piecewise linear approximation.	Integer	17
HEADER_WIDTH	Width of HEADER.	Integer	32
HEADER	Preamble and Start Subframe Delimiter bit pattern.	Integer	0xAAAA54C7
INITIALIZATION_DELAY	Number of samples from the Goertzel Units processed during the Initialization state.	Integer	HEADER_WIDTH
PAYLOAD_WIDTH	Number of bits in a subframe payload.	Integer	304
PAUSE_DELAY	Number of slots of silence between one transmission and the following one.	Integer	24

2.2. Dependent Parameters

Name	Description	Type	Value
DTFT_WIDTH	Width of Goertzel output.	Integer	DTFT_MSB – DTFT_LSB + 1
STATS_WIDTH	Width of parameter estimation registers.	Integer	DTFT_WIDTH + ceil(log2(HEADER_WIDTH / 2))
FLOAT_WIDTH	Width floating-point registers.	Integer	1 + EXPONENT_WIDTH + MANTISSA_WIDTH

3. Inputs

3.1. Configuration Inputs

These inputs are the reset, and configuration for the mark and space frequency of operation. The configuration is based on the "β-parameters" of the demodulation, here defined as

$$\beta_M = \cos(2\pi f_M T_s)$$
$$\beta_S = \cos(2\pi f_S T_s)$$

These are given in signed fixed-point format with 0 integer bits, except for the sign bit.

The module will accept any given configuration parameters, but the following restrictions must be respected lest the risk of overflow:

$$\lceil 2Tf \rceil csc(2\pi f T_s) csc(\pi f T_s) < 2^{BITS1};$$
$$\lceil 2Tf \rceil \frac{1}{2} csc(\pi f T_s) < 2^{\frac{1}{2} \cdot BITS2}.$$

Here, *T* represents maximum length of bit slot, *f* is the signal frequency (either for mark or space channel), and

BITS1 = (GOERTZEL_WIDTH – GOERTZEL_FRACTIONAL_BITS) – (INPUT_WIDTH + 2^{SHIFT_WIDTH} – 1);
BITS2 = (DTFT_MSB + 1) - 2 * (INPUT_WIDTH + 2^{SHIFT_WIDTH} – 1).

For example, for a baud rate of 360 bps plus a jitter of 1µs, sampling frequency of 500 kHz, BITS1=12 and BITS2=18, a safe interval for β is [-0.265; 0.975], which corresponds to frequencies from 18 kHz up to 146 kHz. More simply, if the sampling frequency $1/T_s$ is in the range [258 kHz; 528 kHz], frequencies from 91 kHz up to 122 kHz are supported.

Additionally, for validity of the assumption of channel independence, [1] recommends that $|f_M - f_S| > 10\,kHz$.

Finally, proper functioning is only guaranteed if configuration parameters are kept constant while reset is low.

Name	Description	Type	Bits	Range
reset	Reset signal.	Bool	1	*
config.beta_mark	Beta parameter for mark frequency (data 1).	Signed Fixed	1.(BETA_WIDTH - 1)	*
config.beta_space	Beta parameter for space frequency (data 0).	Signed Fixed	1.(BETA_WIDTH - 1)	*
config.header_threshold	Threshold parameter for header detection.	Floating Point	FLOAT_WIDTH	(normalized)

Design note: Perhaps it is best for [config.header_threshold] to be an unsigned fixed point. Conversion to internal floating point representation should then be done internally. Our current approach is for absolute flexibility.

3.2. Data Inputs

The synchronization signal [sync.zero_cross] should be a regular pulse train. The pulses should have width of 1 clock and be simultaneous with the zero crossings of the phase of the mains the modem is connected to. This corresponds to the expected slot indicators of the communication[1].

The synchronization signal [sync.pulse] should be a pulse train synchronized with [sync.zero_cross], but with a multiple of three times its frequency. That is, pulses of the latter should coincide with every third pulse of the former. This corresponds to the expected beginning of data bits during a transmission[1].

An auxiliary input, [pga.shift_value], is provided to support gain control. In case the Analog Front End employs a programmable gain amplifier, this input should be such that the selected input gain is

$$G = C \cdot 2^{[\text{pga.shift\_value}]}$$

where C is an arbitrary constant.

In the absence of programmable gain, it should be maintained at a fixed value (also, see configuration output [pga.shift_request]).

Name	Description	Type	Bits	Range
signal_in.valid	High if input signal is valid.	Bool	1	*
signal_in.data	Signal received from the ADC.	Signed Integer	INPUT_WIDTH	*
pga.shift_value	Base 2 logarithm of Analog Front End's input gain plus arbitrary constant.	Integer	SHIFT_WIDTH	*
sync.valid	Active if synchronization signals are reliable.	Bool	1	*
sync.zero_cross	Pulse train corresponding to zero crossings of the mains.	Bool	1	*
sync.pulse	Basic synchronization signal, based on zero detection.	Bool	1	*
dem_dec.ready	Indicates that user is ready to receive output.	Bool	1	*

4. Internal Variables and Functions

Name	Description	Type	Bits	Range	Reset Value
last_shift	Value of pga.shift_value in the previous clock.	Integer	SHIFT_WIDTH	*	10...0b
mark_value	n-th entry is n-th last Goertzel sample for mark channel.	Integer[]	DTFT_WIDTH [HEADER_WIDTH]	*	0
space_value	n-th entry is n-th last Goertzel sample for space channel.	Integer[]	DTFT_WIDTH [HEADER_WIDTH]	*	0
param_a	Linear coefficients for piecewise linear approximation	Floating-point[]	FLOAT_WIDTH [LU_TABLE_SIZE]	(constant)	(constant)
param_b	Independent coefficients for piecewise linear approximation	Floating-point[]	FLOAT_WIDTH [LU_TABLE_SIZE]	(constant)	(constant)

5. Outputs

5.1. Configuration Outputs

Design note: Maybe a [pga.ok_to_change] signal would be desirable.

Name	Description	Type	Bits	Range	Reset Value
pga.shift_request	Requested Analog Front End's input gain level.	Integer	SHIFT_WIDTH	*	0

5.2. Data Outputs

Name	Description	Type	Bits	Range	Reset Value
dem_dec.data	An array with PAYLOAD_WIDTH log-likelihood values, one for each bit of a subframe payload. Most significant bits are received first.	Integer[]	OUTPUT_WIDTH [PAYLOAD_WIDTH]	*	0
dem_dec.valid	Indicates that output data is new and valid.	Bool	1	*	0
dem_mon.state	State of the Demodulator, as detailed in section 8.	Enum	3	{RESET, NOI, WFS, INIT, LFH, LFD, RECEIVE, PAUSE}	RESET
dem_mon.carrier_detected	High when the module infers the presence of a carrier in the channel.	Bool	1	*	0
dem_mon.mark_noise	Estimated noise power for the mark channel.	Integer	STATS_WIDTH	*	0
dem_mon.space_noise	Estimated noise power for the space channel.	Integer	STATS_WIDTH	*	0

dem_mon.mark_signal	Estimated signal power for the mark channel.	Integer	STATS_WIDTH	*	0
dem_mon.space_signal	Estimated signal power for the space channel.	Integer	STATS_WIDTH	*	0
dem_mon.message_discarded	Event flag signaling that a received message was discarded for memory exhaustion.	Bool	1	*	0

6. Restrictions

7. Demodulation Overview

7.1. DTFT and Goertzel Algorithm

Demodulation will be based on the computation of the sampled signal's DTFT. More precisely, we evaluate the energy of the DTFT on the frequencies of interest (f_M and f_S) by means of the *Goertzel Algorithm*[3][4].

7.2. Carrier Sense

Currently unspecified.

Speculative idea: Carrier sense is based on a statistical significance test. More concretely, we compare recent signal levels with a long-term noise average. If the power is significantly above the expected, we indicate the presence of a carrier.

Of course, it is fairly likely that events like this are caused by, e.g., a new noise source. To detect an incoming message, we look for its preamble.

7.3. Preamble Identification

The IEC standard specifies a fixed preamble and start subframe delimiter for every transmitted subframe. These indicate where the payload starts, and finding them ensures us that we are sensing a transmitted message, and not just new noise. It also serves the purpose of estimating signal power in each channel.

After initialization, we are constantly trying to fit the expected preamble to the incoming data. If the fit statistic is good enough, we decide that a message is being received. We then try to maximize the fit statistic over the following sample to avoid confusing the preamble with a shifted one. Finally, we start receiving the payload.

7.4. Bit Synchronization

Bit synchronization is based on zero detection. The transmitted bits are evenly spaced such that zero crossings of the mains correspond to the start of a new bit. We work with a fixed bit rate of three bits between two successive zero crossings, that is, 360 bps for 60 Hz, or 300 bps for 50 Hz.

Messages always start at a zero-crossing. This is a simplified version of the synchronization protocol specified in [1], with no client-server distinction, and that only works for single-phase communication.

Design note: Maybe some kind of fine bit synchronization during preamble detection should receive consideration.

7.5. Maximum Likelihood

After preamble identification, we estimate signal and noise power for mark and space channels. These statistics are used to compute likelihood information for “data 1” and “data 0”. The final output is the difference of these likelihoods for each bit of the payload.

Computation of likelihood values involves some transcendent functions. To keep the digital circuit simple, we employ a piecewise linear approximation, encoded as a look-up table, following the idea of [8].

7.6. Gain Control

Currently unspecified.

8. Pseudo-code

8.1. Subfunctions

8.1.1. Goertzel Units

Design note: As specified, units are expected to work fully in parallel. Some sharing of resources, with a faster clock, is possible. We leave that to the future.

The Goertzel Units implement the Goertzel algorithm. Two instances are constantly working in parallel on the input data. Each unit has one input, [u], two

registers, [x1] and [x2], and a registered output, [value]. After every pulse of [sync.pulse], it performs the following procedure:

Reset [x2] to 0.
Assign [u] to [x1].
For each sample:
    Assign [u] + 2 [beta] [x1] – [x2] to [x1].
    Assign [x1] to [x2].

In the fixed-point assignment, result should be truncated towards negative infinity, which amounts to ignoring the least significant bits. There is no special treatment of overflow: higher order bits of the result should be equally ignored.

Also, after every pulse of [sync.pulse], an estimation of the square of the absolute value of the DTFT should be output to the Likelihood Unit. This value is a specific approximation of the following expression

$[x1]^2 – 2[beta][x1][x2] + [x2]^2$,

where [x1] and [x2] are the last values of the state variables before being set to 0 and [u], respectively.

The approximation should be exactly equal to the result of the following computation:

Let [t1] = 2[beta][x2];
Zero all bits of [t1] of order less than (DTFT_LSB – GOERTZEL_MSB – 2),
call the result [t2];

Let [t3] = [x1] – [t2];
Let [t4] = [x1][t3];
Zero all bits of [t4] of order less than DTFT_LSB – 2,
call the result [t5];

Let [t6] = [x2][x2];
Zero all bits of [t6] of order less than DTFT_LSB – 2,
call the result [t7];

Let [t8] = max{0, [t5] + [t7]};
Discard all bits of [t8] of order less than DTFT_LSB and all bits of order higher than DTFT_MSB,
call the result [value];
Return [value].

Here, GOERTZEL_MSB is defined by

GOERTZEL_MSB = (GOERTZEL_WIDTH – GOERTZEL_FRACTIONAL_BITS – 1).

It is currently unspecified whether this computation is to be done, e.g., fully in parallel, procedurally, or via a pipeline.

Note: The above algorithm should guarantee an error of at most one least significant bit in the given result.

8.1.2. Likelihood Unit

The Likelihood Unit computes a function that approximates the log-likelihood of a Rician model of the input data. The function to be approximated is

$$l(r;\nu,\sigma)=\log \mathrm{I}_0\left(\frac{\nu r}{\sigma^2}\right)-\frac{\nu^2}{2\sigma^2}$$

where

- r^2 is the output of the goertzel module, [value];
- ν^2 is the signal power estimate, [signal];
- $2\sigma^2$ is the noise power estimate, [noise].

The approximated output is denoted by likelihood(). The computation follows an adaptation of the ideas in [8].

Let $X=\frac{\nu^2 r^2}{(2\sigma^2)^2}$ and $e=\left\lfloor \log_2(4\mathrm{X}) \right\rfloor^+$. For $e\leq 16$, we approximate $\log \mathrm{I}_0\sqrt{4\mathrm{X}}$ by $a_e X+b_e$, where the coefficients a_e and b_e are specified in the following Table:

Design note: While currently unspecified, given, e.g., the patterns for mantissas of a_e of even and odd indices, respectively, this table could be compressed.

e	***a_e***	***b_e***
0	919 / 1024 x 2^0	0
1	769 / 1024 x 2^0	599 / 1024 x 2^{-3}
2	638 / 1024 x 2^0	821 / 1024 x 2^{-2}
3	1001 / 1024 x 2^{-1}	963 / 1024 x 2^{-1}
4	753 / 1024 x 2^{-1}	977 / 1024 x 2^0
5	553 / 1024 x 2^{-1}	888 / 1024 x 2^1
6	803 / 1024 x 2^{-2}	748 / 1024 x 2^2
7	577 / 1024 x 2^{-2}	600 / 1024 x 2^3

8	826 / 1024 x 2^{-3}	928 / 1024 x 2^{3}
9	589 / 1024 x 2^{-3}	701 / 1024 x 2^{4}
10	837 / 1024 x 2^{-4}	521 / 1024 x 2^{5}
11	594 / 1024 x 2^{-4}	764 / 1024 x 2^{5}
12	843 / 1024 x 2^{-5}	555 / 1024 x 2^{6}
13	597 / 1024 x 2^{-5}	800 / 1024 x 2^{6}
14	846 / 1024 x 2^{-6}	574 / 1024 x 2^{7}
15	598 / 1024 x 2^{-6}	822 / 1024 x 2^{7}
16	847 / 1024 x 2^{-7}	586 / 1024 x 2^{8}

Let S_i, N_i, R_i, A_i and B_i denote, respectively, the values of v^2, $2\sigma^2$, r^2, a_e and b_e for channel i. The log-likelihood difference is then computed as

$$l_0 - l_1 = \frac{1}{N_0^2}\left[A_0(S_0 R_0) + B_0 N_0^2 - N_0 S_0\right] - \frac{1}{N_1^2}\left[A_1(S_1 R_1) + B_1 N_1^2 - N_1 S_1\right] .$$

For the special case OUTPUT_WIDTH = 1, the computation simplifies to the comparison

$$N_1^2[A_0(S_0 R_0) + B_0 N_0^2 - N_0 S_0] < N_0^2[A_1(S_1 R_1) + B_1 N_1^2 - N_1 S_1] .$$

(Note: The mathematically equivalent

$$(A_0 R_0 - N_0) S_0 N_1^2 - (A_1 R_1 - N_1) S_1 N_0^2 < (B_1 - B_0) N_0^2 N_1^2$$

has fewer products, but needs cancellation analysis.)

Values are first converted to an internal floating-point representation, with a sign bit, EXPONENT_WIDTH exponent bits and MANTISSA_WIDTH mantissa bits. Operation order follows usual algebra rules. Floating-point arithmetic is specified as such:

- Floating-point representation is plain, with no special values or offset.
- Addition/Subtraction truncates towards negative infinity.
- Multiplication truncates towards zero.

(We currently don't use division, so it is not specified)

If $e>16$ for any of the channels, we don't resort to floating-point, using integer arithmetic instead, proceeding as follows:

- If $e_0>e_1$:
  - If $4\mathrm{R}_0 \geq S_0+4\mathrm{N}_0$, output maximum
  - Else, output minimum
- Else if $e_1>e_0$:
  - If $4\mathrm{R}_1 \geq S_1+4\mathrm{N}_1$, output minimum
  - Else, output maximum
- Else:
  - If $4\mathrm{R}_1+S_0+4\mathrm{N}_0 \geq 4\mathrm{R}_0+S_1+4\mathrm{N}_1$, output minimum
  - Else, output maximum

Note: Output is signed, so that, for OUTPUT_WIDTH = 1, maximum is 0 and minimum is 1.

8.2. Main Procedure

The overall operation of the Demodulator may be described in terms of a finite state machine (FSM), sketched in Figure 1. During normal operation, the module will cycle through five main states: Initializing, Looking For Header, Looking For Delimiter, Receiving and Pause, corresponding to the different phases of a message reception.

