/\*

\* GSM\_TX.c

\*

\* Code generation for model "GSM\_TX".

\*

\* Model version : 18.0

\* Simulink Coder version : 9.8 (R2022b) 11-Dec-2022

\* C source code generated on : Thu Dec 22 14:44:27 2022

\*

\* Target selection: grt.tlc

\* Note: GRT includes extra infrastructure and instrumentation for prototyping

\* Embedded hardware selection: Intel->x86-64 (Windows64)

\* Code generation objectives: Unspecified

\* Validation result: Not run

\*/

#include "GSM\_TX.h"

#include "rtwtypes.h"

#include <string.h>

#include <math.h>

#include <emmintrin.h>

#include "GSM\_TX\_private.h"

#include "rt\_nonfinite.h"

#include "rt\_defines.h"

/\* Block signals (default storage) \*/

B\_GSM\_TX\_T GSM\_TX\_B;

/\* Block states (default storage) \*/

DW\_GSM\_TX\_T GSM\_TX\_DW;

/\* Real-time model \*/

static RT\_MODEL\_GSM\_TX\_T GSM\_TX\_M\_;

RT\_MODEL\_GSM\_TX\_T \*const GSM\_TX\_M = &GSM\_TX\_M\_;

/\*

\* Time delay interpolation routine

\*

\* The linear interpolation is performed using the formula:

\*

\* (t2 - tMinusDelay) (tMinusDelay - t1)

\* u(t) = ----------------- \* u1 + ------------------- \* u2

\* (t2 - t1) (t2 - t1)

\*/

real\_T rt\_TDelayInterpolate(

real\_T tMinusDelay, /\* tMinusDelay = currentSimTime - delay \*/

real\_T tStart,

real\_T \*uBuf,

int\_T bufSz,

int\_T \*lastIdx,

int\_T oldestIdx,

int\_T newIdx,

real\_T initOutput,

boolean\_T discrete,

boolean\_T minorStepAndTAtLastMajorOutput)

{

int\_T i;

real\_T yout, t1, t2, u1, u2;

real\_T\* tBuf = uBuf + bufSz;

/\*

\* If there is only one data point in the buffer, this data point must be

\* the t= 0 and tMinusDelay > t0, it ask for something unknown. The best

\* guess if initial output as well

\*/

if ((newIdx == 0) && (oldestIdx ==0 ) && (tMinusDelay > tStart))

return initOutput;

/\*

\* If tMinusDelay is less than zero, should output initial value

\*/

if (tMinusDelay <= tStart)

return initOutput;

/\* For fixed buffer extrapolation:

\* if tMinusDelay is small than the time at oldestIdx, if discrete, output

\* tailptr value, else use tailptr and tailptr+1 value to extrapolate

\* It is also for fixed buffer. Note: The same condition can happen for transport delay block where

\* use tStart and and t[tail] other than using t[tail] and t[tail+1].

\* See below

\*/

if ((tMinusDelay <= tBuf[oldestIdx] ) ) {

if (discrete) {

return(uBuf[oldestIdx]);

} else {

int\_T tempIdx= oldestIdx + 1;

if (oldestIdx == bufSz-1)

tempIdx = 0;

t1= tBuf[oldestIdx];

t2= tBuf[tempIdx];

u1= uBuf[oldestIdx];

u2= uBuf[tempIdx];

if (t2 == t1) {

if (tMinusDelay >= t2) {

yout = u2;

} else {

yout = u1;

}

} else {

real\_T f1 = (t2-tMinusDelay) / (t2-t1);

real\_T f2 = 1.0 - f1;

/\*

\* Use Lagrange's interpolation formula. Exact outputs at t1, t2.

\*/

yout = f1\*u1 + f2\*u2;

}

return yout;

}

}

/\*

\* When block does not have direct feedthrough, we use the table of

\* values to extrapolate off the end of the table for delays that are less

\* than 0 (less then step size). This is not completely accurate. The

\* chain of events is as follows for a given time t. Major output - look

\* in table. Update - add entry to table. Now, if we call the output at

\* time t again, there is a new entry in the table. For very small delays,

\* this means that we will have a different answer from the previous call

\* to the output fcn at the same time t. The following code prevents this

\* from happening.

\*/

if (minorStepAndTAtLastMajorOutput) {

/\* pretend that the new entry has not been added to table \*/

if (newIdx != 0) {

if (\*lastIdx == newIdx) {

(\*lastIdx)--;

}

newIdx--;

} else {

if (\*lastIdx == newIdx) {

\*lastIdx = bufSz-1;

}

newIdx = bufSz - 1;

}

}

i = \*lastIdx;

if (tBuf[i] < tMinusDelay) {

/\* Look forward starting at last index \*/

while (tBuf[i] < tMinusDelay) {

/\* May occur if the delay is less than step-size - extrapolate \*/

if (i == newIdx)

break;

i = ( i < (bufSz-1) ) ? (i+1) : 0;/\* move through buffer \*/

}

} else {

/\*

\* Look backwards starting at last index which can happen when the

\* delay time increases.

\*/

while (tBuf[i] >= tMinusDelay) {

/\*

\* Due to the entry condition at top of function, we

\* should never hit the end.

