Application Note AN-20

Transitioning to UCA32 Firmware

abaco SYSTEMS

Copyrights

Abaco Systems, Inc.
26 Castilian Drive, Suite B
Goleta, CA 93117
Main +1 805-965-8000 or +1 805- 883-6101
Support +1 805-965-8000 or +1 805- 883-6097

support@abaco.com (email)

https://www.abaco.com/products/avionics

Additional Resources

For more information, please visit the Abaco Systems website at:

www.abaco.com

Contents

Introduction

The Abaco systems MIL-STD-1553 product line was significantly enhanced to meet the increasing needs of customers. The Version 6 firmware, hereafter referred to as "V6" or UCA32, offers significant enhancements over V4/V5 UCA (legacy) firmware. V6 firmware is recommended for all new applications. Using V6 firmware requires updating to BusTools/1553-API version 8.x.

V6 Firmware.

V6 Enhancements include:

- Multiple BC buffers per message
- High resolution 64-bit time stamping (nanosecond resolution)
- New sequential BM buffer format optimized for efficient burst transfer
- More flexible and efficient memory allocation for message buffering in all modes
- Improved concurrency support for multi-processor and multi-threaded applications
- Enhanced triggering options

This Application Note describes how to transition an existing software application from V4/V5 to V6.

This Application Note assumes a basic knowledge of MIL-STD-1553. For information on the MIL-STD-1553 protocol refer to the "MIL-STD-1553 Tutorial" document available from Abaco Systems.

This Application Note assumes a working knowledge of Abaco Systems UCA products and their programming. Related information can be found in the "MIL-STD-1553 UCA Reference Manual", the "UCA32 LPU Reference Manual", and the "UCA32 Global Register Reference Manual" available from Abaco Systems.

BusTools/1553-API Version 8

BusTools/1553-API version 8.00 or greater is required to run V6 boards. BusTools/1553-API hides the majority of V6 firmware changes from the user application. Most applications can transition from the installed version of BusTools/1553-API to BusTools/1553-API version 8.x by recompiling with the new library and include files. BusTools/1553-API 8.x is compatible with all firmware versions. You can run a mix of V6 and V4/V5 boards in the same system using v8.x. However, you cannot run V6 boards unless BusTools/1553-API v8.x is installed.

Transition Steps

Install BusTools/1553-API Version 8.x

The first step in running with V6 firmware is to install BusTools/1553-API v8.x. If you have a previous BusTools/1553-API installed on your system you will need to uninstall that version. If you have a mix of V6 and V4/V5 boards in the system you will need to uninstall each board and re-install them using the 8.x BusTools/1553-API.

BusTools/1553-API implements the changes required for firmware compatibility across all supported V6/V5/V4 firmware so there is limited impact to existing user code. BusTools/1553-API v8.x has the same look and feel as earlier API versions. It supports the same functions with the same calling parameters as the earlier version. The structures used to configure the Bus Controller, Remote Terminal and Bus Monitor are unchanged (except for the new 64-bit time tag).

Rebuild Your Application

Most existing applications can run BusTools/1553-API version 8.06 with little or no changes. However, you need to recompile applications with the new busapi.h and link with the new libraries to run with the new BusTools/1553-API.

Windows users have pre-compiled libraries busapi32.lib/busapi64.lib and busapi32.dll/busapi64.dll available in the library directories. UNIX users (Linux, Solaris, and QNX) will need to rebuild the BusTools/1553-API library using the Makefiles and scripts that are provided with the installation. For VxWorks, Integrity and LynxOS you will need to create a library project with the BusTools/1553-API 8.06 source code according to instructions in the BusTools/1553-API User's Manual.

Changes to Your Application

Nearly all existing API functions work the same as in previous BusTools/1553-API versions. If your application uses these

BusTools/1553-API application functions without directly accessing the board's memory then your code will run with little or no change.

