APPENDIX A

MATLAB Code of PPM Pulse-Based UWB Communication System

First : Generates the energy normalized pulse waveform Special case of the second derivative Gaussian pulse:

Step: 1:-

```
%Parameters Definitions :
%
% 'fc' is the sampling frequency
% 'Tm' is the pulse duration
% 'tau' is the shaping parameter
function [0-1]= waveform(fc,Tm,tau);
% ------------------------------------
% Step 2 : - Pulse Waveform Generation
% ------------------------------------
dt = 1 / fc;           % reference sampling period
OVER = floor(Tm/dt);    % number of samples representing
                        %  the pulse
e = mod(OVER,2);
kbk = floor(OVER/2);
tmp = linspace(dt,Tm/2,kbk);
s = (1-4.*pi.*((tmp./tau).^2)).* ...
    exp(-2.*pi.*((tmp./tau).^2));

if e               % OVER is odd
   for k=1:length(s)
      y(kbk+1)=1;
      y(kbk+1+k)=s(k);
      y(kbk+1-k)=s(k);
   end
else               % OVER is even
   for k=1:length(s)
      y(kbk+k)=s(k);
      y(kbk+1-k)=s(k);
   end
end

E = sum((y.^2).*dt);     % pulse energy
w0 = y ./ (E^0.5);       % energy normalization
```

Second : IR-UWB Reference Signal Generation

```
% ---------------------------
% Step 1: - Input Parameters
% ---------------------------
Pow = -30;       % Average transmitted power (dBm)
fc = 50e9;       % sampling frequency
numbits = 3000;     % number of bits generated by the source
```

```
Ts = 1e-9;       % frame time, i.e. average pulse
                 %  repetition period [s]
Ns = 1;          % number of pulses per bit
Tc = 1e-9;       % chip time [s]
Nh = floor(Ts/Tc);          % cardinality of the Time Hopping code
Np = 30000;          % periodicity of the Time Hopping code
Tm = 0.5e-9;     % pulse duration [s]
tau = 0.65e-9;   % shaping factor for the pulse [s]
dPPM = 0.5e-9;   % time shift introduced by the PPM [s]
G = 0;
% G=0 -> no graphical output
% G=1 -> graphical output
% ---------------------------------------
% Step 2: - Simulating Transmission Chain
% ---------------------------------------
% binary source
bits = _bits(numbits);
% repetition coder
repbits = _repcode(bits,Ns);
% Time Hopping code
THcode = _TH(Nh,Np);
% Pulse Position Modulation + TH
[PPMTHseq,THseq] = ...
   _2PPM_TH(repbits,fc,Tc,Ts,dPPM,THcode);
% Shaping filter
power = (10^(Pow/10))/1000;     % average transmitted power
                          %  (watt)
Ex = power * Ts;                % energy per pulse
w0 = waveform(fc,Tm,tau);       % Energy Normalized pulse
                          %  waveform
wtx = w0 .* sqrt(Ex);           % pulse waveform
Sa = conv(PPMTHseq,wtx);        % Output of the filter
                          %  (with modulation)
Sb = conv(THseq,wtx);           % Output of the filter
                          %  (without modulation)
% Output generation
L = (floor(Ts*fc))*Ns*numbits;
Stx = Sa(1:L);
ref = Sb(1:L);
% --------------------------
% Step 3: - Graphical Output
% --------------------------
if G
F = figure(5);
set(F,'Position',[32 223 951 420]);

tmax = numbits*Ns*Ts;
time = linspace(0,tmax,length(Stx));
P = plot(time,Stx);
```

```
set(P,'LineWidth',[2]);
ylow=-1.5*abs(min(wtx));
yhigh=1.5*max(wtx);
axis([0 tmax ylow yhigh]);
AX=gca;
set(AX,'FontSize',12);
X=xlabel('Time [s]');
set(X,'FontSize',14);
Y=ylabel('Amplitude [V]');
set(Y,'FontSize',14);
for j = 1 : numbits
   tj = (j-1)*Ns*Ts;
   L1=line([tj tj],[ylow yhigh]);
   set(L1,'Color',[0 0 0],'LineStyle', ...
     '--','LineWidth',[2]);
   for k = 0 : Ns-1
      if k > 0
         tn = tj + k*Nh*Tc;
         L2=line([tn tn],[ylow yhigh]);
         set(L2,'Color',[0.5 0.5 0.5],'LineStyle', ...
           '-.','LineWidth',[2]);
      end
      for q = 1 : Nh-1
         th = tj + k*Nh*Tc + q*Tc;
         L3=line([th th],[0.8*ylow 0.8*yhigh]);
         set(L3,'Color',[0 0 0],'LineStyle', ...
           ':','LineWidth',[1]);
      end
   end
end

end % end of graphical output
```

Third: Introduces additive white Gaussian noise over signal

```
function [output,noise] = ...
  _Gnoise1(input,ebno,numbits)

% ------------------------------
% Step One - Introduction of AWGN
% ------------------------------

Eb = (1/numbits)*sum(input.^2); % measured energy per bit
EbNo = 10.^(ebno./10);          % Eb/No in linear unit
No = Eb ./ EbNo;                % Unilateral spectral
                                %  density
nstdv = sqrt(No./2);            % Standard deviation for
                                %  the noise
```

```
for j = 1 : length(EbNo)

   noise(j,:) = nstdv(j) .* randn(1,length(input));
   output(j,:) = noise(j,:) + input;

end
```

Fourth : Attenuates the input signal 'tx' according to the distance 'd' [m], the decaying factor 'gamma' and the constant term 'c0', which represents the reference attenuation at 1 meter.

```
% The function returns the attenuated signal 'rx'
% and the value of the channel gain 'attn'


function [rx,attn] = _pathloss(Stx,c0,d,gamma)

% ------------------------------
% Step 1: - Path loss evaluation
% ------------------------------

attn = (c0/sqrt(d^gamma));
rx = attn .* Stx;
```