\*/

i = (i > 0) ? i-1 : (bufSz-1); /\* move through buffer \*/

}

i = ( i < (bufSz-1) ) ? (i+1) : 0;

}

\*lastIdx = i;

if (discrete) {

/\*

\* tempEps = 128 \* eps;

\* localEps = max(tempEps, tempEps\*fabs(tBuf[i]))/2;

\*/

double tempEps = (DBL\_EPSILON) \* 128.0;

double localEps = tempEps \* fabs(tBuf[i]);

if (tempEps > localEps) {

localEps = tempEps;

}

localEps = localEps / 2.0;

if (tMinusDelay >= (tBuf[i] - localEps)) {

yout = uBuf[i];

} else {

if (i == 0) {

yout = uBuf[bufSz-1];

} else {

yout = uBuf[i-1];

}

}

} else {

if (i == 0) {

t1 = tBuf[bufSz-1];

u1 = uBuf[bufSz-1];

} else {

t1 = tBuf[i-1];

u1 = uBuf[i-1];

}

t2 = tBuf[i];

u2 = uBuf[i];

if (t2 == t1) {

if (tMinusDelay >= t2) {

yout = u2;

} else {

yout = u1;

}

} else {

real\_T f1 = (t2-tMinusDelay) / (t2-t1);

real\_T f2 = 1.0 - f1;

/\*

\* Use Lagrange's interpolation formula. Exact outputs at t1, t2.

\*/

yout = f1\*u1 + f2\*u2;

}

}

return(yout);

}

real\_T rt\_atan2d\_snf(real\_T u0, real\_T u1)

{

real\_T y;

int32\_T tmp;

int32\_T tmp\_0;

if (rtIsNaN(u0) || rtIsNaN(u1)) {

y = (rtNaN);

} else if (rtIsInf(u0) && rtIsInf(u1)) {

if (u1 > 0.0) {

tmp = 1;

} else {

tmp = -1;

}

if (u0 > 0.0) {

tmp\_0 = 1;

} else {

tmp\_0 = -1;

}

y = atan2(tmp\_0, tmp);

} else if (u1 == 0.0) {

if (u0 > 0.0) {

y = RT\_PI / 2.0;

} else if (u0 < 0.0) {

y = -(RT\_PI / 2.0);

} else {

y = 0.0;

}

} else {

y = atan2(u0, u1);

}

return y;

}

/\* Model step function \*/

void GSM\_TX\_step(void)

{

{

\_\_m128d tmp;

\_\_m128d tmp\_0;

real\_T acc;

real\_T dspPi;

real\_T uPrev;

int32\_T zero;

/\* S-Function (sdspnsamp2): '<S9>/N-Sample Enable' \*/

/\* DSP System Toolbox N-Sample Enable (sdspnsamp2) - '<S9>/N-Sample Enable' \*/

{

{

if (GSM\_TX\_DW.NSampleEnable\_Counter == GSM\_TX\_P.FMDemodulator100MHz\_Ho)

{

GSM\_TX\_B.NSampleEnable = (boolean\_T)(2 -

GSM\_TX\_P.NSampleEnable\_ActiveLevel);

} else {

GSM\_TX\_B.NSampleEnable = (boolean\_T)

(GSM\_TX\_P.NSampleEnable\_ActiveLevel - 1);

(GSM\_TX\_DW.NSampleEnable\_Counter)++;

}

}

}

/\* Sin: '<Root>/Sine Wave2' \*/

if (GSM\_TX\_DW.systemEnable != 0) {

GSM\_TX\_DW.lastSin = sin(GSM\_TX\_P.SineWave2\_Freq \*

(((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001));

GSM\_TX\_DW.lastCos = cos(GSM\_TX\_P.SineWave2\_Freq \*

(((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001));

GSM\_TX\_DW.systemEnable = 0;

}

/\* Sin: '<Root>/Sine Wave2' \*/

GSM\_TX\_B.SineWave2 = ((GSM\_TX\_DW.lastSin \* GSM\_TX\_P.SineWave2\_PCos +

GSM\_TX\_DW.lastCos \* GSM\_TX\_P.SineWave2\_PSin) \* GSM\_TX\_P.SineWave2\_HCos +

(GSM\_TX\_DW.lastCos \* GSM\_TX\_P.SineWave2\_PCos -

GSM\_TX\_DW.lastSin \* GSM\_TX\_P.SineWave2\_PSin) \* GSM\_TX\_P.SineWave2\_Hsin) \*

GSM\_TX\_P.SineWave2\_Amp + GSM\_TX\_P.SineWave2\_Bias;

/\* Sin: '<Root>/Sine Wave3' \*/

if (GSM\_TX\_DW.systemEnable\_a != 0) {

GSM\_TX\_DW.lastSin\_g = sin(GSM\_TX\_P.SineWave3\_Freq \*

(((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001));

GSM\_TX\_DW.lastCos\_j = cos(GSM\_TX\_P.SineWave3\_Freq \*

(((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001));

GSM\_TX\_DW.systemEnable\_a = 0;

}

/\* Sin: '<Root>/Sine Wave3' \*/

GSM\_TX\_B.SineWave3 = ((GSM\_TX\_DW.lastSin\_g \* GSM\_TX\_P.SineWave3\_PCos +

GSM\_TX\_DW.lastCos\_j \* GSM\_TX\_P.SineWave3\_PSin) \* GSM\_TX\_P.SineWave3\_HCos +

(GSM\_TX\_DW.lastCos\_j \* GSM\_TX\_P.SineWave3\_PCos -

GSM\_TX\_DW.lastSin\_g \* GSM\_TX\_P.SineWave3\_PSin) \* GSM\_TX\_P.SineWave3\_Hsin) \*

GSM\_TX\_P.SineWave3\_Amp + GSM\_TX\_P.SineWave3\_Bias;

/\* Sin: '<Root>/Sine Wave1' \*/

if (GSM\_TX\_DW.systemEnable\_e != 0) {

GSM\_TX\_DW.lastSin\_k = sin(GSM\_TX\_P.SineWave1\_Freq \*

(((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001));

GSM\_TX\_DW.lastCos\_a = cos(GSM\_TX\_P.SineWave1\_Freq \*

(((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001));

GSM\_TX\_DW.systemEnable\_e = 0;

