Application Note: Customizing EDB's RTOS Integration Support

This application note covers how to port EDB's RTOS integration DLL to currently unsupported or custom RTOS's. It covers version 1.2 of the EDB's RTOS API.

EDB's RTOS Features

Thread-Specific Breakpoints

The RTOS and in many cases application code itself operates in a shared environment where many tasks can be executing the same code. The ability to set a thread-specific breakpoint allows one to narrow a problem down to the thread causing the fault and avoid many processor stops that would otherwise have occurred.

RTOS Object Browsing

RTOS's operate on the principals of objects. Threads, queues, semaphores, etc. are all objects with control blocks. The data inside these object control blocks can be very useful in understanding the run state of your application. EDB's online help file or printed User's Guide details how EDB presents this data to the user.

RTOS Thread Viewing

The most significant feature EDB has related to RTOS's is the ability to switch the debuggers current view to a specific RTOS thread. In addition this global view change, EDB can also lock a particular window's view to a particular thread.

Theory of Operation

RTOS-specific knowledge is encapsulated in a separate Dynamic Link Library (DLL) called `rtos_api.dll`. This DLL can be modified to support any RTOS. The provided sample DLL is a working example for ATI's Nucleus Plus RTOS. Inside the DLL is all the RTOS-specific logic needed to reach down into the target memory system and examine the RTOS data structures. An RTOS that provide source code to users are the easiest to port to as you can use the symbols within the RTOS itself to determine where in memory to pull significant data from. Binary-based RTOS's (such as pSOS, VRTX, etc) typically provide some sort of known anchor location in memory and all significant RTOS data can then be derived from this anchor.

EDB also provides some callback functions to the DLL to allow the DLL code to perform symbol lookups and memory/register accesses. In addition the RTOS DLL can control thread specific access to support thread views. For breakpoint qualification purposes, EDB requires the DLL to abstract an **id** for each thread object. Typically this **id** can be simply the address in memory of the thread's control block.

Customizing the DLL

As the first step at customizing the RTOS support, we suggest looking at the two structures below, and filling in the details for your RTOS. The structure `OBJECT_HEADER_STRUCT` is the starting point for describing an object class. Note that the sturctures shown here can be found in `rtos_api.h`

```
typedef struct OBJECT_HEADER_STRUCT
{
   int   object_type;      // type of object (OBJECT_THREAD...)
   OBJECT_DISPLAY *od;     // pointer to an array of OBJECT_DISPLAY structs
   void  *object_ptr;      // pointer to first object
   char  *object_name;     // object type name (Thread...)
   BOOL  dirty;            // object class is dirty
```

```
   int   chain_offset;     // within an object, the offset of the next ptr
   int   object_block_size;// object size in 4 byte words.
   char  *symbol;          // symbol to access the first object by
} OBJECT_HEADER;
```

The next structure defines a field within an object. An array of them defines an object class. Each object header contains a pointer to an array of these fields. Together, this data provides the building blocks for EDB's RTOS object browsing window. The structure below details the members of the field display structure (OJBECT_DISPLAY).

```
typedef struct OBJECT_DISPLAY_STRUCT
{
   char  *name;            // field name
   int   field_size;       // numbers of char's in field
   int   offset;           // offset in control block for field
   DISPLAY_ENUM_MODE mode; // how to display this field
   BOOL  button;           // present a button on this field, Generally,
                           // buttons should be turned on for all pointers
   char  *ctype;           // type of the pointer from the RTOS sources
                           // if used, this allows a "add to watch"
                           // menu item to be present in the short-cut menu
} OBJECT_DISPLAY;
```

In `rtos_api.cpp`, you'll see these two structures used to define arrays of object field descriptors and the object header array. Changing the predefined data in these arrays is the first step in customizing to a new RTOS. If you need to add new ways to display data, you can add new types to the `enum DISPLAY_ENUM_MODE` and provide the corresponding display logic in the function `rtos_display_field()`. Another issue to look at is the function `rtos_current_thread()`, which must reach down ito the target memory to retrieve the active thread. This is typically done by looking at a global variable within the RTOS that contains the needed data. You will need to customize this to reference the correct variable or memory location.

The function `rtos_refresh_object_class()` is the workhorse function for gathering object data from the target. You probably will not need to make changes here, but if your RTOS does not link like-kind objects in a chain, then you will need to find another way to find the address of the control blocks.

Thread Viewing

The thread specific viewing features are optional, and if the code for them is left out of your RTOS_API.DLL, they simple get disable in the EDB interface. However, this feature is likely to be the most desirable of all the RTOS related features. The function below is entirely responsible for implementing this feature. It is passed in information about the current access (view, register, etc) and can decide to override the current access with it's own. A typical usage is when a thread's register is asked for that is not the currently executing thread we must instead of doing a register access, access the location in memory where that threads register is stored on the threads stack. Note that this function is new for Ver 1.1 of the interface.

```
RTOS_API_API int rtos_xfer(UINT view, int dir, OBJECTP *objs, BOOL swap_em,
                           char *err )
```

These next two functions are new in version 1.2 of the interface. They are used to signal the architecture in use by EDB (Mips, ARM, etc) and whether or not data swapping is needed. The data swapping is necessary when the endian of the host does not match the endian the target system.

```
RTOS_API_API void rtos_set_architecture( UINT cpu_family_arg )
RTOS_API_API void rtos_set_swap_needed( BOOL swap )
```

Above, we covered most of the changes you are likely to need to make in supporting a new RTOS. However, you will still need to go through each function's description in the code (`rtos_api.c`) to determine if any additional changes are needed.

Building and Using the RTOS_API DLL

EDB's sample RTOS_API DLLwas built using Microsoft's VC++ 6.0. Other PC-based C/C++ compilers might be useable but EPI has not at this time experimented with using any of them. In the directory RTOS_API, you'll find the following files and directories: (Note that we recommend making a separate copy of this directory and performing all customization work within the new directory).

`rtos_api.cpp`	This is workhorse file containing all the RTOS API (application programming interface) functions used by EDB. Note that the sample code is C++ but does not use any classes or advanced C++ features. A C programmer should have no trouble working with this file.
`rtos_api.dsw`	This file is the VC++ specific project (workspace) file.
`rtos_api.h`	This file is shared with EDB. No changes should be made to this file. Note that in EPI's development tree this file lives in higher level inc directory so if you have trouble compiling adjust the project file to contain this file in this directory.
`stdafx.cpp`	This file is provided by VC++ and is used for pre-compiled header purposes. No changes should be done to this file.
`stdafx.h`	This file is provided by VC++ and is used for pre-compiled header purposes. No changes should be done to this file.
`debug\`	This directory is used to hold debug builds of the DLL.
`release\`	This directory is used to hold release builds of EDB.

Once you've successfully built your `rtos_api.dll`, you can copy it to the directory you are storing EDB in. EDB looks for and launches the DLL at EDB startup time. Note that EDB's "about" dialog box displays the copyright/version notice you provided within the DLL. You can use this to verify loading of the correct DLL.

Note that it is assumed that users are familiar with the VC++ IDE environment. If you need help with this Microsoft product, then please consult the documentation provided with VC++.