



Applied Systems Engineering, Inc.

**Technical Note #9**  
**Gpt\_n**  
**Resource Management for the GPT**

## Technical Note 9: Resource Management

<b>Gpt_n</b> .....	3
<b>GPT_n COMM Port properties</b> .....	4
<b>Protocol Handle</b> .....	4
<b>Protocol Properties</b> .....	5
<b>Default Property Values</b> .....	5
<b>Default Property Values</b> .....	5
<b>Properties Common to all protocols</b> .....	5
<b>DacConfigure</b> .....	6
<b>GptOpen</b> .....	6
<b>Message Processing Loop</b> .....	6
<b>WaitComm Wait for I/O completion</b> .....	7
<b>Event Dispatch</b> .....	7
<b>Multiple Gpt_n Instances</b> .....	7
<b>Resource Management</b> .....	8

## Technical Note 9: Resource Management

This document describes Protocol PAK resource management and thread execution.

### Gpt\_n

Gpt\_n is the body of the thread run by the GPT test program TEST.C. There is one instance of GPT\_n running for each channel defined on the system. For systems that use a single COMM port there is only one instance of the Gpt\_n thread running. Each Gpt\_n thread can be running the same protocol or a different protocol.

```
/******\  
* ROUTINE: Gpt_Init_n - Initialize GPT channel  
*  
*   In a multiple line configuration multiple GPTs can be run. Each  
*   line will have its own GPT instance running. Each GPT instance  
*   may be running any of the protocol plug-in translators, or each  
*   instance can run the same protocol plug-in.  
*  
* CALLING SEQUENCE  
*  
*   Gpt_n(args)  
*  
*   LPGPTARGS   args       Gpt arguments  
*  
* RETURNS  
*  
*   This routine does not return.  
*  
\\*****/  
DWORD WINAPI   Gpt_n( pThreadObj op )  
{  
    HANDLE          hCom;   /* handle to COMM      */  
    HANDLE          hPro;   /* handle to protocol */  
    DWORD           events;  
    long            start_time;  
    long            curr_time;  
    long            new_time;  
    COMPARAMS      comprop;  
    int             num;  
    HANDLE          hDac;  
    LPGPTPROPSHEET pSheet;  
    ProtocolList   *p;  
    char            errmsg[80];  
    LPGPTARGS      arg = (LPGPTARGS) op->Param;  
  
    hDac = arg->hDac;  
  
    printf( "Gpt_%d Started\n", (int) arg->Line );
```

## Technical Note 9: Resource Management

### GPT\_n COMM Port properties

The thread builds up a table of comm port properties. These properties will be used to open the communications port. The port open is operating system specific. Therefore the port open is done outside of the GPT by the user application. The user creates a handle to the open port hCom. This handle will be used by the GPT on all COMM calls.

```
/* Initialize COMM properties */
memset( &comprop, 0, sizeof(COMPARAMS) );

strcpy( comprop.DevName, a.DevName );

/* Bias comm port by port number */
num = atoi( &a.DevName[3] );
num = num + arg->Port;
sprintf( comprop.DevName, "COM%d", num );

/* Create backup port name if used */
if ( a.Redundant )
{
    num++;
    sprintf( comprop.BkpName, "COM%d", num );
}

comprop.Parity = CM_NONE;
comprop.Switched = arg->Switched;
comprop.BaudRate = arg->Speed;
comprop.Squelch = 0;
comprop.PreTransmissionDelay = 40;
comprop.PostTransmissionDelay = 40;
comprop.DataLen = 0;
comprop.StopBits = 0;
comprop.TxCarrier = CM_SWITCHED;
comprop.RxCarrier = CM_SWITCHED;
comprop.Redundant = a.Redundant; /* No backup port */
comprop.IntraMessageDelay = 0;
comprop.InterMessageDelay = 0;
comprop.IntraMessageTimeout = 0;
comprop.Smart = arg->Smart;
comprop.Protocol = arg->PT;

#if defined(COMM_EMULATE_VERSION_2)
    if ( (hCom = OpenCommParams( &comprop )) == NULL )
        locHalt("Can't open communications port\n");
#else
    if ( (hCom = OpenComm( &comprop )) == NULL )
        locHalt("Can't open communications port\n");
#endif
#endif
```

### Protocol Handle

Each protocol has a handle. The protocol handle was allocated and passed into Gpt\_n by the routine that created the GPT\_n thread. For DNP 3.0 the protocol handle was constructed by allocating a structure of type CO\_DNP.

```
/* Create protocol instance */
```