Changes to Time Tags

You may need to change how your application processes time tags. The API automatically handles the change from 48-bit microsecond to 64-bit nanosecond time tags. However, if your application is using time tags or converts time tags to a time string using BusTools_TimeGetString, you will need to modify that code. Starting with BusTools/1553-API v8.00, all time tags are stored in a 64-bit time structure. If the board is running V6 firmware the time tag resolution is 1-nanosecond, otherwise it has microsecond resolution. V5/V4 board time tags are 48-bits.

```
typedef struct bt1553_time
        {
        BT_U32BITmicroseconds;          // microseconds since start
        BT_U16BITtopuseconds;           // Most significant part of microseconds
        }
BT1553_TIME;  // Note that this is a 48-bit value for F/W 5.x or earlier
```

```
typedef struct bt1553_time
        {
        BT_U32BITmicroseconds;          // microseconds since start
        BT_U32BITtopuseconds;           // Most significant part of microseconds
        }
BT1553_TIME;  // Note that this is a 64-bit value for F/W 6.0
```

The above shows the structure changes for BT1553_TIME. For compatibility with older firmware the LSB can either be micro- or nanoseconds. When using this structure with V6 firmware the LSB is nanoseconds even though the structure elements still refers to microseconds. You can change to a new structure BT1553_TIME64 that references nanoseconds.

```
typedef struct bt1553_time64
        {
        BT_U32BIT nanoseconds;          // nanoseconds since start
        BT_U32BIT topnseconds;          // Most significant part of nanoseconds
        }
BT1553_TIME64;  // Note that this is a 64-bit value for F/W 6.0
```

If your code is using or displaying time tags it will need to change. BusTools_TimeGetString defaults to microseconds. There are two options for displaying time tags in nanoseconds. The code examples below show these two options.

Option 1: When using BusTools_TimeGetString you pass a pointer to a BT1553_TIME structure and a pointer to the string receiving the converted time string. If you set this string to "NANO" BusTools_TimeGetString will convert to nanoseconds.

```
char outbuf[80];
strncpy(outbuf,"NANO",4);
BusTools_TimeGetString(&bcmessage.time_tag,outbuf);
```

```
printf("Tag Time = %s\n",outbuf);
```

Option 2: Initialize the display option in BusTools_TimeTagMode to nanosecond. The following options provide microsecond time tag conversion.

```
API_TTD_RELM
API_TTD_IRIG
API_TTD_DATE
```

New options for nanosecond time tag conversion.

```
API_TTD_RELM_NS
API_TTD_IRIG_NS
API_TTD_DATE_NS

BusTools_TimeTagMode(cardnum,API_TTD_IRIG_NS,API_TTI_IRIG,API_TTM_FREE,0,0,0,0);
```

32-Bit Addressing and 32-Bit Registers

The internal addressing on the board changes from 19-bit to 32-bit. This allows a more flexible use of memory and the ability to access more than 1 megabyte per channel. The current default is still 1 megabyte per channel, but it is possible to have a larger amount of memory dedicated to a channel.

All register access is now 32-bit instead of 16-bit. There is only a single register region that combines the hardware register and the file registers that was used in the legacy firmware.

Most applications do not directly access registers or memory locations thus these changes are transparent. However, when compiling your application you may get a parameter mismatch warning. If so, change from 16-bit to 32-bit variables.

32-Bit addressing also impacts each of the internal data structures. Address pointers change from 16-bit to32-bit. This shifts the location of data within structures. You will need to review the new structure layouts as defined in the “UCA32 LPU Reference Manual” and the “UCA32 Global Register Reference Manual” if your code is directly accessing these structures in memory.

Directly Accessing Memory and Registers

Applications directly accessing the board’s memory using BusTools_MemoryRead, BusTools_MemoryRead2, BusTools_MemoryWrite, BusTools_MemoryWrite2 or other methods may need to change. Version 6 firmware has a different memory layout and there are changes to many of the internal structures. You will need to review the “UCA32 LPU Reference Manual”, and the “UCA32 Global Register Reference Manual” to become familiar with the board’s new design. BusTools_GetAddr still returns the start and end of the different memory sections, but offsets for specific data have changed. Both Bus

Controller and Bus Monitor structures have significantly changed. If your code directly accesses data in these buffers you will need to change the location from which you access the data.

Bus Controller Buffer Changes

Bus Controller buffers have changed to provide multiple BC data buffers. In earlier firmware revisions there were only one or two data buffers available and the firmware did not automatically switch between buffers. V6 firmware supports programming multiple data buffers setup as a circular linked list. The new structure significantly changes the internal Bus Controller buffers.

With legacy firmware versions, the API configures a Bus Controller control-buffer followed by one or two data buffers. The control-buffer contains all the information about the message including time tag and status. The data buffers contain only the transaction data.

