Appendix A: AES function library

/\*\*

\* @ngdoc function

\* @name aesApp.service:aes

\* @description

\* # aes

\* Store all of the AES algorithm functions

\*/

angular.module('aesApp')

.factory('aes', ['convert', function(convert) {

// Define substitution tables

var sbox = [

//0 1 2 3 4 5 6 7 8 9 A B C D E F

0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0

0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1

0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2

0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3

0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4

0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5

0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6

0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7

0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8

0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9

0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A

0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B

0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C

0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D

0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E

0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 ]; //F

var rsbox = [

0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,

0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,

0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,

0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,

0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,

0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,

0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,

0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,

0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,

0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,

0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,

0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,

0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,

0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,

0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,

0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d ];

var multiplicationMatrix = [

[0x02, 0x03, 0x01, 0x01],

[0x01, 0x02, 0x03, 0x01],

[0x01, 0x01, 0x02, 0x03],

[0x03, 0x01, 0x01, 0x02]

];

var inverseMultiplicationMatrix = [

[0x0E, 0x0B, 0x0D, 0x09],

[0x09, 0x0E, 0x0B, 0x0D],

[0x0D, 0x09, 0x0E, 0x0B],

[0x0B, 0x0D, 0x09, 0x0E]

];

var eTable = [

//0 1 2 3 4 5 6 7 8 9 A B C D E F

0x01, 0x03, 0x05, 0x0F, 0x11, 0x33, 0x55, 0xFF, 0x1A, 0x2E, 0x72, 0x96, 0xA1, 0xF8, 0x13, 0x35, //0

0x5F, 0xE1, 0x38, 0x48, 0xD8, 0x73, 0x95, 0xA4, 0xF7, 0x02, 0x06, 0x0A, 0x1E, 0x22, 0x66, 0xAA, //1

0xE5, 0x34, 0x5C, 0xE4, 0x37, 0x59, 0xEB, 0x26, 0x6A, 0xBE, 0xD9, 0x70, 0x90, 0xAB, 0xE6, 0x31, //2

0x53, 0xF5, 0x04, 0x0C, 0x14, 0x3C, 0x44, 0xCC, 0x4F, 0xD1, 0x68, 0xB8, 0xD3, 0x6E, 0xB2, 0xCD, //3

0x4C, 0xD4, 0x67, 0xA9, 0xE0, 0x3B, 0x4D, 0xD7, 0x62, 0xA6, 0xF1, 0x08, 0x18, 0x28, 0x78, 0x88, //4

0x83, 0x9E, 0xB9, 0xD0, 0x6B, 0xBD, 0xDC, 0x7F, 0x81, 0x98, 0xB3, 0xCE, 0x49, 0xDB, 0x76, 0x9A, //5

0xB5, 0xC4, 0x57, 0xF9, 0x10, 0x30, 0x50, 0xF0, 0x0B, 0x1D, 0x27, 0x69, 0xBB, 0xD6, 0x61, 0xA3, //6

0xFE, 0x19, 0x2B, 0x7D, 0x87, 0x92, 0xAD, 0xEC, 0x2F, 0x71, 0x93, 0xAE, 0xE9, 0x20, 0x60, 0xA0, //7

0xFB, 0x16, 0x3A, 0x4E, 0xD2, 0x6D, 0xB7, 0xC2, 0x5D, 0xE7, 0x32, 0x56, 0xFA, 0x15, 0x3F, 0x41, //8

0xC3, 0x5E, 0xE2, 0x3D, 0x47, 0xC9, 0x40, 0xC0, 0x5B, 0xED, 0x2C, 0x74, 0x9C, 0xBF, 0xDA, 0x75, //9

0x9F, 0xBA, 0xD5, 0x64, 0xAC, 0xEF, 0x2A, 0x7E, 0x82, 0x9D, 0xBC, 0xDF, 0x7A, 0x8E, 0x89, 0x80, //A

0x9B, 0xB6, 0xC1, 0x58, 0xE8, 0x23, 0x65, 0xAF, 0xEA, 0x25, 0x6F, 0xB1, 0xC8, 0x43, 0xC5, 0x54, //B

0xFC, 0x1F, 0x21, 0x63, 0xA5, 0xF4, 0x07, 0x09, 0x1B, 0x2D, 0x77, 0x99, 0xB0, 0xCB, 0x46, 0xCA, //C

0x45, 0xCF, 0x4A, 0xDE, 0x79, 0x8B, 0x86, 0x91, 0xA8, 0xE3, 0x3E, 0x42, 0xC6, 0x51, 0xF3, 0x0E, //D