}

/\* Sin: '<Root>/Sine Wave1' \*/

GSM\_TX\_B.SineWave1 = ((GSM\_TX\_DW.lastSin\_k \* GSM\_TX\_P.SineWave1\_PCos +

GSM\_TX\_DW.lastCos\_a \* GSM\_TX\_P.SineWave1\_PSin) \* GSM\_TX\_P.SineWave1\_HCos +

(GSM\_TX\_DW.lastCos\_a \* GSM\_TX\_P.SineWave1\_PCos -

GSM\_TX\_DW.lastSin\_k \* GSM\_TX\_P.SineWave1\_PSin) \* GSM\_TX\_P.SineWave1\_Hsin) \*

GSM\_TX\_P.SineWave1\_Amp + GSM\_TX\_P.SineWave1\_Bias;

/\* Sum: '<Root>/Sum1' \*/

GSM\_TX\_B.Sum1 = (GSM\_TX\_B.SineWave2 + GSM\_TX\_B.SineWave3) +

GSM\_TX\_B.SineWave1;

/\* TransportDelay: '<Root>/Transport Delay2' \*/

{

real\_T \*\*uBuffer = (real\_T\*\*)

&GSM\_TX\_DW.TransportDelay2\_PWORK.TUbufferPtrs[0];

real\_T simTime = GSM\_TX\_M->Timing.t[0];

real\_T tMinusDelay = simTime - GSM\_TX\_P.TransportDelay2\_Delay;

GSM\_TX\_B.TransportDelay2 = rt\_TDelayInterpolate(

tMinusDelay,

0.0,

\*uBuffer,

GSM\_TX\_DW.TransportDelay2\_IWORK.CircularBufSize,

&GSM\_TX\_DW.TransportDelay2\_IWORK.Last,

GSM\_TX\_DW.TransportDelay2\_IWORK.Tail,

GSM\_TX\_DW.TransportDelay2\_IWORK.Head,

GSM\_TX\_P.TransportDelay2\_InitOutput,

1,

0);

}

/\* Sum: '<Root>/Sum4' incorporates:

\* Constant: '<Root>/Constant3'

\*/

GSM\_TX\_B.Sum4 = GSM\_TX\_B.TransportDelay2 - GSM\_TX\_P.Constant3\_Value;

/\* Switch: '<Root>/Clock' \*/

if (GSM\_TX\_B.Sum4 > GSM\_TX\_P.Clock\_Threshold) {

/\* Switch: '<Root>/Clock' \*/

GSM\_TX\_B.Clock = GSM\_TX\_B.Sum4;

} else {

/\* Switch: '<Root>/Clock' incorporates:

\* Constant: '<Root>/Constant2'

\*/

GSM\_TX\_B.Clock = GSM\_TX\_P.Constant2\_Value;

}

/\* End of Switch: '<Root>/Clock' \*/

/\* Switch: '<Root>/Multiplexer carrier 100MHz' \*/

if (GSM\_TX\_B.Clock > GSM\_TX\_P.Multiplexercarrier100MHz\_Thresh) {

/\* Switch: '<Root>/Multiplexer carrier 100MHz' \*/

GSM\_TX\_B.Multiplexercarrier100MHz = GSM\_TX\_B.Sum1;

} else {

/\* Switch: '<Root>/Multiplexer carrier 100MHz' \*/

GSM\_TX\_B.Multiplexercarrier100MHz = 0.0;

}

/\* End of Switch: '<Root>/Multiplexer carrier 100MHz' \*/

/\* S-Function (sdspcumsumprod): '<S4>/Cumulative Sum' \*/

acc = GSM\_TX\_DW.CumulativeSum\_RunningCumVal;

acc += GSM\_TX\_B.Multiplexercarrier100MHz;

/\* S-Function (sdspcumsumprod): '<S4>/Cumulative Sum' \*/

GSM\_TX\_B.CumulativeSum = acc;

/\* S-Function (sdspcumsumprod): '<S4>/Cumulative Sum' \*/

GSM\_TX\_DW.CumulativeSum\_RunningCumVal = acc;

/\* S-Function (scomclock): '<S17>/Time' \*/

/\* Communications Toolbox (scomclock) - '<S17>/Time' \*/

{

real\_T \*y = (real\_T \*)&GSM\_TX\_B.Time;

real\_T time = (((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001);

\*y = time;

}

/\* UnitDelay: '<S18>/UD' \*/

GSM\_TX\_B.Uk1 = GSM\_TX\_DW.UD\_DSTATE;

/\* Sum: '<S18>/Diff' \*/

GSM\_TX\_B.Diff = GSM\_TX\_B.Time - GSM\_TX\_B.Uk1;

/\* Product: '<S4>/Product' \*/

GSM\_TX\_B.Product\_m = GSM\_TX\_B.CumulativeSum \* GSM\_TX\_B.Diff;

/\* Gain: '<S4>/Sensitivity1' \*/

acc = 6.2831853071795862 \* GSM\_TX\_P.FMModulator100MHz\_Kc;

/\* Gain: '<S4>/Sensitivity1' \*/

GSM\_TX\_B.Sensitivity1 = acc \* GSM\_TX\_B.Product\_m;

/\* Gain: '<S4>/Gain' \*/

acc = 6.2831853071795862 \* GSM\_TX\_P.FMModulator100MHz\_Fc;

/\* Gain: '<S4>/Gain' \*/

GSM\_TX\_B.Gain = acc \* GSM\_TX\_B.Time;

/\* Sum: '<S4>/Sum' incorporates:

\* Constant: '<S4>/Constant'

\*/

GSM\_TX\_B.Sum = (GSM\_TX\_B.Sensitivity1 + GSM\_TX\_B.Gain) +

GSM\_TX\_P.FMModulator100MHz\_Ph;

