An Open Source Motion API

Gary Box
Digital Power Engineering
www.digitalpowerengineering
(Formerly Software Defined Power)

Abstract

The advent of Machine to Machine (M2M) networking presents both an opportunity and challenge to the Motor and Drive industry, particularly for older applications that have yet to move to electronic control and higher efficiency motors. To get on the 'Cloud' these cost sensitive applications need to balance accessing the application's information and functionality, retaining tight control of critical real-time machine processes, and avoiding excessive added cost. In this paper, we will present a framework for an open source motion API designed to meet these requirements.

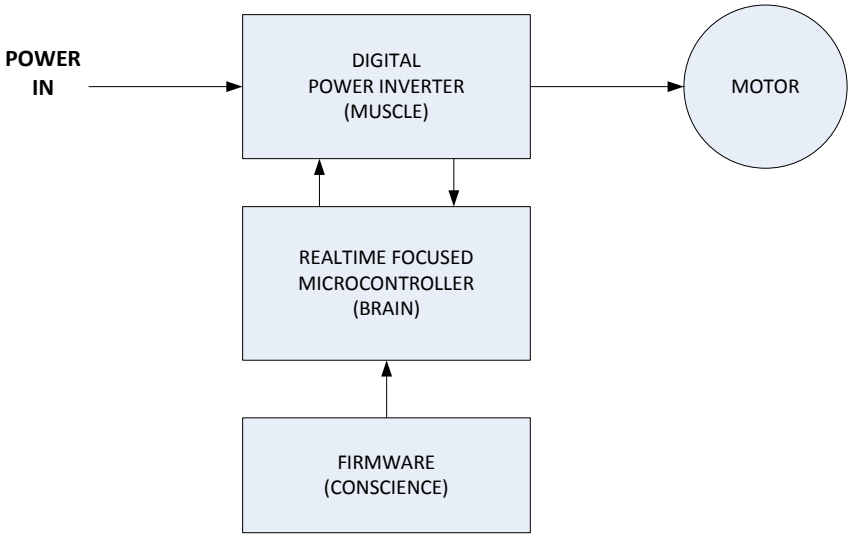
An Open Source Motion API

The trend toward every watt of electricity passing through a semiconductor between generation and consumption is marching on. A big part of that march is the reengineering of applications that currently use a direct connection to the electric source or don't use electric motion at all. At the same time, the "cloud' is getting smarter and more accessible. Machine to Machine (M2M) networking is working its way down the food chain and can eventually come to all motion. Meanwhile, the same M2M technology is enabling data collection from vast arrays of diverse sensors. The resulting M2M infrastructure will open the door to innovative applications that haven't been possible. Most of these will be software applications, all of which haven't been invented yet.

The challenge is how are we to design hardware and software today that would still be useful in unseen applications years down the road? One solution would be to periodically redesign the hardware and software and produce a replacement. But that's not a smart solution, nor a good sell, particularly for the application that has used an induction motor for 50 to 100 years and has a replacement or design cycle of 10 to 20 years.

On the hardware side, the solution may be to embrace the Software Defined Drive architecture, which separates a motor drive into the power handling "muscle", a controlling "brain", and a firmware "conscience" with a well-defined interface between the three. Each is scaled as necessary yet all evolve

separately. They may or may not coexist on a single assembly. Changes to the “muscle” and “brain” involve changing hardware. Changes to the firmware “conscience” involve a major download.



THE SOFTWARE DEFINED DRIVE

The problem with interfacing with the future unseen applications of the M2M infrastructure are that they are just that; unseen. Sitting in our conference rooms today we have no clue what these applications will require of our products, just as the architects of Arpanet in the ‘70’s had no idea of the internet applications we have today. We do, however know they will need access to one or more of the parameters and variables the drive needs to simply do its job. We also know this will require two way accessibility. We also know we will be at the payload end of the very protocols the Arpanet architects devised and that there will be a “host” involved, either locally or in the “cloud”. We also know it’s probably going to be impractical to change anything, including firmware, on the drive in the field.