The next subsections will detail the operations performed in each state, as well as the conditions for leaving them. The states, and their abbreviations are

1. Reset (RESET)
2. No Input (NOI)
3. Waiting for Synchronization (WFS)
4. Initializing (INIT)
5. Looking For Header (LFH)
6. Looking For Delimiter (LFD)

7. Receiving (RECEIVE)

8. Pause (PAUSE)

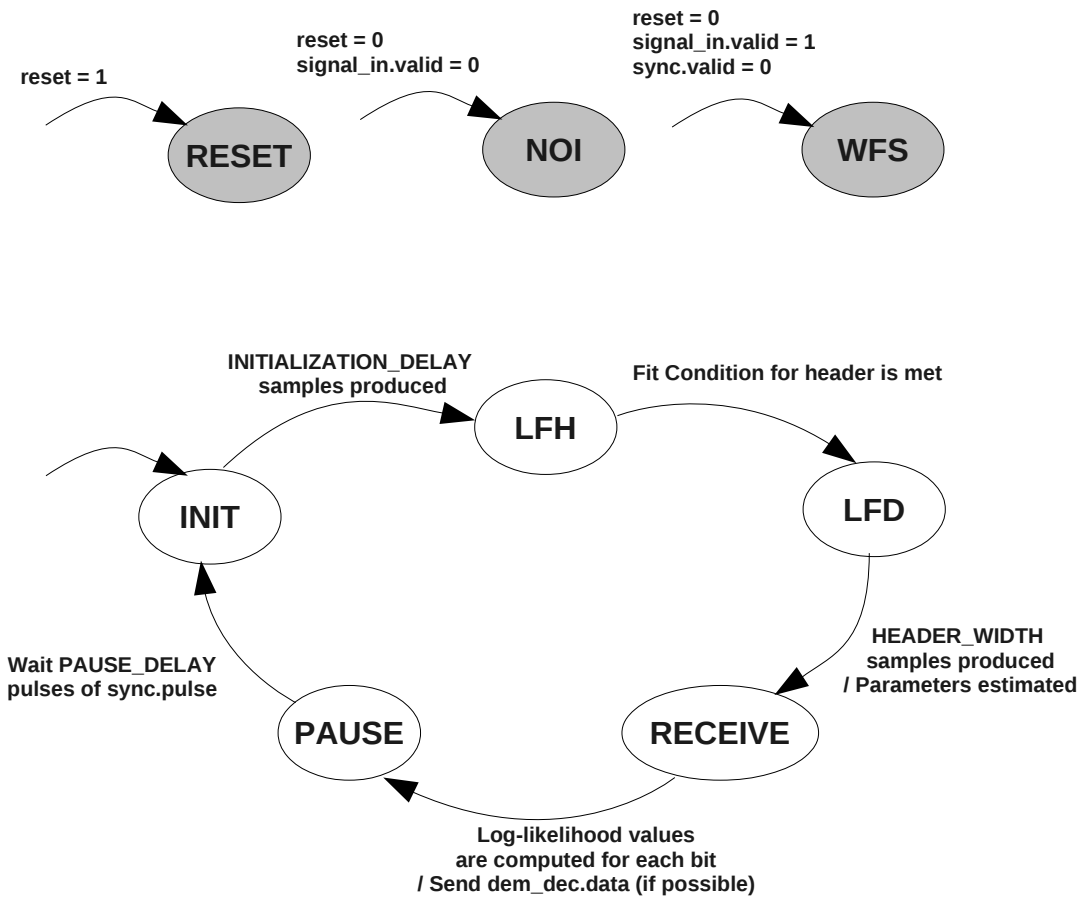

Figure 1: State diagram.

8.2.1. Reset

This state is entered when the reset signal is activated, and left when it is deactivated. All output and internal data is reset to default values. When [reset] is deactivated, if [signal_in.valid] is low, the following state is No Input, else if [sync.valid] is low, the following state is Waiting for Synchronization, otherwise the following state is Initializing.

8.2.2. No Input

Except on reset, this state is entered when [signal_in.valid] is deactivated and left when it is activated. The following state is Initializing, if [sync.valid] is high, or Waiting for Synchronization, otherwise.

8.2.3. Waiting for Synchronization

The Demodulator always falls back to this state if [sync.valid] is not active (except on reset or low [signal_in.valid]). When [sync.valid] is activated, the module proceeds to state Initializing.

8.2.4. Initializing

This state corresponds to Demodulator initialization. The module starts collecting data from [signal_in] to compute channel statistics. After the Goertzel Units have produced INITIALIZATION_DELAY samples, the Demodulator goes to state Looking For Header.

8.2.5. Looking For Header

This state always follows the Initializing state. The Demodulator continually updates channel statistics. When the fit condition for header detection is met, the module goes to state Looking For Delimiter.

The fit condition is summarized by the following equation:

$$A_{01}A_{11} + A_{10}A_{00} > T\,A_{01}A_{10}$$

where T represents [config.header_threshold] and

$$A_{ij} = \sum_{k \in \chi_j} r_i^2[k]$$

where $r_i^2[k]$ is the $(k + 1)$-th last output of the Goertzel module i and

$$\chi_j = \{k \in \{0, \ldots, N-1\}; HEADER[k] = j\}$$

where N = HEADER_WIDTH (Most significant bits are sent first).

8.2.6. Looking For Delimiter

This state always follows the Looking For Header state. In order to securely locate the header limits, we compare the ratio $(A_{01}A_{11} + A_{10}A_{00}) / (A_{01}A_{10})$ obtained at the state transition with the other HEADER_WIDTH-1 corresponding to the following HEADER_WIDTH-1 samples of the Goertzel Units. We determine the index of the first sample of the payload that makes the corresponding ratio maximum.

Notice that the comparisons may be done two by two, and that no actual division is required. The precise procedure is:

Let [header_end] = HEADER_WIDTH – 1;
Let [n] = HEADER_WIDTH – 1;

Let $M_{00} = A_{00}$;
Let $M_{01} = A_{01}$;
Let $M_{10} = A_{10}$;
Let $M_{11} = A_{11}$;
Let [num_max] = $A_{01}A_{11} + A_{10}A_{00}$;
Let [den_max] = $A_{01}A_{10}$;
While [n] > 0,
    Wait sample from Goertzel Units;
    Let [n] = [n] – 1;
    Let [num_cur] = $A_{01}A_{11} + A_{10}A_{00}$;
    Let [den_cur] = $A_{01}A_{10}$;
    If [num_cur] * [den_max] > [num_max] * [den_cur],
        Let [header_end] = [n];
        Let $M_{00} = A_{00}$;
        Let $M_{01} = A_{01}$;
        Let $M_{10} = A_{10}$;
        Let $M_{11} = A_{11}$;
        Let [num_max] = [num_cur];
        Let [den_max] = [den_cur];
Store [space_noise] = M_{01};
Store [space_signal] = max{$M_{00} – M_{01}$, 0};
Store [mark_noise] = M_{10};
Store [mark_signal] = max{$M_{11} – M_{10}$, 0};
Go to state Receiving.

For validity of this procedure, we assume that the Goertzel Units don't produce any new sample outside of the “Wait” statement.

Technical note: This should be asserted during verification.

Design note: Maximum ratio could be exported along with other message statistics. Note, though, that currently we never actually compute it, but only use its numerator and denominator in comparisons.

8.2.7. Receiving

This state always follows the Looking For Delimiter state, such that, upon entering it, noise and signal levels for each frequency have already been estimated. Based on these statistics, log-likelihood values for “data 1” and “data 0” are computed for each of the PAYLOAD_WIDTH bits of the payload. After that, the module proceeds to state Pause.

The difference between the log-likelihoods of “data 1” and “data 0” is output over an AMBA ® AXI [9] protocol, with signals [dem_dec.data], [dem_dec.valid] and [dem_dec.ready]. When a previous transmission is still incomplete, i.e., at entering this state, [dem_dec.valid] is asserted but [dem_dec.ready] has not been asserted

yet, the Demodulator just acknowledges PAYLOAD_WIDTH samples and goes to state Pause.

More precisely, we define the ([header_end]+1)-last sample from the Goertzel Units, counting at the state transition, to be the last header sample. The following PAYLOAD_WIDTH samples are defined as the *payload samples*. We then apply the following procedure:

If [dem_dec.valid] is asserted but [dem_dec.ready] is not,
    Raise [dem_mon.message_discarded] for one clock;
    Wait the production of the last payload sample.
Else,
    For [n] in [1..PAYLOAD_WIDTH]
        Assign to the [n]-th element of [dem_dec.data] the log-likelihood for space channel minus log-likelihood for mark channel relative to the [n]-th payload sample;
    Assert [dem_dec.valid].
Go to state Pause.

Design note: In order to simplify the conditions for discarding data, it was considered to have a likelihood scratch-pad that is just copied to the output in the end of Receiving. Should include internal variable in that case. Idea is currently dismissed.

8.2.8. Pause

This state always follows state Receiving. The modem waits for PAUSE_DELAY pulses of [sync.pulse] after the last decoded bit slot and then proceeds to state Initializing.

IEC standard specifies that three bytes of pause follow every transmission, therefore PAUSE_DELAY = 24.

9. Interfaces

The input and output signals of this module are organized in six interfaces – [dem_dec], [signal_in], [sync], [pga], [config] and [dem_mon].

9.1. [dem_dec]

The [dem_dec] is the main output interface. It is AMBA ® AXI[9] compliant. When a new message is ready to be transmitted, it is made available in the [dem_dec.data] variable, and the [dem_dec.valid] signal is raised. It is kept that way until the [dem_dec.ready] input is high, when transmission is considered successful. Then, [dem_dec.valid] is lowered unless there is new data available.

No new data is written to [dem_dec.data] until transmission is complete. If the module is about to produce new output data, but transmission of previous data is incomplete, the new data is discarded (This is in order to comply with AMBA ® AXI specifications). This is signaled by raising the [dem_mon.message_discarded] event flag for one clock.

The user may employ an output FIFO in order to minimize the chances of this happening.

9.2. [signal_in]

[signal_in] is the main input interface. It may be viewed as a reduced version of an AMBA ® AXI interface, where the ready signal is implicit and forced to high. [signal_in.data] is supposed to be a digital stream corresponding to the digitized signal measured from the mains. This signal is only considered valid when [signal_in.valid] is high, otherwise, the Demodulator goes to No Input state, and waits for the assertion of this input before taking any action.

9.3. [sync]

The [sync] interface operates in the same way. The [sync.valid] signal asserts the validity of the synchronizing pulses of [sync.zero_cross] and [sync.pulse]. When lowered, the module is incapable of receiving any messages.

Design note: Currently the module doesn't eve detect the presence of a carrier if out of synchronization. Maybe this deserves better consideration.

9.4. [pga]

[pga] is a kind of negotiation interface. The module signals with [pga.shift_request] the proposed amplification level for the input signal. The AFE may, however, not support the requested gain. It should select the maximum supported level that does not exceed [pga.shift_request]. In any case, the selected gain should be informed via the input [pga.shift_value].

9.5. [config]

[config] is a direct input. The values given in [config.beta_mark] and [config.beta_space] configure mark and space frequency of operation, respectively, while [config.head_threshold] determines sensitivity and specificity of header identification. These values can be changed at any time, but the module's performance is only guaranteed if they are kept constant except on reset.

9.6. [dem_mon]

A few monitoring signals are provided for the purpose of monitoring the demodulator state and the acquired channel statistics. They are already listed on section 5, but we repeat them here for convenience.

Name	Description	Type	Bits	Range	Reset Value
dem_mon.state	State of the Demodulator, as detailed in section 8.	Enum	3	{RESET, NOI, WFS, INIT, LFH, LFD, RECEIVE, PAUSE}	RESET
dem_mon.carrier_detected	High when the module infers the presence of a carrier in the channel.	Bool	1	*	0
dem_mon.mark_noise	Estimated noise power for the mark channel.	Integer	STATS_WIDTH	*	0
dem_mon.space_noise	Estimated noise power for the space channel.	Integer	STATS_WIDTH	*	0
dem_mon.mark_signal	Estimated signal power for the mark channel.	Integer	STATS_WIDTH	*	0
dem_mon.space_signal	Estimated signal power for the space channel.	Integer	STATS_WIDTH	*	0
dem_mon.message_discarded	Event flag signaling that a received message was discarded for memory exhaustion.	Bool	1	*	0

10. References

[1] IEC 61334-5-1:2001, Distribution automation using distribution line carrier systems – Part 5-1: Lower Layer Profiles – The Spread Frequency Shift Keying (S-FSK) Profile.

[2] http://en.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_avoidance

[3] Goertzel, G. An Algorithm for the Evaluation of Finite Trigonometric Series. *The American Mathematical Monthly*, Vol. 65, No. 1 (Jan., 1958), pp. 34-35.

[4] http://en.wikipedia.org/wiki/Goertzel_algorithm

[5]

[6] PLCM Requirements Specification.

[7]

[8] Artigo sobre S-FSK da palestra de Felipe.

[9] AMBA ® AXI Protocol Specification.

