3. `checkSerRxData()` is called when serial data are available. It processes the data and sends a message to the foundation to notify it that these data belong to the application for processing.

APL.c

```
static void checkSerRxData(APL_SerRxDataMsg *qMsg)
{
    int indexer;

    printf("%s: Rcvd %u bytes (port %hu):\r\n", __FUNCTION__, qMsg->nBytes, qMsg->portNum);
    printUserData(qMsg->data, qMsg->nBytes);

    // Process / check received serial data here

    // Echo char back to port, increment count
    if (SERIAL_write((UARTPORTS)qMsg->portNum, qMsg->data, qMsg->nBytes) != qMsg->nBytes)
    {
        printf ("SERIAL_write failed to echo data.\r\n");
    }

    // Add received data to buffer for processing
    for (indexer = 0; indexer < qMsg->nBytes; indexer++)
    {
        addDataToBuffer (qMsg->data[indexer]);
    }

    // Populate return message and send back to the kernel
    // Set claimedData to TRUE if claiming data, FALSE otherwise
    qMsg->claimedData = TRUE;

    if(qMsg->claimedData == TRUE)
    {
        // Leave nBytes unchanged to claim all bytes,
        // otherwise set it to the number of bytes to claim from the buffer
        // qMsg->nBytes = n;

        // Set retVal to start index at which data is being claimed
        qMsg->retVal = 0;
    }
```

Figure 12-32: DemoAppSERIAL - Processing incoming serial data

4. The data are then sent out over the first available network in the call to `NIMM_send()`.

```
PL.c
/**
    Sends the received data out the network

    @return Nothing
*/
static void sendData ()
{
    // Do not send anything if there is nothing in the buffer
    if (ReceivedDataBufferIndex == 0)
    {
        return;
    }

    // Send the message
    if (NIMM_send(ReceivedDataBuffer, ReceivedDataBufferIndex) < 0)
    {
        printf("APL: Failed to send message.\r\n");
    }

    // Clear the buffer
    ReceivedDataBufferIndex = 0;

    // Clear the timer
    TIMER_clear (SERIAL_PORT_DEMO_QUERY_TIMER_NUM);
}
```

Figure 12-33: DemoAppSERIAL - Call to NIMM_send()

5. Figure 12-34 (from the Logger port) shows that the message has been successfully transmitted with the line: **`Tx[TERR}123456789.`**

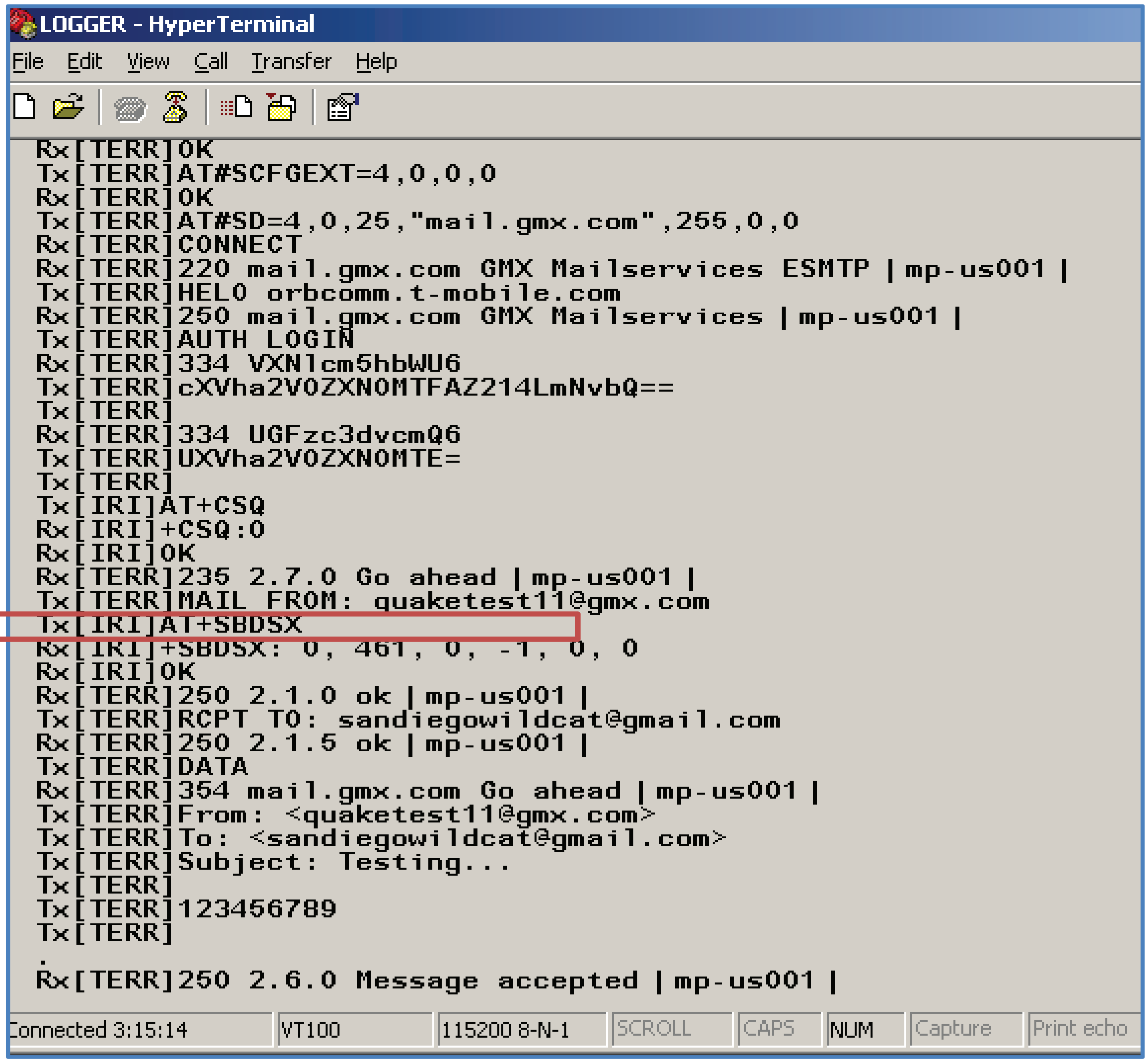


Figure 12-34: DemoAppSERIAL - Logger output of sending serial message

12.4.3 DemoAppREMOTE

This application demonstrates sending remote messages via GSM/GPRS and POP email to the modem and the proper evaluation and operation of these message events. Message events include sending an email to the modem, setting a relay, and remotely downloading a file to the modem. In this example, you must have at least one valid email address for your modem. Note that this sample application uses network-specific calls.