/\* Trigonometry: '<S4>/Trigonometric Function' \*/

GSM\_TX\_B.TrigonometricFunction = cos(GSM\_TX\_B.Sum);

/\* DiscreteFir: '<S2>/Digital Filter' \*/

GSM\_TX\_B.DigitalFilter = GSM\_TX\_B.TrigonometricFunction;

/\* DiscreteFir: '<S2>/Digital Filter' \*/

acc = 0.0;

/\* load input sample \*/

dspPi = GSM\_TX\_B.DigitalFilter;

for (zero = 0; zero < 253; zero++) {

/\* shift state \*/

uPrev = dspPi;

dspPi = GSM\_TX\_DW.DigitalFilter\_states[zero];

GSM\_TX\_DW.DigitalFilter\_states[zero] = uPrev;

/\* compute one tap \*/

uPrev \*= GSM\_TX\_P.DigitalFilter\_Coefficients[zero];

acc += uPrev;

}

/\* compute last tap \*/

uPrev = GSM\_TX\_P.DigitalFilter\_Coefficients[zero] \* dspPi;

acc += uPrev;

/\* DiscreteFir: '<S2>/Digital Filter' \*/

/\* store output sample \*/

GSM\_TX\_B.DigitalFilter = acc;

/\* Sum: '<Root>/Sum of Elements' \*/

acc = GSM\_TX\_B.DigitalFilter;

/\* Sum: '<Root>/Sum of Elements' \*/

GSM\_TX\_B.SumofElements = acc;

/\* DiscreteFir: '<S1>/Digital Filter' \*/

GSM\_TX\_B.DigitalFilter\_m = GSM\_TX\_B.SumofElements;

/\* DiscreteFir: '<S1>/Digital Filter' \*/

acc = 0.0;

/\* load input sample \*/

dspPi = GSM\_TX\_B.DigitalFilter\_m;

for (zero = 0; zero < 253; zero++) {

/\* shift state \*/

uPrev = dspPi;

dspPi = GSM\_TX\_DW.DigitalFilter\_states\_n[zero];

GSM\_TX\_DW.DigitalFilter\_states\_n[zero] = uPrev;

/\* compute one tap \*/

uPrev \*= GSM\_TX\_P.DigitalFilter\_Coefficients\_k[zero];

acc += uPrev;

}

/\* compute last tap \*/

uPrev = GSM\_TX\_P.DigitalFilter\_Coefficients\_k[zero] \* dspPi;

acc += uPrev;

/\* DiscreteFir: '<S1>/Digital Filter' \*/

/\* store output sample \*/

GSM\_TX\_B.DigitalFilter\_m = acc;

/\* S-Function (scomclock): '<S12>/Time' \*/

/\* Communications Toolbox (scomclock) - '<S12>/Time' \*/

{

real\_T \*y = (real\_T \*)&GSM\_TX\_B.Time\_d;

real\_T time = (((GSM\_TX\_M->Timing.clockTick1+GSM\_TX\_M->Timing.clockTickH1\*

4294967296.0)) \* 0.001);

\*y = time;

}

/\* UnitDelay: '<S15>/UD' \*/

GSM\_TX\_B.Uk1\_l = GSM\_TX\_DW.UD\_DSTATE\_p;

/\* S-Function (sdspdelay): '<S10>/Delay' \*/

zero = GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX;

GSM\_TX\_B.Delay = GSM\_TX\_DW.Delay\_IC\_BUFF[zero];

/\* S-Function (sdspdelay): '<S7>/Delay' \*/

zero = GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX\_i;

/\* S-Function (sdspdelay): '<S7>/Delay' \*/

GSM\_TX\_B.Delay\_b = GSM\_TX\_DW.Delay\_IC\_BUFF\_h[zero];

/\* DiscreteFir: '<S7>/Digital Filter' \*/

acc = GSM\_TX\_DW.DigitalFilter\_states\_i[0];

dspPi = GSM\_TX\_B.DigitalFilter\_m;

uPrev = dspPi \* GSM\_TX\_P.DigitalFilter\_Coefficients\_e[0];

acc += uPrev;

/\* DiscreteFir: '<S7>/Digital Filter' \*/

GSM\_TX\_B.DigitalFilter\_k = acc;

for (zero = 0; zero <= 96; zero += 2) {

/\* DiscreteFir: '<S7>/Digital Filter' \*/

tmp = \_mm\_loadu\_pd(&GSM\_TX\_P.DigitalFilter\_Coefficients\_e[zero + 1]);

/\* DiscreteFir: '<S7>/Digital Filter' \*/

tmp = \_mm\_mul\_pd(tmp, \_mm\_set1\_pd(dspPi));

tmp\_0 = \_mm\_loadu\_pd(&GSM\_TX\_DW.DigitalFilter\_states\_i[zero + 1]);

tmp = \_mm\_add\_pd(tmp\_0, tmp);

/\* DiscreteFir: '<S7>/Digital Filter' \*/

\_mm\_storeu\_pd(&GSM\_TX\_DW.DigitalFilter\_states\_i[zero], tmp);

}

/\* DiscreteFir: '<S7>/Digital Filter' \*/

for (zero = 98; zero < 99; zero++) {

acc = GSM\_TX\_DW.DigitalFilter\_states\_i[zero + 1];

uPrev = GSM\_TX\_P.DigitalFilter\_Coefficients\_e[zero + 1] \* dspPi;

acc += uPrev;

GSM\_TX\_DW.DigitalFilter\_states\_i[zero] = acc;

}

acc = dspPi \* GSM\_TX\_P.DigitalFilter\_Coefficients\_e[100];

GSM\_TX\_DW.DigitalFilter\_states\_i[99] = acc;