[10] http://en.wikipedia.org/wiki/Soft-decision_decoder

ANEXO B – Scripts

```
%% Analysis of effect of sample rate on noise power.
%%
%% Compatible with: GNU Octave.
%%
%% Reference: Katayama, M.; Yamazato, T.; Okada, H., "A mathematical model of
%% noise in narrowband power line communication systems," Selected Areas in
%% Communications, IEEE Journal on , vol.24, no.7, pp.1267,1276, July 2006.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-02-23
%% Keywords: PLC

1;

%% Noise autocorrelation according to referenced paper
function y = noise_autocor(x, Ta)
   y = 1./(1+(2*pi*Ta.*x).^2);
end

%% Average component of noise variance
function res = noise_variance_average(N, Ta, Tw)
   x = [1:N-1] / N;
   a1 = 1 / 2;
   b1 = (1 - x) .* noise_autocor(x, Ta) .* cos(Tw * x);

   res = a1 + sum(b1);
   res /= N;
end

%% Phase component of noise variance
function res = noise_variance_phase(N, Ta, Tw)
   x = [1:N-1] / N;
   a2 = .5 * (csc(Tw/N)/N) * sin(Tw);
   b2 = (csc(Tw/N)/N) * noise_autocor(x, Ta) .* sin(Tw * (1 - x));

   res = a2 + sum(b2);
   res /= N;
end


%%--- Fundamental variables
baud = 360  %% Baud rate. Minimum per IEC 61334-5-1, at 60 Hz, is 360 bps.
f = 122e3   %% Carrier frequency. Maximum in CENELEC band B is 125 kHz.
%% Parameter of PDS, in seconds, as per the referenced paper.
%% Value found by authors was 1.2e-5 s.
a = 1.2e-5

%%--- Derived parameters
T = 1/baud; % bit slot, in seconds.
w = 2*pi*f;
Ta = T/a;
Tw = T*w;
af = a*f;

%% Limit of average component
lim_average = quadcc(@(x) (1 - x) .* noise_autocor(x, Ta) .* cos(Tw * x), 0, 1, 1e-6
)
%% Estimate for large Tw and Ta
lim_estimate = 1/ (4 * Ta * exp(af))
%% Compare estimate and numeric integral.
lim_average_to_estimate_ratio = lim_average / lim_estimate


%%--- Limit of phase component
% We maximize it over minor variations in Tw.
lim_max_phase_x = ...
   quadcc(@(x) noise_autocor(x, Ta) .* sin(Tw * (1 - x)), ...
      0, 1, 1e-6) / Tw;
lim_max_phase_y = ...
   quadcc(@(x) noise_autocor(x, Ta) .* sin((Tw + pi/2) * (1 - x)), ...
      0, 1, 1e-6) / (Tw + pi/2);
lim_max_phase = norm([lim_max_phase_x lim_max_phase_y])
```

```
lim_max_phase_to_average_ratio = lim_max_phase / lim_average

%% Following code strictly maximizes phase component by varying Tw parameter
%% in the range [Tw; Tw + 2*pi].
%%
%%    [x, fval, info, output] = fminbnd(...
%%       @(phi)...
%%          quadcc(@(x) noise_autocor(x, Ta) .* sin((Tw + phi) * (1 - x)), ...
%%          0, 1, 1e-6) / (Tw + phi), ...
%%       0, 2*pi);
%%    lim_max_phase = -fval

%% Following code may be used to visualise variation of phase component with
%% minor perturbations in Tw.
%%
%%    Twphi = Tw + linspace(0, 2*pi, 20);
%%    lim_phase = zeros(1, length(Twphi));
%%    for k = [1:length(Twphi)]
%%       lim_phase(k) = ...
%%          quadcc(@(x) noise_autocor(x, Ta) .* sin(Twphi(k) * (1 - x)),...
%%             0, 1, 1e-6) / Twphi(k);
%%    end
%%    plot(Twphi/2/pi, lim_phase);
%%    pause


%%--- Find good sampling rate.

cut = 2 * lim_average; % 3 dB

Ncut = fzero(@(x) noise_variance_average(floor(x), Ta, Tw) - cut, [2*f*T 1e6]);
fcut = Ncut / T

%%--- Check phase component at chosen sampling rate.
% Again, we maximize it over minor variations in Tw.
max_phase_component_x = noise_variance_phase(Ncut, Ta, Tw);
max_phase_component_y = noise_variance_phase(Ncut, Ta, Tw + pi/2);
max_phase_component = norm([max_phase_component_x max_phase_component_y])

max_phase_to_average_ratio = max_phase_component / cut

%% Following code strictly maximizes phase component by varying Tw parameter
%% in the range [Tw; Tw + 2*pi].
%%
%%    [x, fval, info, output] = fminbnd(...
%%       @(phi) noise_variance_phase(Ncut, Ta, Tw + phi),
%%       0, 2*pi);
%%    max_phase_component = -fval

%% Following code may be used to visualise variation of phase component with
%% minor perturbations in Tw.
%%
%%    Twphi = Tw + linspace(0, 2*pi, 20);
%%    phase_component = zeros(1, length(Twphi));
%%    for k = [1:length(Twphi)]
%%       phase_component(k) = noise_variance_phase(Ncut, Ta, Twphi(k));
%%    end
%%    plot(Twphi/2/pi, phase_component);
%%    pause


%%--- Plot average noise variance versus sampling rate.

fs = logspace(floor(log10(fcut) - 1), ceil(log10(fcut) + 1), 100);
N = round(T * fs);

v = zeros(1, length(N));
for k = [1:length(N)]
   v(k) = noise_variance_average(N(k), Ta, Tw);
end

%% Raw plot: variance vs. number of samples.
%%    loglog(N, v, [N(1), N(end)], [cut cut], [N(1), N(end)], [lim_average lim_avera
```

```
ge]);

%% Normalised plot: Normalised variance in dB vs. sampling frequency.
semilogx(...
   fs, 10 * log10(v / lim_average),...
   [fs(1) fs(end)], 10*log10([cut cut] / lim_average));
xlabel("Sampling frequency (Hz)");
ylabel("Relative noise variance (dB)");
pause
```

```
%% Mantissa and exponent of likelihood coefficients for custom floating point.
%%
%% Compatible with: GNU Octave.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-15
%% Keywords: PLC S-FSK

%% Maximum index (counting from 1)
max_index = 17
%% Significant bits in custom floating point
significant_bits = 10

%% Double precision coefficients
[a b] = likelihood_coefficients(max_index);

%% Mantissa and exponent
[af, ae] = log2(a);
[bf, be] = log2(b);

%% Round mantissa to custom precision
a_mantissa = round(af * (2 ^ significant_bits));
b_mantissa = round(bf * (2 ^ significant_bits));

%% Print
A = [ae', a_mantissa']
B = [be', b_mantissa']

a_rounded = a_mantissa .* (2 .^ (ae - significant_bits));
b_rounded = b_mantissa .* (2 .^ (be - significant_bits));

%% Plot approximation based on rounded coefficients
x = [0 2.^linspace(-4, (max_index - 2) * (1 - eps), 1e4)];
yref = log(besseli(0, sqrt(4*x)));
e = 1 + max(floor(log2(4*x)), 0);
y = a_rounded(e) .* x + b_rounded(e);

plot(x, yref, x, y);
pause

%% Relative error
semilogx(x, (y - yref) ./ yref);
pause
```

```
function [a, b] = likelihood_coefficients(max_index)

x = [0 2.^[1:max_index]] / 4;
y = log(besseli(0, sqrt(4*x)));

a = diff(y) ./ diff(x);
b = y(1:max_index) - a.*x(1:max_index);
end
```

```
%% Demonstration of function goertzel_beta_limits.
%%
%% Compatible with: GNU Octave.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-01
%% Keywords: Goertzel PLC

1;

%%--- Fundamental variables
%% Baud rate. Minimum per IEC 61334-5-1, at 60 Hz, is 360 bps.
baud = 360
%% Maximum admissible jitter of bit slot length (in seconds). AFE031 specifies
%% typical value of 10 ns at 50 Hz, 240 V RMS for zero crossing detection.
jitter = 1000e-9
%% Number of bits of order higher than input's MSB.
bits = 12

%%--- Derived parameters
T = 1/baud + jitter; %% Maximum length of bit slot.

%%--- Example call
example_fs = 500e3
[corresponding_beta0 corresponding_beta1] = ...
   goertzel_beta_limits(bits, T, example_fs)

%%--- Plot
wmiddle = 1.44647666836166; % precomputed (see goertzel_beta_limits.m)
fs_upper_bound = ...
   fzero(@(fs) goertzel_overflow_bits(wmiddle,T*fs*wmiddle/(2*pi)+2) - bits, ...
      [1e4, 1e9])
fs = logspace(4, log10(fs_upper_bound), 200);
beta1 = zeros(1, length(fs));
beta2 = zeros(1, length(fs));
for k = [1:length(fs)]
   [beta1(k) beta2(k)] = goertzel_beta_limits(bits, T, fs(k));
end

plot(fs, beta1, fs, beta2);
pause
```

```
%% Choice of number of bits of order higher than input's MSB in Goertzel state
%% variables.
%%
%% Our choice is based on the following constraint: We want to have a range of
%% sampling frequencies [fsmin; fsmax] such that fsmax/fsmin > fs_min_ratio,
%% for a given threshold fs_min_ratio, and that, for all sampling frequencies
%% in this range, carrier frequencies in a given range [fmin; fmax] be
%% supported without possibility of overflow.
%%
%% Compatible with: GNU Octave.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-01
%% Keywords: Goertzel PLC

1;

%% Minimum sampling frequency given number max_bits of extra bits, length T of
%% bit slot and central frequency f of filter.
function res = min_sampling_frequency(max_bits, T, f)
   middle = acos(-1/3); % argmin of goertzel_overflow_bits.
   omega = fzero(@(w) goertzel_overflow_bits(w, T*f) - max_bits, ...
      [middle, pi * (1-eps)]);
   res = 2*pi * f / omega;
end

%% Maximum sampling frequency given number max_bits of extra bits, length T of
%% bit slot and central frequency f of filter.
function res = max_sampling_frequency(max_bits, T, f)
   middle = acos(-1/3); % argmin of goertzel_overflow_bits.
   omega = fzero(@(w) goertzel_overflow_bits(w, T*f) - max_bits, ...
      [eps, middle]);
   res = 2*pi * f / omega;
end

%% Find the bounds fs1 and fs2 of the interval for which
%%    goertzel_overflow_bits(2*pi*f/fs, T*f) <= max_bits,
%% for given T and all f in the interval [fmin; fmax].
function [fs1 fs2] = admissible_interval(max_bits, T, fmin, fmax)
   % Interval intersection.

   % minimax
   %[x val1] = fminbnd(@(f) max_sampling_frequency(max_bits, T, f), fmin, fmax); % m
inimax
   % Here we use the fact that max_sampling_frequency is log-concave in log(f).
   val1 = min([max_sampling_frequency(max_bits, T, fmin), ...
               max_sampling_frequency(max_bits, T, fmax)]);

   % - maximin
   %[x val2] = fminbnd(@(f) -min_sampling_frequency(max_bits, T, f), fmin, fmax);
   % For this one we use the fact that min_sampling_frequency is crescent in f.
   val2 = -min_sampling_frequency(max_bits, T, fmax);

   fs1 = -val2;
   fs2 = val1;
end

function res = span(max_bits, T, fmin, fmax)
   [fs1 fs2] = admissible_interval(max_bits, T, fmin, fmax);
   res = fs2 / fs1;
end

%%--- Fundamental variables
%% Baud rate. Minimum per IEC 61334-5-1, at 60 Hz, is 360 bps.
baud = 360
%% Maximum admissible jitter of bit slot length (in seconds). AFE031 specifies
%% typical value of 10 ns at 50 Hz, 240 V RMS for zero crossing detection.
jitter = 1000e-9
%% Maximum carrier frequency. Maximum in CENELEC band B is 125 kHz.
fmax = 122e3
%% Minimum carrier frequency. Minimum in CENELEC band B is 95 kHz.
fmin = 40e3
%% Required ratio between maximum and minimum sample frequency.
```

```
fs_min_ratio = 2

%%--- Derived parameters
T = 1/baud + jitter; %% Maximum length of bit slot.

%%--- Solve, over b, span(b, T, fmin, fmax) == fs_min_ratio;
middle = acos(-1/3); % argmin of goertzel_overflow_bits.
lower_bound = goertzel_overflow_bits(middle, T*fmax)
bits = fzero(@(b) span(b, T, fmin, fmax) - fs_min_ratio, [lower_bound 50])
bits = ceil(bits)

[fsmin fsmax] = admissible_interval(bits, T, fmin, fmax)

fs_ratio = fsmax / fsmin

assert(fsmin >= 2 * fmax);
assert(fs_ratio >= fs_min_ratio);


%%--- Plot ratio vs. bits
b = linspace(ceil(lower_bound), 2 * bits - ceil(lower_bound), 10);
sp = zeros(1, length(b));
for k = [1:length(b)]
  sp(k) = span(b(k), T, fmin, fmax);
end

semilogy(b, sp, [b(1) b(end)], [fs_min_ratio fs_min_ratio]);
xlabel("Extra bits");
ylabel("fs_{max} / fs_{min}");
pause


%%--- Plot admissible region.
% Magic number sqrt(16/27) equals sin(middle)*sin(middle/2)
upper_bound_for_f = (sqrt(16/27)*2^bits - 1) / T / 2
f1 = fsmin^2 / pi^2 * T / 2^bits; % Guarantee that fs_upper(1) < fsmin.
f = logspace(floor(log10(f1)), log10(upper_bound_for_f));
fs_upper = zeros(1, length(f));
fs_lower = zeros(1, length(f));
for k = [1:length(f)]
   fs_upper(k) = max_sampling_frequency(bits, T, f(k));
   fs_lower(k) = min_sampling_frequency(bits, T, f(k));
end

%%-- Admissible region in fs vs. f.
loglog(f, fs_upper, f, fs_lower, ...
   [fmin fmax], [1 1]*fsmin, 'k', [fmin fmax], [1 1]*fsmax, 'k', ...
   [fmin fmin], [fsmin fsmax], 'k', [fmax fmax], [fsmin fsmax], 'k');
xlabel("Carrier frequency (Hz)");
ylabel("Sampling frequency (Hz)");
title("Admissible region");
pause

%%-- Admissible region in f vs. fs.
loglog(fs_upper, f, fs_lower, f, ...
   [1 1]*fsmin, [fmin fmax], 'k', [1 1]*fsmax, [fmin fmax], 'k', ...
   [fsmin fsmax], [fmin fmin], 'k', [fsmin fsmax], [fmax fmax], 'k');
xlabel("Sampling frequency (Hz)");
ylabel("Carrier frequency (Hz)");
title("Admissible region");
pause

%%-- Admissible region in omega/pi vs. f.
semilogx(f, 2*f./fs_upper, f, 2*f./fs_lower)
xlabel("Carrier frequency (Hz)");
ylabel("w/pi");
title("Admissible region");
pause

%%-- Admissible region in beta vs. fs.
index_lower = (fs_lower >= fs_upper(1));
fs_lower_r = fs_lower(index_lower);
f_r = f(index_lower);
```

```
beta_l = cos(2*pi*f_r./fs_lower_r);
beta_u = cos(2*pi*f./fs_upper);
semilogx(fs_lower_r, beta_l, fs_upper , beta_u)
xlabel("Sampling frequency (Hz)");
ylabel("beta");
title("Admissible region");
pause
```

```
%% usage: RES = dtft_overflow_bits(OMEGA, TF)
%%
%% Return log2 of upper bound on ratio r^2/M^2, where r is the absolute value