What is needed is the simplest of frameworks that can provide external direct memory access. If we dig into the history of computing, we find just an inspiration for that in the “peek” and “poke” instructions of early Basic (ca 1970). If one knew the memory address of the data, these instructions were a simple way to read and write to any location in memory. Equivalent programming language constructs are also available in later, compiled languages like “C” and “C++. Carried to the extreme, every data location on every networked computer could be accessed remotely if one knew the network address of the machine and the local memory address of the data (and of course, had access through all the attendant firewalls).

However, this is academic if we can’t relate the parameters and variables to the memory locations of the data. Fortunately, for compiled languages, this is provided in a map file at the time of compilation. The map file is the “Rosetta stone” linking the drive’s memory locations to the firmware variables and

parameters. All that remains is to define a process and protocol to read the map file and manage the "peek" "poke" functions. By designing this properly, we can minimize the programing overhead on both the drive and the host. In our hypothetical future M2M application, the host would be where the application itself would exist and be under control of the application developers. The drive would merely be a network peripheral; much like a network printer is today.

We would want the host to do the "heavy lifting". That would be where the map file is processed into a memory address table of parameters and variables, where the actual M2M application occurs and where software drivers will issue the "peek" and "poke" instructions to the drive, addressing the drive memory directly. The firmware on the drive need only carry out the "peek" and "poke" instructions much like '70's Basic did. Not only does this approach consume very little valuable drive "brain" resources, it implements a time proven protocol that has been in use for over 40 years, and is unlikely to change in the next 40. A drive shipped with this protocol today will be accessible for life.

For convenience, we would add two more requirements to our framework; ASCII compatibility for human readability and a compact, symbolic syntax. ASCII compatibility would allow simple telnet or terminal access to the drive. With the appropriate map file in hand this would allow simple manual access to the parameters and variables as well as a great troubleshooting tool. Equipped with the map file, users, technicians, engineers and even sales and marketing could build unique applications for a computer, tablet or smartphone without ever touching the firmware on the drive. A compact, symbolic syntax providing a single symbol shorthand for commands would speed data transfer and provide easily remembered mnemonics for manual operation. This is analogous to the old AT modem command set, which is still in use today.

Implementing the Open Source Motion API

Let's summarize our requirements for our framework, which in IT parlance would be an Application Programming Interface or API:

1. Allow remote read/ write access to drive memory by memory address
2. Uses the compiler generated map file to find the parameter or variable to memory address relationship.
3. Minimal impact on drive resources.
4. Uses commonly available processor peripheral, such as SCI, to maintain compatibility with all present and future network protocols.
5. ASCII compatibility
6. Compact, symbolic syntax
7. Open source

Description;

The Open Source Motion API consists of a set of target firmware registers, a command set and syntax rules, all implemented in the language of choice on the target drive. It's derived from an API developed in 1998 by Bitscope Designs of Sydney Australia (www.bitscope.com).

Target Firmware Registers;

These registers are implemented as 16 bit unsigned variables. In the C language their type would be unsigned int.

R0	Data register
R1	Address register
R2	Address page

Command Set

All of the commands are implemented as single ASCII characters. This not only creates a very compact protocol for M2M communication, but one that is easy to remember for manual intervention.

The commands are divided into two groups, Data Entry and Register Operation.

