MCCI Corporation
3520 Krums Corners Road
Ithaca, New York 14850 USA
Phone +1-607-277-1029
Fax +1-607-277-6844
www.mcci.com

DataPump Dynamic Sizing

Engineering Report 950519
Rev. A
Date: 2009/02/20

PROPRIETARY NOTICE AND DISCLAIMER

Document Release History

Rev. A	2009/02/20	Original release

TABLE OF CONTENTS

1 Abstract 1

1.1 Definitions 1

2 Requirements 1

3 Simple-Minded Approach 2

3.1 Header Files 2

3.2 Allocation Measurement Scratch Pad 3

3.3 Begin Allocation Measurement 3

3.4 Recording Memory Allocations 3

3.5 Capturing Allocations for Chip Driver(s) 4

3.6 Capturing allocations for Protocols 5
3.6.1 Redefine USBPUMP_PROTOCOL_CREATE_FN 5
3.6.2 Change `UsbPump_CreateProtocols()` to capture data 6
3.6.3 Adjust Protocol Create Functions to Return Object Pointers 7

4 Implementation 7

4.1 Scratchpad Buffer 8

4.2 UPLATFORM extensions 8

4.3 Provide Method Wrapper Functions and Virtualization Mechanism 10
4.3.1 Implementation of UsbPumpPlatform_InitAllocationScratchPad 10
4.3.2 4.3.2Implementation of UsbPumpAllocationScratchPad_CommitForObject 11

1 Abstract

The MCCI USB DataPump® makes extensive use of dynamically allocated memory. Users need to know how much RAM is required for a given module or configuration of the DataPump. As of V1.96.9, it's hard to know in advance. I sketch a framework that will allow coders to instrument modules to record and display their RAM usages consistently.

1.1 Definitions

Normal MCCI terminology is used in this document.

Module — Functionality within the DataPump typically represented by a singled link library or UsbMakefile.inc.

Reference Documents

If you insert bibliographic references, this section will automatically be populated with a bibliography. This feature is experimental.

2 Requirements

The following general requirements must be satisfied.

1. All work for implementing sizing calculations must be done by central code – the code for a module shall only have to insert trivial (cut-and-paste) annotation code.
2. This shall be a standard feature of the DataPump product.
3. It must be easy to retrofit existing module implementations.
4. It must be easy for DataPump engineers to get a feeling as to whether the changes they've made are actually effective.
5. We need sizing info for both checked and free builds (if they happen to be different). It is acceptable to do this by running the sizing procedure in checked mode and free mode.
6. The sizes must be correct for the target compiler in use. This includes things like alignment constraints.
7. Customers would like to be able to generate sizing information from the USBRC configuration. (This implies protocol-specific size reporting, because the actual size is a function of the base protocol plus options configured by the user.)
8. The customer would like to be able to replace the reporting portion of the implementation with something more to their liking.
9. Ideally, the sizing info would be analytically linked to the code – any parameterized formulae for sizing should be extracted in a provably correct way from the code. If it's not, we have to have a closed-loop process whereby the estimates in the formulae are regularly checked against reality.
10. We have to provide a preliminary result for a customer soon. This customer is presently only using the DataPump device code and a number of protocol modules.
11. Displayed data has to be traced back to modules in a user-understandable form.

3 Simple-Minded Approach

We break the problem down into two parts:

1. gathering the data, and
2. reporting the information.

The most simple-minded approach for gathering the data (for the wired DataPump, excluding host-stack components) is to instrument the code that processes the application and protocol initialization vectors. This has the advantage of allowing RAM allocation to be determined based on the actual entries used on the application initialization vector. Furthermore, application initialization is not recursive, and there is a one-to-one relationship between entries in the application/protocol initialization vector and "modules".

In this approach, we define two functions: "start-measurement" and "end-measurement". The "start-measurement" function (minimally) records heap allocation information in a scratchpad. The "end-measurement" function determines how much memory had been allocated since the snapshot and outputs the results in a standard format.

The tasks are laid out in following sections.

The trickiest part is initialization. The memory tracking system has to be initialized very early in the system's initialization. We don't want to make it part of the object system. To make this relatively easy, we will require the platform implementation to have an IOCTL that returns a pointer to the memory allocation tracking implementation and a context pointer. If the IOCTL is not implemented, the platform initialization code will set the instance pointer to NULL, disabling all tracking.