%% of the DTFT of a signal limited between -M/2 and M/2, at discrete frequency
%% omega, during an interval corresponding to TF periods.
%%
%% Arguments:
%% OMEGA: Central discrete frequency of filter;
%% TF:    Number of periods of the waveform of period OMEGA.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-05
%% Keywords: Goertzel PLC
function res = dtft_overflow_bits(omega, Tf)
   res = 2 * log2((2*Tf + 1) * csc(omega/2) / 2);
end
```

```
%% Choice of width for beta input of Goertzel unit.
%%
%% Beta is the cosine of the central discrete frequency. We base our choice on
%% the constraint that a step in beta correspond to less than half of the
%% "pass band" of the DTFT calculated over the finite interval of a bit slot.
%%
%% Compatible with: GNU Octave.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-02
%% Keywords: Goertzel PLC

%%--- Fundamental variables
%% Baud rate. Minimum per IEC 61334-5-1, at 60 Hz, is 360 bps.
baud = 360
%%-- Frequency versus sample frequency region of interest
%%   (see goertzel_overflow_bits_design.m)
%% Minimum carrier frequency. Minimum in CENELEC band B is 95 kHz.
fmin = 40e3
%% Maximum carrier frequency. Maximum in CENELEC band B is 125 kHz.
fmax = 122e3
%% Minimum sampling frequency
fsmin = 258e3 % (obtained via goertzel_overflow_bits_design.m)
%% Maximum sampling frequency
fsmax = 752e3 % (obtained via goertzel_overflow_bits_design.m)

%%--- Derived parameters
T = 1/baud; % Length of bit slot

% sinc cut frequency
delta_omega_c = pi * fzero(@(x) sinc(x) - sqrt(.5), [0 1]);

% x(1) is carrier frequency; x(2) is sampling frequency.
initial_guess = ([fmin fsmin] + [fmax fsmax]) / 2;

% Maximize N * csc(omega) the easy way.
[x, obj, info, iter, nf, lambda] = ...
   sqp(initial_guess,
      @(x) -T * x(2) * csc(2*pi*x(1)/x(2)),
      [], [],
      [fmin fsmin], [fmax fsmax]);

minimizing_f = x(1)
minimizing_fs = x(2)

beta_required_resolution = 2 * delta_omega_c / -obj

beta_bits = log2(2 / beta_required_resolution)
beta_bits = ceil(beta_bits)
```

```
%% Created: 2014-03-04

%%--- Fundamental variables
%% Baud rate. Minimum per IEC 61334-5-1, at 60 Hz, is 360 bps.
baud = 360
%% Maximum admissible jitter of bit slot length (in seconds). AFE031 specifies
%% typical value of 10 ns at 50 Hz, 240 V RMS for zero crossing detection.
jitter = 1000e-9
%%-- Frequency versus sample frequency region of interest
%%   (see goertzel_overflow_bits_design.m)
%% Minimum carrier frequency. Minimum in CENELEC band B is 95 kHz.
fmin = 40e3
%% Maximum carrier frequency. Maximum in CENELEC band B is 125 kHz.
fmax = 122e3
%% Minimum sampling frequency
fsmin = 258e3 % (obtained via goertzel_overflow_bits_design.m)
%% Maximum sampling frequency
fsmax = 752e3 % (obtained via goertzel_overflow_bits_design.m)
fsmax = fsmin * 2 % Override
%% Number of bits in Goertzel state variables of order higher than input's MSB
goertzel_bits = 12 % (obtained via goertzel_overflow_bits_design.m)
%% Number of bits in square of DTFT of order higher than 4*U^2, where U is the
%% maximum input value.
dtft_bits = 18


%%--- Derived parameters
T = 1/baud + jitter; %% Maximum length of bit slot.

%% Minimum sampling frequency given number goertzel_bits of extra bits, length T of
%% bit slot and central frequency f of filter.
function [fs1 fs2] = goertzel_sampling_frequency_limits(goertzel_bits, T, f)
   middle = acos(-1/3); % argmin of goertzel_overflow_bits.
   omega1 = fzero(@(w) goertzel_overflow_bits(w, T*f) - goertzel_bits, ...
      [middle, pi * (1-eps)]);
   omega2 = fzero(@(w) goertzel_overflow_bits(w, T*f) - goertzel_bits, ...
      [eps, middle]);
   fs1 = 2*pi * f / omega1;
   fs2 = 2*pi * f / omega2;
end

function [fs1 fs2] = dtft_sampling_frequency_limits(dtft_bits, T, f)
   omega2 = 2 * asin((2*T*f + 1) / 2^(dtft_bits/2 + 1));
   fs1 = 2 * f;
   fs2 = 2*pi * f / omega2;
end

% Magic number sqrt(16/27) equals sin(middle)*sin(middle/2)
upper_bound_for_f_goertzel = (sqrt(16/27)*2^goertzel_bits - 1) / T / 2
upper_bound_for_f_dtft = (2^(dtft_bits/2 + 1) + 1) / T / 2

f1 = fsmin^2 / pi^2 * T / 2^goertzel_bits; % Repeat bound from goertzel_overflow_bit
s_design.m
f_goertzel = logspace(floor(log10(f1)), log10(upper_bound_for_f_goertzel), 500);
f_dtft = logspace(floor(log10(f1)), log10(upper_bound_for_f_dtft), length(f_goertzel
));
fs_upper_goertzel = zeros(1, length(f_goertzel));
fs_lower_goertzel = zeros(1, length(f_goertzel));
fs_upper_dtft = zeros(1, length(f_dtft));
fs_lower_dtft = zeros(1, length(f_dtft));

for k = [1:length(f_goertzel)]
   [fs_lower_goertzel(k) fs_upper_goertzel(k)] = ...
      goertzel_sampling_frequency_limits(goertzel_bits, T, f_goertzel(k));
end

for k = [1:length(f_dtft)]
   [fs_lower_dtft(k) fs_upper_dtft(k)] = ...
      dtft_sampling_frequency_limits(dtft_bits, T, f_dtft(k));
end

loglog(fs_upper_goertzel, f_goertzel, fs_lower_goertzel, f_goertzel, ...
   fs_upper_dtft, f_dtft, fs_lower_dtft, f_dtft, ...
```

```
   [1 1]*fsmin, [fmin fmax], 'k', [1 1]*fsmax, [fmin fmax], 'k', ...
   [fsmin fsmax], [fmin fmin], 'k', [fsmin fsmax], [fmax fmax], 'k');
xlabel("Sampling frequency (Hz)");
ylabel("Carrier frequency (Hz)");
title("Admissible region");
pause



%% Plot intersection.
f_goertzel = f_dtft;
fs_upper_goertzel = zeros(1, length(f_goertzel));
fs_lower_goertzel = zeros(1, length(f_goertzel));

for k = [1:length(f_goertzel)]
   [fs_lower_goertzel(k) fs_upper_goertzel(k)] = ...
      goertzel_sampling_frequency_limits(goertzel_bits, T, f_goertzel(k));
end

fs_upper = min(fs_upper_dtft, fs_upper_goertzel);
fs_lower = max(fs_lower_dtft, fs_lower_goertzel);

ind = (fs_lower <= fs_upper);

prev_axis = axis();
loglog(
   fs_upper(ind), f_dtft(ind), ...
   fs_lower(ind), f_dtft(ind), ...
   [1 1]*fsmin, [fmin fmax], 'k', [1 1]*fsmax, [fmin fmax], 'k', ...
   [fsmin fsmax], [fmin fmin], 'k', [fsmin fsmax], [fmax fmax], 'k');
xlabel("Sampling frequency (Hz)");
ylabel("Carrier frequency (Hz)");
title("Admissible region");
axis(prev_axis); % Keep axis from previous plot:
pause
```

```
%% usage: [BETA1 BETA2] = dtft_beta_limits(BITS, T, FS)
%%
%% Return minimum and maximum admissible values for the beta parameter of a
%% DTFT unit that won't overflow its result. Beta is the cosine of the
%% discrete frequency.
%%
%% Arguments:
%% BITS: Number of bits used to represent the square of the absolute value of
%%       the DTFT that have order higher than MSB of square of input.
%% T:    Duration of bit slot;
%% FS:   Sampling frequency.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-05
%% Keywords: Goertzel PLC
function [beta1 beta2] = dtft_beta_limits(bits, T, fs)

   N = T*fs;
   wmin = fzero(@(w) tan(w/2) - w/2 - pi/2/N, [0, pi*(1 - eps)]);

   if (dtft_overflow_bits(wmin, N*wmin/(2*pi)) >= bits)
      beta1 = 1;
      beta2 = -1;
      return
   end

   omega2 = fzero(@(w) dtft_overflow_bits(w, N*w/(2*pi)) - bits, [eps, wmin]);
   %if (N + 1 <= 2^(bits/2 + 1))
   if (dtft_overflow_bits(pi, N/2) <= bits)
      omega1 = pi;
   else
      omega1 = fzero(@(w) dtft_overflow_bits(w, N*w/(2*pi)) - bits, [wmin, pi]);
   end

   beta1 = cos(omega1);
   beta2 = cos(omega2);
end
```

```
%% usage: RES = goertzel_overflow_bits(OMEGA, TF)
%%
%% Return log2 of upper bound on ratio between Goertzel variable and maximum
%% input. This gives a minimum number of bits of order higher than input's MSB
%% for Goertzel filter's state variables that guarantee that no overflow will
%% occur.
%%
%% Arguments:
%% OMEGA: Central discrete frequency of filter;
%% TF:    Number of periods of the waveform of period OMEGA.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-01
%% Keywords: Goertzel PLC
function res = goertzel_overflow_bits(omega, Tf)
   res = log2((2*Tf + 1) * csc(omega/2) .* csc(omega));
end
```

```
%% usage: [BETA1 BETA2] = goertzel_beta_limits(BITS, T, FS)
%%
%% Return minimum and maximum admissible values for the beta parameter of a
%% Goertzel filter. Beta is the cosine of the central discrete frequency.
%%
%% Arguments:
%% BITS: Number of bits of higher order than input's MSB in Goertzel variables;
%% T:    Duration of bit slot;
%% FS:   Sampling frequency.

%% Author: Felipe Goncalves Assis <fgassis@lad.dsc.ufcg.edu.br>
%% Created: 2014-03-01
%% Keywords: Goertzel PLC
function [beta1 beta2] = goertzel_beta_limits(bits, T, fs)
   %wmiddle = fminbnd(@(w) -sin(w).*sin(w/2)./w, 0, pi);
   %wmiddle = fzero(@(w) w + 3*w*cos(w) - 2*sin(w),[eps pi]);
   wmiddle = 1.44647666836166; % precomputed.
   if (goertzel_overflow_bits(wmiddle, T*fs*wmiddle/(2*pi) + 1) >= bits)
      beta1 = 1;
      beta2 = -1;
      return
   end
   omega1 = fzero(@(w) goertzel_overflow_bits(w, T*fs*w/(2*pi)) - bits, [eps, wmiddl
e]);
   omega2 = fzero(@(w) goertzel_overflow_bits(w, T*fs*w/(2*pi)) - bits, [wmiddle, pi
 * (1-eps)]);
   beta1 = cos(omega2);
   beta2 = cos(omega1);
end
```

```
function noise = noise_sim(sigma, freqs)
% usage: NOISE = noise_sim(SIGMA, H)
%
% Simulate noise in a power line channel.
%
% NOISE is the output of a filter with frequency response H having as
% input an independent cyclostationary gaussian process with
% standard-deviation versus phase given by SIGMA.  The number of
% simulated periods of the process (which correspond to a half-period
% of the mains) is given by length(H) / length(SIGMA), which should be
% an integer.
%
% It is best to have length(SIGMA) and length(H) to be powers of 2,
% since an FFT algorithm is used to compute the output of the filter.
%
% See also: make_sigma, make_freqs, demo_noise.
%
% Compatible with: GNU Octave, MATLAB.

% Author: Felipe Gonçalves Assis  <fgassis@lad.dsc.ufcg.edu.br>
% Created: 2012-04-27
% Keywords: PLC noise model

N = length(freqs);
k = length(freqs) / length(sigma);

prenoise = repmat(sigma, 1, k) .* randn(1,N);
nnoise = ifft(fft(prenoise) .* freqs);
noise = real(nnoise);

%d = norm(noise) / norm(prenoise) % debug
%max_arg = max(abs(angle(nnoise.^2))) % debug
end
```

```
% Example demonstrating the simulation of power line noise.
%
% Compatible with: GNU Octave, MATLAB.
fmains = 60; % Mains frequency (Hz).
n = 2^14; % Samples per half-period.

k = 4; % Number of half-periods.

a = 1.2e-5; % Frequency domain parameter (s).
param = [.23 0 0; 1.38 -6 1.91; 7.17 -35 1.57e5]; % Time domain parameters.

fs = 2 * n * fmains; % Sampling frequency.

sigma = make_sigma(param, n);
freqs = make_freqs(a, n, k, fmains);
noise = noise_sim(sigma, freqs);

t = linspace(0, (k*n-1)/fs, k*n); % Time index.
plot(t, noise);
```

```
function freqs = make_freqs(a, n, k, fmains)
% usage: H = make_freqs(A, N, K, FMAINS)
%
% Produce a vector representing a frequency response suitable for use
% by the method noise_sim. The model used is
%
%     H(f) = C * exp(-A * |f| / 2),
%
% where C is a constant such that H preserves the total power of white
% noise of high bandwidth.
%
% N is the number of samples per half-period of the mains.  K is the
% number of half-periods to be sampled.  FMAINS is the frequency of
% the mains signal.
%
% It is best to have N and K to be powers of two, for use of the
% output in FFT algorithms.
%
% See also: noise_sim, make_sigma, demo_noise.
%
% Compatible with: GNU Octave, MATLAB.

% Author: Felipe Gonçalves Assis  <fgassis@lad.dsc.ufcg.edu.br>
% Created: 2012-04-30
% Keywords: PLC noise model

N = n * k;
fs = 2 * fmains * n;
C = N * tanh(a*fs/2 / N) / (1 - exp(-a*fs/2)); % should be approx. a*fs/2 / (1 - exp
(-a*fs/2)) approx. a*fs/2;
B = .5 * log(C);

freqs = exp([linspace(B, B - a/2 * (fs/2 - fs/N), N/2), linspace(B - a/2 * fs/2, B -
 a/2 * fs/N, N/2)]);
end
```

```
function sigma = make_sigma(param, N)
% usage: SIGMA = make_sigma(PARAM, N)
%
% Produce a vector with the values of the standard-deviation of a
% normalized power line noise signal as a function of phase [0, pi),
% according to the model
%
%  SIGMA .^ 2 (PHI) = Sum { A(l) * |sin(PHI + THETA(l))| .^ B(l) }  (l = [1:L])
%
% where A(l) == param(l,1), THETA(l) == param(l,2) is given in
% degrees, and B(l) == param(l,3).
%
% See also: noise_sim, make_freqs, demo_noise.
%
% Compatible with: GNU Octave, MATLAB.

% Author: Felipe Gonçalves Assis  <fgassis@lad.dsc.ufcg.edu.br>
% Created: 2012-04-27
% Keywords: PLC noise model

sigma = sqrt(sum(bsxfun(@times, ...
                        param(:,1), ...
                        bsxfun(@power, ...
                               abs(sin(bsxfun(@plus, ...
                                              repmat(linspace(0, pi - pi/N, N), ...
                                                     size(param, 2), 1), ...
                                              param(:,2) * pi/180))), ...
                               param(:,3)))));
end
```

