

CROSSHAIRS []
EMBEDDED

Invaluable [insight] through precise software solutions



Crosshairs [commros]
User Guide



Contents

Contents	2
Introduction.....	4
Getting started.....	5
Overview	5
A detailed look at initialization.....	5
Commros integration considerations	6
Impact of the communication kernel	6
Crosshairs [commros] and application interrupts	6
Running the ServiceRoutine in the main loop	7
Running the ServiceRoutine in a dedicated interrupt	7
The communication interfaces.....	8
The byte interface	8
The packet interface	9
The datalogger.....	9
Datalogger concepts	10
Setting up the datalogger	10
Commros Variables.....	13
Initialization and Usage	13
Variable Types and Access Restrictions.....	14
Internal Flash Programming.....	15
Commros integration	15
void StartProgram()	15
void StopProgram().....	16
void ResetProcessor().....	16
Flash programming	16
External Flash Programming	17
Creating the algorithms and connecting them to Commros	17
Callback functions.....	19
void (*EraseSector) (UINT16 sector).....	19
void (*EraseChip) ().....	19
bool (*ProgramFlash) (UINT32 StartAddress, unsigned char* data_8_bit_bytes, unsigned char size).....	19
unsigned char (*GetStatus) ().....	19
unsigned char (*GetFlashID) (unsigned char* data_8_bit_bytes)	19

void (*ReadFlash) (UINT32 Address, unsigned char* data_8_bit_bytes, unsigned char size) ..	19
Bitfield support	20
Virtual floating point types.....	21
IQ Math.....	21
SQ Math.....	21
Crosshairs [commros] Feature Matrix.....	22
Crosshairs [commros] for C2000	22
Footprint.....	22
Crosshairs [commros] for MSP430	23
Footprint.....	23
Crosshairs [commros] for Cortex M3/M4.....	24
Footprint.....	24
Feature explanations.....	25
Contact	26

Introduction

Commros is a small communication kernel that is running on the target system enabling communication between a host system and the target. The kernel itself is hardware independent and is only "talking" to the hardware through user specified callback functions.

The commros kernel is not an operating system but it can be used together with operating systems like DSP/BIOS from Texas Instruments. Commros does not depend on any interrupts but it has to be called regularly for it to be able to properly handle communication packets.

The commros kernel comes in several versions for different architectures allowing the user to select the most suitable version depending on the hardware it is running on.

The communication kernel can be easily be integrated with most communication solutions like serial communication (RS-232), Ethernet, CAN etc.

Getting started

Including commros into your projects or starting a new project with commros is a straightforward process and this chapter will guide you through the process of including commros into your projects.

A table listing the different version of the commros library and their features is shown in [this chapter](#).

Overview

The easiest way to include commros in your program is to copy the two source files "commros_user.c" and "commros_user.h" into your project folder and include them into your project. From your startup file (the one that contains main()) add an include statement at the top of the file (line 3):

```
#include "commros_user.h"
```

Inside of main after initializing the core clock frequency add the line (line 9):

```
InitCommros ();
```

And finally inside the main loop add a call to commros so that it can process received data (line 12):

```
ServiceRoutine (&commros);
```

The commros variable used as input to the *ServiceRoutine* function is declared in commros_user.c.

Before the program can be compiled and linked, the proper commros library has to be added to your project.

A small program using commros and the TI header files is shown below

```
1 #include "DSP280x_Device.h" // DSP280x Headerfile Include File
2 #include "DSP280x_Examples.h" // DSP280x Examples Include File
3 #include "commros_user.h" // Commros configuration
4
5 void main(void)
6 {
7     InitSysCtrl();
8     DINT;
9     InitCommros(); // Initialize commros
10    for(;;)
11    {
12        ServiceRoutine (&commros); // Call commros
13    }
14 }
```

A detailed look at initialization

The *InitCommros* function is shown below. The commros structure is initialized on line (4) by calling the *InitCore* function with a pointer to commros as the only parameter. This function call initializes commros to a known state and should always be called before any other reference to commros is done.

Next, on line (7), `commros` is given access to the serial port through three callback functions. The first parameter in the function call is a pointer to the `commros` structure whereas the next three parameters are the functions used to access the serial port. The first of these functions, `SCIDataAvailable`, is used to check if there is any pending data in the serial ports receive buffer/FIFO.

The second function, `SCITransmitByte`, is used to write a single byte to the serial port's transmit buffer. The third function, `SCIReceiveByte`, is used to read a single data byte from the serial port's receive buffer. The serial port is initialized on line (11) through a call to `SetupSerialPort`. This function enables the serial port I/O pins, sets up byte sizes, parity, baudrate and enables the serial port for receiving and transmitting data.