Because we centralize the tracking mechanism, there's no need to provide a conditional compile to take out the function calls to the tracker.

We will provide an implementation of the tracker that uses the debug print system to output results to the debug log. The messages are formatted in a distinctive form so that they're easy to post-process. A bright script will be provided that can reduce the data in various ways and produce reports.

3.1 Header Files

We will need a new header file containing the definitions for the measurement scratch pad, and for the API functions.

The API wrapper file will be i/usbpump_allocation.h.

This header file will contain the references to the method function wrappers.

3.2 Allocation Measurement Scratch Pad

Implement a "start-measurement" function, which takes a pointer to a stack-based structure, and fills it with info. Call the structure `USBPUMP_MODULE_ALLOCATION_SCRATCHPAD`. This structure is owned by the measurement function and is private, but since it will be allocated by modules that are clients, it won't be formally opaque. The implementation must be such that the expected sequence of calls can be generated by the regular expression "(start-measurement+ end-measurement)*".

3.3 Begin Allocation Measurement

The basic start-measurement function must record information about the device pool, as well as about the platform/system pool, and it needs to be a method of the platform. Looks like its signature should be:

```
VOID UsbPumpPlatform_InitAllocationScratchpad(
      UPLATFORM *pPlatform,
      USBPUMP_MODULE_ALLOCATION_SCRATCHPAD *pScratch
      );
```

For convenience, we may want to have a version that will find the platform and works on any valid object header:

```
VOID UsbPumpObjectHeader_InitAllocationScratchpad(
      USBPUMP_OBJECT_HEADER *pObjectHeader,
      USBPUMP_MODULE_ALLOCATION_SCRATCHPAD *pScratch
      );
```

In either case, this function call records the current amount of memory in use by the DataPump, in a form that allows the `UsbPumpAllocationScratchPad_CommitForObject()` family of methods to determine the amount of memory used since the last call to `UsbPumpObjectHeader_CommitForObject()`. The

These routines must be implemented in such a way that repeated calls to `..._InitAllocationScratchpad()` with the same `pScratch` parameter are innocuous. Only the most recent call shall be effective.

3.4 Recording Memory Allocations

We also need a family of functions to record the result of a measurement. This is logically a method of `USBPUMP_MODULE_ALLOCATION_SCRATCHPAD`. We want to associate every recorded allocation with an object, because objects have names that are readily related to the modules that allocated them.

In this section, we present the abstract API. We discuss implementation later.

To commit a measurement, given an object pointer, use:

```
VOID UsbPumpAllocationScratchPad_CommitForObject(
      USBPUMP_MODULE_ALLOCATION_SCRATCHPAD *pScratchPad,
      USBPUMP_OBJECT *pObject
      );
```

To commit a measurement, given an object header pointer, use:

```
VOID UsbPumpAllocationScratchPad_CommitForObjectHeader(
      USBPUMP_MODULE_ALLOCATION_SCRATCHPAD *pScratchPad,
      USBPUMP_OBJECT_Header *pObjectHeader
      );
```

We considered adding parameters for pool size prediction, in order to simplify closing the loop by reporting errors if the prediction doesn't match reality. However, as we start taking advantage of sub-pools of memory (as we already do in the host stack), this becomes unwieldy. Instead, we'll have to close the loop using a pre-supplied package that produces results that can be cut and pasted into an analysis tool.

After calling this routine with a given `pScratchPad`, it is an error to call without re-initializing the scratch pad.

3.5 Capturing Allocations for Chip Driver(s)

In `uappinit_port.c`, function `UsbPump_GenericApplicationInit_Port()`, we need to record the size for each probed port. Each `UDEVICE` is an object, and so we can treat the `UDEVICEs`/ports the same way that we treat protocol instances – record/report info about memory use in association with the `UDEVICE`. *Remember, we need to track device pool as well as normal pool.*

```
for (; nVec > 0; --nVec, ++pThis)
      {
      UDEVICE *pDevice;
      BYTES sizeof_Udevice;
      UDEVICE_INITFN *pDeviceInitFunction;
      USBPUMP_MODULE_ALLOCATION_SCRATCHPAD allocationScratch;

      /* capture the memory allocation data */
      UsbPumpPlatform_InitAllocationScratchpad(
            pPlatform, &allocationScratch
            );

      if (pThis->UsbPortIndex >= 0 &&
          (UINT) pThis->UsbPortIndex != UsbPortIndex)
            {
            continue;
            }
```