ANEXO C – Modelo e Simulação

```
TOOL = vcs
ARGS = -full64 -sverilog -ntb_opts uvm -debug_pp -timescale=1ns/10ps
RUN_OPTIONS = +UVM_TR_RECORD +UVM_VERBOSITY=HIGH
DEPENDENCIES = \
lu_float.svh \
signal_in_packet.svh \
sync_packet.svh \
sync_packet_a.svh \
pga_packet.svh \
config_dem_packet.svh \
channel_sample_packet.svh \
dem_dec_packet.svh \
signal_in_seq_lib.svh \
sync_seq_lib.svh \
pga_seq_lib.svh \
config_dem_seq_lib.svh \
cm_refmod.svh \
lu_refmod.svh \
dem_refmod.svh \
bvm_delayed_writer.svh \
bvm_simple_recorder.svh \
bvm_sink.svh \
dem_single_refmod_env.svh \
dem_single_refmod_basic_test.svh
TB_FILES = dem_single_refmod_top.sv

basic: .basic_made

.basic_made: simv
	echo "done" > .basic_made && ./$< $(RUN_OPTIONS) +UVM_TESTNAME=dem_single_re
fmod_basic_test \
	|| rm .basic_made

simv: $(TB_FILES) $(DEPENDENCIES) dem_single_refmod_basic_test.svh
	$(TOOL) $(ARGS) $(TB_FILES)

clean:
	rm -rf csrc simv.vdb simv.daidir simv vc_hdrs.h ucli.key DVEfiles vcdplus.vp
d .*_made
```

```
TOOL = vcs
ARGS = -full64 -sverilog -ntb_opts uvm -debug_pp -timescale=1ns/10ps
RUN_OPTIONS = +UVM_TR_RECORD +UVM_VERBOSITY=HIGH
DEPENDENCIES = \
lu_float.svh \
signal_in_packet.svh \
sync_packet.svh \
sync_packet_a.svh \
pga_packet.svh \
config_dem_packet.svh \
channel_sample_packet.svh \
signal_in_seq_lib.svh \
sync_seq_lib.svh \
pga_seq_lib.svh \
config_dem_seq_lib.svh \
cm_refmod.svh \
bvm_delayed_writer.svh \
bvm_simple_recorder.svh \
cm_single_refmod_env.svh
TB_FILES = cm_single_refmod_top.sv

basic: .basic_made

.basic_made: simv
        echo "done" > .basic_made && ./$< $(RUN_OPTIONS) +UVM_TESTNAME=cm_single_ref
mod_basic_test \
        || rm .basic_made

simv: $(TB_FILES) $(DEPENDENCIES) cm_single_refmod_basic_test.svh
        $(TOOL) $(ARGS) $(TB_FILES)

clean:
        rm -rf csrc simv.vdb simv.daidir simv vc_hdrs.h ucli.key DVEfiles vcdplus.vp
d .*_made
```

```
TOOL = vcs
ARGS = -full64 -sverilog -ntb_opts uvm -debug_pp -timescale=1ns/10ps
RUN_OPTIONS = +UVM_TR_RECORD +UVM_VERBOSITY=HIGH
DEPENDENCIES = \
./lu_float.svh \
./channel_sample_packet.svh \
./config_dem_packet.svh \
./dem_dec_packet.svh \
./channel_sample_seq_lib.svh \
./config_dem_seq_lib.svh \
./lu_refmod.svh \
./bvm_delayed_writer.svh \
./bvm_sink.svh \
./lu_single_refmod_env.svh \
./lu_single_refmod_basic_test.svh \
./lu_single_refmod_directed_test.svh
TB_FILES = lu_single_refmod_top.sv

basic: .basic_made

directed: .directed_made

.basic_made: simv
	rm .*_made; ./$< $(RUN_OPTIONS) +UVM_TESTNAME=lu_single_refmod_basic_test &&
 echo "done" > .basic_made

.directed_made: simv
	rm .*_made; ./$< $(RUN_OPTIONS) +UVM_TESTNAME=lu_single_refmod_directed_test
 && echo "done" > .directed_made

simv: $(TB_FILES) $(DEPENDENCIES)
	$(TOOL) $(ARGS) $(TB_FILES) -o $@

clean:
	rm -rf csrc simv.vdb simv.daidir simv vc_hdrs.h ucli.key DVEfiles vcdplus.vp
d .*_made
```

```
class dem_refmod extends uvm_component;
   `uvm_component_utils(dem_refmod)

   uvm_put_port #(dem_dec_packet) dem_dec_out;
   uvm_analysis_port #(pga_packet) pga_req_out;
   uvm_analysis_export #(pga_packet) pga_rsp_in;
   uvm_analysis_export #(signal_in_packet) signal_in_in;
   uvm_analysis_export #(sync_packet) sync_in;
   uvm_analysis_export #(config_dem_packet) config_dem_in;

   cm_refmod cm;
   lu_refmod lu;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      dem_dec_out = new("dem_dec_out", this);
      pga_req_out = new("pga_req_out", this);
      pga_rsp_in = new("pga_rsp_in", this);
      signal_in_in = new("signal_in_in", this);
      sync_in = new("sync_in", this);
      config_dem_in = new("config_dem_in", this);
   endfunction

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      cm = cm_refmod::type_id::create("cm", this);
      lu = lu_refmod::type_id::create("lu", this);
   endfunction

   virtual function void connect_phase(uvm_phase phase);
      lu.dem_dec_out.connect(this.dem_dec_out);
      cm.pga_req_out.connect(this.pga_req_out);

      cm.channel_sample_out.connect(lu.channel_sample_in);

      this.pga_rsp_in.connect(cm.pga_rsp_in);
      this.signal_in_in.connect(cm.signal_in_in);
      this.sync_in.connect(cm.sync_in);
      this.config_dem_in.connect(cm.config_cm_in);
      this.config_dem_in.connect(lu.config_lu_in);
   endfunction
endclass
```

```
`uvm_analysis_imp_decl(_pga)
`uvm_analysis_imp_decl(_signal_in)
`uvm_analysis_imp_decl(_sync)
`uvm_analysis_imp_decl(_config_cm)

class cm_refmod extends uvm_component;
   `uvm_component_utils(cm_refmod)

   uvm_analysis_port #(channel_sample_packet) channel_sample_out;
   uvm_analysis_port #(pga_packet) pga_req_out;
   uvm_analysis_imp_pga #(pga_packet, cm_refmod) pga_rsp_in;
   uvm_analysis_imp_signal_in #(signal_in_packet, cm_refmod) signal_in_in;
   uvm_analysis_imp_sync #(sync_packet, cm_refmod) sync_in;
   uvm_analysis_imp_config_cm #(config_dem_packet, cm_refmod) config_cm_in;

   // TODO: plc_globals
   localparam GOERTZEL_WIDTH = 29;
   localparam GOERTZEL_FRACTIONAL_BITS = 0;
   localparam GOERTZEL_MSB = GOERTZEL_WIDTH - GOERTZEL_FRACTIONAL_BITS - 1;
   localparam DTFT_MSB = 51;
   localparam DTFT_LSB = 30;

   channel_sample_packet sample;
   bit do_write;
   bit get_pulse;
   // inputs
   int u; // signal_in
   int unsigned shift;
   int unsigned pulse[];
   int unsigned new_pulse[$];
   int beta[2];

   struct {
      int unsigned k; // index
      int unsigned n; // counter

      // Goertzel
      int x1[2];
      int x2[2];
   } cur, next;
   time last_update;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      channel_sample_out = new("channel_sample_out", this);
      pga_req_out = new("pga_req_out", this);
      pga_rsp_in = new("pga_rsp_in", this);
      signal_in_in = new("signal_in_in", this);
      sync_in = new("sync_in", this);
      config_cm_in = new("config_cm_in", this);

      sample = new("sample");
      last_update = $time;

      pulse = {1};
      new_pulse = {1};
   endfunction: new

   virtual function int goertzel_update(int u, int x1, int x2, int beta);
      longint partial = x1;
      partial *= 2 * beta;
      partial >>>= config_dem_packet::beta_fractional_bits; // Truncation to -inf.
      partial += u * (1 << (shift + GOERTZEL_FRACTIONAL_BITS));
      partial -= x2;
      goertzel_update = signed'(partial[GOERTZEL_WIDTH-1:0]); // Overflow.
   endfunction: goertzel_update

   virtual function int unsigned dtft_square(longint x1, longint x2, longint beta);
      static const int s1 = DTFT_LSB - GOERTZEL_MSB - 2;
      static const int s2 = DTFT_LSB - 2;
      static const int s3 = DTFT_LSB;
      static const int shift1 = s1 + GOERTZEL_FRACTIONAL_BITS;
      static const int shift2 = s2 + 2 * GOERTZEL_FRACTIONAL_BITS;
```

```
      static const int shift3 = s3 - s2;
      longint partial = 2 * beta * x2; // t1
      partial >>>= shift1 + config_dem_packet::beta_fractional_bits; // t2
      partial = shift1 >= 0 ? partial <<< shift1 : partial >>> shift1; // Adjust to
x1 units
      partial = x1 - partial; // t3
      partial *= x1; // t4
      partial >>>= shift2; // t5
      partial += (x2*x2) >>> shift2; // t6, t7
      partial >>>= shift3; // t8.1
      dtft_square = partial >= 0 ? partial[DTFT_MSB-DTFT_LSB:0] : 0; // t8.2, value
   endfunction: dtft_square

   virtual function void update();
      if (last_update != $time) begin
         if (do_write) begin // next.n == 0
            sample.space_value = dtft_square(cur.x1[0], cur.x2[0], beta[0]);
            sample.mark_value  = dtft_square(cur.x1[1], cur.x2[1], beta[1]);
            channel_sample_out.write(sample);
         end
         // No actual need to use copying. Done for simplicity.
         if (get_pulse) // next.n == 0 && next.k == 0
            pulse = new_pulse;
         cur = next;
         last_update = $time;
      end

      if (cur.n + 1 >= pulse[cur.k]) begin
         next.n = 0;
         do_write = 1;
         foreach (next.x1[i]) begin
            next.x1[i] = u * (1 << shift);
            next.x2[i] = 0;
         end
         if (cur.k + 1 >= pulse.size()) begin
            next.k = 0;
            get_pulse = 1;
         end
         else begin
            next.k = cur.k + 1;
            get_pulse = 0;
         end
      end
      else begin
         next.n = cur.n + 1;
         next.k = cur.k;
         do_write = 0;
         get_pulse = 0;
         foreach (next.x1[i]) begin
            next.x1[i] = goertzel_update(u, cur.x1[i], cur.x2[i], beta[i]);
            next.x2[i] = cur.x1[i];
         end
      end
   endfunction: update

   virtual function void write_pga(pga_packet tx);
      shift = tx.shift;
      update();
   endfunction: write_pga

   virtual function void write_signal_in(signal_in_packet tx);
      u = tx.data;
      update();
   endfunction: write_signal_in

   virtual function void write_sync(sync_packet tx);
      static bit first = 1'b1;

      if (first)
         new_pulse = {tx.data};
      else
         new_pulse.push_back(tx.data);
```

```
      if (tx.last)
         first = 1'b1;
      else
         first = 1'b0;

      if (tx.last) // Remove this condition if possible.
         update();
   endfunction: write_sync

   virtual function void write_config_cm(config_dem_packet tx);
      beta[0] = tx.beta_space;
      beta[1] = tx.beta_mark;
      update();
   endfunction: write_config_cm
endclass
```

```
`uvm_analysis_imp_decl(_channel_sample)
`uvm_analysis_imp_decl(_config_lu)

// TODO: Fixed-point multiplications
// TODO: Allow configurable delays before calls to can_put and try_put
// TODO: Port for monitoring state.
class lu_refmod extends uvm_component;
   `uvm_component_utils(lu_refmod)

   uvm_put_port #(dem_dec_packet) dem_dec_out;
   uvm_analysis_imp_channel_sample #(channel_sample_packet, lu_refmod) channel_sampl
e_in;
   uvm_analysis_imp_config_lu #(config_dem_packet, lu_refmod) config_lu_in;

   localparam HEADER_WIDTH = 32;
   localparam bit[HEADER_WIDTH-1:0] HEADER = 32'hAAAA54C7; // Check endianness
   localparam INITIALIZATION_DELAY = HEADER_WIDTH;
   localparam PAYLOAD_WIDTH = dem_dec_packet::DATA_WIDTH;
   localparam PAUSE_DELAY = 24;

   static const lu_float A[0:16] = '{
      '{1'b0, 'sd0, 10'd919},
      '{1'b0, 'sd0, 10'd769},
      '{1'b0, 'sd0, 10'd638},
      '{1'b0, -'sd1, 10'd1001},
      '{1'b0, -'sd1, 10'd753},
      '{1'b0, -'sd1, 10'd553},
      '{1'b0, -'sd2, 10'd803},
      '{1'b0, -'sd2, 10'd577},
      '{1'b0, -'sd3, 10'd826},
      '{1'b0, -'sd3, 10'd589},
      '{1'b0, -'sd4, 10'd837},
      '{1'b0, -'sd4, 10'd594},
      '{1'b0, -'sd5, 10'd843},
      '{1'b0, -'sd5, 10'd597},
      '{1'b0, -'sd6, 10'd846},
      '{1'b0, -'sd6, 10'd598},
      '{1'b0, -'sd7, 10'd847}};

   static const lu_float B[0:16] = '{
      '{1'b0, 'sd0, 10'd0},
      '{1'b0, -'sd3, 10'd599},
      '{1'b0, -'sd2, 10'd821},
      '{1'b0, -'sd1, 10'd963},
      '{1'b0, 'sd0, 10'd977},
      '{1'b0, 'sd1, 10'd888},
      '{1'b0, 'sd2, 10'd748},
      '{1'b0, 'sd3, 10'd600},
      '{1'b0, 'sd3, 10'd928},
      '{1'b0, 'sd4, 10'd701},
      '{1'b0, 'sd5, 10'd521},
      '{1'b0, 'sd5, 10'd764},
      '{1'b0, 'sd6, 10'd555},
      '{1'b0, 'sd6, 10'd800},
      '{1'b0, 'sd7, 10'd574},
      '{1'b0, 'sd7, 10'd822},
      '{1'b0, 'sd8, 10'd586}};

   typedef enum
   {
      INIT,
      LFH,
      LFD,
      RECEIVE_FIRST,
      RECEIVE_REMAINING,
      PAUSE
   } state_e;
   typedef struct {
      state_e s;
      int count;
   } state_t;
   state_t cur, next;
   time last_update;
```

```
   longint unsigned buffer[2][$:INITIALIZATION_DELAY];
   longint unsigned a[2][2]; // accumulate
   lu_float num_cur, den_cur;
   longint unsigned m[2][2]; // maximum
   lu_float num_max, den_max;
   longint unsigned noise[2];
   longint unsigned signal[2];
   int header_end;
   bit can_put;
   lu_float header_threshold;
   dem_dec_packet packet_out;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      dem_dec_out = new("dem_dec_out", this);
      channel_sample_in = new("channel_sample_in", this);
      config_lu_in = new("config_lu_in", this);
      // TODO: initialize
      last_update = 0;
   endfunction: new

   //--- Auxiliary methods
   //-- Input manipulation
   virtual function void rotate_buffers(channel_sample_packet tx);
      assert( buffer[0].size() == HEADER_WIDTH );
      assert( buffer[1].size() == HEADER_WIDTH );
      buffer[0].pop_back();
      buffer[0].push_front(tx.space_value);
      buffer[1].pop_back();
      buffer[1].push_front(tx.mark_value);
   endfunction: rotate_buffers

   virtual function void clear_buffers();
      foreach (buffer[i])
         buffer[i] = {};
   endfunction: clear_buffers

   virtual function void compute_statistics();
      lu_float af[2][2];
      // irun 12.10: Hierarchical name component lookup fail at 'index'
      //foreach (a[i, j])
      //   a[i][j] = buffer[i].sum with (HEADER[item.index] == j ? item : 0);

      foreach(a[i, j])
         a[i][j] = 0;
      foreach(buffer[i, k])
         a[i][HEADER[k]] += buffer[i][k];
      foreach(af[i, j])
         af[i][j] = make_lu_float(a[i][j]);

      //num_cur = af[0][1] * af[1][1] + af[1][0] * af[0][0];
      //den_cur = af[0][1] * af[1][0];
      num_cur = add(mul(af[0][1], af[1][1]), mul(af[1][0], af[0][0]));
      den_cur = mul(af[0][1], af[1][0]);
   endfunction: compute_statistics

   //-- State logic
   // next.count should be updated before.
   virtual function void new_header_end();
      header_end = next.count; // Header end after LFD
      foreach (m[i, j])
         m[i][j] = a[i][j];
      num_max = num_cur;
      den_max = den_cur;
   endfunction: new_header_end

   virtual function bit header_fit_condition();
      header_fit_condition =
         //num_cur > header_threshold * den_cur;
         greater(num_cur, mul(header_threshold, den_cur));
   endfunction: header_fit_condition
```

```
   virtual function void set_parameters();
      foreach (noise[i]) begin
         noise[i] = m[i][1-i];
         signal[i] = m[i][i] > m[i][i-1]? m[i][i] - m[i][1-i] : 0;
      end
   endfunction: set_parameters

   // Positive part of floor(log2(x / y))
   // TODO: Replace by floating point operation.
   virtual function int pos_floor_log2_quot(longint unsigned x, longint unsigned y);
      if (y == 0) begin
         pos_floor_log2_quot = (-1)>>1; // INT_MAX
      end
      else begin
         pos_floor_log2_quot = 0;
         x /= 2;
         while (x >= y) begin
            ++pos_floor_log2_quot;
            x /= 2;
         end
      end
   endfunction: pos_floor_log2_quot

   virtual function bit bit_estimate(longint unsigned r_raw[2]);
      longint unsigned r[2]; // Normalised input value
      int e[2];
      lu_float fnoise[2];
      lu_float fsignal[2];
      lu_float fr[2];

      foreach(r[i]) begin
         r[i] = r_raw[i] * (HEADER_WIDTH / 2);
      end

      foreach(noise[i]) begin
         e[i] = pos_floor_log2_quot(4 * signal[i] * r[i], noise[i]**2);
      end
      foreach(r[i]) begin
         fnoise[i] = make_lu_float(noise[i]);
         fsignal[i] = make_lu_float(signal[i]);
         fr[i] = make_lu_float(r[i]);
      end
      if (e.and() with (item >= $low(A) && item <= $high(A))) begin
         lu_float coeff_a[2];
         lu_float coeff_b[2];
         lu_float lnum[2];
         lu_float lden[2];
         foreach(e[i]) begin
            coeff_a[i] = A[e[i]];
            coeff_b[i] = B[e[i]];

            lden[i] = mul(fnoise[i], fnoise[i]);
            // coeff_a * (signal * r) + coeff_b * noise^2 - signal * noise;
            lnum[i] = sub(
                        add(
                           mul(mul(fsignal[i], fr[i]), coeff_a[i]),
                           mul(coeff_b[i], lden[i])),
                        mul(fsignal[i], fnoise[i]));
         end
         bit_estimate =
            greater(
               mul(lnum[1], lden[0]),
               mul(lnum[0], lden[1]));
      end
      else if (e[0] > e[1]) begin
         bit_estimate = 4 * r[0] < signal[0] + 4 * noise[0]; // ASK
      end
      else if (e[1] > e[0]) begin
         bit_estimate = 4 * r[1] >= signal[1] + 4 * noise[1]; // ASK
      end
      else begin
         bit_estimate =
            (4 * r[1] + signal[0] + 4 * noise[0]) >=
```

```
            (4 * r[0] + signal[1] + 4 * noise[1]);
      end
   endfunction: bit_estimate

   virtual function void write_channel_sample(channel_sample_packet tx);
      const state_t state = get_state();
      unique case (state.s)
         INIT: begin
            assert( buffer[0].size() < INITIALIZATION_DELAY );
            buffer[0].push_front(tx.space_value);
            buffer[1].push_front(tx.mark_value);
            compute_statistics();
         end
         LFH: begin
            rotate_buffers(tx);
            compute_statistics();
         end
         LFD: begin
            rotate_buffers(tx);
            compute_statistics();
         end
         RECEIVE_FIRST: begin
            packet_out = new("packet_out");
            set_parameters();
            for (int i = 0; i < header_end; ++i) begin
               packet_out.data[PAYLOAD_WIDTH - 1 - i] =
                  bit_estimate({buffer[0][header_end - i - 1], buffer[1][header_end
- i - 1]});
            end
            packet_out.data[PAYLOAD_WIDTH - 1 - header_end] =
               bit_estimate({tx.space_value, tx.mark_value});