Fifth : Evaluates the correlation mask ('mask') in the case of binary PPM UWB signals.

```
% Parameters Definitions :
%'ref' is the reference signal % which is produced by the 2PPM+TH transmitter
% 'fc' is the sampling frequency
% 'numpulses' is the number of transmitted pulses
% 'dPPM' is the value of the PPM shift
%
%
function [mask] = _PPMcorrmask(ref,fc,numpulses,dPPM)

% ---------------------------------------
% Step 1: - Evaluation of the correlation mask
% ---------------------------------------

dt = 1 / fc;

% Energy normalization
Epulse = (sum((ref.^2).*dt))/numpulses;
ref = ref./sqrt(Epulse);

% Mask construction
```

```
PPMsamples = floor (dPPM ./ dt);
sref(1:PPMsamples)=ref(length(ref)- ...
  PPMsamples+1:length(ref));
sref(PPMsamples+1:length(ref)) = ref(1:length(ref)- ...
  PPMsamples);
mask = ref-sref;
```

Finally: receiver performance ppm

```
% Step 1: signal generation

[bits1,THcode,s_ppm1,ref1]= _transmitter_2PPM_TH_rec_ppm1;

% Step 2: parameters

fc = 50e9;
dPPM = 0.5e-9;
Ts = 1e-9;
numbit = 3000;
numpulses1 = 2500; % Ns = 1;
[mppm1] = _PPMcorrmask(ref1,fc,numpulses1,dPPM);

%Step 3:  signal to noise ratio values

exno = [0 2 4 6 9];

[rx_ppm1, noise] = _Gnoise2(s_ppm1, exno, numpulses1);

% Step 4: performance evaluation

[RXbits1, BER1] = _PPMreceiver(rx_ppm1, mppm1, fc, bits1, numbit, 1, Ts);

[bits1,THcode,s_ppm1,ref1]= _transmitter_2PPM_TH_rec_ppm1;


% Step 5: graphics

P1 = semilogy(exno,BER1,'ro:');
set(P1,'LineWidth',[3])
title('performance PPM Data rate = 833 Mbps', 'Color','b')
xlabel('signal to noise ratio')
ylabel('BER')
grid on
```

SECOND PART : SIMULINK Main Communication-Link S-Functions Blocks

1- S-Functions of Different Transmitted Pulse Shapes.

R-Z Manchester Pulse Shape

```
function V=rzmanchester(e,tp,tc)
t=0:1e-12:10*tp;
A=(e/tp)^0.5;
y=A*(rectpuls(t-tc,tp)-rectpuls(t-(tc+tp/6),2*tp/3)-rectpuls(t-(tc+tp/3),tp/3));
plot(t,y);
fs=1e12;
a=fft(y,30000);
b=a.*conj(a);
c=sqrt(b);
d=max(b);
f=fs*(0:15000)/30000;
figure;
plot(f,10*log(b(1:15001)/d));
```

Sinusoidal Pulse Shape

```
function V=sinusoidal(fc,n)
t=0:1e-11:n/fc;
V=sin(2*pi*fc*t).*rectpuls(t-(n/2)*(1/fc),n/fc);
plot(t,V);
a=fft(V,30000);
b=a.*conj(a);
c=sqrt(b);
fs=1e11;
f=fs*(0:15000)/30000;
figure;
plot(f,c(1:15001));
```

Rectangular Pulse Shape

```
unction V=rectanglepulse(e,tp,tc)
A=(e/tp)^0.5;
t=0:1e-11:10*tp;
y=A*rectpuls(t-tc,tp);
plot(t,y);
fs=1e11;
a=fft(y,30000);
b=a.*conj(a);
c=sqrt(b);
d=max(b);
f=fs*(0:15000)/30000;
figure;
plot(f,10*log(b(1:15001)/d));
```

-Gaussian-Monopulse Pulse Shape

```
function [V,t]=GaussianMonopulse(tw,fw,s)
close all;
t=-10*tw:tw/100:10*tw;
V=(1-(t/s).^2).*(exp((-t.^2)/(2*s).^2)/(2*pi*s.^2).^0.5);
plot(t,V);
grid on;
title('Gaussian Monopulse in time domain');
xlabel('Time(sec)')
ylabel('Amplitude');
f=0:fw/100:fw;
Vf=((2*pi*s*f).^2).*exp(-(2*pi*s*f).^2)/2;
figure;
plot(f,Vf);
grid on;
title('Gaussian Monopulse in frequency domain');
xlabel('Frequency(Hz)');
ylabel('Amplitude');
```

Scholts Gaussian 2nd Derivative Pulse Shape

```
function [t,w]=scholtz(A,Tc)
% function [t,w]=scholtz(A,Tc)
% input: A(=sqrt(Ep)), Tc(=Time shift)
% ouput: t(time) :[ns], w(wave form)
% tau=0.2877e-9; % impulse width = 0.2877ns
 % [ns]
t=-2e-9:1e-12:2e-9;
w=A*sqrt( 8/3/tau ) * (1-4*pi*(t/tau-Tc/tau).^2).*exp(-2*pi*(t/tau-Tc/tau).^2);
 return;
function [sys,x0,str,ts] =
UWBmodulation(t,x,u,flag,A,tau,Tf,Tc,delta,xini,bitsPerHop)
 switch flag,
    % Initialization %
  case 0,
    sizes = simsizes;
    sizes.NumContStates  = 0;
    sizes.NumDiscStates  = 6;
    sizes.NumOutputs     = 1;
    sizes.NumInputs      = 1;
    sizes.DirFeedthrough = 1;
    sizes.NumSampleTimes = 1;  % at least one sample time is needed
    sys = simsizes(sizes);
      x0  = [xini 0]; % State of the state register and initial c0, m=5
    str = []; ts  = [0 0];
    % Derivatives %
  case 1,
    sys=[];
     % Update %
  case 2,
```

```
    x(6)=0;
    if abs(round(t/Tf) - t/Tf) < 1e-8
       for cnt=bitsPerHop-1:-1:0
          feedback=mod(x(2)+x(5),2);
          x(6) = x(6) + x(5)*2^cnt;
          for k=5:-1:2
             x(k)=x(k-1);
          end
          x(1)=feedback;
       end
       sys=x;
    else
       sys=[];
    end
 % Outputs %
 case 3,
  tmod=mod(t,Tf);
  sys = scholtzValue(tmod,A,x(6)*Tc+u*delta+2e-9,tau);
 % GetTimeOfNextVarHit %
 case 4,
    sys=[];
 % Terminate %
 case 9,
   sys=[];
 % Unexpected flags %
 otherwise
  error(['Unhandled flag = ',num2str(flag)]);
end
```

