GadgetStack(TM)

API Functions Reference V4

This document is intended to provide the following information:

- Overview of GadgetStack.
- Sending and Receiving Messages.
- C API Reference for GadgetStack.

Table of Contents

1.0 Overview of GadgetStack 3
2.0 Receiving and Sending Messages 7
3.0 709.1 Service Interface - C API Reference for GadgetStack 11
4.0 GadgetStack Utility Functions API 19
5.0 GadgetStack Required - User Defined Functions API 31

1.0 Overview of GadgetStack

The Application Program Interface (API) defines the functions and data structures that an application program uses to access the features of the GadgetStack's ANSI/EIA 709.1 implementation

GadgetStack was intended to maintain (where possible) backward compatibility with Echelon's(R) Neuron(R) C, and will continue to represent the Service Interface documented in section 11.3 of the ANSI/EIA 709.1 standard.

1.1 GadgetStack Files

The following files are typically included in the GadgetStack distribution:

api.h

Api.h is the application program interface include file. It contains the ANSI 709.1 address typedefs, 709.1 message structure typedefs, GadgetStack network variable type definitions and the GadgetStack API function prototypes.

lontalk.h

Lontalk.h contains ANSI 709.1 enumerated constants as well as source and destination address type definitions.

custom.h

Custom.h is the custom.c include file. It contains the data types and definitions that can be modified to build a custom 709.1 compliant node. It contains two #define statements specific to the Net40 development board:

1. LOAD_NVRAM_BINDING_DATA
2. STORE_NVRAM_BINDING_DATA

These values control how the custom node will store and retrieve binding and addressing information sent to this node by a network management tool. Custom.h also defines the customizable parameters, such as maximum data bytes per messages, size of receive message pool, NEURON_ID, application ID, self documentation string, etc.

custom.c

Custom.c contains custom node configuration data. It is used to define read-only-memory data like NeuronId and ProgramId, as well as default domain and address table information. This file should be modified to contain the six byte NeuronID supplied with the Gadget Node development kit. The NeuronID can be found on the bar-code label attached to the supplied modified Net40 development board back plate.

apppgm.c

Apppgm.c is an example 709.1 application program. It contains network variable definitions, message tag definitions as well as functions to handle network variable updates, online, offline and wink events. Some functions like AppInit, AppReset, DoApp and others are called by the GadgetStack scheduler and need to be defined in the apppgm.c file. Details as to exactly which functions must be defined are in the Adept GadgetStack API reference. Additional example applications are available in the Examples directory provided on the Adept Gadget Node Developer distribution. These example programs come in pairs, one apppgm.c file and one neuron C file. The apppgm.c file will run on the gadget node and communicate with a Neuron running the corresponding neuron C program. The examples demonstrate 709.1 communication features like network variables and explicit messages.

1.2 Implementation Choices

The GadgetStack has made the following implementation choices:

1. An application running in the GadgetStack operates like a “host application” node.
2. An application using the GadgetStack uses “host selection” for network variables.

1.3 Customization

The file `custom.h` contains various constant declarations that the user of the reference implementation must explicitly define (example: NEURON_ID, APP_ID) as well as those that may optionally be customized (such as buffer sizes).

1.4 Initialization

The user of the API must define the function: `void resetApp(void);`

This function is called at start-up time, and is also called if the node is reset. All application specific setup code should be placed in `resetApp()`.

1.5 Network Variables

Network variables are setup and initialized with structure declarations, as is done in the Echelon Host Application. See the apppgm.c for examples.

1.6 Operation and Scheduling

1.6.1 Application Processing

The reference implementation uses a simple round robin scheduling scheme. The user of the API must define the function:

```
void doApp(void);
```

The scheduler calls `doApp()` once per pass through the round robin schedule loop. The `doApp()` function has four responsibilities:

1. Process incoming messages, if any. Upon completion of processing a message, call msgFree().
2. Process incoming responses, if any. Upon completion of processing a response, call respFree().
3. Perform application specific processing.
4. Return to the round robin scheduler quickly. This is necessary to allow the protocol stack to process new messages. If the application needs to perform a large amount of processing, it must break the processing up into small pieces, and process one piece per call to `doApp()`.

