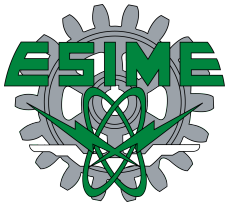
Instituto Politécnico Nacional



Escuela Superior de Ingeniería
Mecánica y Eléctrica

Sección de Estudios de Posgrado e Investigación

Microcontroladores

Proyecto Final

Implementación de un generador síncrono virtual (VSG) con Hardware in the Loop (HIL) en Simulink

Alumno:

Luis Daniel Medina Pérez

Contenido

1. Objetivo

El objetivo de este proyecto es implementar en la tarjeta FRDM-k64f un generador síncrono virtual del tipo “Synchronverter” que interactúe en tiempo real con una simulación de una red eléctrica de prueba en Simulink utilizando comunicación serial.

2. Introducción

Los generadores síncronos virtuales (*VSG*)surgieron como una estrategia de control de convertidores electrónicos de potencia para aplicaciones de generación distribuida e interconexión con redes eléctricas de corriente alterna. Esta técnica de control permite a los convertidores electrónicos imitar ciertas propiedades de las máquinas síncronas que facilitan la sincronización e interconexión con la red, la regulación de potencia activa y frecuencia y de potencia reactiva y el voltaje [1]. La principal ventaja de los *VSG* es la capacidad de proveer una respuesta inercial similar a la de los generadores convencionales que contribuye a la mejora de la estabilidad de la red, a diferencia de otros generadores a base de inversores que no aportan inercia y que cuando existe gran penetración de éstos, pueden poner en riesgo la operación de la red.

Los generadores síncronos virtuales emplean un medio de almacenamiento de energía para absorber o inyectar potencia rápidamente según sea necesario para obtener la respuesta inercial requerida por la red [2]. Para el generador virtual considerado en este trabajo, se considera al inversor conectado a una fuente de corriente directa ideal.

El modelo del VSG considerado se rige por las siguientes ecuaciones, expresadas en p.u.

$$2H\dot{\omega}_r = \frac{P_m - P_e}{\omega_r} - D(\omega_r - \omega_0) \tag{1}$$

$$T_r\dot{P}_m = P_{ref} - P_m - \frac{(\omega_r - \omega_0)}{R} - k_{ip}P_i \tag{2}$$

$$T_f\dot{E}_a = E_{ref} - E_a - k_{pq}(Q_e - Q_{ref}) - k_{iq}Q_i \tag{3}$$

$$\dot{P}_i = \omega_r - \omega_0 \tag{4}$$

$$\dot{Q}_i = Q_e - Q_{ref} \tag{5}$$

El vector de estado seleccionado para el cálculo es el siguiente:

$$\mathbf{x} = [\omega_r, P_m, E_a, P_i, Q_i]^T \tag{6}$$

La red de prueba utilizada en la simulación se compone de un generador síncrono que alimenta en paralelo con el generador síncrono virtual una carga trifásica balanceada. El modelo de la máquina síncrona es el modelo de sexto orden incluido en la librería *Specialized Power Systems* dentro de *Simulink*, que toma en cuenta la dinámica de los devanados de amortiguamiento, del estator y del campo, en adición a la dinámica del rotor [3]. La máquina síncrona modelada es impulsada por una turbina de vapor con recalentamiento y cuenta con un sistema de excitación ST1. Los modelos para estos dos últimos elementos fueron tomados de [4].

En la figura 1 se muestra la red implementada en *Simulink*. El método de integración utilizado es Euler regresivo con un paso de 0.1 mili segundos.

Los datos de las cargas, los generadores y la red se muestran en la tabla 1.

Generador Síncrono						Generador Síncrono Virtual					
S_n	25MVA	V_n	4.16kV	R_a	2e-3pu	S_n	5MVA	V_n	4.16kV	f_n	60Hz
X_d	1.8pu	X'_d	0.3pu	X''_d	0.2pu	X_{vsg}	0.3pu	X_c	22.5pu	R_a	0.05pu
X_q	1.5pu	X''_q	0.2pu	X_l	0.1pu	H	5s	D	20pu	1/R	0pu
T'_{do}	8.0s	T''_{do}	0.03s	T''_{qo}	0.05s	k_{ip}	0.11pu	T_r	0.1s	E_{ref}	1.0pu
rpm	3600	H	3s	D	2pu	k_{pq}	0.5pu	k_{iq}	0.08pu	T_f	0.1s
Regulador de velocidad y turbina						Sat.P_i	2.3s	Sat.Q_i	2.5s		
R	0.08pu	T_g	0.2s	Trh	7.0s	Cargas					
T_{ch}	0.3s	Fhp	0.3			Inicial	15MW	6MVAr	Adicional		2MW
Sistema de excitación						Lineas de conexión					
K_a	200pu	Tr	0.015s			Lin.1	0.7mΩ	9.2μH	Lin.2	6.9mΩ	55μH

Tabla 1: Tabla de datos nominales.

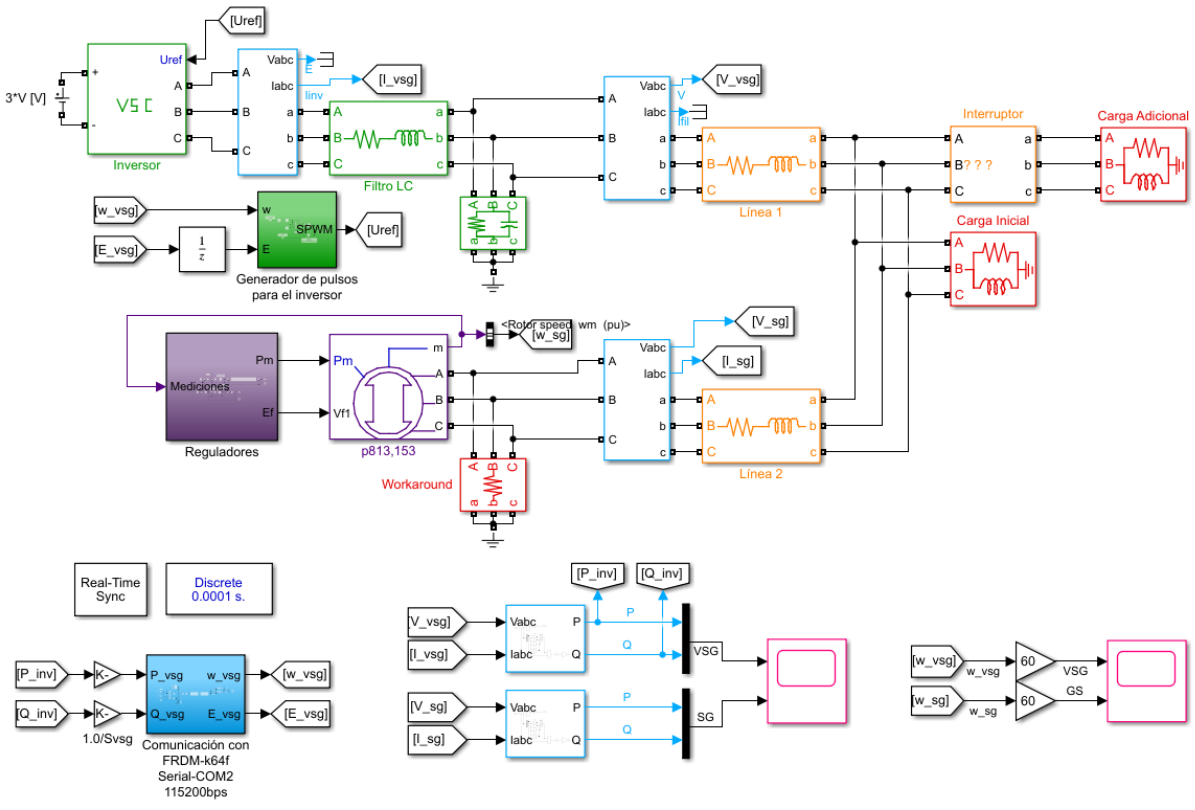


Figura 1: Red implementada en *Simulink*.