1. Select the DemoAppREMOTE Workspace from the drop-down list at the top, left-hand corner of the IAR IDE screen. Open the APL.c file, as shown below:

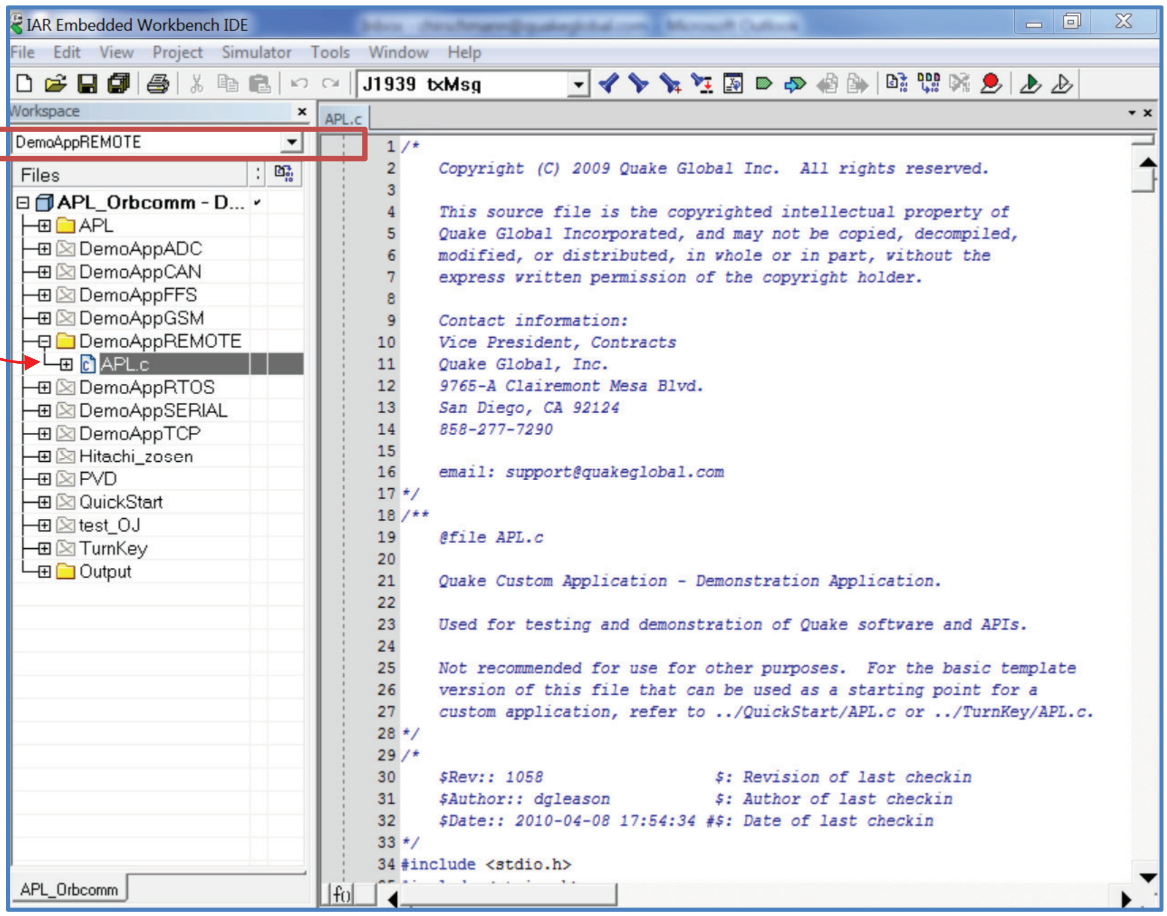


Figure 12-35: DemoAppREMOTE - Selecting the Workspace

2. Now build, load and execute DemoAppREMOTE. The instructions for building, loading and executing the code are the same as in Section 12, except that after building the application, the executable bin file is: …`/DemoAppREMOTE/exe/xxx-DemoAppREMOTE.bin`.

3. After startup, check the Logger output for the line **`APL DEMO: Remote Control.`** This indicates that the correct DemoApp is running.

12.4.3.1 Remotely set a relay (via email)

If the GSM/GPRS network is detected, DemoAppREMOTE checks for an incoming POP email message. You may send the email from any email program such as Microsoft Outlook.

- For an **ORBCOMM** modem, send an email to your modem with the words RELAY0=1 in the body of the message. The subject line of the email doesn't matter. Figure 12-36 is an example of a Relay email to an ORBCOMM modem.

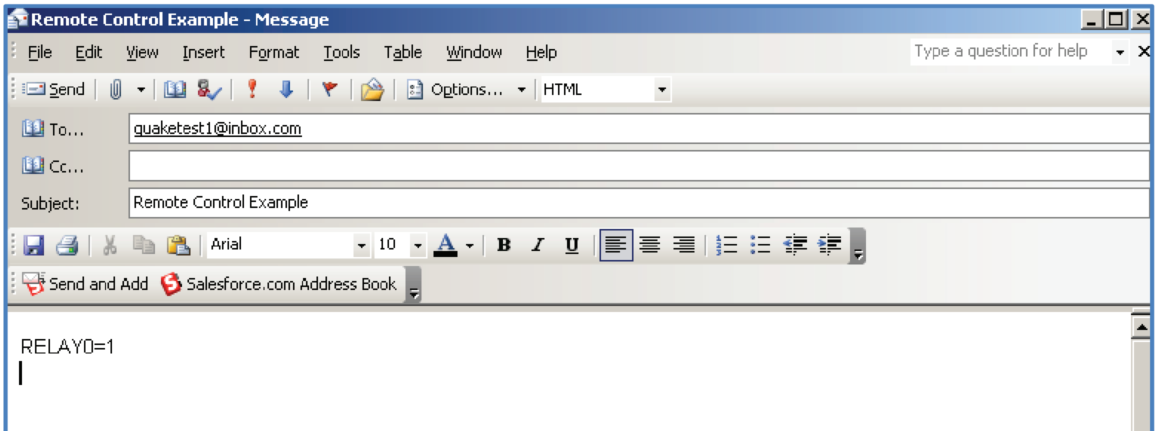


Figure 12-36: DemoAppREMOTE - Set relay email to ORBCOMM modem

- For an **Iridium** modem, send an email to: **data@sbd.iridium.com**, with the IMEI number of your modem in the subject of the message. The IMEI number should be visible on the white modem label. Include an attachment that contains "RELAY0=1". See Figure 12-37 for an example of a Relay email to an Iridium modem.

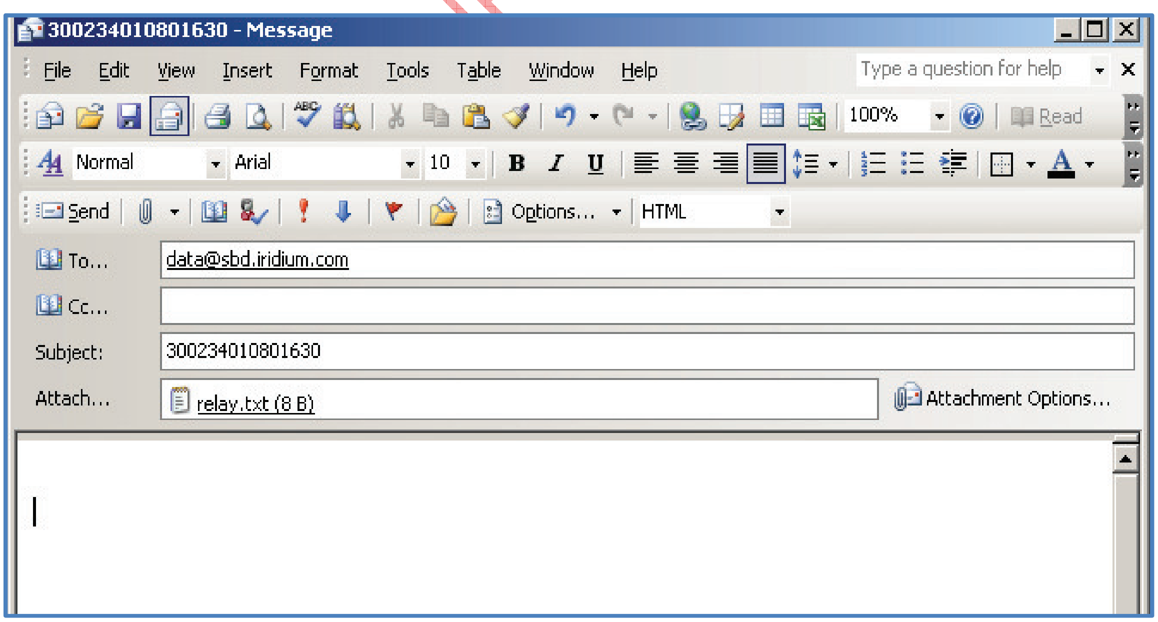


Figure 12-37: DemoAppREMOTE - Set relay email to Iridium modem

Figure 12-38 shows a call to `MSG_receiveTerr()`, which checks for an incoming POP email on the GSM/GPRS network after receiving a `TIMER_EVENT` (that occurs every `REMOTE_CONTROL_DEMO_CHECK_GSM_TIMER_DUR_SECS`).

