Electric Brakes ABS Device Profile & Test Plan

Real Time Objects & Systems, LLC
Proprietary & Confidential
Editor: Don Pieronek
March 16, 2021

Partial Reprint

The following is a partial reprint of a product design document for the Electric Anti-lock Braking System (EABS) Controller. This document is intended to convey how the Extensible Realtime Application Environment (XRAE) is utilized to implement control devices. Although XRAE has been utilized since 2002 this patented[1] control device technology has been licensed to multiple vendors in various industries under mutual non-disclosure agreements and thus their related device profiles and technologies could not be published. Real Time Objects & Systems, LLC (RTOS) is publishing a portion of the RTOS owned design document to convey how XRAE assets and architecture are utilized. The assets are clarified by inserting numerous “NOTES” not contained within a typical Device Profile document.

Prior to reading this document download a copy of “UAP_DEBUG_MSG.pdf” that includes the portion of the User Application Program (UAP) being described. A UAP is an “iconic software schematic” showing how the firmware internal to the EABS Controller works, often referred to as a “Programmable Logic Controller (PLC) on steroid” except XRAE is utilized in any control device, including IO Modules, User Interfaces, Network gateways, sensors, actuaotrs and so forth.

[1] US Patent #7,908,020 and US Patent #7,554,560.

Table of Contents

1. DEFINITION OF TERMS:

For those with no "domain knowledge" relative to the towed trailer market the following definition of terms used in the attached description is provided;

Electric Trailer Brakes: Brakes on the majority of travel trailers are actuated by electric brakes versus hydraulic brakes. Each drum brake contains a magnet, which when electric pulses are applied to the magnet it is attracted to the rotating brake drum. The friction between the magnet and the rotating brake drum applies a force to the mechanical brake shoes, actuating the brakes.

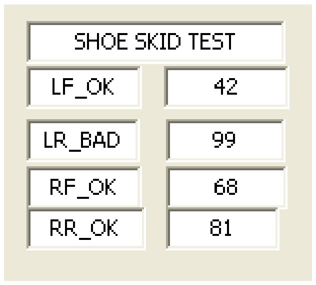
Blue Wire: The electric pulses applied to a brake magnet, in the USA, originate from a "Brake Controller" within the tow vehicle, where the color of this brake wire is "blue", and thus referred to as "blue wire". The majority of brake controllers provide electric pulses at a rate of 300 pulses per second and where the "on time" of these brake pulses vary from 0% (off) to 99% (full on).

2. USER APPLICATION PROGRAMS

2.1 UAP_PROD_DEBUG

Numerous debug messages are contained within the control device, where the content of the message sent is determined by the value contained within a specific instance (PARAM_DEBUG_MSG) of the Parameter Object class within the EABS Controller.

For clarity, an Electric Anti-lock Braking System (EABS) Controller supports over twenty diagnostic screens. When a new screen is selected on a User Interface (UI) device the selected screen number is broadcast in a "Command Request Message", and assuming the EABS Controller produces predefined data for the selected screen it then sends the predefined data in the Debug Message. For example, the following prototype screen is used to determine the condition of each brake[2], where this screen displays the braking force applied (0-99%) to each brake magnet and captures the applied duty cycle at which each wheel skids.



SHOE SKID TEST	
LF_OK	42
LR_BAD	99
RF_OK	68
RR_OK	81

During the test, in this screen the Left Rear (LR_BAD) wheel never skids (99%) and thus needs servicing and the Right Front (RF) and Right Rear (RR) wheels could use some adjustment to skid at a similar brake force.

> NOTE: This operational description, extracted from the Device Profile document, for this User Application Program (UAP) contains numerous "NOTE" sections to provide detailed implementation information using Extensible Real Time Application Environment (XRAE) assets. Those familiar with the logic paradigm and instructions (methods) may choose to skip the "NOTE" sections. The XRAE "architecture" specifies multiple

[2] Patent pending.

> assets, including; a development process, programming rules, naming conventions, documentation standards, object descriptions, source code, test plans, training materials and so on utilized to facilitate rapid product development and simplification of product support, achieved by massive amounts of asset reuse. One of the core assets for any XRAE control device includes the "software schematics" (UAPs), which graphically document product design and match the product implementation exactly.

The operational descriptions of each sheet within the "UAP_PROD_DEBUG.doc" Word document are described in the following subsections. XRAE implements ALL of a device's functionality, including functionality typically called the "operating system", with XRAE objects. This removes the "mystery" of how a device works and in the case of communications UAPs, allows a control device to efficiently implement any desired network behavior and facilitates capturing the underlying operation with visible iconic programs.

2.1.1 SHEET 1: Power Up Initialization

As with the majority of other UAPs, this UAP is called from UAP_MAIN(). During the "first program scan" a TRUE from the left rung is passed through the normally closed instantaneous contact of the TR FIRST SCAN instance of a Timed Delay Relay class thus passing a TRUE to the "conditional execution" block's ENGINE_IfTrue() method, where the logic within the "dashed conditional execution block" is then executed. After the first program scan TR FIRST SCAN is active, thus the normally closed instantaneous contact will be open and this conditional execution block will not be executed again.