3. Diagrama de flujo del programa

El programa creado para el microcontrolador funciona a través de interrupciones generadas por el temporizador de interrupciones periódicas *PIT* cada que se cumple el periodo de muestreo. Al iniciar el programa, se configuran en primer lugar el canal de comunicación y los periféricos para que posteriormente el microcontrolador quede a la espera de un comando de inicio, acompañado de las primeras muestras de las dos señales de entrada: P y Q, que son la potencia activa y reactiva generada por el inversor del *VSG*. Una vez que ha sido reconocido el primer comando, el microcontrolador entra al lazo principal en el cual queda a la espera de la bandera "actualizaModelo" para calcular el estado del *VSG* con los últimos datos recibidos y verifica si la bandera "txEnCurso" está desactivada para preparar el arreglo para enviar las nuevas salidas recién calculadas: la velocidad angular ω y la magnitud de la tensión generada E_A.

Cuanto el temporizador genera la interrupción, se activa la bandera "rxEnCurso" y se llama a la función "Recibe()", la cual genera una solicitud para recibir las nuevas señales de entrada. Cuando se termina la recepción de datos, el *API* del *UART* genera una interrupción que desactiva la bandera "rxEnCurso", activa la bandera "actualizaModelo" y llama a la función "Envia()", que habilita la bandera "txEnCurso" y comienza la transmisión de las últimas señales de control calculadas. Al terminar dicha transmisión, el *API* genera una nueva interrupción que desactiva la bandera "txEnCurso" y dispara el bloque de retardo programado *DPB*.

El *PDB* se encarga de disparar dos canales del convertidor analógico digital *ADC0* en modo "BackToBack" que toman una muestra de dos señales analógicas que representan los comandos de potencia activa y reactiva P_{ref} y Q_{ref} del *VSG*.

Cuando se encuentra activada la bandera "actualizaModelo", se llama a la función "modelo()", que utiliza el método de Runge-Kutta de cuarto orden para calcular el estado siguiente del *VSG* después del paso de integración T_s, que es el periodo de interrupción del *PIT*. Este último se seleccionó de 10 mili segundos para permitir a la computadora que ejecuta la simulación poder establecer correctamente la comunicación. Al concluir el cálculo, la función "modelo()" desactiva la bandera "actualizado", que permite que en el lazo principal se guarden en registro de salida para la comunicación serial los nuevos comandos de control.

En las figuras 2, 3, 4 y 5 se muestra el diagrama de flujo del programa y el correspondiente diagrama de tiempos.

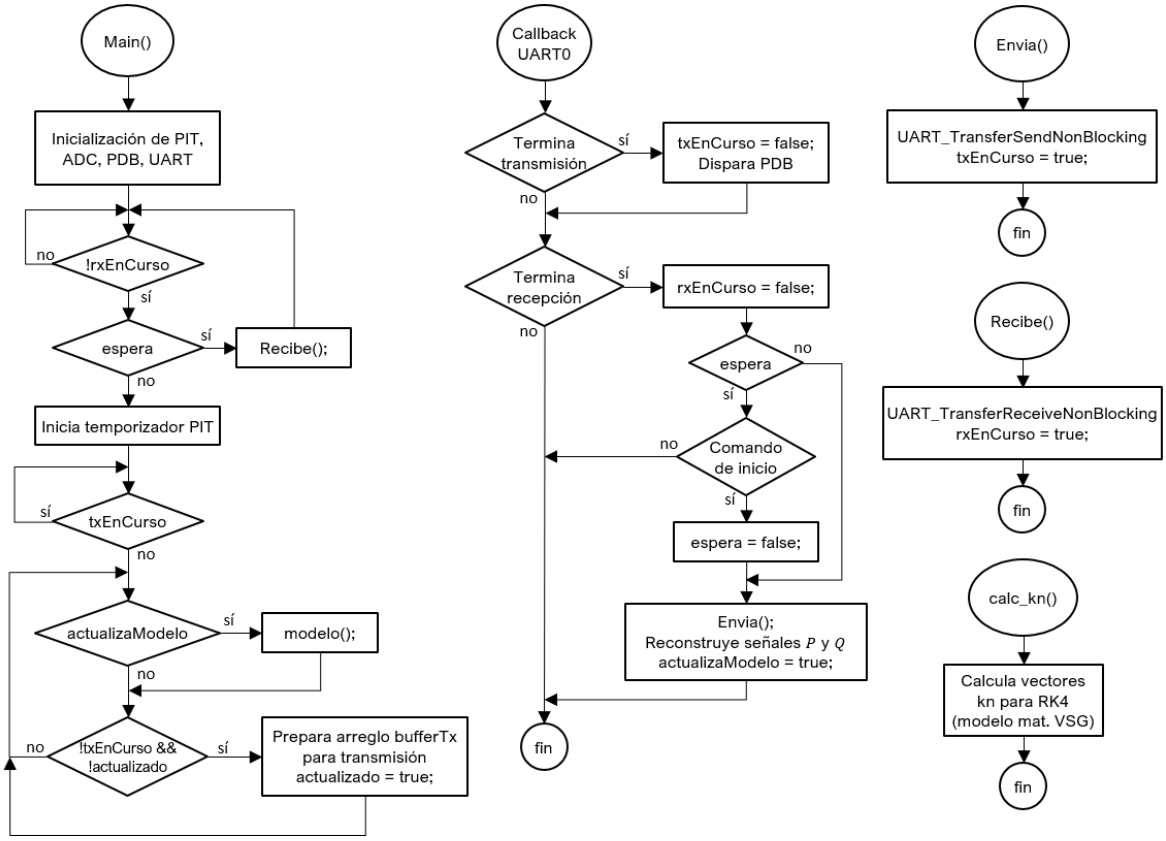


Figura 2: Diagrama de flujo del programa.