Messages and responses must be processed in a timely function, preferably on each pass through `doApp()`. If application messages are not processed quickly enough, messages may be lost because application buffers are full.

The user application must not define the entry point "`main()`". The reference implementation scheduler defines this entry point.

1.6.2 Message Completion

The user of the API must define the function:

```
void msgCompletes(Stat stat, MsgTag tab);
```

The API calls this function when a message completes.

1.7 API Declarations

The following declarations define the API interface. These declarations are located in `api.h`:

```
extern MsgIn   msgIn;              // Incoming message data
extern MsgOut  msgOut;             // Outgoing message data
extern RespIn  respIn;             // Incoming response data
extern RespOut respOut;            // Outgoing response data

extern Boolean msgReceive;        // True if valid data is in msgIn
void           msgFree();          // "Frees" data in msgIn

Boolean msgAlloc();          // Is non-priority msgOut available
Boolean msgAllocPriority(); // Is priority msgOut available
void msgCancel();           // Cancels msgAlloc or msgAllocPriority
void msgSend();                                    // Sends msgOut

extern Boolean respReceive;       // True if valid data in respIn
void           respFree();        // "Frees" data in respIn

Boolean respAlloc();              // Is respOut available
void    respCancel();             // Cancels respAlloc
void    respSend();               // Sends respOut

void doApp(void);                                  // User defined
void resetApp(void);                               // User defined
void msgCompletes(MsgStat stat, MsgTag tag);    // User defined

typedef enum { MSG_FAILED, MSG_SUCCEEDED, MSG_COMPLETED } MsgStat;
```

2.0 Receiving and Sending Messages

2.1 Receiving Messages

```
extern BooleanmsgReceive;         // True if valid data is in msgIn
voidmsgFree();                      // "Frees" data in msgIn
extern MsgInmsgIn;                 // Incomming message data

typedef struct
{
   uint8         code;                          /* message code */
   uint8         len;                  /* length of message data */
   uint8         data[MAX_DATA_SIZE];           /* message data */
   Boolean       authenticated;        /* TRUE if authenticated */
   ServiceType   service;               /* Service used to send */
   RequestId     reqId;        /* Request ID to match responses */
   MsgIn         Addraddr;
} MsgIn;
```

When the protocol receives a message addressed to this node, the data from the message is written to the `msgIn` structure, and when `msgIn` is complete, the `msgReceive` variable is set to `TRUE`. If additional messages are received, they are buffered.

The application uses the data in `msgIn`, and when finished with the data calls `msgFree()`. The `msgFree` function sets `msgReceive` to FALSE.

When another message is received, or if a message is already buffered, the data is placed in `msgIn`, and `msgReceive` is again set to TRUE.

Note that `msgReceive()` does not allocate new system resources when executed. Inappropriate use of `msgFree()` will therefore not result in memory access violations or leaks.

2.2 Receiving Responses

```
extern Boolean respReceive;         // True if valid data in respIn
void   respFree();                   // "Frees" data in respIn
extern RespIn respIn;

typedef struct
{
   MsgTag       tag;                          /*   To match req */
   uint8        code;                         /*   message code */
   uint8        len;                          /* message length */
   uint8        data[MAX_DATA_SIZE];          /*   message data */
   RespInAddr   addr;       /* destination address (see above) */
} RespIn;                   /*  struct for receiving responses */
```

When the protocol receives a response to a request made by this node, the data from the response is written to the '`respIn`' structure, and when `respIn` is complete, the `respReceive` variable is set to TRUE. If additional responses are received, they are buffered.

The application uses the data in `respIn`, and then calls `respFree`(). The `respFree` function sets `respReceive` to FALSE. When another response is received, or if a response is already buffered, the data is placed in `respIn`, and `respReceive` is again set to TRUE.