2- S-Function of UWB-PPM Data Transmission

```
function [sys,x0,str,ts] = UWBPPMDataSource(t,x,u,flag,sigPower,tau,Tf,Tc,delta,M,link,xini,dataOnState,bitsPerHop,Ns,Np,Ts)

global ACQUISITION_ON;

switch flag,

 % Initialization %
  case 0,
  sizes = simsizes;
  sizes.NumContStates  = 0;
  sizes.NumDiscStates  = 10;
  sizes.NumOutputs     = 2;
  sizes.NumInputs      = 0;
  sizes.DirFeedthrough = 1;
  sizes.NumSampleTimes = 1;   % at least one sample time is needed
```

```
    sys = simsizes(sizes);
      state=xini;
   for k=1:link
      [state,nextC]=nextState(state,bitsPerHop);
   end
      x0  = [state 0 0 0 0 0]; % State of the register(m=5), output time hopping unit c,
data on flag
   str = []; ts  = [0 0];
  % Derivatives %
 case 1,
  sys=[];
 % Update %
 case 2,
    if abs(round(t/Tf)-t/Tf) < 1e-8
       if ACQUISITION_ON==0
          x(7)=0;
       end
       [state,nextC]=nextState(x(1:5),bitsPerHop);
       x(1:5)=state; x(6)=nextC;
       if state==dataOnState'
          x(7)=1; %Data On Flag
          x(8)=t; %Data On State Time
          x(10)=floor(rand(1)*M);
       elseif x(7)==1 & mod(round((t-x(8))/Tf),Ns)==0 & round((t-x(8))/Tf)+Ns<Np
          x(10)=floor(rand(1)*M);
       end
    end
    sys=x;
 % Outputs %
 case 3,
   Ep=1e-3*10^(sigPower/10)*Tf; % Average Pulse Energy
   A=sqrt(Ep);
   tmod=mod(t,Tf);
   if ACQUISITION_ON>0 & x(7)==1
      data = x(10);
      output = scholtzValue(tmod, A, x(6)*Tc+x(10)*delta+floor(2*tau/Ts)*Ts,tau);
      sys=[data output];
   else
      sys = [0 0];
   end
 % GetTimeOfNextVarHit %
 case 4,
   sys=[];
 % Terminate %
 case 9,
   sys=[];
 % Unexpected flags %
 otherwise
  error(['Unhandled flag = ',num2str(flag)]);
```

3- S-Function of Pilot-Hopping-Code Block

```
function code=pilotHoppingCode(xini,bitsPerHop,Np)

code=[]; x=xini;

for k=1:Np
   presentC=0;
   for cnt=bitsPerHop-1:-1:0
      feedback=mod(x(2)+x(5),2);
      presentC = presentC + x(5)*2^cnt;
      for k=5:-1:2
         x(k)=x(k-1);
      end
      x(1)=feedback;
   end
   code=[code presentC];
end
```

4- S-Function block of signal Attenuation and Delay

```
function [sys,x0,str,ts] = ChanDelay (t,x,u,flag,delay,Ts)

switch flag,
  % Initialization %
  case 0,
    sizes = simsizes;
     sizes.NumContStates  = 0;
    sizes.NumDiscStates  = floor(delay/Ts);
    sizes.NumOutputs     = 1;
    sizes.NumInputs      = 1;
    sizes.DirFeedthrough = 1;
    sizes.NumSampleTimes = 1;   % at least one sample time is needed
     sys = simsizes(sizes);
       x0  = zeros(1,floor(delay/Ts));
    str = []; ts  = [0 0];
  % Derivatives %
  case 1,
    sys=[];
  % Update %
  case 2,
     if floor(delay/Ts)==0
        sys=[];
     else
        l=length(x);
        x(2:l)=x(1:l-1);
        x(1)=u;
        sys=x;
     end
  % Outputs %
  case 3,
```

```
    if floor(delay/Ts)==0
       sys=u;
    else
       sys=x(length(x));
    end
 % GetTimeOfNextVarHit %
  case 4,
   sys=[];
 % Terminate %
 case 9,
    sys=[];
 % Unexpected flags %
 otherwise
   error(['Unhandled flag = ',num2str(flag)]);
end
```

5- S-Function of AWGN Channel Model

```
function [sys,x0,str,ts] = AWGNchan(t,x,u,flag,noisePower)

switch flag,
  % Initialization %
  case 0,
    sizes = simsizes;

    sizes.NumContStates  = 0;
    sizes.NumDiscStates  = 0;
    sizes.NumOutputs     = 1;
    sizes.NumInputs      = 1;
    sizes.DirFeedthrough = 1;
    sizes.NumSampleTimes = 1;   % at least one sample time is needed

    sys = simsizes(sizes);

    x0  = [];
    str = []; ts  = [0 0];
  % Derivatives %
   case 1,
    sys=[];
  % Update %
  case 2,
     sys=[];
  % Outputs %
  case 3,
     noiseVariance=1e-3*10^(noisePower/10);
     sys=u+sqrt(noiseVariance)*randn(1);
  % GetTimeOfNextVarHit %
  case 4,
    sys=[];
```

```
   % Terminate %
  case 9,
    sys=[];
      % Unexpected flags %
  otherwise
   error(['Unhandled flag = ',num2str(flag)]);
 end

function code=pilotHoppingCode(xini,bitsPerHop,Np)

code=[]; x=xini;

for k=1:Np
   presentC=0;
   for cnt=bitsPerHop-1:-1:0
      feedback=mod(x(2)+x(5),2);
      presentC = presentC + x(5)*2^cnt;
      for k=5:-1:2
         x(k)=x(k-1);
      end
      x(1)=feedback;
   end
   code=[code presentC];
end
```