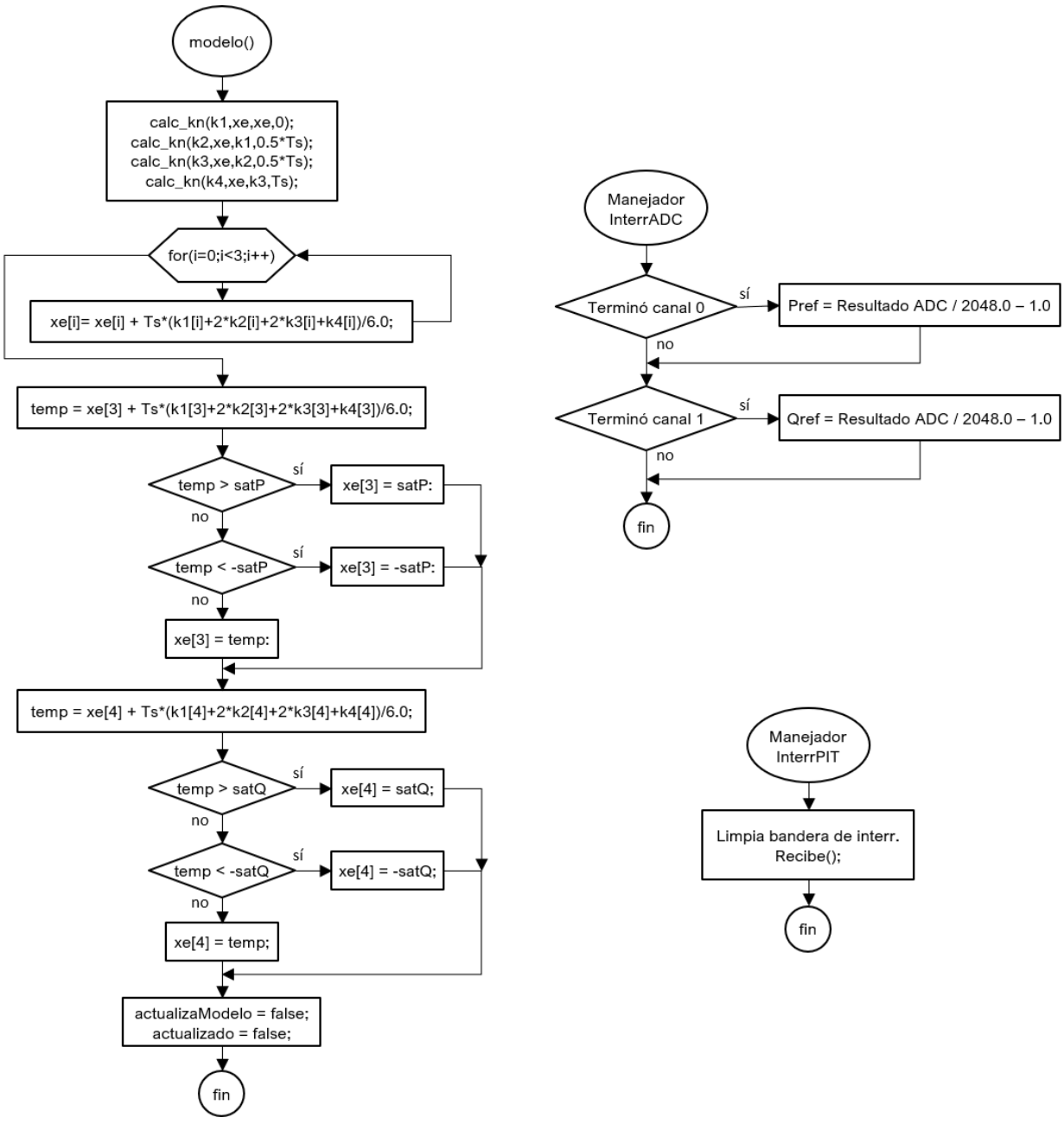


Figura 3: Diagrama de flujo del programa (continuación).

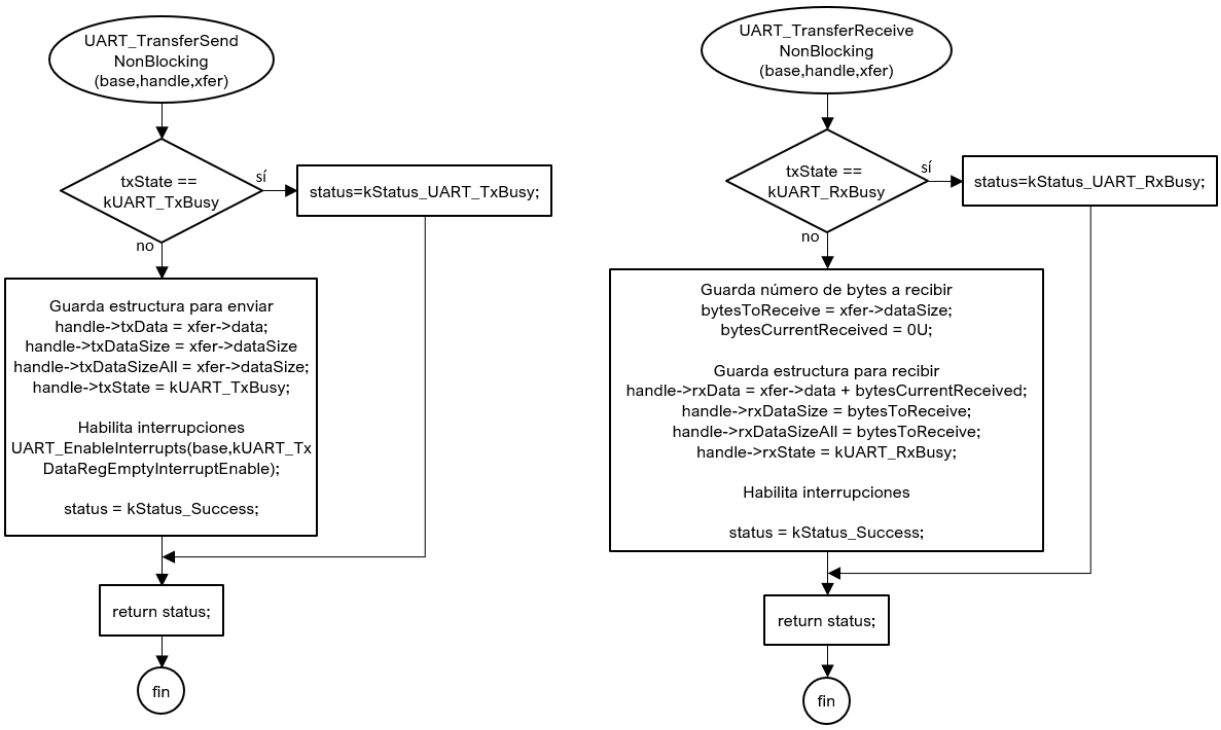


Figura 4: Diagrama de flujo del programa (continuación).

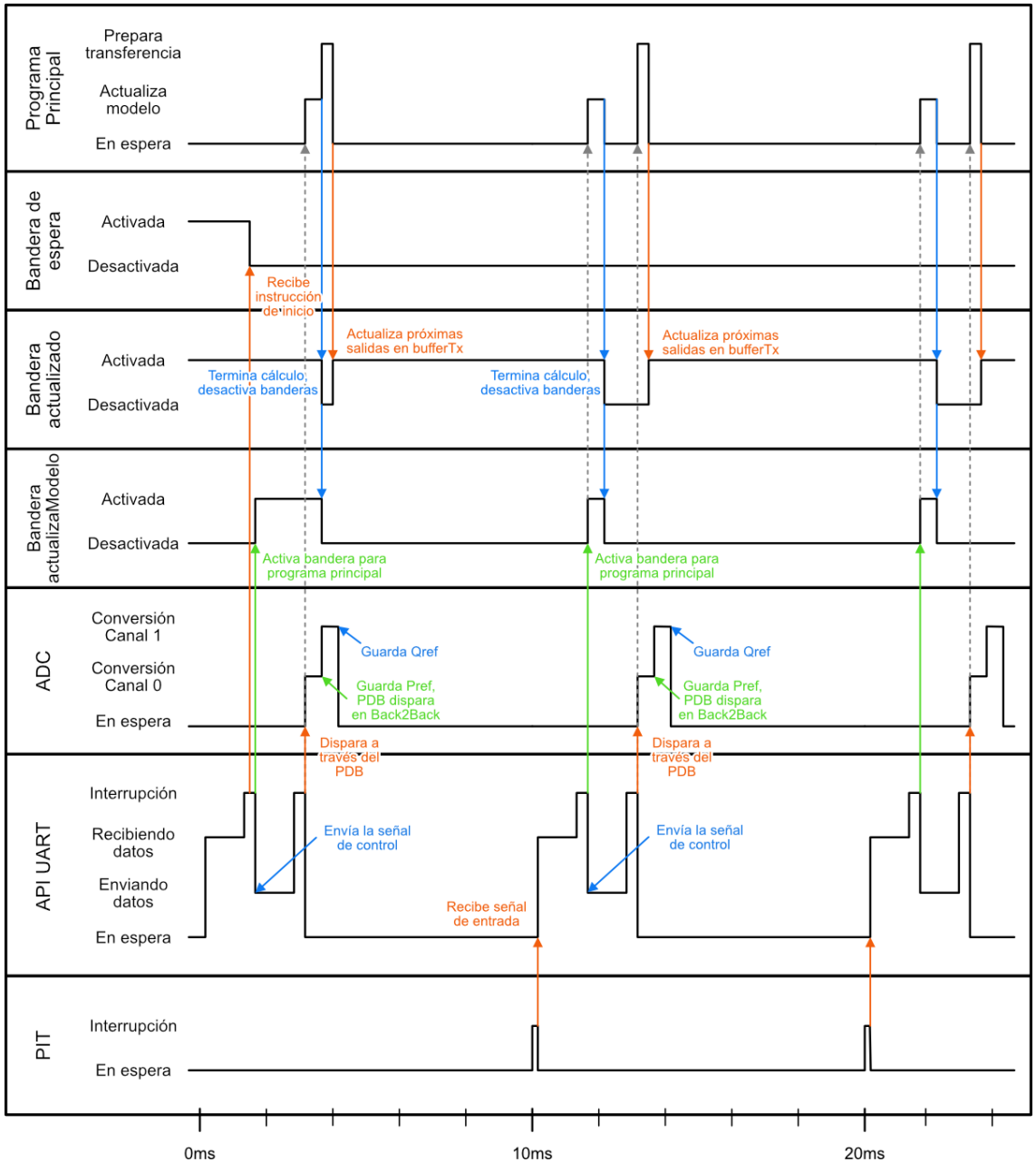


Figura 5: Diagrama de tiempos del programa.