In addition, we'll need to record the memory usage:

```
      /*
      || fill in some stuff that is common, but not likely
      || to be used unless you're using this API anyway.
```

```
*/
pDevice->udev_usbPortIndex = UsbPortIndex;

/* if it worked, call the app init function */
if (pThis->pAppInitFunction &&
    ! (*pThis->pAppInitFunction)(
                pDevice,
                UsbPortIndex,
                pThis,
                pAppInitContext
                ))
      {
      /* XXX need a device deinitialize function --
      || lacking that we leak the memory in this
      || case.  Since we can't free, register the
      || memory use.
      */
      UsbPumpAllocationScratchpad_CommitForObjectHeader(
            &allocationScratch,
            &pDevice->udev_Header
            );
      return NULL;
      }

/* otherwise:  we are OK for this, so we can return the
|| pointer after we record the memory footprint.
*/
UsbPumpAllocationScratchpad_CommitForObjectHeader(
      &allocationScratch,
      &pDevice->udev_Header
      );
return pDevice;
}
```

3.6 Capturing allocations for Protocols

To capture information for each protocol, we must similarly update `common/usbpumpcreateprotos.c`. But here we run into a problem. We don't get a "master" object handle back from the protocol create function. Of course, we could change the protoype of the object create functions, but that's a major API change.

However... returning a `BOOL` is a waste, and doesn't actually reduce complexity in any way. Protocol create was (historically) created before the MCCI object system, but there's no reason for it to work this way now.

So I think we should make the following changes:

3.6.1 Redefine USBPUMP_PROTOCOL_CREATE_FN

In `i/usbprotoinit.h`, change `USBPUMP_PROTOCOL_CREATE_FN` (and accompanying documentation) from:

```
typedef BOOL USBPUMP_PROTOCOL_CREATE_FN(
                UDEVICE *pDevice,
                UINTERFACE *pInterface,
                CONST USBPUMP_PROTOCOL_INIT_NODE *pNode,
                USBPUMP_OBJECT_HEADER *pProtoInitContext
                );

to

typedef USBPUMP_OBJECT_HEADER *USBPUMP_PROTOCOL_CREATE_FN(
                UDEVICE *pDevice,
                UINTERFACE *pInterface,
                CONST USBPUMP_PROTOCOL_INIT_NODE *pNode,
                USBPUMP_OBJECT_HEADER *pProtoInitContext
                );
```

Document that the result should be non-NULL if an object was successfully created and attached, NULL otherwise. The result should be the "governing" object of the created set of objects.

Note that this implies that all protocols must be USBPUMP_OBJECTs.

3.6.2 Change `UsbPump_CreateProtocols()` to capture data

In `common/usbpumpcreateprotos.c`:, change the main loop of `UsbPump_CreateProtocols()` to capture the data, as shown below.

```
      /* test this one and create if possible */
      if (UsbPumpProtocolInitNode_TestInterface(
            pInitNode,
            pProtoInitContext,
            pDevice,
            pIfc))
            {
            USBPUMP_OBJECT_HDR *pNewObjectHdr;
            USBPUMP_MODULE_ALLOCATION_SCRATCHPAD allocationScratch;

            /* prepare to capture the memory allocation data */
            UsbPumpObjectHeader_InitAllocationScratchpad(
                  &pDevice->udev_Header, &allocationScratch
                  );

            if (! pInitNode->pCreateFunction)
                  {
                  TTUSB_PRINTF((pDevice, UDMASK_ERRORS,
                        "?UsbPump_CreateProtocols: "
                        "TestInterface(%p) matched, but "
                        "no pCreateFunction at node %u\n",
                        pIfc,
                        pInitNode - pInitNode_head));
                  }
            else if ((pNewObjectHdr = pInitNode->pCreateFunction(
                        pDevice,
                        pIfc,
```

```
      pInitNode,
      pProtoInitContext
      )) != NULL)
{
TTUSB_DEBUG(UINT numInstances;)

/* count the number of protocols created */
++uNumProtos;

/* now, try to match */
TTUSB_DEBUG(numInstances = )
UsbPump_CreateProtocolsI_AddOtherInterfaces(
      pDevice,
      pInitNode,
      pProtoInitContext,
      pIfc,
      pIfc_tail
      );

TTUSB_PRINTF((pDevice, UDMASK_ANY,
       " UsbPump_CreateProtocols: "
       "collected %u additional interfaces\n",
       numInstances
       ));

/*
|| all interfaces have been colllected for this
|| match.  We might need to create another instance;
|| but that will have another dataplane, so we
|| won't violate dataplane rules.
||
|| Record the memory footprint for this object.
*/
UsbPumpAllocationScratchpad_CommitForObject(
      &allocationScratch,
      &pNewObjectHdr
      );
}
```