APL.c

```
            // can be changed using the APL_taskSetNoEventInterval() API function
            // other event has occurred in that time.  Here is where it is recomm
            // applications perform any periodic status updates that may be requi

            break;

        case TIMER:

            if(qMsg->prm1 == REMOTE_CONTROL_DEMO_CHECK_GSM_TIMER_NUM)
            {
                printf("APL: Checking POP server\r\n");

                MSG_receiveTerr(TERR_POP);  // message arrives with MSG_RCVD

                // Reset the timer
                TIMER_setDuration(REMOTE_CONTROL_DEMO_CHECK_GSM_TIMER_NUM,
                                  REMOTE_CONTROL_DEMO_CHECK_GSM_TIMER_DUR_SECS
            }
            break;

        case CAN_MSG:
            break;

        case ORB_ANTENNA_VSWR:
```

Figure 12-38: DemoAppREMOTE - Checking GSM/GPRSPOP server

If a POP email message is received, it will trigger a `MSG_RCVD` event. Figure 12-39 shows the processing of the retrieved message.

- First, the event's parameter is checked to determine if it is an SMS, POP or satellite incoming message.
- When it is recognized as a POP message, `ParseCommandMessage()` is called to parse the incoming message into its constituent parts.

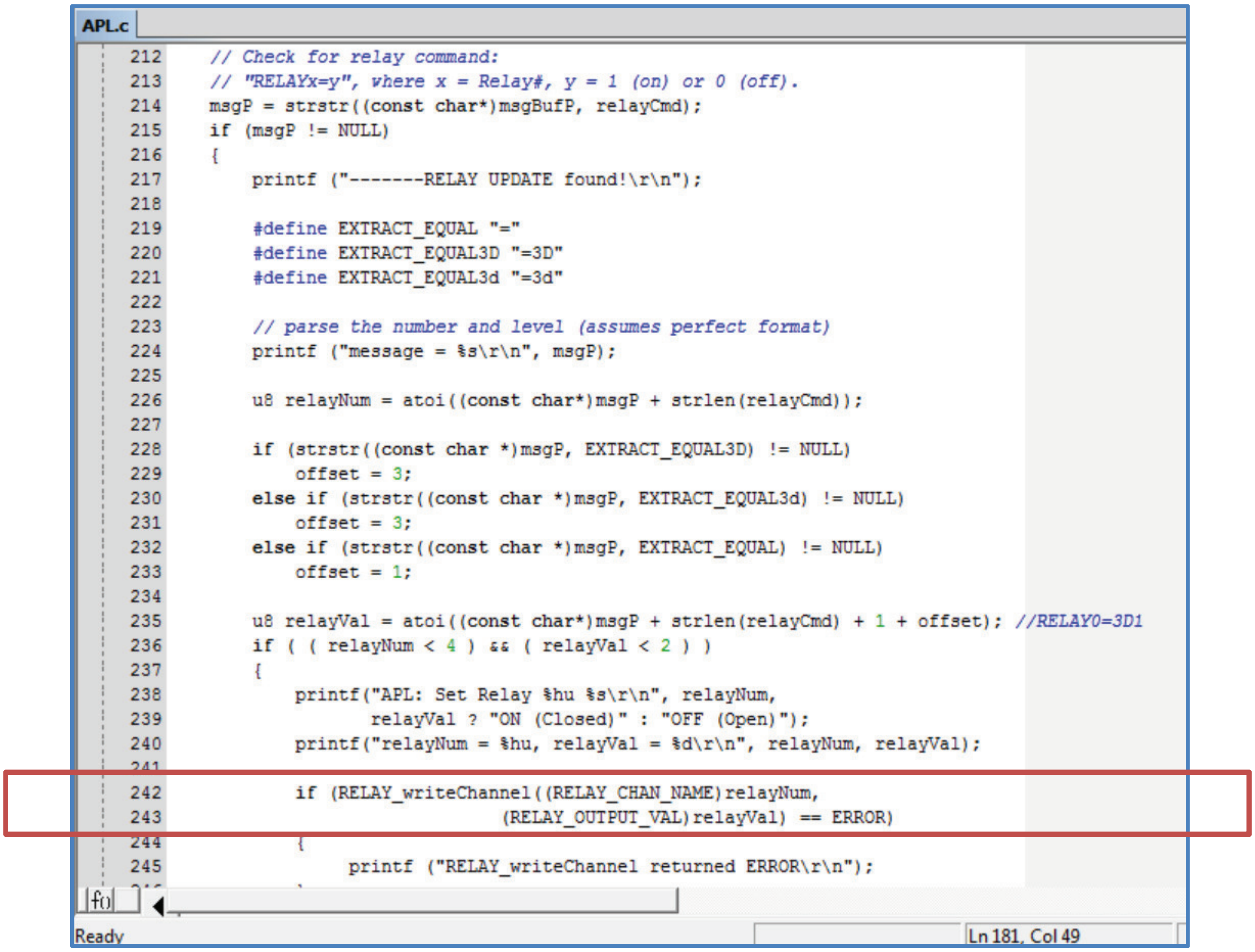
APL.c

```
    case MSG_RCVD: // A message packet was received from the network
       switch (qMsg->prm1)
        {
            case TERR_SMS:
                if (qMsg->msg != NULL)
                {
                    //printf("SMS message: %s\r\n", qMsg->msg);
                    ParseCommandMessage( qMsg->msg, qMsg->msgLen );
                }
                break;

            case TERR_POP:
                if (qMsg->msg != NULL)
                {
                    printf("POP email received (length %d)\r\n", qMsg->msgLen);
                    USER_printUserData( qMsg->msg, qMsg->msgLen );

                    ParseCommandMessage( qMsg->msg, qMsg->msgLen );
                }
                break;

            case SATELLITE:
                {
                    printf("Satellite message\r\n");
                    ParseCommandMessage( qMsg->msg, qMsg->msgLen );
                }
                break;
        }
        break;

case SHUTDOWN:
```

Figure 12-39: DemoAppREMOTE - Evaluating incoming message

In Figure 12-40, `ParseCommandMessage()` does the following:

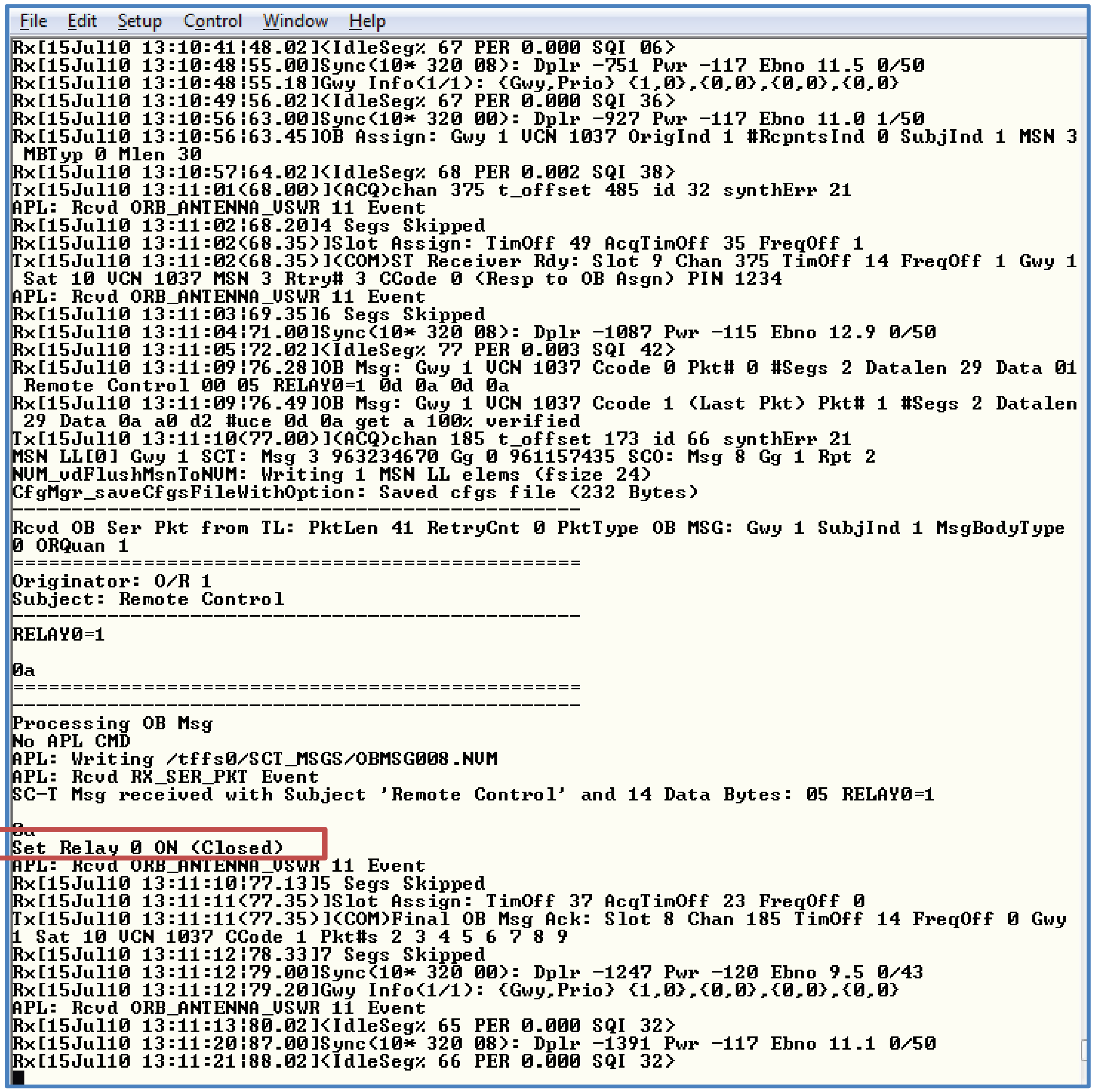
- An occurrence of the relayCmd, "RELAY" is sought.
- If it is found, the relay number and value are extracted by the `atoi()` calls and the relay number is verified to be less than two.
- Using the function, `RELAY_writeChannel()`, a command is sent to the RELAY module to set the appropriate relay.



```
APL.c
// Check for relay command:
// "RELAYx=y", where x = Relay#, y = 1 (on) or 0 (off).
msgP = strstr((const char*)msgBufP, relayCmd);
if (msgP != NULL)
{
    printf ("-------RELAY UPDATE found!\r\n");

    #define EXTRACT_EQUAL "="
    #define EXTRACT_EQUAL3D "=3D"
    #define EXTRACT_EQUAL3d "=3d"

    // parse the number and level (assumes perfect format)
    printf ("message = %s\r\n", msgP);

    u8 relayNum = atoi((const char*)msgP + strlen(relayCmd));

    if (strstr((const char *)msgP, EXTRACT_EQUAL3D) != NULL)
        offset = 3;
    else if (strstr((const char *)msgP, EXTRACT_EQUAL3d) != NULL)
        offset = 3;
    else if (strstr((const char *)msgP, EXTRACT_EQUAL) != NULL)
        offset = 1;

    u8 relayVal = atoi((const char*)msgP + strlen(relayCmd) + 1 + offset); //RELAY0=3D1
    if ( ( relayNum < 4 ) && ( relayVal < 2 ) )
    {
        printf("APL: Set Relay %hu %s\r\n", relayNum,
               relayVal ? "ON (Closed)" : "OFF (Open)");
        printf("relayNum = %hu, relayVal = %d\r\n", relayNum, relayVal);

        if (RELAY_writeChannel((RELAY_CHAN_NAME)relayNum,
                               (RELAY_OUTPUT_VAL)relayVal) == ERROR)
        {
            printf ("RELAY_writeChannel returned ERROR\r\n");
Ready                                   Ln 181, Col 49
```