Finally, on line (14), a reset message is transmitted to inform anyone listening that the DSP and software is awake and ready to communicate.

```
1 void InitCommros ()
2 {
3     //Initialize commros and reset all internal variables
4     InitCore(&commros);
5
6     //Map serial port functions to commros
7     Commros_SetByteProtocolReceive(&commros, &SCIDataAvailable, &SCIReceiveByte);
8     Commros_SetByteProtocolTransmit(&commros, &SCITransmitByte);
9
10    //Initialize the serial port so it is ready for receiving and transmitting
11    data
12    SetupSerialPort();
13
14    //Inform anyone listening on the serial port that the we are awake
15    Commros_SendResetMessage(&commros, 0);
16 }
```

Commros integration considerations

Impact of the communication kernel

The Crosshairs [`commros`] communication kernel is designed on the principle of being non-intrusive. This means that the tasks internal to `Commros` aren't supposed to have a large impact on the running application. The library handles being starved, but if it is, a possible effect is that you may have communication timeouts when connecting with the `Commros` client applications.

The Crosshairs [`commros`] communication kernel does most of its work through the `ServiceRoutine` function. It is up to the software developer to decide how often the `ServiceRoutine`-function is run in the application. If it is run too seldom there may be communication timeouts when connecting to the device from one of the Crosshairs client applications.

Crosshairs [`commros`] and application interrupts

The Crosshairs [`commros`] communication library uses no interrupts internally, and therefore it won't take any priority. If there are interrupts in use in the application code that triggers, the effect will be that the `ServiceRoutine` relinquishes control to the main application and it will hold off on processing the requests until the interrupt has been serviced. The Crosshairs [`commros`] communication library utilizes CRC-16 checks for all incoming and outgoing messages to ensure

that partial writes resulted from triggered interrupts won't have an effect on the running application. Faulty requests due to communication timeout or partial writes will be dropped.

The regular and full version of commros support reading and writing variables atomically, which prevent interrupts from impacting variable-updates.

The regular and full version of commros support safe writing bitfield-values. There is built-in functionality to safeguard updating critical registry-values by disabling and enabling interrupts before writing the values. The method is explained in the chapter on [bitfield support](#).

Running the ServiceRoutine in the main loop

A common placement for the ServiceRoutine-call is in a loop in the main function of the application. If the main-function handles CPU-intensive tasks or it is in some other way prevented from running at regular interval, calls to the commros ServiceRoutine may be infrequent. The result of this may be that you cannot connect to the device using Crosshairs client applications, or that you experience communication timeouts while connected to the device.

Due consideration must be taken when configuring the main-function if the ServiceRoutine is to be called from there. If the main-function is handling CPU-intensive tasks like updating a LCD-display, running a snippet of code that takes a long time before it completes or if there is an built-in sleep-functionality in the main-loop waiting for a timed interrupt to trigger, the ServiceRoutine may not be called often enough for reliable communication.

If the main-loop is configured to sleep until some interrupt has triggered indicating that the embedded device needs to act, it is recommended to use a dedicated timer. This timer should trigger at regular intervals, thus allowing the main-function to be run and the ServiceRoutine to be called (like a SystemTick timed interrupt).

Generally commros uses few resources when there are few requests buffered in the communication channel, so running the ServiceRoutine often should not have an adverse impact on the application, as long as it is not prioritized over the main functionality of the application.

Running the ServiceRoutine in a dedicated interrupt

Commros can be run in a dedicated interrupt. Communication will be more reliable and you will get a better rate of variable-updates from the device, but there will be a greater impact on the application. It is not recommended to have the ServiceRoutine running in a dedicated interrupt for real-time applications like closed-loop motor control, power conversion and similar.

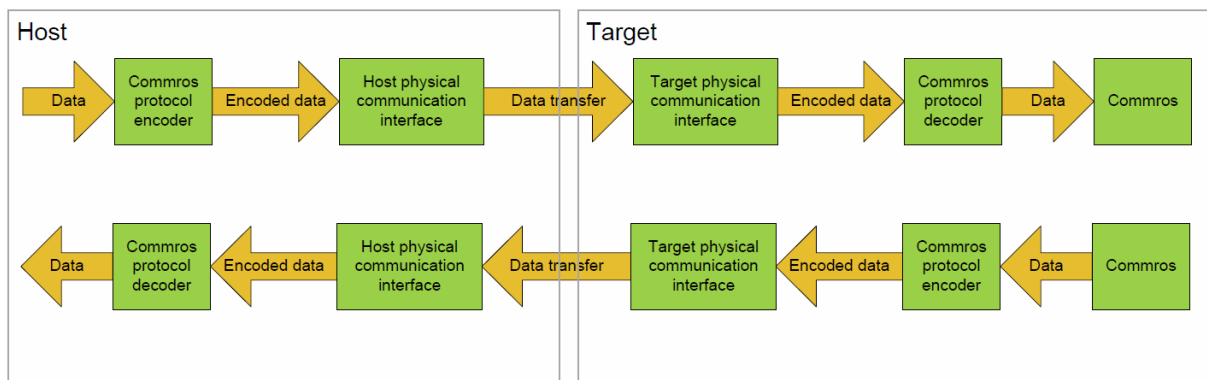
The communication interfaces

The example code so far has only shown how to use commros with a serial port. This is the easiest on the Texas Instruments 28x family since it is a peripheral in all versions of the DSP. Commros is not limited to serial communication but can be interfaced to just about any communication media.