With the new multiple buffers, the BC control-buffer is followed by 1 to *n* data buffers. The time tag and status are now part of the data buffer along with transaction data.

Bus Monitor Buffer Changes

The new Bus Monitor buffer format makes more efficient use of memory by sequentially storing only the amount of data recorded for each transaction. The previous Bus Monitor buffer format stored a fixed size buffer based on a 32 word message. Changing to a variable size buffer makes it impossible to read a specific Bus Monitor buffer. BusTools_BM_MessageGetaddr and BusTools_BM_MessageGetid now return API_NO_BUILD_SUPPORT. You will need to change any code dependent upon using those functions.

When you allocate BM buffers with BusTools_BM_MessageAlloc:

```
NOMANGLE BT_INT CCONV BusTools_BM_MessageAlloc(
  BT_UINT   cardnum,      // (i) card number (0 based)
  BT_UINT   mbuf_count,   // Message count converted to bytes;
  BT_UINT   * mbuf_actual,
  BT_U32BIT enable)
```

You are not allocating the specific number of buffers in *mbuf_count*, but rather the number of bytes it takes to store an *mbut_count* of 32-word messages. BusTools_BM_MessageAlloc converts the message count to the number bytes. A 32-word BM buffer uses 172 bytes. Therefore, if you allocate 100 BM messages, you are really reserving 17200 bytes for the Sequential Bus Monitor. When the Bus Monitor records data to this buffer location, it sequentially writes the data into this location. The data wraps back to the start when the last word in the buffer is written.

All other Bus Monitor functions work the same as before and the API_BM_MBUF structure is unchanged (except for the new nanosecond time tag).

Remote Terminal Buffer Changes

Remote Terminal structures are mostly unchanged with the exception of 32-bit addresses and 64-bit time tags. Check the manuals for the address and offset of any data you are directly accessing.

The V6 firmware design moves the Remote Terminal Address Buffer from RAM memory into the register region. This move is transparent to users, but if your application directly accesses the address buffer, you need to ensure that all accesses to the address buffer are 32-bits.

Interrupt Queue Changes

The structure and size of the interrupt queue has changed. In the legacy firmware the interrupt queue is 296 entries and each entry is a 3-word structure. This structure contains the interrupt mode, a message pointer and pointer to the next interrupt queue entry.

```
typedef struct iq_mblock
  {
  union
    {
    BT_U16BIT                modeword;
    BT1553_INTMODE           mode;
    } t;                                    // Interrupt mode/type bits
  BT_U16BIT                  msg_ptr;       // pointer to message to interrupt message
  BT_U16BIT                   nxt_int;      // points to the next interrupt in the queue
  }
IQ_MBLOCK;
```

This structure has a limit of 16 interrupt types defined in the following structure.

```
typedef struct bt1553_intmode
{
  BT_U16BIT iack:1;          // interrupt acknowledge bit     (LSB)
  BT_U16BIT timer:1;         // timer overflow or load        0x0002
  BT_U16BIT rt:1;            // rt interrupt                  0x0004
  BT_U16BIT bm:1;            // bm interrupt                  0x0008
  BT_U16BIT bc:1;            // bc interrupt                  0x0010
  BT_U16BIT bmtrig:1;        // bm trigger has occurred       0x0020
  BT_U16BIT ext_trig:1;      // External Trigger              0x0040
  BT_U16BIT bm_swap:1;       // BM-Only Buffer Swap*          0x0080
  BT_U16BIT bc_ctl:1;        // BC control interrupt          0x0100
  BT_U16BIT unused:7;        // unused                        (MSB)
}
BT1553_INTMODE;
```

The new interrupt queue is 512 entries of 2 double words defined by the following structure.

```
typedef struct iq_mblock_v6
  {
    BT_U32BIT mode;        //  Interrupt Mode
    BT_U32BIT msg_ptr;     // points to message that generated the interrupt
  }IQ_MBLOCK_V6;
```