            clear_buffers();
         end
         RECEIVE_REMAINING: begin
            packet_out.data[PAYLOAD_WIDTH - 1 - state.count] =
               bit_estimate({tx.space_value, tx.mark_value});
            if ((state.count == (PAYLOAD_WIDTH - 1)) && can_put)
               assert( dem_dec_out.try_put(packet_out) );
         end
         PAUSE: begin
         end
      endcase
      update_state();
   endfunction: write_channel_sample

   virtual function state_t get_state();
      if (last_update != $time)
         get_state = next;
      else
         get_state = cur;
   endfunction: get_state

   virtual function void update_state();
      if (last_update != $time) begin
         cur = next;
         last_update = $time;
      end
      next = cur;
      unique case (cur.s)
         INIT: begin
            if (buffer[0].size() < INITIALIZATION_DELAY)
               next.s = INIT;
            else if (header_fit_condition()) begin
               next.s = LFD;
               next.count = HEADER_WIDTH-1;
               new_header_end();
            end
            else
               next.s = LFH;
         end
         LFH: begin
            if (header_fit_condition()) begin
               next.s = LFD;
```

```
                next.count = HEADER_WIDTH-1;
                new_header_end();
             end
             else
                next.s = LFH;
          end
          LFD: begin
             next.s = LFD;
             next.count = cur.count - 1;
             //if (num_cur * den_max > num_max * den_cur)
             if (greater(mul(num_cur, den_max), mul(num_max, den_cur)))
                new_header_end();
             if (next.count > 0)
                next.s = LFD;
             else begin
                next.s = RECEIVE_FIRST;
                can_put = dem_dec_out.can_put();
             end
          end
          RECEIVE_FIRST: begin
             next.count = header_end + 1;
             next.s = RECEIVE_REMAINING;
          end
          RECEIVE_REMAINING: begin
             if (cur.count < (PAYLOAD_WIDTH - 1)) begin
                next.s = RECEIVE_REMAINING;
                next.count = cur.count + 1;
             end
             else begin
                next.s = PAUSE;
                next.count = 0;
             end
          end
          PAUSE: begin
             next.count = cur.count + 1;
             if (next.count < PAUSE_DELAY)
                next.s = PAUSE;
             else begin
                next.s = INIT;
                assert( buffer[0].size() == 0 );
                assert( buffer[1].size() == 0 );
             end
          end
       endcase
   endfunction: update_state

   virtual function void write_config_lu(config_dem_packet tx);
      header_threshold = tx.header_threshold;
   endfunction: write_config_lu
endclass
```

```
/**
 * A simple floating point for the Likelihood Unit.
 *
 * No hidden bit.
 * Signed exponent, no offset.
 * No special values.
 * Addition truncates towards negative infinity.
 * Multiplication truncates towards zero.
 * All operations assume normalised inputs and deliver normalised outputs.
 * Zero is normalised to positive zero, with minimum exponent.
 */
typedef struct packed
{
   bit sign;
   bit signed [7:0] exponent;
   bit [9:0] mantissa;
} lu_float;


// TODO: Rename functions. SystemVerilog doesn't support overloading.

function real lu_float_to_real(lu_float x);
   lu_float_to_real = x.mantissa;
   lu_float_to_real *= 2.0 ** (x.exponent - $bits(x.mantissa));
   lu_float_to_real = x.sign ? -lu_float_to_real : lu_float_to_real;
endfunction: lu_float_to_real


function lu_float make_lu_float(int unsigned x);
   int i = $bits(x) - 1;
   make_lu_float.sign = 1'b0;

   if (x == 0) begin
      make_lu_float.exponent =
         {1'b1, {($bits(make_lu_float.exponent)-1){1'b0}}}; // Minimum exponent
      make_lu_float.mantissa = '0;
   end
   else begin
      make_lu_float.exponent = $bits(x);
      while (x[$bits(x) - 1] != 1'b1) begin
         --make_lu_float.exponent;
         x <<= 1;
      end
      make_lu_float.mantissa = (x >> ($bits(x) - $bits(make_lu_float.mantissa)));
   end
endfunction: make_lu_float


function lu_float negate(lu_float x);
   negate = x;
   negate.sign = ~x.sign;
endfunction: negate


function lu_float add(lu_float x, lu_float y);
   // We take advantage of the fact that normalised zero has minimum exponent.
   int delta; // = x.exponent - y.exponent;
   int max_shift; // = $bits(y.mantissa) - 1;
   int shift; // = delta >= max_shift ? max_shift : delta;
   int mx; // = x.sign ? -x.mantissa : x.mantissa;
   int my; // = y.sign ? -y.mantissa : y.mantissa;
   int sum;

   if (x.exponent < y.exponent)
      return add(y, x);

   delta = x.exponent - y.exponent;
   max_shift = $bits(y.mantissa) - 1;
   shift = delta >= max_shift ? max_shift : delta;
   mx = x.sign ? -x.mantissa : x.mantissa;
   my = y.sign ? -y.mantissa : y.mantissa;

   sum = mx + (my >>> shift);
```

```
   add.sign = (sum < 0);

   // Overflow
   if (sum[$bits(x.mantissa)] != sum[$bits(x.mantissa) + 1]) begin
      sum >>>= 1;
      assert( sum[$bits(x.mantissa)] == sum[$bits(x.mantissa) + 1] )
      assert( x.sign == y.sign );
      add.exponent = x.exponent + 1;
      add.mantissa = add.sign ? -sum : sum;
      return add;
   end

   // Zero
   if (sum == 0) begin
      add = make_lu_float(0);
      return add;
   end

   add.exponent = x.exponent;
   add.mantissa = add.sign ? -sum : sum;

   // Normalise.
   while (add.mantissa[$bits(add.mantissa) - 1] == 1'b0) begin
      add.mantissa <<<= 1;
      --add.exponent;
   end
endfunction: add


function lu_float sub(lu_float x, lu_float y);
   sub = add(x, negate(y));
endfunction: sub


function lu_float mul(lu_float x, lu_float y);
   int m;
   m = x.mantissa * y.mantissa;

   mul.sign = x.sign ^ y.sign;

   if (m == 0) begin
      mul = make_lu_float(0);
   end
   else if (m[2 * $bits(x.mantissa) - 1] == 1'b1) begin
      mul.exponent = x.exponent + y.exponent;
      mul.mantissa = (m >> $bits(x.mantissa));
   end
   else begin
      assert( m[2 * $bits(mul.mantissa) - 2] == 1'b1 );
      mul.exponent = x.exponent + y.exponent - 1;
      mul.mantissa = (m >> ($bits(x.mantissa) - 1));
      assert( mul.mantissa[$bits(mul.mantissa) - 1] == 1'b1 );
   end
endfunction: mul


function bit less(lu_float x, lu_float y);
   less =
      (y.sign < x.sign) ||
      ((y.sign == x.sign) &&
       ((x.exponent < y.exponent) ||
        ((x.exponent == y.exponent) &&
         (x.mantissa < y.mantissa))));
endfunction: less


function bit greater(lu_float x, lu_float y);
   greater = less(y, x);
endfunction: greater


function bit less_equal(lu_float x, lu_float y);
```

```
   less_equal = !less(y, x);
endfunction: less_equal


function bit greater_equal(lu_float x, lu_float y);
   greater_equal = !less(x, y);
endfunction: greater_equal


/* Operator overloading not yet supported by vcs-G-2012.9
bind - function lu_float negate(lu_float);
bind + function lu_float add(lu_float, lu_float);
bind - function lu_float sub(lu_float, lu_float);
bind * function lu_float mul(lu_float, lu_float);
bind < function lu_float less(lu_float, lu_float);
bind > function lu_float greater(lu_float, lu_float);
bind <= function lu_float less_equal(lu_float, lu_float);
bind >= function lu_float greater_equal(lu_float, lu_float);
*/
```

```
class dem_single_refmod_env extends uvm_env;
   `uvm_component_utils(dem_single_refmod_env)

   dem_refmod dem;

   bvm_delayed_writer #(signal_in_packet) signal_in_writer;
   uvm_sequencer #(sync_packet_a) sync_a_sequencer;
   bvm_delayed_writer #(sync_packet) sync_writer;
   bvm_delayed_writer #(pga_packet) pga_rsp_writer;
   bvm_delayed_writer #(config_dem_packet) config_dem_writer;

   bvm_sink #(dem_dec_packet) dem_dec_sink;
   bvm_simple_recorder #(pga_packet) pga_req_recorder;

   uvm_tlm_fifo #(dem_dec_packet) dem_dec_fifo;

   uvm_tlm_time clock_period;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      dem_dec_fifo = new("dem_dec_fifo", this);
   endfunction: new

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      dem = dem_refmod::type_id::create("dem", this);

      signal_in_writer = bvm_delayed_writer #(signal_in_packet)::type_id::create("si
gnal_in_writer", this);
      sync_a_sequencer = uvm_sequencer #(sync_packet_a)::type_id::create("sync_a_seq
uencer", this);
      sync_writer = bvm_delayed_writer #(sync_packet)::type_id::create("sync_writer"
, this);
      pga_rsp_writer = bvm_delayed_writer #(pga_packet)::type_id::create("pga_rsp_wr
iter", this);
      config_dem_writer = bvm_delayed_writer #(config_dem_packet)::type_id::create("
config_dem_writer", this);

      dem_dec_sink = bvm_sink #(dem_dec_packet)::type_id::create("dem_dec_sink", thi
s);
      pga_req_recorder = bvm_simple_recorder #(pga_packet)::type_id::create("pga_req
_recorder", this);

      //config
      clock_period = new("clock_period");
      clock_period.incr(1us, 1ns);
      uvm_config_db #(uvm_tlm_time)::set(this, "*", "clock_period", clock_period);

      uvm_config_db #(uvm_tlm_time)::set(this.dem_dec_sink, "", "delay", clock_perio
d);
      uvm_config_db #(int unsigned)::set(this.dem_dec_sink, "", "min_transactions",
3);
   endfunction: build_phase

   virtual function void connect_phase(uvm_phase phase);
      signal_in_writer.write_port.connect(dem.signal_in_in);
      sync_writer.write_port.connect(dem.sync_in);
      pga_rsp_writer.write_port.connect(dem.pga_rsp_in);
      config_dem_writer.write_port.connect(dem.config_dem_in);

      dem.dem_dec_out.connect(dem_dec_fifo.put_export);
      dem_dec_sink.get_port.connect(dem_dec_fifo.get_export);

      dem.pga_req_out.connect(pga_req_recorder.analysis_export);
   endfunction: connect_phase

   virtual task run_phase(uvm_phase phase);
      // Set up translation of upstream sequence from sync_a_sequencer to
      // sync_writer.
      sync_packet_a_to_sync_packet_seq sync_seq;

      sync_seq = sync_packet_a_to_sync_packet_seq::type_id::create("sync_seq");
```

```
      sync_seq.up_sequencer = sync_a_sequencer;
      sync_seq.start(sync_writer.m_sequencer);
   endtask: run_phase
endclass
```

```
class dem_single_refmod_basic_test extends uvm_test;
   `uvm_component_utils(dem_single_refmod_basic_test)

   dem_single_refmod_env env;

   function new(string name, uvm_component parent);
       super.new(name, parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      env = dem_single_refmod_env::type_id::create("env", this);
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
      // Sequences
      signal_in_white_noise_seq signal_in_seq = signal_in_white_noise_seq::type_id::
create("signal_in_seq", this);
      sync_infinite_regular_seq sync_seq = sync_infinite_regular_seq::type_id::creat
e("sync_seq", this);
      pga_max_shift_dumb_seq pga_rsp_seq = pga_max_shift_dumb_seq::type_id::create("
pga_rsp_seq", this);
      config_dem_low_threshold_simple_seq config_dem_seq = config_dem_low_threshold_
simple_seq::type_id::create("config_dem_seq", this);

      fork
         signal_in_seq.start(env.signal_in_writer.m_sequencer);
         sync_seq.start(env.sync_a_sequencer);
         pga_rsp_seq.start(env.pga_rsp_writer.m_sequencer);
         config_dem_seq.start(env.config_dem_writer.m_sequencer);
      join
   endtask: run_phase
endclass: dem_single_refmod_basic_test
```

```
`include "uvm_macros.svh"
import uvm_pkg::*;

`include "./lu_float.svh"

`include "./signal_in_packet.svh"
`include "./sync_packet.svh"
`include "./sync_packet_a.svh"
`include "./pga_packet.svh"
`include "./config_dem_packet.svh"
`include "./channel_sample_packet.svh"
`include "./dem_dec_packet.svh"

`include "./signal_in_seq_lib.svh"
`include "./sync_seq_lib.svh"
`include "./pga_seq_lib.svh"
`include "./config_dem_seq_lib.svh"

`include "./cm_refmod.svh"
`include "./lu_refmod.svh"
`include "./dem_refmod.svh"
`include "./bvm_delayed_writer.svh"
`include "./bvm_simple_recorder.svh"
`include "./bvm_sink.svh"
`include "./dem_single_refmod_env.svh"

`include "./dem_single_refmod_basic_test.svh"

module dem_single_refmod_top;
   initial run_test();
endmodule
```

```
class cm_single_refmod_env extends uvm_env;
   `uvm_component_utils(cm_single_refmod_env)

   cm_refmod cm;

   bvm_delayed_writer #(signal_in_packet) signal_in_writer;
   uvm_sequencer #(sync_packet_a) sync_a_sequencer;
   bvm_delayed_writer #(sync_packet) sync_writer;
   bvm_delayed_writer #(pga_packet) pga_rsp_writer;
   bvm_delayed_writer #(config_dem_packet) config_dem_writer;

   bvm_simple_recorder #(channel_sample_packet) channel_sample_recorder;
   bvm_simple_recorder #(pga_packet) pga_req_recorder;

   uvm_tlm_time clock_period;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      cm = cm_refmod::type_id::create("cm", this);

      signal_in_writer = bvm_delayed_writer #(signal_in_packet)::type_id::create("si
gnal_in_writer", this);
      sync_writer = bvm_delayed_writer #(sync_packet)::type_id::create("sync_writer"
, this);
      sync_a_sequencer = uvm_sequencer #(sync_packet_a)::type_id::create("sync_a_seq
uencer", this);
      pga_rsp_writer = bvm_delayed_writer #(pga_packet)::type_id::create("pga_rsp_wr
iter", this);
      config_dem_writer = bvm_delayed_writer #(config_dem_packet)::type_id::create("
config_dem_writer", this);

      channel_sample_recorder = bvm_simple_recorder #(channel_sample_packet)::type_i
d::create("channel_sample_packet", this);
      pga_req_recorder = bvm_simple_recorder #(pga_packet)::type_id::create("pga_req
_recorder", this);

      //config
      clock_period = new("clock_period");
      clock_period.incr(1us, 1ns);
      uvm_config_db #(uvm_tlm_time)::set(this, "*", "clock_period", clock_period);

      uvm_config_db #(int unsigned)::set(this.channel_sample_recorder, "", "min_tran
sactions", 100);
   endfunction

   virtual function void connect_phase(uvm_phase phase);
      signal_in_writer.write_port.connect(cm.signal_in_in);
      sync_writer.write_port.connect(cm.sync_in);
      pga_rsp_writer.write_port.connect(cm.pga_rsp_in);
      config_dem_writer.write_port.connect(cm.config_cm_in);

      cm.channel_sample_out.connect(channel_sample_recorder.analysis_export);
      cm.pga_req_out.connect(pga_req_recorder.analysis_export);
   endfunction

   virtual task run_phase(uvm_phase phase);
      // Set up translation of upstream sequence from sync_a_sequencer to
      // sync_writer.
      sync_packet_a_to_sync_packet_seq sync_seq;

      sync_seq = sync_packet_a_to_sync_packet_seq::type_id::create("sync_seq");
      sync_seq.up_sequencer = sync_a_sequencer;
      sync_seq.start(sync_writer.m_sequencer);
   endtask
endclass
```

```
class cm_single_refmod_basic_test extends uvm_test;
   `uvm_component_utils(cm_single_refmod_basic_test)

   cm_single_refmod_env env;

   function new(string name, uvm_component parent);
       super.new(name, parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      env = cm_single_refmod_env::type_id::create("env", this);
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
      // Sequences
      signal_in_white_noise_seq signal_in_seq = signal_in_white_noise_seq::type_id::
create("signal_in_seq", this);
      sync_infinite_regular_seq sync_seq = sync_infinite_regular_seq::type_id::creat
e("sync_seq", this);
      pga_max_shift_dumb_seq pga_rsp_seq = pga_max_shift_dumb_seq::type_id::create("
pga_rsp_seq", this);
      config_dem_simple_seq config_dem_seq = config_dem_simple_seq::type_id::create(
"config_dem_seq", this);

      fork
         signal_in_seq.start(env.signal_in_writer.m_sequencer);
         sync_seq.start(env.sync_a_sequencer);
         pga_rsp_seq.start(env.pga_rsp_writer.m_sequencer);
         config_dem_seq.start(env.config_dem_writer.m_sequencer);
      join
   endtask: run_phase
endclass: cm_single_refmod_basic_test
```

```
`include "uvm_macros.svh"
import uvm_pkg::*;

`include "./lu_float.svh"

`include "./signal_in_packet.svh"
`include "./sync_packet.svh"
`include "./sync_packet_a.svh"
`include "./pga_packet.svh"
`include "./config_dem_packet.svh"
`include "./channel_sample_packet.svh"

`include "./signal_in_seq_lib.svh"
`include "./sync_seq_lib.svh"
`include "./pga_seq_lib.svh"
`include "./config_dem_seq_lib.svh"

`include "./cm_refmod.svh"
`include "./bvm_delayed_writer.svh"
`include "./bvm_simple_recorder.svh"
`include "./cm_single_refmod_env.svh"

`include "./cm_single_refmod_basic_test.svh"

module cm_single_refmod_top;
   initial run_test();
endmodule
```

```
class lu_single_refmod_env extends uvm_env;
   `uvm_component_utils(lu_single_refmod_env)

   lu_refmod lu;

   bvm_delayed_writer #(channel_sample_packet) channel_sample_writer;
   bvm_delayed_writer #(config_dem_packet) config_dem_writer;

   bvm_sink #(dem_dec_packet) dem_dec_sink;

   uvm_tlm_fifo #(dem_dec_packet) dem_dec_fifo;

   uvm_tlm_time clock_period;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      dem_dec_fifo = new("dem_dec_fifo", this);
   endfunction

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      lu = lu_refmod::type_id::create("lu", this);

      channel_sample_writer = bvm_delayed_writer #(channel_sample_packet)::type_id::
create("channel_sample_writer", this);
      config_dem_writer = bvm_delayed_writer #(config_dem_packet)::type_id::create("