Figure 12-40: DemoAppREMOTE - Parsing command message

Figure 12-41 shows the Logger output at the time the message is received. Note the Logger output from the application: **`Set Relay 0 ON (Closed)`**. This indicates that the message was received, parsed properly and that the relay has been closed.



```
File  Edit  Setup  Control  Window  Help
Rx[15Jul10 13:10:41|48.02]<IdleSeg% 67 PER 0.000 SQI 06>
Rx[15Jul10 13:10:48|55.00]Sync(10* 320 08): Dplr -751 Pwr -117 Ebno 11.5 0/50
Rx[15Jul10 13:10:48|55.18]Gwy Info(1/1): (Gwy,Prio) (1,0),(0,0),(0,0),(0,0)
Rx[15Jul10 13:10:49|56.02]<IdleSeg% 67 PER 0.000 SQI 36>
Rx[15Jul10 13:10:56|63.00]Sync(10* 320 00): Dplr -927 Pwr -117 Ebno 11.0 1/50
Rx[15Jul10 13:10:56|63.45]OB Assign: Gwy 1 VCN 1037 OrigInd 1 #RcpntsInd 0 SubjInd 1 MSN 3
 MBTyp 0 Mlen 30
Rx[15Jul10 13:10:57|64.02]<IdleSeg% 68 PER 0.002 SQI 38>
Tx[15Jul10 13:11:01(68.00)]<ACQ>chan 375 t_offset 485 id 32 synthErr 21
APL: Rcvd ORB_ANTENNA_VSWR 11 Event
Rx[15Jul10 13:11:02|68.20]4 Segs Skipped
Rx[15Jul10 13:11:02(68.35)]Slot Assign: TimOff 49 AcqTimOff 35 FreqOff 1
Tx[15Jul10 13:11:02(68.35)]<COM>ST Receiver Rdy: Slot 9 Chan 375 TimOff 14 FreqOff 1 Gwy 1
 Sat 10 VCN 1037 MSN 3 Rtry# 3 CCode 0 (Resp to OB Asgn) PIN 1234
APL: Rcvd ORB_ANTENNA_VSWR 11 Event
Rx[15Jul10 13:11:03|69.35]6 Segs Skipped
Rx[15Jul10 13:11:04|71.00]Sync(10* 320 08): Dplr -1087 Pwr -115 Ebno 12.9 0/50
Rx[15Jul10 13:11:05|72.02]<IdleSeg% 77 PER 0.003 SQI 42>
Rx[15Jul10 13:11:09|76.28]OB Msg: Gwy 1 VCN 1037 Ccode 0 Pkt# 0 #Segs 2 Datalen 29 Data 01
 Remote Control 00 05 RELAY0=1 0d 0a 0d 0a
Rx[15Jul10 13:11:09|76.49]OB Msg: Gwy 1 VCN 1037 Ccode 1 (Last Pkt) Pkt# 1 #Segs 2 Datalen
 29 Data 0a a0 d2 #uce 0d 0a get a 100% verified
Tx[15Jul10 13:11:10(77.00)]<ACQ>chan 185 t_offset 173 id 66 synthErr 21
MSN LL[0] Gwy 1 SCT: Msg 3 963234670 Gg 0 961157435 SCO: Msg 8 Gg 1 Rpt 2
NVM_vdFlushMsnToNVM: Writing 1 MSN LL elems (fsize 24)
CfgMgr_saveCfgsFileWithOption: Saved cfgs file (232 Bytes)
------------------------------------------------
Rcvd OB Ser Pkt from TL: PktLen 41 RetryCnt 0 PktType OB MSG: Gwy 1 SubjInd 1 MsgBodyType
0 ORQuan 1
================================================
Originator: O/R 1
Subject: Remote Control
------------------------------------------------
RELAY0=1

0a
================================================
------------------------------------------------
Processing OB Msg
No APL CMD
APL: Writing /tffs0/SCT_MSGS/OBMSG008.NVM
APL: Rcvd RX_SER_PKT Event
SC-T Msg received with Subject 'Remote Control' and 14 Data Bytes: 05 RELAY0=1

0a
Set Relay 0 ON (Closed)
APL: Rcvd ORB_ANTENNA_VSWR 11 Event
Rx[15Jul10 13:11:10|77.13]5 Segs Skipped
Rx[15Jul10 13:11:11(77.35)]Slot Assign: TimOff 37 AcqTimOff 23 FreqOff 0
Tx[15Jul10 13:11:11(77.35)]<COM>Final OB Msg Ack: Slot 8 Chan 185 TimOff 14 FreqOff 0 Gwy
1 Sat 10 VCN 1037 CCode 1 Pkt#s 2 3 4 5 6 7 8 9
Rx[15Jul10 13:11:12|78.33]7 Segs Skipped
Rx[15Jul10 13:11:12|79.00]Sync(10* 320 00): Dplr -1247 Pwr -120 Ebno 9.5 0/43
Rx[15Jul10 13:11:12|79.20]Gwy Info(1/1): (Gwy,Prio) (1,0),(0,0),(0,0),(0,0)
APL: Rcvd ORB_ANTENNA_VSWR 11 Event
Rx[15Jul10 13:11:13|80.02]<IdleSeg% 65 PER 0.000 SQI 32>
Rx[15Jul10 13:11:20|87.00]Sync(10* 320 08): Dplr -1391 Pwr -117 Ebno 11.1 0/50
Rx[15Jul10 13:11:21|88.02]<IdleSeg% 66 PER 0.000 SQI 32>
```

Figure 12-41: DemoAppREMOTE - Logger output for set relay

12.4.3.2 Remotely download a file to the modem (via email)

The Q4000/QPRO API includes function calls that allow a user to download a file from a remote server to the modem's file system. This allows you to update an application running on the modem. The DemoAppREMOTE application demonstrates the File Transfer Protocol (FTP) capability by processing an email to download a file to the modem.

The email sent to the Q4000/QPRO should contain the words "APPLICATION UPDATE" in the body of the message, as well as FTP download information. The body of the plain text email should contain information relating to your FTP server, and to the file to be downloaded to the Q4000/QPRO, as shown in Figure 12-42.

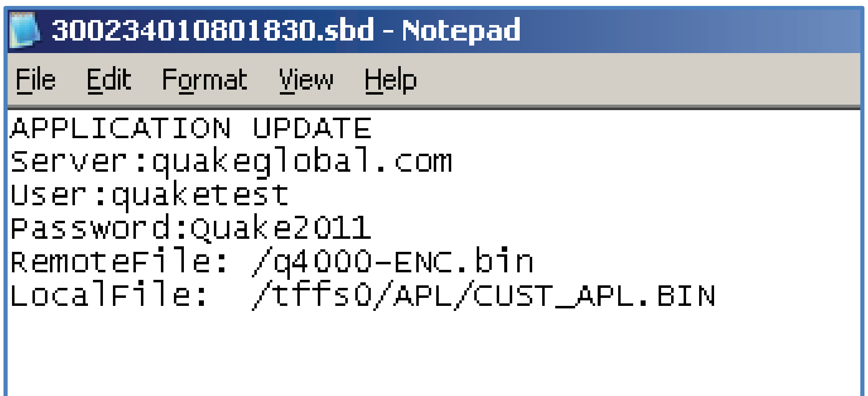


Figure 12-42: DemoAppREMOTE - Remote application update email

Note: For the Iridium network, this file must be created as an attachment and have an additional blank line below the LocalFile name.

The email is processed as in the "set relay" example above. After determining that it is a POP incoming message, it is parsed by `ParseCommandMessage()`. The code first checks to see if "APPLICATION UPDATE" is in the body of the message. If so, it extracts the Server Name, Username, Password, RemoteFile and LocalFile from the message. Note that the Remote File is the name of the file on the FTP server and LocalFile is the name the file will be on the modem itself.

Any file that is named: /tffs0/APL/CUST_APL.bin is executed as the custom application when the modem boots up, so if this file is replaced, the new custom application is executed after the next boot sequence.

Code to extract the relevant information from the email and accomplish the transfer is shown in Figure 12-43.

```
{
    #define MIN(_a, _b) ((_a) < (_b) ? (_a) : (_b))
    #define TOKEN_PARSER_LENGTH 40
    #define EXTRACT_SERVER_NAME "Server:"
    #define EXTRACT_USER_NAME "Username:"
    #define EXTRACT_PASSWORD "Password:"
    #define EXTRACT_REMOTE_FILE_NAME "RemoteFile:"
    #define EXTRACT_LOCAL_FILE_NAME "LocalFile:"

    // Initialize ftp download
    FTP_DownloadRequestData ftp;
    FTP_initializeDownloadRequestData (&ftp);

    char* result = NULL;
    u32 length;
    char token[TOKEN_PARSER_LENGTH];

    // Find server name token
    length = strExtract((char*)msgBodyP, EXTRACT_SERVER_NAME, (char**)&result);
    if ((result != NULL) && (length != 0))
    {
        memset (token, 0, sizeof(token));
        memcpy (token, result, MIN (length, sizeof(token) - 1) );
        FTP_remoteFileSetServerName(&ftp.remoteFile, token);
    }

    // Username
    length = strExtract((char*)msgBodyP, EXTRACT_USER_NAME, (char**)&result);
    if ((result != NULL) && (length != 0))
    {
        memset (token, 0, sizeof(token));
        memcpy (token, result, MIN (length, sizeof(token) - 1) );
        FTP_remoteFileSetUsername(&ftp.remoteFile, token);
    }

    // Password
    length = strExtract((char*)msgBodyP, EXTRACT_PASSWORD, (char**)&result);
    if ((result != NULL) && (length != 0))
    {
        memset (token, 0, sizeof(token));
        memcpy (token, result, MIN (length, sizeof(token) - 1) );
        FTP_remoteFileSetPassword(&ftp.remoteFile, token);
    }
```

Figure 12-43: DemoAppREMOTE - Parse incoming remote application update message

In Figure 12-44, note that in the main application task loop, there a case statement to handle the `APL_FTP_STATE_MSG`. This prints a message to the Logger port when the FTP transfer is complete.