The new interrupt structure eliminates the next entry pointer and the interrupt mode and message pointer are expanded to 32-bits. The interrupt mode can now hold over 4 billion interrupt types. There are several new interrupt mode types. The following list shows the current interrupt supported by BusTools/1553-API.

```
NO_INTERRUPT                0     // No interrupt present
TTIMER_LOAD_INTERRUPT       1     // Tag timer load interrupt (BC,RT,BM)
TRIGGER_IN_INTERRUPT        2     // Trigger input interrupt (BC,RT)
BC_MESSAGE_INTERRUPT        3     // BC Message interrupt (BC)
BC_CNTRLWD_INTERRUPT        4     // BC control word interrupt (BC)
RT_MESSAGE_INTERRUPT        5     // RT Message interrupt (RT)
BM_MESSAGE_INTERRUPT        6     // BM Message interrupt (BM)
BM_TRIGGER_INTERRUPT        7     // BM Message and trigger interrupt (BM)
MF_OVFL_INTERRUPT           8     // Minor Frame overflow interrupt (BC)
BC_BSY_MFOVFL_INTERRUPT     9     // BC busy set at start of frame (BC)
LP_MF_OVFL_INTERRUPT        10    // BC low priority aperiodic minor frame overflow (BC)
HP_MF_OVFL_INTERRUPT        11    // BC high priority aperiodic minor frame overflow (BC)
BM_HW_OVRFLW_INTERRUPT      12    // BM overflow detected head=tail
```

Conditional Branching

If you are using Condition Branch 2 with a memory location allocated by BusTools_MemoryAlloc, the application will now need to allocate a 32-bit word allocated on an even DWORD boundary. You can use either BusTools_MemoryAlloc and specify 4-bytes in the byte count or use BusTools_MemoryAlloc32. The address returned from these functions is used as the test address in the conditional branch set up.

Using New V6 Features

Most V6 improvements are hidden by BusTools/1553-API. They are automatically implemented when you run your application. There are a few new features that require changes to your code to implement. These are Multiple BC buffers, External clock (this was first introduced for some F/W 5.0 boards), and programmable triggers.

Multiple Bus Controller Buffers

Previous versions of BusTools/1553-API and the firmware supported only one or two BC data buffers switched under application control. Starting with V6 and BusTools/1553-API v8.x you have the option for multiple BC data buffers. These buffers are organized as a circular linked list. This allows applications to buffer data in high load environments.

Existing applications will work the same way as they did with earlier API and firmware versions. To use Multiple BC buffers your application will need to specify multiple buffers when initializing the Bus Controller. OR in the multiple buffer option (`MULTIPLE_BC_BUFFERS`) into the BC Option parameter in BusTools_BC_Init:

```
NOMANGLE BT_INT CCONV BusTools_BC_Init(
  BT_UINT   cardnum, // (i) card number (0 based)
  BT_UINT   bc_options,          // (i) REL_GAP                0x0
                                     FIXED_GAP               0xf
                                     MSG_SCHD                0x10
                                     NO_PRED_LOGIC           0x40
                                     FRAME_START_TIMING      0x20
                                     MULTIPLE_BC_BUFFERS     0x80
  BT_U32BIT Enable,              // (i) interrupt enables
  BT_UINT   wRetry,              // (i) retry enables
  BT_UINT   wTimeout1,           // (i) no response time out in microseconds
  BT_UINT   wTimeout2,           // (i) late response time out in microseconds
  BT_U32BIT frame,               // (i) minor frame period, in microseconds
  BT_UINT   num_buffers)   // (i) number of BC message buffers ( 1 or 2 for legacy)
```

```
status = BusTools_BC_Init(cardnum,MULTIPLE_BC_BUFFERS | MSG_SCHD,
BT1553_INT_END_OF_MESS, 0, 16, 14, 1000000, 0);
```

When the multiple BC buffer option is used the num_buffers parameter is ignored during Bus Controller initialization.

After selecting multiple buffers, you allocate each BC message buffer individually to program the number of data buffers for each message.

Use the new function BusTools_MessageBlockAlloc to allocate each buffer.

```
NOMANGLE BT_INT CCONV BusTools_BC_MessageBlockAlloc(
  BT_UINT cardnum,              // (i) card number (0 based)
  BT_UINT bufID,                // (i) BC_BUFFER_NEXT or BC_BUFFER_LAST
  BT_UINT count);               // (i) Number of BC data buffer linked to the
                                //     message buffer
```