> NOTE: An instance of the Timed Relay Object (TR_FIRST_SCAN) is activated at the end of the execution of all the UAPs within every XRAE control device to provide visible and intuitive initialization logic. An "on delay timed closing contact" is utilized when UAP logic actions must be delayed, primarily where a hardware driver object takes time to initialize its functionality before its methods can provide valid results. See UAP_Initialize() where XRAE devices initialize all object classes upon waking up.

The first scan conditional logic sets the CAN Identifier used to produce the debug message where this UAP calls PARAM_GetOutput() method of PARAM_MACID instance of the Parameter Object Class to retrieve the subnet "address". The UAP then calls ADDER_SetInputA() method of ADDER_REUSE instance of the Adder Object Class. The constant value 0x640 is passed to ADDER_SetInputB() method of the same adder object. The UAP calls ADDER_GetSum() method of the same adder object instance to retrieve the sum which then calls the PRODUCER_SetNetID() method of PROD_DEBUG instance of the Producer Object Class, thus setting the CAN Identifier for this message.

> NOTE: This XRAE UAP is intuitively clear how the value of the CAN Identifier is determined by utilizing various XRAE object classes. The resulting value could have been set directly with a constant value, but since this control device enables changing the device address for integration of the EABS Controller with other control devices on the same subnet a Parameter Object instance is used. Changing this parameter instance will change the CAN Identifier of all other messages supported by this control device as well. Since the Parameter Object class already supports the ability to; 1) edit any instance, 2) limit the range of values that may be set (0 to 63 for this instance), 3) supports the network interface to edit and save the parameter value to non volatile memory, and 4) can optionally support a "tag name" for each instance, why replicate this functionality (add code space) in the Producer Object Class? Many XRAE devices re-use core UAPs from another control devices, like UAP_SET_PARAM(), since that tested UAP already supports the editing of parameter instances from a network. This re-use saves development, test, and documentation time of an XRAE control device as well as provides a common configuration tool interface for all control devices to edit each parameter instance from the supported network thus achieving an XRAE mantra; "Do the same thing the same way, always!"

The constant value 0x08 then sets the default message length by calling PRODUCER_SetLength() method of the PROD_DEBUG instance of the Producer Object Class.

> NOTE: XRAE utilizes a pointed icon to indicate the setting of the value passed into any object to a specific data value. XRAE also utilizes the "thick line" to indicate a "data value" versus a "Boolean value" which uses a thin line.

The constant value FALSE is passed to TIMEDRELAY_OnDelayExecute() method of the TR_DEBUG_INTERVAL instance of the Timed Relay Object class to ensure the timer is off and then the constant value 10 is passed to

TIMEDRELAY_SetPreset() method of the TR_DEBUG_INTERVAL instance of the Timed Relay Object class, setting the timer preset to ten milliseconds.

> NOTE: The XRAE Timed Relay Object class sets the accumulated time property with the timer preset property when transitioning from the TIMER_IDLE state to the TIMER_TIMING state. Thus changes in a Timed Relay preset's value made while the timer is running are ignored until the previous timer preset value has expired or until TIMEDRELAY_ OnDelayExecute() method is passed a FALSE, returning it to the TIMER_IDLE state. The Timed Relay Object supports on delay and off delay timer methods as well.

Although various approaches may be used, this control device selects which one of the fifteen predefined messages is produced by setting the position of the PARAM_DEBUG_MSG instance of the Parameter Object Class. When installed on a trailer the desired screen number is sent by a UI device in the Command Request Message, which is consumed by UAP_CAN_CONS_ CMD_MSG() in the EABS Controller. When the screen number value is set to zero the production of the Debug Message is disabled.

> NOTE: The receipt of a Command Request Message by UAP_CAN_ CONS_ CMD_MSG() within the EABS Controller results in the production of a Command Response Message from UAP_CAN_PROD_CMD_MSG() within the EABS Controller, providing a request/response behavior for Command Messages. The value of a byte within the Command Request Message determines if, and when, a Debug Message is produced by the EABS Controller.

In this UAP a PARAM_GetOutput() method reads the current value of the PARAM_DEBUG_MSG instance of the Parameter Object class and passes that value to COMPARATOR_SetInputA() of COMPARATOR_REUSE instance of the Comparator Object Class. A constant value of zero is then passed to COMPARATOR_SetInputB() to the same comparator instance and then COMPARATOR_AequalsB() method is called, which passes provides either a TRUE or a FALSE, and then calls BRANCH_SetField() method of instance 1 of the Branch Object class, setting its value to either TRUE or FALSE.

> NOTE: XRAE UAPs often reuse the same instance of the same object class multiple times throughout all of its UAPs. This is common for objects like the Comparator, Adder, Subtractor, Divider, Multiplier, Data Branch,

Branch and so on to save cpu data space. To make this obvious "REUSE" is usually appended to the end of the instance name and where the icon is modified to make reuse of the instance elsewhere even more intuitive, in this case using "double lines" on either the icon, the icon interfaces and so on.

A TRUE from the left rung is passed through the TR_CMD_MSG_TIMEOUT instance of TIMEDRELAY_OnDelayNOExecute() method of the Timed Relay Object class, and if a Command Request Message has not been received within the timers preset value the normally open timed closing contact would be closed, passing a TRUE to the BRANCH_Execute() method of instance 1 of the Branch Object Class. If either the Comparator Object or the Branch Object has passed a TRUE into the Branch Object class, a TRUE will be passed to the ENGINE_IfTrue() method resulting in the execution of the conditional control block, and if FALSE the conditional control block shall be skipped.