APL.c

```
        case APL_AT_GET_MSG:
            //qcmGetCfgHandler(&qMsg.atOption);
            //free(qMsg.atOption.cmdData);
            break;

        case APL_CAN_MSG:
            //processCanMsg(&qMsg.canOption);
            break;

        case APL_GPS_MSG:
            break;

        case APL_FTP_STATE_MSG:

            printf ("APP FTP: state [%u], error [%u]\r\n",
                    qMsg.ftpOption.state, qMsg.ftpOption.fault);

            if (qMsg.ftpOption.state == FTP_STATE_INITIALIZE)
            {
                printf ("APP FTP State = Initialize!\r\n");
            }

            if (qMsg.ftpOption.state == FTP_STATE_DOWNLOAD)
            {
                 printf ("APP FTP State = Download!\r\n");
            }

            if (qMsg.ftpOption.state == FTP_STATE_STOP && qMsg.ftpOption.fault == OK)
            {
                printf ("APP FTP download successful\r\n");
            }

            break;
```

Figure 12-44: DemoAppREMOTE - Event for FTP Load Successful

As the foundation code goes through the process of downloading the file, the various states are printed out to the Logger. Here are the examples of the states:

```
FTP_STATE_READY,                    // Initial state of the state machine
FTP_STATE_INITIALIZE,               // Request received state. Starts initialization
FTP_STATE_CONTEXT_ACTIVATED,        // State after network context is activated
FTP_STATE_CONNECTION_ACTIVATED,     // State after socket connection is activated
FTP_STATE_DOWNLOAD,                 // State before file is downloaded
FTP_STATE_CONNECTION_CLOSE,         // State where connection is closed
FTP_STATE_STOP                      // State before returning to ready state
```

After the file is downloaded, the new application must execute a modem reboot. An example of this call is:

```
SYS_pwrDownmodem ( s32 duration );
```

12.4.4 DemoAppCAN

The DemoAppCAN sample application demonstrates how the Q4000/QPRO receives Society of Automotive Engineers (SAE) J1939 messages on the CAN bus. SAE J1939 is the vehicle bus standard used for communication and diagnostics among vehicle components, originally by the car and heavy duty truck industry in the United States. Note that this sample application uses network-specific calls.

All J1939 packets, except for the request packet, contain eight bytes of data and a standard header which contains an index called a PGN (Parameter Group Number). A PGN identifies a message's function and associated data. J1939 attempts to define standard PGNs to encompass a wide range of automotive, agricultural, marine and off-road vehicle purposes. PGNs define the data, which are made up of a variable number of Suspect Parameter Number (SPN) elements defined for unique data.

An instrument cluster PGN may be received where the SPNs in the group are fuel level, oil pressure, and coolant temperature. Some of the parameters are 8 bits, some are 3 or 4 bits, and some could be 16 bits. The offsets and size of each parameter within a particular group are specified, like the PGNs, in the SAE documentation.

For example, SPN 184 of PGN 65266 is the "Engine Instantaneous Fuel Economy." PGN 65266 may be obtained from the CAN bus and parsed to get SPN 184 (two bytes at byte positions 3-4, numbering from byte position 1). Based on the SAE documentation, the data may be converted to the appropriate units (1/512 km/L per bit).

1. To run the CAN/J1939 example, select the DemoAppCAN Workspace from the drop-down list at the top, left-hand corner of the IAR IDE screen. Open the APL.c file, as shown below:

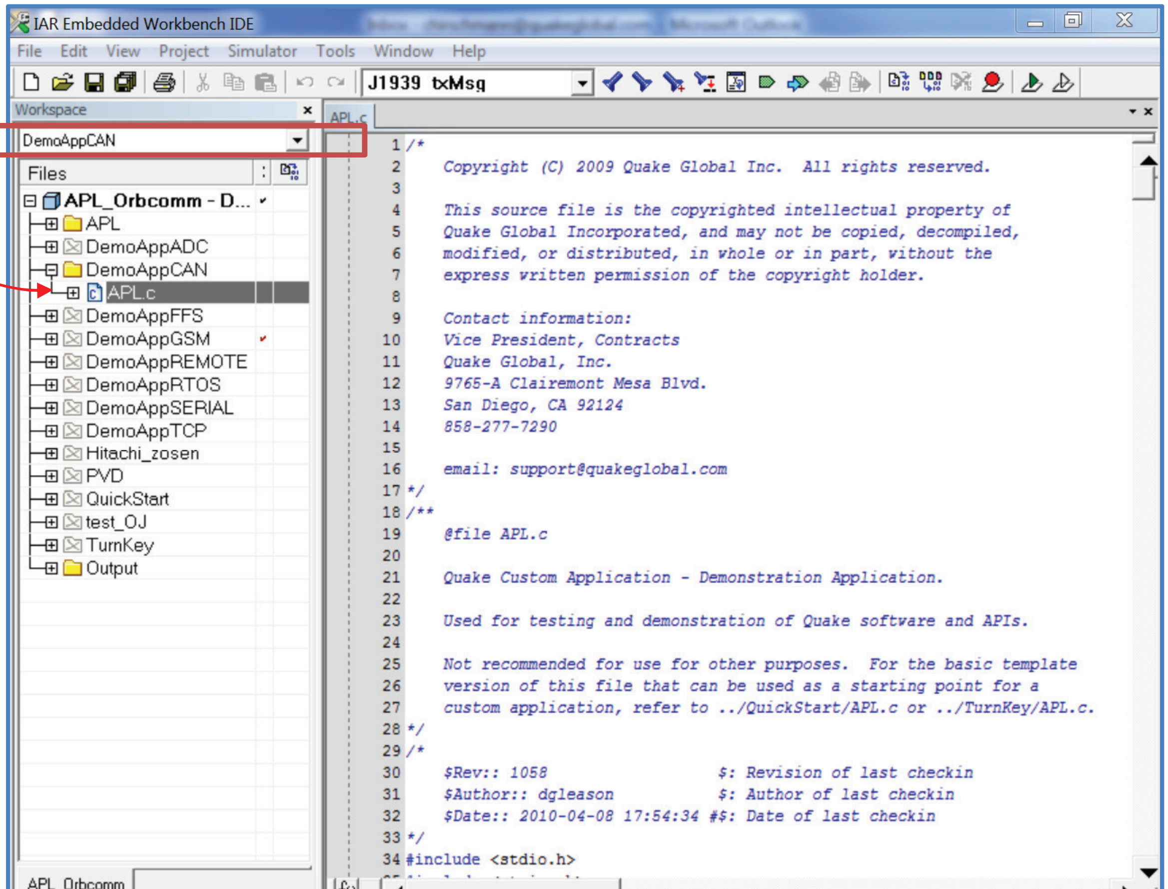


Figure 12-45: DemoAppCAN - Selecting the Workspace

2. Now build, load and execute DemoAppCAN. The instructions for building, loading and executing the code are the same as in Section 12, except that after building the application, the executable bin file is: .../`DemoAppCAN/exe/xxx-DemoAppCAN.bin`.