Typically you would allocate each buffer in a loop as shown below.

```
//New Mechanism to allocate buffers.  Each buffer is allocated separately in loop
//Msg No                    0  1   2  3 4 5 6 7 8 9 10
BT_INT  num_bc_data[]={5, 10, 10,12, 3, 1, 1, 1, 1, 1, 4};  // array with message buffer count
for(index=0;index<11;index++) ///loop through all message needed and specify buffer count
{
  status = BusTools_BC_MessageBlockAlloc(cardnum,BC_BLOCK_NEXT,num_bc_data[index]);
  if(status)
    return status;
}
status = BusTools_BC_MessageBlockAlloc(cardnum,BC_BLOCK_LAST,num_bc_data[index]);
if(status)
  return status;
```

Use BC_BLOCK_NEXT when creating the buffers to link the messages together. Optionally use BC_BLOCK_LAST to link the last message block back to link last message back to the first. The array num_bc_data in the sample code contains a count of the data buffer for each message.

Once the message buffers are allocated you can write the data to the message control buffer and optionally the first data buffer using BusTools_BC_MessageWrite. Use the new function BusTools_BC_DataBufferWrite to fill the data buffers.

```
NOMANGLE BT_INT CCONV BusTools_BC_DataBufferWrite(
  BT_UINT   cardnum,            // (i) card number (0 based)
  BT_UINT   mblock_id,          // (i) BC Message number
  BT_UINT   buffer_id,          // (i) BC data buffer number
  BT_U16BIT * buffer);          // (i) pointer to user's data buffer
```

The following code shows how to fill in the data for a message and all of its data buffers.

```
//Write out command data and first data buffer.
messno = 0;
memset((char*)&bcmessage,0,sizeof(bcmessage));
bcmessage.messno = messno;
bcmessage.messno_next = (BT_U16BIT)(messno + 1);
bcmessage.control = BC_CONTROL_MESSAGE;          // show as a message
bcmessage.control |= BC_CONTROL_CHANNELA;        // Transmit on Channel A
bcmessage.control |= BC_CONTROL_INTERRUPT;
bcmessage.control |= BC_CONTROL_MFRAME_BEG;
bcmessage.mess_command1.rtaddr          = 4;
bcmessage.mess_command1.subaddr         = 4;
bcmessage.mess_command1.wcount          = 4;
bcmessage.mess_command1.tran_rec        = 0;
bcmessage.start_frame                   = 1;
bcmessage.rep_rate                      = 1;
```

```
bcmessage.errorid                       = 0;        // Default error injection buffer (no errors)
bcmessage.long_gap                      = 10;       // 10 microsecond inter-message gap.

status = BusTools_BC_MessageWrite(cardnum,messno,&bcmessage);

//New method to write data buffers.  Write data to each buffer
for(j=0;j<num_bc_data[messno];j++)
{
  for(i=0;i<num_bc_data[messno];i++)   //file in the data buffer
    bc_data[i] = 0x4000 + j;

  BusTools_BC_DataBufferWrite(cardnum,messno,j,bc_data);
}
```

When running multiple-buffer mode, you need to process the data buffer instead of the message buffer. In previous API and firmware versions you would call BusTools_BC_MessageRead to retrieve the Bus Controller data using the message number. When running multiple buffers, use the data buffer address with the new function BusTools_BC_MessageBufferRead to process message data.

```
NOMANGLE BT_INT CCONV BusTools_BC_MessageBufferRead(
BT_UINT cardnum,                // (i) card number (0 based)
BT_U32BIT addr,                 // (i) address of BC Data buffer
API_BC_MBUF * api_message);     // (i) Pointer to buffer to receive msg
```

The code below shows a callback function used in conjunction with BusTools_RegisterFunction to process data from multiple buffers.

```
BT_INT  _stdcall bc_intFunction(BT_UINT cardnum, struct api_int_fifo *sIntFIFO)
{
  API_BC_MBUF bcmessage;
  BT_INT tail, wcount, messno;
  BT_UINT bufaddr;

  tail = sIntFIFO->tail_index;
  while(tail != sIntFIFO->head_index)
  {
    bufaddr = sIntFIFO->fifo[tail].buffer_off;     // Address of the data buffer
    messno = sIntFIFO->fifo[tail].bufferID;        // Message number
    status = BusTools_BC_MessageBufferRead(cardnum,bufaddr,&bcmessage);// read BC data
    if (status)
      return status;
            .
            .
            .
    tail++;
    tail &= sIntFIFO->mask_index;
    sIntFIFO->tail_index = tail;
  }

  return 0;
}
```

Programmable input/output triggers.