0x12, 0x36, 0x5A, 0xEE, 0x29, 0x7B, 0x8D, 0x8C, 0x8F, 0x8A, 0x85, 0x94, 0xA7, 0xF2, 0x0D, 0x17, //E

0x39, 0x4B, 0xDD, 0x7C, 0x84, 0x97, 0xA2, 0xFD, 0x1C, 0x24, 0x6C, 0xB4, 0xC7, 0x52, 0xF6, 0x01 //F

];

var lTable = [

//0 1 2 3 4 5 6 7 8 9 A B C D E F

0x00, 0x00, 0x19, 0x01, 0x32, 0x02, 0x1A, 0xC6, 0x4B, 0xC7, 0x1B, 0x68, 0x33, 0xEE, 0xDF, 0x03, //0

0x64, 0x04, 0xE0, 0x0E, 0x34, 0x8D, 0x81, 0xEF, 0x4C, 0x71, 0x08, 0xC8, 0xF8, 0x69, 0x1C, 0xC1, //1

0x7D, 0xC2, 0x1D, 0xB5, 0xF9, 0xB9, 0x27, 0x6A, 0x4D, 0xE4, 0xA6, 0x72, 0x9A, 0xC9, 0x09, 0x78, //2

0x65, 0x2F, 0x8A, 0x05, 0x21, 0x0F, 0xE1, 0x24, 0x12, 0xF0, 0x82, 0x45, 0x35, 0x93, 0xDA, 0x8E, //3

0x96, 0x8F, 0xDB, 0xBD, 0x36, 0xD0, 0xCE, 0x94, 0x13, 0x5C, 0xD2, 0xF1, 0x40, 0x46, 0x83, 0x38, //4

0x66, 0xDD, 0xFD, 0x30, 0xBF, 0x06, 0x8B, 0x62, 0xB3, 0x25, 0xE2, 0x98, 0x22, 0x88, 0x91, 0x10, //5

0x7E, 0x6E, 0x48, 0xC3, 0xA3, 0xB6, 0x1E, 0x42, 0x3A, 0x6B, 0x28, 0x54, 0xFA, 0x85, 0x3D, 0xBA, //6

0x2B, 0x79, 0x0A, 0x15, 0x9B, 0x9F, 0x5E, 0xCA, 0x4E, 0xD4, 0xAC, 0xE5, 0xF3, 0x73, 0xA7, 0x57, //7

0xAF, 0x58, 0xA8, 0x50, 0xF4, 0xEA, 0xD6, 0x74, 0x4F, 0xAE, 0xE9, 0xD5, 0xE7, 0xE6, 0xAD, 0xE8, //8

0x2C, 0xD7, 0x75, 0x7A, 0xEB, 0x16, 0x0B, 0xF5, 0x59, 0xCB, 0x5F, 0xB0, 0x9C, 0xA9, 0x51, 0xA0, //9

0x7F, 0x0C, 0xF6, 0x6F, 0x17, 0xC4, 0x49, 0xEC, 0xD8, 0x43, 0x1F, 0x2D, 0xA4, 0x76, 0x7B, 0xB7, //A

0xCC, 0xBB, 0x3E, 0x5A, 0xFB, 0x60, 0xB1, 0x86, 0x3B, 0x52, 0xA1, 0x6C, 0xAA, 0x55, 0x29, 0x9D, //B

0x97, 0xB2, 0x87, 0x90, 0x61, 0xBE, 0xDC, 0xFC, 0xBC, 0x95, 0xCF, 0xCD, 0x37, 0x3F, 0x5B, 0xD1, //C

0x53, 0x39, 0x84, 0x3C, 0x41, 0xA2, 0x6D, 0x47, 0x14, 0x2A, 0x9E, 0x5D, 0x56, 0xF2, 0xD3, 0xAB, //D

0x44, 0x11, 0x92, 0xD9, 0x23, 0x20, 0x2E, 0x89, 0xB4, 0x7C, 0xB8, 0x26, 0x77, 0x99, 0xE3, 0xA5, //E

0x67, 0x4A, 0xED, 0xDE, 0xC5, 0x31, 0xFE, 0x18, 0x0D, 0x63, 0x8C, 0x80, 0xC0, 0xF7, 0x70, 0x07 //F

];