3.6.3 Adjust Protocol Create Functions to Return Object Pointers

Finally, each protocol initialization function in the tree must be converted to return a pointer to a `USBPUMP_OBJECT_HEADER`. This is an inherently manual process. The easiest way to identify the changes needed is to change the definition of USBPUMP_PROTOCOL_CREATE_FN as indicated in 3.6.1 above.

4 Implementation

Since this operation is intended to be virtualized, the method functions called directly by clients are simply wrappers for function calls through a table of functions that is provided as a side effect of system configuration.

For convenience, the method functions will be accessed via a pointer through the UPLATFORM object. UML would be convenient at this point, but as a simple summary:

The UPLATFORM cell UPLATFORM::upf_pAllocationTracker, if not NULL, points to a dynamically-allocated structure of type USBPUMP_ALLOCATION_TRACKER that provides the allocation-tracking system.

USBPUMP_ALLOCATION_TRACKER::pContext is opaque context for use by the allocation tracking system.

4.1 Scratchpad Buffer

Each implementation may have a different requirement for storing allocation information in the scratchpad. Rather than recompile, we reserve declare a buffer of sufficient size for almost any purpose. The implementation must then cast this buffer to its actual implementation type.

The virtual type is:

```
__TMS_TYPEDEF_STRUCT(USBPUMP_ALLOCATION_SCRATCHPAD)
      {
      CONST USBPUMP_ALLOCATION_TRACKING_METHOD_TABLE
                                          *pMethods;
      VOID                                *pContext;
      ADDRBITS_PTR_UNION                  info[8];
      };
```

4.2 UPLATFORM extensions

We'll define a new table type which will be used for the dispatch functions for the measurement API. For this purpose, we need function types as well as the dispatch table type.

```
__TMS_TYPEDEF_FUNCTION(
      USBPUMP_ALLOCATION_TRACKING_SYSTEM_INIT_FN,
            VOID *,
                  (
                  CONST USBPUMP_ALLOCATION_TRACKING_CONFIG pConfig,
                  UPLATFORM *pPlatform
                  ));

_TMS_TYPEDEF_FUNCTION(
      USBPUMP_ALLOCATION_SCRATCHPAD_INIT_FN,
            VOID,
                  (
                  USBPUMP_ALLOCATION_SCRATCHPAD *pScratchPad,
                  USBPUMP_ALLOCATION_TRACKING *pTracking
                  ));

__TMS_TYPEDEF_FUNCTION(
      USBPUMP_ALLOCATION_SCRATCHPAD_COMMIT_OBJECT_FN,
            VOID,
                  (
```

```
                    USBPUMP_ALLOCATION_SCRATCHPAD *pScratchPad,
                    USBPUMP_OBJECT *pObject
                    );

__TMS_TYPEDEF_STRUCT(USBPUMP_ALLOCATION_SCRATCHPAD_METHOD_TABLE)
      {
      USBPUMP_ALLOCATION_SCRATCHPAD_INIT_N
                  *pScratchPadInit;
      USBPUMP_ALLOCATION_SCRATCHPAD_COMMIT_OBJECT_FN
                  *pCommitObject;
      };
```

Since the table is in ROM, we also need an instance structure to live in RAM:

```
__TMS_TYPEDEF_STRUCT(USBPUMP_ALLOCATION_TRACKING)
      {
      USBPUMP_OBJECT_HEADER         Header;
      UPLATFORM *pPlatform;
      VOID*pContext;
      CONST USBPUMP_ALLOCATION_SCRATCHPAD_METHOD_TABLE
            *pScratchPadMethods;
      };
```

The name of this object should be defined with the following macro:

```
#define USBPUMP_ALLOCATION_TRACKING_NAME(Name) \
      Name ".allocation-tracking.mcci.com"
```

The tag of this object shall be:

```
#define USBPUMP_ALLOCATION_TRACKING_TAG\
      UHIL_MEMTAG('U', 'T', 'r', 'k')
```