6- S-Function of Pilot-Template-Generator Block.

```
function                                    [sys,x0,str,ts]                                    =
PilotTemplateGen(t,x,u,flag,tau,Tf,Tc,xini,bitsPerHop,Np,Ts)

global UWB_SIG_DETECT_ON UWB_SIG_DETECT_TIME;
global ACQUISITION_ON;
global NEXT_PULSE_TIME;
global NEXT_HOPPING_ON;
global INTEGRATION_START_TIME;
global ACQ_FRAME_TIME ACQ_REGISTER_STATE;
switch flag,

  % Initialization %
  case 0,
   sizes = simsizes;

   sizes.NumContStates  = 0;
   sizes.NumDiscStates  = 9+Np;
   sizes.NumOutputs     = 1;
   sizes.NumInputs      = 0;
   sizes.DirFeedthrough = 0;
   sizes.NumSampleTimes = 1;   % at least one sample time is needed
   sys = simsizes(sizes);
```

```
    code=pilotHoppingCode(xini,bitsPerHop,Np);
    x0   = [0 0 zeros(1,7) code]; % Number of Present Code, Number of try, Time
hopping codes c
    str = []; ts  = [0 0];

    NEXT_PULSE_TIME=-1;
    NEXT_HOPPING_ON=0;
    INTEGRATION_START_TIME=-1;
    ACQ_FRAME_TIME=-1;
    ACQ_REGISTER_STATE=[0 0 0 0 0];
   % Derivatives %
  case 1,
   sys=[];

  % Update %
   case 2,
         if UWB_SIG_DETECT_ON==1
        UWB_SIG_DETECT_ON=2;
        x(2)=1;
        x(1)=1;
        sys=x;
        presentC=x(10+0); nextC=x(10+1);
        NEXT_PULSE_TIME=UWB_SIG_DETECT_TIME+(nextC-
presentC)*Tc+Tf;
        INTEGRATION_START_TIME=NEXT_PULSE_TIME-floor(2*tau/Ts)*Ts;
      end

     time2Change=NEXT_PULSE_TIME+floor(2*tau/Ts)*Ts+Ts;
     if UWB_SIG_DETECT_ON>1 & abs(t/time2Change-1) < 1e-8
        if ACQUISITION_ON==0 & NEXT_HOPPING_ON==1
           NEXT_HOPPING_ON=0;
           x(2)=x(2)+1;
           frameStartTime=UWB_SIG_DETECT_TIME-x(10+x(2)-1)*Tc-
floor(2*tau/Ts)*Ts;
           numFrame=floor((t-frameStartTime)/Tf);
           netNumFrame=mod(numFrame,Np);
           x(1)=mod(x(2)-1+netNumFrame,Np);

nextPulseTime=frameStartTime+numFrame*Tf+x(10+x(1))*Tc+floor(2*tau/Ts)*Ts;
           if nextPulseTime<t+floor(2*tau/Ts)*Ts
              x(1)=mod(x(1)+1,Np);

nextPulseTime=frameStartTime+(numFrame+1)*Tf+x(10+x(1))*Tc+floor(2*tau/Ts)*
Ts;

           end
           NEXT_PULSE_TIME=nextPulseTime;
           INTEGRATION_START_TIME=nextPulseTime-floor(2*tau/Ts)*Ts;
        else
           presentC=x(10+x(1));
```

```
            x(1)=mod(x(1)+1,Np);
            nextC=x(10+x(1));
            NEXT_PULSE_TIME=NEXT_PULSE_TIME+(nextC-presentC)*Tc+Tf;
            if ACQUISITION_ON==1
               ACQUISITION_ON=2;
               ACQ_FRAME_TIME=NEXT_PULSE_TIME-nextC*Tc-
floor(2*tau/Ts)*Ts;
               ACQ_REGISTER_STATE=findState(mod(x(1)-1,Np),xini,bitsPerHop);
               if ACQ_FRAME_TIME < t
                  disp('error');
                  ACQ_FRAME_TIME=ACQ_FRAME_TIME+Tf;

ACQ_REGISTER_STATE=findState(mod(x(1)+1,Np),xini,bitsPerHop);
               end
               fprintf('Acquired Frame Time=%5.3f ns\n',ACQ_FRAME_TIME*1e9);
            end
            if ACQUISITION_ON==2
               ACQUISITION_ON=3;
               INTEGRATION_START_TIME=NEXT_PULSE_TIME-
floor(2*tau/Ts)*Ts;
            end
         end
         sys=x;
      else
         sys=x;
      end
   % Outputs %
   case 3,
      if UWB_SIG_DETECT_ON>1
         pulseTimeBegin=NEXT_PULSE_TIME-floor(2*tau/Ts)*Ts;
         pulseTimeEnd=NEXT_PULSE_TIME+floor(2*tau/Ts)*Ts;
         if (t>pulseTimeBegin & t<pulseTimeEnd)|...
            abs(t/pulseTimeBegin-1) < 1e-8 | abs(t/pulseTimeEnd-1) < 1e-8
            sys = scholtzValue(t-NEXT_PULSE_TIME, 1, 0, tau);
         else
            sys=0;
         end
      else
         sys=0;
      end
   % GetTimeOfNextVarHit %
   case 4,
      sys=[];
   % Terminate %
   case 9,
      sys=[];
   % Unexpected flags %
   otherwise
     error(['Unhandled flag = ',num2str(flag)]);
```