NOTE: The TR_CMD_MSG_TIMEOUT instance of the Timed Relay Object class is contained within UAP_CAN_CONS_CMD_MSG() where an On Delay timer is reset each time a Command Request Message is received (CR_CMD_MSG_RECVD) by the EABS Controller. As long as a message is being received from an installed UI control device the communications functionality within the EABS Controller is executed and if a message has not been received within the preset time interval of the TR_CMD_ MSG_ TIMEOUT timer, the timer will expire and cyclic message production, if currently active, will cease. Thus by merely connecting a UI control device to the subnet, communications within the EABS Controller is enabled, providing system "plug and play" functionality. This network behavior is a result of providing to a peer to peer control where a single point of failure in on control device does not shut down control being provided by other peer control devices on a network. It should be noted the UI control device may provide screens for other control devices as well.

Within this conditional execution block a constant value of FALSE is passed to the PROD_DEBUG_MSG instance's PRODUCER_Enable() method, which will transition the producer instance to the PROD_IDLE state and updates the respective producer's status relays accordingly. A constant value of FALSE is then passed to the TIMEDRELAY_OnDelayExecute() method of the TR_MSG_DEBUG_ INTERVAL instance of the timed relay object placing it in the TIMER_IDLE state and clearing its accumulated time property. Lastly the ENGINE_Return() method is called, which terminates execution of the UAP and returns to the calling UAP_MAIN(). This conditional logic is included to save cpu execution time of the remainder of this UAP when a message is not selected by a UI or other control device on the TCAN network.

NOTE: During product development the PARAM_DEBUG_MSG instance's output value is set with an explicit message by PC test software from the RTOS Object Test Tool versus within a Command Request Message sent by a User Interface (UI). During development, for real time monitoring purposes, unique debug messages are implemented where selected properties of various object instances within any UAP being tested are sent. These messages are cyclically produced at a very high rate and consumed by the RTOS Monitor Tool which saves them to a file. The contents of these files are then plotted using an excel spreadsheet to analyze the operation of the respective UAP, often sending the message every program scan. These same messages are often captured during vehicle test scenarios where the saved real time messages are then sent to a "trailer simulator module" which stimulates the EABS Controller with the same vehicle test data during bench testing, thus replicating the actual captured trailer data to identify and resolve UAP defects. Actual "system tests" often reveals test scenarios not previously envisioned and thus the ability to capture system test data, and then repeat the test in the lab until the root cause is identified and corrected in invaluable. OEM Test tools may also select the debug message to be produced where this test software performs advanced system diagnostics when debugging trailer braking problems.

2.1.2 SHEET 2: Produced Message Selection

A TRUE from the left rung is passed through the CR_DEBUG_ERROR instance's RELAY_NCExecute() method and assuming a communications error has not been detected by the Producer Object this UAP thus applies a TRUE to the PRODUCER_Enable() method of producer object instance, which updates the producer object's state and it's status coils.

NOTE: The first call to the PRODUCER_Enable() method (sheet 1) containing a TRUE after the application of power initializes the instance of the producer object, setting its status relays to FALSE, clearing the message data values and transitioning it from the PROD_NON_EXISTENT state to the PROD_IDLE state. On subsequent scans, when passed a TRUE, the enable method updates the status coils based upon the current state of the message, where these status relay contacts are utilized by this and other

UAPs. XRAE object classes often utilize instances of other object classes, primarily the Relay Object Class, which in this case is used to indicate the status of each Producer Object instance. Why? The code space required to support the normally open and normally closed contacts, set the relay coil status, retrieve the coil status from the network, and so forth already exist in the Relay Object Class so there is no reason to replicate this functionality (code) within other object classes, such as the Producer Object Class! Additionally, assume the CR DEBUG PENDING relay is NOT referenced within any UAPs within this control device, within XRAE this instance declaration can be deleted from the project, saving the respective RAM memory containing its state. Traditional "object languages" do not allow this optimization as they "inherit" all object functionality and properties (memory space) for all object instances. See "UAP_Relay.h" for instance declarations. See "XRAE Programmer's Guide" for methods, interfaces and operation of supported object classes.

If Relay Object instance CR_DEBUG_SENT is TRUE then the RELAY_NOExecute() method passes a TRUE to the PRODUCER_Release() method, where if the producer instance is in the "PROD_SENT" state it will transition to the "PROD_IDLE" state, thus enabling the production of another message. If the value passed is FALSE no change to the producer instance state occurs.

NOTE: Why would a UAP control the transition between the states of each instance of the Producer Object Class? State control provides synchronization of control logic and data between the UAPs in different control devices across the network and enables any desired network behavior to be created; polled, cyclic, change of state, send only, receive only and so forth based upon the state of the process being controlled. For example; Assume parameter data is being sent from EABS Controller to a UI that is editing the value of a parameter instance, where various data sent from the EABS Controller must first be received by the UI in the tow vehicle, perhaps through a gateway, where the UI may then change the pending value being edited and sent within the response message. Once the response message is received by the EABS Controller this triggers the EABS Controller to send the next message. This UAP operates in either; the command/response mode, the cyclic mode or is off depending upon the message selected. In XRAE there is no need to limit the network behaviors

> required to solve a control problem (like DeviceNet[3] and all other control networks) and thus allows multiple network behaviors to be defined for one message in a visible and thus intuitive UAP drawing[4]. Additionally, described below, the content of each message is clearly visible in the UAP, and thus removes the "mystery" of message contents and how a specific configured network dialog works. All other control architectures require product specific functionality, like message types supported, content of messages and so forth to be encrypted in some product support file imported by some proprietary support tools. Another XRAE mantra; "You can't fix what you can't see and XRAE shows you everything!"