/\* RealImagToComplex: '<S7>/Join' \*/

GSM\_TX\_B.Join.re = GSM\_TX\_B.Delay\_b;

GSM\_TX\_B.Join.im = GSM\_TX\_B.DigitalFilter\_k;

/\* Product: '<S3>/Product' incorporates:

\* RealImagToComplex: '<S7>/Join'

\* S-Function (sdspdelay): '<S10>/Delay'

\*/

acc = GSM\_TX\_B.Delay.re \* GSM\_TX\_B.Join.re - GSM\_TX\_B.Delay.im \*

GSM\_TX\_B.Join.im;

uPrev = GSM\_TX\_B.Delay.re \* GSM\_TX\_B.Join.im + GSM\_TX\_B.Delay.im \*

GSM\_TX\_B.Join.re;

/\* Product: '<S3>/Product' \*/

GSM\_TX\_B.Product.re = acc;

GSM\_TX\_B.Product.im = uPrev;

/\* ComplexToMagnitudeAngle: '<S11>/Complex to Magnitude-Angle' incorporates:

\* Product: '<S3>/Product'

\*/

GSM\_TX\_B.ComplextoMagnitudeAngle = rt\_atan2d\_snf(GSM\_TX\_B.Product.im,

GSM\_TX\_B.Product.re);

/\* S-Function (sdspunwrap2): '<S11>/Unwrap' \*/

if (GSM\_TX\_DW.Unwrap\_FirstStep) {

GSM\_TX\_DW.Unwrap\_Prev = GSM\_TX\_B.ComplextoMagnitudeAngle;

GSM\_TX\_DW.Unwrap\_FirstStep = false;

}

acc = GSM\_TX\_DW.Unwrap\_Cumsum;

uPrev = GSM\_TX\_DW.Unwrap\_Prev;

uPrev = GSM\_TX\_B.ComplextoMagnitudeAngle - uPrev;

dspPi = (uPrev + 3.1415926535897931) / 6.2831853071795862;

dspPi = floor(dspPi);

dspPi = uPrev - 6.2831853071795862 \* dspPi;

if ((dspPi == -3.1415926535897931) && (uPrev > 0.0)) {

dspPi = 3.1415926535897931;

}

uPrev = dspPi - uPrev;

if (fabs(uPrev) > 3.1415926535897931) {

acc += uPrev;

}

uPrev = GSM\_TX\_B.ComplextoMagnitudeAngle;

/\* S-Function (sdspunwrap2): '<S11>/Unwrap' \*/

GSM\_TX\_B.Unwrap = GSM\_TX\_B.ComplextoMagnitudeAngle + acc;

/\* S-Function (sdspunwrap2): '<S11>/Unwrap' \*/

GSM\_TX\_DW.Unwrap\_Prev = uPrev;

GSM\_TX\_DW.Unwrap\_Cumsum = acc;

/\* UnitDelay: '<S14>/UD' \*/

GSM\_TX\_B.Uk1\_g = GSM\_TX\_DW.UD\_DSTATE\_m;

/\* TransportDelay: '<Root>/Error correction Delay 1' \*/

{

real\_T \*\*uBuffer = (real\_T\*\*)

&GSM\_TX\_DW.ErrorcorrectionDelay1\_PWORK.TUbufferPtrs[0];

real\_T simTime = GSM\_TX\_M->Timing.t[0];

real\_T tMinusDelay = simTime - GSM\_TX\_P.ErrorcorrectionDelay1\_Delay;

GSM\_TX\_B.ErrorcorrectionDelay1 = rt\_TDelayInterpolate(

tMinusDelay,

0.0,

\*uBuffer,

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.CircularBufSize,

&GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Last,

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Tail,

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head,

GSM\_TX\_P.ErrorcorrectionDelay1\_InitOutpu,

1,

0);

}

/\* Switch: '<Root>/Message 1 ' \*/

if (GSM\_TX\_B.ErrorcorrectionDelay1 > GSM\_TX\_P.Message1\_Threshold) {

/\* Switch: '<S9>/Switch' \*/

if (GSM\_TX\_B.NSampleEnable) {

/\* Switch: '<S9>/Switch' \*/

GSM\_TX\_B.Switch = 0.0;

} else {

/\* Sum: '<S14>/Diff' \*/

GSM\_TX\_B.Diff\_n = GSM\_TX\_B.Unwrap - GSM\_TX\_B.Uk1\_g;

/\* Sum: '<S15>/Diff' \*/

GSM\_TX\_B.Diff\_f = GSM\_TX\_B.Time\_d - GSM\_TX\_B.Uk1\_l;

/\* Product: '<S3>/Product1' \*/

GSM\_TX\_B.Product1 = 1.0 / GSM\_TX\_B.Diff\_f \* GSM\_TX\_B.Diff\_n;

/\* Product: '<S3>/Product3' incorporates:

\* Constant: '<S3>/Constant'

\*/

acc = 1.0 / (6.2831853071795862 \* GSM\_TX\_P.FMDemodulator100MHz\_Kc);

/\* Product: '<S3>/Product3' \*/

GSM\_TX\_B.Product3 = GSM\_TX\_B.Product1 \* acc;

/\* Switch: '<S9>/Switch' \*/

GSM\_TX\_B.Switch = GSM\_TX\_B.Product3;

}

/\* End of Switch: '<S9>/Switch' \*/

/\* Switch: '<Root>/Message 1 ' \*/

GSM\_TX\_B.Message1 = GSM\_TX\_B.Switch;

} else {

/\* Switch: '<Root>/Message 1 ' \*/

GSM\_TX\_B.Message1 = 0.0;

}

/\* End of Switch: '<Root>/Message 1 ' \*/

/\* Gain: '<S10>/Gain' \*/

acc = -6.2831853071795862 \* GSM\_TX\_P.FMDemodulator100MHz\_Fc;