The byte interface

The byte interface is suitable for using byte streams like serial ports for communication. A byte stream like the serial port does not have any inherent error handling beyond that of a single byte and it is therefore necessary to use a communication protocol that can manage the transfer of a whole communication packet with error handling and integrity checks.

Commros has a built in protocol encoder/decoder that is used when data is coming in or going out through the byte interface. The protocol handles the packet synchronization (finding the start of a packet) and uses a CRC16 checksum to verify the packet integrity.



The byte protocol interface is initialized by calling the `Commros_SetByteProtocolReceive/Commros_SetByteProtocolTransmit` functions. These configure the callback functions called by commros to communicate with serial peripherals. The functions are as follows

```
unsigned char SCIDataAvailable()
```

The return value of the function is an unsigned char with a value of 1 if there are any data in the buffer and 0 if it is empty.

The second function is used to transmit data on the serial port

```
void SCITransmitByte(unsigned char data)
```

Where the functions single parameter is the 8-bit byte that should be transmitted. The third and last function is used to receive single data bytes from the serial port

```
unsigned char SCIReceiveByte()
```

The functions return value is the received byte. Commros will always call `SCIDataAvailable` to make sure that there is anything in the FIFO before calling `SCIReceiveByte`.

The packet interface

The packet interface is suitable for communication media where there is an inherent protocol that handles error handling and packet integrity. It is then not necessary to have another protocol layer on top and only the data content of the packet has to be sent to and from commros. If for example Ethernet communication is available, the Ethernet protocol will handle the data transfer and guarantee that the received packet is error free. In these cases the packet interface in commros should be used.

Modified versions of the InitCommros() function that initializes the packet interface in addition to the byte interface are shown below

```
1 void InitCommros()  
2 {  
3 //Initialize commros and reset all internal variables  
4 InitCore(&commros);  
5  
6 //Map serial port functions to commros  
7 SetByteProtocolCallbacks(&commros,&SCIDataAvailable,  
8 &SCITransmitByte,&SCIReceiveByte);  
9  
10 //Map packet communication functions to commros  
11 commros.TransmitPacket = &TransmitPacket;  
12 commros.ReceivePacket = &ReceivePacket;  
13 // Initialize the serial port so it is ready for receiving and  
14 // transmitting data  
15 SetupSerialPort();  
16  
17 //Inform anyone listening on the serial port that the we are awake  
18 SendResetMessage(&commros,0);  
19 }
```

The TransmitPacket callback function must have the following form:

```
void TransmitPacket(unsigned char* pPacket, unsigned char size)
```

where the first parameter is a pointer to an array of unsigned char containing the bytes that should be sent to the host and the number of bytes are given by the second parameter size. The TransmitPacket function is responsible for transmitting the data to the host, either directly or through calls to other functions. The contents of the pPacket array is raw data and it is up to the user to guarantee that the data reaches the host.

The ReceivePacket function must have the following form:

```
unsigned char ReceivePacket(unsigned char** pPacket)
```

where the return value is an unsigned char containing the number of bytes in the packet. The function's parameter is a pointer to a pointer of char and should be set to point to an array containing the packet.

The datalogger

The datalogger is useful when you need to inspect a variable or a group of variables with a high time resolution. The datalogger enables you to select a group of variables and log them to RAM

when a given trigger condition is reached. The number of samples that can be logged is limited by the amount of RAM that is available.

The datalogger is available for the C2000 and Cortex version of Crosshairs [commros] (regular and full version).

Datalogger concepts

This section describes some concepts used when describing the datalogger.

Triggering

Level triggering: Logging will start when the level trigger variable's value is $>$, $<$, $=$, $>=$, $<=$, $!=$ a certain value.

Edge triggering: Logging will start when the edge trigger variable's value is met, either on the rising or falling edge.

Both level triggering and edge triggering can be used at the same time and it is possible to specify two different variables as trigger variables, one for level triggering and one for edge triggering.

Pretriggering

Pretriggering means that the data logger will start logging data as soon as it is configured and enabled but the logged data will start to be overwritten if the end of buffer is reached without any trigger event occurring. When pretriggering is enabled the buffer used to store the logged data will be used as a circular buffer.

When the trigger condition is met, the data logger will log the specified number of samples and at the same time keep up to a specified number of samples before the trigger event occurred. This feature is especially useful when the trigger event is a fault and it is necessary to see what happened a few periods before the fault to understand what led to it.

Probe points

Commros allows you to specify where the data should be sampled. This can be inside an interrupt, inside a loop or function etc. A probe point is used to identify the different places where data can be sampled. Only one probe point can be active at any given time.

Variable buffers

Commros does not allocate any memory for storing information about logged variables and this will have to be done by the user before any logging can take place.