## Technical Note 9: Resource Management

```
hPro = arg->hPro;
```

### Protocol Properties

Each protocol is controlled by its properties. The user allocates a property sheet. For DNP this sheet is DNPPROPSHEET. The user can change property defaults. When the property sheet is completed GptPutProperty is used with the property sheet structure and the protocol handle to change the properties.

```
/* Protocol Definition */  
p = &protocols[arg->PT];  
pSheet = (LPGPTPROPSHEET) GPTCALLOC( 1, p->PropertySize );
```

### Default Property Values

GptInitialize is called to place out protocol properties into a default state. The property sheet can then be used to change the defaults. Each protocol has a constructor for example DnpInitialize. The protocol constructor is passed into GptInitialize and is called by GptInitialize.

```
if ( GptInitialize( hPro, p->ProtocolInit, locStateChange ) != Ok )  
{  
    GPTFREE( hPro );          /* Free protocol resources */  
    GPTFREE( pSheet );  
    sprintf(errmsg, "Error initializing the %s protocol component",  
p->ProtocolName );  
    locHalt( errmsg );  
}
```

### Default Property Values

GptGetProperty is called to move all of the default property values into the property sheet. Once in the sheet the defaults can be changed and GptPutProperty can be called to write the property sheet values back into the protocol component.

```
/* Get Default Protocol Properties */  
if ( GptGetProperty( hPro, IPROP_GPT_SHEET, (PVOID) pSheet, (WORD)  
p->PropertySize ) != Ok )  
{  
    GPTFREE( hPro );          /* Free protocol resources */  
    GPTFREE( pSheet );  
    sprintf(errmsg, "Error retrieving the %s protocol property  
sheet", p->ProtocolName );  
    locHalt( errmsg );  
}
```

### Properties Common to all protocols

Some properties are common to all protocols. These include the logical line that the protocol is assigned to, the port handle (hCom), an optional DAC handle (hDac), and the protocols translation mode. The handles hCom and hDac are user defined. There are never directly used by the GPT, but are only passed to the user application when DAC

## Technical Note 9: Resource Management

and COMM requests are performed.

```
/* Properties common to all protocols */
pSheet->Line = arg->Line;
pSheet->hCom = hCom;
pSheet->hDac = hDac;
pSheet->Translation = (GPTBYTE) p->TranslationMode;
pSheet->Switched = arg->Switched;
```

### DacConfigure

DacConfigure is called to Open the communications port, setup the protocol properites, and build a database of test points. This routine should be replaced by a user written routine to perform the same function on the user's target enviornment.

```
if ( DacConfigure( (BYTE) p->ProtocolType, hPro, pSheet, &comprop ) !=
Ok )
{
    GPTFREE( hPro );          /* Free protocol resources */
    GPTFREE( pSheet );
    sprintf(errmsg, "Error configuring the %s protocol component",
p->ProtocolName );
    locHalt(errmsg);
}

/* Done with the property Sheet, release it */
GPTFREE( pSheet );

/* Put comm in debug mode */
if ( a.Debug & DEBUG_NETWORK )
    SetComm( hCom, IOC_DEBUG, (DWORD) TRUE );
```

### GptOpen

After configuration is complete GptOpen is called to place the protocol in a running state.

```
/* Open Protocol */
if ( GptOpen( hPro ) != Ok )
{
    GPTFREE( hPro );          /* Free protocol resources */
    locHalt("failed to open protocol component");
}

/* Protocol Running, Process protocol events */
start_time = ReadTime();
```

### Message Processing Loop

Processing now enters a loop that runs forever. Timer and COMM events are passed into the protocol for processing.

```
do
{
```

## Technical Note 9: Resource Management

```
curr_time = ReadTime();
new_time = curr_time - start_time;
```

### *Tick*

The GPT is given a timer tick every  $\frac{1}{4}$  or so. This tick is used for I/O timeouts. The tick is not expected to be accurate.

```
if ( new_time >= LOCAL_TICK)
{
    start_time = curr_time;
    GptTick( hPro, (WORD) new_time );
}
```

## WaitComm Wait for I/O completion

The following routine is where the thread spends most of its time. It waits for a COMM read or write I/O completion. If an I/O completion is detected the event is passed into the GPT through the GptCompletion routine.

```
/* Wait for COMM event */
WaitComm(hCom, &events, LOCAL_TICK);
if (events) GptCompletion( hPro, events );

/* ----- */
/* Process Protocol Events. This will potentially */
/* cause DAC calls and COMM calls to process GPT */
/* Messages */
/* ----- */
```