4. Código del programa

4.1. Librerías y definiciones

A continuación se muestran las librerías empleadas en el archivo de código principal del programa, entre ellas se incluyen algunas librerías propias, tres de ellas con definiciones, variables y funciones específicas para los periféricos y una de definiciones globales. Se muestra en el fragmento de código siguiente también las definiciones para el programa principal, que son los parámetros del generador síncrono virtual.

```
#include "board.h"
#include "fsl_uart.h"
#include "pin_mux.h"
#include "clock_config.h"
#include "math.h"

#include "DfnGlb.h"
#include "PITconfig.h"
#include "ADCconfig.h"
#include "PDBconfig.h"

#define FREC_RELOJ_UART CLOCK_GetFreq(UART0_CLK_SRC)

// Parámetros del modelo
#define M_ 10.0 //Cte de inercia2 (2H) [s]
#define D_ 20.0 //Coef. de amortiguamiento [s]
#define kr_ 0.0 //Ganacia de estatismo de frecuencia (1/R) [pu] (Con este modelo D genera estatismo)
#define ki_ 0.11 //Ganancia integral P [pu]
#define Tr_ 0.1 //Cte. de tiempo del regulador de w [s]
#define Tf_ 0.1 //Cte. de tiempo del regulador de E [s]
#define Eref_ 1.0 //Valor de referencia de tensión [pu]
#define kq_ 0.5 //Ganancia proporcional Q [pu]
#define kqi_ 0.08 //Ganancia integral Q [pu]

// Limite (saturación) de integradores
#define satP 2.3
#define satQ 2.5
```

4.2. Variables globales

A continuación se muestran las variables globales declaradas en el programa principal. Las primeras tres son necesarias para la comunicación, mientras que las siguientes cinco, de tipo *bool*, se utilizan como banderas para manejar el flujo del programa. Las últimas cinco variables son de tipo flotante y corresponden a las cuatro señales de entrada y el estado del *VSG*.

```
// Manejador
uart_handle_t manejadorUART0;

// Variables de buffer para transmitir/recibir
uint8_t bufferTx[4] = {0,128,0,64};
uint8_t bufferRx[5] = {0};

// Banderas de estado
```

```
volatile bool txEnCurso = false;
volatile bool rxEnCurso = false;
volatile bool actualizaModelo = false;
volatile bool espera = true;
volatile bool actualizado = true;

// Señales
float P = 0;
float Q = 0;
float Pref = 0;
float Qref = 0.05;
// Estado: [omega,Pm,Ea,Pi,Qi]
float xe[5] = {1.0,0,1.0,0,0};
```

4.3. Función principal

Al iniciar el programa, las primeras instrucciones en ser ejecutadas son para la configuración de los periféricos como se muestra a continuación.

```
BOARD_InitPins();
BOARD_BootClockRUN();
PIT_Config();
ADC_Config();
PDB_Config();
```

4.3.1. Configuración de la comunicación serial

El siguiente paso en la ejecución del programa es la configuración de la configuración serial con el *UART*. Se establece la velocidad de 115200 bits por segundo, se habilitan la transmisión y recepción de datos y a continuación se inicializa el módulo con la estructura generada. Finalmente se crea un manejador para la *API* del *UART* que asocia la instancia de comunicación serial utilizada (UART0) con su correspondiente función de manejador (manejadorUART0).

```
uart_config_t configUART0;

// Configuración estándar del UART
UART_GetDefaultConfig(&configUART0);

// Configura velocidad (115200)
configUART0.baudRate_Bps = 115200U;

// Habilita recepción y transmisión
configUART0.enableTx = true;
configUART0.enableRx = true;

// Inicializa con la estructura de configuración el UART0
UART_Init(UART0, &configUART0, FREC_RELOJ_UART);

// Crea manejador para recibir/enviar datos por paquete
UART_TransferCreateHandle(UART0, &manejadorUART0, Callback_UART0, NULL);
```

4.3.2. Lazo principal

Antes de entrar al lazo principal del programa, el microcontrolador permanece en modo de espera hasta que recibe la instrucción de inicio, posterior a la cual envía las señales de control iniciales y entra al lazo principal al concluir de enviar los datos.

En el lazo principal el programa solo puede ejecutar dos tareas, la primera es actualizar el estado del generador síncrono virtual calculando una iteración de integración con la función "modelo" cada que la bandera "actualizaModelo" se encuentre activada. La segunda función es preparar el buffer de datos para la transmisión de las señales de control. Para esta tarea, el programa convierte las variables de tipo flotante a variables enteras de 16 bits sin signo que separa en bytes. Esta tarea se ejecuta siempre que la bandera "actualizado" se desactive y se haya terminado la última transmisión de datos. Esto último es necesario para evitar que se modifique el buffer de salida mientras los datos que estan ahí contenidos están en proceso de transmisión.

```
1  // Espera a recibir instrucción para comenzar
2  while(1){
3    if(!rxEnCurso){
4      if(espera)
5        Recibe();
6      else
7        break;
8    }
9  }
10
11 // Inicia el timer PIT canal 0
12 PIT_StartTimer(PIT, kPIT_Chnl_0);
13
14 // Al detectar el comando, se llamó a Enviar()
15 while(txEnCurso){} // Espera a que termine de enviar
16
17 while (1){
18   if(actualizaModelo)
19   modelo();
20
21   // Si terminó de enviar los datos anteriores, prepara arreglo para siguiente muestra
22   if(!txEnCurso && !actualizado){
23     // Prepara arreglo para enviar
24     bufferTx[0]=(uint8_t)floor(xe[0]*conv_15a);
25     bufferTx[1]=(uint8_t)floor(xe[0]*conv_15b);
26     bufferTx[2]=(uint8_t)floor(xe[2]*conv_14a);
27     bufferTx[3]=(uint8_t)floor(xe[2]*conv_14b);
28     // Confirma que los últimos datos fueron actualizados
29     actualizado = true;
30   }
31 }
```

4.4. Funciones de comunicación

4.4.1. Función para enviar datos

La función "Envia" genera una solicitud al *API* del *UART* para transmitir un mensaje de cuatro bytes que representa las dos señales de control codificadas en 16 bits. Cuando la función se llama, activa la bandera

txEnCurso para prevenir que se intente mandar otro mensaje cuando el último aún no ha terminado de ser procesado.

Cabe destacar que el uso de la *API* del *UART* permite operar fácilmente el *UART* en modo de interrupciones, de forma que solo es necesario generar las solicitudes para enviar o recibir datos y el *API* enviará o recibirá los mensajes byte a byte en segundo plano.

```
1  void Envia(void){
2    // Estructuras con apuntador y tamaño para enviar
3    uart_transfer_t enviaXfer;
4
5    // Configura las estructuras para enviar datos
6    enviaXfer.data = bufferTx;
7    enviaXfer.dataSize = 4;
8
9    UART_TransferSendNonBlocking(UART0, &manejadorUART0, &enviaXfer);
10   txEnCurso = true;
11 }
```

4.4.2. Función para recibir datos

La función “Recibe” genera una solicitud al *API* del *UART* para recibir un mensaje de cinco bytes que contiene una instrucción de 1 byte y las dos señales de entrada codificadas en 16 bits. Cuando la función se llama, activa la bandera rxEnCurso para prevenir que se intente recibir otro mensaje cuando el último aún no ha terminado de ser procesado.

```
1  void Recibe(void){
2    // Estructuras con apuntador y tamaño para recibir
3    uart_transfer_t recibeXfer;
4
5    // Configura las estructuras para recibir datos
6    recibeXfer.data = bufferRx;
7    recibeXfer.dataSize = 5;
8
9    UART_TransferReceiveNonBlocking(UART0, &manejadorUART0, &recibeXfer, NULL);
10   rxEnCurso = true;
11 }
```