Data buffer

The user has to allocate memory for storing the logged data, this can be done either from the software (recommended) or the buffer can be assigned from the host side. If the buffer is allocated from the software, the compiler/linker will help you make sure that the buffer does not overlap any program or data memory.

Setting up the datalogger

Some basic configuration of the data logger has to be done in software for it to be available on the host side. This initialization should be done at the same time as the rest of commros is configured. The configuration of the data logger can either be done directly from your initialization code called

from main or it can be done inside the *InitCommros* function found in *Commros_user.c*. The necessary configuration steps are described below.

Configuring probe points

We start by configuring the probe points, and in this example code we assume that we need two probe points. One probe point will be placed within the main loop and one at the end of a timer interrupt service routine. We have to declare an array of *LoggerProbeStruct* to store the information about each probe point. The necessary code for this is shown below.

```
1 #define NUM_PROBE_POINTS 2 //The number of probe points
2 #define PROBE_MAIN 0 //Identifier for the first probe
3 #define PROBE_TIMER_INTERRUPT 1 //Identifier for the second probe
4
5 LoggerProbeStruct probes[NUM_PROBE_POINTS]; //Array of probes
```

The probe points should be initialized with a name and a sampling period when *commros* is initialized at startup. It is also necessary to inform *commros* where it can find the probe points using the *Datalogger_AddProbes* function. The necessary code for initializing our two probe points and the call to *Datalogger_AddProbes* is shown below.

```
1 probes[PROBE_MAIN].pName = "Mainloop";
2 probes[PROBE_MAIN].sampleTime = 1.0F;
3 probes[PROBE_TIMER_INTERRUPT].pName = "Timer Interrupt";
4 probes[PROBE_TIMER_INTERRUPT].sampleTime = 0.1;
5 Datalogger_AddProbes(&commros.m_datalogger, probes, NUM_PROBE_POINTS);
```

Line (1) assigns the name "Mainloop" to the probe point with index *PROBE_MAIN* (index 0), this is the name that will be used to identify the probe point on the host side. Line (2) is the sample time of the probe, and since we cannot specify any exact sample time for the main loop we just set it to 1.0 seconds. The sample time is used to scale the time axis in when the logged data is plotted on the host side.

Similarly for the second probe point with the identifier *PROBE_TIMER_INTERRUPT* we assign a name and a sample time for it on lines (3) and (4). The sample timer is equal to the period of our timer interrupt, 0.1 seconds.

Finally on line (5) we inform *commros* about our two probe points using the function *Datalogger_AddProbes*. This function takes the *commros* struct as its first parameters, the address of the probe array as its second parameter and the number of probes as the third parameter.

Allocate memory for variable buffers

Commros does not reserve any memory for storing the address and type information of the variables that should be logged since this will depend on the kind of target it is running on. It is therefore the user's responsibility to first decide the maximum number of variables that he or she want to log concurrently, and reserve the necessary space for them.

The code for reserving space for storing the variable information is shown below. On line (1) we define the maximum number of variables that we want to log concurrently, and at line (3) we define an array of the type *LoggerVarStruct*.

```
1 #define MAX_DATALOGGER_VARIABLES 4 //The number of probe points
2
```

```
3 LoggerVarStruct loggerVars[MAX_DATALOGGER_VARIABLES];
```

Next we have to inform commros where this array is located and the number of elements in it. This is illustrated below where the `Datalogger_AddLoggerVariableBuffers` function is used to do exactly that.

```
1 Datalogger_AddLoggerVariableBuffers(&commros.m_datalogger, loggerVars,  
2                                     MAX_DATALOGGER_VARIABLES);
```

Databuffer

A buffer that is large enough to store the sampled data has to be reserved. This buffer can be placed anywhere in RAM but must be continuous. If the data logger is used inside time critical code it should be placed in internal memory to reduce the time penalty of logging data.

```
1 #define BUFFER_LENGTH 0X800 //The length of the buffer  
2  
3 unsigned char buffer[BUFFER_LENGTH]; //The buffer
```

If the data buffer is declared as shown above, it will be placed in the `.ebss` or `.bss` section depending on your compiler settings. If you want to place it in another section in RAM the `#pragma DATA_SECTION` directive can be used as shown below, where the data buffer is placed in a section called `.buffer`.

```
1 #define BUFFER_LENGTH 0X800 //The length of the buffer  
2  
3 #pragma DATA_SECTION(".buffer", buffer)  
4 unsigned char buffer[BUFFER_LENGTH]; //The buffer
```

After declaring the buffer we have to inform commros where the buffer can be found. This is done using the `Datalogger_SetBuffer` function as shown below.

```
1 Datalogger_SetBuffer(&commros.m_datalogger, buffer, BUFFER_LENGTH);
```

Specifying where the data should be sampled

A call to the datalogger has to be added at the specific locations where the data should be sampled. The probe points are used to identify each of these locations to make them available for selection on the host side.