7- S-Function of Integrate and Sample Block

```
function [sys,x0,str,ts] = IntegrateSample(t,x,u,flag,SampleTime)

switch flag,
  % Initialization %
  case 0,
    sizes = simsizes;

    sizes.NumContStates  = 0;
    sizes.NumDiscStates  = 3;
    sizes.NumOutputs     = 1;
    sizes.NumInputs      = 1;
    sizes.DirFeedthrough = 0;
    sizes.NumSampleTimes = 1;   % at least one sample time is needed
     sys = simsizes(sizes);
    x0  = [0 0 0]; % Output, Subtotal, Present time
    str = []; ts  = [0 0];
  % Derivatives %
  case 1,
    sys=[];
  % Update %
  case 2,
      sys=x;
      if abs(round(t/SampleTime) - t/SampleTime) < 1e-8
         sys(1)=x(2); sys(2)=0;
      else
         sys(2)=x(2)+abs(t-x(3))*u;
      end
      sys(3)=t;
  % Outputs %
  case 3,
          sys = x(1)<0;
  % GetTimeOfNextVarHit %
  case 4,
    sys=[];
  % Terminate %
  case 9,
     sys=[];
  % Unexpected flags %
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);
end
```

8- S-Function of UWB-Detection Block

```
function [sys,x0,str,ts] = uwbDetection(t,x,u,flag,threshold,tau,Ts,Ep)
global UWB_SIG_DETECT_TIME;
global UWB_SIG_DETECT_ON;
switch flag,
```

```
% Initialization %
case 0,
  sizes = simsizes;
  sizes.NumContStates  = 0;
  sizes.NumDiscStates  = floor(2*tau/Ts)+1;
  sizes.NumOutputs     = 0;
  sizes.NumInputs      = 1;
  sizes.DirFeedthrough = 0;
  sizes.NumSampleTimes = 1;   % at least one sample time is needed
  sys = simsizes(sizes);
  x0  = zeros(floor(2*tau/Ts)+1,1); % State of the state register and initial c0, m=5
  str = []; ts  = [0 0];
  UWB_SIG_DETECT_TIME=0;
  UWB_SIG_DETECT_ON=0;
% Derivatives %
case 1,
  sys=[];
% Update %
case 2,
%    coefficient=sum(x.*scholtzValue([-tau:Ts:tau]',1,0,tau))*Ts/sqrt(Ep)*100;
%                                fprintf('Time=%3.3f      ns,      Coefficient=%3.3f\n',(t-
(floor(tau/Ts)+1)*Ts)*1e9,coefficient);
    if UWB_SIG_DETECT_ON==0
       coefficient=sum(x.*scholtzValue([-tau:Ts:tau]',1,0,tau))*Ts/sqrt(Ep)*100;
       if(  coefficient>threshold )
          UWB_SIG_DETECT_ON=1;                UWB_SIG_DETECT_TIME=t-
(floor(tau/Ts)+1)*Ts;
          fprintf('UWB signal Detected. Time=%3.3f ns, Coefficient=%3.3f\n',(t-
(floor(tau/Ts)+1)*Ts)*1e9,coefficient);
       end
    end
        l=length(x);
    x(1:l-1)=x(2:l);x(1)=u;
    sys=x;
% Outputs %
case 3,
  sys=[];
   % GetTimeOfNextVarHit %
case 4,
   sys=[];
% Terminate %
case 9,
  sys=[];
% Unexpected flags %
otherwise
  error(['Unhandled flag = ',num2str(flag)]);
end
```

APPENDIX B

MiXiM Source Code: Omnet.ini:

```
[General]
network = ieee802154a
debug-on-errors = false
fname-append-host = false
record-eventlog = true
**.module-eventlog-recording = false
**.vector-recording = true
#**.debug = false
num-rngs = 88
ned-path = ../../base;../../modules;../../examples;
# tkenv-default-run=1
# cmdenv-runs-to-execute=1
cmdenv-express-mode = true
# Because of the battery module, there are always scheduled events
# thus we need a fallback sim-time-limit.
# Otherwise the battery module will eventually lead to a simtime_t
overflow
sim-time-limit=1 d # 30s for 10000 packets
ieee802154a.**.coreDebug = false
ieee802154a.playgroundSizeX = 500 m
ieee802154a.playgroundSizeY = 500 m
ieee802154a.playgroundSizeZ = 500 m
ieee802154a.world.useTorus = false
ieee802154a.world.use2D = false
ieee802154a.channelControl.sendDirect = false
ieee802154a.channelControl.pMax = 1000 mW
ieee802154a.channelControl.sat = -100 dBm
ieee802154a.channelControl.alpha = 2.0
ieee802154a.channelControl.carrierFrequency = 4500e+6 Hz
ieee802154a.node[*].nic.phy.usePropagationDelay = false
ieee802154a.node[*].nic.phy.thermalNoise = 0 dBm
ieee802154a.node[*].nic.phy.useThermalNoise = false # we use our own
thermal noise model
ieee802154a.node[*].nic.phy.timeRXToTX = 0.00021 s
ieee802154a.node[*].nic.phy.timeRXToSleep = 0.000031 s
ieee802154a.node[*].nic.phy.timeTXToRX = 0.00012 s
ieee802154a.node[*].nic.phy.timeTXToSleep = 0.000032 s
ieee802154a.node[*].nic.phy.timeSleepToRX = 0.000103 s
ieee802154a.node[*].nic.phy.timeSleepToTX = 0.000203 s
ieee802154a.node[*].nic.phy.maxTXPower = 1 mW # not used currently
but required by BasePhyLayer
ieee802154a.node[*].nic.phy.sensitivity = 0.1 dBm # useless but
required by BasePhyLayer
```

```
ieee802154a.node[*].nic.phy.initialRadioState = 0
***.debug = false
**.battery.nominal = 99999mAh
**.battery.capacity = 99999mAh
**.battery.voltage = 3.3V
**.battery.resolution = 10s
**.battery.publishDelta = 0.1
**.battery.publishTime = 0
**.battery.numDevices = 1 # only the PHY module uses energy from
the battery
**.batteryStats.debug = false
**.batteryStats.detail = false
**.batteryStats.timeSeries = false
ieee802154a.node[*].nic.connectionManagerName = "channelControl"
ieee802154a.node[*].nic.mac.headerLength = 16 bit
ieee802154a.node[*].nic.mac.maxRetries = 1
ieee802154a.node[0].nic.mac.stats = true
ieee802154a.node[*].nic.mac.stats = false
ieee802154a.node[*].nic.mac.trace = false
ieee802154a.node[*].nic.mac.debug = false
ieee802154a.node[*].app.trafficParam = 1
ieee802154a.node[0].app.nodeAddr = 0
ieee802154a.node[1].app.nodeAddr = 1
ieee802154a.node[*].app.debug = false
ieee802154a.node[*].app.flood = false
ieee802154a.node[*].app.stats = false
ieee802154a.node[*].app.trace = false
ieee802154a.node[*].app.payloadSize = 8 byte
## Nodes positions
ieee802154a.node[0].mobility.x = 0
ieee802154a.node[0].mobility.y = 0
ieee802154a.node[0].mobility.z = 0
ieee802154a.node[1].mobility.y = 0
ieee802154a.node[1].mobility.z = 0
[Config BERDistance]
description = "Evaluates the bit error rate as a function of
distance with various channel models."
ieee802154a.node[1].app.payloadSize = 8 byte
ieee802154a.node[*].app.dstAddr = 0
**.numHosts = 2
ieee802154a.node[1].mobility.x = ${distance=1..10 step 2, 11..100
step 5}
ieee802154a.node[0].app.nbPackets = 0
ieee802154a.node[1].app.nbPackets = ${nbPackets=150}
ieee802154a.node[*].nic.phy.analogueModels =
xmldoc("channels/${Channel=ghassemzadeh-los}.xml")
ieee802154a.node[*].nic.phy.decider =
xmldoc("receivers/${Receiver=3dB}.xml")
```