4.4.3. Manejador de interrupciones

El manejador de interrupciones se ejecuta a solicitud del *API* del *UART* cada que se termina de procesar un mensaje. Esta función determina si la interrupción es resultado de enviar un mensaje o de recibirlo y modifica la bandera correspondiente. Cuando se termina de recibir un mensaje (las señales de entrada), el programa genera una solicitud para enviar las señales de control más recientes, posteriormente decodifica las señales recibidas y activa la bandera “actualizaModelo” para que se actualice el estado del *VSG* con las últimas entradas recibidas. Por el otro lado si se llamó a la función al concluir la transmisión de un mensaje, el programa dispara el bloque de retardo *PDB* para que este a su vez dispare los dos canales del convertidor analógico “ADC” en modo “back-to-back” y se obtengan las dos señales de entrada analógicas.

```
1  void Callback_UART0(UART_Type *base, uart_handle_t *handle, status_t status, void *userData){
2    userData = userData;
3
```

```
4    // Si terminó de enviar, cambia bandera
5    if (kStatus_UART_TxIdle == status){
6      txEnCurso = false;
7      // Activa el PDB para que dispare los canales del ADC
8      PDB_DoSoftwareTrigger(PDB0);
9    }
10
11   // Si terminó de recibir, cambia bandera
12   if (kStatus_UART_RxIdle == status){
13     rxEnCurso = false;
14
15     // Si se está esperando el inicio, busca comando
16     if(espera){
17       if(bufferRx[0] == '|'){
18         espera = false;
19       }else
20         return;
21     }
22
23     // Comienza inmediatamente nuevo envío
24     Envia();
25
26     // Reconstruye las señales recibidas a 16bits y luego a flotante
27     P = (float) ( (((uint16_t)bufferRx[2])<<8 | (uint16_t)bufferRx[1])-32768 );
28     P = P/4096.0; // Remueve el offset y corrige la magnitud
29     Q = (float) ( (((uint16_t)bufferRx[4])<<8 | (uint16_t)bufferRx[3])-32768 );
30     Q = Q/4096.0; // Remueve el offset y corrige la magnitud
31
32     // Genera la bandera para hacer el cálculo con los nuevos datos
33     actualizaModelo = true;
34   }
35 }
```

4.5. Funciones para el temporizador*PIT*

El temporizador *PIT* se encarga de generar una interrupción cada que se cumple el periodo de muestreo, es decir, cada que transcurren 10 milisegundos. Dicha interrupción se encarga de solicitar la recepción de las nuevas señales de entrada. Este periférico emplea las definiciones siguientes.

```
1 // Manejador de las interrupciones del canal 0
2 #define ManejadorInterrPIT0 PIT0_IRQHandler
3 // Fuente de reloj
4 #define FuenteRelojPIT CLOCK_GetFreq(kCLOCK_BusClk)
```

4.5.1. Configuración del temporizador

Para el temporizador se utiliza la configuración estándar y se configura el periodo de muestreo a 10 milisegundos. Las interrupciones del temporizador se habilitan pero no se inicia la cuenta del *PIT*, dado que el programa primero esperará a recibir el comando de inicio.

```
1 void PIT_Config(void){
```

```
  pit_config_t pitConfig;
  // Obtiene la estructura predeterminada del PIT
  PIT_GetDefaultConfig(&pitConfig);
  // Inicializa el PIT con la estructura predet.
  PIT_Init(PIT, &pitConfig);

  // Configura el periodo (0.1s) [Canal 0]
  PIT_SetTimerPeriod(PIT, kPIT_Chnl_0, MSEC_TO_COUNT(1000*Ts, FuenteRelojPIT));
  // Habilita solicitud de interrupciones del canal 0
  PIT_EnableInterrupts(PIT, kPIT_Chnl_0, kPIT_TimerInterruptEnable);
  // Habilita interrupciones del canal 0
  EnableIRQ(PIT0_IRQn);
}
```

4.5.2. Manejador de interrupciones

El manejador de interrupciones del temporizador reestablece la bandera de interrupción y genera la solicitud para recibir las señales de entrada a través de la comunicación serial.

```
void ManejadorInterrPIT0(void){
  // Reestablece la bandera
  PIT_ClearStatusFlags(PIT, kPIT_Chnl_0, kPIT_TimerFlag);
  // Comienza recepción de datos de la simulación
  Recibe();
  //Correción ensamblador, errata 838869
  __DSB();
}
```

4.6. Funciones para el convertidor*ADC*

El convertidor analógico digital muestrea dos señales de entrada analógicas que corresponden a los comandos de potencia activa y potencia reactiva del generador síncrono virtual. Se utiliza únicamente el convertidor *ADC0* y dos canales asociados con este, cuyas configuraciones se cargan en los dos grupos de configuración que posee. A continuación se muestran las definiciones pertinentes.

```
// Canal del ADC (A)
#define CanalADC_G0 12U
// Canal del ADC (B)
#define CanalADC_G1 13U
// Grupo 0 del ADC
#define Grupo0ADC 0U
// Grupo 1 del ADC
#define Grupo1ADC 1U
// Manejador de las interrupciones del ADC
#define ManejadorInterrADC ADC0_IRQHandler
```

4.6.1. Configuración del convertidor

El convertidor *ADC* se configura con los parámetros predeterminados, con resolución de 12 bits. Se programa la calibración automática del periférico y posteriormente se ajustan los dos grupos de configuración del

ADC0, cada uno asociado a un canal propio. Se habilitan las interrupciones del convertidor.

```
1  void ADC_Config(void)
2  {
3    adc16_config_t adcConfigStruct;
4    adc16_channel_config_t adcCanalConfigStruct;
5    // Obtiene la configuración estándar
6    ADC16_GetDefaultConfig(&adcConfigStruct);
7    // Ajusta resolución a 12 bits (redundante)
8    adcConfigStruct.resolution = kADC16_ResolutionSE12Bit;
9    // Carga la configuración de la estructura al ADC0
10   ADC16_Init(ADC0, &adcConfigStruct);
11
12   // Calibración automática del ADC
13   ADC16_EnableHardwareTrigger(ADC0, false);
14   ADC16_DoAutoCalibration(ADC0);
15
16   // Habilita trigger por hardware
17   ADC16_EnableHardwareTrigger(ADC0, true);
18
19   // Configuración del grupo 0 del ADC
20   // Número de canal
21   adcCanalConfigStruct.channelNumber = CanalADC_G0;
22   // Habilita la interrupción al completar conversión
23   adcCanalConfigStruct.enableInterruptOnConversionCompleted = true;
24   // Deshabilita la conversión diferencial (queda en modo single end)
25   adcCanalConfigStruct.enableDifferentialConversion = false;
26   // Carga la configuración del canal
27   // Queda asociada al grupo 0
28   ADC16_SetChannelConfig(ADC0, Grupo0ADC, &adcCanalConfigStruct);
29
30   // Configuración del grupo 1 del ADC
31   // Número de canal
32   adcCanalConfigStruct.channelNumber = CanalADC_G1;
33   // Habilita la interrupción al completar conversión
34   adcCanalConfigStruct.enableInterruptOnConversionCompleted = true;
35   // Deshabilita la conversión diferencial (queda en modo single end)
36   adcCanalConfigStruct.enableDifferentialConversion = false;
37   // Carga la configuración del canal
38   // Queda asociada al grupo 1
39   ADC16_SetChannelConfig(ADC0, Grupo1ADC, &adcCanalConfigStruct);
40
41   // Habilita las interrupciones del ADC
42   EnableIRQ(ADC0_IRQn);
43 }
```