This message provides two different behaviors that will change based upon the message selected; cyclic message production (sending message at timed interval) and polled response (sending response when a request message is received). Majority of the debug messages are cyclic and two require synchronized polled command/response operation. Thus cyclic behavior occurs for every Debug Message except for; 1) parameter editing messages, 2) retrieval of fault codes from a fault queue, or 3) when message production is stopped when message zero is selected.

The following logic will result in cyclic message production on all selected messages except for the parameter editing message or the fault code retrieval message. A TRUE is passed to the timed closed contact of TR_ DEBUG_ MSG_ INTERVAL by calling its TIMEDRELAY_OnDelayExecute() method, and if the timer has expired, passes a TRUE to RELAY_NCExecute() method of CR_ EDIT_ PARM_ SCRN, and if its coil is FALSE, passes a TRUE to RELAY_NCExecute() method of the CR_FAULT_SCRN relay, and if its coil is also FALSE, passes a TRUE to BRANCH_SetField() method of instance 1 of the Branch Object Class, setting its value to TRUE. If the timed contact was not closed or if either of the normally closed relay contacts were open a FALSE would have been passed to the BRANCH_SetField() method of instance 1 thus setting its value to FALSE.

> NOTE: The XRAE logic execution paradigm evaluates logic from left rung to right rung, when encountering a branch close icon instance of the Branch Object and additional connections to left of a branch close icon exist then, in this case, execution returns to left rung and evaluates logic toward right rung until a branch close is encountered and repeats until an unevaluated branch

[3] DeviceNet is a trademark of the Open DeviceNet Vendors Association.
[4] US Patent #7,908,020 and US Patent #7,554,560.

close does not exist and then passes the last evaluated value of the branch close instance to the next logic to the right.

NOTE: Every XRAE device supports three core Branch Object methods; BRANCH_SetField(), BRANCH_Execute() and BRANCH_GetField(). The BRANCH_SetField() sets the branch instance to the value it receives. BRANCH_Execute(), if it is passed a TRUE, sets the branch instance to TRUE. BRANCH_GetField() retrieves the current value of the branch instance. Instances of this object class are constantly reused throughout UAPs except when an instance zero is often reserved for special purposes (like the status of left rung in user programmable control devices). See "RTOS Programmers Guide" for operation of object classes.

The following logic will result in polled request/response message production on two selected messages; the parameter editing message or the fault code retrieval message. A TRUE is passed to RELAY_NOExecute() method of CR_EDIT_PARAM_SCRN where it will either pass a TRUE or a FALSE depending upon the state of its coil to BRANCH_SetField() method of instance 2 of the Branch Object class. A TRUE is passed to RELAY_NOExecute() method of CR_FAULTS_SCRN where it will either pass a TRUE or a FALSE depending upon the state of its coil to BRANCH_Execute() method of instance 2 of the Branch Object class. If either of the normally open contacts is closed a TRUE would be returned from the BRANCH_Execute() method of instance 2 else a FALSE would be returned. The resulting value is passed to RELAY_NOExecute() method of CR_DEBUG_MSG_RECVD, and this contact will be closed if a message has been consumed by UAP_CONS_CMD_MSG() or will be open if a message has not been received. The value returned from CR_DEBUG_MSG_ RECVD is passed to BRANCH_Execute() method of instance 1 of the branch object, and if either value passed to the branch has been TRUE a TRUE will be returned else a FALSE will be returned and passed to RELAY_NOExecute() method of the CR_DEBUG_MSG_IDLE instance. If the producer instance is in the IDLE state and if a TRUE was passed to the normally open contact, the value is passed to the ENGINE_IfTrue() method of the conditional execution block. If the value is FALSE all logic within the conditional execution block is skipped and if TRUE the logic within the conditional block is executed. See "UAP_CONSUME_CMD_MSG" for logic which sets CR_EDIT_PARAM_SCRN and CR_FAULTS_SCRN when the respective screen is selected by a UI device.

In summary the conditional logic just described will pass a TRUE to the execution block if either;

- the Edit Screen or Fault Screen are NOT selected and if the TR_DEBUG_MSG_INTERVAL time has expired and if a message is NOT currently being sent (CR_DEBUG_MSG_IDLE is closed) or
- the Edit Screen or the Fault Screen is selected, a Command Request Message has been received and a message is not currently being sent (CR_DEBUG_MSG_IDLE is closed).

If screen number zero has been selected, per sheet 1, no message is produced since this UAP is not executed.

Prior to setting the message data of any of the supported messages the instance of the Producer Object Class is set with a constant value PROD_DEBUG_MSG by calling the PRODUCER_SelectInstance() method.