// round key constant table

var rCon = [

[0x01, 0x00, 0x00, 0x00],

[0x02, 0x00, 0x00, 0x00],

[0x04, 0x00, 0x00, 0x00],

[0x08, 0x00, 0x00, 0x00],

[0x10, 0x00, 0x00, 0x00],

[0x20, 0x00, 0x00, 0x00],

[0x40, 0x00, 0x00, 0x00],

[0x80, 0x00, 0x00, 0x00],

[0x1b, 0x00, 0x00, 0x00],

[0x36, 0x00, 0x00, 0x00],

[0x6C, 0x00, 0x00, 0x00],

[0xD8, 0x00, 0x00, 0x00],

[0xAB, 0x00, 0x00, 0x00],

[0x4D, 0x00, 0x00, 0x00],

[0x9A, 0x00, 0x00, 0x00]

];

// private functions

var \_private = {

// arrayToState function

// convert a one-dimensional array to a state matrix

arrayToState : function(message) {

var row, col;

var state = [[],[],[],[]];

// iterate over state matrix by rows and then columns

for (col=0; col<4; col++) {

for (row=0; row<4; row++) {

state[row].push(message[(col\*4)+row]);

}

}

return state;

},

// stateToArray helper function

// transform 4x4 state matrix to one dimensional array

stateToArray : function(state) {

var row, col;

var t = [];

// iterate over state matrix by rows and then columns

for (col=0; col<4; col++) {

for (row=0; row<4; row++) {

t.push(state[row][col]);

}

}

return t;

},

// xor words function

// helper function for key expansion process

// xor two 4-byte words

xorWords : function(word1, word2) {

var t = []; //temp array to hold xor'ed result

for (var i=0; i<4; i++) {

/\*jslint bitwise: true \*/

t[i] = word1[i] ^ word2[i];

}

return t;

}

};

// public functions

var \_pub = {

// add round key function

addRoundKey : function(state, key) {

var row, col;

for (col=0; col<4; col++) {

for (row=0; row<4; row++) {

/\*jslint bitwise: true \*/

state[row][col] = state[row][col] ^ key[(col\*4)+row];

}

}

return state;

},

// decrypt function

// decrypt a message given a key

decrypt : function(message, key) {

var round, roundSize;

var state;

var keyLength = key.length;

var log = [];

if (keyLength === 16) { roundSize = 10;}

if (keyLength === 24) { roundSize = 12;}

if (keyLength === 32) { roundSize = 14;}

// generate an expanded key

var expKey = this.expandKey(key);

// create state and add initial round key before starting rounds

state = this.addRoundKey(\_private.arrayToState(message), this.getRoundKey(expKey,-1,true));

log.push({

round : 0,

description : 'Creating initial state',

state : convert.arrayToHexString(\_private.stateToArray(state))

});

// perform all four encryption steps in the rounds

for (round=0; round<roundSize-1; round++) {

state = this.shiftRows(state, true);

state = this.substitutionBox(state,true);

state = this.addRoundKey(state, this.getRoundKey(expKey,round,true));

state = this.mixState(state,true);

log.push({

round : round+1,

description : 'Decrypting',

state : convert.arrayToHexString(\_private.stateToArray(state))

});

}

// perform final round step without mixing columns

state = this.shiftRows(state, true);

state = this.substitutionBox(state,true);

state = this.addRoundKey(state, this.getRoundKey(expKey,round,true));

log.push({

round : round+1,

description : 'Final decryption result',

state : convert.arrayToHexString(\_private.stateToArray(state))

});

return {

plaintext : \_private.stateToArray(state),

key : key,

keySize : keyLength,

expandedKey : expKey,

log : log

};

},

// encrypt function

// this is the key driver function that encrypts a message based on a key

encrypt : function(message, key) {

var round, roundSize;

var state;

var keyLength = key.length;

var log = [];

//set number of encryption rounds based on key length

if (keyLength === 16) { roundSize = 10;}

if (keyLength === 24) { roundSize = 12;}

if (keyLength === 32) { roundSize = 14;}

// generate an expanded key

var expKey = this.expandKey(key);

// create state and add initial round key before starting rounds

state = this.addRoundKey(\_private.arrayToState(message), this.getRoundKey(expKey,-1,false));

log.push({

round : 0,

description : 'Creating initial state',

state : convert.arrayToHexString(\_private.stateToArray(state))

});

// perform all four encryption steps in the rounds

for (round=0; round<roundSize-1; round++) {

state = this.substitutionBox(state,false);

state = this.shiftRows(state, false);

state = this.mixState(state,false);

state = this.addRoundKey(state, this.getRoundKey(expKey,round,false));

log.push({

round : round+1,

description : 'Encrypting',

state : convert.arrayToHexString(\_private.stateToArray(state))

});