2.3 Sending Messages

```
Boolean msgAlloc();
Boolean msgAllocPriority();
void msgCancel();          // Cancels msgAlloc or msgAllocPriority
void msgSend();             // Sends msgOut
extern MsgOut msgOut;
void msgCompletes(MsgStat stat, MsgTag tag);

typedef struct
{
   Boolean       priorityOn;        /* TRUE if priority message */
   MsgTag        tag;          /* to correlate completion codes */
   uint8         len;          /*  message length in data array */
   uint8         code;                          /* message code */
   uint8         data[MAX_DATA_SIZE];           /* message data */
   Boolean       authenticated;        /* TRUE if authenticated */
   ServiceType   service;     /* service type used to send msg  */
   MsgOutAddr           addr;/* destination address (see above)*/
} MsgOut;

typedef enum
{
   MSG_FAILED,
   MSG_SUCCEEDED,
   MSG_COMPLETED
} MsgStat;
```

Before sending a message, the application calls `msgAlloc()` or `msgAllocPriority()` for a priority message. If the allocation function returns `FALSE`, then an application output buffer is not available, and application must try again later.

When the allocation function returns TRUE, an application message out buffer is allocated for the current message. The application sets various fields in `msgOut` to construct the message.

When the message in `msgOut` is ready to send, the application calls `msgSend()`, and the message is sent.

If, for some reason, it is necessary to de-allocate a message buffer without sending a message, the application calls `msgCancel`().

When a message completes, the API calls the user defined function `msgCompletes()` to return the completion status of the message. The tag parameter is the value from the tag field of the `msgOut` structure.

2.4 Sending Responses

```
Boolean respAlloc();
void respCancel();              // Cancels respAlloc
void respSend();                // Sends respOut
extern RespOut respOut;

typedef struct
{
   RequestId     reqId;        /* Request ID to match responses */
   Boolean       nullResponse;      /* TRUE => no resp goes out */
   uint8         code;                        /*   message code */
   uint8         len;                         /* message length */
   uint8         data[MAX_DATA_SIZE];         /*   message data */
} RespOut;
```

When the application receives a message containing a request, the application returns a response.

Before sending a response, the application calls `respAlloc()`. If `respAlloc()` returns FALSE, then an application output buffer is not available, and application must try again later.

When `respAlloc()` returns TRUE, an application message out buffer is allocated for the current response. The application sets various fields in `respOut` to construct the response. When the response in `respOut` is ready to send, the application calls `respSend()`, and the response is sent.

If, for some reason, it is necessary to de-allocate a message buffer without sending a response, the application calls `respCancel()`.

3.0 709.1 Service Interface - C API Reference for GadgetStack

The GadgetStack API provides C prototypes to functions in two forms.

1. The Adept C prototype which Adept will continue to evolve as future features are incorporated into the GadgetStack and related ANSI/EIA 709.1 standards.
2. C prototypes which are intended to maintain backward compatibility with Echelon's(R) Neuron(R) C, and which represent the Service Interface documented in section 11.3 of the ANSI/EIA 709.1 standard.

In the prototypes documented below, the first prototype represents Adept's prototype and the second represents the backward compatible prototype. Not all functions represented in this document contain backward compatible prototypes. It should also be noted that at present the two prototypes are largely equivalent, however, changes to the Adept prototypes are already in the development schedule.

3.1 Boolean MsgAlloc(void);
Boolean msg_alloc(void);

Parameters:

```
NONE
```

Returns:

```
TRUE          If msg_out is available.
FALSE         If msg_out is not available for use.
```

This function checks to see if a non-priority msg_out data-structure is available for use. If a non-priority msg_out structure is available for sending, the function returns TRUE, otherwise it returns false. This is a non-blocking function, and returns immediately. This function also does not actually allocate memory, so can be called with out fear of exhausting system resources.

Example:

```
if( msg_alloc() )
{
  // construct and send intended msg
}
```

3.2 Boolean MsgAllocPriority(void); Boolean msg_alloc_priority(void);

Parameters:

```
NONE
```

Returns:

```
TRUE          If msg_out is available.
FALSE         If msg_out is not available for use.
```

This function checks to see if a priority msg_out data-structure is available for use. If a priority msg_out structure is available for sending, the function returns TRUE, otherwise it returns false. This is a non-blocking function, and returns immediately. This function also does not actually allocate memory, so can be called with out fear of exhausting system resources.