4.6.2. Manejador de interrupciones

El manejador de interrupciones se encarga de leer el resultado de la conversión y guardarlo en la variable global de la señal de entrada correspondiente, antes corrigiendo su magnitud y compensando el offset. Con esta configuración, al variar las señales analógicas de entrada de 0V a 3.3V, se interpreta como señal de entrada de -1.0 pu a +1.0 pu.

```
1  void ManejadorInterrADC(void){
```

```
  // Evalua si la interrupción es del grupo 0
  if (ADC16_GetChannelStatusFlags(ADC0, Grupo0ADC))
  Pref = ADC16_GetChannelConversionValue(ADC0, Grupo0ADC)/2048.0 - 1.0; //Potencia activa -1.0 a 1.0 pu
  // Evalua si la interrupción es del grupo 1
  if (ADC16_GetChannelStatusFlags(ADC0, Grupo1ADC))
  Qref = ADC16_GetChannelConversionValue(ADC0, Grupo1ADC)/2048.0 - 1.0; //Potencia reactiva -1.0 a 1.0 pu
  //Correción ensamblador, errata 838869
  __DSB();
}
```

4.7. Funciones para el bloque de retardo *PDB*

El bloque de retardo *PDB* se emplea únicamente como una interfaz para disparar el convertidor *ADC* por hardware. La razón de ejecutar los disparos de esta manera en lugar de utilizar disparos por software es que con disparos por hardware se tiene la capacidad para utilizar dos grupos de configuración y hacer disparos en modo *back-to-back*, lo que permite disparar consecutivamente los canales del *ADC* inmediatamente al concluir la conversión sin intervención del procesador principal. A continuación se muestran las definiciones para el bloque de retardo.

```
// Canal del PDB para el disparo del ADC0
#define CanalPDB_ADC0Trig kPDB_ADCTriggerChannel0
// Canal del PDB para el pre-disparo del Grupo 0 del ADC0
#define CanalPDB_ADC0PreTrig0 kPDB_ADCPreTrigger0
// Canal del PDB para el pre-disparo del Grupo 1 del ADC0
#define CanalPDB_ADC0PreTrig1 kPDB_ADCPreTrigger1
```

4.7.1. Configuración

El *PDB* se configura con los parámetros estándar, se ajusta el módulo y se configuran los pre-disparos para el convertidor *ADC*. En este caso, se configura para que el *PDB* dispare el grupo 0, asociado con el primer canal del *ADC* y justo al terminar la conversión, dispare el grupo 1 del convertidor.

```
void PDB_Config(void)
{
  // Estructura para configuración del pre-disparo del ADC
  pdb_adc_pretrigger_config_t pdbAdcPreTriggerConfigStruct;
  // Estructura para el PDB
  pdb_config_t pdbConfigStruct;

  // Habilita interrupciones del PDB
  //PDB_EnableInterrupts(PDB0, kPDB_DelayInterruptEnable);

  // Carga la configuración estándar del PDB
  PDB_GetDefaultConfig(&pdbConfigStruct);
  // Carga la configuración al PDB
  PDB_Init(PDB0, &pdbConfigStruct);
  // Carga el módulo
  PDB_SetModulusValue(PDB0, 1000);

```

```
18   // Configura el pre−disparo para el ADC (Grupo 0 y 1)
19   // Operación BackToBack habilitada para el grupo 1
20   pdbAdcPreTriggerConfigStruct.enableBackToBackOperationMask = 1 << 1U;
21   // Habilitar los pre−disparos
22   pdbAdcPreTriggerConfigStruct.enablePreTriggerMask = (1U<<CanalPDB_ADC0PreTrig0)|(1U<<
         CanalPDB_ADC0PreTrig1);
23   pdbAdcPreTriggerConfigStruct.enableOutputMask = (1U<<CanalPDB_ADC0PreTrig0)|(1U<<
         CanalPDB_ADC0PreTrig1);
24   // Carga la configuración del pre−disparo
25   PDB_SetADCPreTriggerConfig(PDB0, CanalPDB_ADC0Trig, &pdbAdcPreTriggerConfigStruct);
26   // Carga la configuración del retraso del pre−disparo
27   // (Retraso número de cuentas, debe ser menor al módulo)
28   PDB_SetADCPreTriggerDelayValue(PDB0, CanalPDB_ADC0Trig, CanalPDB_ADC0PreTrig0, 500U);
29
30   PDB_DoLoadValues(PDB0);
31 }
```

4.8. Funciones de cálculo

4.8.1. Actualización del modelo

Con cada nueva muestra de las señales de entrada, el microcontrolador actualiza el modelo del generador síncrono virtual utilizando el método de Runge Kutta de cuarto orden con un paso de integración igual al periodo de muestreo. El método numérico fue elegido buscando balance entre precisión y simplicidad en el cálculo.

La función “modelo” ejecuta la secuencia del método numérico de integración empleando la función “calc_kn” para calcular los vectores k_1, k_2, k_3 y k_4 a partir del estado actual “xe” y el vector k_i previo. Las variables de estado se actualizan con los valores obtenidos del cálculo de los vectores k_i y en el caso de las variables P_i y Q_i, se aplican los límites de saturación correspondientes. Esto último se implementa con el objetivo de evitar que la aportación integral de los controladores de potencia incremente excesivamente en el caso de que se presente un error de potencia grande sostenido durante un tiempo considerablemente largo, pues esto podría tener un efecto negativo en la operación del *VSG*.

```
1  void modelo(void){
2    // Estado: delta, omega, Pm, Ea
3    float k1[5],k2[5],k3[5],k4[5],temp;
4    int i;
5    calc_kn(k1,xe,xe,0);
6    calc_kn(k2,xe,k1,0.5*Ts);
7    calc_kn(k3,xe,k2,0.5*Ts);
8    calc_kn(k4,xe,k3,Ts);
9    for(i=0;i<3;i++)
10     xe[i]= xe[i] + Ts*(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6.0;
11   // Integradores saturables
12   // Potencia activa
13   temp = xe[3] + Ts*(k1[3]+2*k2[3]+2*k3[3]+k4[3])/6.0;
14   if(temp > satP)
15     xe[3] = satP;
16   else if(temp < −satP)
17     xe[3] = −satP;
18   else
```

```
19    xe[3] = temp;
20    // Potencia reactiva
21    temp = xe[4] + Ts*(k1[4]+2*k2[4]+2*k3[4]+k4[4])/6.0;
22    if(temp > satQ)
23      xe[4] = satQ;
24    else if(temp < -satQ)
25      xe[4] = -satQ;
26    else
27      xe[4] = temp;
28    // Restablece la bandera
29    actualizaModelo = false;
30    // Indica que hay nuevos datos para actualizar
31    actualizado = false;
32  }
```

4.8.2. Vectores de predicción

Esta función calcula los vectores k_i para el método de Runge Kutta de cuarto orden.

```
1   void calc_kn(float kn[5],float xe[5],float kna[5],float n){
2     // Velocidad angular dw = ( ( Pm -Pe )/w -D(w-1.0) )/2H
3     kn[0] = ( ( xe[1]+n*kna[1] -P )/(xe[0]+n*kna[0]) -D_*(xe[0]+n*kna[0]-1.0) )/M_;
4     // Potencia mecánica dPm = ( Pref -Pm -(1/R)(w-1.0) -kiintegral[P-Pref] )/Tr
5     kn[1] = ( Pref -xe[1]-n*kna[1] /-kr_(xe[0]+nkna[0]-1.0)/ -ki_*(xe[3]+n*kna[3]))/Tr_;
6     // Tensión generada dEa = ( Eref -Ea -kq(Q-Qref) -kqiintegral(Q-Qref) )/Tf
7     kn[2] = ( Eref_ -xe[2]-n*kna[2] -kq_*(Q-Qref) -kqi_*(xe[4]+n*kna[4]) )/Tf_;
8     // Integral del error de P
9     kn[3] = P - Pref;
10    // Integral del error de Q
11    kn[4] = Q - Qref;
12  }
```

5. Simulación con HIL

5.1. Respuesta a incremento de carga

Para poner a prueba la respuesta inercial del *VSG*, se simuló la respuesta de la red a un incremento súbito de la carga adicional cuando se encontraba operando con la carga inicial de la tabla 1. Con el incremento de carga, se genera un desbalance de potencia, la demanda es mayor a la potencia mecánica entregada por la turbina del generador síncrono convencional y temporalmente el déficit de potencia es compensado por la energía cinética del rotor del generador, esta compensación temporal es la respuesta inercial de la máquina.