/\* Gain: '<S10>/Gain' \*/

GSM\_TX\_B.Gain\_e = acc \* GSM\_TX\_B.Time\_d;

/\* Trigonometry: '<S10>/Complex Exponential' \*/

acc = GSM\_TX\_B.Gain\_e;

/\* Trigonometry: '<S10>/Complex Exponential' \*/

GSM\_TX\_B.ComplexExponential.re = cos(acc);

GSM\_TX\_B.ComplexExponential.im = sin(acc);

}

/\* Matfile logging \*/

rt\_UpdateTXYLogVars(GSM\_TX\_M->rtwLogInfo, (GSM\_TX\_M->Timing.t));

{

real\_T HoldCosine;

real\_T HoldSine;

int32\_T buffStartIdx;

/\* Update for Sin: '<Root>/Sine Wave2' \*/

HoldSine = GSM\_TX\_DW.lastSin;

HoldCosine = GSM\_TX\_DW.lastCos;

GSM\_TX\_DW.lastSin = HoldSine \* GSM\_TX\_P.SineWave2\_HCos + HoldCosine \*

GSM\_TX\_P.SineWave2\_Hsin;

GSM\_TX\_DW.lastCos = HoldCosine \* GSM\_TX\_P.SineWave2\_HCos - HoldSine \*

GSM\_TX\_P.SineWave2\_Hsin;

/\* Update for Sin: '<Root>/Sine Wave3' \*/

HoldSine = GSM\_TX\_DW.lastSin\_g;

HoldCosine = GSM\_TX\_DW.lastCos\_j;

GSM\_TX\_DW.lastSin\_g = HoldSine \* GSM\_TX\_P.SineWave3\_HCos + HoldCosine \*

GSM\_TX\_P.SineWave3\_Hsin;

GSM\_TX\_DW.lastCos\_j = HoldCosine \* GSM\_TX\_P.SineWave3\_HCos - HoldSine \*

GSM\_TX\_P.SineWave3\_Hsin;

/\* Update for Sin: '<Root>/Sine Wave1' \*/

HoldSine = GSM\_TX\_DW.lastSin\_k;

HoldCosine = GSM\_TX\_DW.lastCos\_a;

GSM\_TX\_DW.lastSin\_k = HoldSine \* GSM\_TX\_P.SineWave1\_HCos + HoldCosine \*

GSM\_TX\_P.SineWave1\_Hsin;

GSM\_TX\_DW.lastCos\_a = HoldCosine \* GSM\_TX\_P.SineWave1\_HCos - HoldSine \*

GSM\_TX\_P.SineWave1\_Hsin;

/\* Update for TransportDelay: '<Root>/Transport Delay2' \*/

{

real\_T \*\*uBuffer = (real\_T\*\*)

&GSM\_TX\_DW.TransportDelay2\_PWORK.TUbufferPtrs[0];

real\_T simTime = GSM\_TX\_M->Timing.t[0];

GSM\_TX\_DW.TransportDelay2\_IWORK.Head =

((GSM\_TX\_DW.TransportDelay2\_IWORK.Head <

(GSM\_TX\_DW.TransportDelay2\_IWORK.CircularBufSize-1)) ?

(GSM\_TX\_DW.TransportDelay2\_IWORK.Head+1) : 0);

if (GSM\_TX\_DW.TransportDelay2\_IWORK.Head ==

GSM\_TX\_DW.TransportDelay2\_IWORK.Tail) {

GSM\_TX\_DW.TransportDelay2\_IWORK.Tail =

((GSM\_TX\_DW.TransportDelay2\_IWORK.Tail <

(GSM\_TX\_DW.TransportDelay2\_IWORK.CircularBufSize-1)) ?

(GSM\_TX\_DW.TransportDelay2\_IWORK.Tail+1) : 0);

}

(\*uBuffer + GSM\_TX\_DW.TransportDelay2\_IWORK.CircularBufSize)

[GSM\_TX\_DW.TransportDelay2\_IWORK.Head] = simTime;

(\*uBuffer)[GSM\_TX\_DW.TransportDelay2\_IWORK.Head] = GSM\_TX\_B.Clock;

}

/\* Update for UnitDelay: '<S18>/UD' \*/

GSM\_TX\_DW.UD\_DSTATE = GSM\_TX\_B.Time;

/\* Update for UnitDelay: '<S15>/UD' \*/

GSM\_TX\_DW.UD\_DSTATE\_p = GSM\_TX\_B.Time\_d;

/\* Update for S-Function (sdspdelay): '<S10>/Delay' incorporates:

\* Trigonometry: '<S10>/Complex Exponential'

\*/

buffStartIdx = GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX;

GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX = (buffStartIdx + 1) % 50;

GSM\_TX\_DW.Delay\_IC\_BUFF[buffStartIdx] = GSM\_TX\_B.ComplexExponential;

/\* Update for S-Function (sdspdelay): '<S7>/Delay' \*/

buffStartIdx = GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX\_i;

GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX\_i = (buffStartIdx + 1) % 50;

GSM\_TX\_DW.Delay\_IC\_BUFF\_h[buffStartIdx] = GSM\_TX\_B.DigitalFilter\_m;

/\* Update for UnitDelay: '<S14>/UD' \*/

GSM\_TX\_DW.UD\_DSTATE\_m = GSM\_TX\_B.Unwrap;

/\* Update for TransportDelay: '<Root>/Error correction Delay 1' \*/

{

real\_T \*\*uBuffer = (real\_T\*\*)

&GSM\_TX\_DW.ErrorcorrectionDelay1\_PWORK.TUbufferPtrs[0];

real\_T simTime = GSM\_TX\_M->Timing.t[0];

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head =

((GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head <

(GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.CircularBufSize-1)) ?

(GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head+1) : 0);

if (GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head ==

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Tail) {

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Tail =

((GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Tail <

(GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.CircularBufSize-1)) ?

(GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Tail+1) : 0);

}

(\*uBuffer + GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.CircularBufSize)

[GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head] = simTime;

(\*uBuffer)[GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head] = GSM\_TX\_B.Clock;

}

}

/\* signal main to stop simulation \*/

{ /\* Sample time: [0.0s, 0.0s] \*/

if ((rtmGetTFinal(GSM\_TX\_M)!=-1) &&

!((rtmGetTFinal(GSM\_TX\_M)-GSM\_TX\_M->Timing.t[0]) > GSM\_TX\_M->Timing.t[0]

\* (DBL\_EPSILON))) {

rtmSetErrorStatus(GSM\_TX\_M, "Simulation finished");

}

}

/\* Update absolute time for base rate \*/

/\* The "clockTick0" counts the number of times the code of this task has

\* been executed. The absolute time is the multiplication of "clockTick0"

\* and "Timing.stepSize0". Size of "clockTick0" ensures timer will not

\* overflow during the application lifespan selected.

\* Timer of this task consists of two 32 bit unsigned integers.

\* The two integers represent the low bits Timing.clockTick0 and the high bits

\* Timing.clockTickH0. When the low bit overflows to 0, the high bits increment.

\*/

if (!(++GSM\_TX\_M->Timing.clockTick0)) {

++GSM\_TX\_M->Timing.clockTickH0;

}

GSM\_TX\_M->Timing.t[0] = GSM\_TX\_M->Timing.clockTick0 \*

GSM\_TX\_M->Timing.stepSize0 + GSM\_TX\_M->Timing.clockTickH0 \*

GSM\_TX\_M->Timing.stepSize0 \* 4294967296.0;

{

/\* Update absolute timer for sample time: [0.001s, 0.0s] \*/

/\* The "clockTick1" counts the number of times the code of this task has

\* been executed. The resolution of this integer timer is 0.001, which is the step size

\* of the task. Size of "clockTick1" ensures timer will not overflow during the

\* application lifespan selected.

\* Timer of this task consists of two 32 bit unsigned integers.

\* The two integers represent the low bits Timing.clockTick1 and the high bits

\* Timing.clockTickH1. When the low bit overflows to 0, the high bits increment.

\*/

GSM\_TX\_M->Timing.clockTick1++;

if (!GSM\_TX\_M->Timing.clockTick1) {

GSM\_TX\_M->Timing.clockTickH1++;

}

}

}

/\* Model initialize function \*/

void GSM\_TX\_initialize(void)

{

/\* Registration code \*/

/\* initialize non-finites \*/

rt\_InitInfAndNaN(sizeof(real\_T));

/\* initialize real-time model \*/

(void) memset((void \*)GSM\_TX\_M, 0,

sizeof(RT\_MODEL\_GSM\_TX\_T));

{

/\* Setup solver object \*/

rtsiSetSimTimeStepPtr(&GSM\_TX\_M->solverInfo, &GSM\_TX\_M->Timing.simTimeStep);

rtsiSetTPtr(&GSM\_TX\_M->solverInfo, &rtmGetTPtr(GSM\_TX\_M));

rtsiSetStepSizePtr(&GSM\_TX\_M->solverInfo, &GSM\_TX\_M->Timing.stepSize0);

rtsiSetErrorStatusPtr(&GSM\_TX\_M->solverInfo, (&rtmGetErrorStatus(GSM\_TX\_M)));

rtsiSetRTModelPtr(&GSM\_TX\_M->solverInfo, GSM\_TX\_M);

}

rtsiSetSimTimeStep(&GSM\_TX\_M->solverInfo, MAJOR\_TIME\_STEP);

rtsiSetSolverName(&GSM\_TX\_M->solverInfo,"FixedStepDiscrete");

rtmSetTPtr(GSM\_TX\_M, &GSM\_TX\_M->Timing.tArray[0]);

rtmSetTFinal(GSM\_TX\_M, -1);

GSM\_TX\_M->Timing.stepSize0 = 0.001;

/\* Setup for data logging \*/

{

static RTWLogInfo rt\_DataLoggingInfo;

rt\_DataLoggingInfo.loggingInterval = (NULL);

GSM\_TX\_M->rtwLogInfo = &rt\_DataLoggingInfo;

}

/\* Setup for data logging \*/

{

rtliSetLogXSignalInfo(GSM\_TX\_M->rtwLogInfo, (NULL));

rtliSetLogXSignalPtrs(GSM\_TX\_M->rtwLogInfo, (NULL));

rtliSetLogT(GSM\_TX\_M->rtwLogInfo, "tout");

rtliSetLogX(GSM\_TX\_M->rtwLogInfo, "");

rtliSetLogXFinal(GSM\_TX\_M->rtwLogInfo, "");

rtliSetLogVarNameModifier(GSM\_TX\_M->rtwLogInfo, "rt\_");

rtliSetLogFormat(GSM\_TX\_M->rtwLogInfo, 0);

rtliSetLogMaxRows(GSM\_TX\_M->rtwLogInfo, 1000);

rtliSetLogDecimation(GSM\_TX\_M->rtwLogInfo, 1);

rtliSetLogY(GSM\_TX\_M->rtwLogInfo, "");

rtliSetLogYSignalInfo(GSM\_TX\_M->rtwLogInfo, (NULL));

rtliSetLogYSignalPtrs(GSM\_TX\_M->rtwLogInfo, (NULL));

}

/\* block I/O \*/

(void) memset(((void \*) &GSM\_TX\_B), 0,

sizeof(B\_GSM\_TX\_T));