Example:

```
if( msg_alloc_priority() )
{
  // construct and send intended msg
}
```

3.3 void MsgSend(void); void msg_send(void);

Parameters:

```
NONE
```

Returns:

```
NONE
```

This function sends both non-priority and priority message to the network.

NOTE: Even if the associated msg_out structure is available, there is a very small chance the message did not make it onto the network. This only occurs if the message fails both the channel access algorithm and the type of message service selected (e.g. Unacknowledged, Repeat, etc.).

Example:

```
MsgTag tag0;

Status AppInit(void)
{
```

```
   tag0        = NewMsgTag(BINDABLE);

   return(SUCCESS);
}   /* End AppInit */

..........

if(msg_alloc_priority())
{
   msg_out.priority_on        = TRUE;
   msg_out.tag                = tag0;
   msg_out.code               = 1;
   msg_out.data[0]            = 5;    /* Some data */
   msg_out.len                = 1;
   msg_out.authenticated      = FALSE;
   msg_out.service            = ACKD;
   msg_out.addr.no_address    = SUBNET_NODE;
   memcpy(&msg_out.addr.snode,
          &snodeAddr, sizeof(SNodeAddrMode));
   msg_send();
}
```

3.4 void MsgCancel(void);
void msg_cancel(void);

Parameters:

```
NONE
```

Returns:

```
NONE
```

This function is used to cancel either a pending priority or non-priority message which has not already reached a critical point of sending.

Example:

```
MsgTag      tag1;
unsigned    prg_state;

Status AppInit(void)
{
   tag1        = NewMsgTag(BINDABLE);

   return(SUCCESS);
}   /* End AppInit */

..........
```

```
if(msg_alloc())
{
   msg_out.priority_on        = FALSE;
   msg_out.tag                = tag1;
   msg_out.code               = 1;
   msg_out.data[0]            = 5;    /* Some data */
   msg_out.len                = 1;
   msg_out.authenticated      = FALSE;
   msg_out.service            = ACKD;
   msg_out.addr.no_address    = SUBNET_NODE;
   memcpy(&msg_out.addr.snode,
          &snodeAddr, sizeof(SNodeAddrMode));

   if( prg_state = ACTIVE )
   {
      msg_send();
   }
   else
   {
      msg_cancel();
   }
}
```

3.5 void MsgFree(void); void msg_free(void);

Parameters:

NONE

Returns:

NONE

This function frees the msg_in structure for a incoming message.

Example:

```
if( msg_receive() )
{
   ............

   msg_free();
}
```

3.6 Boolean RespAlloc(void);
Boolean resp_alloc(void);

Parameters:

```
NONE
```

Returns:

```
TRUE          If resp_out is avaliable.
FALSE         If resp_out is not avaliable.
```

The resp_alloc function allocates a resp_out structure for an outgoing response. The resp_alloc function returns TRUE if the resp_out structure is available, otherwise, it returns FALSE. Like msg_alloc, this function is non-blocking and does not allocate system resources.

Example:

```
if( resp_alloc() )
{
   ..............

   resp_send();
}
```

3.7 void RespSend(void);
void resp_send(void);

Parameters:

```
NONE
```

Returns:

```
NONE
```

The resp_send function sends a message to the network using the resp_out data structure.

Example:

```
if( resp_alloc() )
{
   ..............

   resp_send();
}
```

3.8 void RespCancel(void);
void resp_cancel(void);

Parameters:

NONE

Returns:

NONE

This function is used to cancel a pending response message which has not already reached a critical point of sending. It reverses the effects of RespAlloc or resp_alloc.