config_dem_writer", this);

      dem_dec_sink = bvm_sink #(dem_dec_packet)::type_id::create("dem_dec_sink", thi
s);

      //config
      clock_period = new("clock_period");
      clock_period.incr(1us, 1ns);
      uvm_config_db #(uvm_tlm_time)::set(this, "*", "clock_period", clock_period);

      uvm_config_db #(uvm_tlm_time)::set(this.dem_dec_sink, "", "delay", clock_perio
d);
      uvm_config_db #(int unsigned)::set(this.dem_dec_sink, "", "min_transactions",
3);
   endfunction

   virtual function void connect_phase(uvm_phase phase);
      channel_sample_writer.write_port.connect(lu.channel_sample_in);
      config_dem_writer.write_port.connect(lu.config_lu_in);

      lu.dem_dec_out.connect(dem_dec_fifo.put_export);
      dem_dec_sink.get_port.connect(dem_dec_fifo.get_export);
   endfunction
endclass
```

```
class lu_single_refmod_basic_test extends uvm_test;
   `uvm_component_utils(lu_single_refmod_basic_test)

   lu_single_refmod_env env;

   function new(string name, uvm_component parent);
       super.new(name, parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      env = lu_single_refmod_env::type_id::create("env", this);
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
      // Sequences
      channel_sample_random_subframe_or_noise_seq_forever_seq channel_sample_seq = c
hannel_sample_random_subframe_or_noise_seq_forever_seq::type_id::create("channel_sam
ple_seq", this);
      config_dem_simple_seq config_dem_seq = config_dem_simple_seq::type_id::create(
"config_dem_seq", this);

      fork
         channel_sample_seq.start(env.channel_sample_writer.m_sequencer);
         config_dem_seq.start(env.config_dem_writer.m_sequencer);
      join
   endtask: run_phase
endclass: lu_single_refmod_basic_test
```

```
class lu_single_refmod_directed_test extends lu_single_refmod_basic_test;
   `uvm_component_utils(lu_single_refmod_directed_test)

   function new(string name, uvm_component parent);
       super.new(name, parent);
   endfunction: new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      channel_sample_random_payload_seq::type_id::set_type_override(channel_sample_d
irected_payload_seq::get_type());
   endfunction: build_phase
endclass: lu_single_refmod_directed_test
```

```
`include "uvm_macros.svh"
import uvm_pkg::*;

`include "./lu_float.svh"

`include "./channel_sample_packet.svh"
`include "./config_dem_packet.svh"
`include "./dem_dec_packet.svh"

`include "./channel_sample_seq_lib.svh"
`include "./config_dem_seq_lib.svh"

`include "./lu_refmod.svh"
`include "./bvm_delayed_writer.svh"
`include "./bvm_sink.svh"
`include "./lu_single_refmod_env.svh"

`include "./lu_single_refmod_basic_test.svh"
`include "./lu_single_refmod_directed_test.svh"

module lu_single_refmod_top;
   initial run_test();
endmodule
```

```
class signal_in_packet #(parameter INPUT_WIDTH = 12) extends uvm_sequence_item;
   typedef signal_in_packet #(INPUT_WIDTH) this_type;

   rand bit signed[INPUT_WIDTH-1:0] data;

   function int unsigned delay;
      delay = 1;
   endfunction

   `uvm_object_param_utils_begin(this_type)
      `uvm_field_int(data, UVM_ALL_ON)
   `uvm_object_utils_end

   function new(string name = "signal_in_packet");
      super.new(name);
   endfunction: new

endclass: signal_in_packet
```

```
class channel_sample_packet extends uvm_sequence_item;

   rand int unsigned mark_value;
   rand int unsigned space_value;

   rand int unsigned delay;

   constraint positive_delay {
      delay > 0;
   }

   `uvm_object_utils_begin(channel_sample_packet)
      `uvm_field_int(mark_value, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(space_value, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(delay, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "channel_sample_packet");
      super.new(name);
   endfunction
endclass
```

```
//     UVM Sequence Item      //
// Created by Plateny Ponchet //

class dem_dec_packet extends uvm_sequence_item;

   localparam DATA_WIDTH = 304;

   rand bit [DATA_WIDTH-1:0]data;

   static const int unsigned data_width = DATA_WIDTH;

   `uvm_object_utils_begin(dem_dec_packet)
      `uvm_field_int(data, UVM_ALL_ON|UVM_HEX)
   `uvm_object_utils_end

   function new(string name = "dem_dec_packet");
      super.new(name);
   endfunction
endclass: dem_dec_packet
```

```
class sync_packet extends uvm_sequence_item;
    rand int unsigned data;
    bit last;
    int unsigned delay;

    `uvm_object_utils_begin(sync_packet)
        `uvm_field_int(data, UVM_ALL_ON | UVM_DEC)
        `uvm_field_int(last, UVM_DEFAULT)
        `uvm_field_int(delay, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
    `uvm_object_utils_end

    function new(string name = "sync_packet");
        super.new(name);
    endfunction: new
endclass: sync_packet
```

```
class sync_packet_a extends uvm_sequence_item;
   rand int unsigned data[];

   int unsigned delay; // Should be data.sum of previous tx.

   `uvm_object_utils_begin(sync_packet_a)
      `uvm_field_array_int(data, UVM_ALL_ON | UVM_DEC)

      `uvm_field_int(delay, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "sync_packet_a");
      super.new(name);
   endfunction: new
endclass: sync_packet_a
```

```
class config_dem_packet #(BETA_FRACTIONAL_BITS = 12) extends uvm_sequence_item;
   typedef config_dem_packet #(BETA_FRACTIONAL_BITS) this_type;

   localparam BETA_WIDTH = BETA_FRACTIONAL_BITS + 1;

   rand bit signed[BETA_WIDTH-1:0] beta_mark;
   rand bit signed[BETA_WIDTH-1:0] beta_space;
   rand lu_float header_threshold;

   rand int unsigned delay;

   static const int unsigned beta_fractional_bits = BETA_FRACTIONAL_BITS;

   constraint normalized_positive_threshold {
      header_threshold.sign == 1'b0;
      /* Not yet supported by vcs-G-2012.9:
      header_threshold.mantissa >= (1 << $bits(header_threshold.mantissa));
      */
      header_threshold.mantissa >= (1 << 9);
   }

   `uvm_object_param_utils(this_type)
   //`uvm_object_param_utils_begin(this_type)
      //`uvm_field_int(beta_mark, UVM_ALL_ON | UVM_DEC)
      //`uvm_field_int(beta_space, UVM_ALL_ON | UVM_DEC)

      //`uvm_field_int(header_threshold.sign, UVM_DEFAULT)
      //`uvm_field_int(header_threshold.exponent, UVM_DEFAULT | UVM_UNSIGNED)
      //`uvm_field_int(header_threshold.mantissa, UVM_DEFAULT | UVM_UNSIGNED)

      //`uvm_field_int(delay, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
   //`uvm_object_utils_end

   function new(string name = "config_dem_packet");
      super.new(name);
   endfunction

   virtual function void do_copy(uvm_object rhs);
      this_type tmp;
      if (!$cast(tmp, rhs)) begin
         `uvm_fatal("TypeMismatch", "")
      end
      super.do_copy(rhs);
      beta_mark = tmp.beta_mark;
      beta_space = tmp.beta_space;
      header_threshold = tmp.header_threshold;
      delay = tmp.delay;
   endfunction: do_copy

   virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
      this_type tmp;
      do_compare =
         $cast(tmp, rhs) &&
         super.do_compare(rhs, comparer) &&
         beta_mark == tmp.beta_mark &&
         beta_space == tmp.beta_space &&
         header_threshold == tmp.header_threshold;
         // delay not compared.
   endfunction: do_compare

   virtual function string convert2string();
      convert2string =
         $sformatf("{beta_mark = %f, beta_space = %f, header_threshold = %f, delay =
 %u}",
            beta_mark / (2.0 ** beta_fractional_bits),
            beta_space / (2.0 ** beta_fractional_bits),
            lu_float_to_real(header_threshold),
            delay);
   endfunction: convert2string

   virtual function void do_print(uvm_printer printer);
      super.do_print(printer);
      printer.print_generic(
```

```
         "beta_mark", "fixed", $bits(beta_mark),
         $sformatf("%f", beta_mark / (2.0 ** beta_fractional_bits)));
      printer.print_generic(
         "beta_space", "fixed", $bits(beta_space),
         $sformatf("%f", beta_space / (2.0 ** beta_fractional_bits)));
      printer.print_generic(
         "header_threshold", "lu_float", $bits(header_threshold),
         $sformatf("%f", lu_float_to_real(header_threshold)));
      printer.print_int("delay", delay, $bits(delay), UVM_UNSIGNED);
   endfunction: do_print

   virtual function void do_record(uvm_recorder recorder);
      super.do_record(recorder);
      recorder.record_generic(
         "beta_mark",
         $sformatf("%f", beta_mark / (2.0 ** beta_fractional_bits)));
      recorder.record_generic(
         "beta_space",
         $sformatf("%f", beta_space / (2.0 ** beta_fractional_bits)));
      recorder.record_generic(
         "header_threshold",
         $sformatf("%f", lu_float_to_real(header_threshold)));
      recorder.record_field("delay", delay, $bits(delay), UVM_UNSIGNED);
      /* Didn't work (vcs-G-2012.9)
      recorder.record_field_real("beta_mark", beta_mark / (2.0 ** beta_fractional_bi
ts));
      recorder.record_field_real("beta_space", beta_space / (2.0 ** beta_fractional_
bits));
      recorder.record_field_real("header_threshold", lu_float_to_real(header_thresho
ld));
      */
   endfunction: do_record
endclass
```

```
class pga_packet extends uvm_sequence_item;

   rand int unsigned shift;

   rand int unsigned delay;

   `uvm_object_utils_begin(pga_packet)
      `uvm_field_int(shift, UVM_DEFAULT | UVM_UNSIGNED)

      `uvm_field_int(delay, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "pga_packet");
      super.new(name);
   endfunction
endclass
```

```
class signal_in_white_noise_seq extends uvm_sequence #(signal_in_packet);
   `uvm_object_utils(signal_in_white_noise_seq)

   function new(string name = "signal_in_white_noise_seq");
      super.new(name);
   endfunction

   virtual task body();
      signal_in_packet tx;

      forever begin
         tx = signal_in_packet::type_id::create("tx");
         start_item(tx);
         assert( tx.randomize() );
         finish_item(tx);
      end
   endtask
endclass
```

```
class channel_sample_uniform_packet extends channel_sample_packet;
   // knobs
   int unsigned min_mark_value = 0;
   int unsigned max_mark_value = 1000;
   int unsigned min_space_value = 0;
   int unsigned max_space_value = 1000;
   int unsigned max_delay = 1000;

   constraint value_distribution {
      mark_value inside { [min_mark_value : max_mark_value] };
      space_value inside { [min_space_value : max_space_value] };
   }

   constraint delay_distribution {
      delay <= max_delay;
   }

   `uvm_object_utils_begin(channel_sample_uniform_packet)
      `uvm_field_int(min_mark_value, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
      `uvm_field_int(max_mark_value, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
      `uvm_field_int(min_space_value, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
      `uvm_field_int(max_space_value, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
      `uvm_field_int(max_delay, UVM_DEFAULT | UVM_NOCOMPARE | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "channel_sample_uniform_packet");
      super.new(name);
   endfunction
endclass: channel_sample_uniform_packet


class channel_sample_noise_seq extends uvm_sequence #(channel_sample_uniform_packet)
;
   int unsigned max_length = 1000;
   int unsigned min_length = 0;
   // TODO: Make following parameters random.
   int unsigned mark_noise = 1000;
   int unsigned space_noise = 1000;

   rand int unsigned length;

   constraint length_distribution {
      length inside { [min_length : max_length] };
   }

   `uvm_object_utils_begin(channel_sample_noise_seq)
      `uvm_field_int(mark_noise, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(space_noise, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(max_length, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(min_length, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(length, UVM_DEFAULT | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "channel_sample_noise_seq");
      super.new(name);
   endfunction

   virtual task body();
      channel_sample_uniform_packet tx;

      repeat (length) begin
         tx = channel_sample_uniform_packet::type_id::create("tx");
         start_item(tx);
         tx.min_mark_value = 0;
         tx.max_mark_value = mark_noise;
         tx.min_space_value = 0;
         tx.max_space_value = space_noise;
         assert( tx.randomize() );
         finish_item(tx);
      end
   endtask: body
endclass: channel_sample_noise_seq
```

```
class channel_sample_pause_seq extends channel_sample_noise_seq;
   `uvm_object_utils(channel_sample_pause_seq)

   localparam PAUSE_DELAY = 24;

   constraint length_distribution {
      length == PAUSE_DELAY;
   }

   function new(string name = "channel_sample_pause_seq");
      super.new(name);
   endfunction
endclass: channel_sample_pause_seq


class channel_sample_signal_base_seq extends uvm_sequence #(channel_sample_uniform_p
acket);
   // knobs
   // TODO: knobs. Make following parameters random.
   int unsigned mark_noise = 1000;
   int unsigned mark_signal = 100_000;
   int unsigned space_noise = 1000;
   int unsigned space_signal = 100_000;
   rand bit pattern[]; // Has to be explicitly configured

   `uvm_object_utils_begin(channel_sample_signal_base_seq)
      `uvm_field_int(mark_noise, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(mark_signal, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(space_noise, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(space_signal, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_array_int(pattern, UVM_DEFAULT)
   `uvm_object_utils_end

   function new(string name = "channel_sample_signal_base_seq");
      super.new(name);
   endfunction

   virtual task body();
      channel_sample_uniform_packet tx;

      foreach (pattern[i]) begin
         tx = channel_sample_uniform_packet::type_id::create("tx");
         start_item(tx);
         if (pattern[i]) begin
            tx.min_mark_value = mark_noise > mark_signal ? 0 : mark_signal - mark_no
ise;
            tx.max_mark_value = mark_signal + mark_noise;
            tx.min_space_value = 0;
            tx.max_space_value = space_noise;
         end
         else begin
            tx.min_mark_value = 0;
            tx.max_mark_value = mark_noise;
            tx.min_space_value = space_noise > space_signal ? 0 : space_signal - spa
ce_noise;
            tx.max_space_value = space_signal + space_noise;
         end
         assert( tx.randomize() );
         finish_item(tx);
      end
   endtask: body
endclass: channel_sample_signal_base_seq


class channel_sample_header_seq extends channel_sample_signal_base_seq;
   `uvm_object_utils(channel_sample_header_seq)

   static const int header_pattern = 32'hAAAA54C7;

   constraint pattern_distribution {
      pattern.size() == 32;
      foreach (pattern[i])
```

```
         pattern[i] == header_pattern[31 - i];
   }

   function new(string name = "channel_sample_header_seq");
      super.new(name);
  endfunction
endclass: channel_sample_header_seq


class channel_sample_random_signal_seq extends channel_sample_signal_base_seq;
   // knobs
   int unsigned percent_mark = 50;
   int unsigned max_length = 1000;
   int unsigned min_length = 0;

   rand int length;

   constraint length_distribution {
      length inside { [min_length : max_length] };
   }

   constraint pattern_distribution {
      pattern.size() == length;
      foreach (pattern[i])
         pattern[i] dist {1'b1 := percent_mark, 1'b0 := (100 - percent_mark)};
   }

   `uvm_object_utils_begin(channel_sample_random_signal_seq)
      `uvm_field_int(percent_mark, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(max_length, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(min_length, UVM_DEFAULT | UVM_UNSIGNED)
   `uvm_object_utils_end


   function new(string name = "channel_sample_random_signal_seq");
      super.new(name);
   endfunction
endclass: channel_sample_random_signal_seq


class channel_sample_random_payload_seq extends channel_sample_random_signal_seq;
   `uvm_object_utils(channel_sample_random_payload_seq)

   localparam PAYLOAD_WIDTH = 304;

   constraint length_distribution {
      length == PAYLOAD_WIDTH;
   }

   function new(string name = "channel_sample_random_payload_seq");
      super.new(name);
   endfunction
endclass: channel_sample_random_payload_seq


class channel_sample_directed_payload_seq extends channel_sample_random_payload_seq;
   `uvm_object_utils(channel_sample_directed_payload_seq)

   localparam PAYLOAD_WIDTH = 304;

   // knobs
   static const bit[PAYLOAD_WIDTH-1:0] directed_pattern =
      {64'h0123456789abcdef, 104'h0, {104{1'b1}}, 32'hdeadbeef};

   constraint length_distribution {
      length == PAYLOAD_WIDTH;
   }

   constraint pattern_distribution {
      pattern.size() == length;
      foreach (pattern[i])
         pattern[i] == directed_pattern[PAYLOAD_WIDTH - 1 - i];
   }
```

```
   function new(string name = "channel_sample_directed_payload_seq");
      super.new(name);
   endfunction
endclass: channel_sample_directed_payload_seq


// Concatenation of:
//    channel_sample_header_seq
//    channel_sample_random_payload_seq
//    channel_sample_pause_seq
class channel_sample_random_subframe_seq extends uvm_sequence #(channel_sample_unifo
rm_packet);
   `uvm_object_utils(channel_sample_random_subframe_seq)