}

// perform final round step without mixing columns

state = this.substitutionBox(state,false);

state = this.shiftRows(state, false);

state = this.addRoundKey(state, this.getRoundKey(expKey,round,false));

log.push({

round : round+1,

description : 'Final encryption result',

state : convert.arrayToHexString(\_private.stateToArray(state))

});

return {

ciphertext : \_private.stateToArray(state),

key : key,

keySize : keyLength,

expandedKey : expKey,

log : log

};

},

// key expansion function

// expands a provided key based on the specified size

// size options: 16 (128bit), 24 (192bit), 32 (256bit)

expandKey : function(key) {

var maxRounds;

var expKey = []; //newly expanded key

var keyLength = key.length;

// each round adds a 4-byte word to the expanded key

if (keyLength === 16) { maxRounds = 44;}

if (keyLength === 24) { maxRounds = 52;}

if (keyLength === 32) { maxRounds = 60;}

for (var round=0; round < maxRounds; round++) {

// copy words from existing key to new key during initial rounds

if (round < keyLength/4) {

expKey = expKey.concat(this.keyOffset(key, round\*4));

} else if (keyLength === 32 && (round-12)%8 === 0) {

expKey = expKey.concat(\_private.xorWords( this.subWord(this.keyOffset(expKey, (round-1)\*4)), this.keyOffset(expKey, (round-(keyLength/4))\*4)));

} else if (round%(keyLength/4) === 0){

//perform complete set of steps every nth round

// this complex looking statement performs the following calculation:

// Sub Word(Rot Word(EK((round-1)\*4))) XOR Rcon((round/4)-1) XOR EK((round-4)\*4)

expKey = expKey.concat(\_private.xorWords( \_private.xorWords( this.subWord(this.rotWord(this.keyOffset(expKey, (round-1)\*4))), this.roundCon((round/(keyLength/4))-1)), this.keyOffset(expKey, (round-(keyLength/4))\*4)));

} else {

//simple XOR every other round

//EK((round-1)\*4)XOR EK((round-4)\*4)

expKey = expKey.concat( \_private.xorWords( this.keyOffset(expKey,(round-1)\*4), this.keyOffset(expKey,(round-(keyLength/4))\*4)));

}

}

return expKey;

},

// getRoundKey function

// return a 16-byte round key when provided an expanded key and round number

// round = -1 for the initial round

// bReverse parameer determines direction

getRoundKey : function(expKey, round, bReverse) {

var offset;

if (bReverse) {

offset = expKey.length - ((round+1) \* 16) - 16;

} else {

offset = (round+1) \* 16;

}

return expKey.slice(offset, offset+16);

},

// key offset function

// get 4-byte word from key based on offset

keyOffset : function(key, offset) {

return key.slice(offset, offset+4);

},

// mix columns in current state matrix

// bReverse parameter determines direction (true = decryption)

mixState : function(state, bReverse) {

var row, col;

var t = [];

// determine which matrix to use for the multiplication

var matrix = bReverse ? inverseMultiplicationMatrix : multiplicationMatrix;

// iterate over each column in state matrix

for (col=0; col<4; col++) {

// copy current column values to temporary array

for (row=0; row < 4; row++) {

t[row] = state[row][col];

}

// mix column

t = this.mixColumn(t, col, matrix);

// copy mixed column back to state matrix

for (row = 0; row < 4; row++) {

state[row][col] = t[row];

}

}

return state;

},

// mix column function

// bReverse parameter determines direction (true = decryption)

mixColumn : function(stateCol, colNum, matrix) {

var row, i, val;

var newCol = [], t = [];

// calculate value for each row in column

for (row=0; row < 4; row++) {

// iterate over each column value during calculation

for (i=0; i < 4; i++) {

if (stateCol[i] === 0) {

//if 0 there is no need to calculate

t[i] = 0x00;

} else {

val = lTable[stateCol[i]] + lTable[matrix[row][i]];

// need to keep val within two-digit hex bound

if (val > 0xFF) {

t[i] = eTable[ val - 0xFF];

} else {

t[i] = eTable[val];

}

}

}

/\*jslint bitwise: true \*/

newCol[row] = t[0] ^ t[1] ^ t[2] ^ t[3];

}

return newCol;

},

// parseKey function

// take an array as input and convert into properly sized key

// accept 128 bit (16 byte), 192 bit (24byte), and 256 bit (32byte) keys

// size param determines key size, pad with 0's if necessary

parseKey : function(key, size) {

var i, padding = 0;

var keyLength = key.length;

// truncate key if longer than specified size

if (keyLength > size) {

key = key.slice(0,size);

keyLength = size;