> NOTE: Why is the select instance method required? XRAE patents[5] create a control device architecture where; "the software interface to every method (function) of every object class is the same" allowing the values returned from any method of any object class to be passed into any method of any other object class providing "software plug and play" functionality. Why? It allows interconnecting of real time object icons without product dependent "custom compilers" and provides portability of objects/UAPS to any microprocessor in any control device in any market which supports XRAE (rapid product development). This standardized function call interface in XRAE devices utilizes two variables passed to and from each method; typically the instance value and the exchange (data) value.
>
> For those interested in "what the code looks like", if you looked at User Application Program (UAP) code written using the 'C' language it would contain a sequence of redundant function calls that contain one to three of the following "generically named" instructions;

```
Set GLOBAL_Logic;
Set GLOBAL_Instance;
Function call();
```

> A "function call" (instruction) always exists. If the value returned from a prior function call (GLOBAL_Logic) is being passed into the next function call (Set GLOBAL_Logic) it is not included in the next XRAE function call

[5] US Patent #7,908,020 and US Patent #7,554,560.

and 99% of the time the instance (Set GLOBAL_Instance) of the object class being called is included. A “code” example of the logic passing data previously described on sheet 1 follows;

```
GLOBAL_Instance = PARAM_MACID;
PARAM_GetOutput();

GLOBAL_Instance = ADDER_REUSE;
ADDER_SetInputA();
```

A code example for the Boolean logic previously described on sheet 2 follows;

```
//===============================================================
// When message production behavior = CYCLIC
//===============================================================
GLOBAL_Logic.exchange = TRUE;
GLOBAL_Instance = TR_DEBUG_MSG_INTERVAL;
TIMRELAY_OnDelayNOExecute();
GLOBAL_Instance = CR_EDIT_PARAM_SCRN;
RELAY_NCExecute();
GLOBAL_Instance = CR_SCREEN_FAULT;
RELAY_NCExecute();
GLOBAL_Instance=1;
        BRANCH_SetField();
//===============================================================
// When message production behavior = POLLED
//===============================================================
GLOBAL_Logic.exchange = TRUE;
GLOBAL_Instance = CR_EDIT_PARAM_SCRN;
RELAY_NOExecute();
GLOBAL_Instance=2;
        BRANCH_SetField();
GLOBAL_Logic.exchange = TRUE;
GLOBAL_Instance = CR_SCREEN_FAULT;
RELAY_NOExecute();
GLOBAL_Instance=2;
        BRANCH_Execute();
GLOBAL_Instance = CR_IMPL_1_RECVD;
RELAY_NOExecute();
GLOBAL_Instance=1;
        BRANCH_Execute();
GLOBAL_Instance = CR_IMPL_test_PROD_IDLE;
RELAY_NOExecute();
if ( GLOBAL_Logic.exchange)
{
        //conditional code here
}
```

In the rare cases where the value returned from the prior function call (GLOBAL_Logic) is passed to multiple function calls of the same instance of the same object class it is possible the UAP could solely include a series of function calls. This scenario does not exist is this UAP.

When GLOBAL_Logic is being set to an initial value at the start of a logic rung or with a constant value (sheet 1) a code example follows;

```
GLOBAL_Logic.exchange16 = 0x08;
```

```
GLOBAL_Instance = PROD_DEBUG_MSG;
PRODUCER_SetLength();
```

The internal executable code within an XRAE programmable control device is either an interpreted list or assembly code, which looks different then the prior examples, but where the "network interface dialog" to read and write the program is the same across all programmable devices, where this revision of this control device is not network programmable. XRAE allows a single programming tool to; retrieve the methods supported by a control device, retrieve the function call addresses for these function calls, to retrieve the number of instances of each object class supported by a control device and other criteria which then enables creation and downloading of programmable UAPs. This control device does not currently support this functionality but future revision of EABS Controller will likely enable the creation and download of a unique Debug Message.

In keeping with the XRAE matra; "Do the same thing the same way, always" in the very rare case where more than two variables are required for a specific method call XRAE could utilize a third generic variable, but since this seldom occurs a method was added to some object classes requiring another variable (Producer & Consumer Objects) where the object instance is selected with an instance selection method, where the GLOBAL_Instance interface variable is used for other purposes. In the case of setting and getting data values from network messages the instance of the producer instance or consumer instance are first selected (sheet 2) with a unique method call and then the byte offset into an array of message bytes is selected utilizing the GLOBAL_Instance variable as follows;

```
GLOBAL_Logic.exchange8  = PROD_DEBUG_MSG;
PRODUCER_SelectInstance();

GLOBAL_Instance = DB_MPW_SENSOR_LF;
DATABRANCH_GetField();
GLOBAL_Instance = 0;            //byte offset
PRODUCER_SetMsgWord16();

GLOBAL_Instance = DB_MPW_SENSOR_LR;
DATABRANCH_GetField();
GLOBAL_Instance = 2;            //byte offset
PRODUCER_SetMsgWord16();
```

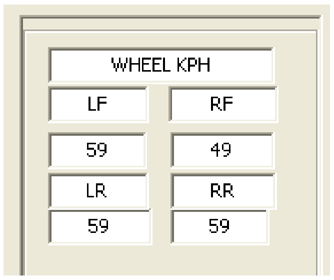
The setting of the data of all other Debug Messages follow similar operations to the one previously described. Individual descriptions of each of these messages are skipped as they are obvious by viewing the UAP drawing and/or the code within each message case.