   // TODO: knobs

   function new(string name = "channel_sample_random_subframe_seq");
      super.new(name);
   endfunction

   virtual task body();
      channel_sample_header_seq header_seq;
      channel_sample_random_payload_seq payload_seq;
      channel_sample_pause_seq pause_seq;

      header_seq = channel_sample_header_seq::type_id::create("header_seq");
      assert( header_seq.randomize() );
      header_seq.start(m_sequencer, this);

      payload_seq = channel_sample_random_payload_seq::type_id::create("payload_seq"
);
      assert( payload_seq.randomize() );
      payload_seq.start(m_sequencer, this);

      pause_seq = channel_sample_pause_seq::type_id::create("pause_seq");
      assert( pause_seq.randomize() );
      pause_seq.start(m_sequencer, this);
   endtask: body
endclass: channel_sample_random_subframe_seq


class channel_sample_random_subframe_or_noise_seq extends uvm_sequence #(channel_sam
ple_uniform_packet);
   typedef enum {SUBFRAME, NOISE} event_type_e;

   // knobs
   int unsigned max_noise_length = 1000;
   int unsigned min_noise_length = 0;
   int unsigned percent_subframe = 50;

   rand event_type_e event_type;
   rand int unsigned noise_length;

   constraint noise_length_distribution {
      noise_length inside { [min_noise_length : max_noise_length] };
   }

   constraint event_type_distribution {
      event_type dist {SUBFRAME := percent_subframe, NOISE := (100 - percent_subfram
e)};
   }

   `uvm_object_utils_begin(channel_sample_random_subframe_or_noise_seq)
      `uvm_field_int(max_noise_length, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(min_noise_length, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(percent_subframe, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_enum(event_type_e, event_type, UVM_DEFAULT)
      `uvm_field_int(noise_length, UVM_DEFAULT | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "channel_sample_random_subframe_or_noise_seq");
      super.new(name);
```

```
   endfunction

   virtual task body();
      uvm_sequence #(channel_sample_uniform_packet) seq;

      unique case (event_type)
         SUBFRAME: seq = channel_sample_random_subframe_seq::type_id::create("seq");
         NOISE: seq = channel_sample_noise_seq::type_id::create("seq");
      endcase
      assert( seq.randomize() );
      seq.start(m_sequencer, this);
   endtask: body
endclass: channel_sample_random_subframe_or_noise_seq


class channel_sample_random_subframe_or_noise_seq_forever_seq extends uvm_sequence #
(channel_sample_uniform_packet);
   `uvm_object_utils(channel_sample_random_subframe_or_noise_seq_forever_seq)

   function new(string name = "channel_sample_random_subframe_or_noise_seq_forever_s
eq");
      super.new(name);
   endfunction

   virtual task body();
      channel_sample_random_subframe_or_noise_seq seq;
      static int i = 0;
      forever begin
         seq = channel_sample_random_subframe_or_noise_seq::type_id::create({"seq",
$sformatf("%6d", i)});
         assert( seq.randomize() );
         seq.start(m_sequencer, this);
      end
   endtask: body
endclass: channel_sample_random_subframe_or_noise_seq_forever_seq


class channel_sample_simple_seq extends uvm_sequence #(channel_sample_packet);
   // knobs
   int unsigned max_delay = 1000;

   `uvm_object_utils_begin(channel_sample_simple_seq)
      `uvm_field_int(max_delay, UVM_DEFAULT | UVM_UNSIGNED)
   `uvm_object_utils_end

   function new(string name = "channel_sample_simple_seq");
      super.new(name);
   endfunction

   virtual task body();
      channel_sample_packet tx;

      forever begin
         tx = channel_sample_packet::type_id::create("tx");
         start_item(tx);
         assert( tx.randomize() with {
            delay < max_delay;
         } );
         finish_item(tx);
      end
   endtask
endclass: channel_sample_simple_seq
```

```
// Translator sequence
class sync_packet_a_to_sync_packet_seq extends uvm_sequence #(sync_packet);
   `uvm_object_utils(sync_packet_a_to_sync_packet_seq)

   uvm_sequencer #(sync_packet_a) up_sequencer;

   function new(string name = "sync_packet_a_to_sync_packet_seq");
      super.new(name);
   endfunction

   virtual task body();
      sync_packet_a up_tx;
      sync_packet tx;
      forever begin
         int i = 0;
         up_sequencer.get_next_item(up_tx);
         // First item with delay
         if (up_tx.data.size() > i) begin
            tx = sync_packet::type_id::create("tx");
            start_item(tx);
            tx.data = up_tx.data[i];
            tx.last = 1'b0;
            tx.delay = up_tx.delay;
            finish_item(tx);
            ++i;
         end
         // Middle items with 0 delay
         for ( ; i < up_tx.data.size() - 1; ++i) begin
            tx = sync_packet::type_id::create("tx");
            start_item(tx);
            tx.data = up_tx.data[i];
            tx.last = 1'b0;
            tx.delay = 0;
            finish_item(tx);
         end
         // Last item with attribute bit 'last' set
         if (up_tx.data.size() > i) begin
            tx = sync_packet::type_id::create("tx");
            start_item(tx);
            tx.data = up_tx.data[i];
            tx.last = 1'b1;
            tx.delay = 0;
            finish_item(tx);
         end
         up_sequencer.item_done();
      end
   endtask: body
endclass: sync_packet_a_to_sync_packet_seq


class sync_unbalanced_seq extends uvm_sequence #(sync_packet_a);
   // knobs
   rand int unsigned max_period;
   rand int unsigned max_size;
   rand int unsigned n_packets;

   int unsigned initial_delay; // Should be data.sum of previous tx, if any.
   int unsigned next_delay; // data.sum of last tx.

   `uvm_object_utils_begin(sync_unbalanced_seq)
      `uvm_field_int(max_period, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(max_size, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(n_packets, UVM_ALL_ON | UVM_DEC)

      `uvm_field_int(initial_delay, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(next_delay, UVM_DEFAULT | UVM_UNSIGNED)
   `uvm_object_utils_end

   constraint basic_knobs_default_constraint {
      n_packets inside {[1:1000]};
      max_period inside {[100:1000]};
      max_size inside {[3:48]};
   }
```

```
   function new(string name = "sync_unbalanced_seq");
      super.new(name);
   endfunction

   virtual task body();
      sync_packet_a tx = sync_packet_a::type_id::create("tx");
      next_delay = initial_delay;

      repeat (n_packets) begin
         start_item(tx);
         assert( tx.randomize() with {
            data.size() inside {[1:max_size]};
            foreach (data[i])
               data[i] inside {[1:max_period]};
         } );
         tx.delay = next_delay;
         next_delay = tx.data.sum();
         finish_item(tx);
      end
   endtask: body
endclass: sync_unbalanced_seq


class sync_jitter_seq extends uvm_sequence #(sync_packet_a);
   // knobs
   rand int unsigned factor_select;
   rand int unsigned period;
   rand int unsigned jitter;
   rand int unsigned n_packets;

   int unsigned initial_delay; // Should be data.sum of previous tx, if any.
   int unsigned next_delay; // data.sum of last tx.

   `uvm_object_utils_begin(sync_jitter_seq)
      `uvm_field_int(factor_select, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(period, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(jitter, UVM_ALL_ON | UVM_DEC)
      `uvm_field_int(n_packets, UVM_ALL_ON | UVM_DEC)

      `uvm_field_int(initial_delay, UVM_DEFAULT | UVM_UNSIGNED)
      `uvm_field_int(next_delay, UVM_DEFAULT | UVM_UNSIGNED)
   `uvm_object_utils_end

   constraint basic_knobs_default_constraint {
      factor_select < 16;
      period inside {[100:1000]};
      jitter < period;
      n_packets < 100;
   }

   function new(string name = "sync_jitter_seq");
      super.new(name);
   endfunction

   virtual task body();
      sync_packet_a tx = sync_packet_a::type_id::create("tx");
      next_delay = initial_delay;

      tx.data = new[3 * (factor_select + 1)];

      repeat (n_packets)  begin
         tx.delay = next_delay;
         start_item(tx);
         assert( tx.randomize() with {
            foreach (data[i])
               data[i] inside {[period - jitter : period + jitter]};
         } );
         tx.delay = next_delay;
         next_delay = tx.data.sum();
         finish_item(tx);
      end
   endtask: body
```

```
endclass: sync_jitter_seq


class sync_infinite_regular_seq extends uvm_sequence #(sync_packet_a);
   `uvm_object_utils(sync_infinite_regular_seq)

   function new(string name = "sync_infinite_regular_seq");
      super.new(name);
   endfunction

   virtual task body();
      sync_jitter_seq seq = sync_jitter_seq::type_id::create("seq");
      assert( seq.randomize() with {
         jitter == 0;
      } );
      seq.n_packets = ~0>>1;
      seq.initial_delay = 0;

      forever begin
         seq.start(m_sequencer, this);
         seq.initial_delay = seq.next_delay;
      end
   endtask: body
endclass: sync_infinite_regular_seq
```

```
class config_dem_simple_seq extends uvm_sequence #(config_dem_packet);
   `uvm_object_utils(config_dem_simple_seq)

   function new(string name = "config_dem_simple_seq");
      super.new(name);
   endfunction

   virtual task body();
      config_dem_packet tx = config_dem_packet::type_id::create("tx");

      start_item(tx);
      assert( tx.randomize() with {
         delay == 0;
         header_threshold == make_lu_float(33);
      } );
      finish_item(tx);
   endtask
endclass

class config_dem_low_threshold_simple_seq extends uvm_sequence #(config_dem_packet);
   `uvm_object_utils(config_dem_low_threshold_simple_seq)

   function new(string name = "config_dem_low_threshold_simple_seq");
      super.new(name);
   endfunction

   virtual task body();
      config_dem_packet tx = config_dem_packet::type_id::create("tx");

      start_item(tx);
      assert( tx.randomize() with {
         delay == 0;
         header_threshold == make_lu_float(1);
      } );
      finish_item(tx);
   endtask
endclass
```

```
// TODO: Get rid of magic numbers.
class pga_dumb_seq extends uvm_sequence #(pga_packet);
   `uvm_object_utils(pga_dumb_seq)

   function new(string name = "pga_dumb_seq");
      super.new(name);
   endfunction

   virtual task body();
      pga_packet tx = pga_packet::type_id::create("tx");

      start_item(tx);
      assert( tx.randomize() with {
         shift < 8;
         delay == 0;
      } );
      finish_item(tx);
   endtask
endclass


class pga_max_shift_dumb_seq extends uvm_sequence #(pga_packet);
   `uvm_object_utils(pga_max_shift_dumb_seq)

   function new(string name = "pga_max_shift_dumb_seq");
      super.new(name);
   endfunction

   virtual task body();
      pga_packet tx = pga_packet::type_id::create("tx");

      start_item(tx);
      assert( tx.randomize() with {
         shift == 7;
         delay == 0;
      } );
      finish_item(tx);
   endtask
endclass
```

```
/**
 * @brief Delayed writer.
 *
 * Generates transactions of type ~T~ from an internal sequencer and writes them
 * to the uvm_analysis_port write_port, with a delay in between. ~T~ must have
 * an integer field named delay. The inter-write delay is given by the product
 * of this field and the uvm_tlm_time configuration parameter clock_period. An
 * optional initial delay is also configurable via the uvm_tlm_time
 * configuration parameter initial_delay (default 0).
 */
class bvm_delayed_writer #(type T = int) extends uvm_component;

   const static string type_name = "bvm_delayed_writer #(T)";
   typedef bvm_delayed_writer #(T) this_type;
   `uvm_component_param_utils(this_type)

   typedef uvm_sequencer #(T) sequencer_t;

   uvm_analysis_port #(T) write_port;

   sequencer_t m_sequencer;

   uvm_tlm_time initial_delay;
   uvm_tlm_time clock_period;

   localparam time CLOCK_PERIOD_DEFAULT = 10ns;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      write_port = new("write_port", this);
   endfunction: new

   virtual function string get_type_name();
      return type_name;
   endfunction: get_type_name

   virtual function sequencer_t get_sequencer();
      return m_sequencer;
   endfunction: get_sequencer

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if (!uvm_config_db #(uvm_tlm_time)::get(this, "", "initial_delay", initial_del
ay))
         initial_delay = new("initial_delay");
      if (!uvm_config_db #(uvm_tlm_time)::get(this, "", "clock_period", clock_period
)) begin
         clock_period = new("clock_period");
         clock_period.incr(CLOCK_PERIOD_DEFAULT, 1ns);
         `uvm_info("DEFAULT", $sformatf("Using default value for clock_period: %g ns
", clock_period.get_abstime(1e-9)), UVM_HIGH)
      end

      m_sequencer = sequencer_t::type_id::create("m_sequencer", this);
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
      T tx;
      time boundary;

      #(initial_delay.get_realtime(1ns));
      forever begin
         boundary = $time;
         m_sequencer.seq_item_export.get_next_item(tx);
         if ($time != boundary)
            `uvm_warning("DELAY", "Delay in get_next_item. Possible desynchronizatio
n.")
         #(tx.delay * clock_period.get_realtime(1ns));
         write_port.write(tx);
         m_sequencer.seq_item_export.item_done();
      end
   endtask: run_phase
endclass
```

```
/**
 * @brief Simple subscriber.
 *
 * Records transactions of type ~T~ observed with the analysis export
 * analysis_export. It also raises an objection until it has received a number
 * of transactions, specified by the unsigned int configuration parameter
 * min_transactions (default 0).
 */
class bvm_simple_recorder #(type T = int) extends uvm_subscriber #(T);

   const static string type_name = "bvm_simple_recorder #(T)";
   typedef bvm_simple_recorder #(T) this_type;
   `uvm_component_param_utils(this_type)

   int unsigned min_transactions;

   int unsigned n_transactions;
   uvm_phase running_phase;
   uvm_objection phase_done;
   T cur_tx;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      n_transactions = 0;
      cur_tx = new;
   endfunction: new

   virtual function string get_type_name();
      return type_name;
   endfunction: get_type_name

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if (!uvm_config_db #(int unsigned)::get(this, "", "min_transactions", min_tran
sactions))
         min_transactions = 0;
   endfunction: build_phase

   virtual function void write(T t);
      if (n_transactions)
         end_tr(cur_tx);
      ++n_transactions;
      if (running_phase != null && n_transactions >= min_transactions) begin
         running_phase.drop_objection(this);
         running_phase = null;
      end
      cur_tx.copy(t);
      assert( begin_tr(cur_tx, this.get_name()) );
   endfunction: write

   virtual task run_phase(uvm_phase phase);
      if (min_transactions) begin
         phase.raise_objection(this);
         running_phase = phase;
      end
   endtask: run_phase
endclass
```

```
/**
 * @brief Simple sink.
 *
 * Receives transactions of type ~T~ from the uvm_blocking_get_port get_port
 * and records them, with a delay in between. This delay is determined by the
 * uvm_tlm_time configuration parameter delay. It also raises an objection
 * until it has received a number of transactions, specified by the unsigned
 * int configuration parameter min_transactions. After that, it keeps accepting
 * (and recording) transactions indefinitely.
 */
class bvm_sink #(type T = int) extends uvm_component;

   const static string type_name = "bvm_sink #(T)";
   typedef bvm_sink #(T) this_type;
   `uvm_component_param_utils(this_type)

   uvm_blocking_get_port #(T) get_port;

   int unsigned min_transactions;
   uvm_tlm_time delay;

   localparam int unsigned MIN_TRANSACTIONS_DEFAULT = 100;
   localparam time DELAY_DEFAULT = 10ns;

   function new(string name, uvm_component parent);
      super.new(name, parent);
      get_port = new("get_port", this);
   endfunction: new

   virtual function string get_type_name();
      return type_name;
   endfunction: get_type_name

   virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if (!uvm_config_db #(int unsigned)::get(this, "", "min_transactions", min_tran
sactions)) begin
         `uvm_info("DEFAULT", $sformatf("Using default value for min_transactions: %
d", MIN_TRANSACTIONS_DEFAULT), UVM_HIGH)
         min_transactions = MIN_TRANSACTIONS_DEFAULT;
      end
      if (!uvm_config_db #(uvm_tlm_time)::get(this, "", "delay", delay)) begin
         delay = new("delay");
         delay.incr(DELAY_DEFAULT, 1ns);
         `uvm_info("DEFAULT", $sformatf("Using default value for delay: %g ns", dela
y.get_abstime(1e-9)), UVM_HIGH)
      end
   endfunction: build_phase

   virtual task run_phase(uvm_phase phase);
      T tx;

      phase.raise_objection(this);
      repeat (min_transactions) begin
         get_port.get(tx);
         assert( begin_tr(tx, this.get_name()) );
         #(delay.get_realtime(1ns));
         end_tr(tx);
      end
      phase.drop_objection(this);

      // Keep accepting transactions until simulation finishes.
      forever begin
         get_port.get(tx);
         assert( begin_tr(tx, this.get_name()) );
         #(delay.get_realtime(1ns));
         end_tr(tx);
      end
   endtask: run_phase
endclass
```