Al tomar energía del rotor en movimiento del generador, la frecuencia comienza a caer, por lo que el regulador de velocidad de la máquina debe mandar la instrucción a la turbina de vapor para que genere mayor potencia. Como la respuesta de la turbina es lenta, se puede presentar una caída notable de frecuencia, por lo que se requiere de un elemento de acción rápida que pueda compensar el déficit de potencia temporalmente y dé suficiente tiempo a la turbina para aumentar la potencia que entrega evitando una caída de frecuencia mayor. Esta es la tarea del generador síncrono virtual.

La figura 6 muestra la respuesta de la frecuencia del generador síncrono y el *VSG* cuando se presenta el incremento súbitos de carga. La potencia de ambos generadores durante este evento se muestra en la figura 7. Se observa que, cuando se conecta la carga adicional, la potencia entregada por el *VSG* incrementa rápidamente y después de unos segundos regresa a su valor inicial. Esta potencia adicional momentánea tiene el efecto de reducir la caída de frecuencia.

Para ver de forma explícita la diferencia que hace tener un generador síncrono virtual en la red, se simuló el mismo escenario retirando el *VSG*, es decir, en la simulación solamente el generador síncrono convencional alimenta la carga. La figura 8 muestra la comparación de la respuesta de la frecuencia del generador síncrono con y sin el apoyo del *VSG*. Es notorio que el generdor virtual reduce de forma importante la rapidez con la que cae la frecuencia y el valor mínimo que alcanza durante el evento. Es interesante además observar que este efecto se logró con un VSG que empleó menos de la mitad de su capacidad nominal, que es de 5 MW, para asistir a una máquina de cinco veces su tamaño (25 MW), lo cual indica que no es necesario utilizar un *VSG* de gran capacidad para asistir con respuesta inercial a una máquina de mucho mayor tamaño.

Es importante aclarar que la ligera diferencia en el valor en régimen permanente de la frecuencia del generador síncrono en el escenario sin el *VSG* es mayor debido a que el incremento de la carga generó una caída de tensión mayor que en el escenario anterior, reduciendo el consumo de la carga.

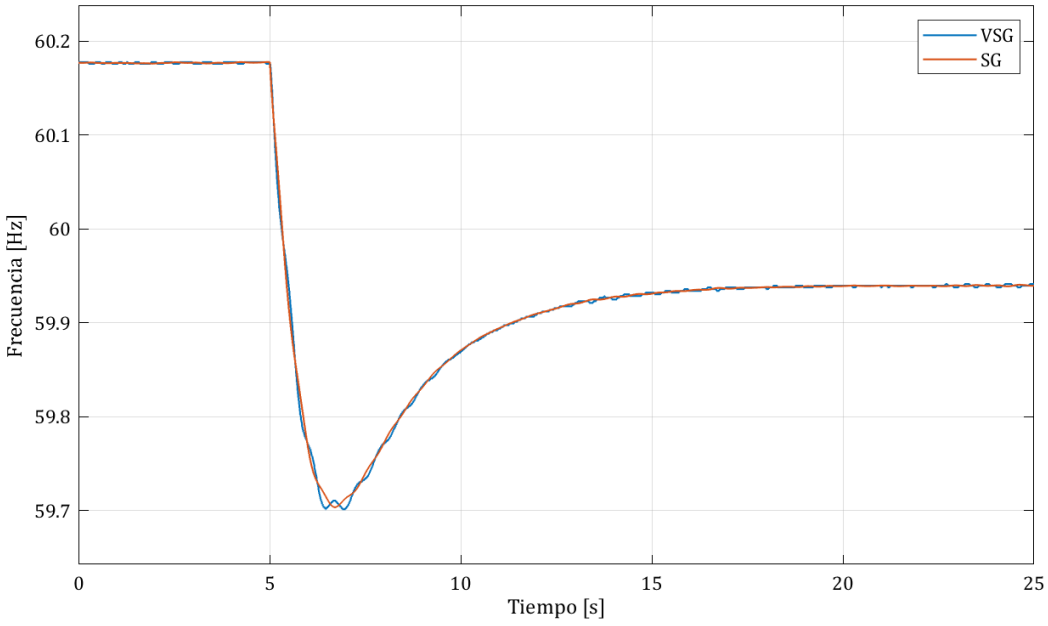


Figura 6: Respuesta de la frecuencia de la red a un incremento súbito de carga.

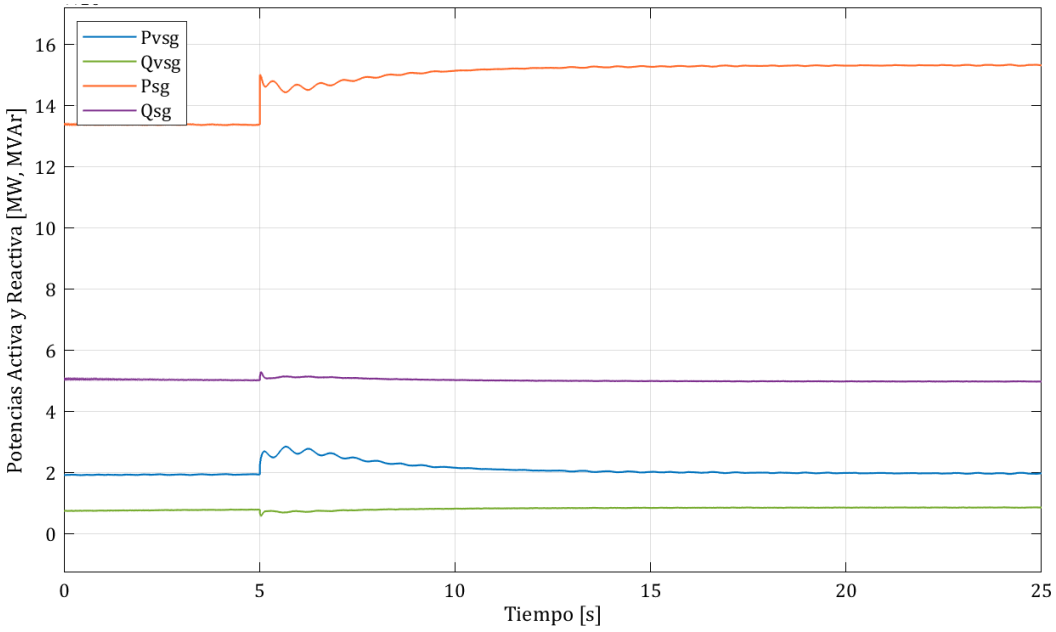


Figura 7: Respuesta de potencia de los generadores a un incremento súbito de carga.

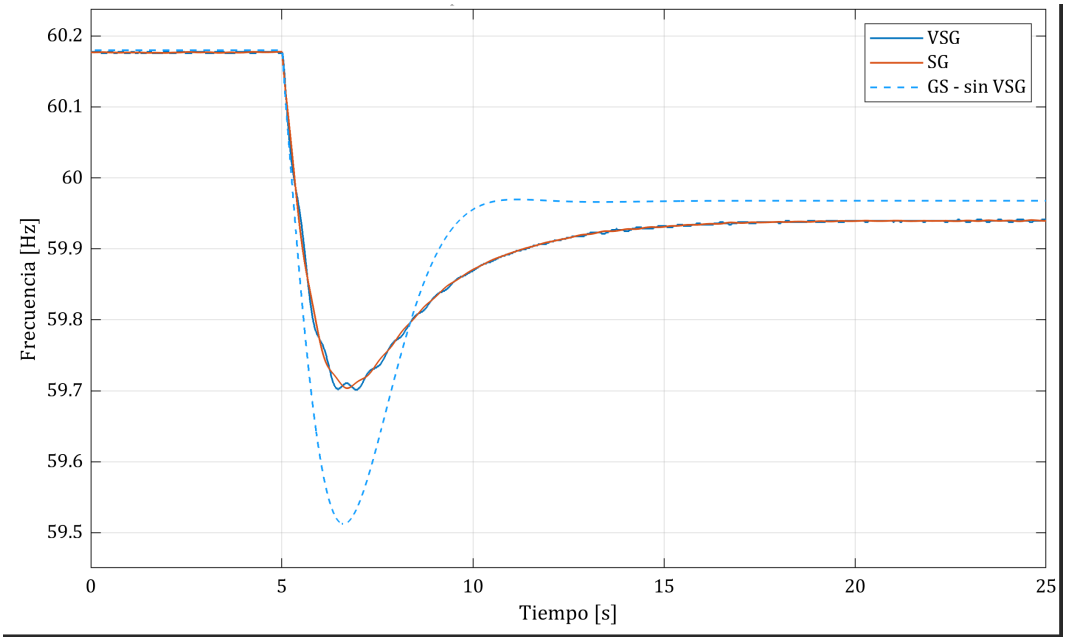


Figura 8: Comparación de la respuesta de la frecuencia de la red a un incremento súbito de carga (con y sin *VSG*).