Example:

```
/*-------------------------------------------------------*/
/* Process incoming messages. If we get a request,   */
/* respond with one more value. Ignore other msgs.   */
/*-------------------------------------------------------*/
if(MsgReceive())
{
   if (gp->resp_in.service == REQUEST)
   {
     if (RespAlloc())
     {
        /* Form the response and send */
        gp->respOut.reqId = gp->msgIn.reqId;
        gp->respOut.code  = gp->msgIn.code;
        gp->respOut.len   = 1;
        gp->respOut.data[0] = gp->msgIn.data[0] + 1;

        if( pgr_state == ACTIVE )
        {
          RespSend();
        }
        else
        {
          RespCancel();
        }
     }
   }

   RespFree();
}
```

3.9 void RespFree(void);
void resp_free(void);

Parameters:

NONE

Returns:

NONE

This function frees the resp_in structure for a received response message.

Example:

```
if( msg_receive() )
{
   ............

   resp_free();
}
```

3.10 Boolean msgReceive(void);
Boolean msg_receive(void);

Parameters:

NONE

Returns:

```
TRUE          If new message is received.
FALSE         If no new message is received.
```

This function receives a message into the msg_in structure. This is a non-blocking function which returns TRUE if a new message is received. Otherwise, it returns FALSE.

This function receives all messages in their raw form, superseding other specialized events like Wink, OfflineEvent, etc.

Example:

```
if( msg_receive() )
{
   ............
   msg_free();
}
```

3.11 Boolean RespReceive(void); Boolean resp_receive(void);

Parameters:

```
NONE
```

Returns:

```
TRUE          If new response is received.
FALSE         If no new response is received.
```

This function cause the resp_in structure to be filled from an incoming response message. If there is data already in the resp_in structure, and there is a pending response message to be received, then the prior data is discarded. This function returns TRUE if the resp_in structure is filled, and returns FALSE if there are no pending response messages. This function is non-blocking.

Example:

```
if( resp_receive() )
{
   ............
   resp_free();
}
```

3.12 void MsgCompletes(Status stat, MsgTag tag);

Parameters:

```
Stat          Indicate SUCCESS,FAILURE, or INVALID
tag           Indicates Completion for this particular
              message tag.
```

Returns:

```
NONE
```

This function is called when a message completes. The status field `stat` is set to either `SUCCESS,FAILURE, or INVALID`. The message tag indicates for which message the function was called. This function is also one of the required user functions defined elsewhere in this document.

4.0 GadgetStack Utility Functions API

4.1 int16 AddNV(NVDefinition* NVdef)

Parameters:

`NVdef`	`Definition structure which describes the attributes of the network variable.`

Returns:

`int16`	`Index number for the network variable.`

This function creates a network variable with the given definition. It returns the index of the network variable. This index is used in other functions like SendNV and GetNVAddr. For array variables, only one index value is returned, however each element is considered like a separate network variable.

The `NVDefinition` structure is used to describe network variable properties and to create a new network variable with proper attributes in network variable tables and SNVT structures.

```
typedef struct
{
   Bits    priority      : 1;  /*         TRUE or FALSE */
   Bits    direction     : 1;  /* NV_INPUT or NV_OUTPUT */
   Bits    selectorHi    : 6;  /* Sel-Hi only for non-bindable */
   Bits    selectorLo    : 8;  /* Sel-Lo only for non-bindable */

   Bits    bind          : 1;  /* 1 = bindable 0 = nobindable */
   Bits    turnaround    : 1;  /*            TRUE or FALSE */
   Bits    service       : 2;  /* ACKD, UNACKD_RPT, UNACKD */
   Bits    auth          : 1;  /*            TRUE or FALSE */
   Bits    :             3;    /*                   Unused */

   Bits    explodeArray  : 1;  /* Explode arrays in */
                               /* SNVT structure    */
   Bits    nvLength      : 7;  /* Length of network variable */
                               /* in bytes.                  */
                               /* For arrays, give the size  */
                               /* of each item.              */

   uint8   snvtDesc;           /*       snvtDesc_struct in byte */
                               /* form. Big_Endian             */
   uint8   snvtExt;             /* Extension record. Big_Endian. */
   int8    snvtType;           /*        0 => non-SNVT variable.*/
   uint8   rateEst;            /* Est. sustained msg. rate     */
                                 /*  (1/10 sec.)                */
```

```
   uint8   maxrEst;           /* Max. msg rate (1/10 sec.) */
   uint16  arrayCnt;          /*      0 for simple variable*/
                              /* dim for arrays.           */
   char    *nvName;      /*    Name of the network variable */
   char    *nvSdoc;      /* Self-doc string for the variable*/
   void    *varAddr;     /*        Address of the variable. */
} NVDefinition;
```