In the mainloop:

```
Datalogger(&commros.m_datalogger, PROBE_MAIN);
```

At the end of the timer interrupt:

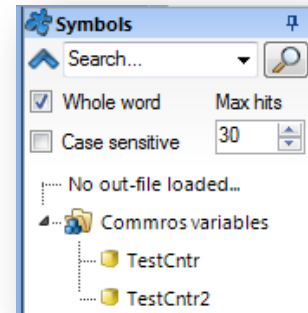
```
Datalogger(&commros.m_datalogger, PROBE_TIMER_INTERRUPT);
```

Commros Variables

Commros Variables provides a mechanism for exposing variables in your application for use in Crosshairs' tools *without* the need for symbolic information or access to the application out/elf file.

This is accomplished by creating a mapping between application variables and Commros Variables that are read and accessed by Crosshairs' tools. Commros Variables are shown in the Symbols view under the "Commros Variables" group, if available and properly configured on the target.

Detailed examples regarding setup and initialization are shown below.



Initialization and Usage

At global scope, define an array of CommrosVariable structs that will hold the exposed Commros Variables. Note that application variables that are exposed through this mechanism must also be globally accessible and have the same lifetime as the application.

```
// The definition of Commros Variables  
CommrosVariable commrosVars[3];
```

Somewhere inside the main portion of the application, before the Commros ServiceRoutine is called, the array of CommrosVariables must be configured and added to Commros.

```
InitCommros();  
  
commrosVars[0].name = "TestCounter";  
commrosVars[0].address = &testcntr;  
commrosVars[0].type = VARIABLE_TYPE_UINT16;  
commrosVars[0].access = ACCESS_READ_WRITE;  
  
commrosVars[1].name = "TestCounter2";  
commrosVars[1].address = &testcntr2;  
commrosVars[1].type = VARIABLE_TYPE_UINT16;  
commrosVars[1].access = ACCESS_READ_WRITE;  
  
commrosVars[2].name = "StaticVar";  
commrosVars[2].address = &staticVar;  
commrosVars[2].type = VARIABLE_TYPE_UINT32;  
commrosVars[2].access = ACCESS_READ_WRITE;  
  
CommrosVariables_AddVariables(&commros.m_commrosVariables, commrosVars, 3);
```

The available variable types are shown in the section Variable Types below.

Variable Types and Access Restrictions

The following types are defined for Commros Variables

```
#define VARIABLE_TYPE_UINT8
#define VARIABLE_TYPE_UINT16
#define VARIABLE_TYPE_UINT32
#define VARIABLE_TYPE_UINT64

#define VARIABLE_TYPE_INT8
#define VARIABLE_TYPE_INT16
#define VARIABLE_TYPE_INT32
#define VARIABLE_TYPE_INT64

#define VARIABLE_TYPE_IQ
#define VARIABLE_TYPE_IQ0
#define VARIABLE_TYPE_IQ1
#define VARIABLE_TYPE_IQ2
#define VARIABLE_TYPE_IQ3
#define VARIABLE_TYPE_IQ4
#define VARIABLE_TYPE_IQ5
#define VARIABLE_TYPE_IQ6
#define VARIABLE_TYPE_IQ7
#define VARIABLE_TYPE_IQ8
#define VARIABLE_TYPE_IQ9

#define VARIABLE_TYPE_IQ10
#define VARIABLE_TYPE_IQ11
#define VARIABLE_TYPE_IQ12
#define VARIABLE_TYPE_IQ13
#define VARIABLE_TYPE_IQ14
#define VARIABLE_TYPE_IQ15
#define VARIABLE_TYPE_IQ16
#define VARIABLE_TYPE_IQ17
#define VARIABLE_TYPE_IQ18
#define VARIABLE_TYPE_IQ19
#define VARIABLE_TYPE_IQ20
#define VARIABLE_TYPE_IQ21
#define VARIABLE_TYPE_IQ22
#define VARIABLE_TYPE_IQ23
#define VARIABLE_TYPE_IQ24
#define VARIABLE_TYPE_IQ25
#define VARIABLE_TYPE_IQ26
#define VARIABLE_TYPE_IQ27
#define VARIABLE_TYPE_IQ28
#define VARIABLE_TYPE_IQ29
#define VARIABLE_TYPE_IQ30
```

The following access restriction flags are defined for Commros Variables

```
#define ACCESS_READ
#define ACCESS_WRITE
#define ACCESS_READ_WRITE
```

Internal Flash Programming

Internal flash programming is available as a Commros feature on C2000-based hardware. In order for Commros to be used as a flash programming interface, the user application must fulfill certain requirements and additional user-defined callback functions must be implemented for Commros.

The following sections will detail the *minimum* requirements for using Commros as a flash programming interface but it is highly recommended to consult the example applications that follow the installation of the Crosshairs' tools for further information. Note also that a keen understanding of how your hardware and application works is required before attempting to flash program the hardware.

Commros integration

As indicated above, Commros must be integrated into the application the same way it would be in any other Commros-enabled application. For details on this process please see Getting started on page 5 and Commros integration considerations on page 6.