MiXiMSourceCode:IEEE802154A.cc

```
/*-*-mode:c++ -*-
**************************************************************
*file:IEEE802154A.cc
*
*author:JeromeRousselot<jerome.rousselot@csem.ch>
*
*copyright: (C) 2008 Centre Suisse d'Electronique et
Microtechnique (CSEM) SA
* Systems Engineering
* Real-Time Software and Networking
* Jaquet-Droz 1, CH-2002 Neuchatel, Switzerland.
*
* This program is free software; you can redistribute
it
* and/or modify it under the terms of the GNU General
Public
* License as published by the Free Software
Foundation; either
* version 2 of the License, or (at your option) any
later
* version.
* For further information see file COPYING
* in the top level directory
* description: this class holds constants specified in IEEE
802.15.4A UWB-IR Phy
* acknowledgment: this work was supported (in part) by the National
Competence
* Center in Research on Mobile Information and
Communication Systems
* NCCR-MICS, a center supported by the Swiss
National Science
* Foundation under grant number 5005-67322.
**************************************************************
*******/
#include "IEEE802154A.h"
#include <cassert>
// bit rate (850 kbps)
const int IEEE802154A::mandatory_bitrate = 850000;
// mandatory data symbol length (1.025 ms)
const simtime_t IEEE802154A::mandatory_symbol = 0.00000102364;
//102564
// 0.5 * mandatory_symbol (0.5 ms)
const simtime_t IEEE802154A::mandatory_timeShift = 0.00000051282;
// mandatory pulse duration ( = 1 / bandwidth = 2 ns)
const simtime_t IEEE802154A::mandatory_pulse = 0.000000002003203125;
// burst duration (32 ns)
const simtime_t IEEE802154A::mandatory_burst = 0.00000003205;
// number of consecutive pulses forming a burst
```

```
const int IEEE802154A::mandatory_pulses_per_burst = 16;
// Center frequency of band 3 in UWB lower band (1081.6 MHz wide
channel)
const double IEEE802154A::mandatory_centerFreq = 6489.6; // MHz
// default sync preamble length
const_simtime_t IEEE802154A::mandatory_preambleLength = 0.0000715;
// Tpre=71.5 µs
// Normalized pulse envelope peak
const double IEEE802154A::maxPulse = 1;
// Duration of a sync preamble symbol
const_simtime_t IEEE802154A::Tpsym = 0.001 * 0.001 * 0.001 * 993.6;
// 993.6 ns
const_simtime_t IEEE802154A::tFirstSyncPulseMax =
IEEE802154A::mandatory_pulse/2;
// Ci values
const short IEEE802154A::C31[8][31] = {
// C1
{ -1, 0, 0, 0, 0, +1, 0, -1, 0, +1, +1, +1, 0, +1, -1,
0, 0, 0, +1, -1, +1, +1, +1,
0, 0, -1, +1, 0, -1, 0, 0 },
// C2
{ 0, +1, 0, +1, -1, 0, +1, 0, +1, 0, 0, 0, -1, +1, +1,
0, -1, +1, -1, -1, -1, 0, 0,
+1, 0, 0, +1, +1, 0, 0, 0 },
// C3
{ -1, +1, 0, +1, +1, 0, 0, 0, -1, +1, -1, +1, +1, 0, 0,
+1, +1, 0, +1, 0, 0, -1, 0,
0, 0, 0, -1, 0, +1, 0, -1 },
// C4
{ 0, 0, 0, 0, +1, -1, 0, 0, -1, 0, 0, -1, +1, +1, +1,
+1, 0, +1, -1, +1, 0, 0, 0,
+1, 0, -1, 0, +1, +1, 0, -1 },
// C5
{ -1, 0, +1, -1, 0, 0, +1, +1, +1, -1, +1, 0, 0, 0, -1,
+1, 0, +1, +1,
+1, 0, -1, 0, +1, 0, 0, 0, 0, -1, 0, 0 },
// C6
{ +1, +1, 0, 0, +1, 0, 0, -1, -1, -1, +1, -1, 0, +1, +1,
-1, 0, 0, 0,
+1, 0, +1, 0, -1, +1, 0, +1, 0, 0, 0, 0 },
// C7
{ +1, 0, 0, 0, 0, +1, -1, 0, +1, 0, +1, 0, 0, +1, 0, 0,
0, +1, 0, +1, +1, -1, -1,
-1, 0, -1, +1, 0, 0, -1, +1 },
// C8
{ 0, +1, 0, 0, -1, 0, -1, 0, +1, +1, 0, 0, 0, 0, -1, -1,
+1, 0, 0, -1, +1, 0, +1,
+1, -1, +1, +1, 0, +1, 0, 0 },
};
```

```
const short IEEE802154A::shortSFD[8] = { 0, 1, 0, -1, 1, 0, 0, -1 };
short IEEE802154A::s_array[maxS] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1,
0, 0, 0 };
int IEEE802154A::last_s = 15;
double IEEE802154A::signalStart = 0;
int IEEE802154A::psduLength = 0;
//const_simtime_t IEEE802154A::MaxFrameDuration =
IEEE802154A::MaxPSDULength*IEEE802154A::mandatory_symbol +
IEEE802154A::mandatory_preambleLength;
const IEEE802154A::config IEEE802154A::cfg_mandatory_16M = {
3, // channel
NOMINAL_16_M, // PRF
NON_RANGING, // Frame type
PSR_DEFAULT, // preamble length (number of
preamble symbols)
31, // Spreading code length
16, // spreading delta L
16, // chips per burst
850000, // bit rate (bps)
16, // pulses per data burst
993.6E-9, // preamble symbol duration