Data Entry

[	0x5B	Clear R0
0…9	0x30…0x39	Increment R0 by specified digit and shift left 4 bits
a…f	0x61…0x66	Increment R0 by hex digit and shift left 4 bits
]	0x5D	Shift right four bits

Register Operations

@	0x40	Move R0 -> R1
s	0x73	Store R0 to memory location(R1)
p	0x70	Print contents of memory location (R1) as four ASCII characters LF, CR
P	0x50	Print contents of memory location (R1) and increment R1
S	0x53	Store R0 to memory location (R1) and increment R1
+	0x2B	Increment register R1
-	0x70	Decrement register R1
%	0x45	Move R0 -> R2, to set address page, default is page 0

Syntax and Use

The Open Source Motion API operates in an operand/ operator mode, much like a reverse Polish calculator. The registers serve as temporary storage for either the address of the target memory location or the data to/ from that location. Each single character command executes functions either on or with the data in the registers and stands alone. The commands can be strung together and executed in any sequence.

For example, to enter 0x0bcd into location 0x0800, the host would send;

[0800]@[0bcd]s

The leading '[' command clears the data register R0.

The four ASCII characters, '0800' are then entered. As each is entered, it is converted to a binary number and stored in R0 followed by shifting R0 to the left by 4 bits to make room for the next hex digit.

The trailing ']' command shifts R0 to the right by 4 bits. R0 now contains the hex representation of the entered data, or 0x0800.

The '@' command moves the contents of R0 to the address register R1.

The ASCII sequence '[0bcd]' then loads R0 with the hex number 0x0bcd.

Finally, the command 's' stores the contents of R0, 0x0bcd, into the memory location pointed to by R2:R1, 0x00000800.

Commands can be strung together, such as;

[0003]%[0800]@[0bcd]s[0000]%[09a0]@p

This command sequence does the following;

'[0003]%' sets the address page, R2, to 0x0003.

'[0800]@' sets the address to 0x0800.

'[0bcd]s' stores the number 0xbcd in the addressed memory location, 0x00030800.

'[0000]%[09a0]@p' changes the R2:R1 address to 0x000009a0 and sends its contents to the host.

Expanding the Open Source Motion API

Since the Open Source Motion API is open source, users are free to expand the command set to meet specific needs or provide shorthand macro commands for data collection or machine control. For a C language implementation, this would be a matter of adding additional case statements to the API servicing function. If we stay with the single character protocol, we can implement 256 commands. Some useful possibilities are;

	0x00	Reset and print header
?	0x3F	Print software revision number
>	0x3E	Run from address R2:R3
T	0x54	Start data collection to circular data buffer from address (R2:R1)
D	0x44	Dump circular data buffer
u	0x75	Reset data buffer pointer to zero
v	0x76	Reset address registers R2:R1 to zero

Host Applications using the Open Source Motion API

To use the API, a host would first have to be connected to the drive. This could be a direct wired connection via RS232, a local network connection via RS485, ZIgbee, CAN, WiFI, Power Line Communication (PLC) or even USB or Bluetooth, via appropriate hardware, or even via the internet. In any case, most, if not all, of the work of establishing and maintaining the connection is done by equipment and services upstream of both the host and the drive and invisible to either. Communication between the host and the drive can be considered the same as a direct serial connection.

Since the Open Source Motion API is host operating system and language independent, host applications can be on any type of hardware including smartphones and servers. In fact, a well designed application, written in an operating system and device independent language like Python can be ported to several different host devices. Existing terminal programs on all hosts can use the API manually.

The only rigid requirement for the application is that it must use the map file to establish the relationships between parameters or variables and actual memory locations on the drive. But even this has a wide range of flexibility. An application can read and parse the entire map file, extracting the address data and putting it in an array. Another application may use a database, generated for it by the developer, containing nothing but the array of variable and parameter names and their addresses, or a limited subset. Still another may have the relationship cast in concrete in the application code.

A Word on Security

It is beyond the scope of this paper to explore all the security possibilities of the Open Source Motion API. In fact, security details are best left to individual implementations. For many networks, security is handled upstream of the drive or host. However, the flexibility of the Open Source Motion API allows for a high degree of local security.