Example:

```
SNVT_temp    nv_temp;
NVDefinition nv_tempDef =
{
   FALSE, NV_INPUT, 0, 0, 1, 0, ACKD, FALSE,
   0, sizeof(SNVT_temp), 0xC0, 0x30, 39, 0, 0,
   0, "nv_temp", "Temperature", &nv_temp
};
int16 nv_temp_index;

nlong        nv_encoder;
NVDefinition nv_encoderDef =
{
   FALSE, NV_INPUT, 0, 0, 1, 0, ACKD, FALSE,
   0, sizeof(nlong), 0xC0, 0x30, 0, 0, 0,
   0, "nv_encoder", "Encoder", &nv_encoder
};
int16 nv_encoder_index;

Status AppInit(void)
{
   nv_temp_index    = AddNV(&nv_tempDef);
   nv_encoder_index = AddNV(&nv_encoderDef);

   if ((nv_temp_index == -1) || (nv_encoder_index == -1))
   {
        return(FAILURE);
   }

   return(SUCCESS);
}   /* End AppInit */
```

4.2 void Propagate(void);

Parameters:

NONE

Returns:

NONE

This function causes all output network variables to update and send.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

      intOut++;
      longOut++;
      intArrayOut[0]++;
      intArrayOut[1]++;
      intArrayOut[2]++;

      /* update all network variables.            */
      Propagate();
   }

   /* Ignore incoming messages                    */
   msg_receive();

   /* Ignore incoming responses                   */
   resp_receive();
}   /* End DoApp() */
```

4.3 void PropagateNV(int16 nvIndex);

Parameters:

nvIndex	This is the network variable index number generated by the AddNV.

Returns:

NONE

This function causes one simple network variable or one entire array to be updated and sent.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

      intOut++;
      PropagateNV( intOutIndex );
      longOut++;
      PropagateNV( longOutIndex );

      /* Update whole array                        */
      intArrayOut[0]++;
      intArrayOut[1]++;
      intArrayOut[2]++;
      PropagateNV( intArrayOutIndex );
   }

   /* Ignore incoming messages                     */
   msg_receive();

   /* Ignore incoming responses                    */
   resp_receive();
}   /* End DoApp() */
```

4.4 void PropagateArrayNV(int16 arrayNVIndex, int16 nvIndex);

Parameters:

```
arrayNVIndex  Network variable array index or 0 if
              index is simple variable.
nvIndex       This is the network variable index number
              generated by the AddNV.
```

Returns:

NONE

This function causes an array element or any other simple network variable to update and send if necessary.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

      intOut++;
      PropagateNV( intOutIndex );
      longOut++;
      PropagateNV( longOutIndex );

      /* Update only 2nd array element                      */
      intArrayOut[2]++;

      /* Only update the 2nd array element        */
      PropagateArrayNV( intArrayOutIndex, 2 );
   }

   /* Ignore incoming messages                   */
   msg_receive();

   /* Ignore incoming responses                  */
   resp_receive();
}   /* End DoApp() */
```

4.5 void Poll(void);

Parameters:

NONE

Returns:

NONE

This function polls all input network variables.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

      /* update all input NV's                    */
      Poll();
   }

   /* Ignore incoming messages                    */
   msg_receive();

   /* Ignore incoming responses                   */
   resp_receive();
}   /* End DoApp() */

/* The update of the the Polled NV will hit here  */
void NVUpdateOccurs(int16 nvIndex, int16 nvArrayIndex)
{

   if (nvIndex == intInIndex)
   {
      printf("NVUpdateOccurs(intInIndex)\n");
      intSum = intSum + intIn;
   }
   else if (nvIndex == longInIndex)
   {
      printf("NVUpdateOccurs(longInIndex)\n");
      longSum = longSum + longIn;
   }
   else if (nvIndex == intArrayInIndex)
   {
      printf("NVUpdateOccurs(intArrayInIndex)\n");
      intSum = intSum + intArrayIn[nvArrayIndex];
   }
```

```
}   /* End NVUpdateOccurs() */
```

4.6 void PollNV(int16 nvIndex);

Parameters:

NONE

Returns:

nvIndex index of network variable.