3. After startup, check the Logger output for the line **`APL DEMO: CAN/J1939.`** This indicates that the correct DemoApp is running.

This application was tested at QUAKE by using two Q4000 modems connected together. Their CAN inputs and outputs each sent and received data from each other at the same time. They were configured to use Parameter Group Number (PGN) message 61444. Figure 12-46 shows the initialization of the J1939 stack.

APL.c

```
}

// Initialize J1939 stack
u8 node = 83;

u8 j1939name[] = {
(J1939CFG_N_IN),
(J1939CFG_N_IN >> 8),
((uint8_t)((J1939CFG_N_MC << 5) & 0xff) | (J1939CFG_N_IN >> 16)),
(J1939CFG_N_MC >> 3),
((J1939CFG_N_FI << 3) | J1939CFG_N_EI),
 (J1939CFG_N_F),
 (J1939CFG_N_VS << 1),
 ((J1939CFG_N_AAC << 7) | (J1939CFG_N_IG << 4) | (J1939CFG_N_VSI))
};

J1939_init (node, j1939name, sizeof(j1939name));

```

Figure 12-46: DemoAppCAN - Initialization of J1939

Figure 12-47 shows the `POWER_ON` event. Note that the:

- buffer pointer in the J1939 message structure must be initialized to some allocated memory of sufficient size to hold the message data
- `buf_len` member in the structure must be initialized as well
- CAN timer is set to expire in `CAN_INTERVAL_SECS`.

APL.c

```
switch(qMsg->event)
{
    case POWER_ON: // Power on event received
        printf("APL: Examples enabled for:\r\n%s",exampleName);

        //be sure to provide an actual buffer in message structure, and set the
        //buf_len to the size of the buffer you have provided
        rxMsg.buf = rxData;
        rxMsg.buf_len = MAX_NUM_J1939_DATA_BYTES;

        //Set a timer, and when it expires send out a query
        //for a particular PGN on the CAN/J1939 bus
        if (TIMER_setDuration(CAN_TIMER_NUM,
                              CAN_INTERVAL_SECS) == ERROR)
        {
            printf ("TIMER_setDuration returned ERROR!\r\n");
        }

        // Request a GPS Fix using Measurement Table #0
        if (GPS_read(0) == ERROR)
        {
            printf ("GPS_read returned ERROR!\r\n");
        }
        break;
```

Figure 12-47: DemoAppCAN - Allocating CAN message buffer

Figure 12-48 shows the TIMER event, which first checks for the CAN timer number, then sends the CAN message.

```
APL.c
        case TIMER:

            if(qMsg->prm1 == CAN_TIMER_NUM)
            {
                loopCnt++;

                // send txMsg
                memset (txData, 0x55, sizeof(txData));
                txData[0] = 0x01;
                txData[1] = 0x23;
                txData[2] = 0x45;
                txData[3] = loopCnt & 0xff;
                txData[4] = (loopCnt >> 8) & 0xff;

                txMsg.buf = txData;
                txMsg.pgn = CANtxPgn;
                txMsg.buf_len = CANtxLen;
                txMsg.dst = J1939_ADDR_GLOBAL;
                txMsg.src = J1939_ADDR_EXPERIMENTAL_USE;
                txMsg.pri = 1;
                J1939_txMsg(&txMsg, &j1939Status);

                printf ("!!!!Sent CAN Msg at loop %d !!!\r\n", loopCnt);

                // receive rxMsg
                memset (rxData, 0x55, sizeof(rxData));
                rxData[3] = loopCnt & 0xff;
                rxData[4] = (loopCnt >> 8) & 0xff;
```

Figure 12-48: DemoAppCAN - Transmit J1939 data

In Figure 12-49, the TIMER event then reads the CAN data for this PGN from the other modem with the call `J1939_getPgnMsg()`.

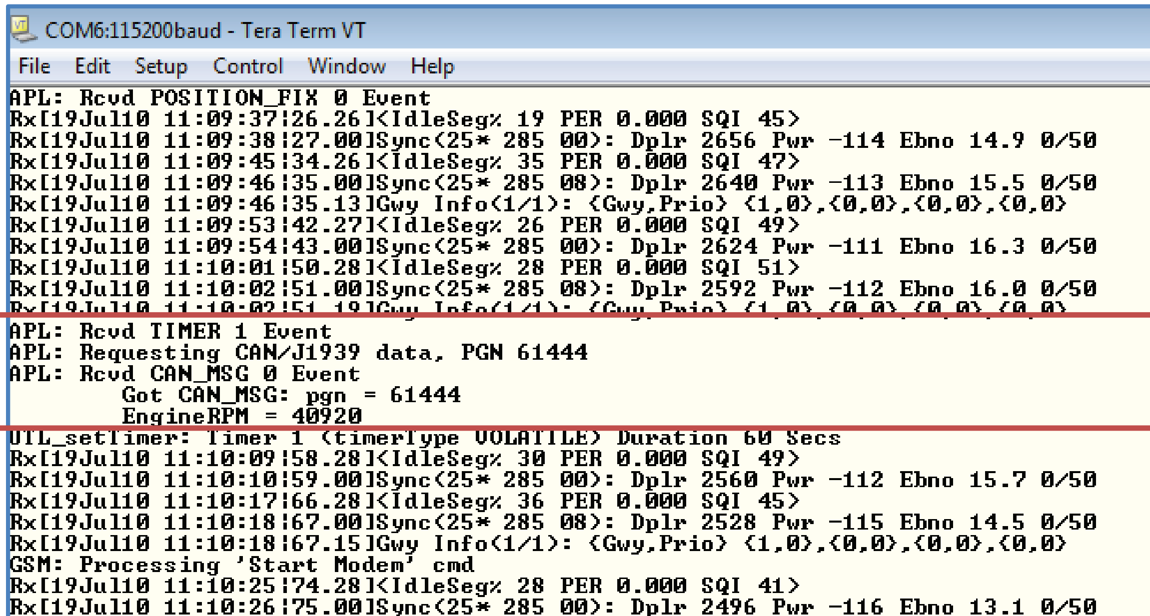
APL.c

```
                txMsg.pri = 1;
                J1939_txMsg(&txMsg, &j1939Status);

                printf ("!!!!Sent CAN Msg at loop %d !!!\r\n", loopCnt);

                // receive rxMsg
                memset (rxData, 0x55, sizeof(rxData));
                rxData[3] = loopCnt & 0xff;
                rxData[4] = (loopCnt >> 8) & 0xff;

                rxMsg.buf = rxData;
                rxMsg.pgn = CANrxPgn;
                rxMsg.buf_len = MAX_NUM_J1939_DATA_BYTES;
                rxMsg.dst = J1939_ADDR_GLOBAL;
                rxMsg.src = J1939_ADDR_EXPERIMENTAL_USE;
                rxMsg.pri = 1;
                if (J1939_getPgnMsg (&rxMsg, CANrxPgn, 1) != OK)
                {
                    printf ("!!! Bad status from J1939_getPgnMsg !!!\r\n");
                }
            }
            break;
```