Since, in its simplest implementation, the Open Source Motion API provides read/ write access to the entire memory space of the drive, there is the possibility of accessing memory locations that could have catastrophic effects on the application or the drive. The need to know the map file in order to access the drive parameters and variables is the first line of defense. At the host level, the application may have a restricted map file and simply not know where some parameters or variables are or even that they exist.

At the drive level, an implementation can require a pin number or key be saved to a specific memory location before permitting writing and/ or reading to specific memory addresses or the entire memory. Upon receipt of an invalid pin or attempted access to restricted memory, the drive may lock out all communication and require a power on reset or customer service intervention.

Applying the Open Source Motion API

The source code for the Open Source Motion API implemented in ‘C’ and ported to a Texas Instruments TMS320F28xxx microcontroller is included in the appendix. This implementation consumes 360 words of

program memory and 25 words of data memory. Only the **SCIA_Init**() function need be changed to port the code to another processor.

The Open Source Motion API has been in use at Digital Power Engineering since 2007 and has been deployed on about 40000 custom motor drives, test stands and digital power conversion devices since then. We have ported it to TI, Microchip, Atmel and ARM processors and have created host applications for Windows, Linux and Android devices, including smartphones and cloud servers. Data links used include direct TTL serial, RS232, RS485, CAN, Zigbee, wired Ethernet and Bluetooth. We've even hosted applications on Raspberry Pi, Beaglebone Black and Arduino platforms, which leads to some pretty inexpensive and compact systems.

Just as the Open Source Motion API has proved a valuable tool for Digital Power Engineering, it can prove just as valuable in your application also.