This function polls a specific simple input network variable or array.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

      /* update just LongInIndex                  */
      PollNV( LongInIndex);
   }

   /* Ignore incoming messages                    */
   msg_receive();

   /* Ignore incoming responses                   */
   resp_receive();
}   /* End DoApp() */

/* The update of the the Polled NV will hit here  */
void NVUpdateOccurs(int16 nvIndex, int16 nvArrayIndex)
{

   if (nvIndex == intInIndex)
   {
      printf("NVUpdateOccurs(intInIndex)\n");
      intSum = intSum + intIn;
   }
   else if (nvIndex == longInIndex)
   {
      printf("NVUpdateOccurs(longInIndex)\n");
      longSum = longSum + longIn;
   }
   else if (nvIndex == intArrayInIndex)
```

```
   {
      printf("NVUpdateOccurs(intArrayInIndex)\n");
      intSum = intSum + intArrayIn[nvArrayIndex];
   }
}   /* End NVUpdateOccurs() */
```

4.7 void PollArrayNV(int16 arrayNVIndex, int16 nvIndex);

Parameters:

```
arrayNVIndex  Network variable array index or 0 if
              index is simple variable.
nvIndex       This is the network variable index number
              generated by the AddNV.
```

Returns:

NONE

This function is used to poll a specific array element or any other simple network variable.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

    /* update just intArrayInIndex element 2  */
      PollArrayNV( intArrayInIndex, 2);
   }

   /* Ignore incoming messages                    */
   msg_receive();

   /* Ignore incoming responses                   */
   resp_receive();
}   /* End DoApp() */

/* The update of the the Polled NV will hit here  */
void NVUpdateOccurs(int16 nvIndex, int16 nvArrayIndex)
{

   if (nvIndex == intInIndex)
   {
      printf("NVUpdateOccurs(intInIndex)\n");
      intSum = intSum + intIn;
```

```
   }
   else if (nvIndex == longInIndex)
   {
      printf("NVUpdateOccurs(longInIndex)\n");
      longSum = longSum + longIn;
   }
   else if (nvIndex == intArrayInIndex)
   {
      printf("NVUpdateOccurs(intArrayInIndex)\n");
      intSum = intSum + intArrayIn[nvArrayIndex];
   }
}   /* End NVUpdateOccurs() */
```

4.8 void GoOffline(void);

Parameters:

NONE

Returns:

NONE

An application can call this function to put itself offline.

Example:

```
void NVUpdateOccurs(int16 nvIndex, int16 nvArrayIndex)
{

   if (nvIndex == NVInTurnOff)
   {
      printf("Application Going Offline");
      GoOffline();
   }
}   /* End NVUpdateOccurs() */
```

4.9 void GoUnconfigured(void);

Parameters:

NONE

Returns:

NONE

An application can call this function to set itself unconfigured.

Example:

```
void DoApp()
{
   ............

   if( Error == REAL_BAD_ERROR )
   {
      printf("Unrecoverable Error: go unconfigured");
      GoUnconfigured();
   }
}
```

4.10 MsgTag NewMsgTag(BindNoBind bindStatusIn);

Parameters:

`bindStatusIn` Value to indicate binding status, see enum BindNoBind below.

Returns:

`MsgTag` The created message tag.

This function generates a new message tag. The bindStatusIn value is selected from the `BindNoBind` enumeration:

```
typedef enum
{
   NOBIND = 0,
   NON_BINDABLE = 0,   /* Same as NOBIND. */
   BIND   = 1,
   BINDABLE = 1       /* Same as BIND.   */
} BindNoBind;
```

Example:

```
MsgTag tag0;
MsgTag tag1;

Status AppInit(void)
{
   tag0        = NewMsgTag(BINDABLE);
   tag1        = NewMsgTag(BINDABLE);

   /* Make sure we got the tags successfully */
   if (tag0 == -1 || tag1 == -1)
   {
      return(FAILURE);
```

```
   }

   return(SUCCESS);
}   /* End AppInit */
```

4.11 Boolean ServicePinMessage(void);

Parameters:

```
NONE
```

Returns:

```
TRUE          Indicates service pin sent                    .
FALSE         Indicates service pin failed to send.
```

This function sends a service pin message.