## Event Dispatch

### Process Timer and COMM events

```
GptDispatch( hPro );

} while( ! ThreadIsStopping( (pThread) op->Thread ));
```

### *Shutdown*

Thread shutdown has been requested. The Communications port is closed. The GPT protocol component is closed, the handle to the thread is destroyed and the thread exits.

```
/* Cleanup and exit thread/procedure */
CloseComm( hCom );          /* Close Comm Port */
GptClose( hPro );          /* Shutdown GPT */
GPTFREE( hPro );          /* Free protocol resources */

return 0;
}
```

## Multiple Gpt\_n Instances

Multiple instances of the Gpt\_n thread can be constructed by the user application. Each

## Technical Note 9: Resource Management

instances manages its own communication port. Each instance also has its own DAC environment which means the separate events queues and static data can be supported for each Gpt\_n thread. Each instance can run any of the protocols in any of the translation modes supported by the GPT library source code. Reasons that a user might want to create multiple instances of the GPT include the following:

- The GPT acting as an RTU might be communicating with multiple remote masters. Each master uses its own circuit so that the user can maintain multiple event queues for each instance.
- Multiple protocols. The user might construct multiple instances so that different remote masters can communicate with a GPT RTU using different protocols. One master might be using the DNP protocol while another master is using the IEC protocol.
- Data Concentrator. The user might construct multiple instances so that the GPT can act as a data concentrator. Once instance is acting as an RTU and responds to requests from a remote master. The other instance acts as a master and polls remote RTUs.

## Resource Management

When a thread is created to manage a line the user provides as properties two handles. A COMM handle and DAC handle. In addition a logical line assignment for each instance is passed to the GPT as a property. Consider the case were the user wants to talk to two DNP masters over two different COMM channels. The user would create two separate threads each thread managing communication to 1 DNP remote master as follows:

```
/* Thread 1 */
Gpt_1()
{
    GPTPROPSHEET sheet;

    Sheet.Line = 0;      /* Logical Line */
    Sheet.Translation = GPT_RTU;
    Sheet.hDac = hDac1; /* Handle to first DAC */
    Sheet.hCom = hCom1; /* Handle to COMM */

}

/* Second thread talking to second remote master */
Gpt_2()
{
    Sheet.Line = 1;      /* Logical Line */
    Sheet.Translation = GPT_RTU;
    Sheet.hDac = hDac1; /* Handle to first DAC */
    Sheet.hCom = hCom1; /* Handle to COMM */

}
```

With two threads running the user now has the problem of determining which thread is accessing DAC and COMM resources. When the GPT makes a COMM request the first argument is a handle created by the user and assigned as a property to the GPT.

```
ReadComm( GPTHANDLE hCom, ... )
```

## Technical Note 9: Resource Management

In the above example hCom = hCom1 if Gpt\_1 is making the request or it is hCom2 if Gpt\_2 is making the request. DAC services requests are managed in a similar way. A DAC request to read variables can identify the source of the request as follows:

```
DacReadAnalogInput( LPDACENV ep, DACOBJID objId );
```

If Gpt\_1 is making the request then :

```
Ep->Line == 0 (GPTPROPSHEET.Line)  
Ep->hDac = hDac1 (GPTPROPSHEET.hDac);
```

If Gpt\_2 is make the request then:

```
Ep->Line == 1 (GPTPROPSHEET.Line)  
Ep->hDac == hDac1 (GPTPROPSHEET.hDac);
```

The user can use either the logical line or the hDac handle to locate the request variables. DAC services can employ 1 of two strategies for locating the request data. Either the 3-tuple <DACENV.Line,DACENV.Device,objId> can be used as a key. In this case all Gpt\_n instances can use the same hDac. This is how the test program works. Another approach would be to maintain data for each Gpt\_n instance in a separate context. Each Gpt\_n instance would have its own unique DAC handle. In this case the 2-type (DACENV.Device,objId) uniquely identifies the data. The user manages storage of the actual variable data. If the user wants to build a data concentrator then the variables returned to the remote master are probably over-mapped onto the variables requested from the remote RTUs. If the user wants to support variable access by two different masters then static variable data can be shared between the Gpt\_n instances while events are maintained in separate events queues. The test software shipped with the Gpt\_n always maintains separate static and event data for each GPT instance.