2.1.3 SHEET 3 : Eight message setting sheets removed

The content of the operational descriptions for sheet 3 to sheet 10 within the EABS Controller Device Profile document have been removed as they offer no additional value to this XRAE overview document.

2.1.4 SHEET 4: Wheel KPH Message

In the EABS UI multiple screens require one or more of the wheels speeds and one or more of the brake magnet settings which are provided by the debug message. One example is the "Wheel KPH" message which displays the four wheel speeds. This message may be used identify low tire pressure or wheel speeds during braking.



WHEEL KPH	
LF	RF
59	49
LR	RR
59	59

As with other messages, this message is updated in Case 10 of the Engine Object switch instruction. This message retrieves its values from instances of the Databranch Object Class, which is a UAP icon used to maintain and alter a data variable.

> NOTE: The Databranch Object Class is a variable that can contain either a; bit, byte, word or dword (union of data types). Unlike the traditional PLC paradigm; XRAE devices display and control "data flow" using intuitive UAPs. A thick line in a UAP drawing indicates a data type other than a Boolean value. Within XRAE data may or may not flow through a normally open or normally closed contact, depending upon its state, thus controlling data flow. A circle with a dot (arrowhead), a "+" (arrow feathers) and the connection location to the Databranch icon indicate the method called. Connection to the left side of the icon indicates a DATABRANCH_ SetField() method, which sets its initial value. Connection to the right side of a the icon indicates a DATABRANCH_GetField() method, which reads its value. Connection to any other location from the left side of the icon indicates a DATABRANCH_Execute() method, where when the value applied "exceeds its current value" it sets the value, but if value applied is

> "less than the current value" it is ignored. Every XRAE control device uses these three methods where a couple other Databranch methods are used in a minority of other control devices, but not in this control device.

This UAP calls DATABRANCH_GetField() to instance DB_KPH_LF to retrieve the wheel speed of the Left Front (LF) wheel in Kilometers Per Hour (0-255) and then calls PRODUCER_SetMsgByte() where the instance variable contains the byte offset into the message where the value is written, in this case the offset is zero (first byte). This sequence is repeated for the other three wheel speeds written into the next three message bytes at offsets 1, 2 and 3. The UAP then does a DATABRANCH_ GetField() call to instance DB_BRAKE_SETTING_LF to retrieve the brake pulse width modulation (PWM) setting being applied to the left front brake magnet, which in this control device this variable internally has a range of zero to 4095 clock ticks. To fit this range into a byte this value is divided by sixteen to provide a range of zero to 255 to the UI. Rather than using the Divider Object Class this XRAE control device supports that faster execution method ENGINE_ BitShiftRight() method saving cpu execution time and UAP code space.

> NOTE: Since this control device contains one cpu, it only contains one instance of the Engine Object Class and thus assume instance zero is selected upon application of power for this object class, where the XRAE control variable "instance" contains the number of bits to shift GLOBAL_Exchange to the right when ENGINE_BitShiftRight() is called. This control device contains a single CPU and thus the discussion of multiple instances of the Engine Object class shall be ignored.

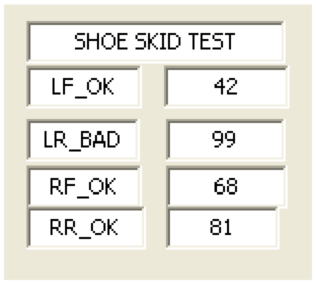
> NOTE: In 2002, when XRAE was invented, patented and implemented two variables were used for the "common object interface"; initially named GLOBAL_Logic and GLOBAL_Instance. This is the minimum number of variables to implement the software architecture and these two variable names were used for clarity, versus using "generic names" like GLOBAL_Variable_1 and GLOBAL_Variable_2 in architecture descriptions and code. When new methods were created, a third variable was required, to maintain compatibility with the XRAE architecture mantra; "Do the same thing the same way, always" it was decided to add a method to set any additional variables thus maintaining the common object interface and portability of UAPs. In retrospect, for clarity in viewing code, the same naming decision would still be made today. However, newer XRAE object classes utilize ENGINE_SetGlobal2() method thus not requiring a new method for object classes that require an additional variable while

maintaining backwards compatibility with existing object classes and maintaining UAP reuse.

The ENGINE_Break() indicates the end of the conditional execution of the message data setting logic.

2.1.5 SHEET 5: Brake Shoe Skid Test

This message provides status information to update the Brake Shoe Skid Test[6] screen, where each brake is sequentially activated with a brake signal ranging from 0 to 99% where when the activated wheels' speed drops by 10 KPH the blue wire brake's duty cycle is captured and displayed in the right column. If the wheel speed successfully dropped by 10 KPH and the blue wire value was less than 99% then in this example "LF_OK" is indicated. If blue wire reaches 99% and wheel speed did NOT drop by 10 KPH then in this example "LR_BAD" is indicated. See the respective UI manual[7] for a detailed operational description of this test.