1025.64E-9, // data symbol duration
512.82E-9, // burst time shift duration
2.003E-9, // pulse duration (chip)
32.05E-9, // burst duration (pulses per
data burst * chip)
71.5E-6, // synchronization preamble
duration
6489.6 // center frequency
};
const IEEE802154A::config IEEE802154A::cfg_mandatory_4M = {
3, // channel
NOMINAL_4_M, // PRF
NON_RANGING, // Frame type
PSR_DEFAULT, // preamble length (number of
preamble symbols)
31, // Spreading code length
64, // spreading delta L
4, // chips per burst
850000, // bit rate (bps)
4, // pulses per data burst
3974.36E-9, // preamble symbol duration
1023.64E-9, // data symbol duration
512.82E-9, // burst time shift duration
2.003E-9, // pulse duration (chip)
8.01E-9, // burst duration (pulses per
data burst * chip)
286.2E-6, // synchronization preamble
```

```
duration
6489.6 // center frequency
};
IEEE802154A::config IEEE802154A::cfg =
IEEE802154A::cfg_mandatory_16M;
void IEEE802154A::setConfig(config newCfg) {
cfg = newCfg;
}
void IEEE802154A::setPSDULength(int _psduLength) {
//assert(_psduLength < IEEE802154A::MaxPSDULength+1);
IEEE802154A::psduLength = _psduLength;
}
simtime_t IEEE802154A::getMaxFrameDuration() {
return IEEE802154A::MaxPSDULength*cfg.data_symbol_duration +
cfg.preambleLength;
}
IEEE802154A::signalAndData
IEEE802154A::generateIEEE802154AUWBSignal(
simtime_t signalStart, bool allZeros) {
// 48 R-S parity bits, the 2 symbols phy header is not modeled
as it includes its own parity bits
// and is thus very robust
unsigned int nbBits = IEEE802154A::psduLength * 8 + 48;
IEEE802154A::signalStart = signalStart.dbl(); // use the
signalStart time value as a global offset for all Mapping values
simtime_t signalLength = cfg.preambleLength;
signalLength += static_cast<double> (nbBits) *
cfg.data_symbol_duration;
Signal* s = new Signal(signalStart, signalLength);
vector<bool>* bitValues = new vector<bool> ();
signalAndData res;
int bitValue;
// data start time relative to signal->getSignalStart();
simtime_t dataStart = cfg.preambleLength; // = Tsync + Tsfd
TimeMapping<Linear>* mapping = new TimeMapping<Linear> ();
Argument* arg = new Argument();
setBitRate(s);
generateSyncPreamble(mapping, arg);
generateSFD(mapping, arg);
//generatePhyHeader(mapping, arg);
// generate bit values and modulates them according to
// the IEEE 802.15.4A specification
simtime_t symbolStart = dataStart;
simtime_t burstPos;
for (unsigned int burst = 0; burst < nbBits; burst++) {
if(allZeros) {
bitValue = 0;
} else {
bitValue = intuniform(0, 1, 0);
```

```
}
bitValues->push_back(static_cast<bool>(bitValue));
burstPos = symbolStart + bitValue*cfg.shift_duration +
getHoppingPos(burst)*cfg.burst_duration;
generateBurst(mapping, arg, burstPos, +1);
symbolStart = symbolStart + cfg.data_symbol_duration;
}
//assert(uwbirMacPkt->getBitValuesArraySize() == dataLength);
//assert(bitValues->size() == nbBits);
// associate generated pulse energies to the signal
s->setTransmissionPower(mapping);
delete arg;
res.first = s;
res.second = bitValues;
return res;
}
void IEEE802154A::generateSyncPreamble(Mapping* mapping, Argument*
arg) {
// NSync repetitions of the Si symbol
for (short n = 0; n < cfg.NSync; n = n + 1) {
for (short pos = 0; pos < cfg.CLength; pos = pos + 1) {
if (C31[Ci - 1][pos] != 0) {
if(n==0 && pos==0) {
// we slide the first pulse slightly
in time to get the first point "inside" the signal
arg->setTime(1E-12 + n *
cfg.sync_symbol_duration + pos * cfg.spreadingdL *
cfg.pulse_duration);
} else {
arg->setTime(n * cfg.sync_symbol_duration
+ pos * cfg.spreadingdL * cfg.pulse_duration);
}
//generatePulse(mapping, arg, C31[Ci -
1][pos],
// IEEE802154A::maxPulse,
IEEE802154A::mandatory_pulse);
generatePulse(mapping, arg, 1,
// always positive polarity
IEEE802154A::maxPulse,
cfg.pulse_duration);
}
}
}
}
void IEEE802154A::generateSFD(Mapping* mapping, Argument* arg) {
double sfdStart = NSync * Tpsym;
for (short n = 0; n < 8; n = n + 1) {
if (IEEE802154A::shortSFD[n] != 0) {
for (short pos = 0; pos < cfg.CLength; pos = pos +
```

```
1) {
if (C31[Ci - 1][pos] != 0) {