} else {

// calculate amount of padding

padding = size - keyLength;

// add padding if necessary

if (padding) {

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

key.push(0);

}

}

}

return {

key : key,

size : key.length,

padding : padding

};

},

// parseMessage function

// take an array as input and convert into properly sized 16-byte block

// type parameter accepts ascii/hex

parseMessage : function(message, type) {

var i, padding = 0;

var messageLength = message.length;

padding = 16 - messageLength;

if (padding) {

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

if (type === 'ascii') {

message.push(32); //ascii space

} else {

message.push(0);

}

}

}

return {

message : message,

size : messageLength,

padding : padding

};

},

// Rcon function

// get round key constant value

roundCon : function(val) {

return rCon[val];

},

// rotate word function

// shifts 4 bytes as part of the key expansion process

rotWord : function(word) {

var t = word[0];

for (var i=0; i<3; i++) {

word[i] = word[i+1];

}

word[3] = t;

return word;

},

// shift row function

// shift matrix columns in each row based on a shift amount

// bReverse parameter determines direction (true = decryption)

shiftRows : function(state, bReverse) {

var row, col;

var temp = []; // temp array to copy row values in matrix

var shift = 0; // counter to track shift amount

// loop through each row in matrix

// first row (index=0) is not shifted

for (row=1; row < 4; row++) {

shift++; // each successive row is shifted by one additional byte

for (col=0; col < 4; col++) {

temp[col] = state[row][col]; // copy values from current row to temp array

}

for (col=0; col < 4; col++) {

// rotate the column values based on the shift amount and direction

if (bReverse) {

state[row][col] = temp[col - shift < 0 ? col - shift + 4 : col - shift];

} else {

state[row][col] = temp[col + shift > 3 ? col + shift - 4 : col + shift];

}

}

}

return state;

},

// substitution box function

// replace state element with value in substitution box

// reverse param determines which sbox is used

substitutionBox : function(state, bReverse) {

var row, col;

var box = bReverse ? rsbox : sbox;

for (row=0; row<4; row++) {

for (col=0; col<4; col++) {

state[row][col] = box[state[row][col]];

}

}

return state;

},

// substitute word function

// perform sbox substitution on each word for key expansion

// bReverse param determines direction

subWord : function(word, bReverse) {

// determine which sbox to use

var box = bReverse ? rsbox : sbox;

for (var i=0; i<4; i++) {

word[i] = box[word[i]];

}

return word;

}

};

return \_pub;

}]);

Appendix B: Convert library functions

/\*\*

\* @ngdoc function

\* @name aesApp.service:convert

\* @description

\* # convert

\* conversion helper functions

\*/

angular.module('aesApp')

.factory('convert', function() {

// Service logic

// ...

var \_pub = {

// arrayToHex helper function

// converts an array of decimal numbers to an array of hex string values for display

arrayToHex : function(a) {

var t = [];

var aLength = a.length;

for (var i=0; i<aLength; i++) {

t[i] = a[i].toString(16);

}

return t;

},

// arrayToString helper function

// converts an array of decimal numbers to an ASCII string for display

arrayToString : function(a) {

var s = '';

var aLength = a.length;

for (var i=0; i<aLength; i++) {

s += String.fromCharCode(a[i]);

}

return s;

},

// arrayToHexString

// converts an array of numbers to a hex string for display

// bWithSpaces can be set to add a space between hex values

// code derived from crypto-js (https://code.google.com/archive/p/crypto-js/)

arrayToHexString: function (a, bWithSpaces) {

var s = [];

for (var i=0; i<a.length; i++) {

/\*jslint bitwise: true \*/

s.push((a[i] >>> 4).toString(16));

/\*jslint bitwise: true \*/

s.push((a[i] & 0xF).toString(16));

if (bWithSpaces) { s.push(' ');}

}

return s.join('').trim();

},

// hexToArray helper

// converts a string containing hex to a decimal array

// a string with non-hex characters returns an empty array

hexToArray : function(s) {

var stringLen;

var t = [];

//remove all whitespace from the string

s = s.replace(/\s+/g,'');

stringLen = s.length;

// process the string if it doesn't have any non-hex characters

if (/^[0-9a-fA-F]+$/.test(s)) {

for (var i=0; i < stringLen; i+=2) {

t.push(parseInt(s.slice(i,i+2),16));

}

}

return t;

},

// stringToArray helper function

// converts an ASCII string to an array of decimal values

stringToArray : function(s) {

var t = [];

for (var i=0; i< s.length; i++) {

t[i] = s.charCodeAt(i);

}

return t;

}

};

return \_pub;

});