A configuration object is needed in order to pass information through the initialization system to the UPLATFORM initialization routines.

```
__TMS_TYPEDEF_STRUCT(USBPUMP_ALLOCATION_TRACKING_CONFIG);


struct __TMS_STRUCTNAME( USBPUMP_ALLOCATION_TRACKING_CONFIG)
      {
      USBPUMP_ALLOCATION_TRACKING_SYSTEM_INIT_FN*pTrackingSystemInitFn;
      CONST USBPUMP_ALLOCATION_SCRATCHPAD_METHOD_TABLE
            *pScratchPadMethods;
      CONST TEXT *pTrackingObjectName;
      BYTES TrackingObjectSize;
      CONST VOID *pImplementationDefined;
      };
```

Add an entry to the UPLATFORM:

```
USBPUMP_ALLOCATION_TRACKING *upf_pAllocationTracking;
```

Change the expansion (in `uplatform.h`), of `UPLATFORM_INIT_V5()` macro, based on `UPLATFORM_INIT_V4()`. Expand it to take a new parameter:

```
VOID UPLATFORM_INIT_V45(
        UPLATFORM *pSelf,       // structure to init.
        VOID *pContext,         // arbitrary context pointer.
        UEVENTCONTEXT *pEvctx,  // event context
        UPOLLCONTEXT *pPoll,    // "polling context" -- not used.
        CONST UHIL_INTERRUPT_SYSTEM_INTERFACE
                *pIntIfc,       // interrupt system transfer vector
        UPLATFORM_DEBUG_PRINT_CONTROL *pPrintControl OPTIONAL
        //
        // the method functions
        //
        UPLATFORM_MALLOC_FN *pMalloc,
        UPLATFORM_FREE_FN *pFree,
        UPLATFORM_POST_EVENT_FN *pPostEvent,
        UPLATFORM_GET_EVENT_FN *pGetEvent,
        UPLATFORM_CHECK_EVENT_FN *pCheckEvent,
        UPLATFORM_YIELD_FN *pYield OPTIONAL,
        UPLATFORM_IOCTL_FN *pIoctl OPTIONAL,
        UPLATFORM_CLOSE_FN *pPlatformClose OPTIONAL,
        UPLATFORM_DI_FN *pDi,
        UPLATFORM_SETPSW_FN *pSetPsw,
        CONST UTASK_ROOT_CONFIG *pTaskRootConfig OPTIONAL,
        CONST USBPUMP_TIMER_SWITCH *pTimerSwitch OPTIONAL,
        USBPUMP_ALLOCATION_TRACKING *pTracking OPTIONAL
        );
```

In the expansion, add:

```
__pSelf->upf_pAllocationTracking = (pTracking);
```

4.3 Provide Method Wrapper Functions and Virtualization Mechanism

In fact, all the provided functions listed in section 3 are virtual functions (methods) of the formal interface provided at configuration time.

4.3.1 Implementation of UsbPumpPlatform_InitAllocationScratchPad

The implementation of this function shall be as follows:

```
VOID UsbPumpPlatform_InitAllocationScratchpad(
     UPLATFORM *pPlatform,
     USBPUMP_MODULE_ALLOCATION_SCRATCHPAD *pScratchPad
     )
     {
     USBPUMP_ALLOCATION_TRACKING * CONST pTracking =
          pPlatform->upf_pAllocationTracking;

     /*
```

```
        || If no allocation tracking, then pTracking pointer will
        || be NULL, otherwise will point to the management
        || object.
        */
        pScratchPad->pAllocationTracking = pTracking;

        if (pTracking)
              {
              (*pTracking->pScratchPadMethods->pScratchPadInit)(
                    pScratchPad,
                    pTracking
                    );
              }
        }
```

4.3.2 4.3.2Implementation of UsbPumpAllocationScratchPad_CommitForObject

The implementation of this wrapper function shall be as follows:

```
VOID UsbPumpPlatform_CommitForObject(
      USBPUMP_MODULE_ALLOCATION_SCRATCHPAD *pScratchPad,
      USBPUMP_OBJECT *pObject
      )
      {

      USBPUMP_ALLOCATION_TRACKING * CONST pTracking =
            pScratchPad->pAllocationTracking;

      if (pTracking)
            {
            /* record the information */
            (*pTracking->pScratchPadMethods->pCommitObject)(
                  pScratchPad,
                  pObject
                  );

            /* require another initialization after commit */
            pScratchPad->pAllocationTracking = NULL;
            }
      }
```