/\* states (dwork) \*/

(void) memset((void \*)&GSM\_TX\_DW, 0,

sizeof(DW\_GSM\_TX\_T));

/\* Matfile logging \*/

rt\_StartDataLoggingWithStartTime(GSM\_TX\_M->rtwLogInfo, 0.0, rtmGetTFinal

(GSM\_TX\_M), GSM\_TX\_M->Timing.stepSize0, (&rtmGetErrorStatus(GSM\_TX\_M)));

/\* Start for TransportDelay: '<Root>/Transport Delay2' \*/

{

real\_T \*pBuffer = &GSM\_TX\_DW.TransportDelay2\_RWORK.TUbufferArea[0];

GSM\_TX\_DW.TransportDelay2\_IWORK.Tail = 0;

GSM\_TX\_DW.TransportDelay2\_IWORK.Head = 0;

GSM\_TX\_DW.TransportDelay2\_IWORK.Last = 0;

GSM\_TX\_DW.TransportDelay2\_IWORK.CircularBufSize = 1024;

pBuffer[0] = GSM\_TX\_P.TransportDelay2\_InitOutput;

pBuffer[1024] = GSM\_TX\_M->Timing.t[0];

GSM\_TX\_DW.TransportDelay2\_PWORK.TUbufferPtrs[0] = (void \*) &pBuffer[0];

}

/\* Start for TransportDelay: '<Root>/Error correction Delay 1' \*/

{

real\_T \*pBuffer = &GSM\_TX\_DW.ErrorcorrectionDelay1\_RWORK.TUbufferArea[0];

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Tail = 0;

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Head = 0;

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.Last = 0;

GSM\_TX\_DW.ErrorcorrectionDelay1\_IWORK.CircularBufSize = 1024;

pBuffer[0] = GSM\_TX\_P.ErrorcorrectionDelay1\_InitOutpu;

pBuffer[1024] = GSM\_TX\_M->Timing.t[0];

GSM\_TX\_DW.ErrorcorrectionDelay1\_PWORK.TUbufferPtrs[0] = (void \*) &pBuffer[0];

}

{

int32\_T i;

/\* InitializeConditions for S-Function (sdspnsamp2): '<S9>/N-Sample Enable' \*/

/\* DSP System Toolbox N-Sample Enable (sdspnsamp2) - '<S9>/N-Sample Enable' \*/

GSM\_TX\_DW.NSampleEnable\_Counter = (uint32\_T) 0;

/\* InitializeConditions for S-Function (sdspcumsumprod): '<S4>/Cumulative Sum' \*/

GSM\_TX\_DW.CumulativeSum\_RunningCumVal = 0.0;

/\* InitializeConditions for UnitDelay: '<S18>/UD' \*/

GSM\_TX\_DW.UD\_DSTATE = GSM\_TX\_P.Difference\_ICPrevInput;

for (i = 0; i < 253; i++) {

/\* InitializeConditions for DiscreteFir: '<S2>/Digital Filter' \*/

GSM\_TX\_DW.DigitalFilter\_states[i] = GSM\_TX\_P.DigitalFilter\_InitialStates;

/\* InitializeConditions for DiscreteFir: '<S1>/Digital Filter' \*/

GSM\_TX\_DW.DigitalFilter\_states\_n[i] =

GSM\_TX\_P.DigitalFilter\_InitialStates\_b;

}

/\* InitializeConditions for UnitDelay: '<S15>/UD' \*/

GSM\_TX\_DW.UD\_DSTATE\_p = GSM\_TX\_P.Difference\_ICPrevInput\_n;

/\* InitializeConditions for S-Function (sdspdelay): '<S10>/Delay' \*/

GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX = 0;

/\* InitializeConditions for S-Function (sdspdelay): '<S7>/Delay' \*/

GSM\_TX\_DW.Delay\_CIRC\_BUF\_IDX\_i = 0;

/\* InitializeConditions for S-Function (sdspdelay): '<S10>/Delay' \*/

memset(&GSM\_TX\_DW.Delay\_IC\_BUFF[0], 0, 50U \* sizeof(creal\_T));

/\* InitializeConditions for S-Function (sdspdelay): '<S7>/Delay' \*/

memset(&GSM\_TX\_DW.Delay\_IC\_BUFF\_h[0], 0, 50U \* sizeof(real\_T));

/\* InitializeConditions for DiscreteFir: '<S7>/Digital Filter' \*/

for (i = 0; i < 100; i++) {

GSM\_TX\_DW.DigitalFilter\_states\_i[i] =

GSM\_TX\_P.DigitalFilter\_InitialStates\_h;

}

/\* End of InitializeConditions for DiscreteFir: '<S7>/Digital Filter' \*/

/\* InitializeConditions for S-Function (sdspunwrap2): '<S11>/Unwrap' \*/

GSM\_TX\_DW.Unwrap\_FirstStep = true;

GSM\_TX\_DW.Unwrap\_Cumsum = 0.0;

/\* InitializeConditions for UnitDelay: '<S14>/UD' \*/

GSM\_TX\_DW.UD\_DSTATE\_m = GSM\_TX\_P.Difference\_ICPrevInput\_e;

}

/\* Enable for Sin: '<Root>/Sine Wave2' \*/

GSM\_TX\_DW.systemEnable = 1;

/\* Enable for Sin: '<Root>/Sine Wave3' \*/

GSM\_TX\_DW.systemEnable\_a = 1;

/\* Enable for Sin: '<Root>/Sine Wave1' \*/

GSM\_TX\_DW.systemEnable\_e = 1;

}

/\* Model terminate function \*/

void GSM\_TX\_terminate(void)

{

/\* (no terminate code required) \*/

}