Example:

```
void DoApp(void)
{
   if (MsTimerExpired(&lightTimer))
   {
      SetMsTimer(&lightTimer, 5000);

      /* Send service pin message every 5 sec.    */
      ServicePinMessage();
   }

   /* Ignore incoming messages                    */
   msg_receive();

   /* Ignore incoming responses                   */
   resp_receive();
}   /* End DoApp() */
```

4.12 void SetMsTimer(MsTimer *timerOut, uint16 initValueIn);

Parameters:

```
timerOut      Address of the timer (MsTimer struct)

initValueIn   Timer expiration value in milliseconds.
```

Returns:

```
NONE
```

This function sets a timer. The MsTimer defines a timer is as follows:

```
/* Software Timer definition. */
typedef struct
{
   uint32 curTimerValue; /* Number of clock ticks left in timer.*/
   uint32 lastUpdatedTime;/* Time when last updated.           */
   Boolean expired;       /* TRUE => it has already expired.    */
} MsTimer;
```

Example:

```
MsTimer msgOutTimer;

Status AppInit(void)
{
    /* Set a millisecond timer msgOutTimer for 3 seconds. */
    SetMsTimer(&msgOutTimer, 3000);
}
```

4.13 Boolean MsTimerExpired(MsTimer *timerInOut);

Parameters:

timerInOut	Address of the timer (MsTimer struct).

Returns:

Boolean	TRUE if timer expired, FALSE otherwise.

This function returns the status of a timer.

Example:

```
MsTimer msgOutTimer;

   if( MsTimerExpired(&msgOutTimer) )
   {
       SetMsTimer(&msgOutTimer, 3000);
   }
```

5.0 GadgetStack Required - User Defined Functions API

The following prototypes **MUST** be present in the User's application program in order for successful linking with the GadgetStack. The target application is implemented by filling in the functions listed below. The DoApp() is similar to the main, in that the body of the application and custom code will reside with in the DoApp() function. The other functions are executed under defined conditions.

5.1 void MsgCompletes(Status stat, MsgTag tag); Boolean msg_completes(void);

Parameters:

```
Stat          Indicate SUCCESS,FAILURE, or INVALID
tag           Indicates Completion for this particular
              message tag.
```

Returns:

```
NONE
```

This function is called when a message completes, either Success, Failure or Invalid. The message tag indicated for which message the function was called. This function is also one of the required user functions defined later in this document. It is defined in both places for convenience.

5.2 Status AppInit(void);

This function should contain any variable or data that needs to be initialized prior to the application starting.

5.3 void AppReset(void);

Place code in this function that needs to be executed each time the node or stack is reset. This is true for both hardware and software resets.

5.4 void DoApp(void);

DoApp is the function where the main application code is located. The DoApp is executed as part of the GadgetStacks main execution loop. Most custom event handling is done by the application program inside the DoApp function.

5.5 void NVUpdateCompletes(Status stat, int16 nvIndex, int16 nvArrayIndex);

NVUpdateCompletes is called when an network variable update or a network variable poll completes. The second parameter is the array index for array variables. A value of zero is used for simple variables.

5.6 void NVUpdateOccurs(int16 nvIndex, int16 nvArrayIndex);

NVUpdateOccurs is called when an input network variable has been changed. The second parameter is the array index for the array variable. A value of zero in the field is used for simple variables.

5.7 void Wink(void);

The Wink function is executed when the GadgetStack receives a wink network management message.

5.8 void OfflineEvent(void);

The OfflineEvent function is called when the GadgetStack has been instructed to go offline by a network management message.

5.9 void OnlineEvent(void);

The OnlineEvent function is called when the GadgetStack is instructed by a management tool to go online.