5.2. Respuesta a entradas manuales

Para probar la respuesta del *VSG* a entradas manuales de comandos de potencia activa y reactiva, en la simulación se probaron dos casos. En el primero, el *VSG* inicialmente aportaba menos del 10 % de la potencia nominal activa y a continuación se cambió la referencia de potencia activa a cerca del 100 % y permaneció así durante 12 segundos. Posteriormente, se cambió la referencia de potencia activa a negativo 60 % de la capacidad nominal. La figura 9 ilustra el comportamiento del generador virtual siguiendo la referencia.

El segundo caso, el generador virtual comenzó entregando 15 % de la potencia nominal con un factor de potencia unitario y a continuación se aplicaron dos incrementos consecutivos a la referencia de potencia reactiva. Al cabo de unos segundos, se cambió la referencia a -90 % de la capacidad nominal en cinco pasos. La figura 10 muestra la respuesta del *VSG* siguiendo la referencia de potencia.

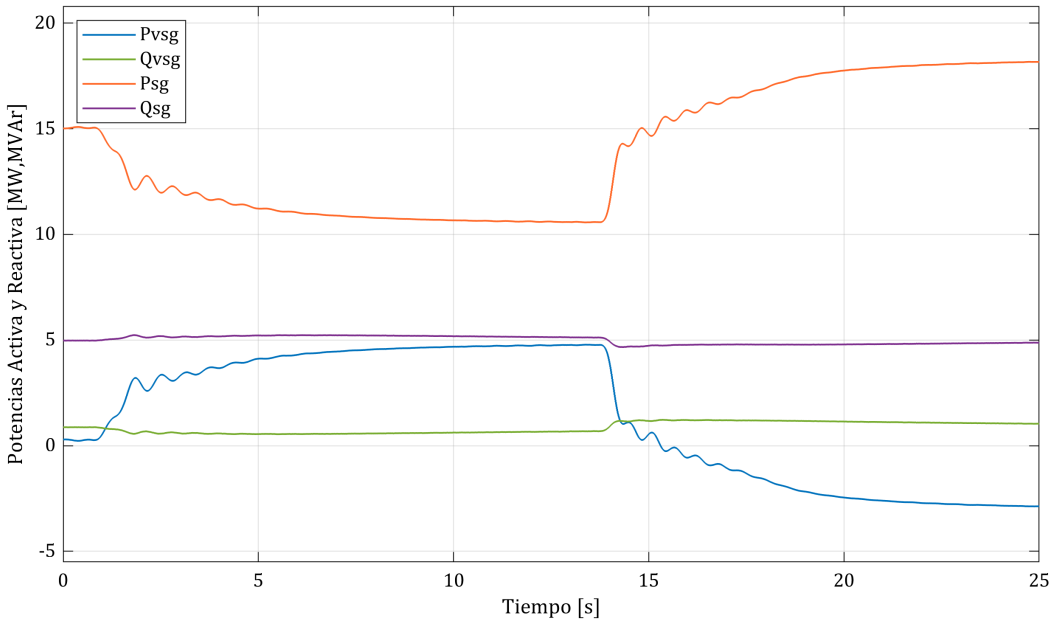


Figura 9: Respuesta del *VSG* a comando de potencia activa.

6. Conclusión

En este trabajo se implementó el control de un generador síncrono virtual en una tarjeta FRMD-k64f con la capacidad de interactuar con una simulación en tiempo real a través de comunicación serial. Se cumplió satisfactoriamente el objetivo establecido, el controlador se comunicó correctamente con la simulación de la red de prueba en *Simulink* y fue posible observar las propiedades del generador síncrono virtual y ejemplificar su efecto benéfico en un sistema eléctrico.

El principal reto de este trabajo no fue el modelo del generador virtual, pues ya se contaba con trabajo bastante avanzado sobre el tema, las comunicaciones y la sincronización con la simulación en tiempo real fueron el aspecto que fue notoriamente más difícil de solucionar, pues para cumplir este requisito fue necesario llevar a cabo un gran número de pruebas para determinar la frecuencia de muestreo que permitiera evaluar correctamente el modelo matemático en el microcontrolador, garantizar que las señales de control fueran enviadas en los tiempos correctos y dar suficiente tiempo a la computadora para ejecutar la simulación en tiempo real.

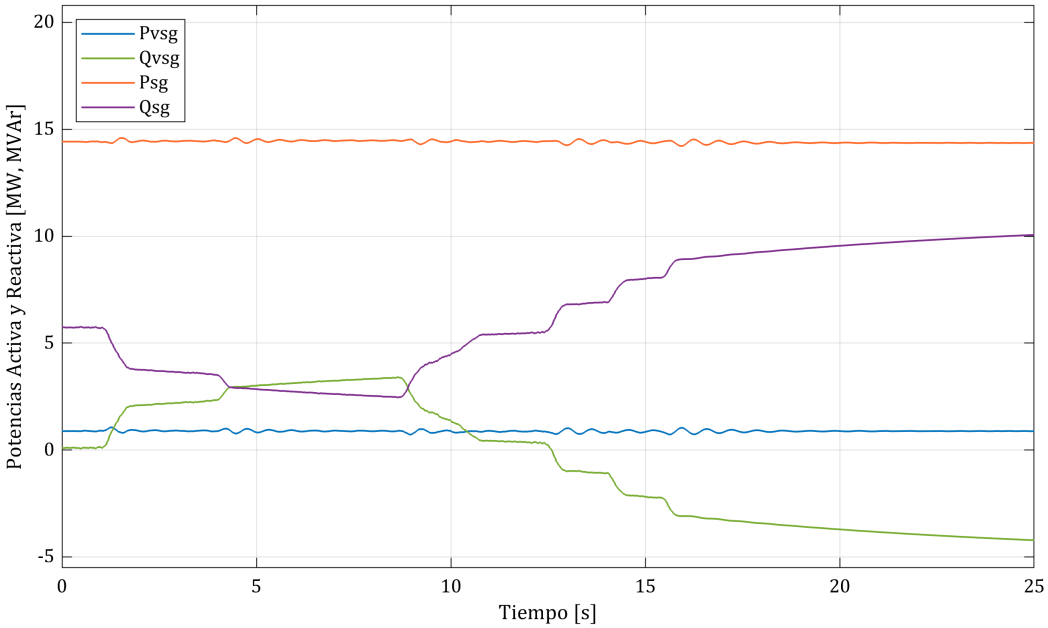


Figura 10: Respuesta del *VSG* a comando de potencia reactiva.

El principal limitante de este trabajo fue la capacidad de procesamiento de la computadora para la simulación. Se utilizó un equipo con procesador core i5 de cuarta generación, 12 GB de RAM y sistema operativo Windows 10 ejecutando Matlab 2020a. El máximo paso de integración satisfactorio fue de 0.0001 segundos con una frecuencia de muestreo de 100Hz. Incrementar la frecuencia de muestreo o reducir el paso de integración ocasionaba que la computadora terminara la simulación después de un muy corto tiempo, reportando que no había sido capaz de mantener la simulación en tiempo real sin retrasos.

Se recomienda para futuros trabajos similares, indagar sobre la posibilidad de llevar a cabo la comunicación a través del software FreeMaster. Asimismo se recomienda utilizar equipos de cómputo de mayor capacidad para reducir el paso de integración e incrementar la frecuencia de muestreo.