Figure 12-49: DemoAppCAN - Receive J1939 data

Figure 12-50 shows the output after the CAN timer has expired, with the printout of the RPMs from the PGN 61444 message:

```
Got CAN_MSG: pgn = 61444
EngineRPM = 40920
```



```
COM6:115200baud - Tera Term VT
File  Edit  Setup  Control  Window  Help
APL: Rcvd POSITION_FIX 0 Event
Rx[19Jul10 11:09:37|26.26]<IdleSeg% 19 PER 0.000 SQI 45>
Rx[19Jul10 11:09:38|27.00]Sync(25* 285 00): Dplr 2656 Pwr -114 Ebno 14.9 0/50
Rx[19Jul10 11:09:45|34.26]<IdleSeg% 35 PER 0.000 SQI 47>
Rx[19Jul10 11:09:46|35.00]Sync(25* 285 08): Dplr 2640 Pwr -113 Ebno 15.5 0/50
Rx[19Jul10 11:09:46|35.13]Gwy Info(1/1): <Gwy,Prio> <1,0>,<0,0>,<0,0>,<0,0>
Rx[19Jul10 11:09:53|42.27]<IdleSeg% 26 PER 0.000 SQI 49>
Rx[19Jul10 11:09:54|43.00]Sync(25* 285 00): Dplr 2624 Pwr -111 Ebno 16.3 0/50
Rx[19Jul10 11:10:01|50.28]<IdleSeg% 28 PER 0.000 SQI 51>
Rx[19Jul10 11:10:02|51.00]Sync(25* 285 08): Dplr 2592 Pwr -112 Ebno 16.0 0/50
Rx[19Jul10 11:10:02|51.19]Gwy Info(1/1): <Gwy,Prio> <1,0>,<0,0>,<0,0>,<0,0>
APL: Rcvd TIMER 1 Event
APL: Requesting CAN/J1939 data, PGN 61444
APL: Rcvd CAN_MSG 0 Event
        Got CAN_MSG: pgn = 61444
        EngineRPM = 40920
UTL_setTimer: Timer 1 (timerType VOLATILE) Duration 60 Secs
Rx[19Jul10 11:10:09|58.28]<IdleSeg% 30 PER 0.000 SQI 49>
Rx[19Jul10 11:10:10|59.00]Sync(25* 285 00): Dplr 2560 Pwr -112 Ebno 15.7 0/50
Rx[19Jul10 11:10:17|66.28]<IdleSeg% 36 PER 0.000 SQI 45>
Rx[19Jul10 11:10:18|67.00]Sync(25* 285 08): Dplr 2528 Pwr -115 Ebno 14.5 0/50
Rx[19Jul10 11:10:18|67.15]Gwy Info(1/1): <Gwy,Prio> <1,0>,<0,0>,<0,0>,<0,0>
GSM: Processing 'Start Modem' cmd
Rx[19Jul10 11:10:25|74.28]<IdleSeg% 28 PER 0.000 SQI 41>
Rx[19Jul10 11:10:26|75.00]Sync(25* 285 00): Dplr 2496 Pwr -116 Ebno 13.1 0/50
```

Figure 12-50: DemoAppCAN - Logger output for engine RPM

12.4.5 DemoAppFFS

The FFS example demonstrates use of the Flash File System (FFS). The application gets a GPS position each time it starts, and creates a trail of the last positions which is stored in the FFS, so that the trail of positions is retained over power cycles. The FFS functionality demonstrated is fairly basic. Note that this application is network-specific.

1. Select the DemoAppFFS Workspace from the drop-down list at the top, left-hand corner of the IAR IDE screen. Open the APL.c file, as shown below:

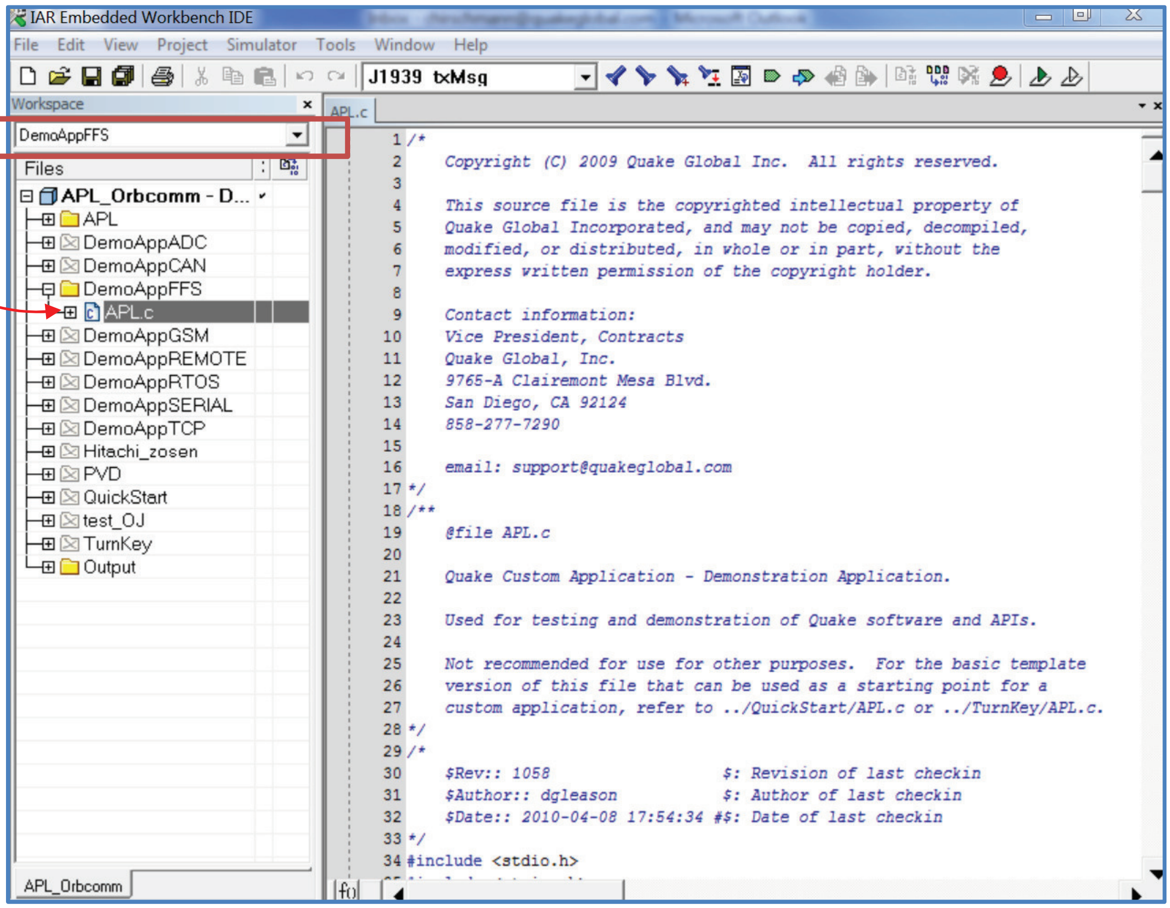


Figure 12-51: DemoAppFFS - Selecting the Workspace

2. Now build, load and execute DemoAppFFS. The instructions for building, loading and executing the code are the same as in Section 12, except that after building the application, the executable bin file is: …/`DemoAppFFS/exe/xxx-DemoAppFFS.bin`.

3. After startup, check the Logger output for the line **`APL DEMO: FFS`**. This indicates that the correct DemoApp is running.