It is also very important to have a good understanding of memory layout and section placement of your application. If during the flash programming phase important areas of the application is overwritten (in RAM), the flash programming will fail. For example, downloading the flash algorithms to RAM may overwrite important parts of the Commros library, causing the flash process to fail.

In addition to having Commros integrated, 3 additional user-defined callback functions must be enabled. These are

```
void StartProgram();  
void StopProgram();  
void ResetProcessor();
```

void StartProgram()

The StartProgram function is used to signal that the user-portion of the application may start. This must also be called after Commros has been initialized, typically before the main application loop starts executing. An example implementation is shown below.

```
// Function to start running the user program  
void StartProgram()  
{  
    if(commros.m_bIsProgramRunning == 1)  
        return;  
  
    // Enable interrupts  
    EINT;  
  
    // Enable timer(s) etc...  
  
    commros.m_bIsProgramLoaded = 1;  
    commros.m_bIsProgramRunning = 1;  
}
```

void StopProgram()

The StopProgram function is used to signal that the user-portion of the application must stop executing. This includes all timers, tasks and interrupts (with the exception of communication tasks/interrupts if used by Commros.)

This is called by Commros before flash programming starts. An example implementation is shown below.

```
// Function to stop running the user program
// This function is used when you are flash programming using Commros
void StopProgram()
{
    if(!commros.m_bIsProgramRunning == 1)
        return;

    // disable interrupts
    DINT;

    // disable timer(s) etc...

    commros.m_bIsProgramRunning = 0;
}
```

void ResetProcessor()

The ResetProcessor function is called by Commros after the flash programming has completed. This forces the processor to reset and load the newly flashed application from flash. An example implementation is shown below.

```
void ResetProcessor()
{
    // Writing to watchdog should reset the DSP
    EALLOW;
    SysCtrlRegs.WDCR= 0x0FFF;
    EDIS;
}
```

Flash programming

To program flash using the Commros library, a licensed copy of the Crosshairs Debugger must be available. The Crosshairs Debugger provides a dedicated "Internal Flash Programming" module which is described in detail in the Crosshairs Debugger User Guide.

External Flash Programming

The external flash programming tool can be used to program most types of flashes that can be connected to the DSP. This includes flash devices connected to the external data bus or serial flash devices that are connected to the DSP using communication busses like I²C or SPI.

External flash is only available for selected architectures and kits. Please contact support@crosshairembedded.com for information about support for your kit.

Creating the algorithms and connecting them to Commros

Commros only provides an interface for programming external flash and a set of functions or algorithms has to be connected. The provided functions must be able to handle all communication with the flash (direct memory mapped devices, I2C, SPI etc.) and be able to erase and read/write to it.

The commros external flash interface is set up using the following structures that can be found in the commros header file.

```
typedef struct {  
    UINT32 StartAddress;  
    UINT32 StopAddress;  
} SectorStruct;
```

Variables of type SectorStruct are usually defined as arrays. The sectors for a flash device that has 18 sectors is defined as below:

```
SectorStruct sectors[18] = {..};
```

The actual flash is described using the ExternalFlashInfo:

```
typedef struct
{
    unsigned char* Name;
    UINT32 StartAddress;
    UINT32 StopAddress;
    unsigned char datawidth;
    unsigned char noSectors;
    SectorStruct* Sectors;
    void (*EraseSector)(UINT16 sector);
    void (*EraseChip)();
    bool (*ProgramFlash)(UINT32 StartAddress, unsigned char*
        data_8_bit_bytes, unsigned char size);
    unsigned char (*GetStatus)();
    unsigned char (*GetFlashID)(unsigned char* data_8_bit_bytes);
    void (*ReadFlash)(UINT32 Address, unsigned char* data_8_bit_bytes,
        unsigned char size);
} ExternalFlashInfo;
```

Create a variable of the type ExternalFlashStruct:

```
ExternalFlashInfo XFlash;
```

During configuration and initialization of the software, initialize the structure as described below:

```
(1) XFlash.Name           = xflash_name;
(2) XFlash.StartAddress   = XFLASH_STARTADDRESS;
(3) XFlash.StopAddress    = XFLASH_STARTADDRESS + XFLASH_LENGTH -1;
(4) XFlash.datawidth      = 16;
(5) XFlash.noSectors      = 18;
(6) XFlash.Sectors        = pSectorInfo;
(7) XFlash.EraseSector    = &EraseSector;
(8) XFlash.EraseChip      = &EraseChip;
(9) XFlash.ProgramFlash   = &ProgramFlash;
(10) XFlash.GetStatus      = &GetStatus;
(11) XFlash.GetFlashID    = &GetFlashID;
(12) XFlash.ReadFlash     = &ReadFlash;

(14) commros.FlashCount   = 1;
(15) commros.ExternalFlash = &XFlash;
```

Line (1) makes the pointer to char Name point to a name describing the memory. This is especially useful if there are several flash devices available. The pointer should be set to zero if no name is necessary to identify the device.