Appendix

Open Source Motion API

```
//===========================================================================
//===========================================================================
//
// FILE:  opsys.h
//
// TITLE:          GP Comms kernel header
//
// Version: 22 April 2009 - Release 1.2
//
//Target: TMS320F28xxx
//===========================================================================
//===========================================================================
#ifndef OPSYS_H_
#define OPSYS_H_

EXT long unsigned int R0;
EXT unsigned int R1;
EXT unsigned int R2;
EXT char data_buffer2[20];
EXT char out_data2;
EXT int xmit_pntr;

void SCIA_Init(void);
void service_opsys(char data);
void SCIrx_isr(void);
void SCItx_isr(void);

#endif /*OPSYS_H_*/


//===========================================================================
//===========================================================================
//
// FILE:  opsys.c.c
//
// TITLE:          GP Comms kernel
//
// Version: 22 April 2009 - Release 1.2
//
```

```
//Target: TMS320F28xxx
//===========================================================================
//===========================================================================
#define EXT extern
#include "PeripheralHeaderIncludes.h"  // processor dependent
#include "opsys.h"

void SCIA_Init()   // this function is processor dependent
{
// Note: Assumes Clocks to SCIA are turned on in InitSysCtrl()
// Note: Assumes GPIO pins for SCIA are configured to Primary function

        SciaRegs.SCICCR.all =0x0007;   // 1 stop bit,  No loopback
                // No parity,8 char bits,
                // async mode, idle-line protocol
        SciaRegs.SCICTL1.all =0x0003;  // enable TX, RX, internal SCICLK,
                // Disable RX ERR, SLEEP, TXWAKE
        SciaRegs.SCICTL2.all =0x0003;
        SciaRegs.SCICTL2.bit.TXINTENA =1;
        SciaRegs.SCICTL2.bit.RXBKINTENA =1;
        SciaRegs.SCIHBAUD    =0x0000;

        SciaRegs.SCILBAUD = 0x000a;                     // 0ah = 9600 baud @ LSPCLK = 60 MHz

        SciaRegs.SCICTL1.all =0x0023;           // Relinquish SCI from Reset

        SciaRegs.SCIFFTX.all=0x8040;            // DISable FIFO enhancement
        SciaRegs.SCIPRI.bit.SOFT=0x0;
        SciaRegs.SCIPRI.bit.FREE=0x1;

        PieVectTable.SCIRXINTA = &SCIrx_isr;
        PieVectTable.SCITXINTA = &SCItx_isr;

        PieCtrlRegs.PIEIFR9.all = 0;
        PieCtrlRegs.PIEIER9.bit.INTx1 = 1;
        PieCtrlRegs.PIEIER9.bit.INTx2 = 1;
        IER = IER | M_INT9; // Enable CPU Interrupt 6 TEST
        EINT;       // Enable Global interrupt INTM
        ERTM;       // Enable Global realtime interrupt DBGM

}
```

```
void SCIrx_isr(void)
{
        if (SciaRegs.SCIRXST.bit.RXRDY == 1)      // check if a char has been received
                service_opsys(SciaRegs.SCIRXBUF.all);
        PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;   // Acknowledge interrupt to PIE

  return;
}

void SCItx_isr(void)
{
        ++xmit_pntr;
        if(xmit_pntr<7)
                SciaRegs.SCITXBUF = data_buffer2[0];
        PieCtrlRegs.PIEACK.all = PIEACK_GROUP9;   // Acknowledge interrupt to PIE
  return;
}

void HEXtoASCII(unsigned int data)
{
        char i;

        data_buffer2[3]=data & 0x000f;
        data=data>>4;
        data_buffer2[2]=data & 0x000f;
        data=data>>4;
        data_buffer2[1]=data & 0x000f;
        data=data>>4;
        data_buffer2[0]=data & 0x000f;
        for(i=0;i<4;i++)
        {
                if(data_buffer2[i]<0x0a)
                        data_buffer2[i]=data_buffer2[i]+0x30;
                else
                        data_buffer2[i]=data_buffer2[i]+0x37;
        }
   data_buffer2[4]=0x0d;
   data_buffer2[5]=0x0a;
   data_buffer2[6]=0;
   xmit_pntr=0;
   SciaRegs.SCITXBUF = data_buffer2[0];
```

```
}


//interrupt void service_opsys_isr(void)
void service_opsys(char data)
{
   unsigned int *temp;
   unsigned int btemp;


        if(data>0x2F && data <0x3A)
        {
                R0=R0+(data-0x30);      /*Increment R0 by specified digit and shift left 4 bits*/
                R0=R0<<4;
        }
        if(data>0x60 && data<0x67)
        {
                R0=R0+(data-0x57);      /*Increment R0 by hex digit and shift left 4 bits*/
                R0=R0<<4;
        }

        switch ( data )
        {
                case '[':       /*Clear R0*/
                                R0=0;
                                break;
                case ']':       /*Shift right four bits*/
                                R0=R0>>4;
                                break;
                case '@':               /*Set address register R1*/
                                R1=R0;
                                break;
                case '#':       /*Set source address register R2*/
                                R2=R0;
                                break;
                case 's':       /*Store R0 to register (R2:R1)*/
                                btemp=(R2<<8)+R1;
                                temp = 0;
                                temp = temp + btemp;
                                *temp=R0;
                                break;
                case 'S':       /*Store R0 to register (R2:R1) and increment R1*/
```

```
                btemp=(R2<<8)+R1;
                temp = 0;
                temp = temp + btemp;
                        *temp=R0;
                        ++R1;
                        break;
        case 'p':                       /*Print register (R2:R1)*/
            btemp=(R2<<8)+R1;
            temp = 0;
            temp = temp + btemp;
                        out_data2=*temp;
                        HEXtoASCII(out_data2);
                        break;
        case 'P':           /*Print register (R2:R1)  and increment R1*/
            btemp=(R2<<8)+R1;
            temp = 0;
            temp = temp + btemp;
                        out_data2=*temp;
                        HEXtoASCII(out_data2);
                        ++R1;
                        break;
        case '+':           /*Increment register R1*/
                        ++R1;
                        break;
        case '-':           /*Decrement register (R1)*/
                        --R1;
                        break;
        case ' ':
                        break;
    }
}
```