Each Mil-Std-1553B interface board has different discrete, RS-485 and triggering capabilities. Refer to the MIL-STD-1553 Hardware Installation Manual to find out board specific capabilities.

V6 firmware supports programmable input and output triggers. Any discrete or 485 channel can be programmed as either an input or output trigger (some boards use dedicated discrete and this section does not apply). Legacy firmware allowed only discrete 7, 8 and 485 channels as input or output triggers. The following code shows how to set up a discrete as an input and output trigger.

```
status = BusTools_V6_SetDiscreteOut(cardnum,7,DISCRETE_GROUND);  //Ground Discrete 7
status = BusTools_SetV6TrigIn(cardnum, DISCRETE,7);
```

The cardnum parameter determines the input trigger channel. The following code sets up an output trigger.

```
status = BusTools_V6_SetDiscreteOut(cardnum,8,DISCRETE_GROUND); //Ground Discrete 8
status = BusTools_SetV6TrigOut(cardnum, DISCRETE,8);  //Enable discrete 8 for output trigger
```

External Clock and Time Tag Reset

The external clock mode was added in V5 firmware. It is fully supported in V6 and BusTools/1553. External Time Tag reset was added in V6. Check the MIL-STD-1553 Hardware Installation Manual to see which boards support these features.

The external clock mode switches timing from the internal clock to an external clock that can run between 1 MHz and 10 MHz. Setup the external clock mode using BusTools_TimeTagMode.

```
status = BusTools_TimeTagMode(cardnum, API_TTD_RELM, API_TTI_DAY, API_TTM_XCLK,
NULL,1000000,0,0);  //Enable external clock for 1MHZ input clock
```

Time tags can be reset to zero (0) on an external input pulse. Enable this feature by calling

```
NOMANGLE BT_INT CCONV BusTools_TimeTagReset(
  BT_UINT cardnum,              // (i) card number (0 - based)card number (0 - based)
  BT_UINT tflag);               // (i) EXT_RESET_ENABLE (1) EXT_RESET_DISABLE (0)
```

Other V6 Changes

The changes described above are the ones that most commonly will effect applications. The following is a list of other V6 changes.

1. BusTools_SetOptions adds an option to generate interrupts on a minor-frame-overflow.

2. There are new interrupt options for minor-frame-overflows and for aperiodic message overflows.

EVENT_MF_OVERFLO	Minor-frame-overflow interrupt
EVENT_LP_MF_OVFL	Minor-frame-overflow on low priority aperiodic list
EVENT_HP_MF_OVFL	Minor-frame-overflow on high priority aperiodic list
EVENT_BC_BSY_OVFL	BC busy on frame start interrupt

3. There is a new Playback file format. The old format, with a .bmd extension, is based on a 48-bit microsecond time tag. The new format for Playback files, created by BusTools/1553-API v8.x, has a .bmdx file extension. Bmdx files use a 64-bit time tag. The time tag resolution can be either micro- or nanoseconds. A header record in the .bmdx defines the resolution. BusTools/1553-API v8.x can run either .bmd or .bmdx files. You cannot use .bmdx files on earlier API versions.
4. Conditional Branch on a memory location now uses 32-bit address and data. Use BusTools_MemoryAlloc32 to allocate a 32-bit address. Use BusTools_MemoryWrite2 to write and BusTools_MemoryRead2 to read the address. Use the RAM32 read/write option in those functions.
5. There have been several changes to discrete, RS485 and trigger configurations. You will need to review the discrete, RS485, and trigger configuration on your specific board. Check the MIL-STD-1553 Hardware Installation Manual for the connector descriptions.

BusTools/1553 Analyzer

If you are running BusTools/1553 Analyzer you will need to update to latest version, v8.0, for V6 support. This version supports operation on V4/V5 firmware as well as V6 firmware boards.

Support and Contact information

If you have any questions or need technical support transitioning from legacy boards to the new V6 boards please contact us immediately.

Contact Information:

Abaco Systems, Inc.
26 Castilian Drive, Suite B
Goleta, CA 93117
Main +1 805-965-8000 or +1 805- 883-6101
Support +1 805-965-8000 or +1 805- 883-6097

support@abaco.com (email)

https://www.abaco.com/products/avionics

Additional Resources

For more information, please visit the Abaco Systems website at:

www.abaco.com