Line (2) is used to inform the commros core of the start address of the external flash. This address is not used directly by commros, but is used by the engine to determine if a section of code or data is located in the external flash memory and which functions that should be called to perform flash operations on this memory range.

Line (3) is the last accessible address in the flash. This information is used together with the start address given in line (2).

Line (4) configures the data width of the flash memory. The engine uses this information during the programming process. If the data width is 16 bit, the engine will always try to program an even number of bytes each time the program flash function is called.

Line (5) configures the number of sectors that the flash chip contains. This information can be found from the components data sheet.

Line (6) tells commros where it can find the address range of each sector.

Lines (7) through (12) initialize the flash operation callback functions that are described below. The algorithms have to be implemented by the user since they are vendor dependent.

Finally, the flash information structure is sent to commros on lines (14) and (15).

Callback functions

void (*EraseSector) (UINT16 sector)

This function is used to erase one of the sectors in flash. The sector to be erased is specified by the input variable "sector".

void (*EraseChip) ()

This function is called when a complete erase of the whole flash memory is requested.

bool (*ProgramFlash) (UINT32 StartAddress, unsigned char* data_8_bit_bytes, unsigned char size)

This function is used to program a block of data or code into the flash memory. The start address of the block is given by the 32-bit input variable "StartAddress". The input variable "data_8_bit_bytes" is a pointer to the actual data/code to be written into the flash. The data/code pointed to is stored in an array of 8 bit bytes and this is true for all platforms.

unsigned char (*GetStatus) ()

This function returns the status of an ongoing erase or write operation. The function should return 1 if an erase or write operation is in progress and 0 if it is finished.

unsigned char (*GetFlashID) (unsigned char* data_8_bit_bytes)

This function can be used to read a vendor and device specific code from the flash device.

void (*ReadFlash) (UINT32 Address, unsigned char* data_8_bit_bytes, unsigned char size)

This function reads a block of data/code from the flash device. This function is not needed on a standard flash device connected to the data bus. If a serial flash or other type of flash memory is used this function should be implemented as it is used by commros/engine to verify the written data/code.

Bitfield support

The full version of Commros supports writing bitfield-values. Reading bitfield-values is supported by all version of the Commros library. Commros modifies bits in a bitfield by doing a read-modify-write of the whole data-range of the bitfield.

When writing to bitfield-values it is important to prevent interrupts from triggering that may lead to partial writes. To prevent this from happening, the Commros library has two member function-pointers that can be hooked up to functions that disable interrupts while writing the bitfield-values and restoring the interrupts afterwards.

The member function pointers to disable and restore interrupts are defined in the commros struct/object as:

```
1 void (*RestoreInterrupts)();  
2 void (*DisableInterrupts)();
```

The functions to enable and disable interrupts must return void and have no arguments.

Here is a skeleton implementation:

```
1 void MyDisableInterrupts()  
2 {  
3     // Disable any interrupts that might trigger here  
4 }
```

```
1 void MyRestoreInterrupts()  
2 {  
3     // Restore any interrupts previously disabled here  
4 }
```

After these functions have been defined, their addresses must be assigned to the member function pointers:

```
1 commros.DisableInterrupts = &MyDisableInterurpts();  
2 commros.RestoreInterrupts = &MyRestoreInterurpts();
```

Assigning the function pointers must happen after Commros has been initialized through calling InitCommros.

It is important to define these functions pointers if you want to write to peripheral registries, in order to prevent unstable operation.

Virtual floating point types

IQ Math

The IQMath library available from Texas Instruments is a library that provides a virtual floating point format designed for fixed point devices. It is available for C28x and Cortex devices. Integer types are used to represent floating point types with the ability to change precision by setting a Q-value for a given type indicating the number of bits to use for precision.

The IQMath library is available through the controlSUITE distribution or from this webpage:

<http://www.ti.com/tool/sprc087>

The Crosshairs Client applications will represent the variable values of IQMath variables with their associated Q-value, or use the fallback global q-value for representation for types that haven't got a defined Q-value. When connecting to an embedded device using one of the Crosshairs client applications it is possible to set what the global Q-value should be.

IQMath is defined to use 32 bits on the C28x architecture.

It is possible to change the Q-value for an IQMath variable in run-time from the Crosshairs client applications.

SQ Math

SQ Math is a specialized data-type to handle formatting of 16 bit Q-math values in the Crosshairs client applications. The Q-value associated with the type will be used for the variable representation in the Crosshairs client application. SQMath data-types work in the same fashion as IQMath variables, with a Q-value indicating the precision of the variable.

The SQMath types are defined as follows:

```
49 typedef int16 _sq;  
50 typedef int16 _sq15;  
51 typedef int16 _sq14;  
52 typedef int16 _sq13;  
53 typedef int16 _sq12;  
54 typedef int16 _sq11;  
55 typedef int16 _sq10;  
56 typedef int16 _sq9;  
57 typedef int16 _sq8;  
58 typedef int16 _sq7;  
59 typedef int16 _sq6;  
60 typedef int16 _sq5;  
61 typedef int16 _sq4;  
62 typedef int16 _sq3;  
63 typedef int16 _sq2;  
64 typedef int16 _sq1;  
65 typedef int16 _sq0;
```

The valid Q-value number is between 0 and 15. There are helper conversion-functions to convert to and from IQMath types in the SQMath.h. It is possible to change the Q-value for a SQMath variable in run-time from our Crosshairs client applications.