arg->setTime(sfdStart +
n*cfg.sync_symbol_duration +
pos*cfg.spreadingdL*cfg.pulse_duration);
//generatePulse(mapping, arg, C31[Ci -
1][pos] * shortSFD[n]); // change pulse polarity
generatePulse(mapping, arg, 1); //
always positive polarity
}
}
}
}
}
void IEEE802154A::generatePhyHeader(Mapping* mapping, Argument* arg)
{
// not implemented
}
void IEEE802154A::generatePulse(Mapping* mapping, Argument* arg,
short polarity, double peak, simtime_t chip) {
assert(polarity == -1 || polarity == +1);
arg->setTime(arg->getTime() + IEEE802154A::signalStart); //
adjust argument so that we use absolute time values in Mapping
mapping->setValue(*arg, 0);
arg->setTime(arg->getTime() + chip / 2);
// Maximum point at symbol half (triangular pulse)
mapping->setValue(*arg, peak * polarity);
arg->setTime(arg->getTime() + chip / 2);
mapping->setValue(*arg, 0);
}
void IEEE802154A::generateBurst(Mapping* mapping, Argument* arg,
simtime_t burstStart, short polarity) {
assert(burstStart <
cfg.preambleLength+(psduLength*8+48+2)*cfg.data_symbol_duration);
// 1. Start point = zeros
simtime_t offset = burstStart;
for (int pulse = 0; pulse < cfg.nbPulsesPerBurst; pulse++) {
arg->setTime(offset);
generatePulse(mapping, arg, 1);
offset = offset + cfg.pulse_duration;
}
}
void IEEE802154A::setBitRate(Signal* s) {
 Argument arg = Argument();
// set a constant value for bitrate
TimeMapping<Linear>* bitrate = new TimeMapping<Linear> ();
arg.setTime(IEEE802154A::signalStart); // absolute time
(required for compatibility with MiXiM base RSAM code)
bitrate->setValue(arg, cfg.bitrate);
```

```
arg.setTime(s->getSignalLength());
bitrate->setValue(arg, cfg.bitrate);
s->setBitrate(bitrate);
}
simtime_t IEEE802154A::getThdr() {
switch (cfg.channel) {
default:
switch (cfg.prf) {
case NOMINAL_4_M:
//error("This optional mode is not implemented.");
return 0;
break;
case NOMINAL_16_M:
return 16.4E-6;
case NOMINAL_64_M:
return 16.8E-6;
case PRF_OFF:
return 0;
}
}
return 0;
}
int IEEE802154A::s(int n) {
assert(n < maxS);
for (; last_s < n; last_s = last_s + 1) {
// compute missing values as necessary
s_array[last_s] = (s_array[last_s - 14] + s_array[last_s
- 15]) % 2;
}
assert(s_array[n] == 0 || s_array[n] == 1);
return s_array[n];
}
int IEEE802154A::getHoppingPos(int sym) {
//int m = 3; // or 5 with 4M
int pos = 0;
int kNcpb = 0;
switch(cfg.prf) {
case NOMINAL_4_M:
kNcpb = sym * cfg.Ncpb;
pos = s(kNcpb) + 2*s(1+kNcpb) + 4*s(2+kNcpb) +
8*s(3+kNcpb) + 16*s(4+kNcpb);
break;
case NOMINAL_16_M:
pos = s(kNcpb) + 2*s(1+kNcpb) + 4*s(2+kNcpb);
break;
case NOMINAL_64_M:
case PRF_OFF:
default:
assert(0==1); // unimplemented or invalid PRF value
```

```
}
// assert(pos > -1 && pos < 8); // TODO: update to reflect
number of hopping pos for current config
return pos;
}
simtime_t IEEE802154A::getPhyMaxFrameDuration() {
simtime_t phyMaxFrameDuration = 0;
simtime_t TSHR, TPHR, TPSDU, TCCApreamble;
TSHR = IEEE802154A::getThdr();
TPHR = IEEE802154A::getThdr();
phyMaxFrameDuration = phyMaxFrameDuration + TSHR;
return phyMaxFrameDuration;
}
```

APPENDIX C

PUBLICATIONS

Sanam, E., Seman, K., & Jali, M. Z. (2017). Tuning Physical Parameters For Enhanced Receiver Performance Of Pulsed-Based Ultra-Wideband In Short Range Wireless Communications. Journal of Theoretical & Applied Information Technology, 95(11).

Sanam, E., Seman, K., Jawad, M., Hussain, A. S. T., & Jali, M. Z. (2015). Impulse-Based UWB for Next Generation Secure and Tunable Short-Range Wireless Infrastructures. Paper presented at the Applied Mechanics and Materials.

Sanam, E., Seman, K., & Jali, M. Z. (2018). Tuning the Physical Parameters of Most Influence Effect for Enhanced Receiver Performance of Pulsed-Based Ultra-Wideband in Short-Range Wireless Solutions. International Journal of Technology and Engineering Studies, 4(6), 219-226.

Abdala, T. M., Daud, N., Sanam, E., Ahmed, M. S., Abdalla, A. A., & Aboghsesa, S. M. (2015). Performance tradeoffs of routing protocols in wireless sensor networks. Paper presented at the International Conference on Network Security and Computer Science (ICNSCS-15).