SHOE SKID TEST	
LF_OK	42
LR_BAD	99
RF_OK	68
RR_OK	81

Figure 1: Shoe Skid Test Screen

A traditional view of the content of the Shoe Skid Message follows;

Byte	Protocol Layer	Data Type	Description	Values
x	Media Access		CAN Identifier [0x64B]	See
x	Transport		Length	8 bytes
0	Application Data	UINT8	DB_BRAKE_SETTING_LF	0-255
1	Application Data	UINT8	DB_BRAKE_SETTING_LR	0-255
2	Application Data	UINT8	DB_BRAKE_SETTING_RF	0-255
3	Application Data	UINT8	DB_BRAKE_SETTING_RR	0-255
4	Application Data	UINT8	DB_KPH	0-255
5	Application Data	UINT8	DB_BLUE_AVG_DUTY_CYCLE	0-255
6	Application Data	UINT8	SELECTOR_SHOE_SCAN	0-6 (Selector Switch position value) 0 = Off 1 = Check Shoe LF 2 = Check Shoe LR 3 = Check Shoe RF

[6] Patent Pending functionality
[7] "PC Based Configuration & Monitoring Tool", Preliminary Prototype, Rev 0.21

Byte	Protocol Layer	Data Type	Description	Values
				4 = Check Shoe RR 5 = Waiting Blue Wire Release 6 = Check Shoes Done

7	Application Data	BOOL[8]	BIT7 = CR_FAIL_SHOE_SKID_RR
			BIT6 = CR_FAIL_SHOE_SKID_RF
			BIT5 = CR_FAIL_SHOE_SKID_LR
			BIT4 = CR_FAIL_SHOE_SKID_LF
			BIT3 = TR_MODULATE_RR
			BIT2 = TR_MODULATE_RF
			BIT1 = TR_MODULATE_LR
			BIT0 = TR_MODULATE_LF	

x	Transport		16 bit CRC	See CAN Specification

Table 2-1: Shoe Skid Test Message

When this screen is selected, UAP_DIAGNOSE_SHOES() is conditionally executed within the EABS Controller to provide the desired brake test functionality and the object properties needed to perform the brake shoe test. Within the UI the UAP_DIAGNOSE_SHOES_SCRN() is conditionally executed when this screen is selected to process the prior Debug Message values and update the screen with the respective values.

> NOTE: XRAE technologies are utilized in all control devices, where a User Interface device is considered to be a control device as well. A UI control device includes additional object classes that update display screens, monitor keypads and so forth.

When this screen is selected Engine Object "Case 11" is executed. The DATABRANCH_GetField() to the DB_BRAKE_SETTING_LF instance retrieves the current value of the blue wire duty cycle currently applied to the Left Front Brake Magnet, which internally has a value range of 0-4095 clock ticks. This value is divided by sixteen to provide a range of 0-255 to the UI where the ENGINE_BitShiftRight() is called with a value of 4 as this method executes faster than utilizing an instance of the Divider Object class and consumes less UAP code space as well. The resulting value is then passed to the PROD_SetMsgByte() method to byte offset zero of the CAN_PROD_DEBUG instance (selected on sheet 2) of the Producer Object Class. Similar logic occurs for the next three Databranches for the three other wheels except setting the next three bytes (1, 2, and 3) in the message.

The DATABRANCH_GetField() to the DB_KPH instance retrieves the trailer speed value and then PROD_SetMsgByte() method to byte offset four of the CAN_PROD_DEBUG instance thus setting the fifth message byte.

The DATABRANCH_GetField() to the DB_BLUE_AVG_DUTY_CYCLE instance retrieves the currently applied blue wire duty cycle value and then PROD_SetMsgByte() method sets byte offset five of the CAN_PROD_DEBUG instance of the Producer Object Class.

The SELECTOR_GetPosition() method of the SELECTOR_SHOE_SCAN instance of the Selector Object class retrieves the current position of the selector switch, which indicates which wheel is currently being tested and then PROD_SetMsgByte() method to byte offset six of the CAN_PROD_DEBUG instance of the Producer Object Class is called.

> NOTE: XRAE allows the bits of a byte within a message to be set, where in this case a single relay or timer contact are required to provide the desired information where combinational logic is often required in some messages to provide a single Boolean value. The Producer instance was selected previously (sheet 2) via a call to PROD_SelectInstance(). Setting bits within a selected byte of a selected instance of the Producer Object requires the designation of the bit offset within a byte. Similar to XRAE decisions made previously and to maintain backwards compatibility and a "plug and play" object interface, a unique method call for each bit offset from zero to seven is utilized.

A TRUE is passed to CR_CHECK_SHOE_LF instance of RELAY_NOExecute() method which passes the resulting value to PROD_SetMsgBit0() to byte offset 7, setting the bit to the value provided. This logic repeats for the next six relay contacts and remaining bits of message byte offset 7.

The ENGINE_Break() indicates the end of the conditional execution of the message setting logic.

2.1.6 SHEET 6: One sheet removed

One sheet of message setting logic has be removed.

2.1.7 SHEET 7: Message Production

After the message values are conditionally set, this sheet passes a TRUE to the PROD_DEBUG_MSG instance's PROD_Send() method, resulting in the producer instance transitioning from the PROD_IDLE state to either the PROD_SENDING or PROD_PENDING state. If another message is currently being sent by the CAN Driver Object class the producer instance would enter the PROD_PENDING state versus the PROD_SENDING state until the prior message is sent.