The header-file also includes macros to conveniently convert to and from IQMath representation, multiplication and division and Q-Value alteration of SQMath variables.

The header-file SQMath.h is available from the start-menu in the Crosshairs [commros]-folder.

Crosshairs [commros] Feature Matrix

Crosshairs [commros] for C2000

	Tiny	Small	Regular	Full
Ping	X	X	X	X
Reset	X	X	X	X
R/W memory (non-atomic)	X	X	X	X
Serial Communication		X	X	X
Packet interface	X	X	X	X
Multi-variable read		X	X	X
Start / Stop		X	X	X
Internal Flash Programming		X	X	X
Commros Variables			X	X
R/W variables (atomic)			X	X
CRC32			X	X
Datalogger*			X	X
Bitfields support				X
External Flash Programming				X

Crosshairs [commros] library is available for the C2000 architecture for Code Composer Studio v3, v4 and v5.

*The regular commros version has a datalogger that supports only level triggering while the full Commros version supports level and edge triggering

Footprint

		RAM (bytes, 8-bit)	FLASH (bytes, 8-bit)
C2000	Full	800	6364
	Regular	780	4424
	Small	632	2304
	Tiny*	300	934

Commros v2.5, TMS320C2000 compiler, v5.2.11

Crosshairs [commros] for MSP430

	Micro	Tiny*	Small
Ping	X	X	X
Reset	X	X	X
R/W memory (non-atomic)	X	X	X
Serial Communication	X		X
Packet interface		X	X
Commros Variables			X
Multi-variable read			X

Crosshairs [commros] is available for the MSP430 architecture for the following tool chains:

- Texas Instruments Code Composer Studio
- IAR

Footprint

		RAM (bytes, 8-bit)	FLASH (bytes, 8-bit)
MSP430	Small	140	1612
	Micro	88	1028
	Tiny*	70	736

Commros v2.5, MSP430 Code Generator Tools v3.3.3

Crosshairs [commros] for Cortex M3/M4

	Tiny*	Small	Regular	Full
Ping	X	X	X	X
Reset	X	X	X	X
R/W memory (non-atomic)	X	X	X	X
Serial Communication		X	X	X
Packet interface	X	X	X	X
Multi-variable read		X	X	X
Start / Stop		X	X	X
Commros Variables			X	X
R/W variables (atomic)			X	X
CRC32			X	X
Datalogger			X	X
Bitfields support				X
External Flash Programming				X

Crosshairs [commros] is available for the TMS470/Cortex architecture for the following tool chains; Texas Instruments Code Composer Studio v4 and v5, IAR and Keil

*The regular commros version has a datalogger that supports only level triggering while the full Commros version supports level and edge triggering

Footprint

		RAM (bytes, 8-bit)	FLASH (bytes, 8-bit)
Cortex M3/4	Full	520	7090
	Regular	512	4834
	Small	372	2626
	Tiny*	192	2070
Commros v2.5, TMS470 Code Generator Tools v4.6.4			

Feature explanations

Ping

The ping command can be used to check if the target is alive/connected, without "doing any harm", as it does not change anything on the target side.

Reset

The host can reset the DSP using this function if the *ResetProcessor* function pointer has been initialized with the address of a function that can reset the DSP (this is normally done using the watchdog).

Read/Write memory (not atomic)

The processor's memory can be read from or written to. The read/write operation(s) are not atomic (with the exception of single byte writes/reads).

Packet interface

Support for packet based communication is available. This can be used for Ethernet, CAN etc.

Read multiple variables at the same time

Add support for reading several variables in one command from the host side. This will increase the maximum refresh rate on the host side.

Start/Stop

Supports user-programs that can be started and stopped from the host.

Internal Flash Programming

Support for flash-programming the device either by serial bootloading or flash-programming of a live embedded system.

Atomic Read/Write of variables

All variables (not compound types) can be read or written to atomically.

CRC32

Support for CRC32. This feature is necessary for programming external flash memory from the host.

Datalogger

The datalogger provides a high resolution graphing interface to Commros.

Bitfields

Add support for writing to bit fields from the host side. Reading bit fields is always supported.

Contact

If you have any questions regarding this document or any of the products and/or services described herein, please do not hesitate to contact us at support@crosshairseembedded.com

All other enquiries including license sales and enterprise solutions please contact us at sales@crosshairseembedded.com.

During business hours we may also be reached at the following number (+47) 7352 6000.

We are committed to total customer satisfaction and a member of our team will be in touch within one working day.

You may also find up-to-date news, company and product information on our website at www.crosshairseembedded.com