NOTE: The Producer Object Class, its properties and methods, are identical for multiple networks; J1939, J1850 VPW, J1850 PWM, J1708, TCAN, DeviceNet[8], DirecLink[9], Bluetooth, USB and so on where different driver object classes for these specific networks interface to selected instances the Consumer Object or Producer Object class within a control device that support multiple networks. Internal UAPs then interface to the Producer and Consumer Objects providing UAP portability across different networks. Although the majority of XRAE objects are "drag and drop" from control device having different microprocessors, objects that interact with hardware (Driver Objects) or objects that interface to some driver objects (Producer & Consumer Objects) may require some modification when reusing object code across different types of microprocessors.

NOTE: XRAE technologies may utilize any language to implement its methods, where RTOS utilizes the 'C' language as it is extremely portable across different microprocessors. If XRAE is written in an "object oriented language" such as C++ then if one UAP of any instance of one object class utilizes one of the objects status relays of the Producer Object class, for example, then every instance of the producer object class would contain a property for the specific relays status, whether the other instances of that object class utilize the specific status relay or not, consuming CPU memory for unused property values. More specifically, the UAP described here does not reference the CR_DEBUG_MSG_PENDING or CR_DEBUG_MSG_SENDING status relays of the PROD_DEBUG_MSG instance of the producer object class. Since these relay instances are not used, memory is NOT allocated for these instances within UAP_RELAY.h.

NOTE: XRAE coding conventions utilize the object's name for the file containing the object's executable code, in this case, using the "*objectname*.c". The respective "*objectname*.h" file utilizes the same object name. The object instance declarations are contained within a file matching the respective objects name except preceded by "UAP_ *objectname*.h". All UAP code is named for the respective functionality of the UAP and

[8] DeviceNet is a trademark of the Open DeviceNet Vendors Association (ODVA)
[9] DirecLink is supported by the Tuson DirecLink Brake Controller.

preceded by "UAP*_functionality*.c" in its name. Thus when creating a new product, the "*objectname*.c" files and the "*objectname*.h" files are dragged and dropped into the new project file and then "built" to resolve any minor declaration issues as the contents of these two files seldom change with each new control device. Next common core "UAP*_functionalityName*.c" files are individually dragged and dropped from an existing project containing similar core functionality. These common core "UAP_ *functionality Name*.c" files provide common functionality requiring minimal modifications, like UAP_Initialize.c, UAP_BrownOut.c, UAP_PowerUp.c, UAP_NV_Save.c, UAP_Faults.c and so forth. After dropping each of these into the project, a build is performed and unresolved object instance names are added to each "UAP*_objectName*.h" file. Then the new "UAP*_functionalityName*.doc" files (UAP drawings) are incrementally created and tested. Assuming a fast prototype of the initial hardware platform is available, the core functionality of an XRAE based product reuses "*objectname*.c" and "*objectname*.h" files with little to no modification, reuse many of the common core "UAP*_functionalityName*.c" files with minor modification and thus have core control device limping in a few of days. The remainder of the development time is expended doing more extensive modification of other reused UAPs and creation and testing of completely new UAPs. Reuse of tested objects and UAPs from previously released products improves product quality, results in common interfaces and behaviors across all products, saving development time and maintaining focus on "new functionality" versus "reinventing the wheel".

The last rung of functionality sends the Debug Messages at a constant time interval. It passes a TRUE to the RELAY_NOExecute() method of the CR_DEBUG_IDLE instance of the Relay Object class, which then passes the result to the TIMEDRELAY_OnDelayNCExecute() method of the TR_DEBUG_MSG_INTERVAL instance of the Timed Relay Object class. The resulting value is then passed to the TIMEDRELAY_OnDelayExecute() method of the TR_DEBUG_MSG_INTERVAL instance of the Timed Relay Object class, where if it remains TRUE the timer runs until its preset property value expires, at which time a timed closing normally open contact on sheet 2 would initiate the production of any selected cyclic message. Since the timer expired, on the next program scan the normally closed time opening contact of this timer would open, de-energizing and thus resetting the timer. On the next program scan the time opening contact would again be closed, again activating time timer until its preset is achieved.

NOTE: To change the cyclic message production interval the TIMEDRELAY_SetPreset() method could either; 1) be set directly from the network if direct access to for the Timed Relay Object class is enabled, 2) could be contained within a instance of the Parameter Object Class similar to setting the CAN Identifier in sheet one, 3) could be set to constant values selected by logic contacts or any other means appropriate to the application.

NOTE: The operational behavior of the production of messages when utilizing XRAE is intuitively obvious as it is shown in the respective UAP. In this case the peer Debug Message behavior is either; off, cyclically produced, or is polled. It does not "require" a "Consumer Object" directly associated with its message production as it utilizes a value delivered in another Producer/Consumer message pair operating in the polled request/response mode (Command Message). To perform both cyclic and polled behavior using another network, such as DeviceNet, would utilize functionality, data and code space used by two consumers and two producers. All the different message definitions, and where these values came from, would not be visible and would be contained in a vendor/product specific file as DeviceNet utilizes extremely cryptic, invisible and thus seldom used "dynamic assemblies" to define its message contents. Numerous other networks can and have been implemented with XRAE, but where additional code space is generally required to provide and limit the network functionality to behave consistent with the respective specifications.

The ENGINE_Return() method results in a return to the calling UAP_MAIN().